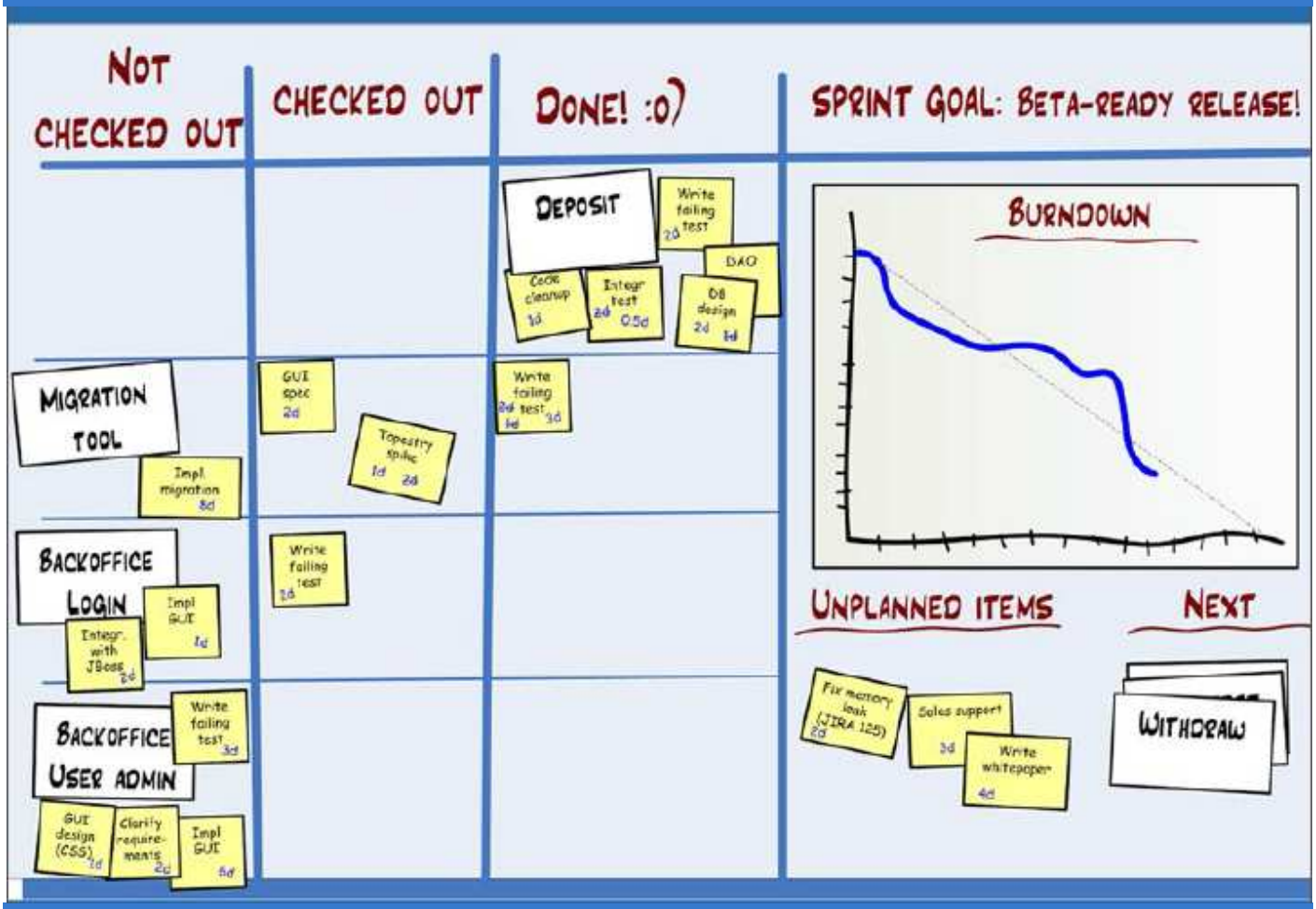
2. Ausgabe – Director's Cut



Scrum und XP Feldbericht

Wie wir das mit Scrum machen

Henrik Kniberg

→ Aus dem Englischen übersetzt von Maria Oelinger

Quelle: http://www.infoq.com/minibooks/scrum-xp-from-the-trenches-2

Inhalt

Dank

Der erste Entwurf dieser Publikation wurde an nur einem Wochenende getippt, aber was für ein intensives Wochenende! 150% Fokusfaktor. :o)

Dank an meine Frau Sophia und die Kids Dave und Jenny, die meine Ungeselligkeit an diesem Wochenende ertragen haben. Und an Sophias Eltern Eva und Jörgen, die vorbeikamen, um zu helfen, sich um die Familie zu kümmern.

Dank auch an meine Kollegen bei Crisp in Stockholm und an die Leute der Scrum Users Yahoo Group, die korrekturgelesen und mir geholfen haben, das Buch zu verbessern.

Und schließlich Dank an alle meine Leserinnen und Leser, die mir einen steten Zufluss an nützlichem Feedback zukommen ließen. Ich bin insbesondere froh zu hören, dass dieses Buch so viele von Euch angefeuert hat, den Startschuss für agile Softwareentwicklung zu geben!

Vorwort von Jeff Sutherland

Teams müssen die Scrum-Grundlagen kennen. Wie erstellt und schätzt man ein Produkt-Backlog? Wie macht man daraus ein Sprint-Backlog? Wie managt man ein Burndown-Diagramm und berechnet die Teamgeschwindigkeit? Henriks Buch ist ein Baukasten grundlegender Methoden, die Teams helfen, weiter zu kommen, wenn sie versuchen, Scrum zu nutzen, um Scrum erfolgreich anzuwenden.

Scrum erfolgreich anzuwenden wird für Teams, die Investitionsbudgets möchten, immer wichtiger. Als agiler Coach für eine Risikokapitalgesellschaft unterstütze ich das Ziel, nur in agile Unternehmen zu investieren, die agile Methoden angemessen einsetzen. Der Seniorpartner der Gesellschaft befragt alle beteiligten Unternehmen, ob sie die Geschwindigkeit ihrer Teams kennen. Sie tun sich derzeit schwer, diese Frage zu beantworten. Künftige Investitionsmöglichkeiten werden erfordern, dass Entwicklungsteams die Geschwindigkeit, in der sie Software produzieren können, kennen und verstehen.

Warum ist das so wichtig? Wenn die Teams diese Geschwindigkeit nicht kennen, kann der Product Owner keinen Produktplan mit plausiblen Releaseterminen erstellen. Ohne verlässliche Releasetermine könnte das Unternehmen scheitern und Investoren verlieren ihr Geld! Diesem Problem stehen große und kleine Unternehmen gegenüber, alte und neue, ausreichend finanzierte und solche ohne Förderung. In einer unlängst geführten Diskussion unter Teilnehmern einer Google-Konferenz in London, auf der sich die Google-Implementierer von Scrum austauschten, fragte ich 135 Leute im Publikum, wie viele Scrum einsetzten, und 30 sagten, dass sie dies täten. Dann fragte ich sie, ob sie nach Nokia-Standards iterativ entwickeln. Iteratives Entwickeln ist grundlegend für das Agile Manifest – früh und häufig funktionierende Software ausliefern. Nach Jahren voller Retrospektiven mit Hunderten von Scrumteams entwickelte Nokia einige Grundanforderungen für iterative Entwicklung: Iterationen müssen feste Zeitfenster[1] haben, die nicht mehr als 6 Wochen lang sein dürfen. Der Code muss am Ende der Iteration durch QA getestet werden und korrekt laufen.

Von den 30 Leuten, die sagten, dass sie Scrum machen, sagte nur die Hälfte, dass sie das erste Prinzip des Agilen Manifests mit Hilfe von Nokia-Standards einhalten. Ich fragte sie dann, ob sie die Nokia-Standards für Scrum einhalten: Ein Scrumteam muss einen Product Owner haben und wissen, wer diese Person ist. Der Product Owner muss ein Produkt-Backlog haben, das (Aufwands-)Schätzungen des Teams enthält. Das Team muss ein Burndown-Diagramm haben und seine Geschwindigkeit kennen. Während eines Sprints darf keiner außerhalb des Teams Einfluss auf das Team nehmen.

Von den 30 Leuten, die Scrum machen, bestanden nur 3 den Nokia-Test für ein Scrumteam. Diese sind die einzigen Teams, die künftig Investitionen von meinen Risikokapitalgesellschaften erhalten werden.

Der Wert von Henriks Buch ist, dass – wenn Du die Methoden befolgst, die er skizziert – Du ein Produkt-Backlog, Schätzungen für das Produkt-Backlog und ein Burndown-Diagramm erhälst und die Geschwindigkeit Deines Teams kennst, zusammen mit vielen anderen unverzichtbaren Methoden für ein hochfunktionales Scrum.

Dr. Jeff Sutherland, Mitbegründer von Scrum

[1] Timeboxes

Vorwort von Mike Cohn

Beide, Scrum und Extreme Programming (XP), fordern von Teams, ganz konkrete Teile an auslieferbarer Arbeit bis zum Ende jeder Iteration fertigzustellen. Diese Iterationen müssen kurz und zeitlich begrenzt sein. Diese Konzentration auf die Auslieferung von funktionierendem Code in einem kurzen Zeitfenster bedeutet, dass Scrum- und XP-Teams keine Zeit für Theorie haben. Sie betreiben kein Zeichnen perfekter UML-Modelle in einem Case-Tool, schreiben keine perfekten Anforderungsdokumente und schreiben keinen Code, der in der Lage ist, allen künftigen Änderungen, die man sich vorstellen kann, Rechnung zu tragen. Statt dessen fokussieren sich Scrum- und XP-Teams darauf, Dinge fertig zu bekommen. Diese Teams akzeptieren, dass sie unterwegs Fehler machen könnten, aber sie wissen auch, dass der beste Weg, diese Fehler zu finden, ist, dass man aufhört, über die Software auf theoretischer Ebene von Analyse und Design nachzudenken, und dass man eintaucht, seine Hände schmutzig macht und anfängt, das Produkt zusammen zu schrauben.

Es ist dieser Fokus auf das Tun statt auf das Theoretisieren, das dieses Buch von anderen unterscheidet. Dass Henrik Kniberg dies versteht, ist von Beginn an offensichtlich. Er liefert keine langatmige Beschreibung, was Scrum ist; dazu verweist er einfach auf einige Websites. Statt dessen springt Henrik direkt ins Thema und beginnt sofort damit zu beschreiben, wie sein Team sein Produkt-Backlog managt und wie es damit arbeitet. Von da aus bewegt er sich durch alle anderen Elemente und Methoden eines gut laufenden agilen Projekts. Kein Theoretisieren. Keine Verweise. Keine Fußnoten[2]. Braucht man nicht. Henriks Buch ist keine philosophische Erläuterung darüber, warum Scrum funktioniert oder warum Sie dies oder jenes probieren sollten. Es ist eine Beschreibung, wie ein gut aufgestelltes agiles Team funktioniert.

Darum ist der Untertitel "Wie wir Scrum machen" so passend. Es mag nicht genauso sein, wie Sie Scrum betreiben, es ist so, wie Henriks Team Scrum macht. Sie könnten sich fragen, warum Sie sich darum kümmern sollten, wie ein anderes Team Scrum macht. Sie sollten sich darum kümmern, weil wir alle lernen können, wie man Scrum besser einsetzt, indem wir Geschichten lauschen, wie es von anderen gemacht wurde, insbesondere von jenen, die es richtig gut machen. Es gibt keine Liste von empfehlenswerten Scrum-Methoden und es wird sie nicht geben, weil Team- und Projektkontext alle anderen Überlegungen ausstechen. Statt Best Practices zu kennen, müssen wir wissen, welche guten Methoden es gibt und in welchem Kontext sie erfolgreich waren. Lies genug Geschichten erfolgreicher Teams und wie sie Dinge taten, und Du wirst vorbereitet sein auf die Hindernisse, die Dir beim Einsatz von Scrum und XP in Deinen Weg geworfen werden.

Henrik stellt eine Menge guter Methoden bereit, zusammen mit dem notwendigen Kontextwissen, um uns zu helfen, besser zu lernen, wie man Scrum und XP im Feld Deiner eigenen Projekte einsetzt.

Mike Cohn, Autor von "Agile Estimating and Planning and User Stories Applied for Agile Software Development"

[2] zum besseren Verständnis gibt es in der dt. Ausgabe Fußnoten

Vorwort: Hey, Scrum hat funktioniert!

Scrum hat funktioniert! Zumindest für uns (d. h. für meinen aktuellen Kunden in Stockholm, dessen Name ich nicht beabsichtige, hier zu nennen). Ich hoffe, es wird auch für Euch funktionieren! Vielleicht wird dieses Buch Euch auf dem Weg dahin helfen. Dies ist das erste Mal, dass ich eine Entwicklungsmethode gesehen habe (sorry, Ken, ein Rahmen), das auf Anhieb mittels eines Buchs funktioniert hat. Plug 'n' play. Alle bei uns sind damit zufrieden – Entwickler, Tester, Manager. Es half uns, aus einer harten Situation herauszukommen und hat uns ermöglicht, Fokus und Schwung trotz ernster Marktturbulenzen und Personalabbau aufrecht zu halten.

Ich sollte nicht verraten, dass ich überrascht war, aber, naja, das war ich. Nachdem ich erstmal ein paar Bücher zu diesem Thema verdaut hatte, erschien mir Scrum gut, aber fast zu gut, um wahr zu sein (und wir alle kennen das Sprichwort "zu gut, um wahr zu sein"…). So war ich berechtigterweise ein bisschen skeptisch. Aber nachdem ich Scrum ein Jahr lang praktiziert habe, bin ich dermaßen beeindruckt (und die meisten Leute in meinen Teams sind das auch), dass ich wahrscheinlich damit weitermachen werde, Scrum standardmäßig bei neuen Projekten einzusetzen, wo immer nicht ein gewichtiger Grund dagegen spricht.

Vorwort – 2. Auflage

Acht Jahre sind vergangen, und dieses Buch ist immer noch sehr beliebt. Wow! Ich hätte den Einfluss nicht vorhersehen können, den dieses kleine Buch haben würde! Ich treffe oft und überall Teams, Manager, Coachs und Trainer, die es als ihren obersten Leitfaden für agile Softwareentwicklung nutzen.

Nur, die Sache ist die, ich habe seit 2007 eine Menge gelernt! Deswegen braucht das Buch wirklich ein Update.

Seit der Veröffentlichung des Buches hatte ich die Möglichkeit, mit vielen Führungskräften zu arbeiten, die agil und mit dem Gedanken an "Lean" vorgehen; einige sind für mich sogar zu persönlichen Mentoren geworden. Besonderen Dank an Jeff Sutherland, Mary und Tom Poppendieck, Jerry Weinberg, Alistair Cockburn, Kent Beck, Ron Jeffries, und Jeff Patton – ich kann mir keine bessere Gruppe von Ratgeberinnen und Ratgebern vorstellen!

Ich hatte auch die Gelegenheit, einer Menge Firmen zu helfen, die Ideen in der Praxis zu implementieren: Firmen in Krisensituationen genauso wie super erfolgreiche Firmen, die noch besser werden wollten. Insgesamt war es eine ziemlich hirndurchpustende Reise!

Als ich die alte Ausgabe nochmal gelesen habe, war ich überrascht, wie vielen Aussagen darin ich immer noch zustimmen kann. Aber es gibt auch einige Seiten, die ich gern rausreißen würde und sagen: "What the *&€# hatte ich mir bei diesem oder jenem gedacht? Mach's doch nicht so! Es gibt einen viel besseren Weg!"

Da das Buch eine Studie aus dem echten Leben ist, kann ich die Geschichte nicht ändern. Was passiert ist, ist passiert. Aber ich kann es kommentieren!

Und das ist es, was die zweite Ausgabe ausmacht – es ist eine kommentierte Version des ursprünglichen Buches. Wie ein "Director's Cut"[3]. Denk es Dir so, dass ich hinter Dir stehe, während Du das Buch liest, die Dinge kommentiere, Dich anfeuere, mit gelegentlichem Schmunzeln und Ächzen.

> Hier sieht man, wie die Kommentare aussehen. Alles andere (außer dieser Einleitung) ist das Originalbuch, unverändert, und die ausgegrauten Kästen sind meine Kommentare und Betrachtungen für die zweite Auflage.

Ich werde außerdem einige Beispiele anderer Firmen einbringen, die meisten davon von Spotify (denn das ist's wo ich in letzter Zeit die meiste Zeit verbracht habe), einige Beispiele aber auch aus anderen Wirkungsstätten.

Hab Spaß!

Henrik, März 2015

[3] Anmerkung der Übersetzerin: Director's Cut ist eine vom Regisseur gestaltete, geschnittene Version eines Films

TEIL
EINS
Intro

Du hast vor, Scrum in Deiner Organisation einzusetzen. Oder vielleicht hast Du Scrum schon ein paar Monate lang im Einsatz. Du hast die Grundlagen, Du hast das Buch gelesen, vielleicht hast Du sogar das Zertifikat als Scrum Master. Glückwunsch!

Aber Du bist irritiert.

Mit Ken Schwabers Worten: Scrum ist keine Methodik, es ist ein Bezugssystem (oder Rahmen). Das bedeutet, dass Scrum Dir nicht genau sagt, was zu tun ist. Verdammt!

Die gute Nachricht ist, dass ich Dir exakt sagen werde, wie ich Scrum einsetze, und zwar bis in alle quälend schmerzhaften Details. Die schlechte Nachricht ist, nun ja, dass dies nur das ist, wie *ich* Scrum nutze. Das bedeutet nicht, dass *Du* es in genau derselben Weise tun solltest. Vielmehr ist es so, dass ich es in einer anderen Weise einsetzen würde, wenn ich in einer anderen Situation wäre. Die Stärke und das Beschwernis bei Scrum ist, dass Du gezwungen bist, es auf Deine spezifische Situation anzupassen.

Mein aktueller Ansatz bei Scrum ist das Ergebnis aus einem Jahr Herumexperimentieren mit Scrum in einem Entwicklungsteam von ungefähr 40 Leuten. Die Firma befand sich in einer schwierigen Situation mit hohem Überstundenaufkommen, ernsten Qualitätsproblemen, ständigem Feuerlöschen, verpassten Deadlines usw. Die Firma hatte sich entschieden, Scrum einzusetzen, hatte dies jedoch nicht vollständig implementiert, das war mein Part. Für die meisten Leute im Entwicklungsteam war "Scrum" zu der Zeit nur ein komisches Buzzword, das in ihren Fluren ab und an als Echo zu hören war, ohne Einfluss auf ihre tägliche Arbeit.

Nachdem ein Jahr vergangen war, hatten wir Scrum über alle Ebenen der Firma hinweg implementiert, verschiedene Teamgrößen (3 – 12 Leute) und verschiedene Sprintlängen (2 – 6 Wochen), verschiedene Arten für Definitionen von "Fertig", verschiedene Formate an Produkt-Backlogs und Sprint-Backlogs (Excel, Jira, Karteikarten), verschiedene Teststrategien, verschiedene Arten für das Durchführen von Demos, verschiedene Arten für die Synchronisation von mehreren Scrumteams etc. ausprobiert. Wir hatten auch mit XP-Methoden rumgespielt, verschiedene Arten für Continuous Builds, Pair-Programming, testgetriebene Entwicklung etc., und wie man dies mit Scrum kombinieren kann. Dies ist ein stetiger Lernprozess, d. h. die Geschichte endet hier nicht. Ich bin davon überzeugt, dass diese Firma immer weiter lernt (wenn sie mit den Sprintretrospektiven weitermachen) und neue Einsichten gewinnt, wie Scrum in ihrem speziellen Kontext am besten eingesetzt werden kann.

Disclaimer

Dieses Buch erhebt nicht den Anspruch, "die richtige Art" von Scrum zu präsentieren! Es zeigt lediglich eine Art, wie man Scrum umsetzen kann, das Ergebnis konstanter Verbesserung über den Zeitraum von einem Jahr. Du könntest sogar entscheiden, dass wir alles falsch gemacht haben. Alles in diesem Buch spiegelt meine persönlichen, subjektiven Meinungen wider und ist keinesfalls ein offizielles Statement seitens Crisp oder seitens meines aktuellen Kunden. Aus diesem Grund habe ich bewusst vermieden, irgendwelche speziellen Produkte oder Leute zu erwähnen.

Warum ich das schrieb

Als ich von Scrum hörte, las ich die relevanten Bücher zu Scrum und Agilität, verschlang Websites und Foren über Scrum, erlangte Ken Schwabers Zertifizierung, befeuerte ihn mit Fragen und verbrachte jede Menge Zeit damit, mit meinen Kollegen zu diskutieren. Eine der wertvollsten Informationsquellen waren jedoch die wahren Geschichten. Die Geschichten von der Front verwandelten die Prinzipien und Methoden in, naja… Wie Man Es Tatsächlich Macht. Sie halfen mir auch dabei, typische Fehler von Scrumneulingen zu identifizieren (und manchmal zu vermeiden).

Also, dies war meine Chance, etwas zurück zu geben. Hier ist meine Geschichte von der Front. Ich hoffe, dieses Buch wird diejenigen von Euch, die in derselben Situation sind, motivieren, ein paar nützliche Rückmeldungen zu geben. Bitte erleuchtet mich!

Und was ist dieses Scrum?

Oh, Entschuldigung. Du bist total neu in Sachen Scrum oder XP? In diesem Fall möchtest Du vielleicht einen Blick auf die folgenden Links werfen:

- http://agilemanifesto.org
- http://www.mountaingoatsoftware.com/scrum
- http://www.xprogramming.com/xpmag/whatisxp.htm

> Nimm Dir die auch den Scrum Guide vor. Er ist jetzt die offizielle Beschreibung zu Scrum, gepflegt von Jeff Sutherland und Ken Schwaber.
> http://www.scrumguides.org

Wenn Du zu ungeduldig bist, das zu tun, fühl Dich eingeladen, einfach weiter zu lesen. Das meiste an Scrum-Jargon wird an entsprechender Stelle erklärt, so dass Du es trotzdem interessant finden könntest.

TEIL **ZWEI**

Wie wir Produkt-Backlogs machen

Das Produkt-Backlog ist das Herz von Scrum. Dies ist das, wo alles beginnt.

> Äh, nein, das Produkt-Backlog ist nicht der Startpunkt. Ein gutes Produkt beginnt mit einem Kundenbedarf und einer Vision, wie dieser zu decken ist. Das Produkt-Backlog ist das Ergebnis der Verfeinerung dieser Vision in Form konkreter Ergebnisse. Die Reise von der Vision zum Backlog kann ziemlich komplex sein, und viele Techniken sind aufgepoppt, um diese Lücke zu füllen. Dinge wie User-Story-Mapping (lies Jeff Pattons Buch, es ist großartig!), Lean UX, Impact-Mapping und anderes. Aber nimm das bloß nicht als Ausrede für ein im Voraus festgelegtes Riesendesign! Lass das Produkt-Backlog iterativ entstehen, wie alles andere.

Das Produkt-Backlog ist im Grunde eine priorisierte Liste von Anforderungen oder (User) Storys oder Features oder Was-auch-immers. Dinge, die der Kunde möchte, beschrieben in der Fachsprache des Kunden.

Wir nennen dies Storys oder manchmal einfach Backlog-Einträge.

Unsere Storys enthalten die folgenden Felder:

- **ID** – ein eindeutiger Identifikator, einfach eine automatisch hochgezählte Zahl. Das dient dazu, die Spur der Storys nicht zu verlieren, wenn wir sie umbenennen.
- **Name** – ein kurzer, beschreibender Name der Story. Zum Beispiel "Sieh Deine Transaktionshistorie".. Aussagekräftig genug, damit Entwickler und Product Owner ungefähr verstehen, worüber sie reden und aussagekräftig genug, und es von anderen Storys zu unterscheiden. Normalerweise 2 bis 10 Wörter.
- **Reihenfolge** – das Wichtigkeitsranking des Product Owners für diese Story. Zum Beispiel 10. Oder 150. Höher = wichtiger.

  Ich tendiere dazu, den Ausdruck "Priorität" zu vermeiden, da die Priorität 1 typischerweise als "höchste" Priorität verstanden wird, was eklig wird, wenn Du später entscheidest, dass etwas anderes sogar noch wichtiger ist. Welche Priorität sollte das dann bekommen?

  Priorität 0? Priorität -1?
- **Schätzung** – die Ausgangsschätzung des Teams, wieviel Arbeit im Vergleich zu anderen Storys erforderlich ist, um diese Story umzusetzen. Die Einheit ist Story Point und normalerweise deckt sich das ungefähr mit "idealen Personentagen".

  Frag das Team, "wenn Ihr die optimale Anzahl an Personen für diese Story nehmt (nicht zu wenig und nicht zu viele, typischerweise zwei), und Euch in einem Raum mit Mengen an Essen einschließt und völlig ungestört arbeitet, nach wie vielen Tagen werdet Ihr mit einer fertigen, vorführbaren, getesteten und releasefähigen Realisierung wieder herauskommen?" Wenn die Antwort "drei Jungs, eingeschlossen in einen Raum, werden ungefähr vier Tage brauchen" ist, dann ist die Ausgangsschätzung 12 Story Points.

  Wichtig ist, nicht die absoluten Schätzungen korrekt hinzubekommen (d. h. dass eine 2-Punkte-Story 2 Tage dauern sollte), wichtig ist, die relativen Schätzungen korrekt hinzubekommen (d. h. dass eine 2-Punkte-Story ungefähr halb so viel Arbeit erfordert wie eine 4-Punkte-Story).
- **Demo-HowTo** – eine Beschreibung auf hohem Niveau davon, wie diese Story beim Sprint-Demo demonstriert werden soll. Dies ist im Wesentlichen eine simple Testspezifikation. "Tu dies, dann tu das, dann sollte dies passieren."

  Wenn Ihr TDD (testgetriebene Entwicklung, Test-Driven Development) praktiziert, kann diese Beschreibung als Pseudocode für Euren Akzeptanz-Test-Code genutzt werden.
- **Kommentar** – jede andere Info, Klarstellungen, Verweise auf andere Informationsquellen etc. Normalerweise sehr kurz.

PRODUKT-BACKLOG (Beispiel)					
ID	**Name**	**Reihenf**	**Schätz**	**Wie geht die Demo**	**Kommentar**
1	Einzahlung	30	5	Einloggen, Einzahlungsseite öffnen, 10,- € einzahlen, auf meine Kontoübersichtsseite wechseln und prüfen, dass die Summe um 10,- € erhöht ist.	Erfordert ein UML-Sequenzdiagramm. Um das Thema Verschlüsselung kümmern wir uns später.
2	Die eigene Transaktionshistorie ansehen	10	8	Einloggen, auf "Transaktionen" klicken. Eine Einzahlung machen. Zurück zu Transaktionen gehen, prüfen, dass die neue Zahlung angezeigt wird.	Paging verwenden, um große DB-Queries zu vermeiden. Design wie Ansicht der Nutzerseite.

Wir haben es mit vielen verschiedenen anderen Spalten probiert, aber schließlich waren die sechs Spalten oben die einzigen, die wir tatsächlich Sprint für Sprint genutzt haben.

> Es gibt zwei Dinge, die ich mittlerweile fast immer anders mache. Zuerst ist da die Spalte "Gewichtung". Statt dieser sortiere ich einfach die Reihenfolge in der Liste. So ziemlich alle Backlog-Management-Tools bieten Drag-und-Drop-Sortierung (sogar Excel und Google Spreadsheets, wenn Du die streng geheime Tastaturkombi dafür kennst). Einfacher und schneller. Zweitens, keine Personentage. Schätzungen erfolgen in Story Points oder T-Shirt-Größen (S/M/L), oder es gibt überhaupt keine Schätzungen. Aber davon später mehr.

Normalerweise machen wir das in einem Exceldokument mit Kollaborationsfunktion (d. h. mehrere Nutzer können simultan editieren). Offiziell gehört das Dokument dem Product Owner, aber wir wollen andere nicht aussperren. Oft möchte ein Entwickler das Dokument öffnen, um etwas zu erläutern oder eine Schätzung zu ändern.

Aus demselben Grund legen wir dieses Dokument nicht ins Version Control Repository; wir legen es statt dessen auf einem freigegebenen Laufwerk ab. Das hat sich als der einfachste Weg herausgestellt, um mehrere simultane Bearbeiter ohne Sperre oder Merge-Konflikte zuzulassen.

Fast alle anderen Artefakte liegen jedoch im Version Control Repository.

> Excel, ja? Wow, das waren Zeiten. Ich würde heute niemals mehr in Erwägung ziehen, Excel fürs Backlogmanagement zu benutzen, es sei denn, als Cloudversion. Das Produkt-Backlog muss als Onlinedokument verfügbar sein, auf das jeder Zugriff hat, das er einfach und simultan bearbeiten kann. Entweder eins der Gazillionen Backlogmanagementtools die es so gibt (Trello und LeanKit und Jira sind beliebt) oder als Google Spreadsheet (sehr praktisch!).

Zusätzliche Story-Felder

Manchmal setzen wir zusätzliche Felder im Produkt-Backlog ein, meist als Erleichterung für den Product Owner, um ihm dabei zu helfen, seine Prioritäten zu ordnen:

- **Track** – eine grobe Kategorisierung der Story, z. B. "Back-Office" oder "Verbesserung". So kann der Product Owner einfach alle "Verbesserung"-Items per Suche filtern, und deren Priorität auf niedrig setzen etc.
- **Component** – Normalerweise als "Checkboxes" im Exceldokument umgesetzt, z. B. "Datenbank, Server, Client". Hierüber kann das Team oder der Product Owner identifizieren, welche technischen Komponenten bei der Implementierung dieser Story beteiligt sind. Das ist nützlich, wenn Ihr mehrere Scrum-Teams habt – z. B. ein Back-Office-Team und ein Client-Team – und es für jedes Team einfacher machen möchtet zu entscheiden, welche Story dran ist.
- **Requestor** – der Product Owner möchte vielleicht nachhalten, welcher Kunde oder Stakeholder das Item ursprünglich als Anforderung eingebracht hat, um diesem eine Rückmeldung zum Fortschritt zu geben.
- **Bug tracking ID** – wenn Ihr ein separates Bug-Tracking-System habt, wie wir mit Jira, ist es nützlich, jede direkte Beziehung zu dokumentieren, die zwischen einer Story und einem oder mehreren gemeldeten Fehlern besteht.

Wie wir es mit dem Produkt-Backlog auf der Geschäftsebene halten

Wenn der Product Owner einen technischen Hintergrund hat, möchte er vielleicht Storys wie "Indexe zur Event-Tabelle hinzufügen" aufnehmen. Warum möchte er das? Das dahinter liegende Ziel ist wahrscheinlich sowas wie "die Suche im Back-Office beschleunigen".

Es könnte sich herausstellen, dass Indexe nicht der Flaschenhals waren, die die Suche langsam gemacht haben. Es könnte etwas total anderes dahinterstecken. Das Team ist normalerweise besser darin herauszufinden, *wie* etwas gelöst wird, deshalb sollte der Product Owner sich auf die Geschäftsziele konzentrieren.

Wenn ich technisch klingende Storys wie diese sehe, stelle ich normalerweise dem Product Owner eine Reihe von "aber *warum*?"-Fragen, bis wir das zugrunde liegende Ziel finden. Dann formulieren wir die Story entsprechend dem zugrunde liegenden Ziel um ("die Suche im Back-Office beschleunigen"). Die ursprüngliche technische Beschreibung endet als Notiz ("Indexe zur Event-Tabelle hinzufügen").

> Dafür gibt es ein altes und gut etabliertes Template: "Als X möchte ich Y, um Z zu erreichen." Zum Beispiel: "Als Käufer möchte ich meinen Warenkorb speichern, damit ich morgen meinen Einkauf fortsetzen kann." Ich bin wirklich überrascht, dass ich 2007 davon nicht gehört hatte! Wäre sehr zweckdienlich gewesen. Ja, es gibt heutzutage ausgefeiltere Templates, die zur Verfügung stehen, aber dieses einfache ist ein guter Ausgangspunkt, besonders für Teams, für die das ganze agile Ding neu ist. Das Template zwingt Dich dazu, die richtige Art von Fragen zu stellen, und verringert das Risiko, sich in technischen Details zu verlieren.

TEIL
DREI
Wie wir die Sprintplanung vorbereiten

OK, der Sprintplanungstag ist immer überraschend schnell schon wieder da. Eine Lektion, die wir immer wieder gelernt haben: Stell' sicher, dass das Produkt-Backlog *vor* dem Sprintplanungstreffen picobello ist.

> Amen dazu! Ich habe viele Sprintplanungstreffen erlebt, die geplatzt sind, weil das Produkt-Backlog unaufgeräumt ist. Kennst Du den Spruch "shit in = shit out"? Genau.

Und was *bedeutet* das? Dass alle Storys perfekt festgelegt sein müssen? Dass alle Schätzungen korrekt sein müssen? Dass alle Prioritäten gesetzt sein müssen? Nein, Nein und Nein! Es bedeutet lediglich:

- Das Produkt-Backlog sollte existieren! (Stell Dir mal vor?)
- Es sollte *ein* Produkt-Backlog und *einen* Product Owner geben (das heißt pro Produkt).
- Alle wichtigen Einträge / Items sollten ihr Wichtigkeitsranking haben, *verschiedene* Wichtigkeitsrankings.
    - Eigentlich ist es OK, wenn weniger wichtige Einträge / Items alle denselben Wert haben, da sie wahrscheinlich sowieso nicht während der Sprintplanungssitzung auf den Tisch kommen.
    - Jede Story, von der der Product Owner glaubt, dass es auch nur eine vage Möglichkeit gibt, dass sie im nächsten Sprint reinkommt, sollte einen eindeutigen Wichtigkeitsrang haben.
    - Das Wichtigkeitsranking wird nur genutzt, um die Einträge nach Wichtigkeit zu sortieren. Also wenn Eintrag A eine Wichtigkeit von 20 und Eintrag B eine Wichtigkeit von 100 hat, dann bedeutet das einfach, dass B wichtiger als A ist. Es bedeutet *nicht*, dass B fünfmal so wichtig ist wie A. Wenn B ein Wichtigkeitsranking von 21 hätte, wäre dies genau dasselbe!
    - Es ist nützlich, Platz zwischen den Nummernreihenfolge zu lassen, falls ein Eintrag C auftaucht, dass wichtiger als A aber weniger wichtig als B ist. Natürlich könnte man ein Wichtigkeitsranking von 20,5 für C nehmen, aber das ist hässlich, also lassen wir stattdessen Platz!
- Der Product Owner sollte jede Story verstehen (normalerweise ist er der Autor, aber manchmal fügen andere Leute Anfragen hinzu, die der Product Owner priorisieren kann). Er muss nicht genau wissen, was implementiert werden muss, aber er sollte verstehen, warum die Story da ist.

> Bäh, sortier einfach die Liste und Du brauchst nicht mit Wichtigkeitsrankings herumzufummeln.

Hinweis: Andere Leute als der Product Owner können Storys zum Produkt-Backlog hinzufügen. Aber sie dürfen nicht das Wichtigkeitsranking setzen – das ist allein die Rolle des Product Owners. Sie können auch keine Schätzungen hinzufügen, dass ist allein das Recht des Teams.

Andere Ansätze, die wir ausprobiert oder geprüft haben:

- Jira (unser Bug-Tracking-System) benutzen, um das Produkt-Backlog zu beherbergen. Die meisten unserer Product Owner finden jedoch, dass das zu viele Klicks erfordert. Excel ist nett und einfach in der Handhabung. Du kannst einfachen Farbcode benutzen, Einträge in eine andere Reihenfolge bringen, neue Spalten ad hoc hinzufügen, Hinweise dazuschreiben, Daten importieren und exportieren etc.

> Dasselbe geht mit Google Spreadsheets. Und es ist in der Cloud. Multiuser, gleichzeitiges Editieren. Ich sag's nur.

- Ein agiles Tool benutzen, das Prozesse unterstützt, wie VersionOne, ScrumWorks, Xplanner etc. Wir sind noch nicht dazu gekommen, irgendwelche davon zu testen, aber werden das wahrscheinlich noch tun.

TEIL VIER

Wie wir die Sprints planen

Das Sprintplanungstreffen ist ein kritisches Treffen, wahrscheinlich das wichtigstes Treffen in Scrum (meiner Meinung nach, die natürlich subjektiv ist). Ein schelcht ausgeführtes Sprintplanungstreffen kann einen ganzen Sprint verderben.

> Wichtig? Ja. Das wichtigste Treffen in Scrum? Nein! Retrospektiven sind viiiiieeel wichtiger! Denn gute Retrospektiven tragen dazu bei, andere Dinge, die nicht laufen, zu reparieren. Sprintplanung tendiert dazu, ziemlich trivial zu sein, so lange andere Dinge ihre Funktion erfüllen (ein gutes Produkt-Backlog, ein engagierter Product Owner und ein engagiertes Team etc.). Also, sprinten ist nicht die einzige Art, agil zu sein – viele Teams setzen stattdessen Kanban ein. Ich habe darüber sogar mal ein ganzes Buch geschrieben: "Kanban and Scrum: Making the Most of Both"[4].
>
> http://www.infoq.com/minibooks/kanban-scrum-minibook

Das Ziel des Sprintplanungstreffens ist, dem Team genug Informationen zu geben, damit es ungestört und in Ruhe einige Wochen lang arbeiten kann, und dem Product Owner genug Vertrauen zu geben, damit er die Teammitglieder dies tun lassen kann.

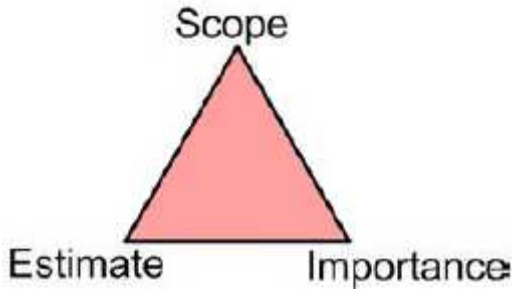
OK, das war haarig. Das konkrete Ergebnis des Sprintplanungstreffens ist:

- Ein Sprint-Ziel
- Eine Liste mit Teammitgliedern (mit dem Maß ihrer Zusage[5], falls es nicht 100 % sind)
- Ein Backlog für den Sprint (= eine Liste von Storys, die zu diesem Sprint gehören)
- Ein festgelegtes Datum für die Sprint-Demo
- Eine festgelegte Zeit und ein festgelegter Ort für die tägliche Scrum-Stehung

Warum der Product Owner teilnehmen muss

Manchmal sträuben sich Product Owner dagegen, Stunden mit dem Team zu verbringen, um den Sprint zu planen. "Leute, ich habe doch schon aufgelistet, was ich will. Ich hab' keine Zeit für Euer Planungstreffen." Das ist ein ziemlich ernstes Problem.

Der Grund dafür, warum das ganze Team *und* der Product Owner am Sprintplanungstreffen teilnehmen müssen, ist, dass jede Story drei Variablen enthält, die in höchstem Maß voneinander abhängen.



Umfang, Bedeutung, Schätzung

Scope (Umfang) und Importance (Bedeutung) werden vom Product Owner festgelegt. Estimate (Schätzung) wird vom Team festgelegt. Während eines Sprintplanungstreffens werden diese drei Variablen in einem persönlichen Dialog zwischen Team und Product Owner abgestimmt. Normalerweise beginnt der Product Owner das Treffen dadurch, dass er das Ziel für den Sprint und die wichtigsten Storys zusammenfasst. Als nächstes geht das Team alles durch und schätzt für jede Story die Zeit, angefangen mit der wichtigsten. Während sie dies tun, kommen wichtige Fragen zum Umfang auf – "Gehört das Durchgehen und Aufheben jeder anstehenden Transaktion für diesen User zur 'User löschen'-Story dazu?" In manchen Fällen werden die Antworten für das Team überraschend ausfallen und es dazu veranlassen, die Schätzungen zu ändern. In einigen Fällen wird die Zeitschätzung für eine Story nicht das sein, was der Product Owner erwartet hat. Das könnte ihn dazu veranlassen, die Bedeutung der Story anzupassen. Oder den Umfang der Story zu ändern, was wiederum vom Team eine erneute Schätzung veranlasst usw. usf.

[4] dt.: " Kanban und Scrum: Optimaler Einsatz beider Methoden "
http://www.infoq.com/resource/news/2010/01/kanban-scrum-minibook/en/resources/KanbanAndScrum-German.pdf
[5] Zusage oder "Commitment"

Diese Art von direkter Zusammenarbeit ist für Scrum und für alle agilen Softwareentwicklungsmethoden grundlegend.

Was, wenn der Product Owner immer noch darauf besteht, dass er keine Zeit hat, am Sprintplanungstreffen teilzunehmen? Ich versuche normalerweise eine der folgenden Strategien, in dieser Reihenfolge:

- Ich versuche, dem Product Owner zu helfen zu verstehen, warum seine direkte Beteiligung entscheidend ist und hoffe, dass er seine Meinung ändert.
- Ich versuche jemanden im Team als Freiwilligen zu bekommen, der während des Treffens als Product Owner fungiert. Ich sage dem Product Owner, "da Du am Treffen nicht teilnehmen kannst, wird Dich Jeff hier als Stellvertreter repräsentieren. Er wird die volle Entscheidungsbefugnis dafür haben, Prioritäten und Umfang der Storys während des Treffens in Deinem Namen zu ändern. Ich schlage vor, Du stimmst Dich mit ihm so gut wie möglich vor dem Treffen ab. Wenn Du nicht möchtest, dass Jeff Dich vertritt, schlage bitte jemand anderen vor, so lange diese Person uns während des gesamten Treffens zur Verfügung steht."
- Ich versuche, das Managementteam zu überzeugen, einen neuen Product Owner zu bestimmen.
- Ich verschiebe die Einführung des Produkts aus diesem Sprint, bis der Product Owner Zeit findet, um am Treffen teilzunehmen. In der Zwischenzeit weigere ich mich, irgendwelche Produktbereitstellungen zuzusagen. Ich lasse das Team jeden Tag tun, was es für das wichtigste an diesem Tag hält.

> Tolle Sachen in diesem Abschnitt! <klopfe mir selbst auf die Schulter>
>
> Nur eins – ich empfehle dringend, das Backlogverfeinern (Schätzung, Storys aufteilen etc.) in einem separaten Treffen zu machen, so dass die Sprintplanung mehr fokussiert erfolgen kann. Product-Owner-Beteiligung ist jedoch auch dann entscheidend, in beiden Treffen.

Warum Qualität nicht verhandelbar ist

In dem Dreieck oben habe ich bewusst eine vierte Variable vermieden: Qualität.

Ich versuche, zwischen interner und externer Qualität zu unterscheiden:

- Externe Qualität ist das, was vom Nutzer des Systems wahrgenommen wird. Ein langsames und nicht-intuitives User-Interface ist ein Beispiel für schlechte externe Qualität.
- Interne Qualität bezieht sich auf Punkte, die normalerweise für den Nutzer nicht sichtbar sind, die aber eine tiefgehende Wirkung auf die Wartbarkeit des Systems haben. Dinge wie Konsistenz des Systemdesigns, Testabdeckung, Lesbarkeit von Code, Refaktorierbarkeit etc.

Im allgemeinen kann ein System mit hoher interner Qualität immer noch eine schlechte externe Qualität haben. Aber ein System mit schlechter interner Qualität wird selten eine hohe externe Qualität haben. Es ist schwer, auf einem verrotteten Fundament etwas Feines zu bauen.

Ich sehe externe Qualität als Teil des Umfangs. In einigen Fällen könnte es aus Geschäftssicht absolut sinnvoll sein, eine Version des Systems auszuliefern, die ein plumpes und langsames User-Interface hat, um dann später eine aufgeräumte Version auszuliefern. Ich belasse diesen Zielkonflikt beim Product Owner, da er für die Festlegung des Umfangs verantwortlich ist.

Interne Qualität jedoch steht nicht zur Diskussion. Es liegt in der Verantwortung des Teams, die Qualität des Systems unter allen Umständen zu pflegen und dies ist schlicht nicht verhandelbar. Niemals.

(Naja, OK, fast niemals.)

Also, wie stellen wir bei Fragen die Unterscheidung zwischen interner Qualität und externer Qualität her?

Lass uns mal annehmen, der Product Owner sagt: "OK Jungs, ich respektiere Eure Zeitschätzung von sechzig Story-Punkten, aber ich bin sicher, Ihr könnt in der halben Zeit irgendwas an schneller Lösung finden, wenn Ihr Euer Hirn drauf ansetzt."

Aha! Er versucht gerade, interne Qualität als Variable zu behandeln. Woher ich das weiß? Weil er möchte, dass wir die Schätzung der Story heruntersetzen, ohne dass er "den Preis zahlt", den Umfang zu reduzieren. Der Ausdruck "schnelle Lösung" sollte in Deinem Kopf einen Alarm auslösen…

Und warum erlauben wir das nicht?

Meiner Erfahrung nach ist das Opfern interner Qualität fast immer eine entsetzliche, entsetzliche Idee. Die gesparte Zeit wird von den kurz- und langfristigen Kosten bei weitem erschlagen. Erlaubt man einmal, dass die Codebasis beginnt zu verkommen, ist es sehr schwer, später die Qualität wieder hinein zu bekommen.

Statt dessen versuche ich die Diskussion in Richtung Umfang zu lenken. "Da es wichtig für Dich ist, dieses Feature früh auszuliefern: Können wir den Umfang reduzieren, so dass es schneller zu implementieren ist? Vielleicht können wir die Fehlerbehandlung vereinfachen und 'fortgeschrittene Fehlerbehandlung' als eigene Story aufnehmen, die wir für die Zukunft vorsehen? Oder können wir die Priorität anderer Storys mindern, so dass wir uns auf diese Story konzentrieren können?"

Sobald der Product Owner gelernt hat, dass interne Qualität nicht verhandelbar ist, wird er normalerweise gut darin, statt dessen die anderen Variablen zu beeinflussen.

> Im Prinzip, ja. Aber in der Praxis tendiere ich heute dazu, pragmatischer vorzugehen. Manchmal ist es aus Geschäftssicht absolut sinnvoll, Qualität für eine kurze Zeit zu opfern – zum Beispiel, weil wir diese superwichtige Messe nach dem nächsten Sprint bestücken müssen oder weil wir einfach einen Prototypen brauchen, um eine Hypothese zum Nutzerverhalten zu überprüfen. Aber in diesen Fällen muss der Product Owner klar machen, warum wir diese Dinge tun, und er muss zusagen, dass das Team die technischen Schulden in naher Zukunft tilgen kann (manchmal wird das Team eine "Aufräum"-Story ins Backlog aufnehmen, als Erinnerung). Hohe interne Qualität sollte die Norm sein und Ausnahmen sollten als außerordentlich behandelt werden.

Immer und immer wieder Sprintplanungstreffen

Das Schwierigste beim Sprintplanungstreffen ist folgendes:

1) Leute denken nicht, dass sie so viel Zeit brauchen werden…
2) ... aber sie brauchen sie!

> Nein, brauchen sie nicht! Nicht, wenn Du das Backlog in einem separaten Treffen weiterentwickelst. Viele Teams, die ich gesehen habe, hielten eine Stunde pro Woche ein Treffen zum Backlog-Weiterentwickeln ab, so dass man sich auf die Sprintplanung konzentrieren kann während der, naja, Sprintplanung! Das gibt außerdem dem Product Owner mehr Gelegenheit, das Produkt-Backlog *vor* dem Sprintplanungtreffen zu aktualisieren, was dies wiederum verkürzt. Merke: ein Sprintplanungstreffen sollte normalerweise nicht mehr als eine Stunde pro Sprintlängenwochen dauern (deutlich weniger bei erfahrenen Teams), also drei Stunden oder weniger für einen Drei-Wochen-Sprint.

Alles in Scrum hat feste Zeitfenster. Ich liebe diese einfache, konsequente Regel. Wir versuchen uns daran zu halten.

Also, was tun wir, wenn ein Sprintplanungstreffen mit einer festgelegten Dauer gegen Ende noch keine Anzeichen eines Sprintziels oder Sprint-Backlogs zeigt? Brechen wir einfach vorzeitig ab? Oder verlängern wir es um eine Stunde? Oder beenden wir das Treffen und setzen es am nächsten Tag fort?

Das passiert immer wieder, besonders bei neuen Teams. Also, was tust Du? Ich weiß es nicht. Aber was tun wir? Oh, ähm, naja, normalerweise breche ich das Treffen brutal ab. Beende es. Lasse den Sprint leiden. Genauer, ich sage dem Team und dem Product Owner: "So, dieses Treffen endet in 10 Minuten. Wir haben nicht wirklich viel in Sachen Sprintplan. Begnügen wir uns damit, was wir haben, oder sollten wir noch ein Vier-Stunden-Sprintplanungstreffen morgen früh um 8 Uhr ansetzen?" Du kannst Dir vorstellen, was sie antworten werden. :o)

Ich habe versucht zuzulassen, dass sich das Treffen in die Länge zieht. Das bringt in der Regel nichts, da die Leute erschöpft sind. Wenn sie keinen anständigen Sprintplan in den zwei bis acht Stunden (oder wie groß auch immer Euer Zeitfenster ist) hervorgebracht haben, werden sie das wahrscheinlich auch in einer weiteren Stunde nicht zustande bringen. Die nächste Option ist ziemlich OK: ein neues Treffen für den nächsten Tag anzusetzen. Außer dass die Leute normalerweise ungeduldig sind und mit dem nächsten Sprint anfangen wollen und nicht noch einen Haufen Stunden mit Planung verbringen möchten.

Deshalb breche ich es ab. Und ja, der Sprint leidet. Das Gute daran ist jedoch, dass das Team eine sehr wertvolle Lektion gelernt hat und beim nächsten Sprintplanungstreffen viel effizienter sein wird. Außerdem werden die Leute

weniger widerstreben, wenn Du eine Dauer für das Treffen veranschlagst, von der sie beim letzten Mal dachten, es wäre zu lang.

Lerne, Deine festgesetzten Zeitfenster einzuhalten, lerne, eine realistische Dauer für Zeitfenster festzulegen. Das betrifft sowohl die Dauer für Treffen als auch Sprintlängen.

> Ich traf mal ein Team, das sagte: "Wir haben Scrum ausprobiert und hassten es, wir werden es nicht noch einmal einsetzen!" Ich fragte, warum, und sie sagten: "Zu viel Zeit in Sitzungen! Wir bekamen nichts fertig." Ich fragte, welches Treffen die meiste Zeit in Anspruch genommen hatte, und sie sagten, die Sprintplanung. Ich fragte, wie lang das gedauert hatte, und sie sagten: "Zwei oder drei Tage!" Zwei oder drei TAGE der Planung für jeden Sprint?! Kein Wunder, dass sie es hassten! Sie haben die Time-Boxing-Regel missachtet: entscheide vorher, wieviel Zeit Du gewillt bist zu investieren, und dann halte Dich daran! Scrum ist wie jedes andere Werkzeug – Du kannst einen Hammer so oder so benutzen, um etwas zu bauen oder um Dir auf den Daumen zu hauen. So oder so, gib nicht dem Werkzeug die Schuld.

Agenda für Sprintplanungstreffen

Eine Art vorläufigen Ablauf für das Sprintplanungstreffen zu haben, reduziert das Risiko, das Zeitfenster zu sprengen.

Hier ein Beispiel eines typischen Ablaufs bei uns.

Sprintplanungstreffen: 13:00-17:00 (10 Minuten Pause pro Stunde)

- **13:00-13:30** – Der Product Owner geht das Sprintziel durch und fasst das Produkt-Backlog zusammen. Ort, Datum und Uhrzeit für das Demo werden festgesetzt.
- **13:30-15:00** – Das Team schätzt die Zeit und teilt bei Bedarf die Einträge auf. Einträge werden geklärt: "How To Demo" wird für alle Einträge mit hoher Wichtigkeit ausgefüllt.
- **15:00-16:00** – Das Team wählt Storys aus, die in den Sprint kommen. Zum Gegenchecken berechne die Umsetzungsgeschwindigkeit.
- **16:00-17:00** – Wähle Zeit und Ort für die Daily Scrums, tägliche Stehungen (falls abweichend vom letzten Sprint). Weitere Aufteilung der Storys in Aufgaben[6].

> Das Stück zwischen 13:30 und 15:00 ist die Weiterentwicklung des Produkt-Backlogs (ich nannte das mal "Backlog grooming", lernte aber, dass grooming in einigen Kulturen die Bedeutung von Schlimmen Dingen hat). Nimm das raus und in ein eigenes Treffen rein und, ta-daa, Du erhälst kürzere und possierlichere Sprintplanungstreffen. OK, einige kleinere Anpassungen mögen notwendig sein, aber das meiste an Backlog-Weiterentwicklung sollte *vor* der Sprintplanung erledigt sein.

Der Ablauf sollte auf keinen Fall streng erzwungen werden. Der Scrum Master kann, während das Treffen fortschreitet, die Unterteilung in weitere, kürzere oder längere (Teil-)Zeitfenster nach Bedarf anpassen.

Sprintlänge festlegen

Eines der Ergebnisse des Sprintplanungstreffens ist das Festlegen eines Sprint-Demo-Termins. Das bedeutet, Du musst Dich für eine Sprintlänge entscheiden.

Also, was ist eine gute Sprintlänge?

Naja, kurze Sprints sind gut. Sie erlauben der Firma, "agil" zu sein, d. h. oft die Richtung zu wechseln. Kurze Sprints = kurzer Feedbackzyklus = häufigeres Ausliefern = häufigeres Kundenfeedback = weniger Zeit Herumrennen in die falsche Richtung = schnelleres Lernen und Verbessern etc.

Andererseits, auch lange Sprints sind gut. Das Team bekommt mehr Zeit, Schwungkraft aufzubauen, sie bekommen mehr Raum, um nach Problemen wieder zu Kräften zu kommen und trotzdem das Sprintziel zu erreichen, Du hast weniger Overhead, was Sprintplanungstreffen, Demos etc. angeht.

[6] Tasks

Gewöhnlich mögen Product Owner kurze Sprints, und Entwickler mögen lange Sprints. Also ist die Sprintlänge ein Kompromiss. Wir haben viel damit experimentiert und kamen zu unserer bevorzugten Länge: drei Wochen. Die meisten unserer Teams (aber nicht alle) machen Drei-Wochen-Sprints. Kurz genug, um uns ausreichend Unternehmensagilität zu geben, lang genug für das Team, um Flow zu erreichen und sich von Problemen zu erholen, die während des Sprints auftreten.

Eine Sache, die wir erkannt haben, ist: *macht* zunächst mit Sprintlängen Experimente. Verschwendet nicht zuviel Zeit durch *Analysieren*, wählt einfach eine annehmbare Länge und probiert einen oder zwei Sprints lang, danach ändert Ihr die Länge.

Sobald Ihr Euch jedoch entschieden habt, welche Länge Euch am besten gefällt, *bleibt* über einen längeren Zeitraum *dabei*. Nach ein paar Monaten Experimentieren fanden wir, dass drei Wochen gut sind. Also machen wir Zeiträume von Drei-Wochen-Sprints.

Manchmal wird es sich etwas zu lang anfühlen, manchmal etwas zu kurz. Aber dadurch, dass man bei derselben Länge bleibt, wird dies wie ein betrieblicher Herzschlag, in den sich jeder bequem einfindet. Es gibt keinen Streit über Releasetermine und sowas, weil jeder weiß, dass alle drei Wochen ein Release ist.

> Die meisten Scrum-Teams, die ich treffe (tatsächlich fast alle) landen bei Zwei- oder Drei-Wochen-Sprints. Eine Woche ist fast immer zu kurz ("Wir haben kaum mit dem Sprint angefangen, und jetzt planen wir schon die Demo! Stressig! Wir schaffen es nie, Fahrt aufzunehmen und den Entwicklungsflow zu genießen!"). Und vier Wochen ist fast immer zu lang ("Unsere Planungstreffen sind eine Folter, und unsere Sprints werden ständig unterbrochen!"). Nur eine Beobachtung.

Das Sprintziel definieren

Es passiert fast jedes Mal. An irgendeinem Punkt während des Sprintplanungstreffens frage ich: "Also, was ist das Ziel dieses Sprints?" und jeder starrt mich ausdruckslos an, und der Product Owner runzelt die Stirn und kratzt sich am Kinn.

Aus irgendeinem Grund ist es schwierig, sich ein Sprintziel einfallen zu lassen. Aber ich habe beobachtet, dass es sich wirklich auszahlt, darauf zu drängen. Besser als eine halbherzige Scheiße als Ziel ist gar keins. Das Ziel könnte sein "mehr Geld damit machen" oder "die drei TOP-Priorität-Storys fertigstellen" oder "den CEO beeindrucken" oder "das System gut genug machen, um es für eine Live-Beta-Group auszuliefern" oder "füge einfachen Back-Office-Support hinzu" oder was-auch-immer. Wichtig ist, dass es aus Geschäftssicht formuliert ist, nicht in technischen Ausdrücken. Das bedeutet, mit einer Formulierung, die Leute außerhalb des Teams verstehen können.

Das Sprintziel sollte die fundamentale Frage "Warum machen wir diesen Sprint? Warum gehen wir statt dessen nicht alle in Urlaub?" beantworten. In der Tat ist ein Weg, ein Sprintziel aus dem Product Owner herauszukitzeln, wortwörtlich diese Frage zu stellen.

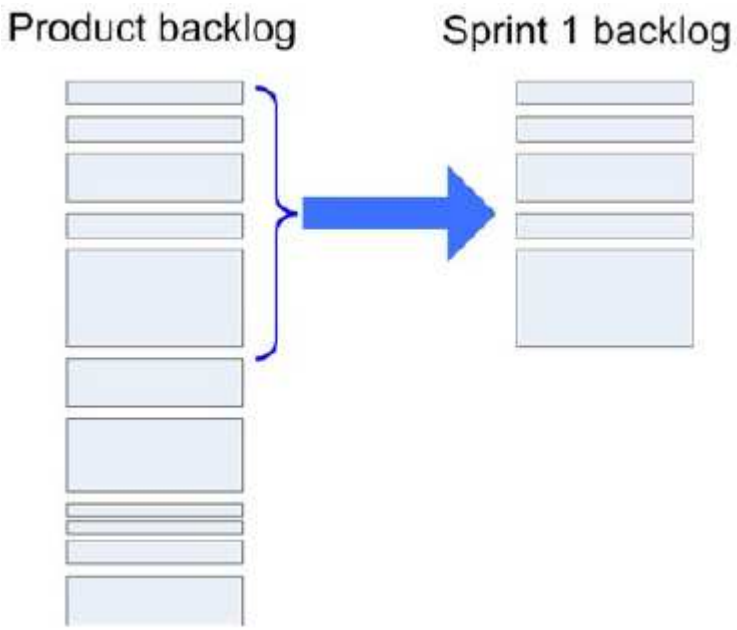
Das Ziel sollte etwas sein, das noch nicht erreicht worden ist. "Den CEO beeindrucken" könnte ein feines Ziel sein, aber nicht, wenn er bereits vom System, wie es jetzt schon läuft, beeindruckt ist. In diesem Fall könnte jeder nach Hause gehen und das Sprintziel wäre schon erreicht.

Das Sprintziel mag während der Sprintplanung eher blöd und gekünstelt erscheinen, aber oft kommt es mitten im Sprint zum Einsatz, wenn Leute beginnen, irritiert zu sein darüber, was sie tun sollen. Wenn Du mehrere Scrumteams hast (wie wir), die an verschiedenen Produkten arbeiten, ist es sehr nützlich, in der Lage zu sein, die Sprintziele aller Teams auf einer einzelnen Wikiseite (oder sonstwo) aufzulisten und sie an prominenter Stelle zu platzieren, so dass jeder im Unternehmen (nicht nur das Topmanagement) weiß, was das Unternehmen gerade tut – und warum!

> OK, sogar der Scrum-Guide stimmt damit überein und sagt, dass alle Sprints ein Sprintziel haben sollten. Aber ich finde, dass es nicht wichtig ist, ein Ziel auf dem Sprintlevel zu haben; es kann genauso gut sein, ein Ziel auf einem höheren Level zu haben, das mehrere Sprints umfasst oder den nächsten Releasezyklus. Stell nur sicher, dass ein Sprint *irgendwas* mehr ist als nur "lass uns ein Bündel Storys durchackern", oder Du könntest feststellen, dass das Team sich einen ernsten Fall von Langweilig einfängt.

Entscheiden, welche Storys im Sprint sind

Eine der Hauptaufgaben im Sprintplanungstreffen ist zu entscheiden, welche Storys im Sprint sein sollen. Genauer gesagt, welche Storys aus dem Produkt-Backlog in das Sprint-Backlog kopiert werden sollen.



Sieh Dir obige Grafik an. Jedes Rechteck steht für eine Story, sortiert nach Wichtigkeit. Die wichtigste Story steht ganz oben. Die Größe des Rechtecks stellt die Größe der Story dar (d. h. geschätzter Aufwand in Story-Punkten). Die Höhe der Klammer stellt die *geschätzte Umsetzungsgeschwindigkeit* dar, d. h. wie viele Story-Punkte das Team glaubt schaffen zu können, um den nächsten Sprint fertigzustellen.

Das Sprint-Backlog rechts ist eine Momentaufnahme von Storys aus dem Produkt-Backlog. Es repräsentiert die Liste an Storys, für die das Team für diesen Sprint die Umsetzung zusichert.

Das *Team* entscheidet, wie viele Storys im nächsten Sprint enthalten sind. Nicht der Product Owner oder sonst irgendwer.

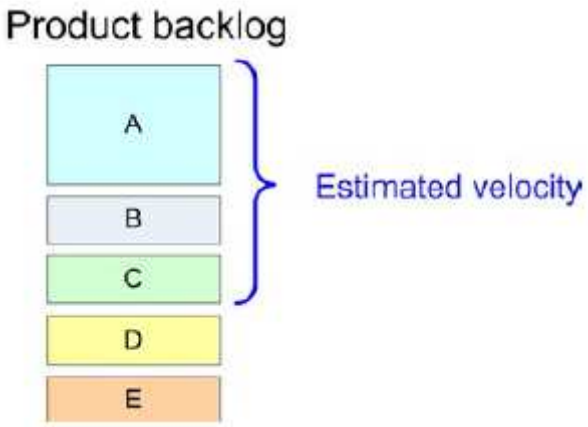
Das wirft zwei Fragen auf:

1) Wie entscheidet das Team, welche Storys im Sprint enthalten sein sollen?
2) Wie kann der Product Owner ihre Entscheidung beeinflussen?

Ich werde mit der zweiten Frage beginnen.

Wie kann der Product Owner beeinflussen, welche Storys es in den Sprint schaffen?

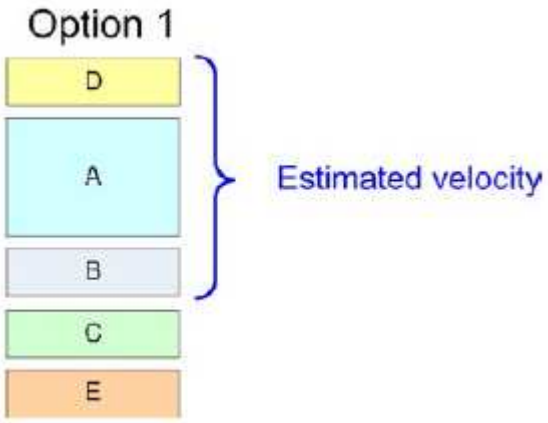
Sagen wir, wir haben im Sprintplanungstreffen folgende Situation.



Produkt-Backlog mit geschätzter Umsetzungsgeschwindigkeit

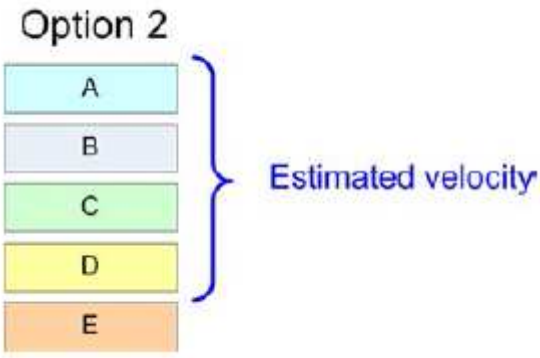
Der Product Owner ist enttäuscht, dass Story D nicht im Sprint sein wird. Welche Möglichkeiten hat er im Sprintplanungstreffen?

Eine Möglichkeit ist, anders zu priorisieren. Wenn er Eintrag D den höchsten Wichtigkeitslevel gibt, ist das Team verpflichtet, dies zuerst in den Sprint reinzunehmen (in diesem Fall fällt Story C raus).



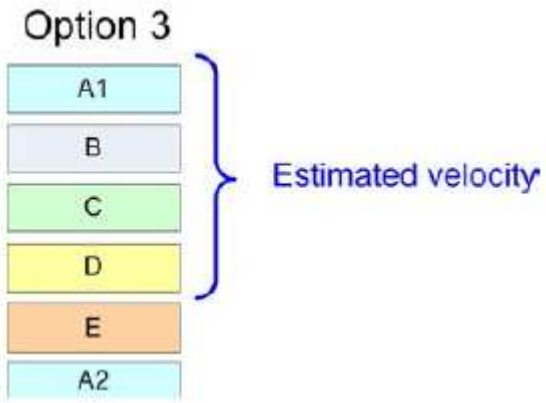
Möglichkeit 1 mit geschätzter Umsetzungsgeschwindigkeit

Die zweite Möglichkeit ist, den Umfang zu ändern – den Umfang der Story A zu verringern, bis das Team glaubt, dass Story D in den Sprint passt.



Möglichkeit 2 mit geschätzter Umsetzungsgeschwindigkeit

Die dritte Möglichkeit ist, eine Story aufzuteilen. Der Product Owner könnte entscheiden, dass es einige Aspekte einer Story gibt, die tatsächlich nicht so wichtig sind, so dass er A in zwei Storys A1 und A2 mit unterschiedlichen Wichtigkeitsstufen aufteilt.



Möglichkeit 3 mit geschätzter Umsetzungsgeschwindigkeit

Wie Du siehst, obwohl der Product Owner die geschätzte Umsetzungsgeschwindigkeit im Allgemeinen nicht kontrollieren kann, gibt es viele Möglichkeiten, durch die er beeinflussen kann, welche Storys es in den Sprint schaffen.

Wie entscheidet ein Team, welche Storys im Sprint enthalten sein sollen?

Dafür nutzen wir zwei Methoden:

1) Bauchgefühl
2) Kalkulation der Umsetzungsgeschwindigkeit

Schätzen mit Hilfe des Bauchgefühls

Scrum Master (zeigt auf den als wichtigsten eingestuften Eintrag im Produkt-Backlog): Hey, Leute, können wir Story A in diesem Sprint fertigstellen?

Lisa: Ob das fertig wird….!? Natürlich wird das fertig! Wir haben drei Wochen, und das ist ein völlig triviales Feature.

Scrum Master (zeigt auf den zweitwichtigesten Eintrag): OK, was, wenn wir Story B auch noch reinnehmen?

Tom und Lisa (unisono): Auch ein Kinderspiel.

Scrum Master: OK, was ist mit Story A und B und C?

Sam (an den Product Owner): Ist in Story C erweiterte Fehlerbehandlung mit drin?

Product Owner: Nein, das kannst Du für diesmal weglassen. Implementiert nur die einfache Fehlerbehandlung.

Sam: Dann sollte C auch kein Problem sein.

Scrum Master: OK, was, wenn wir Story D dazu nehmen?

Lisa: Hmm....

Tom: Ich denke, das könnten wir schaffen.

Scrum Master: 90% sicher? 50%?

Lisa und Tom: Ziemlich sicher mit 90%.

Scrum Master: OK, dann ist D drin. Was, wenn wir Story E dazu nehmen?

Sam: Vielleicht.

Scrum Master: 90%? 50%?

Sam: Ich würde sagen, eher bei 50%.

Lisa: Ich bin skeptisch.

Scrum Master: OK, dann lassen wir es raus. Wir sagen die Umsetzung für A, B, C und D zu. Wir werden natürlich E ferstigstellen, wenn wir es schaffen, aber niemand sollte fest damit rechnen, also lassen wir es aus dem Spritplan raus. Wie klingt das?

Alle: OK!

Bauchgefühl funktioniert ziemlich gut bei kleinen Teams und kurzen Sprints.

Schätzen mit Hilfe von Kalkulation der Umsetzungsgeschwindigkeit

Diese Methode umfasst zwei Schritte:

1) Geschätzte Umsetzungsgeschwindigkeit festlegen.
2) Kalkulieren, wie viele Storys man hinzunehmen kann, ohne die geschätzte Umsetzungsgeschwindigkeit zu gefährden.

Umsetzungsgeschwindigkeit ist ein Maß für die "Menge an Arbeit, die fertig ist", wobei jeder Eintrag im Sinne von einer ursprünglichen Schätzung gewichtet ist. Das Bild unten zeigt ein Beispiel für die *geschätzte Umsetzungsgeschwindigkeit* zu Beginn eines Sprints und die *tatsächliche Umsetzungsgeschwindigkeit* am Ende desselben Sprints. Jedes Rechteck steht für eine Story, und die Anzahl darin ist die ursprüngliche Schätzung für diese Story.

Beginn und Ende des Sprints mit geschätzter und tatsächlicher Umsetzungsgeschwindigkeit
Fertiggestellte, begonnene und noch nicht begonnene Storys

Denk dran, dass die tatsächliche Umsetzungsgeschwindigkeit auf der *ursprünglichen* Schätzung jeder Story beruht. Alle Aktualisierungen der Zeitschätzung für die Story während des Sprints werden ausgeklammert.

Ich kann Deinen Einwand schon hören: "Wie nützlich soll das sein? Eine hohe oder geringe Umsetzungsgeschwindigkeit mag von einem ganzen Haufen von Faktoren abhängen! Programmierer, die schwer von Begriff sind, unzutreffende erste Schätzungen, unauffällige Ausdehnungen des Umfangs, ungeplante Störungen während des Sprints etc.!"

Ich stimme dem zu, es handelt sich um einen groben Wert. Aber trotzdem ist es ein nützlicher Wert, besonders wenn man ihn damit vergleicht, gar nicht zu schätzen. Er liefert Dir einige harte Fakten. "Ungeachtet der Gründe, hier ist die ungefähre Differenz zwischen dem, wieviel wir dachten, was wir geschafft kriegen, und dem, was wir tatsächlich schafften."

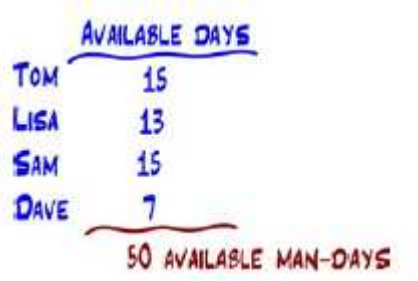
Was ist mit einer Story, die in einem Sprint *fast* fertiggestellt wird? Warum vergeben wir keine Teilpunkte dafür in unserer tatsächlichen Umsetzungsgeschwindigkeit? Naja, das ist so, um die Tatsache zu betonen, dass der Schwerpunkt von Scrum (und im Grunde von allen agilen Softwareentwicklungsmethoden und jeder Lean-Produktion im allgemeinen) darin besteht, Dinge komplett, auslieferbar, fertig zu bekommen! Der Wert von halbfertigem Zeugs ist Null (könnte sogar im Minusbereich liegen). Nimm Donald Reinertsens *Managing the Design Factory* oder eins von Poppendiecks Büchern, um mehr darüber zu erfahren.

Also, durch welche geheimnisvolle Magie kommen wir zu Schätzungen für die Umsetzungsgeschwindigkeit?

Eine sehr einfache Möglichkeit, die Umsetzungsgeschwindigkeit zu schätzen, ist, sich die Geschichte des Teams anzusehen. Wie war während der letzten paar Sprints die Umsetzungsgeschwindigkeit? Dann nimm an, dass die Umsetzungsgeschwindigkeit im nächsten Sprint ungefähr genauso sein wird.

Diese Technik ist als *Yesterday's Weather* bekannt. Dies ist für Teams, die bereits ein paar Sprints durchgeführt haben und die im nächsten Sprint ziemlich genauso vorgehen werden, mit derselben Teamgröße und denselben Arbeitsbedingungen etc., nur vernünftig (sofern Statistiken verfügbar sind). Das ist natürlich nicht immer der Fall.

Eine anspruchsvollere Variante ist, eine simple Ressourcenkalkulation zu machen. Sagen wir, wir planen einen Drei-Wochen-Sprint (15 Arbeitstage) mit einem 4-Personen-Team. Lisa wird an zwei Tagen im Urlaub sein. Dave stehen nur 50 % dafür zur Verfügung und er wird an einem Tag im Urlaub sein. Nimmt man all dies zusammen…



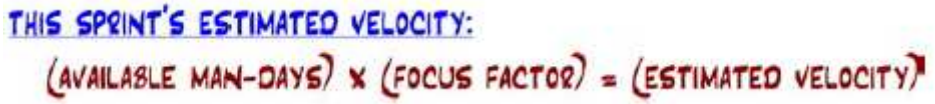
Verfügbare Tage und verfügbare Manntage

… ergibt 50 Manntage, die wir für diesen Sprint zur Verfügung haben.

> Warnung: Hier ist der Teil, den ich wirklich hasse. Ich würde gern die nächsten paar Seiten rausreißen! Aber mach weiter und lies es, wenn Du neugierig bist, und ich werde später erklären, warum.

Ist das unsere geschätzte Umsetzungsgeschwindigkeit? Nein! Denn unsere Schätzeinheit ist *Story-Punkte*, die, in unserem Fall, ungefähr mit "idealen Manntagen" korrespondieren. Ein idealer Manntag ist ein perfekt effektiver Tag ohne Störungen, was sehr selten ist. Außerdem haben wir Sachen zu berücksichtigen wie ungeplante Arbeiten, die zum Sprint dazu kommen, Leute werden krank etc.

Also wird unsere geschätzte Umsetzungsgeschwindigkeit sicherlich weniger als 50 sein. Aber wieviel weniger? Wir setzen dafür den Begriff "Fokusfaktor" ein.

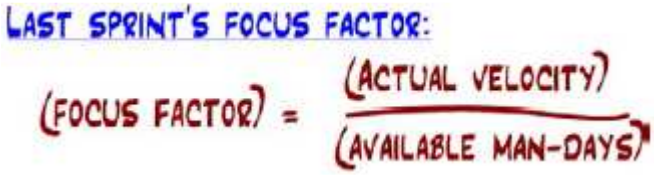


Geschätzte Umsetzungsgeschwindigkeit für diesen Sprint:
(verfügbare Manntage) x (Fokusfaktor) = (geschätzte Umsetzungsgeschwindigkeit)

Der Fokusfaktor ist eine Schätzung darüber, wie konzentriert das Team ist. Ein niedriger Fokusfaktor könnte bedeutet, dass das Team mit vielen Störungen rechnet oder seine eigenen Zeitschätzung für optimistisch hält.

> Bla, bla, bla. Ein Haufen Mathe-Mumbo-Jumbo. Nimm einfach Yesterday's Weather (oder Bauchgefühl, wenn Du keine Daten hast) und ignorier den Fokusfaktorunsinn.

Die beste Möglichkeit, einen vernünftigen Fokusfaktor festzusetzen, ist, auf den letzten Sprint zu gucken (oder noch besser auf den Durchschnitt der letzten paar Sprints).



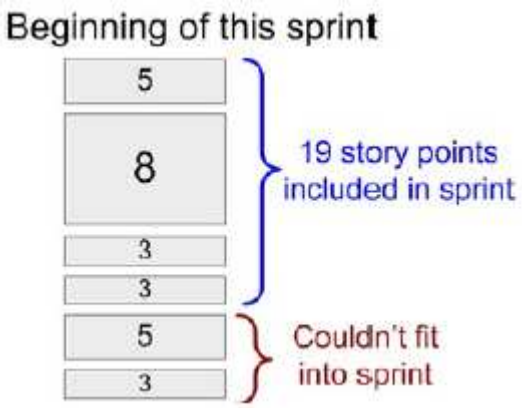
Fokusfaktor für den letzten Sprint:
(Fokusfaktor) = (tatsächliche Umsetzungsgeschwindigkeit) / (verfügbare Manntage)

Die *tatsächliche Umsetzungsgeschwindigkeit* ist die Summe der ursprünglichen Schätzungen aller Storys, die wir im letzten Sprint fertig gestellt haben.

Sagen wir, der letzte Sprint stellte 18 Story-Punkte fertig durch den Einsatz eines Drei-Personen-Teams, das aus Tom, Lisa und Sam bestand, die drei Wochen lang insgesamt 45 Manntage abgearbeitet haben. Und jetzt versuchen wir, unsere geschätzte Umsetzungsgeschwindigkeit für den kommenden Sprint herauszufinden. Um die Sache komplizierter zu machen, kommt ein Neuer, Dave, für diesen Sprint ins Team. Berücksichtigt man Urlaub und anderes Zeug, haben wir für den nächsten Sprint 50 Manntage.

Fokusfaktor für den letzten Sprint und geschätzte Umsetzungsgeschwindigkeit für diesen Sprint: Verhältnis Story-Punkte zu Manntagen vom letzten Mal fließt in die Rechnung für das nächste Mal ein

Also liegt unsere geschätzte Umsetzungsgeschwindigkeit für den kommenden Sprint bei 20 Story-Punkten. Das bedeutet, das Team sollte Storys zum Sprint hinzufügen, bis es ungefähr 20 sind.



Beginn des Sprints mit 19 Story-Punkten im Sprint und Storys, die nicht hinein passen

In diesem Fall könnte das Team die vier obersten Storys mit einer Gesamtsumme von 19 Story-Punkten auswählen oder die fünf obersten Storys mit einer Summe von 24 Story-Punkten. Sagen wir, sie wählen vier Storys, da das den 20 Story-Punkten am nächsten kommt. Im Zweifel wähle weniger Storys.

Da diese vier Storys sich auf 19 Story-Punkte aufsummieren, liegt ihre abschließende geschätzte Umsetzungsgeschwindigkeit bei 19.

Yesterday's Weather ist eine handliche Methode, aber setze sie mit der richtigen Dosis gesunden Menschenverstand ein. Wenn der letzte Sprint ein ungewöhnlich schlechter Sprint war, weil die meisten aus dem Team eine Woche lang krank waren, dann könnte es ungefährlich sein anzunehmen, dass Du nicht wieder so ein Pech hast, und Ihr könnt einen höheren Fokusfaktor für den nächsten Sprint schätzen. Wenn das Team in letzter Zeit ein neues, blitzschnelles, System mit kontinuierlicher Auslieferung installiert hat, könntest Du den Fokusfaktor wahrscheinlich erhöhen, eben dieser Tatsache geschuldet. Wenn eine neue Person im Sprint dazu kommt, musst Du den Fokusfaktor heruntersetzen, um seine Einarbeitung zu berücksichtigen. Etc.

Wann immer möglich, sieh ein paar Sprints in die Vergangenheit und ermittle den Durchschnitt der Zahlen, um verlässlichere Schätzungen zu bekommen.

Was, wenn das Team ganz neu ist, so dass Du gar keine Statistiken hast? Sieh Dir den Fokusfaktor anderer Teams in ähnlichen Situationen an.

Was, wenn Du keine anderen Teams hast, um dort abzugucken? Mutmaße den Fokusfaktor. Die gute Nachricht ist, dass Deine Mutmaßung nur auf den ersten Sprint zutrifft. Danach hast Du Daten und kannst kontinuierlich messen und den Fokusfaktor und die Schätzung der Umsetzungsgeschwindigkeit verbessern.

Der Vorgabewert für den Fokusfaktor, den ich bei neuen Teams nutze, ist normalerweise 70 %, da das der Wert ist, bei dem über die Zeit unsere anderen Teams schließlich gelandet sind.

Welche Schätzmethode setzen wir ein?

Ich habe oben ein paar Methoden erwähnt: Bauchgefühl, Kalkulation der Umsetzungsgeschwindigkeit mit Hilfe von Yesterday's Weather und Kalkulation der Umsetzungsgeschwindigkeit mit Hilfe von verfügbaren Manntagen und geschätztem Fokusfaktor.

Und welche Methoden setzen wir ein?

Wir kombinieren normalerweise alle diese Methoden zu einem gewissen Grad. Dazu braucht es wirklich nicht viel.

Wir sehen uns den Fokusfaktor und die tatsächliche Umsetzungsgeschwindigkeit vom letzten Sprint an. Wir sehen uns die Gesamtressourcenverfügbarkeit dieses Sprints an und schätzen einen Fokusfaktor. Wir diskutieren jede vorhandene Abweichung zwischen diesen zwei Fokusfaktoren und machen Verbesserungen, wo notwendig.

> OK, das ist das Ende des schmerzhaften Abschnitts. Ich benutze den Fokusfaktor gar nicht mehr, weil es Zeit frisst, einen falschen Eindruck von Genauigkeit vermittelt und Dich zwingt, Storys in idealen Manntagen zu schätzen.
>
> Außerdem, der Fokusfaktor beinhaltet die Annahme, dass mehr Leute = höhere Umsetzungsgeschwindigkeit. Manchmal stimmt das, aber manchmal nicht. Wenn wir eine neue Person ins Team nehmen, wird die Umsetzungsgeschwindigkeit normalerweise für die ersten ein oder zwei Sprints *abnehmen*, da Leute Zeit damit verbringen, die neue Person einzuarbeiten. Wenn ein Team zu groß wird (so 10+ Leute), geht die Umsetzungsgeschwindigkeit definitiv runter. Außerdem impliziert der Ausdruck "Fokusfaktor", dass ein Wert unter 100 % bedeutet, dass das Team *unfokussiert* ist, was ein sehr missverständliches Signal ans Management sendet.
>
> Also überspring all das Fokusfaktor- und Manntagzeugs. Sieh nur an, wieviel Du in den letzten paar Sprints fertig gestellt hast, indem Du Story-Punkte zählst, oder zähl sogar nur die Anzahl der Storys, wenn Du überhaupt keine Schätzungen hast. Dann schnapp Dir ungefähr so viele Storys für diesen Sprint. Wenn Du viele Unterbrechungen im Sprintplan hast (wie z. B. dadurch, dass zwei Leute auf Fortbildung sind), dann nimm ein paar Storys raus, bis es sich genau richtig anfühlt. Je weniger historische Daten Du hast, desto mehr musst Du auf Dein Bauchgefühl vertrauen.
>
> Mach dies und Deine Sprintplanungstreffen werden kürzer, effektiver und machen mehr Spaß. Und Deine Pläne werden, entgegen jeder Erwartung, am Ende wahrscheinlich sogar öfter eintreffen.

Sobald wir eine vorläufige Liste an Storys haben, die in den Sprint kommen sollen, mache ich den "Bauchgefühl"-Check. Ich bitte das Team, die Anzahl für einen Moment zu ignorieren und nur darüber nachzudenken, ob sich dies zum Schlucken für einen Sprint wie ein reeller Brocken *anfühlt*. Wenn es sich nach zuviel anfühlt, nehmen wir eine oder zwei Storys heraus. Und umgekehrt.

Letzten Endes ist das Ziel einfach zu entscheiden, welche Storys in den Sprint kommen. Fokusfaktor, Verfügbarkeit von Ressourcen und geschätzte Umsetzungsgeschwindigkeit sind einfach ein Mittel, dieses Ziel zu erreichen.

Warum wir Karteikarten benutzen

Den größten Teil eines Sprintplanungstreffens verbringen wir damit, uns mit Storys im Produkt-Backlog auseinander zu setzen. Schätzen, umpriorisieren, klären, große in kleinere aufteilen etc.

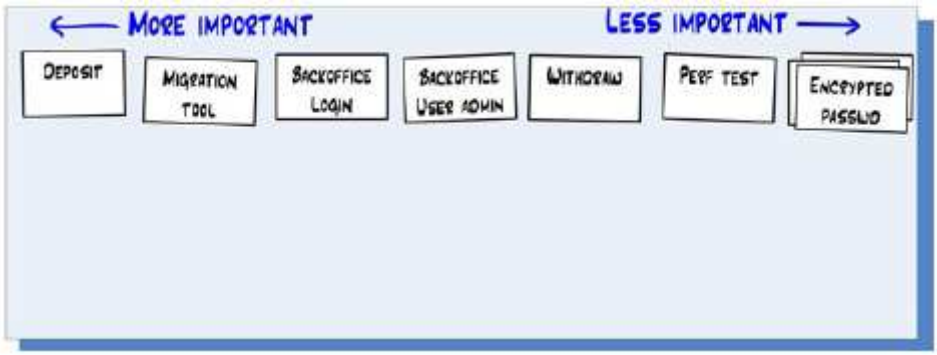
Wie machen wir dies in der Praxis?

Naja, gewöhnlich schalteten die Teams den Beamer ein, zeigten das Excel-basierte Backlog und Einer (typischerweise der Product Owner oder Scrum Master) nahm die Tastatur, murmelte durch jede Story durch und lud zur Diskussion ein. Während das Team und der Product Owner die Prioritäten und Details diskutierten, aktualisierte der Typ an der Tastatur direkt im Excel.

Klingt gut? Nun, das ist es nicht. Es funktioniert üblicherweise beschissen. Und, was schlimmer ist, das Team bemerkt normalerweise nicht, dass es beschissen ist, bis sie das Ende des Treffens erreichen und feststellen, dass sie noch nicht geschafft haben, durch die Liste von Storys durchzukommen!

> Oh, diese Schmerzen… .

Eine Lösung, die viel besser funktioniert, ist, Karten zu beschriften und diese an die Wand zu hängen (oder auf einen großen Tisch zu legen).



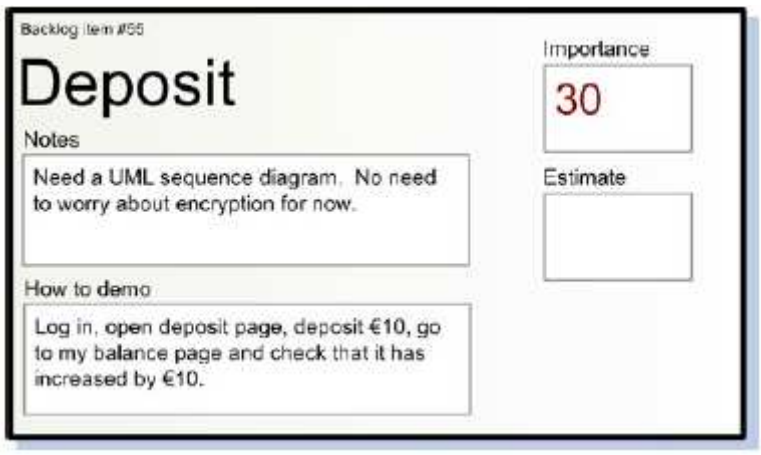
wichtiger …. weniger wichtig

Anzahlung | Migrationstool | Backoffice-Login | Backoffice- User-Admin | Perf. Test | verschlüsseltes Passwt

Dies ist, verglichen zu einem Rechner mit Beamer, ein überlegenes User Interface, denn:

- Leute stehen auf und laufen rum => sie bleiben wach und länger bei der Sache.
- Jeder fühlt sich persönlich mehr involviert (eher als nur der Typ an der Tastatur).
- Mehrere Storys können gleichzeitig bearbeitet werden.
- Umpriorisieren ist sehr einfach – einfach die Karten rumschieben.
- Nach dem Treffen können die Karten direkt von der Wand mitgenommen werden in das Teambüro und als Wand-basierte Aufgabentafel genutzt werden (s. Seite 47 "Wie wir Sprint-Backlogs führen").

Du kannst sie entweder mit der Hand schreiben oder (wie wir es normalerweise machen) ein einfaches Script benutzen, um druckbare Karten direkt aus dem Produkt-Backlog zu generieren.



Karte "Anzahlung"

P.S. Das Script steht auf meinem Blog bereit unter http://blog.crisp.se/henrikkniberg.

> Der einfachste Weg, ein Script zu finden ist "index card generator" zu googeln. Ich kann nicht glauben, dass das gute alte Hacken immer noch erfolgreich ist! Einige nette Leute haben geholfen, es auch auf Google Spreadsheets zu portieren. Jedes ordentliche Backlog-Management-Tool wird eine Druckfunktion wie diese haben. Probier mit verschiedenen Tools herum und finde heraus, was in Deinem Kontext am besten funktioniert. Stell einfach sicher, dass Du das Tool für Deinen Prozess anpasst und nicht umgekehrt.

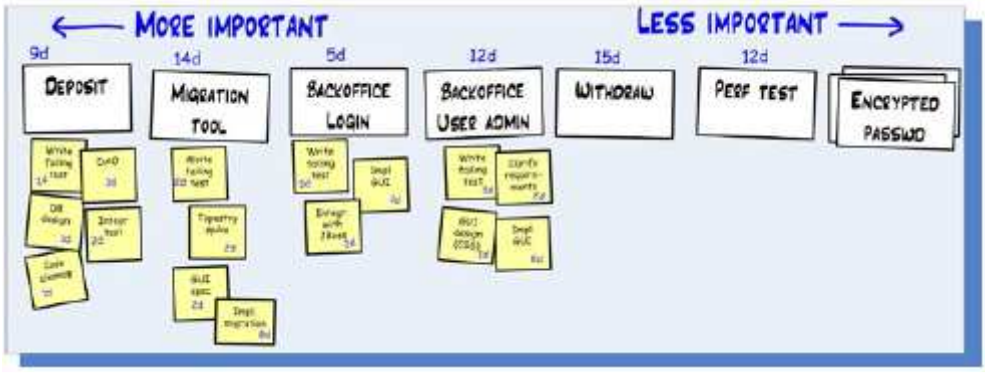
Wichtig: Nach dem Sprintplanungstreffen aktualisiert unser Scrum Master manuell das Excel-basierte Produkt-Backlog unter Berücksichtigung jedweder Änderungen, die gemacht wurden, um die physikalischen Story-Karten zu machen. Ja, dies ist ein kleiner verwaltungstechnischer Aufwand, aber wir denken, das ist absolut akzeptabel, berücksichtigt man, wie viel effizienter das Sprintplanungstreffen mit physikalischen Karten ist.

Ein Hinweis zum "Wichtig"-Feld… Dies ist die Wichtigkeit, wie sie im Excel-basierten Produkt-Backlog spezifiziert ist, zum Zeitpunkt des Ausdruckens. Sie auf der Karte zu haben, macht es leicht, die Karten physisch nach Wichtigkeit zu sortieren (normalerweise positionieren wir die wichtigeren Einträge links und die weniger wichtigen Einträge rechts). Wenn die Karten jedoch einmal an der Wand hängen, könnt Ihr das Wichtigkeitsranking ignorieren und statt dessen die physikalische Sortierung an der Wand nutzen, um die relative Wichtigkeit zu kennzeichnen. Wenn der Product Owner zwei Einträge vertauscht, verschwende keine Zeit, das Wichtigkeitsranking auf dem Papier zu ändern. Stell nur sicher, dass Du das Wichtigkeitsranking im Excel-basierten Produkt-Backlog nach dem Treffen aktualisierst.

> Oder lass Wichtigkeitsrankings einfach weg. Ups, Ich habe das schon ein paar Mal erwähnt. Wiederhole ich mich? Wiederhole ich mich?

Zeitschätzungen sind normalerweise leicht zu machen (und genauer), wenn eine Story in mehrere Aufgaben aufgeteilt wird. Ehrlich gesagt, benutzten wir den Ausdruck "Aktivität", weil das Wort "Task" im Schwedischen etwas *komplett* anderes bedeutet. :o) Dies ist außerdem nett und einfach mit den Karten zu machen. Du kannst das Team in Zweiergruppen aufteilen und parallel jeweils eine Story herunterbrechen lassen.

Physikalisch machen wir das, indem wir kleine Haftnotizzettel unter jede Story packen, jede Haftnotiz repräsentiert eine Aufgabe innerhalb dieser Story.



Wir aktualisieren das Excel-basierte Produkt-Backlog nicht unter Berücksichtigung unserer Aufgabenaufteilung, aus zwei Gründen:

Die Aufgabenaufteilung ist normalerweise sehr unbeständig, d. h. während des Sprints ändert sie sich ständig und wird verfeinert, so dass es zuviel Ärger macht, das Excel-Produkt-Backlog zu synchronisieren.

Der Product Owner muss ohnehin auf diesem Detaillierungsgrad nicht einbezogen werden.

Genau wie bei den Story-Karten können die Aufgabenaufteilungshaftnotizen direkt im Sprint-Backlog wiederverwendet werden (s. Seite 47 "Wie wir Sprint-Backlogs führen").

Definition von *Fertig* – Definition of *Done*

Es ist wichtig, dass der Product Owner und das Team sich auf eine klare Definition für "Definition of Done"[7] einigen.

> SEHR wichtig!

Ist eine Story abgeschlossen, wenn der ganze Code eingecheckt ist? Oder ist sie abgeschlossen, wenn sie in der Testumgebung vorhanden ist und vom Team, das für den Integrationstest verantwortlich ist, verifiziert ist? Wann immer möglich, nutzen wir "fertig zur Auslieferung im Produktivsystem" als Definition of Done, aber manchmal müssen wir "auf dem Testserver ausgeliefert und fertig für Akzeptanztest" als Definition of Done einsetzen.

Zu Beginn hatten dafür wir eine detaillierte Checkliste. Jetzt sagen wir häufig einfach "eine Story ist abgeschlossen, wenn der Tester im Scrumteam das sagt". Dann liegt es beim Tester sicherzustellen, dass die Intention des Product Owners vom Team verstanden ist und dass der Eintrag genug "abgeschlossen" ist, um ihn als "Definition-of-done"-akzeptiert anzusehen.

> OK, das ist ein bisschen lahm. Eine konkrete Checkliste ist nützlicher – stell nur sicher, dass sie nicht zu lang ist. Behandle sie als Empfehlung, nicht als heilige Schrift. Konzentrier Dich auf die Dinge, die Leute leicht vergessen (wie "Releasenotizen aktualisieren" oder "keine technischen Schulden hinterlassen" oder "Feedback von echten Nutzern einholen").

Wir haben festgestellt, dass nicht alle Storys gleich behandelt werden können. Eine Story namens "Anfrageformular Nutzer" wird ganz anders gehandhabt als eine Story namens "Betriebsanleitung". Im letzteren Fall könnte die Definition of Done einfach "abgenommen vom Team, das für den Betrieb verantwortlich ist" bedeuten. Deswegen ist gesunder Menschenverstand oft besser als eine formale Checkliste.

Wenn Du Dich oft in Irritationen verläufst, was die Definition of Done betrifft (was uns zu Beginn passiert ist), solltest Du wahrscheinlich besser ein "Definition of Done"-Feld bei jeder einzelnen Story haben.

Zeitschätzung mit Hilfe von Planungspoker

Schätzen ist eine Teamaktivität – normalerweise ist jedes Teammitglied in das Schätzen jeder einzelnen Story involviert. Warum?

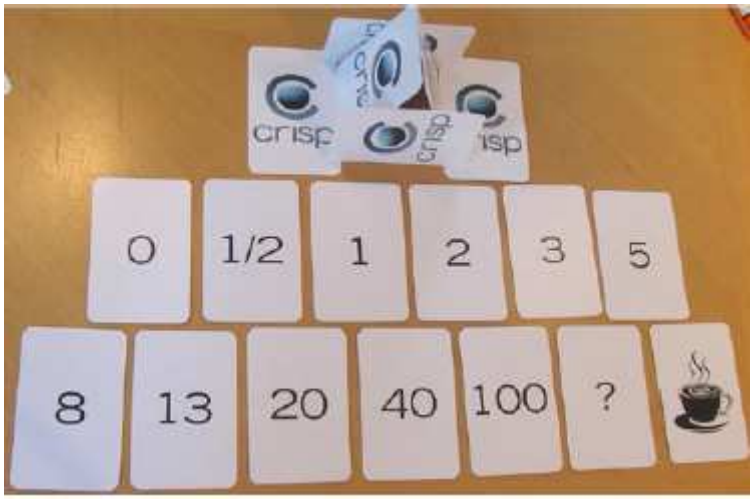
- Zum Zeitpunkt des Planens wissen wir üblicherweise nicht genau, wer welche Teile welcher Story implementieren wird.
- Storys erfordern normalerweise mehrere Leute und verschiedene Arten von Expertise (User-Interface-Design, Codieren, Testen etc.).
- Um eine Schätzung geben zu können, braucht ein Teammitglied ein gewisses Verständnis dessen, worum es in der Story geht. Indem wir jedes Teammitglied um eine Schätzung zu einem Punkt bitten, gewährleisten wir, dass jedes Teammitglied versteht, worum es bei dem Punkt geht. Dies erhöht die Wahrscheinlichkeit, dass Teammitglieder einander während des Sprints helfen. Dies erhöht außerdem die Wahrscheinlichkeit, dass wichtige Fragen zur Story früh angeschnitten werden.
- Während wir alle bitten, eine Story zu schätzen, stoßen wir oft auf Abweichungen, bei denen verschiedene Teammitglieder wild voneinander abweichende Schätzungen für dieselbe Story haben. Solch ein Zeugs entdeckt und diskutiert man besser früher als später.

Wenn du das Team bittest, eine Schätzung abzugeben, wird normalerweise diejenige Person, die die Story am besten versteht, die erste sein, die mit einer Schätzung herausplatzt. Leider hat dies einen starken Einfluss auf die Schätzungen aller andern.

[7] "Definition of Done" ist ein Fachbegriff aus Scrum, der die Kriterien bezeichnet, die erfüllt sein müssen, damit eine Anforderung als umgesetzt gilt. (Anm. der Übersetzerin)

Es gibt eine exzellente Methode, um das zu verhindern – sie heißt Planungspoker (als Begriff geprägt von Mike Cohn, glaube ich).

Tatsächlich erzählte Mike, er lernte ihn von James Grenning und James wird wahrscheinlich sagen, er bekam die Idee von jemand anderem. Egal. Wir stehen alle auf den Schultern von Giganten[8]. Oder vielleicht sind wir ein Haufen Zwerge, die einander auf den Schultern stehen. Äh, wie auch immer…. – Du weißt, was ich meine.



Jedes Teammitglied bekommt ein Kartenset mit 13 Karten wie auf dem Bild oben.

<marktstand>Wir verkaufen diese Kartensets auf planningpoker.crisp.se. Und sie sehen jetzt cooler aus als auf dem Foto, obwohl Du wahrscheinlich billigere finden kannst, wenn Du rumgoogelst. Oh, und auf der Website verkaufen wir auch noch etwas wirklich Cooles, genannt Jimmy Cards von meinem Kollegen Jimmy (ja, wir hatten Probleme, einen Namen für die Karten zu finden). Schau vorbei!</markstand>

Wann immer eine Story geschätzt werden soll, wählt jedes Teammitglied eine Karte aus, die seine Zeitschätzung repräsentiert (in Story-Punkten) und legt sie mit der Zahlseite nach unten auf den Tisch. Wenn alle Teammitglieder fertig sind, werden die Karten auf dem Tisch gleichzeitig aufgedeckt. So ist jedes Teammitglied gezwungen, selbst zu denken statt sich an der Schätzung eines andern zu orientieren.

Wenn es große Abweichungen zwischen zwei Schätzungen gibt, diskutiert das Team die Abweichungen und versucht, ein gemeinsames Bild davon aufzubauen, was zur Story gehört. Sie könnten auch die Aufgabe irgendwie aufteilen.

Danach schätzt das Team noch einmal. Diese Schleife wird wiederholt, bis die Zeitschätzungen zusammenlaufen, d. h. alle ungefähr dasselbe für diese Story schätzen.

Es ist wichtig, die Teammitglieder zu erinnern, dass sie die Gesamtmenge an Arbeit schätzen, die mit der Story zusammenhängt. Nicht nur ihren Teil der Arbeit. Der Test sollte nicht nur den Testaufwand schätzen.

Beachte, dass die Reihe der Zahlen nicht linear ist. Zum Beispiel gibt es nichts zwischen 40 und 100. Warum?

Dies ist so, um gar nicht erst ein falsches Gefühl von Genauigkeit für große Zeitschätzungen aufkommen zu lassen. Wenn eine Story auf ungefähr 20 Story-Punkte geschätzt wird, ist es nicht relevant, darüber zu diskutieren, ob es eher 20 oder 18 oder 21 sein sollten. Alles, was wir wissen, ist, dass es eine große Story ist und dass sie schwer zu schätzen ist. Also ist 20 unser grober Richtwert.

Du willst detailliertere Schätzungen? Teile die Story in kleinere Storys auf und schätze statt dessen die kleineren Storys!

Und nein, Du kannst nicht mogeln, indem Du eine Kombination aus 5 und 2 zu 7 machst. Du musst Dich entscheiden zwischen 5 oder 8; es gibt keine 7.

[8] Das bedeutet, wir bauen auf den Errungenschaften unserer Vorgänger auf. Redewendung im Englischen: "We all stand on shoulders of giants." (Anm. der Übersetzerin)

Einige Spezialkarten, die es zu beachten gilt:

- 0 = "Diese Story ist bereits abgeschlossen" oder "diese Story ist so gut wie nix, nur ein paar Minuten Arbeit"
- ? = "Ich habe absolut überhaupt keine Vorstellung. Gar keine."
- Kaffeetasse = "Ich bin zu erschöpft, um zu denken. Lass uns eine kurze Pause machen."

Eine andere Firma ließ sich ein noch cooleres Kartenset einfallen, die No-Bullshit-Schätzkarten (estimation.lunarlogic.io). Es enthält nur drei Karten[9]:

1 (eins)
WSG (wirklich scheiß groß)
KVA (kein verdammter Anhaltspunkt)

Ziemlich cool! Ich wünschte, mir wäre das eingefallen. Obwohl ich sehr wohl die Kaffeetassenidee für mich in Anspruch nehme.

Storys klären

Am schlimmsten ist, wenn ein Team bei der Sprint-Demo stolz ein neues Feature demonstriert und der Product Owner die Stirn runzelt und sagt: "Naja, das ist ganz nett, aber das ist *nicht das, worum ich gebeten hatte!*"

Wie gewährleistest Du, dass das Story-Verständnis des Product Owners zum Verständnis des Teams über diese Story passt? Oder dass jedes Teammitglied dasselbe Verständnis der Story hat? Naja, das kannst Du nicht. Aber es gibt eine einfache Methode, um die krassesten Missverständnisse zu identifizieren.

Die einfachste Methode ist schlicht sicherzustellen, dass alle Felder für jede Story ausgefüllt werden (oder konkreter, für jede Story, die wichtig genug ist, um für diesen Sprint berücksichtigt zu werden).

Einige nennen das "Definition of Ready". Also, "Definition of Done" ist eine Checkliste dafür, wann eine Story abgeschlossen ist, und "Definition of Ready" ist eine Checkliste dafür, wann eine Story bereit ist, in einen Sprint gezogen zu werden. Sehr nützlich.

Beispiel 1

Das Team und der Product Owner sind mit dem Sprintplan glücklich und bereit, das Treffen zu beenden. Der Scrum Master sagt, "Warte eine Sekunde. Für die Story mit dem Namen 'Nutzer hinzufügen' gibt es keine Schätzung. Lasst sie uns schätzen!" Nach ein paar Runden Planungspoker einigt sich das Team auf 20 Story-Punkte, woraufhin der Product Owner wütend aufsteht: "Waaaas?!" Nach ein paar Minuten hitziger Diskussion stellt sich heraus, dass das Team den Umfang von "Nutzer hinzufügen" falsch verstanden hat; sie dachten, damit ist "ein nettes Web-GUI zum Hinzufügen, Entfernen, Löschen und Suchen von Nutzern" gemeint, während der Product Owner einfach an "Nutzer durch manuelles SQL-Tun Richtung Datenbank hinzufügen" dachte.

Beispiel 2

Das Team und der Product Owner sind mit dem Sprintplan glücklich und bereit, das Treffen zu beenden. Der Scrum Master sagt, "Warte eine Sekunde. Wie wird die Story mit dem Namen 'Nutzer hinzufügen' später demonstriert?"

Es folgt Gemurmel und nach einer Minute steht jemand auf und sagt, "Naja, zuerst loggen wir uns in der Website ein und dana…" – und der Product Owner unterbricht.

"In die Website einloggen?! Nein, nein, nein, diese Funktionalität darf überhaupt nicht Teil der Website sein. Das soll ein dummes kleines SQL-Script nur für Admins werden."

Die "How-to-Demo"-Beschreibung einer Story kann (und sollte) *sehr kurz* sein! Sonst wird man das Sprintplanungstreffen nicht rechtzeitig abschließen können. Es handelt sich im Wesentlichen um eine Beschreibung in einfachen

[9] Englisches Original:
- 1 (one)
- TFB (too f*cking big)
- NFC (no f*ucking clue)

Worten auf hohem Niveau darüber, wie die charakteristischen Testszenarien manuell ausgeführt werden. "Tu dies, dann das, dann verifiziere dies."

Ich habe herausgefunden, dass diese einfache Beschreibung *oftmals* wichtige Missverständnisse über den Umfang einer Story sichtbar macht. Gut, sie früh zu entdecken, richtig?

> Ich mag diese Methode immer noch und setze sie ein, wann immer eine Story gefühlt vage daher kommt. Mach Dinge konkret. Eine Alternative ist, eine Wireframe-Skizze zu malen oder eine Liste von Akzeptanzkriterien anzulegen. Denk bei einer Story an eine Problemaussage auf hohem Niveau und bei der "Definition of Done" an ein konkretes Beispiel, wie eine Story aussehen könnte, wenn sie fertiggestellt ist.

Storys aufschlüsseln in kleinere Storys

Storys sollten nicht zu klein oder zu groß sein (im Sinne von Schätzungen). Wenn Du einen Haufen 0,5-Punkte-Storys hast, bist Du wahrscheinlich ein Opfer des Mikromanagements geworden. Andererseits stellt eine 40-Punkte-Story ein hohes Risiko dar, *teilweise* fertiggestellt zu werden, was keinen Wert für Deine Firma erzeugt und auch noch die Administrationsarbeit erhöht. Außerdem, wenn Deine geschätzte Umsetzungsgeschwindigkeit 70 ist und Deine zwei toppriorisierten Storys jeweils mit 40 Story-Punkten gewichtet sind, wird die Planung irgendwie schwierig. Du musst zwischen Unter-Verpflichtung (d. h. nur einen Eintrag reinnehmen) und Über-Verpflichtung (d. h. beide Einträge reinnehmen) wählen.

Ich habe beobachtet, dass es fast immer möglich ist, eine große Story in kleinere Storys zu unterteilen. Stelle nur sicher, dass die kleineren Storys immer noch Leistungen mit Geschäftswert darstellen.

Wir bemühen uns um Storys, die mit zwei bis acht Personentagen gewichtet sind. Unsere Umsetzungsgeschwindigkeit liegt normalerweise bei ungefähr 40 bis 60 für ein klassisches Team, und so haben wir irgendwas um die 10 Storys pro Sprint. Manchmal mit fünf Storys eher wenig und manchmal mit 15 eher viele. Das ist eine handhabbare Anzahl von Karteikarten, um damit umzugehen.

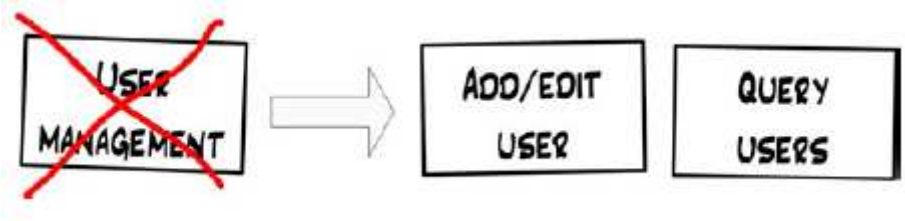
> Fünf bis 15 Storys in einem Sprint ist eine nützliche Richtschnur. Weniger als fünf bedeutet normalerweise, dass die Storys zu groß sind für die Länge des Sprints, während mehr als 15 normalerweise bedeutet, dass das Team zu viel hineingenommen hat und nicht alles fertigstellen wird (oder die Storys sind zu klein, sind ein Anzeichen für Mikromanagement).

Storys aufschlüsseln in Aufgaben

Warte eine Sekunde… Was ist der Unterschied zwischen "Aufgabe"[10] und "Story"? Eine sehr berechtigte Frage.

Die Unterscheidung ist recht einfach. Storys stellen auslieferbares Zeugs dar, für das sich der Product Owner interessiert. Aufgaben sind nicht-auslieferbares Zeugs bzw. Zeugs, das den Product Owner nicht interessiert.

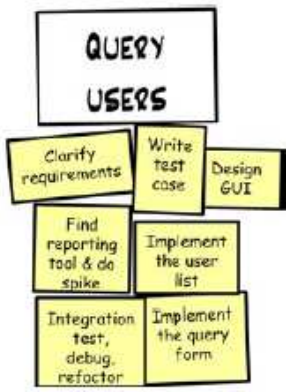
Beispiel einer Aufschlüsselung einer Story in kleinere Storys:



Nutzerverwaltung => Nutzer hinzufügen / bearbeiten | Nutzer suchen

[10] engl.: Task (Anm. der Übersetzerin)

Beispiel einer Aufschlüsselung einer Story in Aufgaben:



Nutzer suchen: Anforderungen klären | Testfall schreiben | GUI designen | Berichtstool finden | Nutzerliste implementieren | Integrationstest, Debuggen, Refaktorieren | Suchformular implementieren

Hier einige interessante Beobachtungen:

- Neue Scrumteams sträuben sich, Zeit damit zu verbringen, einen Haufen Storys wie hier beschrieben im Voraus in Aufgaben herunterzubrechen. Einige haben das Gefühl, dass dies ein wasserfall-artiger Ansatz ist.
- Um ein klares Verständnis der Storys zu bekommen, ist es genauso einfach, diese Aufteilung im Voraus zu machen, wie dies später zu tun.
- Diese Art des Aufteilens offenbart häufig zusätzliche Arbeit, was dazu führt, dass die Zeitschätzung hoch geht und ergibt dadurch einen realistischeren Sprintplan.
- Diese Art des Aufteilens im Voraus macht die täglichen Scrumtreffen merklich effizienter (s. Seite 59 "Wie wir den Daily Scrum gestalten").
- Sogar wenn das Aufteilen vorläufig ist und sich ändern wird, sobald die Arbeit losgeht, treffen die obigen Vorteile noch zu.

Also versuchen wir, das Sprintplanungszeitfenster groß genug zu machen, damit das alles passt, aber wenn die Zeit knapp wird, lassen wir was rausfallen (siehe Abschnitt unten "Wo wir die Linie ziehen").

> Aufgabenaufteilung ist eine große Chance, Abhängigkeiten zu identifizieren – wie "wir werden Zugang zu den Produktionslogs brauchen" oder "wir werden Hilfe von Jim aus der Personalabteilung brauchen" – und Möglichkeiten herauszufinden, mit diesen Abhängigkeiten umzugehen. Vielleicht Jim anrufen und sehen, ob er für uns Zeit während des Sprints aufbringen kann. Je eher Du eine Abhängigkeit entdeckst, desto weniger wahrscheinlich ist, dass sie den Sprint sprengt!

Hinweis: Wir praktizieren TDD, was genaugenommen bedeutet, dass die erste Aufgabe für fast jede Story "schreibe einen fehleraufdeckenden Test" lautet und die letzte Aufgabe "refaktoriere" (= verbessere die Code-Lesbarkeit und entferne Doppelungen).

Zeit und Ort für das Daily Scrum festlegen

Ein regelmäßig vergessenes Ergebnis des Sprintplanungstreffens ist "eine definierte Zeit und ein definierter Ort für das Daily Scrum[11]". Ohne das hat Dein Sprint denkbar schlechte Startbedingungen. Das erste Daily Scrum ist das entscheidende Kick-Off, bei dem jeder entscheidet, wo mit der Arbeit zu beginnen ist.

Ich bevorzuge Treffen am Morgen. Aber ich muss zugeben, wir haben bisher nicht versucht, Daily Scrums nachmittags oder mittags abzuhalten.

> Jetzt hab ich's ausprobiert. Funktioniert prima. Lass das Team entscheiden. Wenn Du unsicher bist, experimentiere. Die meisten Teams bevorzugen allerdings den Vormittag.

[11] Tägliche Stehungen

Nachteile von Nachmittags-Scrums: Wenn Du früh zur Arbeit kommst, musst Du versuchen, Dich zu erinnern, was Du den Leuten gestern darüber erzählt hast, was Du heute tun wolltest.

Nachteile von Vormittags-Scrums: Wenn Du früh zur Arbeit kommst, musst Du versuchen, Dich zu erinnern, was Du gestern getan hast, um heute davon zu berichten.

Meine Meinung ist der erste Nachteil schlimmer, da das Wichtigste ist, was Du *tun wirst*, nicht das, was Du *getan hast*.

Unsere Standardprozedur ist, die früheste Zeit zu nehmen, bei der niemand aus dem Team aufstöhnt. Normalerweise 9:00, 9:30 oder 10:00 Uhr. Das Wichtigste ist, dass es eine Zeit ist, die jeder aus dem Team aus ganzem Herzen akzeptieren kann.

Wo wir die Grenze ziehen

OK, also, die Zeit wird knapp. Von all' dem Zeugs, das wir während der Sprintplanung erledigen wollen, was können wir rausschmeißen, wenn die Zeit knapp wird?

Naja, ich verwende die folgende Prioritätenliste:

Priorität 1: Ein Sprintziel und ein Datum für die Demo. Das ist das Allermindeste, was Du bauchst, um einen Sprint zu starten. Das Team hat ein Ziel, einen Zieltermin und sie können direkt aus dem Produkt Backlog heraus arbeiten. Das ist dürftig, ja, und Du solltest ernsthaft in Erwägung ziehen, ein neues Sprintplanungstreffen für morgen anzusetzen, aber wenn Du wirklich den Sprint zum Starten bringen musst, dann reicht dies wahrscheinlich erst einmal. Obwohl, um ehrlich zu sein, ich habe niemals einen Sprint mit so wenig Information begonnen.

Priorität 2: Liste der Storys, die das Team für diesen Sprint angenommen hat.

Priorität 3: "Schätzung" eingetragen für jede Story im Sprint.

Priorität 4: "How to Demo" eingetragen für jede Story im Sprint.

Priorität 5: Umsetzungsgeschwindigkeit und Ressourcenberechnungen, als Realitätscheck für Deinen Sprintplan. Inklusive einer Liste von Teammitgliedern und ihren Zusagen (sonst kannst Du die Umsetzungsgeschwindigkeit nicht berechnen).

> Halte es einfach und auf einem groben Detaillierungsgrad, nimm Dir dafür maximal fünf Minuten Zeit. Frage: "Aus Personalsicht, ist bei diesem Sprint irgend etwas *bedeutend* anders als bei den vergangenen Sprints?" Falls nicht, setze Yesterday's Weather ein. Falls ja, nimm entsprechende Anpassungen vor.

Priorität 6: Festgelegte Zeit und festgelegter Ort für das Daily Scrum. Das zu entscheiden, benötigt nur einen Moment, aber wenn Dir die Zeit wegrennt, kann der Scrum Master das einfach nach dem Treffen entscheiden und dann an alle per Email rumschicken.

Priorität 7: Auf Aufgaben aufgeteilte Storys. Dieses Herunterbrechen kann statt dessen täglich erfolgen, in Verbindung mit den Daily Scrums, aber das wird den Fluss im Sprint etwas stören.

Technische Storys

Hier kommt ein komplexer Punkt: Technische Storys. Oder nicht-funktionale Themen oder wie immer Du das nennen möchtest.

> Ich kann's nicht lassen loszukichern, wann immer jemand über "nicht-funktionale Anforderungen" spricht. Klingt zu sehr nach "Dinge, die nicht funktionieren sollten". :o)

Ich beziehe mich auf Sachen, die getan werden müssen, aber nicht auslieferungsfähig sind, nicht direkt zu irgendeiner speziellen Story gehören und keinen direkten Wert für den Product Owner haben.

Wir nennen sie "Technische Storys".

Zum Beispiel:

- **Continuous-Build-Server installieren**
  Warum es gemacht werden muss: Weil es immensen Aufwand an Zeit für die Entwickler spart und das Risiko von Urknall-großen Integrationsproblemen am Ende einer Iteration reduziert.
- **Eine Systemdesignübersicht schreiben**
  Warum es gemacht werden muss: Weil Entwickler immer wieder das Gesamtdesign vergessen und dadurch inkonsistenten Code schreiben. Man braucht ein "Big Picture"-Dokument, um jeden designmäßig auf dem gleichen Stand zu halten.
- **DAO-Layer refaktorieren**
  Warum es gemacht werden muss: Weil der DAO-Layer wirklich unordentlich geworden ist und durch Konfusion und unnötige Bugs Zeit eines jeden Einzelnen kostet. Den Code aufzuräumen, wird für jeden Einzelnen Zeit sparen und die Robustheit des Systems erhöhen.
- **Upgrade für Jira (Bug-Tracker) machen**
  Warum es gemacht werden muss: Die derzeitige Version ist zu fehlerhaft und langsam. Ein Upgrade wird jedem Einzelnen Zeit sparen.

Sind dies Storys im üblichen Sinne? Oder sind es Aufgaben, die nicht einer bestimmten Story zugeordnet sind? Wer priorisiert sie? Sollte der Product Owner bei diesen Sachen einbezogen werden?

Wir haben viel mit verschiedenen Arten herumexperimentiert, um diese technischen Storys zu handhaben. Wir versuchten, sie als Erste-Klasse-Storys zu behandeln, einfach wie alle andern Storys auch. Das war nicht gut; wenn der Product Owner das Produkt Backlog priorisierte, war es, wie Äpfel und Birnen zu vergleichen. In der Tat wurde den technischen Storys aus offensichtlichen Gründen eine niedrige Priorität gegeben, mit einer Motivation wie "Ja, Leute, ich bin sicher, ein Continuous-Build-Server ist wichtig und alles, aber lasst uns zuerst ein paar Einnahmen-steigernde Features einbauen, ja? Dann könnt Ihr später Euer Technik-Bonbon einbauen, OK?"

In einigen Fällen liegt der Product Owner richtig, aber oft auch nicht. Wir haben daraus geschlossen, dass der Product Owner nicht immer qualifiziert ist, diese Abstimmung durchzuführen. So, hier ist das, was wir tun:

1) Wir versuchen, technische Storys zu vermeiden. Wir suchen gewissenhaft nach einem Weg, eine technische Story in eine normale Story mit messbarem Geschäftswert zu transformieren. Auf diese Weise hat der Product Owner eine reellere Chance, korrekte Abstimmungen durchzuführen.
2) Wenn wir eine technische Story nicht in eine normale Story transformieren können, prüfen wir, ob die Arbeit als Aufgabe innerhalb einer Story erledigt werden kann. Zum Beispiel könnte "DAO-Layer refaktorieren" eine Aufgabe innerhalb der Story "User bearbeiten" werden, da diese den DAO-Layer umfasst.
3) Wenn beide vorigen Punkte scheitern, definieren wir es als technische Story und führen eine separate Liste solcher Storys. Wir lassen den Product Owner diese Liste sehen, aber nicht bearbeiten. Wir nutzen den "Fokusfaktor"- und "Geschätzte Umsetzungsgeschwindigkeit"-Parameter, um mit dem Product Owner zu verhandeln und reservieren einige Zeit im Sprint, um technische Storys zu implementieren.

Ich empfinde dieses Vorgehen für technische Storys immer noch als großartiges Pattern und benutze es oft. Kleinere technische Storys werden einfach in der täglichen Arbeit eingebettet, während größere Storys aufgeschrieben und in ein technisches Backlog geschrieben werden, sichtbar für den Product Owner, aber vom Team geführt. Das Team und der Product Owner einigen sich auf eine Richtschnur wie "10 bis 20 % unserer Zeit widmen wir technischen Storys". Es bedarf keiner ausgefeilten Trackingschemata wie Fokusfaktor oder Stundenzettel, nutze einfach das Bauchgefühl. Frage in der Retro, "Ungefähr wieviel unserer Sprintkapazität haben wir mit technischen Storys verbraucht und fühlte sich das richtig an?"

Ein diesem Weg ähnlicher Dialog lief während eines unserer Sprintplanungstreffen ab:

Team: Wir haben einiges internes Technikzeugs, das gemacht werden muss. Wir würden das gern mit 10 % unserer Zeit budgetieren, d. h. den Fokusfaktor von 75 % auf 65 % reduzieren. Ist das OK?

Product Owner: Hölle, nein! Wir haben keine Zeit!

Team: Naja, sieh Dir den letzten Sprint an. (Alle Köpfe drehen sich zur Kritzelei zur Umsetzungsgeschwindigkeit auf dem Whiteboard.) Unsere geschätzte Umsetzungsgeschwindigkeit war 80, und unsere tatsächliche Umsetzungsgeschwindigkeit war 30, richtig?

Product Owner: Genau! Also haben wir keine Zeit, internes Technikzeugs zu machen! Wir brauchen neue Features!

Team: Naja, der *Grund*, warum unsere Umsetzungsgeschwindigkeit so schlecht war, war der, dass wir soviel Zeit darauf verwendet haben, zu versuchen, konsistente Releases für den Test zusammenzusetzen.

Product Owner: Ja, und?

Team: Naja, unsere Umsetzungsgeschwindigkeit wird wahrscheinlich so schlecht bleiben, wenn wir nichts ändern.

Product Owner: Ja, und?

Team: Deswegen schlagen wir vor, dass wir ungefähr 10 % von diesem Sprint herausnehmen, um einen Continuous-Build-Server und andere ähnliche Sachen aufzusetzen, die diese Schmerzen bei der Integration abstellen. Dies wird unsere Sprint-Umsetzungsgeschwindigkeit wahrscheinlich *um mindestens* 20 % erhöhen, für jeden folgenden Sprint, für immer!

Product Owner: Oh, echt? Warum haben wir das dann nicht im letzten Sprint schon gemacht?!

Team: Äh, weil Du es uns nicht hast machen lassen…

Product Owner: Oh, ähm, naja, gut – klingt wie eine gute Idee, dass dann jetzt zu machen!

Natürlich ist die andere Möglichkeit, den Product Owner einfach außen vor zu lassen, statt ihn auf dem Laufenden zu halten oder ihm einen nicht-verhandelbaren Fokusfaktor zu nennen. Aber es gibt keine Entschuldigung, nicht zuerst zu *versuchen*, einen Konsens zu erreichen.

Wenn der Product Owner ein kompetenter und vernünftiger Zeitgenosse ist (und wir hatten da Glück), schlage ich vor, ihn so informiert wie möglich zu halten und ihn die Gesamtprioritäten setzen zu lassen. Transparenz ist einer der Kernwerte von Scrum, richtig?

> Wenn Du keine offenen Gespräche mit dem Product Owner über Dinge wie technische Storys, Qualität und technische Schulden führen kannst, dann hast Du ein größeres Problem, mit dem man sich wirklich befassen muss! Ein Symptom dafür ist, wenn Du Dich dabei erwischst, bewusst Informationen vor dem Product Owner zu verstecken. Du musst den Product Owner nicht in jedes Gespräch einbeziehen, aber die Beziehung sollte wirklich auf Vertrauen und Respekt basieren; ohne das wirst Du schwerlich Erfolg haben mit dem, was Du baust.

Bug-Tracking-System vs. Produkt-Backlog

Das ist eine verzwickte Angelegenheit. Excel ist ein großartiges Format für das Produkt-Backlog. Aber Du brauchst trotzdem ein Bug-Tracking-System und Excel ist dafür wahrscheinlich nicht geeignet. Wir benutzen Jira.

Also, wie bringen wir die Themen aus Jira in das Sprintplanungstreffen? Ich denke, es geht nicht, sie einfach zu ignorieren und sich nur auf die Storys zu konzentrieren.

Wir haben verschiedene Strategien ausprobiert:

1) Der Product Owner druckt die am höchsten priorisierten Jira-Einträge aus, bringt sie zum Sprintplanungstreffen mit und hängt sie zusammen mit den andern Storys an die Wand (hierdurch implizit die Priorität dieser Einträge gegenüber der Priorität der anderen Storys spezifizierend).
2) Der Product Owner erzeugt Storys, die sich auf Jira-Einträge beziehen. Zum Beispiel: "Die kritischsten Back-Office-Berichtsbugs fixen: Jira-124, Jira-126 und Jira-180."
3) Fehlerbehebung wird als außerhalb des Sprints liegend betrachtet, d. h. das Team hält den Fokusfaktor niedrig genug (zum Beispiel 50 %), um sicherzustellen, dass sie genug Zeit haben, um Fehler zu fixen. Dann wird einfach angenommen, dass das Team einen bestimmten Zeitaufwand in jedem Sprint für das Fixen Jira-dokumentierter Fehler treibt.
4) Pack das Produkt-Backlog in Jira (d. h. wirf Excel weg). Behandle Fehler einfach wie jede andere Story.

Wir haben leider nicht herausgefunden, welche Strategie für uns die beste ist; tatsächlich variiert sie von Team zu Team und Sprint zu Sprint. Obwohl ich dazu tendiere, mich aus dem Fenster zu lehnen in Richtung der ersten Strategie. Sie ist nett und einfach.

> Acht Jahre später kann ich nur zustimmen. Es gibt keine einzelne Best Practice; jede obige Strategie kann fein funktionieren, abhängig vom Kontext. Experimentiere, bis Du weißt, was für Euch am besten funktioniert.

Das Sprintplanungstreffen ist schließlich vorbei!

Wow, ich hätte nie gedacht, dass dieses Kapitel über Sprintplanungstreffen so lang werden würde! Ich vermute, das spiegelt meine Meinung wider, dass das Sprintplanungstreffen die wichtigste Sache ist, die Du in Scrum tust. Investiere viel Aufwand, das richtig zu machen, und der Rest wird viel leichter sein.

> Oder umgekehrt – mach die andern Sachen richtig und das Sprintplanungstreffen ist ein Kinderspiel. :o)

Das Sprintplanungstreffen ist erfolgreich, wenn alle (Teammitglieder und der Product Owner) das Treffen mit einem Lächeln verlassen und am nächsten Morgen mit einem Lächeln aufwachen und ihr erstes Daily Scrum mit einem Lächeln durchführen.

Dann, klar, können später immer noch alle möglichen Dinge schrecklich falsch laufen, aber Du kannst wenigstens nicht dem Sprintplan dafür die Schuld geben. :o)

TEIL FÜNF

Wie wir Sprintinformationen kommunizieren

Es ist wichtig, das ganze Unternehmen darüber zu informieren, was läuft. Sonst werden sich die Leute beschweren oder, schlimmer noch, falsche Annahmen darüber treffen, was vor sich geht.

Wir setzen dafür eine "Sprint-Info-Seite" ein.

Jackass–Team, Sprint 15

Sprintziel
- Beta-Release!

Sprint-Backlog (Schätzungen in Klammern)
- Anzahlung (3)
- Migrationstool (8)
- Login im Backoffice (5)
- Nutzerverwaltung im Backoffice (5)

Geschätzte Umsetzungsgeschwindigkeit: 21

Zeitplan
- Sprintdauer: 2006-11-06 bis 2006-11-24
- Daily Scrum: 9:30 bis 9:45 im Teamraum
- Sprintdemo: 2006-11-24, 13:00 in der Cafeteria

Team
- Jim
- Erica (Scrum Master)
- Tom (75 %)
- Eva
- John

Manchmal fügen wir auch eine Info darüber ein, wie jede Story demonstriert wird.

Möglichst bald nach dem Sprintplanungstreffen erstellt der Scrum Master diese Seite, stellt sie ins Wiki und schickt SPAM ans ganze Unternehmen.

```
Betreff: Jackass Sprint 15 begonnen

Hi alle!
Das Jackass-Team hat jetzt mit Sprint 15 angefangen.
Unser Ziel ist, am 24. Nov ein Beta-Release zu de-
monstrieren.

Details siehe Sprint-Info-Seite:
http://wiki.mycompany.com/jackass/sprint15
```

Wir haben in unserem Wiki auch eine Dashboard-Seite, die auf alle aktuell laufenden Sprints verweist.

Gemeinsames Dashboard

Laufende Sprints
- Team X Sprint 15
- Team Y Sprint 12
- Team Z Sprint 1

Zusätzlich druckt der Scrum Master die Sprint-Info-Seite aus und hängt sie außerhalb seines Teamraums an die Wand. So kann jeder, der vorbei läuft, sich die Sprint-Info-Seite ansehen und herausfinden, was das Team gerade tut. Da sie Zeit und Ort des Daily Scrum und die Sprint-Demo enthält, weiß er auch, wo er hingehen müsste, um mehr herauszufinden.

Wenn sich das Sprintende nähert, erinnert der Scrum Master alle an die anstehende Demo.

> **Betreff: Jackass Sprint-Demo morgen um 13:00 in der Cafeteria.**
>
> Hi alle!
> Ihr seid willkommen, an unserm Sprintdemo morgen (Freitag) um 13:00 in der Cafeteria teilzunehmen.
>
> Details siehe Sprint-Info-Seite:
> http://wiki.mycompany.com/jackass/sprint15

Wenn all dies gegeben ist, hat niemand wirklich eine Entschuldigung, *nicht* zu wissen, was läuft.

He, was für eine großartige Idee! Ich sollte das öfter machen!

TEIL SECHS

Wie wir Sprint-Backlogs führen

Soweit gekommen? Uff, das war ganze Arbeit.

Also, jetzt, da wir das Sprintplanungstreffen abgehandelt und der Welt von unserm glänzenden neuen Sprint erzählt haben, ist es Zeit für den Scrum Master, ein Sprint-Backlog zu erstellen. Dies muss *nach* dem Sprintplanungstreffen, aber *vor* dem ersten Daily Scrum gemacht werden.

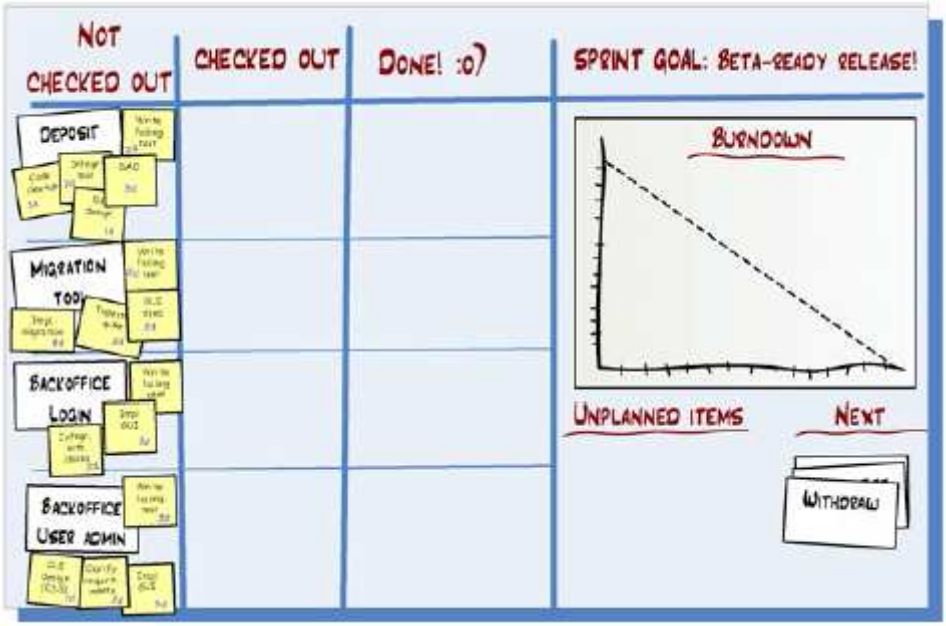
Sprint-Backlog-Format

Wir haben mit verschiedenen Formaten für das Sprint-Backlog experimentiert, inklusive Jira, Excel und einer physischen Aufgabentafel an der Wand. Am Anfang benutzten wir meist Excel; es gibt viele öffentlich zugänglichen Excel-Templates für Sprint-Backlogs, inklusive sich automatisch generierenden Burndown-Diagrammen und solchen Sachen. Ich könnte eine Menge darüber erzählen, wie wir unsere Excel-basierten Sprint-Backlogs verfeinert haben. Aber ich werde es nicht tun. Ich werde nicht mal ein Beispiel hier einschieben.

Statt dessen werde ich detailliert beschreiben, was wir als effektivstes Format für das Sprint-Backlog entdeckt haben – eine wand-basierte Aufgabentafel!

> Genau, wand-basierte Aufgabentafeln (a.k.a. Scrum-Boards) sind immer mein Werkzeug erster Wahl! Überraschend mächtig. Für ein verteiltes Team benutze ein Tool, das eine Scrum-Board-Sicht anbietet (fast alle Tools haben das), und packe es an jedem Standort auf einen großen Bildschirm. Beim Daily Scrum stehen alle am Bildschirm an der Wand und reden per Skype (oder ähnlichem) miteinander.

Finde eine große Wand, die ungenutzt ist oder an der nutzloses Zeug wie das Firmenlogo, alte Diagramme oder hässliche Bilder hängen. Räum das von der Wand (frag nur um Erlaubnis, wenn Du musst). Kleb ein großes, großes Blatt Papier an (mindestens 2 x 2 Meter oder 3 x 2 Meter für ein großes Team). Dann tu dies:



Nicht ausgecheckt | Ausgecheckt | Fertig :o) | Sprintziel: Beta-Release fertig
Burndown-Diagramm
Ungeplante Einträge, Nächstes…

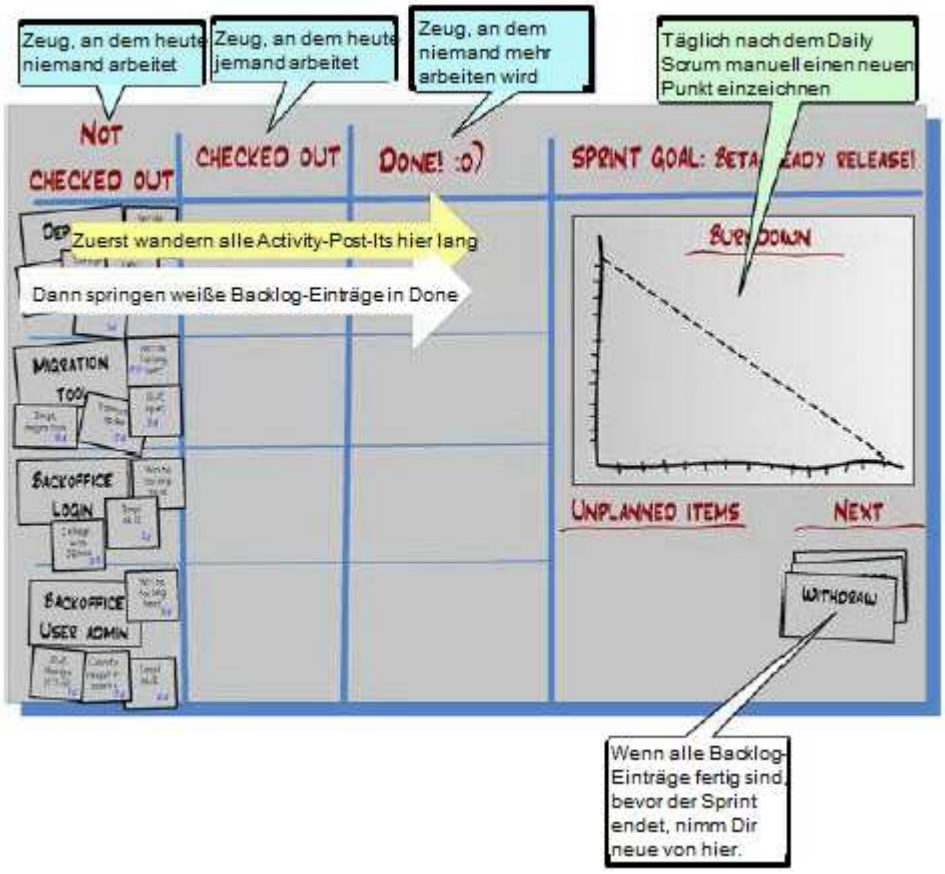
Du könntest natürlich ein Whiteboard benutzen. Aber das ist ein bisschen Verschwendung. Falls möglich, spare Dir die Whiteboards für Designskizzen und nutze Nicht-Whiteboards als Aufgabentafel.

> Oder besser, pflastere Deine Wände mit Whiteboards. Eine angemessene Investition!

Hinweis: Wenn Du Haftnotizen für Aufgaben nutzt, vergiss nicht, sie mit richtigem Klebeband anzukleben, oder Du wirst eines Tages alle Haftnotizen in einem hübschen Stapel auf dem Boden vorfinden.

> Oder kauf einfach Superhaftnotizzettel, etwas teurer, aber sie fallen nicht herunter! (Nein, ich werde nicht gesponsort, wirklich.)

Wie Aufgabentafeln funktionieren



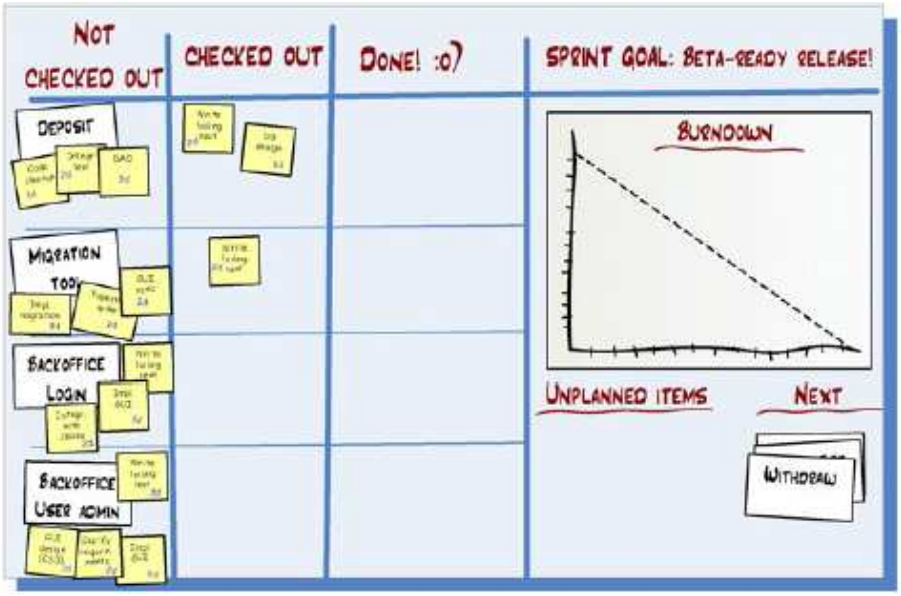
Nicht ausgecheckt | Ausgecheckt | Fertig :o) | Sprintziel: Beta-Release fertig
Burndown-Diagramm
Ungeplante Einträge, Nächstes…

Du könntest natürlich alle Arten von zusätzlichen Spalten hinzufügen. "Wartet auf Integrationstest" zum Beispiel. Oder "storniert". Aber, bevor Du die Sache verkomplizierst, denk gründlich nach. Ist dieser Zusatz wirklich, *wirklich* nötig?

Ich habe die Erfahrung gemacht, dass Schlichtheit bei solchen Dingen extrem wertvoll ist, also füge ich nur dann zusätzliche Komplikationen hinzu, wenn der Preis, es *nicht* zu tun, zu hoch ist.

Beispiel 1: Nach dem ersten Daily Scrum

Nach dem ersten Daily Scrum könnte die Aufgabentafel wie folgt aussehen:



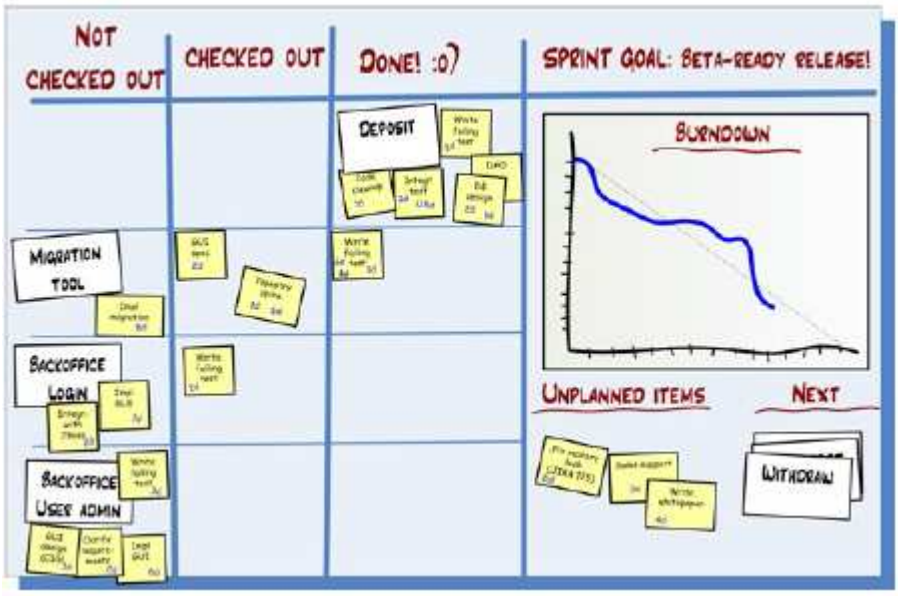
Wie Du siehst, sind drei Aufgaben ausgecheckt, d. h. das Team wird heute an diesen Einträgen arbeiten.

Manchmal, bei größeren Teams, steckt eine Aufgabe in der "Ausgecheckt"-Spalte fest, weil niemand sich daran erinnert, wer daran gearbeitet hatte. Wenn dies oft im Team passiert, führen die Teammitglieder normalerweise Spielregeln ein wie die, jede ausgecheckte Aufgabe mit dem Namen der Person zu kennzeichnen, die sie ausgecheckt hat.

> Fast alle Teams benutzen heutzutage Avatare. Jedes Teammitglied nimmt sich seinen Avatar (eine South-Park-Figur oder sowas), druckt sie aus und klebt sie auf Magnete. Großartig, um zu sehen, wer an was arbeitet. Außerdem, wenn jede Person nur, sagen wir, zwei Magnete hat, begrenzt das indirekt die Sachen, die gleichzeitig in Arbeit[12] sind und das Multitasking. "WTF, ich habe keine Avatare mehr!" Jawoll, also hör auf mit Anfangen und fang an mit Aufgabenbeendigung!

Beispiel 2: Nach ein paar Tagen

Einige Tage später könnte die Aufgabentafel ungefähr so aussehen:



Wie Du siehst, wir haben die "Deposit"-Story fertiggestellt (d. h. sie ist im Sourcecode-Repository eingecheckt, getestet, überarbeitet etc.). Das Migrationstool ist teilweise fertig, das Backoffice-Login ist angefangen und die Backoffice-Nutzerverwaltung ("Backoffice User Admin") ist noch nicht begonnen.

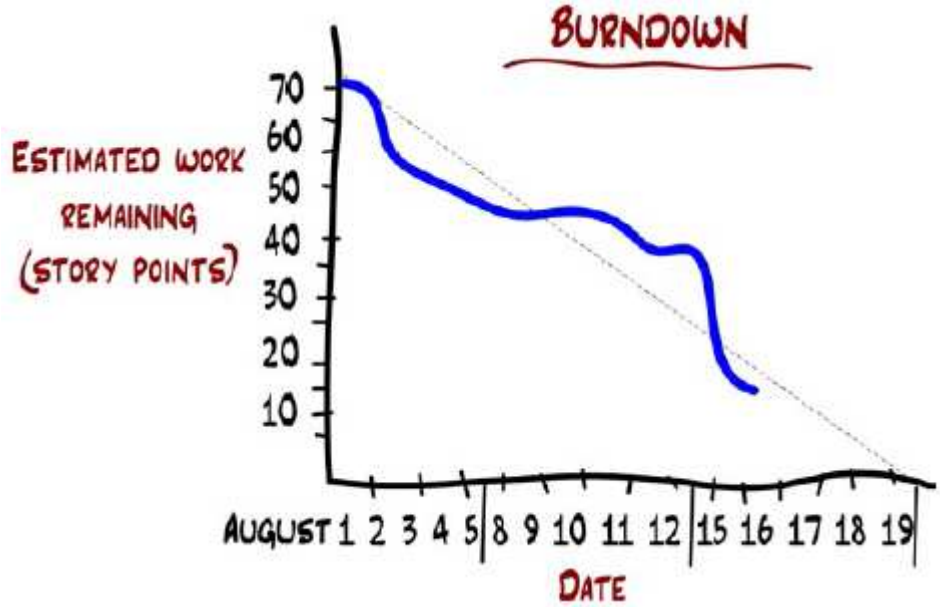
Wir hatten drei ungeplante Einträge, wie Du unten rechts sehen kannst. Es ist nützlich, sich das in Erinnerung zu rufen, wenn Du die Sprint-Retrospektive machst.

Hier ist ein Beispiel eines echten Sprint-Backlogs zu einem späten Zeitpunkt im Sprint. Es sieht ziemlich unordentlich aus, während der Sprint fortschreitet, aber das ist OK, da dies nur von kurzer Dauer ist. In jedem neuen Sprint stellen wir ein frisches, sauberes, neues Sprint-Backlog zusammen.

[12] Work in progress = WIP

Wie Burndown-Diagramme funktionieren

Lass uns in das Burndown-Diagramm reinzoomen:



Geschätzte verbleibende Arbeit (in Story-Punkten) nach Datum

Dieses Diagramm zeigt, dass:

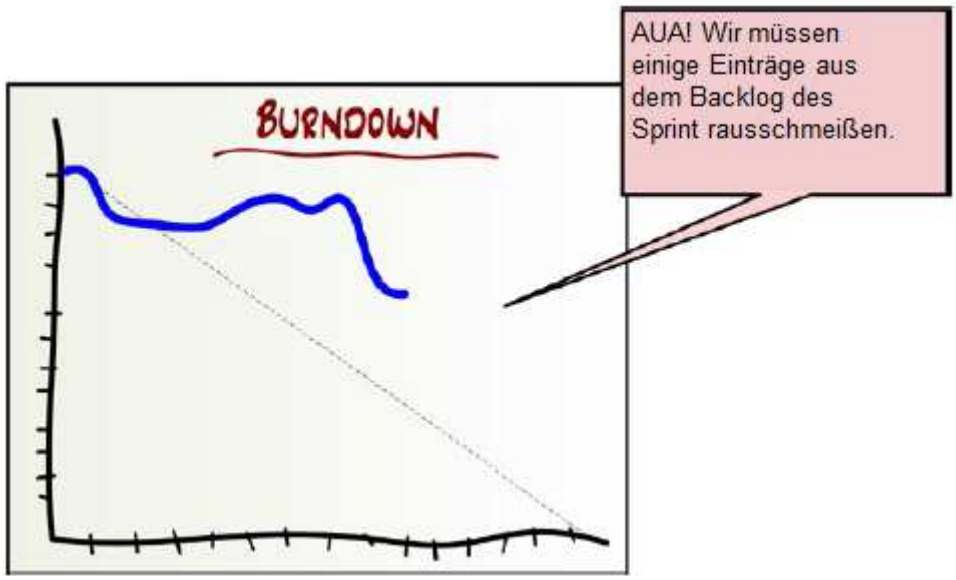
- Am ersten Tag des Sprints, am 1. August, hat das Team geschätzt, dass es ungefähr 70 Story-Punkte an verbleibender Arbeit gab. Dies war die gültige *geschätzte Umsetzungsgeschwindigkeit* des ganzen Sprints.
- Am 16. August schätzte das Team, dass es ungefähr 15 Story-Punkte an verbleibender Arbeit gab. Die gestrichelte Trendlinie zeigt, dass sie auf dem richtigen Weg waren, d. h. bei diesem Tempo werden sie alles am Ende des Sprints fertiggestellt haben.

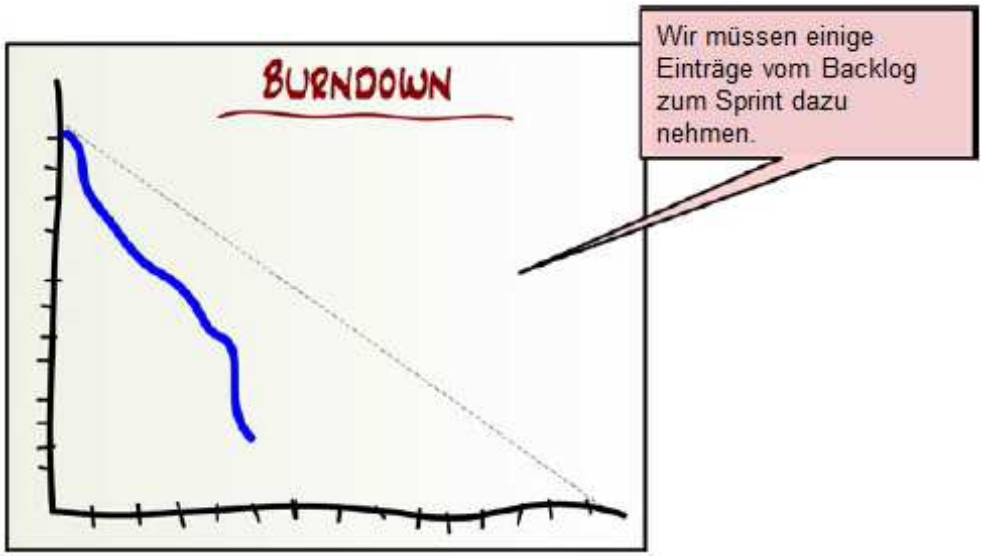
Wir überspringen Wochenenden auf der x-Achse, da kaum Arbeit an Wochenenden erledigt wird. Wir haben früher Wochenende mit reingenommen, aber das machte das Burndown-Diagramm etwas unübersichtlich, da es sich über die Wochenenden "verflachte", was nach einem Warnsignal aussah.

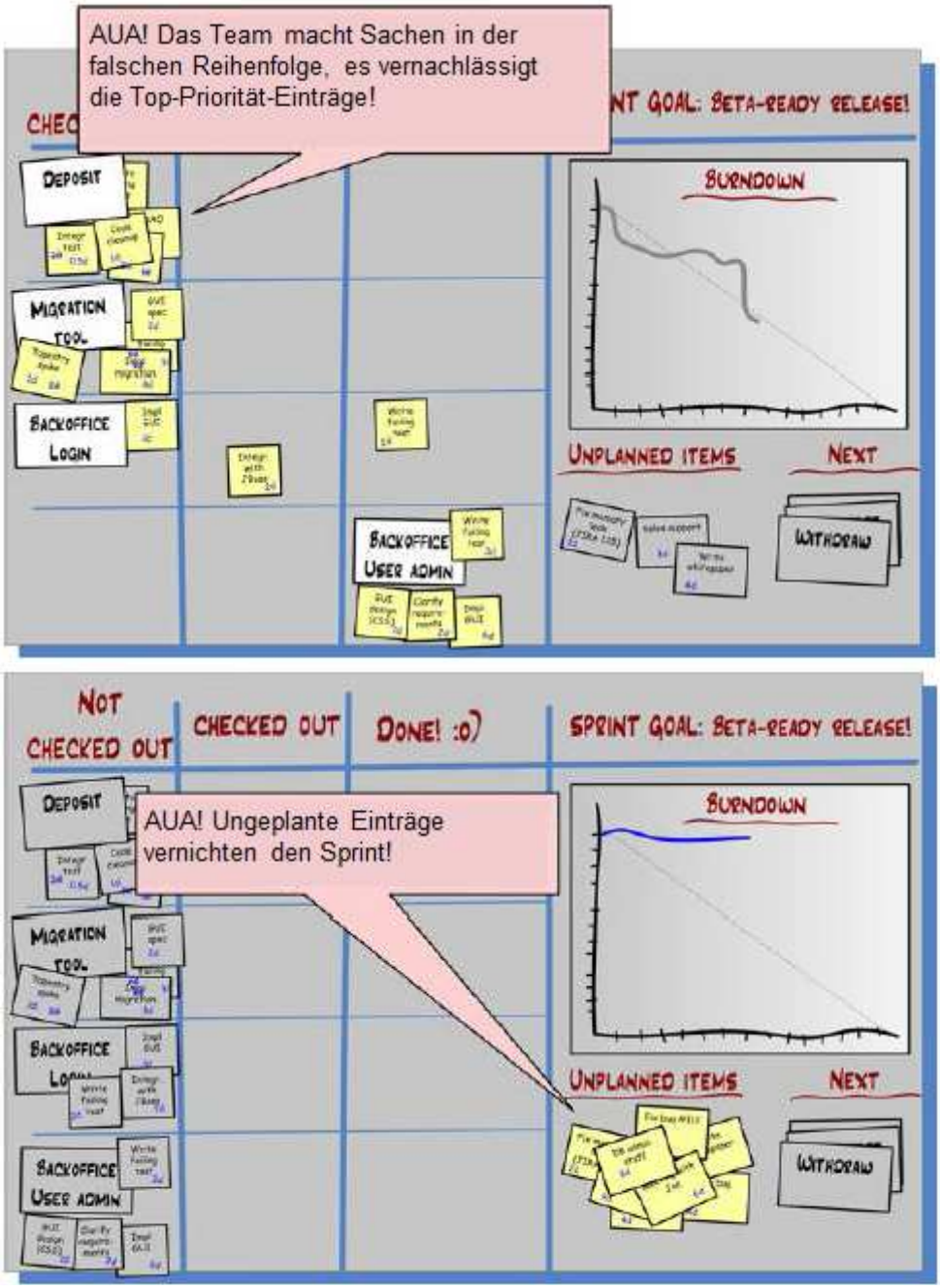
> Scrum war ursprünglich für Teams designt, die Ein-Monats-Sprints machten und Excel nutzten, um Aufgaben nachzuverfolgen. In dem Fall ist ein Burndown-Diagramm eine wirklich nützliche visuelle Zusammenfassung davon, wo wir stehen. Heute allerdings machen die meisten Teams kürzere Sprints und haben visuell ausgereifte Scrumboards. Also lassen sie mehr und mehr Burndown-Diagramme komplett weg, weil das Scrumboard ihnen auf einen Blick die Info zeigt, die sie brauchen. Versuch mal, das Burndown-Diagramm wegzulassen und warte ab, ob Ihr es vermisst!

Aufgabentafelwarnsignale

Ein kurzer Blick auf die Aufgabentafel sollte jedem einen Anhaltspunkt dafür geben, wie gut der Sprint vorankommt. Der Scrum Master ist dafür verantwortlich sicherzustellen, dass das Team auf Warnsignale wie die folgenden reagiert:







He, was ist mit Nachverfolgbarkeit?!

Die beste Nachverfolgbarkeit[13], die ich anbieten kann, ist das Modell, bei dem man täglich ein Foto der Aufgabentafel macht. Wenn Du nachverfolgen musst. Ich mache das manchmal, aber finde nie den Bedarf, diese Fotos wieder hervorzukramen.

Wenn Nachverfolgbarkeit für Dich sehr wichtig ist, dann ist die Aufgabentafellösung vielleicht nichts für Dich.

Aber ich schlage vor, dass Du ernsthaft ausprobierst, den greifbaren Wert von detaillierter Sprintnachverfolgbarkeit für Dich einzuschätzen. Sobald der Sprint fertig ist und der laufende Code ausgeliefert wurde und die Dokumentation eingecheckt ist, interessiert sich dann noch jemand tatsächlich dafür, wie viele Storys an Tag 5 im Sprint fertiggestellt wurden? Interessiert sich irgendwer wirklich noch dafür, wie die Zeitschätzung für "schreibe einen fehleraufdeckenden Test für die Anzahlung" war?

[13] Traceability

Ich finde immer noch, dass Nachverfolgbarkeit überbewertet wird. Einige Werkzeuge bieten diese Art von Daten, aber die Leute nutzen das im Prinzip nie. Dein Codeversionskontrollsystem wird Dir das meiste von dem geben, was Du brauchst ("WTF? Wer hat diese Änderung gemacht?!"). Füge ein paar simple Vereinbarungen hinzu, wie die Story-ID beim Einchecken in den Kommentar zu schreiben, und Du bekommst für die meisten Situationen mehr als genug Nachverfolgbarkeit.

Tage schätzen vs. Stunden schätzen

In den meisten Büchern und Artikeln über Scrum wirst Du lesen, dass Aufgaben in Stunden geschätzt werden, nicht in Tagen. Wir haben das mal gemacht. Unsere allgemeine Formel dafür war: 1 effektiver Personentag = 6 effektive Personenstunden.

Jetzt haben wir damit aufgehört, zumindest in den meisten unserer Teams, und zwar aus folgenden Gründen:

Personenstundenschätzungen waren zu freingranular; dies führt leicht dazu, zu viele kleinteilige Ein-Stunden- oder Zwei-Stunden-Aufgaben hervorzubringen und so Mikromanagement zu fördern.

Es hat sich herausgestellt, dass jeder sowieso eher in Personentagen dachte und einfach mit sechs multipliziert hat, bevor er die Personenstunden aufgeschrieben hat. "Hmmmm, diese Aufgabe dauert wohl ungefähr einen Tag. Oh, ich muss Stunden aufschreiben. Ich werde mal sechs Stunden aufschreiben."

Zwei verschiedene Einheiten zu haben, verursacht Irritationen. "War die Schätzung in Personentagen oder in Personenstunden?"

Also verwenden wir jetzt Personentage als Grundlage für alle Zeitschätzungen (obwohl wir es Story-Punkte nennen). Unser kleinster Wert ist 0,5, d. h. jede Aufgabe, die kleiner als 0,5 ist, wird entweder herausgenommen und mit einer anderen Aufgabe kombiniert oder einfach mit einer 0,5-Schätzung stehen gelassen (es schadet nicht großartig, etwas höher zu schätzen). Hübsch und einfach.

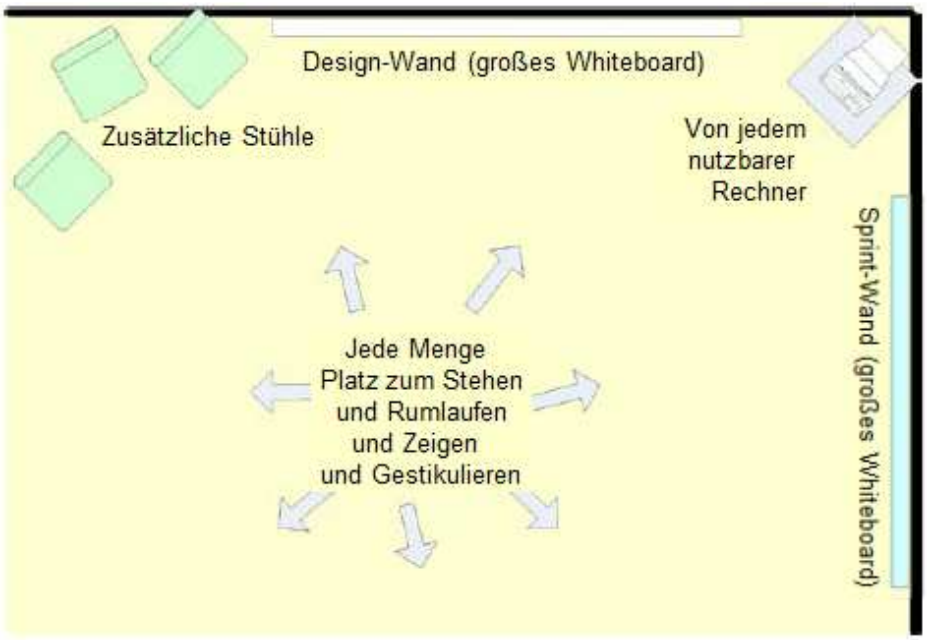
Noch einfacher: Überspring das Schätzen von Aufgaben komplett! Die meisten Teams lernen mit der Zeit, wie sie ihre Arbeit in Aufgaben herunterbrechen können, die ungefähr einen Tag für ein oder zwei Personen entsprechen. Wenn Du das tun kannst, brauchst Du Dich nicht mit der Schätzung der Aufgaben herumzuschlagen, was eine Menge überflüssige Arbeit erspart. Burndown-Diagramme können immer noch eingesetzt werden (wenn es sein muss) – in diesem Fall zähl die Aufgaben einfach anstatt die Stunden aufzuaddieren.

TEIL
SIEBEN

Wie wir den Teamraum gestalten

Die Design-Ecke

Ich habe festgestellt, dass viele der interessantesten und wertvollsten Designdiskussionen spontan vor der Aufgabentafel stattfinden. Aus diesem Grund versuchen wir, diesen Bereich explizit als "Design-Ecke" zu gestalten.



Das ist ziemlich hilfreich. Es gibt keine bessere Möglichkeit, eine Übersicht über das System zu bekommen, als in der Design-Ecke zu stehen und beide Wände anzuschauen, dann kurz einen Blick in den Computer zu werfen und zu versuchen, den neuesten Build des Systems aufzusetzen (wenn Du das Glück hast, dass Ihr kontinuierliche Builds habt, siehe Seite 80 "Wie wir Scrum und XP kombinieren").

Die "Design-Wand" ist einfach ein großes Whiteboard, das die wichtigsten Designskizzen und Ausdrucke der wichtigsten Designdokumentation (Sequenzdiagramme, GUI-Prototypen, Domänenmodelle etc.) enthält.

Oben: Im gerade beschriebenen Bereich läuft gerade ein Daily Scrum. Hmm… das Burndown sieht verdächtig fein und geradlinig aus, oder?

Doch das Team versichert, dass es echt ist. :o)

> Der Evil Coach stellt fingierte Burndown-Zeichenlineale zur Verfügung. Sehr komfortable für hoch disfunktionale Umgebungen. :o)
> http://blog.crisp.se/2013/02/15/evil-coach/fake-burndown-ruler

Lass das Team zusammen sitzen!

Wenn es an die Sitzverteilung und die Schreibtischanordnung geht, gibt es eine Sache, die nicht genug betont werden kann.

Lass das Team zusammen sitzen!

Um ein bisschen klarer zu machen, was ich sagen will:

Lass das Team zusammen sitzen!

> Nach acht Jahren, in denen ich Unternehmen bei Scrum unterstützt habe, möchte ich hinzufügen:
>
> **LASS DAS TEAM ZUSAMMEN SITZEN!**

Leute sträuben sich umzuziehen. Zumindest da, wo ich bisher gearbeitet habe. Sie möchten nicht gezwungen sein, alle ihre Sachen zusammen zu klauben, ihren Computer abzustöpseln, all ihren Müll zu einem andern Schreibtisch zu tragen und alles wieder einzustöpseln. Je kürzer die Strecke, desto größer das Sträuben. "KOMM schon, Boss, was hat das für einen Sinn, schlappe fünf Meter weiter zu rutschen?"

Wenn man effektive Scrum-Teams aufbauen möchte, gibt es jedoch keine Alternative. Bring halt das Team zusammen. Selbst wenn Du jeden einzeln persönlich bedrohen, all seine Ausrüstung herumtragen und seine alten Kaffeeflecken wegwischen musst. Wenn es keine Platz für das Team gibt, mache Platz. Irgendwo. Sogar wenn Du das Team in den Keller schicken musst. Schieb Tische herum, bestech den Leiter der Büroausstattung, mach, was immer nötig ist. Nur bring das Team zusammen.

Sobald Du das Team zusammen gebracht hast, wird sich der Erfolg sofort zeigen. Nach nur einem Sprint wird das Team sich einig sein, dass es eine gute Idee war, zusammen zu ziehen (wo wir gerade von persönlicher Erfahrung sprechen – es sagt nichts darüber aus, ob Dein Team nicht zu störrisch ist, das einzusehen).

Also, was bedeutet "zusammen"? Wie sollten die Tische aufgestellt werden? Naja, ich vertrete bei der optimalen Schreibtischanordnung keine fest vorgegebene Meinung. Und selbst wenn ich das täte, vermute ich, die meisten Teams genießen nicht den Luxus, die Möglichkeit zu haben, im Detail zu entscheiden, wie sie ihre Schreibtische anordnen wollen. Normalerweise gibt es bauliche Einschränkungen – das Nachbarteam, die Toilettentür, der große Automat in der Mitte des Raums, was auch immer.

"Zusammen" bedeutet:

- **Hörweite**: Jede im Team kann mit jedem anderen im Team reden, ohne zu rufen oder seinen Schreibtisch zu verlassen.
- **Sicht**: Jeder im Team kann jede andere im Team sehen. Jeder kann die Aufgabentafel sehen. Nicht notwendigerweise nah genug, um alles zu *lesen*, aber zumindest, um alles zu *sehen*.
- **Abgeschiedenheit**: Wenn Dein ganzes Team plötzlich aufsteht und sich spontan und lebhaft in eine Design-Diskussion stürzt, gibt es niemanden außerhalb des Teams, der nah genug ist, um dadurch gestört zu werden. Und umgekehrt.

"Abgeschiedenheit" bedeutet nicht, dass das Team komplett isoliert sein muss. In einer Umgebung mit Arbeitsnischen könnte es ausreichen, dass Dein Team seinen eigenen Bereich hat und Trennwände, die hoch genug sind, den *Großteil* des Lärms durch Faktoren, die nicht zum Team gehören, herauszufiltern.

Und was ist, wenn Du ein verteiltes Team hast? Naja, dann hast Du Pech. Nutze so viel technische Hilfestellung, wie Du kannst, um den Schaden zu minimieren – Videokonferenzen, Webcams, Desktop-Sharing-Tools etc.

Den Product Owner in Schach halten

Der Product Owner sollte nah genug sein, damit das Team rüber gehen und ihn etwas fragen kann, und so nah, dass er zur Aufgabentafel herüber kommen kann. Aber er sollte nicht beim Team sitzen. Warum? Weil die Wahrscheinlichkeit besteht, dass er sich nicht zurück halten kann, sich in Details einzumischen, und weil das Team nicht gut Gestalt annehmen kann (d. h. einen engen, selbstorganisierten, hyperproduktiven Status erreicht).

Um ehrlich zu sein, dies ist Spekulation. Mir ist tatsächlich noch kein Fall begegnet, in dem der Product Owner beim Team sitzt, also habe ich keinen empirischen Grund zu behaupten, dass das eine schlechte Idee ist. Nur Bauchgefühl und Hörensagen von anderen Scrum Mastern.

> Naja, rate mal. Ich habe mich geirrt! Völlig falsch! Die besten Teams, die ich gesehen habe, haben den Product Owner integriert. Teams leiden sehr, wenn der Product Owner zu weit weg ist, und das ist ein viel größeres Problem als zu nah zu sein. Der Product Owner muss jedoch seine Zeit gut zwischen dem Team und den Stakeholdern ausbalancieren, so dass er nicht ALL' seine Zeit beim Team verbringt. Aber im Allgemeinen, je näher, desto besser.

Manager und Coach in Schach halten

Das aufzuschreiben ist ein bisschen hart für mich, da ich beides war, Manager und Coach…

Es war mein Job, so eng wie möglich mit den Teams zusammen zu arbeiten. Ich setzte Teams zusammen, bewegte mich von einem zum anderen, machte mit Leuten Pairprogramming, coachte Scrum Master, organisierte Sprintplanungstreffen etc. Im Rückblick hielten die meisten Leute dies für eine Gute Sache, da ich einige Erfahrung mit agiler Softwareentwicklung hatte.

Aber andererseits war ich auch (hier spiel Darth-Vader-Musik ein) der Häuptling der Entwicklung, eine funktionale Managementrolle. Was bedeutet, dass wenn ich zu einem Team dazu kam, dieses automatisch weniger selbstorganisiert wurde. "Obacht, der Boss ist da. Er hat wahrscheinlich eine eigene Meinung dazu, was wir tun sollten und wer was tun sollte. Ich werde ihm das Reden überlassen."

Mein Punkt ist der folgende: Wenn Du Scrum-Coach bist (und vielleicht außerdem Manager), beteilige Dich so eng wie möglich. Aber nur für eine begrenzte Dauer. Dann geh weg und lass sich das Team formen und selbst organisieren. Überprüfe das Team ab und zu (nicht zu oft) durch Teilnahme an Sprintdemos und durch einen Blick auf die Aufgabentafel und durch das Reinhorchen in die morgendlichen Scrumtreffen. Wenn Du einen Bereich mit Verbesserungspotenzial entdeckst, nimm den Scrum Master zur Seite und coache ihn. *Nicht* vorm Team. Eine andere gute Idee ist, an Sprint-Retrospektiven teilzunehmen (siehe Seite 66 "Wie wir Sprintretrospektiven machen"), wenn Dein Team Dir genug vertraut, dass Deine Anwesenheit sie nicht dazu veranlasst, keinen Piep mehr zu sagen.

Für gut funktionierende Scrum-Teams stell sicher, dass sie alles bekommen, was sie brauchen, dann geh ihnen höllisch aus dem Weg (außer bei Sprintdemos).

> Der letzte Satz ist immer noch der beste allgemeine Ratschlag, den ich für Manager im agilen Kontext habe. Gute Manager sind ein entscheidender Erfolgsfaktor, aber als Manager solltest Du versuchen, Dich selbst überflüssig zu machen. Du wirst wahrscheinlich nicht erfolgreich damit sein, aber der Versuch an sich wird Dich in diese Richtung tragen. Frage "Was braucht dieses Team, um sich selbst zu managen?" statt "Wie kann ich dieses Team managen?" Sie brauchen Dinge wie Transparenz, ein klares Ziel, ein Spaß bringendes und motivierendes Arbeitsumfeld, eine gute Lüftung für Frischluft und einen Eskalationspfad, falls Hindernisse auftauchen.

TEIL ACHT

Wie wir den Daily Scrum gestalten

Unsere Daily Scrums verlaufen ziemlich streng nach Vorschrift. Sie beginnen absolut pünktlich, jeden Tag am selben Ort. Früher gingen wir in einen anderen Raum, um die Sprintplanung zu machen (das war die Zeit, als wir elektronische Sprint-Backlogs hatten), jetzt machen wir Daily Scrum jedoch im Teambüro direkt vor der Aufgabentafel. Das ist durch nichts zu schlagen.

Normalerweise stehen wir während der Treffen, da dies das Risiko reduziert, 15 Minuten zu überschreiten.

> Daily Scrum ist wirklich wichtig! Es ist das Instrument, durch das die meiste Abstimmung passiert und bei dem im Team wichtige Hindernisse sichtbar werden. Nichtsdestotrotz, wenn sie schlecht durchgeführt werden, können sie wiiiiirklich langweilig sein – ein Haufen Leute, die schwafeln, und niemand hört wirklich zu.

Der Scrum Guide aktualisierte kürzlich die drei Fragen, die man anspricht:

- Was habe ich gestern getan, das unserem Team half, das Sprintziel zu erreichen?
- Was will ich heute tun, was unserem Team hilft, das Sprintziel zu erreichen?
- Sehe ich irgendwelche Hindernisse, die mich oder das Team davon abhalten, das Sprintziel zu erreichen?

> Beachte den Fokus auf das Sprintziel, dem vom Team geteilten Zweck auf oberer Ebene! Wenn Deine Daily Scrums anfangen, sich lasch anzufühlen, versuch es mit diesen Fragen! Oder vielleicht die Version von Dan North: Frage "Was ist das beste 'Heute', das wir haben können?", gefolgt von offener Diskussion. Was immer Du tust, lass die Daily Scrums nicht langweilig werden. Immer weiter experimentieren!

Wir wir die Aufgabentafel aktualisieren

Normalerweise aktualisieren wir die Aufgabentafel während des Daily Scrums. Während jede Person beschreibt, was sie gestern gemacht hat und was sie heute tun wird, verschiebt sie die Haftnotizen auf der Aufgabentafel. Während sie einen (bisher) ungeplanten Eintrag erläutert, hängt sie dafür eine Haftnotiz auf. Während sie ihre Zeitschätzung aktualisiert, schreibt sie die neue Zeitschätzung auf die Haftnotiz und streicht die alte durch. Manchmal macht der Scrum Master die Haftnotiz-Sachen, während die Leute reden.

Fehleraufdeckenden Test schreiben

Einige Teams haben die Regel, dass jede Person die Aufgabentafel *vor* jedem Treffen aktualisieren soll. Das funktioniert auch fein. Entscheide Dich einfach für eine Regel und dann halt' Dich dran.

Viele Teams verschwenden einen übermäßigen Zeitaufwand damit, Zahlen auf Haftnotizen während des Daily Scrum zu aktualisieren. Unnützer Aufwand! Der Zweck des Daily Scrum ist, sich abzustimmen, deswegen finde ich es in der Regel am besten, das Board "in Echtzeit" (d. h. während des Arbeitstages, wenn die Sachen passieren) zu aktualisieren und die Aufgabenschätzung komplett wegzulassen. Auf diese Weise wird das Daily Scrum tatsächlich eher zum *Kommunizieren*, als zum Verwalten genutzt.

Ungeachtet dessen, welches Format Euer Sprint-Backlog hat, versuche, das *ganze Team* daran zu beteiligen, das Sprint-Backlog auf dem aktuellen Stand zu halten. Wir haben einige Sprints ausprobiert, in denen der Scrum Master der einzige ist, der das Sprint-Backlog pflegt und der jeden Tag herumlaufen und die Leute über ihre verbleibende Zeitschätzung befragen muss. Die Nachteile davon sind:

- Der Scrum Master verwendet zuviel Zeit damit, Sachen zu verwalten, statt das Team zu unterstützen und Hindernisse zu beseitigen.
- Teammitglieder sind sich des aktuellen Status' des Sprints nicht bewusst, da das Sprint-Backlog nichts ist, um das sie sich kümmern müssen. Dieser Mangel an Feedback reduziert die Gesamtagilität und den Fokus des Teams.

Wenn das Sprint-Backlog gut gestaltet ist, sollte es genauso einfach sein, dass jedes Teammitglied es selbst aktualisieren kann.

Sofort nach dem Daily-Scrum-Treffen addiert irgendwer all' die Zeitschätzungen (die in der "Done"-Spalte werden natürlich ignoriert) und zeichnet einen neuen Punkt ins Burndown-Diagramm ein.

Mit Zuspätkommern umgehen

Einige Teams haben eine Dose mit Münzen und Scheinen. Wenn Du zu spät kommst, und sei es auch nur eine Minute zu spät, gibst Du einen festgelegten Betrag in die Dose. Es werden keine Fragen gestellt. Du kannst vor dem Treffen anrufen und sagen, dass Du später kommen wirst, bezahlen musst Du musst trotzdem.

Du kommst nur vom Haken, wenn Du eine gute Entschuldigung hast wie einen Arzttermin oder Deine eigene Hochzeit oder sowas.

Das Geld in der Dose wird für Gemeinschaftserlebnisse verwendet. Hamburger kaufen, wenn wir Gaming Nights haben, zum Beispiel. :o)

Das funktioniert gut. Aber es ist nur für Teams notwendig, bei denen Leute oft zu spät kommen. Einige Teams brauchen diese Art von Maßnahme nicht.

> Teams verwenden alle möglichen Arten von Maßnahmen, um sich gegenseitig zur Pünktlichkeit zu erziehen (falls notwendig). Stell einfach sicher, dass das Team etwas für sich selbst findet; dränge dem Team keine Maßnahme von oberhalb oder außerhalb des Teams auf. Und lass es Spaß machen. In einem Team mussten die Zuspätkommer ein albernes Lied singen. Wenn Du das zweite Mal zu spät bist, musst Du den dazu passenden Tanz auch noch machen. :o)

Mit "Ich weiß nicht, was ich heute tun soll" umgehen

Es ist nicht selten, dass jemand sagt, "Gestern hab ich bla bla blu, aber heute hab ich nicht die leiseste Vorstellung davon, was ich tu" (he, das reimt sich am Schluss). Was jetzt?

Sagen wir, Joe und Lisa sind diejenigen, die nicht wissen, was sie heute tun sollen.

Wenn ich Scrum Master bin, mache ich einfach weiter und lass die nächste Person reden, notiere mir aber, welche Leute nichts zu tun haben. Nachdem jeder etwas gesagt hat, gehe ich mit dem gesamten Team die Aufgabentafel durch, von oben nach unten, und checke, dass alles abgestimmt ist, dass jeder weiß, was jeder Eintrag bedeutet, etc. Ich lade die Leute ein, mehr Haftnotizen hinzuzufügen. Dann komme ich zurück zu den Leuten, die nicht wussten, was sie tun sollen: "Jetzt, da wir die Aufgabentafel durchgegangen sind, hast Du eine Idee, was Du tun kannst?" Hoffentlich haben sie die.

Wenn nicht, überlege ich, ob es hier eine Möglichkeit zum Pairprogramming gibt. Sagen wir, Niklas will heute die Backoffice-Nutzer-GUI implementieren. In diesem Fall schlage ich höflich vor, dass vielleicht Joe oder Lisa mit Niklas daran im Pairprogramming arbeiten könnten. Das funktioniert normalerweise.

Und wenn das nicht funktioniert, hier ist der nächste Trick.

Scrum Master: OK, wer möchte uns das Beta-Release demonstrieren? (Angenommen, das war das Sprintziel.)

Team: (Irritiertes Schweigen.)

Scrum Master: Sind wir damit nicht fertig?

Team: Ähmm… nein.

Scrum Master: Oh verdammt. Warum nicht? Was muss noch gemacht werden?

Team: Naja, wir haben noch nicht mal einen Testserver, um das laufen zu lassen, und das Build-Script ist kaputt.

Scrum Master: Aha. (Hängt zwei neue Haftnotizen an die Aufgabentafel.) Joe und Lisa, wie könnt Ihr uns heute helfen?

Joe: Ähmm… ich denke, ich werde versuchen, irgendwo Testserver aufzutreiben.

Lisa: Und ich werde versuchen, das Build-Script zu fixen.

Wenn Du Glück hast, wird tatsächlich jemand das Beta-Release demonstrieren, um das Du gebeten hast. Großartig! Du hast Dein Sprintziel erreicht. Aber was, wenn Du erst in der Mitte vom Sprint bist? Einfach. Gratuliere dem Team dazu, eine gute Leistung erbracht zu haben, schnapp Dir ein oder zwei Storys aus dem Bereich "Nächste" unten rechts an Deiner Aufgabentafel und schiebe sie in die linke Spalte "nicht ausgecheckt". Dann wiederhole den Daily Scrum. Benachrichtige den Product Owner, dass Du dem Sprint einige Einträge hinzugefügt hast.

> Oder nutze die Zeit, einige technische Schulden zu tilgen oder technische Erkundigungen einzuziehen. Doch halte den Product Owner auf dem Laufenden.

Aber was, wenn das Team das Sprintziel noch nicht erreicht hat und Joe und Lisa sich doch weigern, mit irgendeiner sinnvollen Tätigkeit aufzuwarten? Normalerweise fasse ich eine der folgenden Strategien ins Auge (keine davon ist sehr nett, aber andererseits ist es der letzte Ausweg):

- **Schande:** "Naja, wenn Du keine Idee hast, wie Du dem Team helfen kannst, schlage ich vor, Du gehst nach Hause oder liest ein Buch oder sowas. Oder sitz einfach rum, bis jemand Dich um Hilfe herbei ruft."
- **Alte Schule:** Einfach eine Aufgabe zuweisen.
- **Gruppendruck:** "Fühlt Euch frei, Euch Zeit zu nehmen, Joe und Lisa, wir werden alle einfach hier stehen und es locker nehmen, bis Ihr mit etwas um die Ecke kommt, was Ihr tun werdet, das uns helfen wird, das Ziel zu erreichen."
- **Sklaverei:** "Naja, Ihr könnt dem Team indirekt helfen, indem Ihr heute den Butler spielt. Bringt Kaffee, gebt Leuten die Post, bringt den Müll raus, kocht uns Mittagessen und wasimmer wir sonst erbitten den Tag über." Du wärst erstaunt, wie schnell Joe und Lisa es hinbekommen, sich nützliche technische Aufgaben zu besorgen. :o)

Wenn eine Person Dich regelmäßig zwingt, so weit zu gehen, dann solltest Du diese Person wahrscheinlich beiseite nehmen und ein ernsthaftes Coaching mit ihr durchführen. Wenn das Problem bestehen bleibt, musst Du abschätzen, ob diese Person für Dein Team wichtig ist oder nicht.

Wenn sie *nicht* allzu *wichtig ist*, versuche sie aus dem Team zu entnehmen.

Wenn sie *wichtig ist*, dann versuche, sie mit jemand anderem in einer Zweiergruppe unterzubringen, der als ihr Schafhirt fungiert. Joe könnte ein großartiger Entwickler und Architekt sein, es ist nur so, dass er es wirklich vorzieht, wenn andere Leute ihm sagen, was zu tun ist. Schön. Verpflichte Niklas, Joes permanenter Schafhirt zu sein. Oder übernimmt selbst diese Pflicht. Wenn Joe für Dein Team wichtig genug ist, wird das die Mühe wert sein. Wir hatten Fälle wie diesen und es funktionierte mehr oder weniger.

> Das "Ich-weiß-nicht-was-ich-heute-tun-soll"-Problem ist typisch für Teams, für die Scrum neu ist und die es gewohnt sind, dass andere Leute Dinge für sie entscheiden. Sobald sie erfahrener in Sachen Selbstorganisation sind, verschwindet das Problem. Leute lernen herauszufinden, was zu tun ist. Also, wenn Du Scrum Master bist und Dich selbst dabei wieder findest, zu oft zu den obigen Tricks Zuflucht zu suchen, solltest Du darüber nachdenken, einen Schritt zurück zu gehen. Trotz Deiner hilfreichen Intention, könntest Du das größte Hindernis für das Team sein, indem Du sie davon abhälst zu lernen, wie man sich selbst organisiert!

TEIL NEUN

Wie wir Sprintdemos machen

Die Sprintdemo (oder das Sprint-Review, wie manche Leute es auch nennen) ist ein wichtiger Teil von Scrum, den Leute leicht unterschätzen.

"Oh, müssen wir *wirklich* eine Demo machen? Das macht wirklich nicht viel Spaß!"

"Wir haben keine Zeit, eine &%$#–Demo vorzubereiten!"

"Ich habe keine Zeit, an den Demos anderer Teams teilzunehmen!"

> Ich kann nicht verstehen, warum ich das "Sprintdemo" genannt habe! Was um alles in der Welt hatte ich mir dabei gedacht? Der offizielle Ausdruck ist "Sprint-Review", und das ist der viel bessere Ausdruck. Demo impliziert eine Einbahnstraßenkommunikation ("los geht's, das ist das, was wir gebaut haben"), während Review eine Kommunikation in beide Richtungen impliziert ("hier ist das, was wir gebaut haben, was denkt Ihr?"). Im Sprint-Review dreht sich alles ums Feedback! Also, wenn Du unten "Demo" liest, denk "Review", OK?

Warum wir darauf bestehen, dass alle Sprints mit einer Demo enden

Eine gut ausgeführte Sprintdemo, auch wenn das wenig bahnbrechend erscheinen mag, hat einen tiefgreifenden Effekt:

Das Team bekommt Anerkennung für seine Leistung. Sie *fühlen sich gut*.

- Andere Leute lernen, was Dein Team macht.
- Die Demo regt lebendiges Feedback der Stakeholder an.
- Demos sind (oder sollten sein) ein gemeinsames Ereignis, bei dem sich verschiedene Teams miteinander austauschen können und bei dem sie ihre Arbeit diskutieren können. Dies ist wertvoll.
- Eine Demo zu machen, zwingt das Team, *Sachen tatsächlich fertig zu stellen* und etwas auszuliefern (sogar, wenn es nur eine Testumgebung ist). Ohne Demos bleiben wir dabei, gewaltige Haufen an "99%-fertig"-Zeugs zu bekommen. Mit Demos mögen wir weniger Elemente fertigstellen, aber diese Elemente sind *wirklich fertig*, was (in unserm Fall) viel besser ist als einen ganzen Stapel an Zeug zu haben, das nur *sozusagen fertig* ist und den nächsten Sprint verunreinigen würde.

Wenn ein Team mehr oder weniger gezwungen ist, eine Sprintdemo zu machen, gerade wenn sie nicht viel haben, das funktioniert, wird die Demo peinlich. Das Team wird stottern und stocken, während es die Demo macht und der Applaus am Ende wird nur halbherzig kommen. Den Leuten wird es für das Team ein bisschen Leid tun, einige könnten genervt sein, dass sie Zeit verschwendet haben, um zu einer lausigen Demo zu gehen.

Das tut weh. Aber der Effekt ist wie eine bitter schmeckende Medizin. Im *nächsten Sprint* wird das Team versuchen, Sachen zu *erledigen*! Sie werden das Gefühl haben, dass "naja, vielleicht können wir nur zwei Sachen im nächsten Sprint demonstrieren statt fünf, aber verdammt, aber dieses Mal muss es verdammt nochmal LAUFEN!". Das Team weiß, dass sie eine Demo machen müssen, egal was, was die Chance wesentlich erhöht, dass es etwas Nützliches zu präsentieren gibt. Ich habe ein paar Mal miterlebt, dass dies passierte.

> Dies ist besonders entscheidend in einem Multi-Team-Kontext. Jeder, der beteiligt ist, muss regelmäßig zusammen kommen und das gemeinsame Produkt sehen. Es werden immer Integrationsprobleme auftreten, aber je früher Du sie entdeckst, desto leichter sind sie zu lösen. Selbstorgansiation funktioniert nur mit Transparenz und Feedbackschleifen, und eine gut ausgeführte Sprintdemo liefert beides.

Checkliste für Sprintdemos

- Stell sicher, dass Du das Sprintziel klar darstellst. Wenn Leute bei der Demo sind, die nichts über Dein Produkt wissen, nimm Dir ein paar Minuten, um das Produkt zu beschreiben.
- Verwende nicht zuviel Zeit darauf, die Demo vorzubereiten, besonders nicht auf protzige Präsentationen. Nimm alles Unnütze raus und konzentrier Dich nur darauf, den derzeit laufenden Code zu demonstrieren.
- Halte eine hohe Geschwindigkeit, d. h. richte Deine Vorbereitungen darauf aus, die Demo eher rasant als hübsch zu machen.
- Halte die Demo auf einem hohen, unternehmensaffinen Level. Lass die technischen Details raus. Konzentrier Dich eher auf "was haben wir gemacht" als auf "wie haben wir es gemacht".
- Wenn möglich, lass das Publikum das Produkt selbst ausprobieren.
- Demonstriere kein Bündel von kleineren Bugfixes und trivialen Features. Erwähne sie, aber demonstriere sie nicht, da das im allgemeinen zu lange dauert und die Aufmerksamkeit von den wichtigeren Storys ablenkt.

Einige Teams machen zwei Reviews: Eine kurze öffentlichen rückblickende Besprechung mit externen Stakeholdern als Zielgruppe, gefolgt von einer internen Bewertung mit mehr Details und Sachen wie den wesentlichen Herausforderungen und technischen Entscheidungen, die auf dem Weg getroffen wurden. Eine großartige Art, Wissen zwischen Teams zu verbreiten, und es erspart den Stakeholdern technische Details, die sie nicht interessieren.

Mit undemonstrierbarem Zeug umgehen

Teammitglied: Ich werde diesen Punkt nicht demonstrieren, weil ich es nicht demonstrieren kann. Die Story lautet "verbessere die Skalierbarkeit, so dass das System 10.000 Anwender gleichzeitig bewältigen kann". Ich kann ja schlecht 10.000 Leute zur gleichzeitigen Anwendung zur Demo einladen, oder?

Scrum Master: Bist Du mit dem Punkt fertig?

Teammitglied: Ja, natürlich.

Scrum Master: Woher weißt Du das?

Teammitglied: Ich habe das System in einer Performanztestumgebung eingerichtet, acht Lastserver gestartet und bin dem System mit gleichzeitigen Anfragen auf die Pelle gerückt.

Scrum Master: Aber hast Du irgendeinen Indikator dafür, dass das System 10.000 Anwender gleichzeitig bewältigt?

Teammitglied: Ja. Die Testmaschinen sind beschissen, doch sie konnten bei meinem Test 50.000 simultane Anfragen schaffen.

Scrum Master: Woher weißt Du das?

Teammitglied (frustriert)**:** Naja, ich habe diesen Bericht! Du kannst es selbst sehen. Es zeigt, wie der Test aufgesetzt war und wie viele Anfragen gesendet wurden!

Scrum Master: Oh, exzellent! Dann ist das Deine "Demo". Zeig einfach den Bericht und geh ihn mit dem Publikum durch. Besser als nichts, richtig?

Teammitglied: Oh, das ist genug? Aber er ist hässlich. Ich muss ihn aufpolieren.

Scrum Master: OK, aber mach nicht zu lang. Er muss nicht hübsch sein, nur informativ.

TEIL ZEHN

Wie wir Sprintretrospektiven machen

Warum wir darauf bestehen, dass alle Teams Retrospektiven durchführen

Das Wichtigste bei Retrospektiven ist *sicherzustellen, dass sie stattfinden*.

Aus irgendeinem Grund scheinen Teams nicht immer geneigt zu sein, Retrospektiven durchzuführen. Ohne sanftes Stupsen würden die meisten unserer Teams die Retrospektive oft auslassen und statt dessen zum nächsten Sprint übergehen. Es mag ein kulturelles Ding in Schweden sein. Ich bin nicht sicher.

> Nö, ich habe das in vielen Ländern gesehen, also liegt es einfach in der menschlichen Natur. Wir möchten immer zum Nächsten weiter gehen. Ironischerweise, je mehr Du gestresst bist, desto wahrscheinlicher möchtest Du die Retrospektive auslassen. Aber je mehr Du gestresst bist, desto dringender brauchst Du die Retrospektive! Sowas wie "Ich bin so in Eile, die Bäume zu fällen, ich habe keine Zeit inne zu halten und meine Säge zu schärfen!" Also sollte ein Scrum Master nachdrücklich darauf bestehen, Retrospektiven durchzuführen! Doch sie brauchen nicht so lange zu dauern. Für Zwei-Wochen-Sprints begrenze die Retrospektive auf eine Stunde. Aber mache eine längere Retrospektive (einen halben oder ganzen Tag) alle paar Monate, so dass Du stachligere Angelegenheiten abhandeln kannst.

Dennoch scheint jeder zuzustimmen, dass Retrospektiven extrem nützlich sind. Tatsächlich würde ich sagen, die Retrospektive ist das zweitwichtigste Ding in Scrum (das wichtigste ist das Sprintplanungstreffen), weil dies Deine *beste Chance für Verbesserung ist*!

> Exakt. Und das ist es, warum die Retrospektive das *Nummer-Eins-allerwichtigste* Ding in Scrum ist, nicht das zweitwichtigste!

Natürlich brauchst Du ein Retrospektiventreffen nicht, um auf gute Ideen zu kommen – Du kannst das zu Hause in Deiner Badewanne machen! Aber wird das Team Deine Idee akzeptieren? Vielleicht, aber die Wahrscheinlichkeit, dass das Team sie voll mitträgt, ist sehr viel höher, wenn die Idee "vom Team" kommt, d. h. wenn sie während der Retrospektive aufkommt, wenn es jedem möglich ist, dazu beizutragen und die Ideen zu diskutieren.

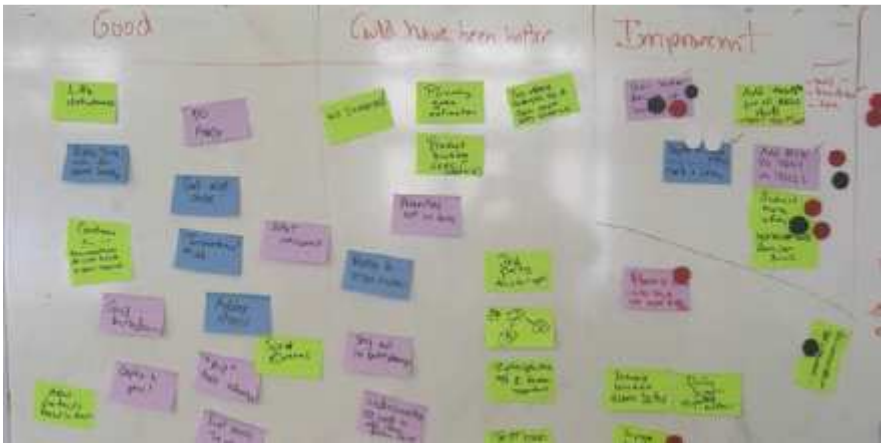
Du wirst sehen, ohne Retrospektiven macht das Team immer wieder dieselben Fehler, immer und immer wieder.

Wie wir Retrospektiven organisieren

Das allgemeine Format variiert ein bisschen, aber üblicherweise machen wir es in etwa wie folgt:

- Wir nehmen uns eine bis drei Stunden Zeit, abhängig davon, wie viel Diskussion zu erwarten ist.
- Teilnehmer: Product Owner, das ganze Team, und ich.
- Wir gehen in einen gesonderten Raum, eine gemütliche Sofaecke, auf die Dachterrasse oder an einen ähnlichen Ort. Hauptsache, wir können dort ungestört diskutieren.
- Normalerweise halten wir Retrospektiven nicht im Teamraum ab, da die Aufmerksamkeit der Leute dort dazu tendiert, weg zu wandern.
- Irgendwer wird zum Schriftführer ernannt.
- Der Scrum Master zeigt das Sprint-Backlog und fasst mit der Hilfe des Teams den Sprint zusammen. Wichtige Ereignisse und Entscheidungen etc.
- Wir machen "die Runde". Jede Person bekommt – ohne unterbrochen zu werden – die Gelegenheit zu sagen, was sie denkt, was gut war, was sie denkt, was hätte besser laufen können und was sie beim nächsten Sprint gern anders hätte.
- Wir sehen uns die geschätzte vs. tatsächliche Umsetzungsgeschwindigkeit an. Wenn es einen großen Unterschied gibt, versuchen wir zu analysieren, warum.
- Wenn die Zeit fast rum ist, versucht der Scrum Master konkrete Vorschläge dazu zusammenzufassen, was wir im nächsten Sprint besser machen können. Unsere Retrospektiven sind im allgemeinen nicht zu strukturiert. Das zugrundeliegende Thema ist immer derselbe Gedanke: "Was können wir im nächsten Sprint besser machen?"

Hier ist ein Whiteboard-Beispiel einer unserer letzten Retrospektiven:



Drei Spalten:

- **Gut:** Wenn wir denselben Sprint noch einmal machen könnten, würden wir diese Dinge genauso wieder machen.
- **Könnten wir besser gemacht haben:** Wenn wir denselben Sprint noch einmal machen könnten, würden wir diese Dinge anders machen.
- **Verbesserungen:** Konkrete Ideen darüber, wie wir uns in der Zukunft verbessern können.

Die Spalten eins und zwei blicken also in die Vergangenheit, während die Spalte drei in die Zukunft blickt.

Nachdem das Team alle diese Haftnotizen gebrainstormt hat, legten sie per "Punktabstimmung" fest, auf welche Verbesserungen im nächsten Sprint fokussiert werden sollte. Jedes Teammitglied bekam drei Magnete und war eingeladen, für diejenigen Verbesserungsvorschläge abzustimmen, die dieses Teammitglied während des nächsten Sprints gern mit hoher Priorität im Team sehen würde. Jedes Teammitglied konnte die Magnete beliebig verteilen, sogar alle drei zum selben Punkt.

Auf dieser Grundlage wählten sie fünf Prozessverbesserungen aus, auf die sie sich konzentrieren wollten und die sie während der nächsten Retrospektive nachverfolgen würden.

Es ist wichtig, hierbei nicht überambitioniert zu sein. Konzentriert Euch auf nur einige wenige Verbesserungen pro Sprint.

> Es gibt eine Menge originelle Arten, Retrospektiven durchzuführen. Variiere das Format, so dass die Treffen nicht fade werden. Du findest viele Ideen im Buch *Agile Retrospectives*. Retromat, ein Zufallsretrospektivengenerator, macht auch Spaß (www.plans-for-retrospectives.com). :o)
>
> Allerdings ist mir aufgefallen, dass ich immer wieder auf das simple, oben beschriebene Format zurückkomme. Es funktioniert für die meisten Fälle. Oder noch einfacher, nimm Dir eine 20-Minuten-Kaffeepause mit zwei Diskussionsthemen: "Was beibehalten" und "Was ändern". Ein bisschen mager, aber besser als nichts!

Gewonnene Erkenntnisse zwischen Teams verbreiten

Die Informationen, die während einer Sprintretrospektive zur Sprache kommen, sind normalerweise extrem wertvoll. Hat dieses Team bezüglich seiner Ausrichtung eine harte Zeit gehabt, weil der Vertriebsleiter immer wieder Programmierer kidnappt, damit sie als "Technikexperten" an Vertriebstreffen teilnehmen? Dies ist eine wichtige Information. Vielleicht hatten andere Teams dasselbe Problem? Sollten wir das Produktmanagement mehr über unsere Produkte aufklären, so dass diese Leute den Vertriebssupport leisten können?

Oder noch besser, lade die Vertriebsleiter zu einem Treffen ein, lerne seine Bedürfnisse kennen und diskutiert gemeinsam mögliche Lösungen!

In einer Sprintretrospektive geht es nicht nur darum, wie dieses eine Team während des nächsten Sprints einen besseren Job erledigen kann; es hat weitergehende Auswirkungen als das.

Unsere Strategie, dies zu handhaben, ist sehr einfach. Eine Person (in diesem Fall ich) nimmt an allen Sprintretrospektiven teil und fungiert als Wissensbrücke. Sehr informell.

Eine Alternative wäre, dass jedes Scrumteam einen Sprintretrospektivenbericht veröffentlichen muss. Wir haben das ausprobiert, aber gemerkt, dass solche Berichte nicht viele Leute lesen, und noch weniger darauf reagieren. Also machen wir es statt dessen wieder auf die einfache Art.

Wichtige Regeln für die "Wissensbrücke"-Person:

- Er sollte ein guter Zuhörer sein.
- Wenn die Retrospektive zu ruhig ist, sollte er darauf vorbereitet sein, simple, aber sehr gezielte Fragen zu stellen, die die Diskussion innerhalb der Gruppe anregen. Zum Beispiel, "Wenn Du die Zeit zurückdrehen und denselben Sprint von Tag eins an noch einmal machen könntest, was würdest Du anders machen?"
- Er sollte bereit sein, Zeit darauf zu verwenden, alle Retrospektiven aller Teams zu besuchen.
- Er sollte eine Art Autoritätsposition haben, so dass er auf Verbesserungsvorschläge reagieren kann, die außerhalb der Kontrolle des Teams liegen.

Dies funktioniert ganz gut, aber es mag andere Ansätze geben, die ein ganzes Stück besser funktionieren. In diesem Fall, erleuchte mich bitte.

"Trading facilitator"[14] ist ein nettes Pattern. Nach dem Muster "Ich werde für Dein Team die Retrospektive moderieren, wenn Du für meins moderierst." Sorgt für einfache Zwei-Wege-Wissensverteilung und erlaubt Dir als Scrum Master außerdem, uneingeschränkt an der Retrospektive Deines Teams teilzunehmen (anstatt zu moderieren).

Ändern oder nicht ändern

Sagen wir, das Team kommt zum Schluss, "wir haben zu wenig innerhalb des Teams kommuniziert, so dass wir uns immer wieder gegenseitig auf die Füße getreten sind und gegenseitig unsere Designs durcheinander gebracht haben."

Was würdest Du machen? Tägliche Designtreffen einführen? Neue Tools zur leichteren Kommunikation anschaffen? Mehr Wikiseiten anlegen? Naja, vielleicht. Aber andererseits, vielleicht doch nicht.

Wir haben festgestellt, dass in vielen Fällen ein Problem genau zu identifizieren Klärung genug ist, damit es sich automatisch im nächsten Sprint löst. Besonders wenn Du die Sprintretrospektive im Teambüro an die Wand pinnst (was wir immer vergessen, wir sollten uns schämen!). Jede Änderung, die Du einführst, bringt irgendwelche Kosten mit, also, bevor Du Änderungen einführst, berücksichtige die Alternative, überhaupt nichts zu tun und zu hoffen, dass das Problem automatisch verschwindet (oder kleiner wird).

Das Beispiel oben ("wir kommunizieren zu wenig innerhalb des Teams…") ist ein typisches Beispiel von etwas, das am besten gelöst wird, indem man nichts ändert.

Wenn Du jedesmal, wenn sich jemand über irgend etwas beschwert, eine Änderung einführst, könnte es sein, dass die Leute unwillig werden, kleinere Problembereiche offen zu legen, was fürchterlich wäre.

Beispiele von Dingen, die während einer Retrospektive sichtbar werden könnten

Hier sind ein paar Beispiele typischer Dinge, die während der Sprintplanung auftauchen, und typische Maßnahmen.

[14] Anm. d. Übersetzerin: Jemand, der Handel (z. B. über nationale Grenzen hinweg) fördert oder erleichtet, s. https://en.wikipedia.org/wiki/Trade_facilitation

"Wir hätten mehr Zeit dafür verwenden sollen, Storys in Untereinträge und Aufgaben herunterzubrechen"

Dies ist weit verbreitet. Jeden Tag beim Daily Scrum ertappen sich Teammitglieder dabei zu sagen, "Ich weiß wirklich nicht, was ich heute tun soll." Also treibst Du nach jedem Daily Scrum Aufwand, um konkrete Aufgaben zu finden. Normalerweise ist es effektiver, dies im Voraus zu erledigen.

Typische Maßnahmen: Keine. Das Team wird dies wahrscheinlich während der nächsten Sprintplanung klären. Wenn das wiederholt passiert, mache das Zeitfenster für die Sprintplanung größer.

"Zu viele projektfremde Störungen"

Typische Maßnahmen:

- Bitte das Team, ihren Fokusfaktor im nächsten Sprint zu reduzieren, so dass sie realistischer planen können.
- Bitte das Team, Störungen im nächsten Sprint besser zu dokumentieren. Wer störte, wie lange dauerte das. Wird es einfacher machen, das Problem später zu lösen.
- Bitte das Team, alle Störungen dem Scrum Master oder Product Owner einzutrichtern.
- Bitte das Team, eine Person als "Torhüter" zu bestimmen. Alle Störungen werden zu ihm geleitet, so dass der Rest des Teams konzentriert arbeiten kann. Könnte der Scrum Master machen oder eine rotierende Funktion sein.

Das rotierende Torhüter-Pattern ist extrem üblich und funktioniert normalerweise gut. Versuch es!

"Wir haben zuviel zugesagt und nur die Hälfte der Sachen fertig bekommen"

Typische Maßnahmen: Keine. Das Team wird im nächsten Sprint wahrscheinlich nicht zuviel zusagen. Oder wenigsten nicht genauso schlimm danebenliegen.

Übrigens, seit 2014 ist der Ausdruck "Sprintzusage" (sprint commitment) komplett aus dem Scrum Guide herausgenommen worden. Statt dessen wurde er umbenannt in "Sprintvorhersage" (sprint forecast). Viel besser! Das Wort "Zusage" hat soviel Missverständnisse verursacht. Viele Teams dachten, der Sprintplan war eine Art von Versprechen (ein bisschen töricht, berücksichtigt man, dass einer von vier Schlüsselwerten der Agilität ist "eher auf Änderungen reagieren als einem Plan folgen"). Der Sprintplan ist keine Zusage, es ist eine Vorhersage und eine Hypothese – "Dies ist das, was wir denken, wie wir das Sprintziel am besten erreichen können."

Nichtsdestotrotz ist es immer noch ziemlich beschissen, ständig weniger als vorhergesagt abzuliefern. Wenn das ein Problem ist, fangt konsequent mit "Yesterday's Weather" an und zieht nur so viele Story-Punkte, wie Ihr im letzten Sprint geschafft habt (oder den Durchschnitt der letzten drei Sprints, wenn Du originell sein möchtest). Dieser simple und mächtige Trick lässt das Problem normalerweise einfach wegschmelzen, indem Deine Umsetzungsgeschwindigkeit sich allmählich selbst reguliert.

"Unser Büroumfeld ist zu laut und chaotisch"

Typische Maßnahmen:

- Versuche ein besseres Umfeld zu schaffen oder ziehe mit dem Team woandershin um. Miete ein Hotelzimmer. Was auch immer. Siehe Seite 55 "Wie wir den Teamraum gestalten".
- Wenn das nicht möglich ist, sag dem Team, dass es seinen Fokusfaktor für den nächsten Sprint herunter setzen soll und dass es klar sagen soll, dass dies aufgrund des lauten und chaotischen Umfelds geschieht. Hoffentlich wird dies den Product Owner dazu veranlassen, das obere Management damit zu belästigen.

Glücklicherweise musste ich nie drohen, mit dem Teambüro vom Gelände abzuziehen. Aber ich werde, wenn ich es muss. :o)

TEIL ELF

Freie Zeiten[15] zwischen den Sprints

[15] Slack time

Im echten Leben kannst Du nicht immer sprinten. Du musst zwischen den Sprints ausruhen. Wenn Du immer sprintest, machst Du effektiv nur Jogging.

> Das trifft auch auf's Leben zu, nicht nur auf Scrum! Zum Beispiel, wenn ich in meinem Leben keine freie Zeit hätte, würde dieses Buch nicht existieren (noch eine zweite Auflage oder irgendeins meiner anderen Bücher, Artikel, Videos etc.). Freie Zeit ist superwichtig sowohl für die Produktivität als auch für das persönliche Wohlergehen! Wenn Du einer dieser Kalender-Immer-Voll-Leute bist, versuch folgendes: Öffne Deinen Kalender und blocke einen halben Tag pro Woche, schreib "freie Zeit"[16] oder "nicht verfügbar" oder sowas rein. Entscheide Dich nicht im Voraus dafür, was Du in dieser Zeit tun wirst, sieh einfach, was passiert. :o)

Dasselbe gilt für Scrum und für Softwareentwicklung im Allgemeinen. Sprints sind ziemlich intensiv. Als Entwickler kommst Du nie wirklich dazu, freie Zeit zu haben; jeden Tag musst Du bei diesem verfluchten Treffen stehen und jedem erzählen, was Du gestern fertig gemacht hast. Wenige werden dazu neigen zu sagen, "Ich habe den Großteil des Tages darauf verwendet, meine Füße auf den Tisch zu legen, in Blogs zu schmökern und Cappuccino zu trinken."

Zusätzlich zum eigentlichen Ausruhen selbst gibt es einen andern guten Grund, einige freie Zeit zwischen den Sprints zu haben. Nach der Sprintdemo und Retrospektive werden sowohl das Team als auch der Product Owner voller Informationen und Ideen sein, die sie zu verdauen haben. Wenn sie sofort loslaufen und mit der nächsten Sprintplanung beginnen, ist die Chance hoch, dass niemand die Gelegenheit haben wird, irgendeine Information oder Gelerntes zu verdauen, der Product Owner wird keine Zeit haben, seine Prioritäten nach der Sprintdemo zu justieren etc.

Schlecht:

Montag
09-10: Sprint 1 Demo 10-11: Sprint 1 Retrospektive
13-16: Sprint 2 Planung

Besser:

Montag	**Dienstag**
09-10: Sprint 1 Demo 10-11: Sprint 1 Retrospektive	9-13: Sprint 2 Planung

Noch besser:

Freitag	**Samstag**	**Sonntag**	**Montag**
09-10: Sprint 1 Demo 10-11: Sprint 1 Retrospektive			9-13: Sprint 2 Planung

Eine Art, dies zu tun, sind "Labortage" (oder wie auch immer Ihr sie nennen möchtet). Das sind Tage, an denen Entwicklern erlaubt ist, in erster Linie das zu tun, was immer sie möchten (OK, ich geb's zu, inspiriert von Google). Zum Beispiel etwas über die neuesten Tools und APIs lesen, für eine Zertifizierung lernen, Nerdzeug mit Kollegen diskutieren, ein Hobbyprojekt programmieren etc.

Unser Ziel ist, je einen Labortag zwischen zwei Sprints zu haben. Auf die Art bekommst Du eine natürliche Auszeit zwischen den Sprints und wirst ein Entwicklungsteam haben, dass eine realistische Chance hat, sein Wissen Up-to-date zu halten. Plus, es ist ein ziemlich attraktiver Benefit, was das Arbeitsverhältnis betrifft.

Am besten?

Donnerstag	**Freitag**	**Samstag**	**Sonntag**	**Montag**
09-10: Sprint 1 Demo 10-11: Sprint 1 Retrospektive	LAB Tag			9-13: Sprint 2 Planung

Im Augenblick haben wir Labortage einmal im Monat. Der erste Freitag jeden Monat, um genau zu sein. Warum nicht statt dessen zwischen den Sprints? Naja, weil ich das Gefühl hatte, es wäre wichtig, dass sich das ganze Unterneh-

[16] Slack oder Slack time im englischen Original (Anm. der Übersetzerin)

men einen Labortag zur selben Zeit nimmt. Andererseits, Leute neigen dazu, das nicht ernst zu nehmen. Und weil wir (bis jetzt) keine aneinander ausgerichteten Sprints über alle unsere Produkte haben, musste ich statt dessen ein sprintunabhängiges Labortagintervall wählen.

Wir könnten eines Tages ausprobieren, die Sprints über alle Produkte zu synchronisieren (d. h. dasselbe Sprintstart- und Sprintendedatum für alle Produkte und Teams). In diesem Fall würden wir definitiv einen Labortag zwischen den Sprints platzieren können.

> Nach vielen Experimenten bei Spotify landeten wie dabei, unternehmensweite Hack-Wochen abzuhalten. Zweimal pro Jahr haben wir eine komplette Woche an Mach-was-immer-Du-möchtest, mit einer Demo und einer Party am Freitag. Das ganze Unternehmen, nicht nur Technikleute. Die Menge an Innovationen, die dies anstößt, ist einfach erstaunlich! Und weil jeder das zur selben Zeit tut, ist es weniger wahrscheinlich, dass Teams wegen Abhängigkeiten entgleisen. Ich habe ein Video über Spotifys Ingenieurskultur gedreht, das die Hack-Woche und viele andere Dinge beschreibt. Sieh's Dir an unter
> http://tinyurl.com/spotifyagile

TEIL ZWÖLF

Wir wir Releaseplanung und Festpreisverträge machen

Manchmal müssen wir mehr voraus planen als einen Sprint nach dem anderen – typischerweise in Verbindung mit einem Festpreisvertrag, bei dem wir im Voraus planen *müssen* oder bei dem wir sonst riskieren, dass wir etwas unterschreiben, was wir nicht rechtzeitig liefern können.

Typischerweise ist Releaseplanung für uns ein Versuch, die Frage "*wann spätestens* werden wir in der Lage sein, Version 1.0 dieses neuen Systems abzuliefern" zu beantworten.

Wenn Du *wirklich* etwas über Releaseplanung lernen möchtest, schlage ich vor, Du überschlägst dieses Kapitel und kaufst statt dessen Mike Cohns Buch *Agile Estimating and Planning*. Ich wünschte wirklich, ich hätte das Buch früher gelesen (ich las es, *nachdem* wir diese Sachen selbst herausgeknobelt hatten…). Meine Version von Releaseplanung ist ein bisschen grob vereinfachend, aber sollte als guter Startpunkt dienen können.

> Sieh Dir auch das Buch *Lean Startup* von Eric Ries an. Ein großes Problem ist, dass die meisten Projekte dazu neigen, ein Big-Bang-Release aufzubauen statt kleine Erweiterungen auszuliefern und zu messen, um zu sehen, ob sie auf dem richtigen Weg sind. Lean Startup, wenn richtig angewendet, reduziert radikal Risiko und Kosten für's Versagen.

Definiere Deine Akzeptanzgrenzwerte

Zusätzlich zum üblichen Produkt-Backlog definiert der Product Owner eine Liste von *Akzeptanzgrenzwerten*, was eine einfache Klassifizierung davon ist, was das Wichtigkeitsranking im Produkt-Backlog im Sinne des Vertrags tatsächlich aussagt.

Hier ist ein Beispiel für Akzeptanzgrenzwertregeln:

- Alle Einträge mit Wichtigkeit >= 100 *müssen* in Version 1.0 enthalten sein oder wir werden die Todesstrafe bekommen.
- Alle Einträge mit Wichtigkeit 50 bis 99 *sollten* in Version 1.0 enthalten sein, aber wir könnten damit durchkommen, wenn wir sie in einem schnell folgenden Release nachliefern.
- Einträge mit Wichtigkeit 25 bis 49 sind erforderlich, aber sie können in einem nachfolgenden Release 1.1 gemacht werden.
- Einträge mit Wichtigkeit < 25 sind spekulativ und werden vielleicht überhaupt nie benötigt.

Und hier ist ein Beispiel eines Product Backlogs, farbcodiert, basierend auf den oben genannten Regeln.

Wichtigkeit	Name
130	Banane
120	Apfel
115	Orange
110	Guave
100	Birne
95	Rosine
80	Erdnuss
70	Donut
60	Zwiebel
40	Grapefruit
35	Papaya
10	Blaubeere
10	Pfirsich

Rot = *muss* enthalten sein in Version 1.0 (Banane bis Birne)

Gelb = *sollte* enthalten sein in Version 1.0 (Rosine bis Zwiebel)

Grün = könnte später gemacht werden (Grapefruit bis Pfirsich)

Also, wenn wir alles von Banane bis Zwiebel bis zur Deadline liefern, sind wir auf der sicheren Seite. Wenn uns die Zeit ausgeht, *könnten* wir damit durchkommen, Rosine, Erdnuss, Donut oder Zwiebel rauszulassen. Alles unter Zwiebel ist Dreingabe.

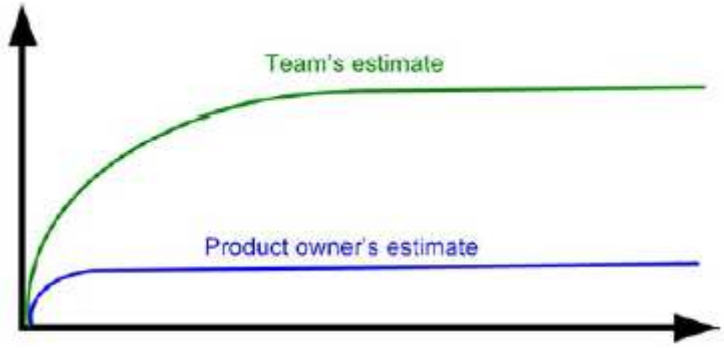
> Und Du kannst, natürlich, diese Analyse machen, ohne numerische Wichtigkeitsrankings! Sortiere einfach die Liste. Allerdings hast Du das ja schon gemacht.

Zeit schätzen für die wichtigsten Punkte

Um die Releaseplanung zu machen, braucht der Product Owner Schätzungen, mindestens für alle Storys, die im Vertrag drin sind. Genau wie bei der Sprintplanung ist dies eine kooperative Leistung von Product Owner und Team – das Team schätzt, der Product Owner beschreibt die Punkte und beantwortet Fragen.

Eine Zeitschätzung ist *wertvoll*, wenn sich herausstellt, dass sie nah an der Wirklichkeit ist, weniger wertvoll, wenn sich herausstellt, dass sie, sagen wir, um einen Faktor 30 % abweicht, und komplett wertlos, wenn sie keinerlei Verbindung zur Realität hat.

Hier ist mein Verständnis über den Wert einer Zeitschätzung in Relation dazu, wer sie berechnet und wieviel Zeit das Team schließlich benötigt hat.



x-Achse: Wert der geschätzten Zeit
y-Achse: Zeit, die für die Zeitschätzung aufgewendet wird

All das war nur ein umständlicher Weg, um zu sagen:

- Lass das *Team* schätzen.
- Lass sie nicht zu lange Zeit darauf verwenden.
- Stelle sicher, dass sie verstehen, dass Zeitschätzungen *grobe Schätzungen* sind, keine *Zusagen*.

Normalerweise versammelt der Product Owner das ganze Team in einem Raum, stellt einige Erfrischungen bereit und sagt dem Team, dass das Ziel dieses Treffens ist, Zeitschätzungen für die Top 20 (oder so) Storys im Produkt Backlog zu machen. Er geht einmal durch jede Story und lässt dann das Team arbeiten. Der Product Owner bleibt im selben Raum, um Fragen zu beantworten und den Umfang für jeden Punkt zu klären, soweit notwendig. Genau wie bei der Sprintplanung ist das "How to Demo"-Feld eine sehr nützliche Art, das Risiko von Missverständnissen zu verringern.

Dieses Treffen muss einem strengen Zeitfenster unterliegen, sonst neigt das Team dazu, zuviel Zeit auf die Zeitschätzung zu weniger Storys zu verwenden.

Wenn der Product Owner mehr Zeit darauf verwenden möchte, setzt er einfach ein weiteres Treffen später an. Das Team muss sicherstellen, dass der Einfluss dieser Treffen auf ihren aktuellen Sprint für den Product Owner klar ersichtlich ist, so dass er versteht, dass ihre Zeitschätzarbeit nicht umsonst zu haben ist.

Hier ist ein Beispiel, wie Zeitschätzungen am Ende aussehen könnten (in Story-Punkten):

Wichtigkeit	Name	Schätzung
130	Banane	12
120	Apfel	9
115	Orange	20
110	Guave	8
100	Birne	20
95	Rosine	12
80	Erdnuss	10
70	Donut	8
60	Zwiebel	10
40	Grapefruit	14
35	Papaya	4
10	Blaubeere	
10	Pfirsich	

> Sehr verwirrend, all dies "Zeitschätzung" zu nennen. Nenn das lieber "Größenabschätzung". Ich weiß nicht, wie lang "Banane" brauchen wird, aber ich bin ziemlich sicher, es dauert ein bisschen länger als "Apfel" und viel weniger lang als "Orange". Besser, *grob richtig* zu liegen als *genau falsch*!

Umsetzungsgeschwindigkeit schätzen

OK, dann haben wir jetzt eine Art von grober Zeitschätzung für die wichtigsten Storys gefunden.

Der nächste Schritt ist, unsere durchschnittliche Umsetzungsgeschwindigkeit pro Sprint zu schätzen.

Das bedeutet, dass wir uns für einen Fokusfaktor entscheiden müssen. Siehe Seite 27 " Wie entscheidet ein Team, welche Storys im Sprint enthalten sein sollen?"

> Oh nein, nicht wieder der verdammte Fokusfaktor…

Der Fokusfaktor heißt im Grunde "wieviel Zeit des Teams wird dafür verwendet, sich auf die aktuell zugesagten Storys zu konzentrieren". Das sind nie 100 %, da das Team Zeit verliert, um ungeplante Punkte umzusetzen, Kontextwechsel zu vollziehen, anderen Teams zu helfen, ihre Email zu checken, ihre Computer zu reparieren, in der Küche gegen die Politik zu argumentieren etc.

Lass uns annehmen, wir setzten den Fokusfaktor für das Team auf 50 % (ziemlich niedrig – normalerweise bewegen wir uns um 70 % herum). Und lass uns annehmen, unsere Sprintlänge wird drei Wochen (15 Tage) betragen und unsere Teamgröße ist sechs.

Jeder Sprint ist dann 90 Manntage lang, aber es kann nur erwartet werden, den Wert von 45 Manntagen an Storys umzusetzen (aufgrund des 50%-Fokusfaktors).

Also beträgt unsere geschätzte Umsetzungsgeschwindigkeit 45 Story-Punkte.

> Lass den Fokusfaktor weg. Bitte das Team, auf die Liste zu starren und auf Erfahrung basierend zu raten, wie weit sie in einem Sprint kommen können. Zähl die Punkte. Das wird schneller gehen als der Fokusfaktor und in etwa genauso exakt / ungenau sein. Besser grob richtig zu liegen als… - oh, Moment. Das sagte ich bereits.

Wenn jede Story eine Zeitschätzung von fünf Tagen hätte (was sie nicht haben), dann würde dieses Team ungefähr neun Storys pro Sprint wie am Fließband herstellen.

Einen Releaseplan zusammenstellen

Jetzt, da wir Zeitschätzungen und eine Umsetzungsgeschwindigkeit (45) haben, können wir leicht das Produkt-Backlog in Sprints zerteilen:

Wichtigkeit	Name	Schätzung
Sprint 1		
130	Banane	12
120	Apfel	9
115	Orange	20
Sprint 2		
110	Guave	8
100	Birne	20
95	Rosine	12
Sprint 3		
80	Erdnuss	10
70	Donut	8
60	Zwiebel	10
40	Grapefruit	14
Sprint 4		
35	Papaya	4
10	Blaubeere	
10	Pfirsich	

Jeder Sprint beinhaltet so viele Storys wie möglich, ohne die geschätzte Umsetzungsgeschwindigkeit von 45 zu überschreiten.

Jetzt können wir sehen, dass wir wahrscheinlich drei Sprints brauchen werden, um alle "Must haves" und "Should haves" umzusetzen.

Drei Sprints = neun Kalenderwochen = zwei Kalendermonate. Nun, ist das die Deadline, die wir dem Kunden versprechen? Das hängt komplett von der Natur des Vertrags ab – wie sehr der Umfang festgeschrieben ist etc. Wir fügen normalerweise ein erhebliches Polster hinzu, um uns gegen schlechte Zeitschätzungen, unerwartete Probleme, unerwartete Features etc. zu wappnen. Also würden wir uns in diesem Fall darauf einigen, das Lieferdatum auf drei Monate in der Zukunft zu setzen, was uns einen Monat "Reserve" gibt.

> Hier ist ein alternativer Ansatz, der hübsch funktioniert. Schätze die Umsetzungsgeschwindigkeit als Bereich (30 – 50 Punkte). Dann teile das Backlog in drei Listen:
>
> **Alle**: Alle diese Storys werden fertiggestellt werden, wenn unsere Umsetzungsgeschwindigkeit niedrig ist (30).
>
> **Einige**: Einige dieser Storys werden fertiggestellt werden, aber nicht alle.
>
> **None**: Keine dieser Storys wird fertiggestellt werden, sogar wenn unsere Umsetzungsgeschwindigkeit hoch ist (50).

Das nette daran ist, dass wir dem Kunden alle drei Wochen etwas Benutzbares demonstrieren und ihn einladen können, die Anforderungen zu ändern, während das Projekt fortschreitet (natürlich abhängig davon, wie der Vertrag aussieht).

Den Releaseplan anpassen

Die Realität wird sich nicht an den Plan anpassen, also müssen wir es anders herum machen.

Nach jedem Sprint sehen wir uns die tatsächliche Umsetzungsgeschwindigkeit für diesen Sprint an. Wenn die tatsächliche Umsetzungsgeschwindigkeit sich sehr von der geschätzten unterscheidet, ändern wir die geschätzte Umsetzungsgeschwindigkeit für die künftigen Sprints und aktualisieren den Releaseplan. Wenn uns das in Schwierigkeiten bringt, kann der Product Owner damit beginnen, mit dem Kunden zu verhandeln, oder er überlegt, wie er den Umfang reduzieren kann, ohne den Vertrag zu verletzen. Oder vielleicht kommen er und das Team auf eine Idee, um die Umsetzungsgeschwindigkeit zu erhöhen oder den Fokusfaktor zu reduzieren, indem einige gravierende Hindernisse, die während des Sprints identifiziert wurden, beseitigt werden.

Der Product Owner könnte den Kunden anrufen und sagen, "Hallo, wir sind etwas hinter dem Plan, aber ich glaube, wie können die Deadline halten, wenn wir nur das embedded Pac-Man-Feature rausnehmen, dessen Bau viel Aufwand erfordert. Wir können es ins Folgerelease reinnehmen, das drei Wochen nach dem ersten Release kommt, wenn Sie möchten."

Vielleicht keine gute Nachricht für den Kunden, aber wenigstens sind wir ehrlich und geben dem Kunden früh die Chance zu entscheiden – sollen wir die wichtigen Dinge pünktlich liefern oder alles später? Normalerweise keine schwere Entscheidung. :o)

> Ich habe ein 15-Minuten-Video gemacht mit dem Namen "Agile Product Ownership in a Nutshell". Es enthält ein Bündel nützlicher Tipps und Tricks rund um Releaseplanung und Backlogmanagement in Scrum. Sieh's Dir mal an!
>
> http://tinyurl.com/ponutshell

TEIL DREIZEHN

Wie wir Scrum und XP kombinieren

Zu sagen, dass Scrum und XP (eXtreme Programming) nutzbringend kombiniert werden können, ist eigentlich keine kontroverse Aussage. Die meisten Dinge, die ich in der Netzgemeinde sehe, unterstützen diese Hypothese, also werde ich nicht darüber argumentieren, warum.

Naja, eine Sache will ich erwähnen. Scrum konzentriert sich auf Management- und Organisationsmethoden, während sich XP größtenteils auf die eigentlichen Programmiermethoden fokussiert. Darum spielen sie gut zusammen – sie adressieren verschiedene Bereiche und ergänzen einander.

Hiermit füge ich meine Stimme dem existierenden erfahrungsbasierten Beweis hinzu, dass Scrum und XP nutzbringend kombiniert werden können!

> Ich habe von Jeff Sutherland gelernt, dass das ursprüngliche Scrum tatsächlich alle XP-Methoden enthielt. Aber Ken Schwaber überzeugte ihn, die rein technischen Methoden aus Scrum rauszulassen, um das Modell einfach zu halten und den Teams die Verantwortung für ihre technischen Methoden selbst zu überlassen. Vielleicht half dies, Scrum schneller zu verbreiten, aber die Kehrseite ist, dass viele Teams leiden, weil ihnen die technischen Methoden fehlen, die nachhaltige agile Entwicklung ermöglichen.

Ich werde einige der nützlicheren XP-Methoden vorstellen und wie sie in unserer täglichen Arbeit angewendet werden. Nicht allen unseren Teams ist es gelungen, alle Methoden zu einzusetzen, aber im Großen und Ganzen haben wir mit den meisten Aspekten der XP/Scrum-Kombination experimentiert. Einige XP-Methoden adressieren Scrum direkt und können als überlappend betrachtet werden, zum Beispiel "das ganze Team", "zusammen sitzen", "Storys" und "Planungsspiel". In diesen Fällen bleiben wir einfach bei Scrum.

Pair Programming

Wir haben kürzlich in einem unserer Teams begonnen, das auszuprobieren. Funktioniert ganz gut, wirklich. Die meisten unserer Teams programmieren immer noch nicht viel im Zweierteam, aber ich bin, nachdem wir das tatsächlich in einem Team einige Sprints lang ausprobiert haben, inspiriert. Ich will versuchen, mehr Teams dahingehend zu coachen, es zu probieren.

Einige Punkte zu Pair Programming:

- Pair Programming verbessert die Codequalität.
- Pair Programming verbessert den Teamfokus (zum Beispiel wenn der Typ hinter Dir sagt, "Hey, ist das Zeug wirklich notwendig für den Sprint?").
- Überraschenderweise sind viele Entwickler strikt gegen Pair Programming, ohne es tatsächlich ausprobiert zu haben und lernen schnell, es zu mögen, wenn sie es einmal versucht haben.
- Pair Programming ist anstrengend und sollte nicht den ganzen Tag lang praktiziert werden.
- Die Zweierteams regelmäßig zu wechseln ist gut.
- Pair Programming verbessert die Wissensverteilung innerhalb der Gruppe. Und das auch überraschend schnell.
- Einige Leute fühlen sich bei Pair Programming einfach unwohl. Schmeiß keinen exzellenten Programmierer raus, nur weil er sich mit Pair Programming nicht wohl fühlt.
- Codereviews sind als Alternative zum Pair Programming OK.
- Der "Navigator" (der Typ, der die Tastatur nicht benutzt), sollte auch einen eigenen Computer haben. Nicht für die Entwicklung, sondern um kleine Stacheln anzubringen, wo nötig, um die Dokumentation zu durchstöbern, wenn der "Fahrer" (der Typ an der Tastatur) stockt etc.
- Zwinge den Leuten Pair Programming nicht auf. Ermutige Leute dazu und stelle die richtigen Werkzeuge bereit, aber lass sie nach ihrem eigenen Rhythmus experimentieren.

Testgetriebene Entwicklung (TDD[17])

Amen! Dies ist – für mich – wichtiger als beides, Scrum und XP. Du kannst mein Haus und meinen Fernseher und meinen Hund haben, aber versuche nicht, mich davon abzuhalten, TDD zu machen! Wenn Du TDD nicht magst, lass mich nicht in Dein Gebäude rein, weil ich versuchen werde, auf dem einen oder anderen Weg hineinzuschleichen. :o)

> OK, ich bin hier nicht mehr so religiös. Ich habe gemerkt, dass TDD einen ziemliche Nischentechnik ist, die zu meistern sehr wenige Leute die Geduld haben. Statt dessen unterrichte ich die Techniken und lass die Teams dann entscheiden, wie viel davon sie tun und wann.

[17] TDD = test-driven developemnt

Hier ist eine 10-Sekunden-Zusammenfassung von TDD:

Testgetriebene Entwicklung bedeutet, dass Du einen automatisierten Test schreibst, dann
schreibst Du gerade soviel Code, dass dieser den Test schafft, dann überarbeitest Du den Code im Wesentlichen daraufhin, die Lesbarkeit zu verbessern, und entfernst Doppelungen. Durchspülen und wiederholen.

Einige Reflektionen über die testgetriebene Entwicklung:

- TDD ist *schwer*. Es erfordert Zeit von einem Programmierer, um es zu *erfassen*. In der Tat ist es in vielen Fällen wirklich unerheblich, wie viel Du erklärst und coachst und demonstrierst – in vielen Fällen ist der einzige Weg für einen Programmierer, es zu *erfassen*, ihn mit jemandem Pair Programming machen zu lassen, der gut in TDD ist. Sobald ein Programmierer es jedoch *erfasst hat*, wird er üblicherweise ernstlich infiziert sein und wird nie wieder auf eine andere Weise arbeiten wollen.
- TDD hat eine tiefgreifende positive Wirkung auf das Systemdesign.
- Es braucht Zeit, TDD aufzubauen und wirkungsvoll bei einem neuen Produkt ans Laufen zu bringen, besonders bei Black-Box-Integrationstests, aber der Return on Invest ist *schnell*.
- Stelle sicher, dass Du die notwendige Zeit investierst, es *leicht* zu machen, Tests zu schreiben. Das bedeutet, die richtigen Werkzeuge zu besorgen, Leute auszubilden, die richtigen Hilfsklassen oder Basisklassen bereitzustellen etc.

Da TDD so schwierig ist, versuche nicht, es Leuten aufzuzwingen, statt dessen trainiere ich diese Prinzipien:

1) Stelle sicher, dass jedes Schlüsselfeature mindestens einen Ende-zu-Ende-Akzeptanztest hat, wobei die Interaktion über die GUI oder kurz dahinter startet.
2) Stelle sicher, dass jeder komplexe oder geschäftskritische Code durch Unit-Tests abgedeckt ist.
3) Dies wird einigen Code nicht abdecken. Das ist okay. Aber sei Dir bewusst, welcher Code nicht abgedeckt ist; stelle sicher, dass es eine bewusste Auslassung ist und keine Nachlässigkeit.
4) Schreibe die Tests während Du fortschreitest, verschiebe sie nicht auf später (Du wirst später nur genauso beschäftigt sein wie jetzt).

Das scheint genug Nachhaltigkeit zu geben, ohne die Einschränkung von komplettem TDD. Testabdeckung liegt normalerweise bei um die 70 %, weil das Gesetz vom abnehmenden Ertragszuwachs eintritt. Kurz, Testautomatisierung ist lebenswichtig, aber TDD ist optional.

Wir nutzen die folgenden Werkzeuge für die testgetriebene Entwicklung:

- jUnit/httpUnit/WebUnit. Wir ziehen TestNG und Selenium in Erwägung.
- HSQLDB als eingebettete In-Memory-DB für Testzwecke.
- Jetty als eingebetteter In-Memory-Web-Container für Testzwecke.
- Cobertura für Testabdeckungsmetriken.
- Framework Spring, um verschiedene Typen von Testgerüsten zu verdrahten (mit Attrappen[18], ohne Attrappen, mit externer Datenbank, mit In-Memory-Datenbank etc.)

In den meisten ausgereiften Produkten (aus einer TDD-Perspektive) haben wir automatisierte Black-Box-Akzeptanztests. Diese Tests nehmen das ganze gespeicherte System in Betrieb, inklusive Datenbanken und Webserver, und greifen auf das System zu, indem sie nur seine öffentlichen Schnittstellen nutzen (zum Beispiel HTTP).

Diese Art zu arbeiten ist jetzt einfacher als sie war, wegen all der cleveren Testtools und Frameworks, die zur Verfügung stehen. Sehr nützlich, egal ob Du TDD machst oder nicht.

Dies sorgt für extrem schnelle Entwicklung-Build-Test-Zyklen. Dies dient außerdem als Sicherheitsnetz, da es den Entwicklern genug Zutrauen gibt, oft zu refaktorieren, was bedeutet, dass das Design sauber und simpel bleibt, auch wenn das System wächst.

[18] Mock-Ups von engl. mocks (Anm. der Übersetzerin)

TDD bei neuem Code

Wir machen TDD bei jeder neuen Entwicklung, auch wenn das bedeutet, dass das initiale Projekteinrichten länger dauert (da wir mehr Tools brauchen und Testgerüste unterstützen müssen etc.). Das ist schon eher eine Kleinigkeit; der Benefit ist so bedeutend, dass es eigentlich keine Entschuldigung gibt, TDD *nicht* einzusetzen.

TDD bei altem Code

TDD ist schwierig, aber zu versuchen, TDD auf einer Codebasis einzuführen, die nicht von vornherein für TDD gebaut wurde… das ist *wirklich schwierig*! Warum? Naja, gewissermaßen…, ich könnte viele Seiten zu diesem Thema schreiben, also denke ich, ich höre hier auf. Ich spare mir das für meine nächste Publikation "TDD Feldbericht" auf. :o)

> Hab nie mit dem Schreiben davon angefangen… Es gibt heute ja keinen Mangel an TDD-Büchern! Und ein großartiges über Codealtlasten namens *Working Effectively with Legacy Code* von Michael Feathers ist ein echter Klassiker. Ich habe auch einige Artikel zu technischen Schulden geschrieben, steht in meinem Blog.
>
> http://blog.crisp.se/tag/technical-debt

Wir haben ziemlich viel Zeit damit verbracht zu versuchen, Integrationstests in einem unserer komplexeren Systeme zu automatisieren, eine Codebasis, die schon eine Weile existierte und in einem ernstlich unaufgeräumten Zustand und völlig ohne Tests war.

Für jedes Release des Systems hatten wir ein Team an Testern bestimmt, die ein Bündel an komplexen, immer wiederkehrenden Tests sowie Performanztests durchführten. Die wiederkehrenden Tests bestanden größtenteils aus manueller Arbeit. Dies verlangsamte unsere Entwicklung und unseren Releasezyklus ganz wesentlich. Unser Ziel war, diese Tests zu automatisieren. Allerdings waren wir, nachdem wir uns die Köpfe ein paar Monate lang daran gestoßen hatten, nicht wirklich näher an einer Lösung.

Danach änderten wir unseren Ansatz. Wir zogen den Schluss, dass wir die Tatsache, dass wir manuell wiederholte Tests machen mussten, nicht loswerden würden, und begannen uns statt dessen zu fragen, "Wie können wir den manuellen Testprozess weniger zeitaufwendig gestalten?" Bei dem System handelte es sich um ein Spiel und wir merkten, dass viel von der Zeit des Testteams für ziemlich triviale Setup-Aufgaben draufging, wie durch das Backoffice zu stöbern, um Turniere für Testzwecke einzurichten, oder durch das Warten auf den Start eines angesetzten Turniers. Also erstellten wir dafür Hilfsprogramme. Kleine, einfach zu erreichende Abkürzungen und Skripte, die all die Routinearbeit übernahmen und dafür sorgten, dass die Tester sich auf das eigentliche Testen konzentrieren konnten.

Dieser Aufwand zahlte sich wirklich aus! In der Tat, das ist wahrscheinlich das, was wir von Anfang an hätten tun sollen. Wir waren zu begierig darauf, das Testen zu automatisieren, dass wir vergaßen, Schritt für Schritt vorzugehen, wobei der erste Schritt war, Zeugs zu bauen, dass das *manuelle* Testen effizienter macht.

Lessons learned: Wenn Du dabei feststeckst, dass Du wiederholt manuell testen musst und dies auf solche Weise automatisieren möchtest, mach's nicht (es sei denn, es ist wirklich einfach). Statt dessen baue Zeugs, das manuelles wiederholtes Testen einfacher macht. *Dann* überlege, das eigentliche Testen zu automatisieren.

Nach-und-nach-Design

Das bedeutet, das Design von Anfang an einfach zu halten und es kontinuierlich zu verbessern, statt zu versuchen, alles sofort vom Start weg richtig zu designen und das Design dann einzufrieren.

Wir fahren recht gut damit, d. h. wir wenden eine vernünftige Menge Zeit darauf, existierendes Design zu überarbeiten und zu verbessern, und wir verwenden kaum Zeit darauf, umfangreiche Designs im Voraus zu machen. Manchmal bauen wir damit Mist, natürlich – zum Beispiel, indem wir einem heiklen Design erlauben, sich zu sehr "reinzubohren", so dass die Überarbeitung ein großes Projekt wird. Aber im Großen und Ganzen sind wir recht zufrieden damit.

Kontinuierliches Designverbessern ist meist automatisch ein Seiteneffekt davon, wenn man TDD betreibt.

Kontinuierliche Integration

Die meisten unserer Produkte haben ein recht ausgereiftes kontinuierliches Integrationssetup auf der Basis von Maven und QuickBuild. Das ist extrem wertvoll und zeitsparend. Es ist die ultimative Lösung zum guten alten "hey, aber es läuft doch auf *meiner* Maschine"-Punkt. Unser Server für den kontinuierlichen Build agiert als "Richter" oder Referenzpunkt, von dem aus die Gesundheit aller Codebasen bestimmt wird. Jedes Mal, wenn jemand etwas ins Ver-

sionssystem eincheckt, wird der Server für den kontinuierlichen Build wach, baut alles von Grund auf auf einem gemeinsam genutzten Server auf und lässt alle Tests laufen. Wenn irgend etwas schief geht, wird er eine Emailbenachrichtigung an das gesamte Team schicken, dass der Build fehlgeschlagen ist, inklusive einer Info darüber, exakt welcher geänderte Code den Build zerbrochen hat, mit einem Link zu Testberichten etc.

Jede Nacht baut der Server für den kontinuierlichen Build das Produkt von Grund auf neu auf und veröffentlicht ausführbare Dateien (Ohren, Kriege etc.), Dokumentation, Testberichte, Testabdeckungsberichte, Abhängigkeitsberichte etc. auf unserem internen Dokumentationsportal. Einige Produkte werden außerdem automatisch in die Testumgebung ausgeliefert.

Das aufzusetzen, war *eine Menge Arbeit*, aber jede Minute war es wert.

> Wenn Du das nur ein bisschen weiter denkst, landest Du bei kontinuierlicher Auslieferung. Jede Zusage ist ein Releasekandidat und ein Release zu veröffentlichen ist eine Ein-Klick-Operation. Ich sehe mehr und mehr Teams, die das praktizieren, und ich mache es in all meinen eigenen Projekten. Es ist erstaunlich effektiv und cool! Ich schlage vor, Du liest (oder durchstöberst wenigstens) das Buch *Continuous Delivery*. Das Setup ist eine Menge Arbeit, aber es ist es definitiv wert, dies am Anfang jedes neuen Produkts zu tun. Es zahlt sich fast immer sofort aus. Und es gibt heutzutage großartige Toolunterstützung dafür.

Gemeinschaftliches Code-Ownership

Wir ermutigen zum gemeinschaftlichen Code-Ownership (oder Code-Eigentum), aber das haben bisher nicht alle Teams übernommen. Wir haben herausgefunden, dass Pair Programming mit regelmäßigem Wechsel in den Zweierteams automatisch zu einem höheren Level an gemeinschaftlichem Code-Ownership führt. Teams mit einem hohen Level an gemeinschaftlichen Code-Ownership haben sich als sehr robust heraus gestellt – zum Beispiel stirbt ihr Sprint nicht einfach, weil eine Schlüsselperson krank ist.

> Spotify (und viele andere sich schnell bewegende Unternehmen) haben ein "internes Open-Source"-Modell. Alle Codes leben in einem internen GitHub und die Leute können Repositorys und Issue-Pull-Requests klonen wie in jedem anderen öffentlichen Open-Source-Projekt. Sehr komfortabel.

Informative Arbeitsumgebung

Alle Teams haben Zugang zu Whiteboards und leerem Wandplatz und nutzen das auch gut. In den meisten Räumen wirst Du die Wände mit allen Arten von Information über das Produkt und Projekt zugepflastert finden. Das größte Problem ist alter Müll, der sich an den Wänden nach und nach ansammelt – wir könnten dafür eine "Haushälter"-Rolle pro Team einführen.

Wir ermutigen den Einsatz einer Aufgabentafel, aber nicht alle Teams haben dies bisher übernommen. Siehe Seite 55 "Wie wir den Teamraum gestalten".

Code-Standards

Kürzlich haben wir damit begonnen, einen Code-Standard zu definieren. Sehr nützlich, ich wünschte, wir hätten das früher gemacht. Es erfordert fast überhaupt keine Zeit, fang einfach simpel an und lass ihn wachsen. Schreib nur Sachen auf, die nicht für jeden offensichtlich sind und verlinke zu existierendem Material, wo immer dies möglich ist.

Die meisten Programmierer haben ihren eigenen ausgeprägten Programmierstil. Kleine Details wie die Art und Weise, wie sie Exceptions handhaben, wie sie Code kommentieren, wann sie NULL zurückgeben etc. In einigen Fällen sind die Unterschiede egal; in anderen Fällen kann das zu ernsthaftem inkonsistenten Systemdesign und schwer zu lesendem Code führen. Ein Code-Standard ist hier sehr nützlich, so lange Du Dich auf die Sachen konzentrierst, die von Bedeutung sind.

Hier sind einige Beispiele aus unserem Code-Standard:

Du kannst die Regeln brechen, aber stell sicher, dass es dafür einen guten Grund gibt und dokumentier das.

Standardmäßig nutze die Sun-Code-Conventions: http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html

Niemals, nie, nie Exceptions abfangen, ohne den Stacktrace zu loggen oder rethrowing.log.debug() zu verwenden, verlier einfach nicht die Spur.

Nutze setter-basierte Dependency-Injection, um Klassen voneinander zu entkoppeln (außer natürlich, wenn enge Kopplung gewünscht ist).

Vermeide Abkürzungen. Allgemein bekannte Abkürzungen wie DAO sind okay.

Methoden, die Collections oder Arrays zurückgeben, sollten nicht NULL zurückgeben. Gib statt dessen leere Collections und Arrays zurück.

> Code-Standards und Styleguides sind super. Aber es ist nicht notwendig, dass Rad nochmal zu erfinden - Du kannst dies mit Hilfe meines Freundes Google kopieren:
>
> http://google-styleguide.googlecode.com

Verträgliches Tempo / Arbeit, die mit Energie versorgt

Viele Bücher über agile Softwareentwicklung behaupten, dass ausgedehnte Überstundenkonten für die Softwareentwicklung kontraproduktiv sind.

Nach einigem unwilligen Damit-Herumexperimentieren kann ich dem nur von ganzem Herzen zustimmen!

Vor ungefähr einem Jahr arbeitete eins unserer Teams (das größte) mit wahnsinnigen Mengen an Überstunden. Die Qualität der existierenden Codebasis war bejammernswert, und sie mussten die meiste Zeit als Feuerwehr unterwegs sein. Das Testteam (das auch Überstunden machte) hatte keine Möglichkeit, irgendeine ernsthafte Qualitätssicherung durchzuführen. Unsere Anwender waren verärgert und die Boulevardzeitungen fraßen uns bei lebendigem Leib.

Nach ein paar Monaten hatten wir geschafft, die Arbeitsstunden der Leute auf ein annehmbares Maß herunterzuschrauben. Die Leute arbeiteten die normale Anzahl an Stunden (außer manchmal während Projektkrisen). Und, Überraschung, die Produktivität und Qualität stieg merklich an.

Natürlich war die Reduzierung der Arbeitsstunden beileibe nicht der *einzige* Aspekt, der zu der Verbesserung führte, aber wir waren alle überzeugt, dass er einen großen Teil davon ausmachte.

> Ich sehe mir das immer und immer wieder an. In der Softwareentwicklung (und in anderen komplexen, kreativen Arbeitsbereichen) gibt es wenig Zusammenhang zwischen aufgewendeten Arbeitsstunden und dem gelieferten Wert. Was zählt, sind die Stunden, die man *konzentriert* und *motiviert* arbeitet. Die meisten von uns kennen das Gefühl, wenn man Samstagmorgen reinkommt und in Ruhe und Frieden für ein paar Stunden arbeitet und in etwa die Arbeit einer ganzen Woche in dieser Zeit fertig bekommt! Das ist es, was ich meine, wenn ich von *konzentrierter, motivierter* Arbeitszeit spreche. Also zwing Leute nicht zu Überstunden, außer in den seltenen Ausnahmefällen, wenn es *wirklich* eine kurze Zeit lang notwendig ist. Plus: Leute ausbrennen zu lassen ist BÖSE.

TEIL VIERZEHN

Wie wir testen

Das ist der schwierigste Teil. Ich bin nicht sicher, ob es der schwierigste Teil von Scrum ist oder einfach der schwierigste Teil von Softwareentwicklung im Allgemeinen.

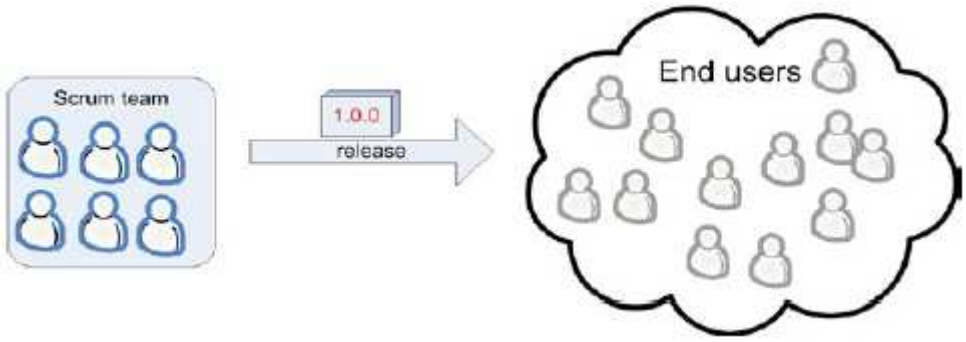
Testen ist der Teil, der in unterschiedlichen Organisationen wahrscheinlich am meisten variiert. Abhängig davon, wie viele Tester Du hast, wie viel Testautomatisierung Du hast, welche Art von System Du hast (nur Server und Webapplikationen oder machst Du tatsächlich ein Softwareprodukt, das Du im Karton verkaufst?), Größe der Releasezyklen, wie kritisch die Software ist (Blogserver vs. Flugkontrollsystem) etc.

Wir haben ziemlich viel herumexperimentiert, wie man in Scrum testet. Ich versuche zu beschreiben, was wir gemacht haben und was wir bisher gelernt haben.

Du wirst die Akzeptanztestphase wahrscheinlich nicht los

In der idealen Scrumwelt mündet ein Sprint in einer potenziell auslieferbaren Version Deines Systems. Dann liefer es einfach aus, oder?

> Exakt!



Falsch.

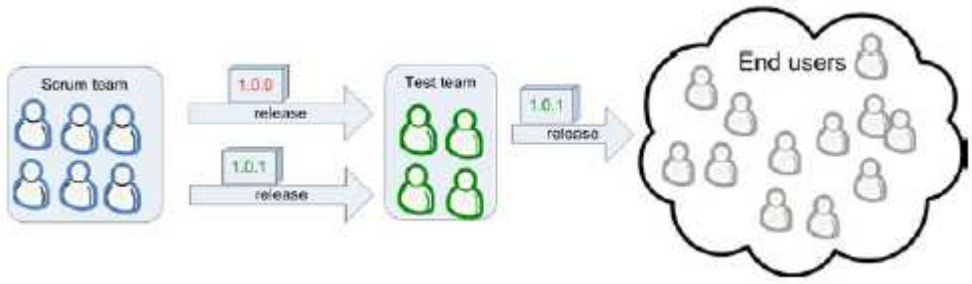
> Hä?

Unsere Erfahrung ist, dass das wahrscheinlich nicht funktioniert. Es wird fiese Bugs geben. Wenn die Qualität irgendeinen Wert für Dich hat, ist eine Phase mit irgendeiner Art von manuellem Akzeptanztest erforderlich.

> Absoluter Müll! Ich kann nicht glauben, dass ich das geschrieben habe! Und dann verbreitete sich das Buch viral und Leute aus aller Welt lasen es und glaubten meinen Worten. Schande über mich! Böööööööööser Autor, mach das nicht nochmal! *klaps*

> Ja, manuelles Testen ist wichtig und zu einem gewissen Ausmaß unvermeidlich. Aber es sollte *im Sprint durch das Team* erfolgen, nicht an irgendeine separate Gruppe weitergereicht werden oder für eine spätere Testphase aufgespart werden. Das war's, warum wir dem Wasserfallmodell den Laufpass gegeben haben, erinnerst Du Dich?

Das ist der Zeitpunkt, wenn festgelegte Tester, die *nicht* Teil des Teams sind, mit der Art von Tests auf das System einhämmern, auf die Scrumteams nicht kommen oder für die sie keine Zeit haben oder für die sie die Hardware nicht haben. Die Tester haben exakt denselben Zugriff auf das System, wie ihn schließlich die Anwender haben, was bedeutet, sie müssen das manuell tun (vorausgesetzt, Dein System ist für menschliche Anwender gemacht).



Das Testteam wird Fehler finden, das Scrumteam wird Releases mit behobenen Fehlern bauen, und früher oder später (hoffentlich früher) wirst Du in der Lage sein, eine fehlerbereinigte Version 1.0.1 an die Anwender auszuliefern statt eine wackelige Version 1.0.0.

Wenn ich "Akzeptanztestphase" sage, beziehe ich mich auf die gesamte Periode des Testens, Debuggens und nochmaligen Releasebauens, bis es eine Version gibt, die gut genug für das produktive Release ist.

Die Akzeptanztestphase minimieren

Die Akzeptanztestphase tut weh. Sie fühlt sich deutlich un-agil an.

> Exakt! Also mach's nicht.
>
> Naja, OK, ich weiß. In mancher Umgebung mag das unvermeidlich erscheinen. Aber mein Punkt ist, dass ich dachte, das wäre unvermeidlich. Aber jetzt habe ich beobachtet, wie wirklich agile Unternehmen sich schnell bewegen und die Qualität erhöhen, indem sie die separate Akzeptanztestphase abschafften und diese Arbeit in den Sprint integriert haben. Wenn Du also denkst, etwas wäre unvermeidlich, könnte es sein, dass Du durch Deinen Status Quo blind bist (wie ich es war). Nichtsdestotrotz liefert dieses Kapitel einige nützliche Muster dafür, wie man mit separaten Akzeptanztests umgeht, als vorübergehende Maßnahme, bis Du es schaffst, das alles in einen Sprint zu integrieren. :o)

Obwohl wir diese Phase nicht loswerden, können wir (und wir tun's) versuchen, sie zu minimieren. Konkreter, minimiere den Aufwand an *Zeit*, der für die Akzeptanztestphase benötigt wird. Dies wird erreicht durch:

- Die Qualität des durch das Scrumteam ausgelieferten Codes maximieren und
- die Effizienz des manuellen Testens maximieren (d. h. finde die besten Tester, gib ihnen die besten Tools, stell sicher, dass sie zeitaufwendige Aufgaben melden, die automatisiert werden könnten).

Also, wie maximieren wir die Qualität des Codes, der vom Scrumteam geliefert wird? Naja, es gibt viele Möglichkeiten. Hier sind zwei, die wir gefunden haben und die gut funktionieren:

Nimm Tester ins Scrumteam hinein.

Mach weniger pro Sprint.

Qualität verbessern, indem Tester im Scrumteam sind

Ja, ich höre beide Einwände:

- "Aber das ist offensichtlich! Scrumteams sollen *funktionsübergreifend* sein!"
- "Scrumteams sollen keine Rollen haben! Wir können nicht einen Typen drin haben, der *nur* Tester ist!"

Lass es mich klarstellen. Was ich in diesem Fall mit "Tester" meine, ist eher "ein Typ, dessen Haupttalent das Testen ist" als "ein Typ, dessen einzige Rolle im Testen besteht".

> Dies ist ein wichtiger Punkt. "Funktionsübergreifend" bedeutet nicht, dass jeder alles weiß. Es bedeutet nur, dass jeder gewillt ist, mehr als nur sein eigenes Ding zu machen. Wir möchten einen netten Mix an Spezialisierung und Cross-Funktionalität. Damit ist das gesamte Team gemeinsam für die Qualität seines Produkts verantwortlich und absolut jeder wird am Testen beteiligt sein. Der Tester jedoch wird diese Arbeit leiten, im Zweierteam mit einem Entwickler an Testautomatisierung arbeiten und persönlich die mehr komplexen manuellen Tests durchführen.

Entwickler sind oft lausige Tester. *Besonders* Entwickler, die ihren eigenen Code testen.

Der Tester ist der "Freigabe-Mensch"

Zusätzlich dazu, dass er "einfach" ein Teammitglied ist, hat der Tester eine wichtige Aufgabe. Er ist der Freigabe-Mensch. Nichts wird in einem Sprint als "fertiggestellt" betrachtet, bis *er* sagt, dass es fertiggestellt ist. Ich weiß aus Erfahrung, dass Entwickler oft sagen, dass etwas fertiggestellt ist, wenn es das eigentlich noch nicht ist. Sogar wenn Du eine klare Definition von "fertiggestellt" hast (was Du wirklich haben solltest, siehe Seite 35 "Definition von *Fertig* – Definition of *Done*"), werden Entwickler das öfters vergessen. Wir Programmierer sind ungeduldige Leute und möchten ASAP zum nächsten Punkt kommen.

Also, woher weiß nun Mr. T (unser Tester), dass etwas fertiggestellt ist? Naja, zuerst einmal sollte er (Überraschung) es *testen*! In vielen Fällen stellt sich heraus, dass ein Entwickler etwas als "fertig" ansieht, dass noch nicht einmal *getestet werden konnte*! Weil es nicht eingecheckt war oder weil es nicht auf dem Testserver eingespielt war oder nicht gestartet werden konnte oder was auch immer. Sobald Mr. T das Feature getestet hat, sollte er mit dem Entwickler die "Fertiggestellt"-Checkliste (wenn Ihr eine habt) durchgehen. Zum Beispiel… wenn die Definition für fertiggestellt vorschreibt, dass es eine Releaseinformation geben sollte, dann checkt Mr. T, dass es eine Information für die Neuerungen im Release gibt. Wenn es eine Art von formaler Spezifikation für das Feature gibt (in unserm Fall selten), dann checkt Mr. T das ebenfalls, und so weiter.

Ein netter Nebeneffekt davon ist, dass das Team jetzt einen Typen hat, der perfekt für das Organisieren der Sprintdemo geeignet ist.

> Ich bin kein großer Fan mehr vom Freigabe-Mensch-Modell. Es führt einen Flaschenhals ein und bringt zuviel Verantwortlichkeit in die Hand einer einzigen Person. Aber ich kann verstehen, dass es unter bestimmten Umständen nützlich ist (es war sicherlich damals nützlich). Außerdem, wenn irgendwer sich für die Qualität verantwortlich fühlt, sollte es ein echter Anwender sein.

Was tut der Tester, wenn es nicht zu testen gibt?

Diese Frage kommt immer wieder auf. Mr. T: "Hey, Scrum Master, es gibt momentan nichts zu testen, was *soll ich* tun?" Es mag noch eine Woche dauern, bis das Team die erste Story abarbeitet, also was sollte der Tester während *dieser* Zeit tun?

Naja, zuerst einmal sollte er *sich auf die Tests vorbereiten*. Das heißt, Testspezifikationen schreiben, eine Testumgebung vorbereiten etc. Dadurch sollte es, wenn ein Entwickler etwas hat, das bereit ist für den Test, keine Verzögerung geben, Mr. T sollte sofort eintauchen und mit dem Test beginnen.

Wenn das Team TDD einsetzt, gibt es Leute, die von Tag eins an die Zeit damit verbringen, Testcode zu schreiben. Der Tester sollte mit den Entwicklern, die Testcode schreiben, per Pair Programming zusammenarbeiten. Wenn der Tester überhaupt nicht programmieren kann, sollte er trotzdem mit den Entwicklern per Pair Programming zusammenarbeiten, außer dass er nur navigiert und dem Entwickler das Tippen überlässt. Ein guter Tester kommt normalerweise mit anderen Typen von Tests um die Ecke als ein guter Entwickler, so können sie einander ergänzen.

Wenn das Team TDD nicht einsetzt oder wenn es nicht genug Testfallbeschreibung gibt, um die Zeit des Testers auszufüllen, sollte er einfach tun, was immer er kann, um dem Team zu helfen, das Sprintziel zu erreichen. Genau wie jedes andere Teammitglied. Wenn der Tester programmieren kann, dass ist das super. Wenn nicht, wird Dein Team alle Nicht-Programmier-Aufgaben identifizieren müssen, die im Sprint erledigt werden müssen.

Wenn man während des Sprintplanungstreffens Storys in Aufgaben aufteilt, tendiert das Team dazu, seine Aufmerksamkeit auf *Programmieraufgaben* zu richten. Normalerweise jedoch gibt es jede Menge *Nicht-Programmier-Aufgaben*, die im Sprint erledigt werden müssen. Wenn Du Dich damit beschäftigst, die Nicht-Programmier-Aufgaben während der Sprintplanungsphase zu identifizieren, stehen die Chancen gut, dass Mr. T in der Lage sein wird, ziemlich viel beizutragen, sogar wenn er nicht programmieren kann und es genau jetzt nichts zum Testen gibt.

Beispiele von Nicht-Programmier-Aufgaben, die oft in einem Sprint anfallen:

- Eine Testumgebung aufsetzen.
- Anforderungen klären.
- Auslieferungsdetails mit dem Team vom Betrieb besprechen.
- Auslieferungsdokumente schreiben (Releaseinformationen, RFC[19] oder was auch immer Deine Organisation hat).
- Kontakt aufnehmen mit externem Personal (zum Beispiel GUI-Designern).
- Buildskripte verbessern.
- Weitere Aufteilung von Storys in Aufgaben.
- Schlüsselfragen der Entwickler identifizieren und für die Beantwortung sorgen.

In der entgegengesetzten Situation, was tun wir, wenn Mr. T einen Flaschenhals darstellt? Sagen wir, wir stehen am letzten Tag des Sprints und plötzlich ist eine Menge an Sachen erledigt und Mr. T hat keine Chance, alles zu testen. Was tun wir dann? Naja, wir könnten jeden im Team zu Mr. Ts Assistenten machen. Er entscheidet, welche Sachen er selbst machen muss und delegiert das restliche Testen an den Rest vom Team. Das ist es, worum es bei funktionsübergreifenden Teams geht!

Also ja, Mr. T *hat* im Team eine spezielle Rolle, aber er darf doch andere Arbeiten machen und andere Teammitglieder dürfen auch mal seine Arbeit machen.

> Gut gesagt! (Es ist mir erlaubt, mir manchmal selbst zu gratulieren, OK?) Und das ist eine gute Art, wie man all die anderen Kompetenzen im Team sieht.

Qualität verbessern, indem man weniger pro Sprint tut

Dies geht zurück in das Sprintplanungstreffen. Schlicht gesagt, stopft nicht zu viele Storys in den Sprint! Wenn Ihr Qualitätsprobleme habt oder lange Akzeptanztestrunden, macht weniger pro Sprint! Dies wird fast automatisch zu einer höheren Qualität führen, zu kürzeren Akzeptanztestrunden, weniger Fehlern, die die Anwender tangieren und auf lange Sicht zu einer höheren Produktivität, da das Team sich die ganze Zeit auf die neuen Sachen konzentrieren kann statt altes Zeug zu reparieren, das ständig aufpoppt.

Es ist fast immer billiger, weniger zu bauen mit einem stabileren Build, als viele Sachen einzubauen und dann über die Hotfixes in Panik zu geraten.

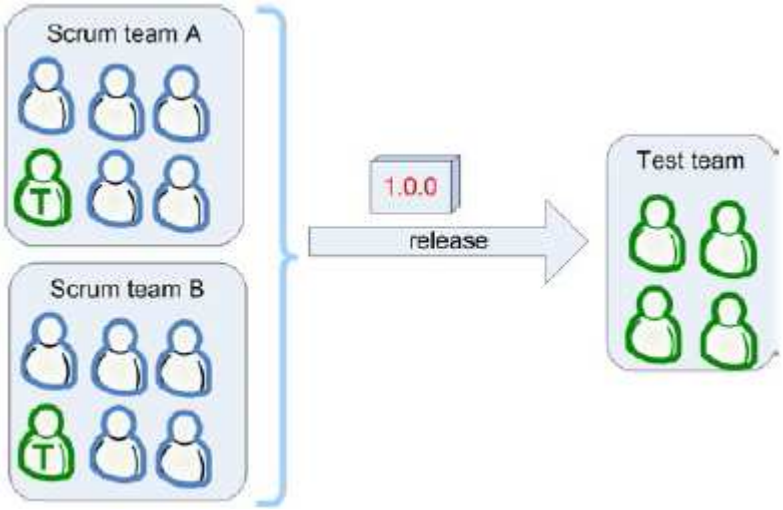
> Das ist einfach so komplett wahr! Ich sehe das immer und immer wieder bei Teams auf der ganzen Welt.

Sollten Akzeptanztests Teil des Sprints sein?

Hier sind wir sehr unschlüssig. Einige unserer Teams nehmen Akzeptanztests mit in den Sprint auf. Die meisten unserer Teams jedoch tun das nicht, aus zwei Gründen:

- Ein Sprint hat ein festes Zeitfenster. Akzeptanztests (nach meiner Definition inklusive Debugging und wiederholtem Release) sind sehr schwierig innerhalb eines festen Zeitfensters zu machen. Was, wenn die Zeit knapp wird und Du immer noch einen kritischen Fehler hast? Wirst Du ein System mit einem kritischen Fehler produktiv ausliefern? Wartest Du bis zum nächsten Sprint? In den meisten Fällen sind beide Lösungen nicht akzeptabel. Also lassen wir den manuellen Akzeptanztest draußen.
- Wenn Du mehrere Scrumteams hast, die parallel am selben Produkt arbeiten, muss der Akzeptanztest am kombinierten Ergebnis der Arbeit beider Teams erfolgen. Wenn beide Teams innerhalb des Sprints manuelle Akzeptanztests machen, bräuchtest Du immer noch ein Team, um das endgültige Release zu testen, in dem die Builds beider Teams zusammengefasst sind.

[19] https://de.wikipedia.org/wiki/Request_for_Comments

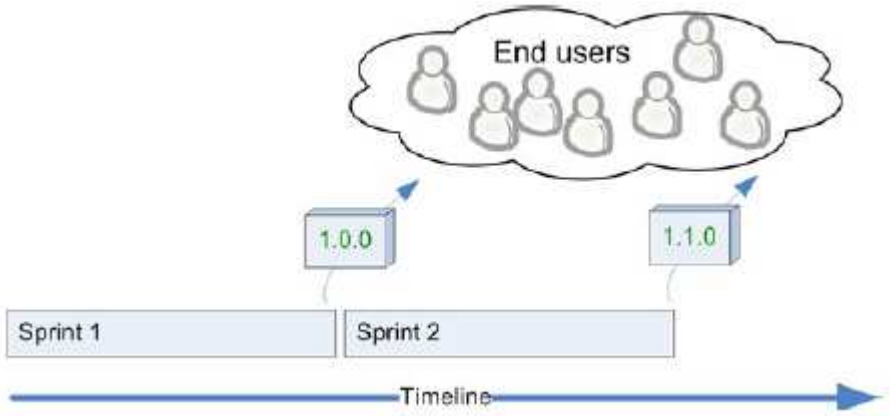


Das ist beileibe keine perfekte Lösung, aber in den meisten Fällen gut genug für uns.

> Nochmal, bemüh Dich, Akzeptanztests als Teil eines jeden Sprints zu machen. Es dauert eine Weile, bis man soweit ist, aber Du wirst es nicht bereuen. Sogar, wenn Du niemals ankommst, wird der Versuch Dir jede Menge Verbesserungen beim Vorgehen in Eurer Arbeit einbringen.

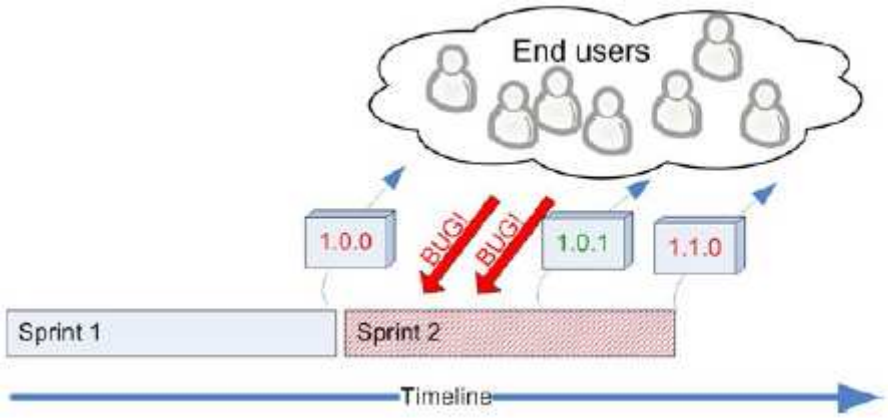
Sprintzyklen vs. Akzeptanztestrunden

In einer perfekten McScrum-Welt brauchst Du keine Akzeptanztestphase, weil jedes Scrumteam nach jedem Sprint eine neue produktionsreife Version Eures Systems abliefert.



> Das kann klappen! Ich habe Teams in der echten Welt gesehen, die jeden Tag produktiv ausliefern, manchmal mehrfach täglich. Wenn ein Scrumausbilder ihnen sagt, "Ihr müsst ein komplett getestetes, potenziell auslieferbares Produktinkrement am Ende jedes Sprints haben", ist ihre Reaktion, "Häh? Warum so lange warten?"
>
> Also nein, dafür du brauchst keine perfekte Scrumwelt. Krempel einfach Deine Ärmel hoch, finde heraus, was Dich davon abhält, einen releasefähigen Code in jedem Sprint zu bekommen, und behebe die Probleme eins nach dem anderen. Natürlich kann das mehr oder weniger schwierig sein, abhängig von Deinem Einsatzbereich, aber es ist doch einen Versuch wert. Starte einfach damit, was immer Dein Releasezyklus heute ist (ob er monatlich oder jährlich oder was auch immer ist) und kürze ihn schrittweise, aber kontinuierlich.

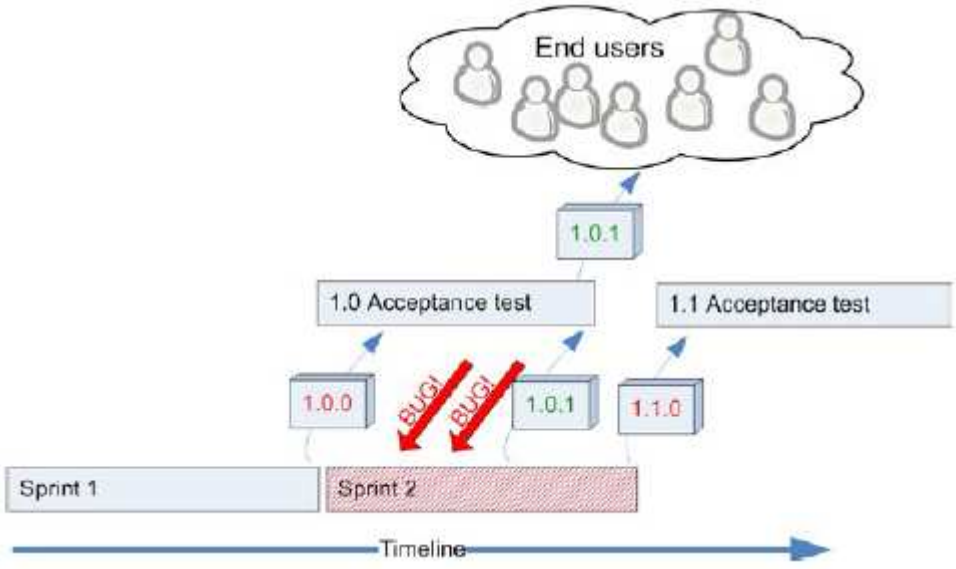
Naja, hier ist ein realistischeres Bild:



Nach Sprint 1 wird eine fehlerbehaftete Version 1.0.0 ausgeliefert. Während Sprint 2 beginnen Fehlerberichte hereinzuströmen und das Team verwendet die meiste Zeit aufs Debugging und ist gezwungen, ein fehlerbereinigtes Release 1.0.1 mitten im Sprint auszuliefern. Dann am Ende von Sprint 2 liefern sie eine Version 1.1.0 mit neuen Features aus, die natürlich noch fehlerbehafteter ist, da sie noch weniger Zeit hatten, es in dieser Zeit richtig zu machen, geschuldet all den Unterbrechungen vom letzten Release. Usw. usf.

Die diagonale rote Line in Sprint 2 symbolisiert Chaos.

Nicht allzu schön, oder? Naja, das Traurige ist, dass das Problem bestehen bleibt, sogar wenn Du ein Akzeptanztestteam hast. Der einzige Unterschied ist, dass die meisten Fehlerberichte vom Testteam statt vom verärgerten Anwender kommen werden. Das ist aus Geschäftssicht ein gewaltiger Unterschied, aber für Entwickler läuft es auf dasselbe heraus. Außer dass Tester normalerweise weniger aggressiv als Anwender sind. Normalerweise.



Anwender, Akzeptanztests, Fehler, Zeitlinie

Wir haben keine simple Lösung zu diesem Problem gefunden. Wir haben allerdings viel mit verschiedenen Modellen herumexperimentiert.

Vor allem, noch einmal, maximiere die Qualität des Codes, den das Scrumteam abliefert. Die Kosten, Fehler früh zu finden und zu beheben, innerhalb eines Sprints, sind einfach so extrem niedrig verglichen mit den Kosten, Fehler später zu finden und zu beheben.

Aber die Tatsache bleibt bestehen, sogar wenn wir die Anzahl an Fehlern minimieren können, es wird trotzdem immer Fehlerberichte geben, nachdem ein Sprint beendet ist. Wie gehen wir damit um?

Ansatz 1: "Beginne nicht damit, neue Sachen zu bauen, bevor die alten Sachen produktiv sind"

Klingt gut, oder? Bekommst Du auch so ein warmes, flauschiges Gefühl?

> Ja, das ist großartig!

Wir waren einige Male nah dran, diesen Ansatz einzusetzen und haben schicke Modelle davon gezeichnet, wie wir dies tun würden. Allerdings änderten wir jedes Mal unsere Meinung, wenn uns die Kehrseite bewusst wurde. Wir würden einen Zeitraum zwischen den Sprints ohne festes Zeitfenster hinzufügen, in der wir nur testen und debuggen, bis wir ein produktives Release erstellen können.

Sprints und Releases, Zeitlinie

> Nicht, wenn Deine Definition von Fertiggestellt "produktiv" heißt. In diesem Fall kannst Du den nächsten Sprint sofort starten, da der Code aus dem letzten Sprint bereits produktiv ist. Er wurde während des Sprints kontinuierlich ausgeliefert. Ein bisschen extrem, ja, aber man kann es machen.

Wir mochten die Vorstellung nicht, keine zeitbegrenzten Releasezeiträume zwischen den Sprints zu haben, hauptsächlich weil es den regelmäßigen Sprintherzschlag stören würde. Wir könnten nicht länger sagen, "alle drei Wochen beginnen wir einen neuen Sprint". Übrigens löst das das Problem auch nicht komplett. Selbst wenn wir feste Releasezeiträume haben, wird es dringende Fehlerberichte geben, die von Zeit zu Zeit hereinkommen, und wir müssten darauf vorbereitet sein, damit umzugehen.

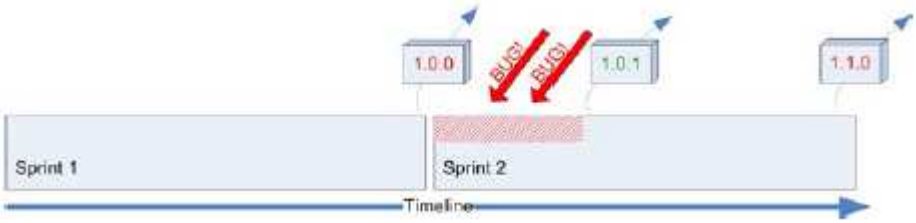
> Das ist tatsächlich wahr. Sogar wenn Du es hinbekommst, kontinuierlich auszuliefern, musst Du mit dringenden Fehlern umgehen, die hereinkommen. Weil das manchmal passieren wird. Kein Team ist *so* gut, dass es die nicht hätte. Und die beste Art, damit umzugehen, ist, ein bisschen Puffer im Sprint zuzulassen.

Ansatz 2: "OK, um mit neuen Sachen zu beginnen, aber sorge dafür, dass zuerst die alten Sachen ins Produktivrelease kommen"

Dies ist unser bevorzugter Ansatz. Wenigstens zum jetzigen Zeitpunkt.

Im Prinzip gehen wir von einem zum nächsten, wenn wir einen Sprint beenden. Aber wir rechnen damit, dass wir im nächsten Sprint einige Zeit darauf verwenden, Fehler aus dem letzten Sprint zu beheben. Wenn der nächste Sprint ernsthaft in Gefahr gerät, weil wir so viel Zeit für die Behebung der Fehler aus dem vorangegangenen Sprint brauchen, werten wir aus, warum dies passierte und wie wir die Qualität verbessern können. Wir stellen sicher, dass die Sprints lang genug sind, um eine angemessene Menge an Fehlerbehebung aus dem vorigen Sprint zu überleben.

Allmählich, nach einem Zeitraum von vielen Monaten, verringerte sich die Zeit, die wir mit Fehlerbehebung aus dem vorigen Sprint benötigten. Zusätzlich konnten wir weniger Leute beteiligen, wenn Fehler auftraten, so dass nicht jedes Mal das *gesamte* Team Unterbrechungen erfuhr. Jetzt sind wir auf einem akzeptableren Level.



Sprints, Fehler, Zeitlinie

Während der Sprintplanungstreffen setzen wir den Fokusfaktor niedrig genug an, um die Zeit, die wir für Fehlerbehebung aus dem letzten Sprint erwarten, zu berücksichtigen. Mit der Zeit wurden die Teams ziemlich gut darin, dies einzuschätzen. Die Umsetzungsgeschwindigkeitsmetrik hilft sehr (siehe Seite 27 "Wie entscheidet ein Team, welche Storys im Sprint enthalten sein sollen?").

Oder nutze einfach "Yesterday's Weather" – zieh nur so viele Storypunkte, wie Du im letzten Sprint oder im Durchschnitt über die letzten drei Sprints fertigstellen konntest. Dann wird Dein Sprint automatisch einen eingebauten Puffer haben, um mit Störungen und Hotfixes umzugehen. Du wirst automatisch die Arbeit auf die Kapazität begrenzen und als Ergebnis schneller vorwärtskommen (lies irgendein Buch über Lean oder Warteschlangentheorie, wenn Du überzeugt werden musst, dass Deinen Sprint zu überlasten eine schlechte Idee ist).

Schlechter Ansatz: "Fokussier Dich darauf, neue Sachen zu bauen"

Dies bedeutet soviel wie "fokussier dich darauf, *eher neue Sachen zu bauen, als bisherige Dinge produktiv zu machen*". Wer würde so vorgehen? Doch wir machten diesen Fehler anfangs ziemlich oft, und ich bin sicher, andere Firmen ebenso. Es ist eine stressbedingte Krankheit. Viele Manager verstehen im Grunde nicht, dass Du normalerweise noch weit weg vom Produktivrelease bist, auch wenn der ganze Code fertig ist. Zumindest bei komplexen Systemen. Also bittet der Manager (oder Product Owner) das Team, neue Sachen hinzuzunehmen, während die Last des alten, fast für's Release fertigen Codes schwerer und schwerer wiegt und alles verlangsamt.

Ich bin beständig erstaunt darüber, wie viele Firmen in diese Falle tappen. Alles, was sie tun müssen, ist, die Anzahl der Projekte oder Features, die parallel gemacht werden, zu begrenzen. Ich habe Fälle gesehen, in denen Firmen buchstäblich siebenmal schneller wurden, indem sie den Rat befolgten. Stell Dir das vor – genauso viel Sachen ausgeliefert, aber siebenmal schneller, ohne härter zu arbeiten oder Leute einzustellen. Und außerdem wegen der kürzeren Feedbackschleife in besserer Qualität. Verrückt, aber wahr.

Lauf nicht schneller als der Langsamste in Deiner Kette

Sagen wir, den Akzeptanztest zu machen ist die langsamste Stelle. Du hast zu wenige Tester oder die Dauer des Akzeptanztests ist wegen der düsteren Codequalität lang.

Sagen wir, Dein Akzeptanztestteam kann maximal drei Features pro Woche testen (nein, wir nutzen "Features pro Woche" nicht als Metrik; ich verwende das nur für dieses Beispiel). Und sagen wir, Deine Entwickler können sechs neue Features pro Woche entwickeln.

Manager und Product Owner werden versucht sein, die Entwicklung auf sechs neue Features pro Woche anzusetzen.

Macht's nicht! Die Realität wird Euch auf die eine oder andere Weise einholen, und das wird schmerzen.

Statt dessen, setzt drei neue Features pro Woche an und verwendet den Rest der Zeit darauf, den Flaschenhals beim Testen abzumildern. Zum Beispiel:

- Lasst ein paar Entwickler statt dessen als Tester arbeiten (oh, sie werden Dich dafür lieben…).
- Führt Tools und Skripte ein, die das Testen vereinfachen.
- Fügt mehr automatisierten Testcode ein.
- Erhöht die Sprintlänge und bezieht den Akzeptanztest mit ein.
- Definiert einige Sprints als "Testsprints", in denen das gesamte Team als Akzeptanztestteam arbeitet.
- Stellt mehr Tester ein (sogar, wenn das bedeutet, Entwickler abzuziehen).

Wir haben alle diese Lösungen ausprobiert (außer die letzte). Die langfristig besten Lösungen sind natürlich Punkt zwei und drei, d. h. bessere Tools und Skripte und Testautomatisierung.

Retrospektiven sind ein gutes Forum, die langsamste Stelle in der Kette zu identifizieren.

Akzeptanztests innerhalb des Sprints vorzusehen statt separat, wird sich als selbstjustierende Anpassung auswirken. Versuch es – nimm in Deine Definition von Fertiggstellt den Akzeptanztest mit auf und sieh, was über die Zeit passiert.

Zurück zur Realität

Ich habe Dir wahrscheinlich den Eindruck vermittelt, dass wir Tester in allen Scrumteams haben, dass wir mächtige Akzeptanztestteams für jedes Produkt haben, das wir nach jedem Sprint ausliefern etc. etc.

Naja, das haben wir nicht.

Wir haben *manchmal* geschafft, diese Sachen zu machen, und wir haben die positiven Auswirkungen gesehen. Aber wir sind noch weit von einem akzeptablen Qualitätssicherungsprozess entfernt, und wir haben da noch eine Menge zu lernen.

> In der Tat, wir hatten eine Menge zu lernen. :o)

TEIL FÜNFZEHN

Wir wir mit mehreren Scrumteams pro Produkt umgehen

Viele Dinge sind viel schwieriger, wenn Du mehrere Scrumteams hast, die am selben Produkt arbeiten. Das ist ein grundlegendes Problem, und es hat eigentlich nichts mit Scrum zu tun. Mehr Entwickler = mehr Komplikationen.

> Ich habe viel mit Skalierung gearbeitet, seitdem ich dieses Buch schrieb. Sieh *Dir Lean from the Trenches* an, um mehr darüber zu erfahren. Das Buch ist in einem ganz ähnlichen Stil wie dies hier. Es illustriert, wie ein 60-Personen-Regierungsprojekt unter Verwendung einer Kombination aus Scrum und Kanban und XP lief.
>
> https://pragprog.com/book/hklean/lean-from-the-trenches

Wir haben (wie üblich) damit experimentiert. Wir hatten ein Team von maximal ungefähr 40 Leuten, die am selben Produkt gearbeitet haben.

Die Schlüsselfragen sind:

- Wie viele Teams aufstellen?
- Wie die Leute den Teams zuordnen?

> Und, natürlich, wie die Teams miteinander synchronisieren.

Wie viele Teams aufstellen?

Wenn die Handhabung von mehreren Scrumteams so schwierig ist, warum scheren wir uns darum? Warum nehmen wir nicht alle in dasselbe Team?

Das größte einfache Scrumteam, das wir hatten, bestand aus etwa 11 Leuten. Es funktionierte, aber nicht allzu gut. Daily Scrums tendierten dazu, die 15 Minuten zu überziehen. Teammitglieder wussten nicht, was andere Teammitglieder gerade machten, so entstanden Irritationen. Für den Scrum Master war es schwierig, für alle die Ausrichtung auf das Ziel aufrecht zu halten, und es war schwierig, Zeit zu finden für alle Hindernisse, die ihm berichtet wurden.

Die Alternative ist, das Team in zwei Teams aufzuteilen. Aber ist das besser? Nicht notwendigerweise.

Wenn das Team erfahren ist und sich in Scrum zu Hause fühlt, und wenn es einen logischen Schnitt gibt, wo man den Plan in zwei verschiedene Pfade schneiden kann, und wenn man in diesen zwei Pfaden nicht am selben Quellcode arbeitet, dann würde ich sagen, es ist eine gute Idee, das Team aufzuteilen. Andernfalls würde ich erwägen, bei einem Team zu bleiben, trotz der Nachteile, die große Team haben.

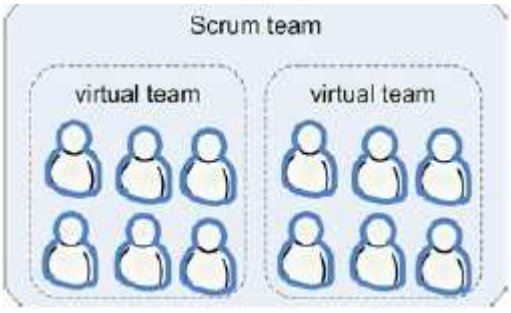
Meine Erfahrung ist, dass es besser ist, weniger Teams zu haben, die zu groß sind, als viele kleine Teams zu haben, die untereinander interagieren. Mach nur kleine Teams, wenn sie nicht miteinander interagieren müssen!

Virtuelle Teams

Woher weißt Du, ob Du die richtige oder falsche Entscheidung im Zielkonflikt "großes Team" vs. "viele Teams" getroffen hast?

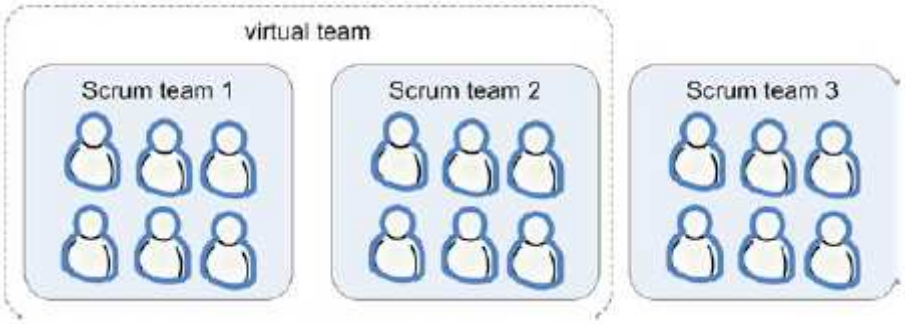
Wenn Du Deine Augen und Ohren offen hälst, könntest Du bemerken, wenn sich "virtuelle Teams" bilden.

Beispiel 1: Du entscheidest Dich für ein Team mit zahlreichen Mitgliedern. Aber als Du damit anfängst zu sehen, wer während des Sprints mit wem spricht, stellst Du fest, dass das Team sich effektiv in zwei Teilteams aufgespalten hat.



Beispiel 1

Beispiel 2: Du entscheidest Dich, drei kleinere Teams zu bilden. Aber als Du damit anfängst zu sehen, wer während des Sprints mit wem spricht, stellst Du fest, dass Team 1 und Team 2 die ganze Zeit miteinander reden, während Team 3 isoliert arbeitet.



Beispiel 2

Und was bedeutet das? Dass Deine Teamaufteilungsstrategie falsch war? Ja, wenn die virtuellen Teams als permanent erscheinen. Nein, wenn die virtuellen Teams zeitweise auftreten.

Sieh Dir nochmal Beispiel 1 an. Wenn die zwei virtuellen Teilteams dazu tendieren, sich hin und wieder zu verändern (d. h. Leute bewegen sich zwischen den virtuellen Teilteams), dann hast Du vermutlich die richtige Entscheidung getroffen, ein einzelnes Scrumteam zu organisieren. Wenn die zwei Teilteams während des gesamten Sprints dieselben bleiben, möchtest Du die Leute im nächsten Sprint wahrscheinlich direkt in zwei Scrumteams verteilen.

Jetzt sieh Dir Beispiel 2 noch einmal an. Wenn Mitglieder von Team 1 und Team 2 (aber nicht Team 3) während des gesamten Sprints miteinander reden, möchtest Du vermutlich Team 1 und Team 2 im nächsten Sprint zu einem Scrumteam zusammenfassen. Wenn Team 1 und Team 2 während der ersten Hälfte des Sprints viel miteinander reden und dann Team 1 und Team 3 während der zweiten Hälfte des Sprints viel miteinander reden, dann solltest Du überlegen, alle drei Teams in einem Team zusammenzufassen oder Du lässt es einfach bei drei Teams. Bring die Frage während der Sprintretrospektive zur Sprache und lass die Teams selbst entscheiden.

Die Einteilung in Teams ist einer der wirklich schwierigen Teile in Scrum. Überleg nicht zu gründlich und optimiere nicht zu verbissen. Experimentiere, behalte die virtuellen Teams im Auge und stelle sicher, Euch während Eurer Retrospektiven ausgiebig Zeit dafür zu geben, über diese Art von Dingen zu diskutieren. Früher oder später wirst Du die richtige Lösung für Eure jeweilige Situation finden. Das wichtige daran ist, dass die Teams sich wohlfühlen und nicht zu oft übereinander stolpern.

> Eine zunehmend populäre, fortgeschrittene Methode ist die, sich Teams dynamisch selbst bilden zu lassen, wie sie dies brauchen. Ich nenne das manchmal "Super-Team"-Pattern oder "dynamische Teilteams". Es funktioniert am besten, wenn Du 12 bis 16 Leute hast, die nah beieinander sitzen, einander gut kennen und am selben Produkt arbeiten. Gib ihnen ein einziges, gemeinsam genutztes Produkt-Backlog und lass die Leute kleinere Grüppchen rund um die einzelnen Storys bilden ("Pat und Tom und ich werden jetzt an Story X arbeiten") und lass sie sich dynamisch umgruppieren, wenn die Storys fertiggestellt sind. Etwas Moderation ist notwendig, damit dies in der Praxis funktioniert, aber es ist perfekt für den Fall, dass Du kein einzelnes großes Team haben möchtest und doch keinen passenden Weg finden kannst, die Leute in kleinere Teams zu gruppieren.

Optimale Teamgröße

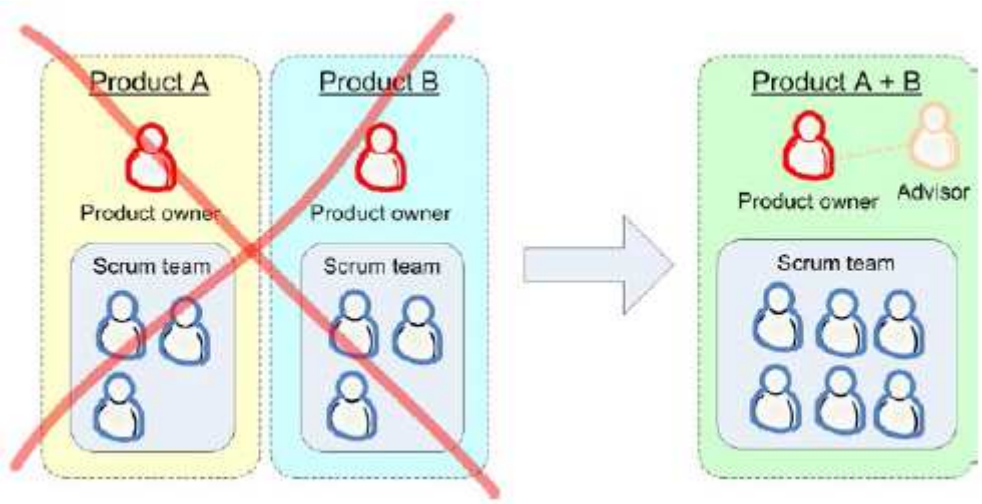
Die meisten Bücher, die ich gelesen habe, proklamieren, dass die "optimale" Teamgröße irgendwo zwischen fünf und neun Leuten liegt.

Von dem, was ich bisher gesehen habe, kann ich dem nur zustimmen. Obwohl ich sagen würde, drei bis acht Leute. In der Tat glaube ich, dass es es wert ist, einige Mühen auf sich zu nehmen, um eine solche Teamgröße zu erreichen.

Sagen wir, Du hast ein einziges Scrumteam mit 10 Leuten. Überlege, ob Du die zwei schwächsten Teammitglieder herausnimmst. Ups, hab ich das gerade gesagt?

> LOL. Wie zur Hölle kam ich da wieder raus? :o)
>
> Trotzdem, ich habe bemerkt, dass ein großes Team sich manchmal schneller bewegt, wenn einige Teammitglieder im Urlaub sind. Aber das ist normalerweise nicht deswegen so, weil die fehlenden Leute schwache Teammitglieder wären. Es liegt einfach daran, dass die Teamgröße handhabbarer wurde und der Mehraufwand für Kommunikation und Durcheinander reduziert war.

Sagen wir, Du hast zwei verschiedene Produkte mit einem Drei-Personen-Team für jedes Produkt, und beide bewegen sich zu langsam. Es *könnte* eine gute Idee sein, sie zu einem einzigen Sechs-Personen-Team zu vereinen, das für beide Produkte verantwortlich ist. In diesem Fall lass einen der beiden Product Owner gehen (oder gib ihm eine beratende Rolle oder sowas).



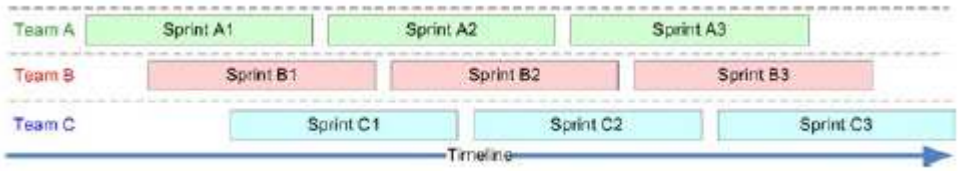
Sagen wir, Du hast ein einzelnes 12-Personen-Scrumteam, weil die Codebasis in so einem beschissenen Zustand ist, dass es keine Möglichkeit gibt, dass zwei Teams unabhängig voneinander daran arbeiten. Stecke ernsthaft Aufwand in die Ausbesserung der Codebasis (statt neue Features einzubauen), bis Du an einen Punkt kommst, an dem Du das Team teilen kannst. Diese Investition wird sich wahrscheinlich ziemlich schnell rechnen.

> Dies ist es wert, hervorgehoben zu werden. Wenn Du Dich abmühst, die richtige Teamstruktur zu finden, ist der wahre Schuldige oft die Systemarchitektur. Eine gute Architektur lässt das Team schnell und unabhängig voneinander vorankommen, eine schlechte Architektur bewirkt, dass Teams übereinander stolpern und bis zum Hals in Abhängigkeiten stecken. Also solltest Du ständig am Ball bleiben und beides weiter entwickeln, Deine Architektur und Deine Teamstruktur.

Synchronisierte Sprints – oder nicht?

Sagen wir, Du hast drei Scrumteams, die am selben Produkt arbeiten. Sollten ihre Sprints synchronisiert werden, d. h. gleichzeitig starten und enden? Oder sollten sie sich überlappen?

Unser erster Ansatz war, mit überlappenden Sprints zu arbeiten (was die Zeit anbelangt).



Das klang nett. Zu jedem gegebenen Zeitpunkt würde es einen aktuellen Sprint geben, der gerade endet, und einen neuen Sprint, der gerade beginnt. Die Arbeitslast des Product Owners würde sich gleichmäßig über die Zeit verteilen. Es würde Releases geben, die kontinuierlich aus dem System flössen. Demos jede Woche. Hallelujah.

Ja, ich weiß, aber es *klang* damals wirklich überzeugend!

Wir hatten gerade angefangen, so vorzugehen, als ich eines Tages die Möglichkeit hatte, mit Ken Schwaber zu sprechen (im Zusammenhang mit meiner Scrumzertifizierung). Er wies darauf hin, dass dies eine *schlechte* Idee war, dass es viel besser wäre, die Sprints zu synchronisieren. Ich erinnere mich nicht mehr an die konkreten Gründe, aber nach einigem Diskutieren war ich überzeugt.

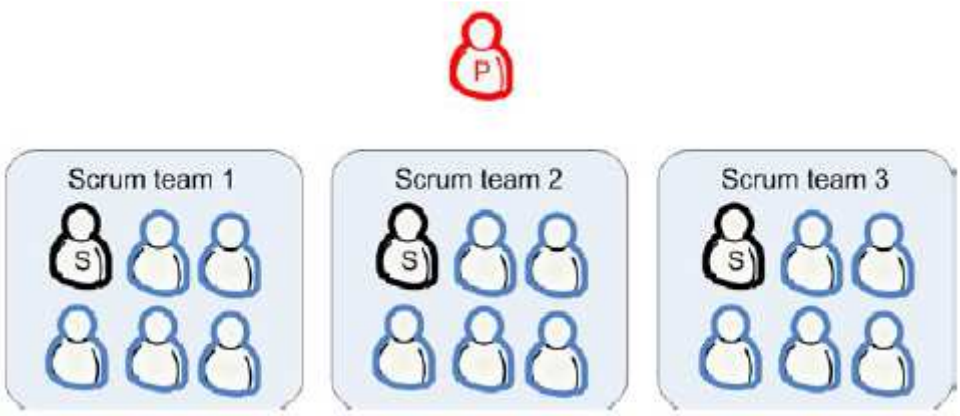
Dies ist die Lösung, die wir seitdem immer benutzt haben, und das haben wir nie bereut. Ich werde nie wissen, ob die Strategie der überlappenden Sprints gescheitert wäre, aber ich denke doch. Der Vorteil synchronisierter Sprints ist:

- Es gibt eine offensichtliche Zeit, zu der Teams sich neu ordnen – zwischen den Sprints! Bei überlappenden Sprints gibt es keine Möglichkeit, Teams neu zu ordnen, ohne zumindest ein Team mitten im Sprint zu stören.
- Alle Teams können auf das gemeinsame Ziel im Sprint hin arbeiten und ihre Sprintplanungstreffen zusammen durchführen, was zu einer besseren Zusammenarbeit zwischen den Teams führt.
- Weniger Verwaltungsaufwand, d. h. weniger Sprintplanungsmeetings, Sprintdemos und Releases.

Bei Spotify hatten wir synchronisierte Sprints, bis wir entschieden, jedes Team in seiner eigenen Geschwindigkeit arbeiten zu lassen und seine eigene Sprintlänge zu wählen (einige nutzen Kanban statt Sprints). Als wir eine Menge Abhängigkeiten zwischen den Teams feststellten, führten wir eine Art Daily Scrum-of-Scrums-Treffen ein und eine wöchentliche Demo des integrierten Produkts, ungeachtet des Rhythmus der individuellen Teams. Ich kann nicht sagen, welches Muster das beste ist, nur dass es kontextabhängig ist.

Warum wir die Rolle "Teamleitung" einführten

Angenommen, wir haben ein einziges Produkt mit drei Teams.



Der rote Typ mit dem Etikett P ist der Product Owner. Die schwarzen Typen mit dem Etikett S sind Scrum Master. Der Rest sind Arbeitsknechte… - äh, respektable Teammitglieder.

Wer entscheidet in dieser Konstellation, welche Leute in welchem Team sein sollten? Der Product Owner? Die drei Scrum Master gemeinsam? Oder sucht sich jede Person ihr eigenes Team aus? Und in dem Fall, was wenn alle in Team 1 sein wollen (weil der Scrum Master 1 so *gut aussieht*)?

Was, wenn sich später herausstellt, dass es wirklich nicht möglich ist, mehr als zwei Teams zu haben, die parallel an der Codebasis arbeiten, so dass wir die drei Sechs-Personen-Teams in zwei Neun-Personen-Teams überführen müssen. Das bedeutet zwei Scrum Master. Also, wem der aktuellen drei Scrum Master wird sein Titel abgenommen?

In vielen Firmen sind das ziemlich sensible Punkte.

Die Versuchung ist groß, den Product Owner die Zuordnung und die Neuzuordnung der Leute machen zu lassen. Aber das ist nicht wirklich die Sache eines Product Owners, richtig? Der Product Owner ist der Fachexperte, der dem Team sagt, in welche Richtung sie laufen sollen. Er sollte tatsächlich nicht in diese grundlegenden Details verwickelt werden. Besonders, seit er ein "Huhn" ist (wenn Du von der Huhn- und Schwein-Metapher gehört hast, sonst such in Google nach "chickens and pigs").

> Oh, ich hasse diese dumme Metapher (such in Google, wenn Du meinst). Aus irgendeinem seltsamen Grund war sie in der Scrumwelt beliebt, um die Behandlung des Product Owners als eine Art von Außenseiter zu behandeln, der nicht auf die zu erledigende Arbeit verpflichtet ist. Sehr beleidigend. Nichtsdestotrotz denke ich immer noch, dass der Product Owner nicht derjenige sein sollte, der die Teamstruktur vorantreibt, denn der Job eines Product Owners ist schon so schwer genug. Also wer sollte das dann tun? Lies weiter …

Wir haben das durch die Einführung einer "Teamleitung"-Rolle gelöst. Das passt dazu, was Du "Scrum-of-Scrums-Master" oder "den Boss" oder "Haupt-Scrum-Master" etc. nennen könntest. Er muss nicht nur ein einzelnes Team führen, sondern ist für die teamübergreifenden Punkte verantwortlich wie die Punkte, wer Scrum Master für die Teams sein sollte, wie Leute in Teams eingeteilt werden etc.

Wir hatten eine schwere Zeit, bis wir für diese Rolle einen guten Namen gefunden hatten. "Teamleitung" war der letzte lausige Name, den wir finden konnten.

Diese Lösung funktionierte gut für uns, und ich kann sie empfehlen (ungeachtet dessen, wie Du Dich entscheidest, wie Du die Rolle nennst).

> In den meisten anderen Firmen, die ich gesehen habe, ist der Linienmanager für die Teamstruktur verantwortlich, und es gibt den Bedarf einer Teamleitungsrolle nicht (übrigens, "Teamleitung" ist ein eher irritierender Name, da er so klingt, als ob es nur ein Team gäbe). Die besten Manager finden jedoch eine Möglichkeit, dem Team zu helfen, sich eher selbst in einer passenden Struktur zu organisieren, als dass sie von oben herab eine Struktur vorgegeben bekommen.

Wie wir die Leute den Teams zuordnen

Es gibt zwei grundlegende Strategien, wie man Leute Teams zuordnet, wenn Du mehrere Teams für dasselbe Produkt hast:

Lass eine benannte Person die Zuordnung machen, zum Beispiel die "Teamleitung", die ich oben erwähnt habe, den Product Owner oder den Hauptabteilungsleiter (wenn er genug involviert ist, um in der Lage zu sein, hier eine gute Entscheidung zu treffen).

Lass die Teams es irgendwie selbst hinkriegen.

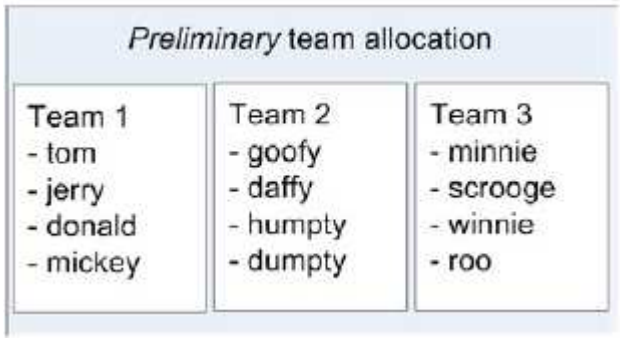
Wir haben es mit allen dreien probiert. Drei? Ja. Strategie 1, Strategie 2 und eine Kombination der beiden.

Wir haben festgestellt, dass die Kombination der beiden am besten funktioniert.

> Und nach vielen weiteren Jahren des Experimentierens damit kann ich dem nur zustimmen.

Vor dem Sprintplanungstreffen ruft die Teamleitung den Product Owner und die Scrum Master zu einem Teamzuordnungstreffen zusammen. Wir reden über den letzten Sprint und entscheiden, ob irgendwelche Teamneuzuordnungen gerechtfertigt sind. Vielleicht wollen wir zwei Teams zusammenlegen oder einige Leute von einem Team in ein anderes schieben. Wir entscheiden uns für irgend etwas und schreiben es nieder als *Teamzuordnungsvorschlag*, den wir ins Sprintplanungstreffen mitnehmen.

Das erste, was wir während des Sprintplanungstreffens machen ist, die hoch priorisierten Einträge im Produkt-Backlog durchzugehen. Die Teamleitung sagt dann etwas wie, "Hallo alle. Wir schlagen die folgende Teamzuordnung für den nächsten Sprint vor."



Vorläufige Teamzuordnung

"Wie Ihr seht, würde dies eine Verringerung von vier auf drei Teams bedeuten. Wir haben die Mitglieder pro Team aufgelistet. Bitte gruppiert Euch an je einem Wandabschnitt."

(Die Teamleitung wartet, während die Leute durch den Raum laufen, nach einer Weile gibt es drei Gruppen von Leuten, jede steht neben einem leeren Wandabschnitt.)

"Nun, diese Teamaufteilung ist *vorläufig*! Es ist nur der Ausgangspunkt, um Zeit zu sparen. Während das Sprintplanungstreffen voran geht, könnt Ihr nach Belieben zu einem anderen Team wandern, Euer Team zweiteilen, es mit einem anderen Team zusammenlegen oder was auch immer. Setzt Euren gesunden Menschenverstand ein, basierend auf den Prioritäten des Product Owners."

Dies ist es, was wir als beste Strategie herausgefunden haben. Ein bestimmter Grad an zentraler Kontrolle zu Anfang, gefolgt von einem bestimmten Grad an dezentraler Optimierung im Anschluss.

> Es ist eine großartige Methode. Behalte im Hinterkopf, dass es viele verschiedene Möglichkeiten gibt, so etwas zu tun – es mit der Sprintplanung zu kombinieren, ist nur eine Möglichkeit, und nicht immer die beste. Heutzutage halte ich normalerweise Reorganisationsworkshops ab, die von der Sprintplanung getrennt sind, um den Fokus in den Sprintplanungstreffen zu behalten (denn wir haben eine stabile Teamstruktur und ein stabiles Produkt-Backlog, wenn wir den Sprint planen). Für große Projekte ist es jedoch nützlich, alle Sprintplanungstreffen gleichzeitig in einem großen Raum abzuhalten. Manchmal kann eine knorrige Abhängigkeit einfach dadurch gelöst werden, dass jemand die Teams wechselt (zeitweise oder langfristig).

Spezialisierte Teams – oder nicht?

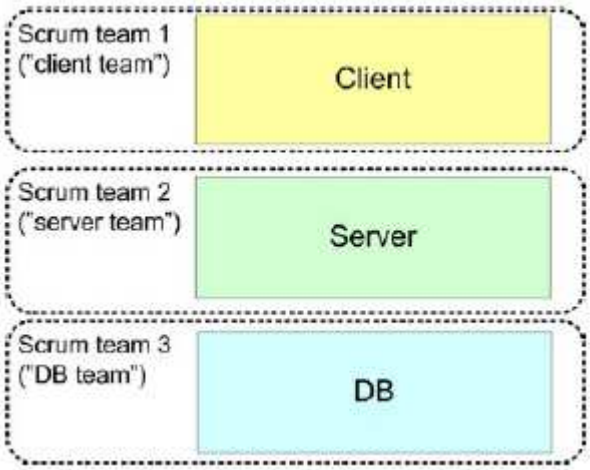
Nehmen wir an, Deine Technik hat drei Hauptkomponenten:

Client, Server, Datenbank

Und nehmen wir an, Du hast 15 Leute, die an diesem Produkt arbeiten, die möchtest Du eigentlich nicht als ein großes Team organisieren. Welche Teams stellst Du auf?

Ansatz 1: Teams nach Komponenten

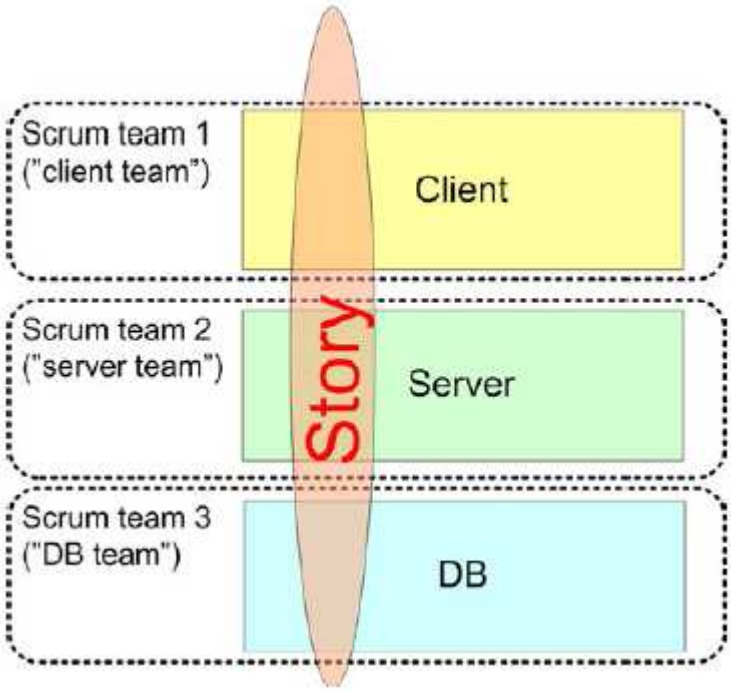
Ein Ansatz ist, Teams zusammenzustellen, die auf die Komponenten spezialisiert sind, also ein Clientteam, ein Serverteam und ein DB-Team.



Clientteam, Serverteam, Datenbankteam

Damit sind wir gestartet. Das funktioniert nicht allzu gut, zumindest nicht, wenn die meisten Storys mehrere Komponenten berühren.

Zum Beispiel, nehmen wir an, wir haben eine Story mit Namen "Notiztafel, auf der User einander Nachrichten schreiben können." Dieses Notiztafel-Feature würde enthalten, dass das User Interface im Client aktualisiert, Logik zum Server hinzugefügt und die Datenbank um einige Tabellen in der Datenbank ergänzt wird.



Das bedeutet, alle drei Teams – das Clientteam, das Serverteam und das DB-Team – müssen zusammen arbeiten, um diese Story fertig zu bekommen. Nicht so gut.

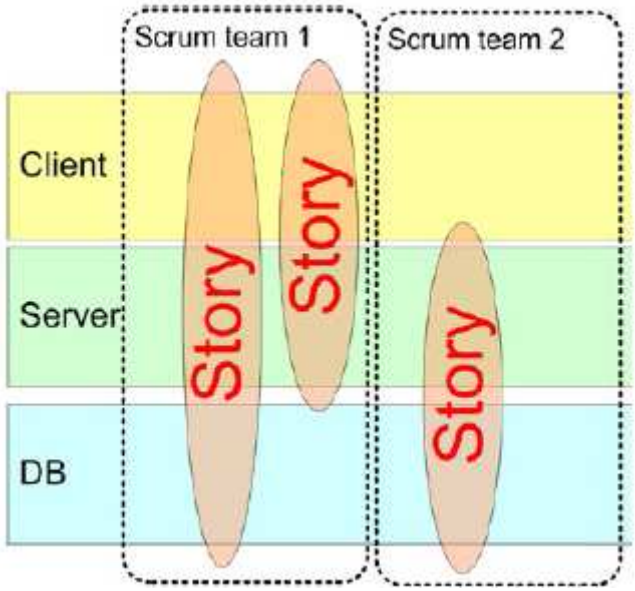
> Ein in vielen Firmen überraschend verbreitetes Problem!

Ansatz 2: Komponentenübergreifende Teams

Ein zweiter Ansatz ist, komponentenübergreifende Teams zusammenzustellen, d. h. Teams, die nicht an eine spezifische Komponente gebunden sind.

Wenn viele unserer Storys mehrere Komponenten berühren, wird diese Art der Aufteilungsstrategie besser funktionieren. Jedes Team kann eine komplette Story implementieren, inklusive der Clientanteile, der Serveranteile und der Datenbankanteile. Die Teams können dadurch unabhängiger voneinander arbeiten, was eine Gute Sache ist.

Eine der ersten Dinge, die wir gemacht haben, als wir Scrum einführten, war, die bestehende Teamsstruktur nach Komponenten (Ansatz 1) aufzubrechen und statt dessen komponentenübergreifende Teams einzuführen. Dies verringerte die Anzahl der Fälle von "wir können den Punkt nicht fertigstellen, weil wir darauf warten, dass die Servertypen ihren Teil erledigen."



> Das ist in fast jeder Firma, mit der ich gearbeitet habe, dieselbe Geschichte. Umorganisieren von Komponententeams kann ziemliche Unruhe stiften, aber der Benefit ist immens!

Was wir jedoch manchmal machen, ist, zeitweise Komponententeams aufzubauen, wenn es dafür einen zwingenden Bedarf gibt.

> Manchmal ist ein Komponententeam sinnvoll, sogar über längere Zeit. Aber es sollte die Ausnahme bleiben, nicht die Norm. Eine gute Testfrage für ein Team ist, "wer ist unser Kunde und können wir seine Bedürfnisse erfüllen, ohne andere Teams blockieren zu müssen?" In den meisten Fällen werden Featureteams diesen Test bestehen und Komponententeams nicht. Die häufigste Ausnahme ist ein nach innen gerichtetes Team wie es Teams für Werkzeuge und Plattformen darstellen (z. B. ein Serverinfrastrukturteam). Sie sollten die anderen Teams (die Featureteams) buchstäblich als ihre Kunden verstehen. Größere Firmen kommen am Ende meist zu einer Mischung aus nach außen gerichteten Featureteams und nach innen gerichteten Komponententeams. Es gibt jedoch keinen Königsweg, also bleib dabei zu experimentieren!

Teams zwischen Sprints neu ordnen – oder nicht?

Jeder Sprint ist üblicherweise ganz anders als jeder andere, abhängig davon, welche Arten von Storys zu diesem bestimmten Zeitpunkt gerade die höchste Priorität haben.

Tatsächlich ist es so, dass wir fast in jedem Sprint festgestellt haben, dass wir uns selbst etwas gesagt haben wie "*dieser* Sprint ist wirklich kein *normaler* Sprint, weil (bla bla bla)…" Nach einer Weile gaben wir die Vorstellung von "normalen" Sprints auf. Es gibt keine normalen Sprints. Genausowenig wie es normale Familien oder normale Leute gibt.

In einem Sprint mag es eine gute Idee sein, ein Team ausschließlich für den Client zu haben, das aus allen besteht, die die Clientcodebasis gut kennen. Im nächsten Sprint mag es eine gute Idee sein, zwei komponentenübergreifende Teams zu haben und die Clienttruppe zwischen diesen aufzuteilen.

Einer der Schlüsselaspekte von Scrum ist "Teamzusammenhalt", d. h. wenn ein Team über mehrere Sprints miteinander zusammenarbeitet, kommen sich die Teammitglieder normalerweise *sehr nah*. Sie werden lernen, einen *Status der Gruppendynamik (group flow)* zu erreichen, der dazu führt, dass das Team ganz in seiner Arbeit aufgeht und Zeit und Raum vergessen lässt, und einen unglaublichen Level an Produktivität zu erreichen. Aber es braucht ein paar Sprints, um an diesen Punkt zu kommen. Wenn Du ständig die Teams herumtauschst, wirst Du nie diese Form von starkem Teamzusammenhalt erreichen.

Also, wenn Du die Teams ändern möchtest, stelle sicher, dass Du die Folgen berücksichtigst. Handelt es sich um eine langfristige oder eine kurzfristige Änderung? Wenn es eine kurzfristige Änderung ist, überlege, sie wegzulassen. Wenn es eine langfristige Änderung ist, dann los.

Eine Ausnahme ist der Fall, wenn Du zum ersten Mal beginnst, mit einem großen Team mit Scrum zu arbeiten. In diesem Fall lohnt es sich wahrscheinlich, ein bisschen mit Teamaufteilung herum zu experimentieren, bis Du etwas findest, mit dem jeder zufrieden ist. Stelle sicher, dass jeder versteht, dass es OK ist, die ersten paar Male alles falsch zu machen, solange Ihr Euch weiterentwickelt.

> Daumenregel: Nach einigen Sprints sollte Eure Teamstruktur einigermaßen stabil sein. D. h. für jedes der Teams bleibt die Zusammensetzung für mindestens ein Quartal unverändert (keine Leute hinzu- oder rausnehmen). Wenn das Team sich öfter ändert, werden sie vermutlich nie den hyper-produktiven Status erreichen, den Scrum anstrebt. Änderungen jedoch, die sich aus dem Inneren ergeben (initiiert durch die Teammitglieder), sind normalerweise weniger störend als Teamänderungen, die von oben verhängt werden. Also, wenn Du ein Manager bist, versuche standhaft der Versuchung zu widerstehen, Dich einzumischen. Lass sich die Leute selbst einrichten, ermutige zu Teamstabilität, aber erlaube den Leuten auch, die Teams zu wechseln, wann immer sie den Bedarf sehen. Du wirst wahrscheinlich über die Ergebnisse erstaunt sein!

Teammitglied für einen Teil der Zeit

Ich kann nur bestätigen, was die Scrumbücher sagen – Teammitglieder nur für einen Teil der Zeit im Scrumteams zu haben, ist keine gute Idee.

> … wie in "es ist für gewöhnlich total beschissen!"

Nehmen wir an, Du bist drauf und dran, Joe als Teilzeitteammitglied in Dein Scrumteam aufzunehmen. Denk zuerst gründlich nach. Brauchst Du Joe wirklich im Team? Bist Du sicher, dass Du Joe nicht für die volle Zeit bekommen kannst? Was sind seine anderen Verpflichtungen? Kann jemand anderes Joes anderen Einsatzbereich übernehmen und Joe eine mehr passive, unterstützende Rolle in diesem Einsatzbereich einnehmen lassen? Kann sich Joe ab dem *nächsten* Sprint Deinem Team in Vollzeit anschließen und bis dahin seine anderen Verantwortlichkeiten einem anderen übergeben?

Manchmal gibt es einfach keine Lösung. Du brauchst Joe dringend, weil er der einzige DB-Administrator im Haus ist, aber die andern Teams brauchen ihn genauso dringend, so dass er niemals die volle Zeit in Deinem Team sein kann, und die Firma kann nicht weitere DB-Admins einstellen. Fein. Das ist ein gerechtfertigter Fall, ihn auf Teilzeitbasis zu nehmen (was übrigens genau das ist, was uns passiert ist). Aber stell sicher, dass Du diese Abschätzung jedes Mal machst.

Im Allgemeinen habe ich lieber ein Team mit drei Vollzeitlern als mit acht Teilzeitlern.

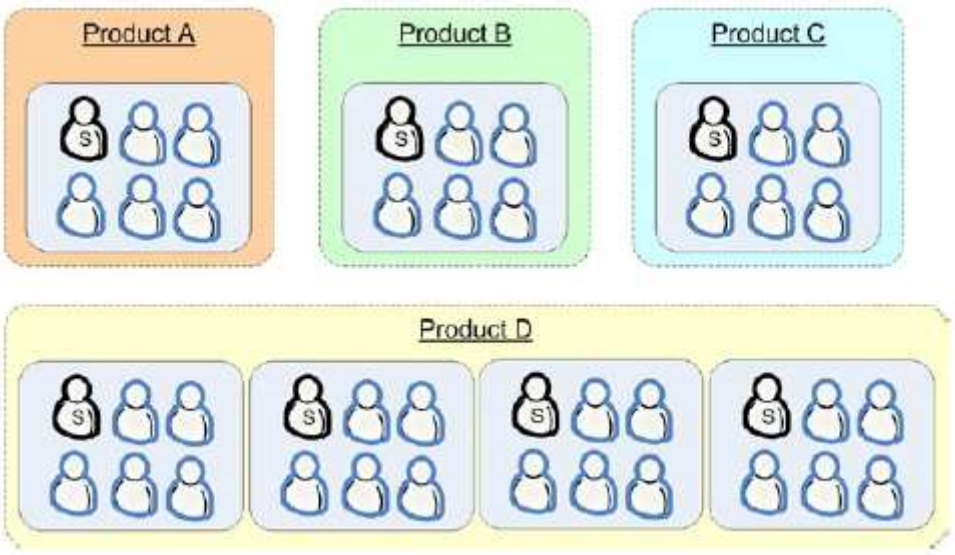
Wenn Du eine Person hast, die ihre Zeit auf mehrere Teams aufteilen wird, wie der vorhin genannte DB-Admin, ist es eine gute Idee, ihn trotzdem hauptsächlich einem Team zuzuordnen. Finde raus, welches Team ihn wahrscheinlich am meisten benötigt, und mache dies zu seinem "Heimatteam". Wenn ihn sonst niemand rauszieht, wird er an den Daily Scrums dieses Teams teilnehmen, an dessen Sprintplanungstreffen, Retrospektiven etc.

> Daumenregel: Vollzeit bedeutet mindestens 90 % für das Team verfügbar. Teilzeit bedeutet 50 bis 90 % ("Dies ist mein Heimatteam, aber ich habe auch andere Verpflichtungen"). Weniger als 50 % bedeutet, dass es kein Teammitglied ist ("Ich kann Euch Jungs von Zeit zu Zeit unterstützen, aber bin nicht Teil Eures Teams und Ihr könnt nicht jederzeit mit mir rechnen"). Wenn ein Team nur einen Teilzeitler hat und den Rest Vollzeitler, dann gräme Dich nicht. Wenn es jedoch mehr als einen Teilzeitler hat, dann solltest Du überlegen, das Team umzuorganisieren oder die Gesamtmenge an Aufgaben, die in Arbeit sind, zu reduzieren, so dass einige Teilzeitler Vollzeitler werden können. Multitasking ist eine Bestie, die Produktivität, Motivation und Qualität frisst, also begrenze diese Fälle auf das absolute Minimum!

Wie wir Scrum-of-Scrums machen

Scrum-Of-Scrums ist im Grunde ein regelmäßiges Treffen, an dem alle Scrum Master sich sammeln, um miteinander zu reden.

Wir hatten einmal vier Produkte, von denen drei Produkte jeweils nur ein Scrumteam hatten, und das letzte Produkt hatte 25 Leute, die in einige Scrumteams aufgeteilt waren. Etwa so wie hier:



Das heißt, wir hatten zwei Level für Scrum-of-Scrums. Wir hatten ein Produkt-Scrum-of-Scrums, der aus allen Teams für das Produkt D bestand, und ein unternehmensweites Scrum-of-Scrums, bestehend aus allen Produkten.

Produkt-Scrum-of-Scrums

Dieses Treffen war sehr wichtig. Wir hielten es einmal pro Woche ab, manchmal auch öfter. Wir diskutierten Integrationsfragen, Aspekte der Ausgeglichenheit der Teams, Vorbereitungen für das nächste Sprintplanungstreffen etc. Wir setzten 30 Minuten an, überzogen aber regelmäßig. Eine Alternative wäre gewesen, jeden Tag Scrum-of-Scrums abzuhalten, aber wir konnten uns nie überwinden, das mal auszuprobieren.

Unsere Scrum-of-Scrums-Agenda lautete:

Jeder am Tisch beschreibt, was sein Team letzte Woche erreicht hat, was es diese Woche zu erreichen plant und welche Hindernisse ihm im Weg stehen.

Und jedwede andere teamübergreifenden Angelegenheiten, die angesprochen werden mussten, zum Beispiel Integrationsfragen.

Die Agenda der Scrum-of-Scrums ist für mich eigentlich nicht wichtig; wichtig ist, dass Du regelmäßig Scrum-of-Scrums-Treffen *abhälst*.

> Bei Spotify haben wir das normalerweise in der Form von "Daily Sync" zwischen den Teams, die gemeinsam an etwas beteiligt waren, gemacht. Ein kurzes Treffen, üblicherweise maximal 15 Minuten. Die Agenda ist hauptsächlich auf Abhängigkeiten gerichtet, weniger auf Statusberichte. Je Team spricht eine Person darüber, was ihr Team von den anderen Teams benötigt, ob ihr Team durch irgend etwas blockiert ist und so weiter. Manchmal setzen wir eine simple Tafel mit Haftzetteln ein, um die ungelösten teamübergreifenden Abhängigkeiten nachzuvollziehen. Das Treffen wird zu einem kleinen Marktplatz zur Identifizierung, wo es Abstimmungsbedarf gibt. Die Abstimmung selbst erfolgt separat – das Treffen hilft nur herauszufinden, wer mit wem reden muss, um die aktuellen Abhängigkeiten aufzulösen.

Unternehmensweites Scrum-of-Scrums

Wir nannten dieses Treffen "den Puls". Wir haben dieses Treffen in verschiedenen Formaten durchgeführt, mit verschiedenen Teilnehmern. Kürzlich haben wir das ganze Konzept verworfen und es durch ein wöchentliches "Alle-Hände"-Treffen (naja, alle an der Entwicklung beteiligten Leute) ersetzt. 15 Minuten.

Was? 15 Minuten? Alle Hände? Alle Mitglieder aller Teams aller Produkte? Funktioniert das?

Ja, es funktioniert, wenn Du (oder wer immer das Treffen durchführt) beim Thema Zeitlimit streng bist.

Das Meetingformat ist:

1) Neuigkeiten und Aktuelles vom Entwicklungsleiter. Info über künftige Maßnahmen, zum Beispiel.
2) Round Robin[20]. Aus jeder Produktgruppe berichtet eine Person darüber, was sie letzte Woche erreicht haben, was sie diese Woche zu erreichen planen und welche Probleme sie haben. Einige andere Personen berichten genau so (CM-Leitung, QA[21]-Leitung etc.).
3) Alle anderen sind eingeladen, jede Art von Information zu ergänzen oder Fragen zu stellen.

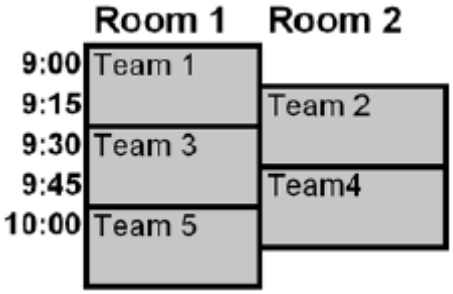
Dies ist ein Forum für den kurzen Informationsaustausch, nicht für Diskussionen oder für ein Reflektieren. Belässt man es dabei, funktionieren 15 Minuten meistens. Manchmal überziehen wir, aber sehr selten landen wir insgesamt bei mehr als 30 Minuten. Wenn erforderliche Diskussionen aufkommen, unterbreche ich sie und lade diejenigen, die es interessiert, ein, nach dem eigentlichen Treffen zu bleiben und die Diskussion fortzusetzen.

Warum veranstalten wir ein "Alle-Hände-Puls-Treffen"? Weil wir gemerkt haben, dass es in den unternehmensweiten Scrum-of-Scrums hauptsächlich ums Berichten ging. Wir hatten kaum effektive Diskussionen in jener Gruppe. Außerdem lechzten viele andere Leute außerhalb der Gruppe nach dieser Art von Info. Im Wesentlichen wollten die Teams wissen, was die anderen Teams gerade tun. Also überlegten wir, wenn wir uns eh trafen und Zeit darauf verwendeten, uns gegenseitig zu informieren, was jedes Team gerade tat, warum nicht jeden teilnehmen lassen?

> Wie Du siehst, das Setup des Scrum-of-Scrums ist sehr kontextabhängig. Es gibt definitiv keine Einheitsform. Ich bin manchmal geschockt davon, wie Firmen jede Woche (oder jeden Tag!) dasselbe todlangweilige, öde und ineffektive Meeting abhalten, mit Leuten, die mit glasigem Blick auf den Boden starren und Statusberichte murmeln, als läsen sie sie vom Blatt ab. Sei kein Scrum-Zombie! Ändert etwas! Experimentiert! Fragt einander ständig Fragen wie, "Erzeugt dieses Treffen wirklich mehr Wert? Könnte es möglicherweise mehr Wert erzeugen? Was müssten wir ändern, um mehr Wert zu erhalten?" Das Leben ist zu kurz für langweilige Meetings.

Die Verschachtelung von Daily Scrums

Wenn Du viele Scrumteams innerhalb desselben Produkts hast und alle ihre Daily Scrums zur selben Zeit machen, hast Du ein Problem. Der Product Owner (und neugierige Leute wie ich) können pro Tag nur das Daily Scrum eines Teams besuchen.



Raum 1 – Raum 2

[20] "Reihum-Modell", s. https://de.wikipedia.org/wiki/Round_Robin_%28Informatik%29
[21] CM = Konfigurationsmanagement; QA = Qualitätssicherung (beide Fußnoten: Anm. der Übersetzerin)

Der Musterplan oben ist aus einer Zeit, als wir Daily Scrums in gesonderten Räumen und nicht im Teambüro abhielten. Die Treffen dauern normalerweise 15 Minuten, aber jedes Team bekommt im Raum einen 30-Minuten-Slot für den Fall, sie überziehen ein bisschen.

Dies ist aus zwei Gründen *extrem nützlich*:

Leute wie der Product Owner und ich selbst können *alle* Daily Scrums an einem einzigen Vormittag besuchen. Es gibt keinen besseren Weg, ein genaues Bild davon zu bekommen, wie der Sprint voran geht und wo die zentralen Bedrohungen liegen.

Teams können gegenseitig ihre Daily Scrums besuchen. Es passiert nicht allzu oft, aber hin und wieder, während zwei Teams im selben Bereich arbeiten, so dass einige Mitglieder ihre Köpfe in die Daily Scrums des anderen Teams hereinstecken, um sich abzustimmen.

Die Kehrseite ist, weniger Freiheit für das Team – sie können nicht jede beliebige Zeit für ihre Daily Scrums wählen. Dies war für uns jedoch nie ein Problem.

> Dies ärgert mich ein bisschen. Der oberste Zweck der Daily Scrums ist für die Teammitglieder, sich miteinander abzustimmen. Sie sollten eine Zeit wählen, die für sie passt. Neugierige Leute wie mich auf dem Laufenden zu halten – naja, das ist zweitrangig. Also ja, es ist nett, überschneidungsfreie Daily Scrums zu haben. Aber nicht zu Lasten des Teams.

Feuerwehrteams

Wir hatten einmal die Situation, wo ein großes Produkt Scrum nicht adaptieren konnte, weil das Team zuviel Zeit investierte, um Brände zu löschen, d. h. panikartige Bugfixes an ihrem verfrüht ausgelieferten System zu machen. Dies war ein echter Teufelskreis, sie waren so beschäftigt mit dem Löschen von Bränden, dass sie keine Zeit hatten, im Voraus daran zu arbeiten, dem Ausbruch von Feuern *vorzubeugen* (d. h. das Design verbessern, Tests automatisieren, Monitoring- und Alarmwerkzeuge zu entwickeln etc.).

Wir reagierten auf dieses Problem, in dem wir ein Feuerwehrteam bildeten und ein Scrumteam.

Die Aufgabe des Scrumteams war, (mit dem Segen des Product Owners) zu versuchen, das System zu stabilisieren und effektiv Brände zu vermeiden.

Das Feuerwehrteam (tatsächlich nannten wir es "Support") hatte zwei Aufgaben:

- Brände löschen
- Das Scrumteam vor allen Arten von Störungen zu schützen, inklusive solcher Dinge wie das Abblocken von Ad-Hoc-Featureanforderungen, die aus dem Nichts kommen.

Das Feuerwehrteam wurde neben der Tür platziert; das Scrumteam wurde hinten im Raum platziert. So konnte das Feuerwehrteam das Scrumteam tatsächlich *physikalisch* vor Unterbrechungen durch eifrige Vertriebler oder ärgerliche Kunden *schützen*.

Erfahrene Entwickler wurden in jedes Team integriert, so dass ein Team nicht allzu abhängig von der Kernkompetenz des anderen war.

Dies war im Wesentlichen ein Versuch, ein Scrum-Initialisierungsproblem zu lösen. Wie können wir mit Scrum beginnen, wenn das Team keine Chance hat, ihre Arbeit über den Tag hinaus zu planen? Unsere Strategie war, wie gesagt, die Gruppe zu teilen.

Das funktionierte ziemlich gut. Seit das Scrumteam mehr Raum bekommen hatte, um vorausschauend zu arbeiten, waren sie schließlich in der Lage, das System zu stabilisieren. In der Zwischenzeit gab das Feuerwehrteam komplett die Vorstellung auf, vorausplanen zu können, und arbeitete völlig reaktiv, gerade nur das behebend, was immer an Panikpunkten als nächstes reinkommen würde.

> Ich hätte definitiv Kanban für das Feuerwehrteam eingesetzt, wenn ich es zu jener Zeit bereits gekannt hätte.

Natürlich blieb das Scrumteam nicht *komplett* ungestört. Das Feuerwehrteam musste regelmäßig auf Leute aus dem Scrumteam zurückgreifen oder im schlimmsten Fall auch auf das ganze Team.

Trotzdem war das System nach ein paar Monaten stabil genug, dass wir das Feuerwehrteam abschaffen und statt dessen ein weiteres Scrumteam installieren konnten.

> Dies ist ein großartiges Pattern, aber nur als übergangsweise Krisenmanagementstrategie. Im Normalfall solltest Du kein eigenes Feuerwehrteam benötigen. Wenn Du die anderen Teams von den Bränden isolierst, werden sie nie lernen, neue Brände abzuwenden! Lass statt dessen jedes Team herausfinden, wie man mit Fehlern und Feuern umgeht, wenn Du von Zeit zu Zeit Probleme damit hast (was beinahe jede Firma hat). Die meisten Teams kommen am Ende an den Punkt, dass sie irgendeine Art von rotierender Feuerwehrrolle innerhalb des Teams haben. Einfach und effektiv. Und es gibt ihnen ganz klar einen Anreiz, Code zu schreiben, der kein Feuer fängt!

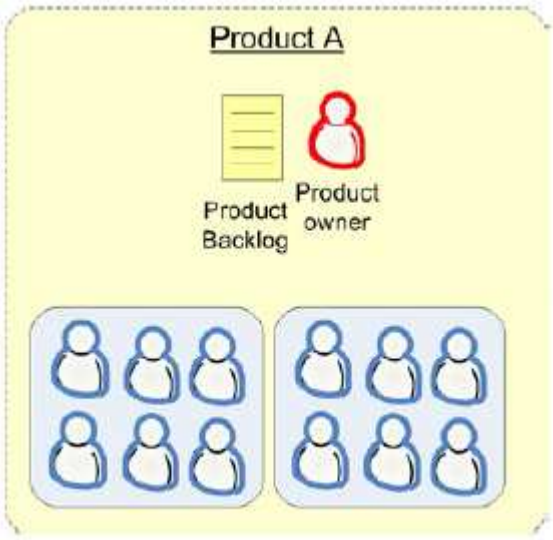
Das Produkt-Backlog aufteilen – oder nicht?

Angenommen, Du hast ein Produkt und zwei Scrumteams. Wie viele Produkt-Backlogs solltest Du führen? Wie viele Product Owner haben? Bei uns haben sich drei Modelle herausgebildet. Welches man auswählt, hat einen ziemlich großen Effekt darauf, wie die Sprintplanungstreffen ablaufen.

Strategie 1: Ein Product Owner, ein Backlog

Dies ist das "Es kann nur einen geben"-Modell. Unser bevorzugtes Modell.

Das Gute an diesem Modell ist, dass Du die Teams so ziemlich sich selbst formen lassen kannst, auf der Basis der aktuellen Top-Prioritäten des Product Owners. Der Product Owner kann sich darauf konzentrieren, *was er braucht*, und die Teams entscheiden lassen, wie sie die Arbeit aufteilen.



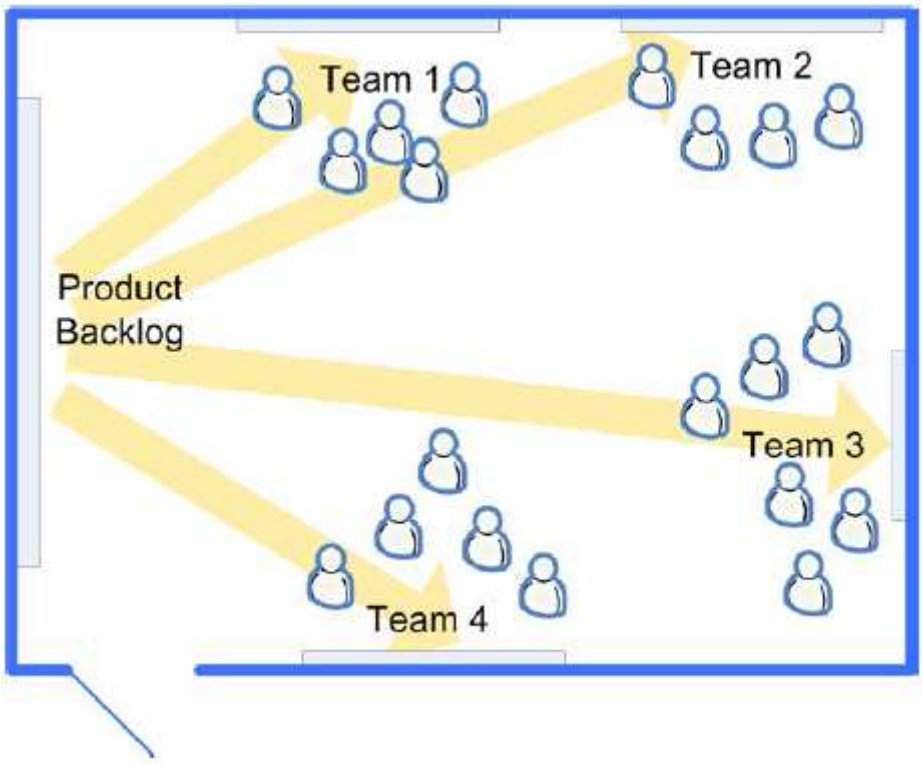
Um konkreter zu werden, ist im folgenden beschrieben, wie das Sprintplanungstreffen für dieses Team funktioniert.

Das Sprintplanungstreffen findet an einem externen Konferenzplatz statt.

Kurz vor dem Meeting erklärt der Product Owner eine Wand zur Produkt-Backlog-Wand und hängt Storys dorthin (auf Karten), sortiert nach relativer Priorität. Er hängt sie auf, bis die Wand voll ist, was normalerweise mehr als genug Einträge für einen Sprint ausmacht.

Jedes Scrumteam wählt einen leeren Wandbereich für sich aus und hängt seinen Teamnamen dort an. Das ist ihre Teamwand. Jedes Team nimmt sich dann Storys von der Produkt-Backlog-Wand, beginnend mit den Storys mit der höchsten Priorität, und zieht die Karten auf seine eigene Teamwand.

Dies ist im Bild unten dargestellt, die gelben Pfeile symbolisieren den Fluss der Story-Karten von der Produkt-Backlog-Wand zu den Teamwänden.



Während das Meeting weiter geht, schachern der Product Owner und die Teams um die Karten, bewegen sie zwischen den Teams, hängen sie höher und tiefer, um die Priorität zu ändern, teilen sie in kleinere Aufgaben auf etc. Nach einer Stunde oder so hat jedes Team eine erste mögliche Version eines Sprint-Backlogs an seiner Teamwand. Danach arbeiten die Teams unabhängig voneinander daran, die Zeiten zu schätzen und die Storys in Aufgaben aufzuteilen.

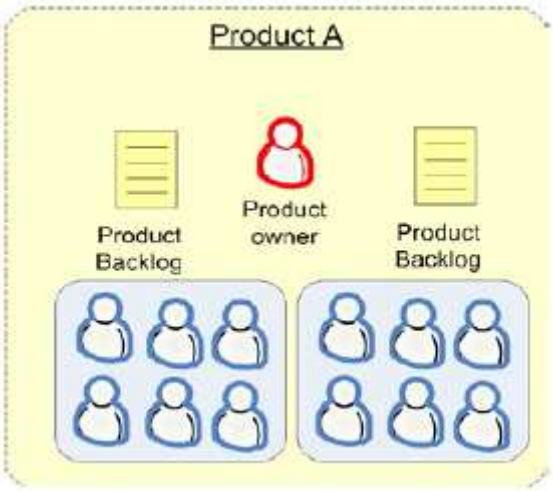
Es ist unordentlich und chaotisch und ermüdend, aber auch effektiv und spaßig und gesellig. Wenn die Zeit vorbei ist, haben die Teams normalerweise genug Informationen, um ihren Sprint zu beginnen.

Dieses Muster verbreitet sich immer mehr; es wird sogar in einige neuere Frameworks für agile Entwicklung eingebaut, wie SAFe (Scaled Agile Framework). Neulich bei LEGO führten wir eine zweitätige Planungsveranstaltung mit mehr als 130 Leuten durch! Manchmal wird diese Planungstechnik als temporäre Maßnahme für ein halbes Jahr oder so eingesetzt, bis wir eine Teamstruktur und Architektur finden, die es ermöglicht, Sprintplanungstreffen unabhängiger voneinander durchzuführen.

Strategie 2: Ein Product Owner, mehrere Backlogs

Bei dieser Strategie pflegt der Product Owner mehrere Produkt-Backlogs, eins pro Team. Wir haben diesen Ansatz nicht exakt so ausprobiert, aber wir waren nah dran. Dies ist im Grunde unser Ausweichplan für den Fall, dass der erste Ansatz scheitert.

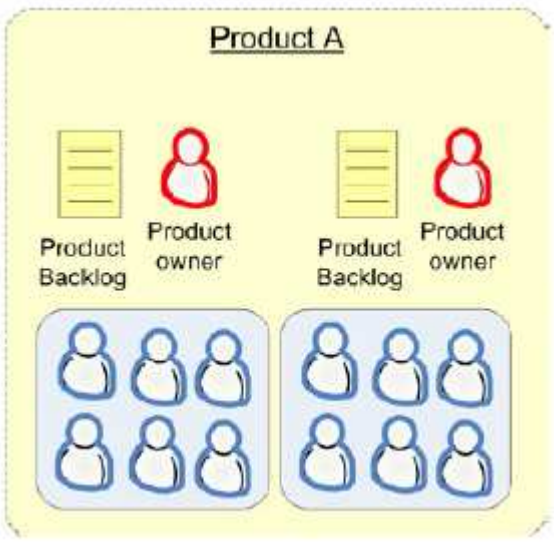
Die Schwäche dieser Strategie liegt darin, dass der Product Owner den Teams die Storys zuweist, eine Aufgabe, die die Teams wahrscheinlich besser selbst machen.



Üblicherweise wird der Product Owner in diesem Fall zum Flaschenhals.

Strategie 3: Mehrere Product Owner, ein Backlog pro Owner

Diese Strategie hat wie die zweite Strategie ein Produkt-Backlog pro Team, außerdem aber noch mit einem *Product Owner* pro Team!



Wir haben das nicht ausprobiert, und wir werden es wahrscheinlich auch nie tun.

Wenn zwei Produkt-Backlogs denselben Code einbinden, verursacht dies wahrscheinlich ernsthafte Interessenskonflikte zwischen den beiden Product Ownern.

Wenn die beiden Produkt-Backlogs auf separatem Code basieren, bedeutet dies im Prinzip dasselbe, als würde man das ganze Produkt in separate Teilprodukte aufsplitten und diese unabhängig voneinander laufen lassen. Dies bedeutet, wir befinden uns wieder in der Ein-Team-Pro-Produkt-Situation, was nett und einfach ist.

> Dies ist das Modell, das bei Spotify im Einsatz ist. Jedes der mehr als 70 Teams hat seinen eigenen Product Owner und sein eigenes Produkt-Backlog. Tatsächlich fahren wir in einigen Fällen ein bisschen Strategie 1 (ein Backlog, verteilt auf mehrere Teams), aber für die meisten Bereiche hat jedes Team seinen eigenen Product Owner und sein eigenes Backlog. Gut daran ist, dass wir kaum große Planungstreffen brauchen. Die Kehrseite dagegen ist, dass wir eine Menge Mühe in die Architektur stecken müssen, was das Auflösen von Abhängigkeiten angeht, um dies zu ermöglichen. Wie bei allem, es gibt immer Vorteile und Nachteile. Also immer schön weiter… äh… (ich war drauf und dran zu sagen, "immer schön weiter experimentieren", aber das habe ich schon so viele Male gesagt… ich möchte mich nicht wiederholen, oder?)

Codeverzweigung[22]

Wenn man mit mehreren Teams an derselben Codebasis arbeitet, hat man es im Softwarekonfigurationsmanagementsystem (SCM) zwangsläufig mit Codebranches zu tun. Es gibt eine Menge Bücher und Artikel darüber, wie man mit mehreren Leuten auf derselben Codebasis arbeitet, deswegen werde ich hier nicht ins Detail gehen. Ich habe nichts Neues oder Revolutionäres hinzuzufügen. Ich werde jedoch einige der wichtigsten Lehren zusammenfassen, die unsere Teams bisher gelernt haben:

Sei streng darin, die Hauptlinie (den Codestamm) immer in einem konsistenten Zustand vorzuhalten. Das bedeutet allerwenigstens, dass alles compilieren muss und alle Unit-Tests durchlaufen müssen. Es sollte möglich sein, *zu jedem beliebigen Zeitpunkt* ein arbeitsfähiges Release zu erstellen. Vorzugsweise sollte das System als Continuous Build jede Nacht erstellt und automatisch als Testumgebung ausgeliefert werden.

Hänge jedem Release ein Schildchen um. Wann immer Du ein Release machst, sei es als Akzeptanztestrelease oder als Produktivrelease, sei sicher, dass es eine Versionsbezeichnung an Deiner Hauptlinie gibt, die genau identifiziert, was als Release erstellt wurde. Das bedeutet, Du kannst in der Zukunft jederzeit zurückgehen und einen Zweig von jenem Punkt aus erstellen.

[22] engl. Code branching (Anm. der Übersetzerin)

Erstelle neue Zweige nur, wenn es notwendig ist. Eine gute Daumenregel ist, eine neue Codelinie *nur dann* abzuzweigen, wenn Du eine vorhandene Codelinie nicht nehmen kannst, ohne die Logik dieser Codelinie zu brechen. Im Zweifel keine Verzweigung. Warum? Weil jeder aktive Zweig Administrations- und Komplexitätsaufwand mit sich bringt.

Nutze Verzweigungen in erster Linie, um *verschiedene Lebenszyklen* voneinander zu trennen. Du kannst Dich dafür oder dagegen entscheiden, den Code jedes Scrumteams in seiner eigenen Codelinie zu haben. Aber wenn Du in derselben Codelinie schnell eingebaute Fixes mit langfristigen Änderungen vermischst, wirst Du feststellen, dass es sehr schwierig wird, die schnell eingebauten Fixes ins Release zu bringen!

Synchronisiere häufig. Wenn Du an einem Zweig arbeitest, synchronisiere diesen mit der Hauptlinie, wann immer Du etwas hast, das lauffähig ist. Jeden Tag, an dem Du arbeitest, synchronisiere die Hauptlinie mit Deinem Zweig, so dass Dein Zweig aktuell ist, bezogen auf die Änderungen anderer Teams. Wenn dies zur Merge-Hölle führt, akzeptiere einfach die Tatsache, dass es noch schlimmer wäre, noch länger zu warten.

Ja, dieser Rat gilt immer noch in starkem Maße (außer für häufiges Tagging[23] – Du bekommst das heutzutage in den meisten Systemen automatisch). Mit fortschrittlichen Versionskontrollsystemen (wie GIT) gibt es keine Entschuldigung mehr dafür, nicht oft einzuchecken, den Stamm sauber zu halten, oft ein Release zu erstellen und die Zweige kurzlebig zu gestalten. Ich habe einen Artikel darüber geschrieben mit dem Titel "Version Control for Multiple Agile Teams":
http://www.infoq.com/articles/agile-version-control

Ich finde es ein bisschen lustig, dass einem Systeme wie GIT als "dezentrales" Versionskontrollsystem schmackhaft gemacht werden, während es in fast jedem Projekt immer noch ein zentrales Repository und einen Hauptzweig gibt, in das jeder einspeist und wo die Releases erstellt werden. Gleich, aber anders[24]. :o)

Retrospektiven mehrerer Teams

Wie machen wir Sprintretrospektiven, wenn mehrere Teams am selben Produkt arbeiten?

Sofort nach der Sprintdemo, nach dem Applaus und dem Zusammenkommen, verschwindet jedes Team wieder in seinem eigenen Büro oder an einen netten Ort außerhalb des Büros. Sie machen ihre Retrospektiven in etwa so, wie ich es auf Seite 66 "Wie wir Sprintretrospektiven machen" beschrieben habe.

Während des Sprintplanungstreffens (an dem alle Teams teilnehmen, da wir die Sprints für jedes Produkt gleich laufen lassen) ist das erste, was wir tun, dass wir einen Sprecher aus jedem Team aufstehen und die Schlüsselaspekte aus seiner Perspektive zusammenfassen lassen. Braucht etwa fünf Minuten pro Team. Dann haben wir circa 10 bis 20 Minuten lang eine offene Diskussion. Dann machen wir eine Pause. Dann beginnen wir mit der eigentlichen Sprintplanung.

Wir haben keine anderen Möglichkeiten für Mehrfachteams ausprobiert; diese funktioniert gut genug. Der größte Nachteil ist, dass es keine ungeplante Zeit[25] nach dem Sprintretrospektiventeil gibt, bevor der Sprintplanungsteil des Treffens anfängt (siehe Seite 71 "Freie Zeiten zwischen den Sprints").

[23] engl. Tagging: Ein Begriff aus der Versionskontrolle: Ein Tag ist eine „Momentaufnahme“ eines Projekts. Jede Revision im Projektarchiv ist genau das – eine Momentaufnahme des Dateisystems nach einer Übergabe.
Quelle: http://svnbook.red-bean.com/de/1.6/svn.branchmerge.tags.html
[24] engl. "Same but different" = minimale Abweichung
[25] engl. Slack (alle Fußnoten auf dieser Seite: Anm. der Übersetzerin)

Exakt! Und das ist ein ziemlich großer Nachteil! Also heutzutage, in einem Szenario mit mehreren Teams und mit Abhängigkeiten, bevorzuge ich Zusammenfassungen der einzelnen Retrospektiven als Teil der eigentlichen Retrospektiven zu machen. Das heißt, jedes Team zieht sich für eine eigene Retrospektive zurück, und nach einer Stunde oder so kommen sie zurück und wir treffen uns alle in einem großen Raum. Jedes Team fasst die Schlüsselergebnisse seiner Retrospektive zusammen und listet seine wichtigsten ungelösten Dinge auf, die seine Arbeit behindern. Nachdem alle Teams angehört wurden, wird normalerweise klar, dass Hindernis X unser größtes teamübergreifendes Hindernis ist (z. B. "inkonsistente Definition of Done zwischen den Teams" oder "Codestamm bricht ständig zusammen"). Das ist eine perfekte Gelegenheit, einen Miniworkshop darüber abzuhalten oder einige Freiwillige zu finden (z. B. eine Person aus jedem Team plus der Manager), die abziehen und eine Lösung austüfteln. Manchmal tüfteln sie sie innerhalb einiger Stunden aus, deswegen ist es gut, das Sprintplanungstreffen am darauf folgenden Tag anzusetzen.

Für ein Produkt, an dem nur ein Team arbeitet, machen wir keine Retrospektivenzusammenfassung während des Sprintplanungstreffens. Braucht man nicht, da alle beim eigentlichen Retrospektiventreffen dabei sind.

Kontext ist König, sagt das Sprichwort. Und für Skalierung gilt das ganz besonders! Es gibt wirklich keine Eine-für-Alle-Lösung – so ziemlich jeder Ansatz kann höchst erfolgreich oder ein totaler Reinfall sein, abhängig vom Kontext.

Obwohl ich einige relevante gemeinsame Nenner gefunden habe: Arbeite mit funktionsübergreifenden Teams, visualisiere die Dinge, liefere häufig aus, beteilige echte Anwender, automatisiere Tests und den Releaseprozess und probiere viel aus. Die wesentlichen agilen Prinzipien, im Grunde.

TEIL SECHZEHN

Wie wir mit geografisch getrennten Teams umgehen

Was passiert, wenn Teammitglieder an geografisch verschiedenen Orten verteilt sind? Viel der "Magie" von Scrum und XP basiert darauf, dass Teammitglieder nah beieinander sitzen, eng zusammen arbeiten und Pairprogramming machen, während sie sich jeden Tag persönlich sehen.

Wir haben ein paar örtlich getrennte Teams, und wir haben Teammitglieder, die von Zeit zu Zeit von zu Hause aus arbeiten.

Unsere Strategie dafür ist ziemlich einfach. Wir benutzen jeden Trick, der uns einfällt, um die Kommunikationsbandbreite zwischen den physisch voneinander getrennten Teams zu maximieren. Ich meine nicht nur die Kommunikationsbandbreite im Sinne von Megabit pro Sekunde (obwohl das natürlich auch wichtig ist). Ich meine die Kommunikationsbandbreite in einem weiteren Sinn:

- Die Möglichkeit, zusammen Pairprogramming zu betreiben
- Die Möglichkeit, sich persönlich beim Daily Scrum zu treffen
- Die Möglichkeit, jederzeit persönlich zu diskutieren
- Die Möglichkeit, sich physisch zu treffen und Kontakte zu knüpfen und zu halten
- Die Möglichkeit, spontane Treffen mit dem ganzen Team abzuhalten
- Die Möglichkeit, denselben Blick auf das Sprint-Backlog, auf das Burndown-Diagramm, auf das Produkt-Backlog und auf andere Informationsquellen zu haben

Einige der Maßnahmen, die wir durchgeführt haben (oder die wir gerade einführen – wir haben sie noch nicht alle durchgeführt), sind:

- Webcam und Headset an jedem Arbeitsplatz
- Konferenzräume mit Fernzugriffmöglichkeiten, mit Webcams, Konferenzmikros, immer bereitstehenden Computern, Desktop-Sharing-Software etc.
- Fernbildfenster – große Monitore an jedem Ort, die einen permanenten Blick auf andere Orte zeigen, sowas wie ein virtuelles Fenster zwischen zwei Abteilungen. Du kannst dort stehen und winken. Du kannst sehen, wer an seinem Schreibtisch ist und wer gerade mit wem spricht. Dies dient dazu, ein Gefühl herzustellen, das ausdrückt: "Hey, wir sind da gemeinsam dran."
- Austauschprogramme, bei denen Leute von allen Standorten regelmäßig reisen und sich gegenseitig besuchen können

Indem wir diese und weitere Methoden einsetzen, beginnen wir langsam, aber sicher, den Dreh herauszubekommen, wie wir Sprintplanungstreffen, Demos, Retrospektiven, Daily Scrums etc. mit Teammitgliedern hinbekommen, die an getrennten Orten arbeiten.

> Verteilte Teams sind überall (der Wortwitz ist beabsichtigt). Die meisten Open-Source-Projekte werden von völlig voneinander getrennten Teams gebaut, also gibt es keinen Zweifel, dass so etwas möglich ist. Nichtsdestotrotz kann nichts die Produktivität eines kleinen, funktionsübergreifenden Teams schlagen, das im selben Raum zusammen sitzt und sich zu 100 % auf ein einzelnes, gemeinsames Ziel konzentriert. Also sollte die oberste Priorität immer sein, solch eine Situation zu erreichen oder ihr so nah wie möglich zu kommen. Wenn Du es trotzdem nicht vermeiden kannst, Teammitglieder an verschiedenen Orten zu haben, sind die oben aufgelisteten Methoden eine großartige Möglichkeit, den Schaden zu begrenzen. Also sei nicht geizig, besorg die besten Tools, die man für Geld kaufen kann! Du wirst nie so produktiv sein wir mit einem Team, das an einem Ort zusammen ist, aber Du könntest dem nahe kommen.

Wie immer geht es im Kern ums Experimentieren:
Prüfen => Anpassen => Prüfen => Anpassen => Prüfen => Anpassen => Prüfen => Anpassen => Prüfen => Anpassen

Offshoring[26]

Wir haben einige Teams, die im Ausland arbeiten und haben herumexperimentiert, wie man damit effizient umgehen kann, wenn man Scrum einsetzt.

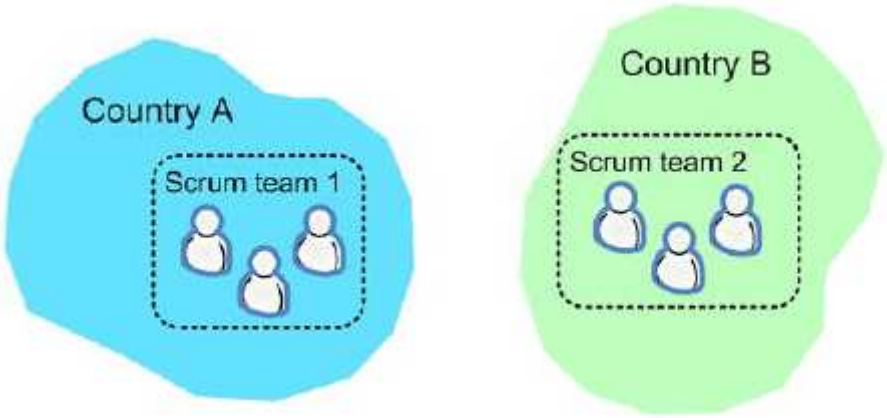
Es gibt hierzu zwei wesentliche Strategien: "getrennte Teams" oder "verteiltes Team".

[26] Produktionsverlagerung ins Ausland

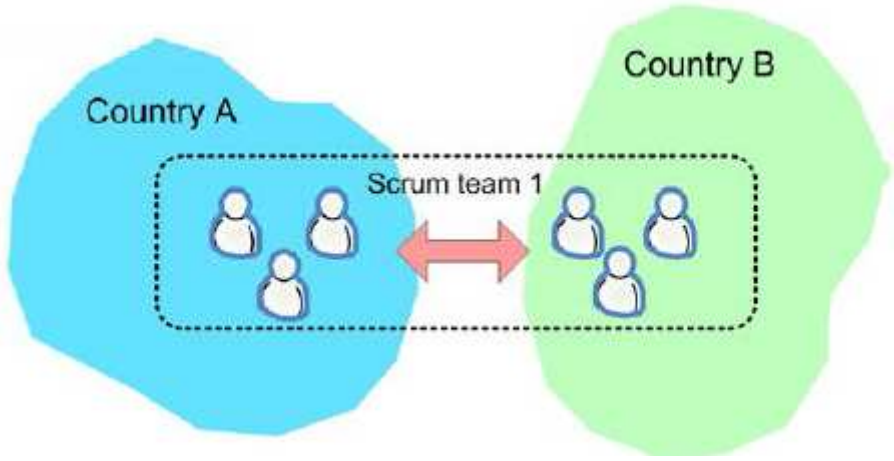
Die erste Strategie, getrennte Teams, ist eine überzeugende Alternative. Nichtsdestotrotz haben wir mit der zweiten Strategie begonnen, mit dem verteilten Team. Es gibt dafür einige Gründe:

1) Wir möchten, dass sich die Teammitglieder gut kennenlernen.
2) Wir möchten eine erstklassige Kommunikationsinfrastruktur zwischen zwei Orten, und wir möchten den Teams einen großen Anreiz geben, diese aufzusetzen.
3) Zu Anfang ist ein Offshore-Team zu klein, um allein ein effektives Scrumteam zu bilden.
4) Wir nehmen uns eine Zeit intensiven Wissensaustauschs, bevor unabhängige Offshore-Teams eine praktikable Alternative sind.

Später könnte es sein, dass wir uns in Richtung der Strategie "separate Teams" bewegen.



Getrennte Teams



Verteiltes Team

Das ist eine gute Gesamtstrategie. Auf lange Sicht willst Du wirklich Teams, die am selben Ort zusammen arbeiten, weil ein verteiltes Team einfach nie dieselbe "Teaminess" erreichen wird. Also selbst wenn Du ursprünglich ein verteiltes Team hast, suche ständig nach Wegen, separate, klar voneinander abgegrenzte Teams an jedem Standort zu haben. Du wirst immer noch mit Abhängigkeiten und Kommunikationsfragen zwischen den zwei Teams umgehen müssen, aber das ist als Problem einfacher zu handhaben. Zusätzlich zur besseren täglichen Kommunikation ist einer der wesentlichen Gewinne Motivation. Die Teammitglieder im selben Raum zu haben, bringt Spaß! Und motivierte Leute bauen bessere Sachen, und das schneller.

Teammitglieder, die von zu Hause aus arbeiten

Von zu Hause aus arbeiten kann manchmal wirklich gut sein. Manchmal bekommst Du an einem Tag zu Hause mehr programmiert als in einer ganzen Woche im Büro. Zumindest, wenn Du keine Kinder hast. :o)

Doch einer der Grundsätze in Scrum lautet, dass das ganze Team physisch am selben Ort arbeiten sollte. Also, was tun?

Im Prinzip lassen wir die Teams entscheiden, wann und wie oft es okay ist, von zu Hause aus zu arbeiten. Einige Teammitglieder arbeiten regelmäßig von zu Hause aus, was ihrem langen Anfahrtsweg geschuldet ist. Wir ermutigen die Teams jedoch, die "meiste" Zeit persönlich während der Zusammenarbeit anwesend zu sein.

Wenn Teammitglieder von zu Hause aus arbeiten, gesellen sie sich zum Daily Scrum dazu, indem sie die Skype-Telefonfunktion nutzen (manchmal Video). Sie sind per Instant Messaging den ganzen Tag online. Nicht so gut, wie im selben Raum zu sein, aber gut genug.

> Es stehen heute jede Menge richtig coole und erschwingliche Telekommunikationssysteme zur Verfügung. Guck Dir zum Beispiel Double Robotics unter http://doublerobotics.com an. Sie verkaufen ein Produkt, das im Grunde ein iPad an einem Stock auf Rädern ist. Es ist wie bei einem Skype-Call zu sein, aber Du kannst auch noch herumsausen! Ich benutze es immer öfter; es fühlt sich wirklich an wie Teleportieren an einen anderen Ort!

Wir haben einmal das Konzept ausprobiert, den Mittwoch als *Fokustag* zu deklarieren. Das meint im Grunde, "wenn Du von zu Hause aus arbeiten möchtest, ist das OK, aber mach es am Mittwoch. Und sprich es mit Deinem Team ab." Das hat in dem Team, in dem wir es ausprobiert haben, ganz gut funktioniert. Normalerweise blieben die meisten im Team am Mittwoch zu Hause und bekamen eine Menge geschafft, während sie immer noch gut zusammenarbeiteten. Da es nur ein Tag in der Woche war, lief das Ganze nicht zu sehr auseinander. Aus irgendeinem Grund übernahmen die anderen Teams das Konzept jedoch nicht.

Alles in allem war es kein Problem für uns, wenn Leute von zu Hause aus arbeiteten.

> Eine der Schlüsselideen bei Scrum ist, dass das Team sich selbst organisiert. Die Relevanz dieses Punktes kann nicht hoch genug eingeordnet werden. Dem Team sollte soviel Verantwortung wie möglich gegeben werden, inklusive Dingen wie Arbeitsstunden und Regelungen zur Von-zu-Hause-aus-Arbeit. Selbstorganisation ist der Schlüssel zur Kreativität, Motivation, Innovation und vielen anderen Guten Dingen! Einige Manager hatten Befürchtungen, dass Teams dieses Vertrauen missbrauchen würden, aber ich habe selten erlebt, dass dies in der Praxis passiert. So lang das Team unmissverständlich für das Produkt, das sie liefern, zur Rechenschaft gezogen wird, tendiert es dazu, verantwortungsvoll zu handeln.

TEIL SIEBZEHN

Scrum-Master-Checkliste

In diesem letzten Kapitel werde ich Dir unsere "Scrum-Master-Checkliste" zeigen, die die alltäglichen administrativen Tätigkeiten unserer Scrum Master auflistet. Zeug, das man leicht vergisst. Wir überspringen offensichtliche Dinge wie "befrei das Team von Arbeitshindernissen".

Sieh Dir außerdem die Scrum-Checkliste an. https://www.crisp.se/gratismaterial-och-guider/scrum-checklist

Es ist offiziell die inoffizielle Scrum-Checkliste, aber sie ist mittlerweile so verbreitet im Einsatz, dass Du sagen könntest, sie ist inoffiziell die offizielle Scrum-Checkliste geworden. :o)

Der Sprintstart

- Erstelle nach dem Sprintplanungstreffen eine Sprint-Info-Seite
    - Füge einen Link zu Deiner Dashboard-Seite im Wiki hinzu.
    - Drucke die Seite aus und hänge sie da an die Wand, wo die Leute Deines Teams vorbeigehen.
- Schicke jedem eine Email, in der Du ankündigst, dass ein neuer Sprint begonnen hat. Inklusive Sprintziel und einem Link zur Sprint-Info-Seite.
- Aktualisiere das Dokument mit den Sprint-Statistiken. Füge die geschätzte Umsetzungsgeschwindigkeit hinzu, die Teamgröße, Sprintlänge etc.

Täglich

- Stelle sicher, dass die Daily Scrum pünktlich anfangen und enden.
- Stelle sicher, dass Storys ins Sprint-Backlog hineinkommen bzw. entfernt werden, so wie es notwendig ist, um im Sprintplan zu bleiben.
    - Stelle sicher, dass der Product Owner über diese Änderungen informiert wird.
- Stelle sicher, dass das Sprint-Backlog und das Burndown-Diagramm vom Team aktuell gehalten werden.
- Stelle sicher, dass Probleme und Hindernisse beseitigt werden oder dass darüber an den Product Owner und / oder an den Entwicklungsleiter berichtet wird

Das Sprintende

- Führe eine offene Sprint-Demo durch.
- Jeder sollte über die Demo einen oder zwei Tage vorher informiert sein.
- Mach eine Sprintretrospektive mit dem ganzen Team und dem Product Owner. Lade auch den Entwicklungsleiter ein, so dass er helfen kann, die gemachten Erfahrungen weiter zu verbreiten.
- Aktualisiere das Dokument mit den Sprint-Statistiken. Füge die tatsächliche Umsetzungsgeschwindigkeit und die Schlüsselergebnisse aus der Retrospektive hinzu.

Nette kleine Checkliste. Obwohl Du zusehen solltest, dass Du Dich als Scrum Master mit der Zeit überflüssig machst. Trainiere das Team so, dass es diese Dinge ohne Dich tun kann. Sogar wenn Du keinen Erfolg dabei hast, Dich selbst überflüssig zu machen, wird allein der Akt des Versuchens Dich zu Guten Dingen führen. Einige Scrum Master landen am Ende in der Rolle eines Scrum-Verwalters oder eines Scrum-Sklaven, weil sie so begierig darauf sind, dem Team zu gefallen. Wenn das Team sich zu sehr auf Dich verlässt, dann bremst Du effektiv die Fähigkeiten des Teams zur Selbstorganisation aus. Am Anfang mag das okay sein, wenn das Team Scrum neu lernt und Deine Hilfe braucht. Aber mit der Zeit solltest Du langsam zurücktreten von den administrativen Sachen und dem Team mehr und mehr Verantwortung übergeben. Das spart Dir Zeit, die Du für das Aufstöbern und Beseitigen von Hindernissen einsetzen kannst, oder für Tätigkeiten, die nichts mit der Scrum-Master-Rolle zu tun haben, wie Programmieren, Testen und so weiter.

TEIL
ACHTZEHN
Schlusswort

Huch! Ich dachte nicht, dass es so lang werden würde.

Ich hoffe, dieses Buch hat Dir einige nützliche Ideen gezeigt, ob Du jetzt neu mit Scrum beginnst oder ein alter Veteran bist.

Da Scrum individuell an jede Umgebung angepasst werden muss, ist es schwierig, auf einem allgemeinen Level konstruktiv über die besten Alternativen zu diskutieren. Nichtsdestotrotz möchte ich gern Dein Feedback kennenlernen. Sag mir, wie sich Dein Ansatz von meinem unterscheidet. Verrate mir Deine Ideen, wie man etwas verbessern kann!

Fühl Dich eingeladen, mit mir in Kontakt zu treten: **henrik.kniberg@crisp.se**.

> Oh, DAS erklärt, warum ich so viele Emails bekomme… Ich schätze Feedback sehr, aber sei nicht beleidigt, wenn ich nicht antworte. Ich versuche auch noch ein Leben zu haben, manchmal.

Ich behalte außerdem **scrumdevelopment@yahoogroups.com** im Auge.

Wenn Dir dieses Buch gefällt, könnte es sein, dass Du von Zeit zu Zeit auch einen Blick in meinen Blog werfen möchtest. Ich hoffe, dass ich demnächst noch einige Beiträge zu Java und agiler Softwareentwicklung dort hinzufügen werde. **http://blog.crisp.se/henrikkniberg/**.

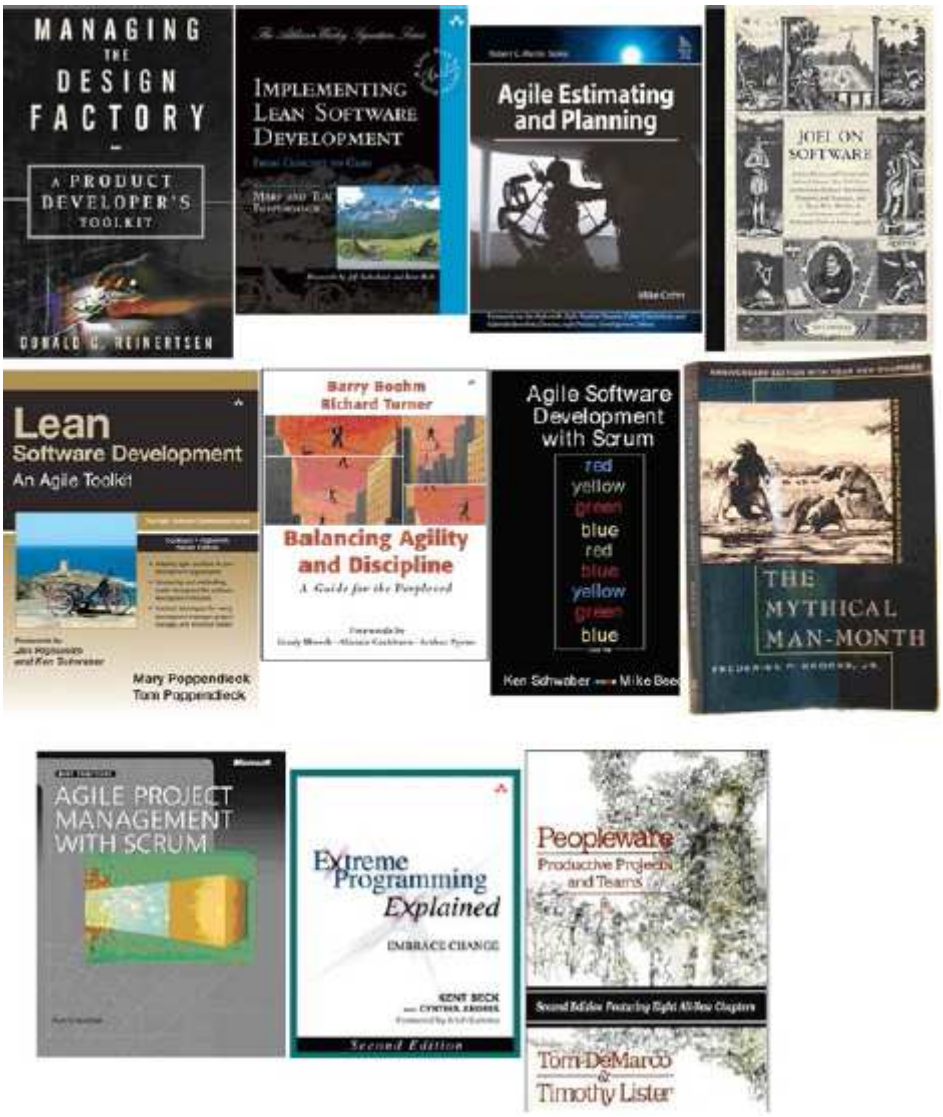
> Es gibt mittlerweile VIELE Artikel und Videos und Sachen in diesem Blog! Nicht nur meine Beiträge, sondern auch die meiner Crisp-Kollegen und verschiedener Partner und Freunde. Es ist eine Goldmine, ehrlich!
>
> Ich bin außerdem von Zeit zu Zeit ziemlich aktiv auf Twitter (@henrikkniberg).

Oh, und vergiss nicht…

Es ist nur ein Job, ja?

Empfohlene Literatur

Hier sind einige Bücher, die mich mit vielen Inspirationen und Ideen versorgt haben. Sehr empfehlenswert!



Äh… Ich sollte das wirklich aktualisieren. Ich habe seit 2007 so viele interessante Bücher gelesen! Aber das wird ein separater Beitrag irgendwann #cliffhanger

Über den Autor

Henrik Kniberg (henrik.kniberg@crisp.se) ist bei Crisp in Stockholm (www.crisp.se) Berater mit den Spezialgebieten Java und agile Softwareentwicklung.

> Heute bin ich mehr so eine Art Managementberater, Organisationsforscher oder Lean Coach / agiler Coach. Im Prinzip helfe ich Firmen zu refaktorisieren, zu debuggen und zu optimieren. Ich programmiere noch von Zeit zu Zeit, aber eher in Art eines Hobbys. Es macht einfach zu viel Spaß!

Von Anfang an, seitdem die ersten XP-Bücher und das agile Manifest erschienen, hieß Henrik die agilen Prinzipien willkommen und versuchte zu lernen, diese in verschiedenen Arten von Organisationen effizient anzuwenden. Als Mitgründer und CTO von Goyada (1998 – 2003) hatte er reichlich Gelegenheit, beim Erstellen und Managen einer technischen Plattform in einem 30-Personen-Team mit testgetriebener Entwicklung und anderen agilen Methoden herum zu experimentieren.

Ende 2005 bekam Henrik einen Vertrag als Entwicklungsleiter einer schwedischen Softwarefirma in der Spielebranche. Die Firma befand sich in einer Krise und hatte ernste organisatorische und technische Schwierigkeiten. Indem er Scrum und XP als Methoden nutzte, half Henrik der Firma durch die Einführung von agilen und Lean-Prinzipien auf allen Organisationsebenen aus der Krise heraus.

Eines Freitags im November 2006 lag Henrik mit Fieber im Bett und kam auf die Idee, schnell einige Notizen für sich selbst darüber zu Papier zu bringen, was er im vergangenen Jahr gelernt hatte. Einmal mit dem Schreiben begonnen, konnte er jedoch nicht mehr aufhören – und nach drei Tagen wilden Tippens und Zeichnens waren die ursprünglichen Notizen zu einem 80-Seiten-Artikel mit dem Titel "Scrum and XP from the Trenches" angewachsen, aus dem schließlich dieses Buch wurde.

> Bin immer noch überrascht, dass ich es geschafft habe, fast das ganze verflixte Ding in einem Wochenende zu schreiben! Andere Autoren hassen mich dafür.

Henrik nimmt sich einen ganzheitlichen Ansatz vor und erfreut sich daran, verschiedene Rollen wie Manager, Entwickler, Scrum Master, Lehrer und Coach einzunehmen. Er ist leidenschaftlich darin, Firmen zu helfen, exzellente Software zu erstellen und exzellente Teams aufzubauen, indem er jedwede Rolle einnimmt, die dafür notwendig ist.

Henrik wuchs in Tokio auf und lebt nun mit seiner Frau Sophia und zwei Kindern in Stockholm.

> Dieselbe Frau und Kinder, nur plus eins. Also Kinder :o)

Er ist in seiner freien Zeit aktiver Musiker, komponiert Musik und spielt Bass und Keyboard in Bands seiner Heimat.

Mehr Infos unter http://www.crisp.se/henrik.kniberg