mehr zum thema:
http://blogs.msdn.com/b/team_foundation/
http://blogs.msdn.com/b/jasonz/about.aspx

CHAOTISCH, FORMAL ODER AGIL: WIE PLANT UND ENTWICKELT EIGENTLICH MICROSOFT?

der autor

Matthias Zieger
(E-Mail: mzieger@microsoft.com)
ist bei der Microsoft Deutschland GmbH im Bereich „Application Lifecycle Management" (ALM) als Berater tätig.

Der Artikel gibt einen Einblick in die Arbeit einer der zahlreichen Entwicklungsabteilungen bei Microsoft. Als Beispiel werden die Planungs- und Entwicklungsprozesse der Microsoft Developer Division dargestellt, die unter anderem Produkte wie „Visual Studio", „.NET" Framework und „Silverlight" entwickelt. Aus welchen Fehlern in der Vergangenheit wurde gelernt, was bedeutet das für eine große, weltweite verteilte Entwicklungsorganisation und welche Verbesserungen wurden in den letzten Jahren mit den neuen Prozessen erzielt?

Bei Microsoft gibt es zwei große Varianten der Softwareentwicklung:

- Die *MSIT* ist verantwortlich für die Entwicklung und Bereitstellung intern verwendeter Software für die Personalabteilung, das Management, den Vertrieb und das Marketing.
- Die *Product Divisions* entwickeln die Produkte für den Kunden, also Windows-Client und -Server, Microsoft Office, SQL Server, Dynamics CRM usw. In einer dieser Divisions, der Microsoft *Developer Devision*, entstehen unter anderem Produkte wie Visual Studio, der Team Foundation Server und das .NET Framework. Die Zahlen aus dem Juli 2010 für die Developer Division in Tabelle 1 verdeutlichen die Größenordnung.

Gesetzmäßigkeiten hinterfragen

„Conway's Law" besagt, dass man einem Produktdesign immer die dahinter liegende Organisationsstruktur ansehen kann (einige einfache Beispiele findet man in Wikipedia, vgl. http://en.wikipedia.org/wiki/Conway's_Law). Das hat auch die Developer Division Anfang des Jahrtausends gespürt. Dem Produkt war anzumerken, welches Entwicklungsteam für welche Funktion zuständig war. Es gab zum Beispiel keine einheitlichen Quality-Gates für den abgelieferten Code. Außer Conway's Law gab es typische Anzeichen dysfunktionaler Prozesse über die Organisationgrenzen hinweg. Hier ein paar Beispiele:

- Auf der Personenebene (z. B. Entwickler, Tester, Projektleiter): „Stelle einem anderen Bereich keine Fragen und antworte nicht, wenn du gefragt wirst".
- Auf der Teilprojektebene: „Die Projektmetriken sind für unseren Bereich nicht gültig, sondern nur für andere".
- Einige Entwickler: „Wir haben das Planen und Entwickeln schon immer so gemacht, seit den 80er Jahren".
- Ganze Produktteams: „Unsere Kunden sind immer anders als eure Kunden".

Das Ergebnis davon war, das die einzelnen Teilprojekte immer mehr Altlasten (vor allem Änderungsanträge und Fehler) mit sich herum trugen, also Arbeit, an die sich keiner traute. Intern heißt das im Microsoft-Sprachgebrauch „Bug Debt" und „Work Debt", also Schulden, die man mit sich trägt. Abbildung 1 zeigt die Altlasten für Visual

Interne Bezeichnung der Metrik	Anzahl	Beschreibung
Active Developers	3.600	Am System registrierte Entwickler, Tester, Programm-Manager, Management, andere Rollen
Work Items	720.000	User-Storys, Anforderungen, Änderungsanträge, Arbeitsaufgaben, Testfälle, gefundene Fehler
Team-Builds pro Monat	2.800	Auf verschiedenen Build-Servern durchgeführte (meist automatisierte) Compile-Deploy-Test-Vorgänge
Files	24.000.000	Einzelne, versionierte Quellcode-Dateien und Skripte, jeweils zusätzlich in mehreren Revisionen vorhanden
Data Size	15 TB	Größe aller versionierten Daten, inklusive Quellcode, Dokumentation, Build-Skripte, Testdaten usw.

Tabelle 1: *Größenordnung der Developer Division bei Microsoft.*

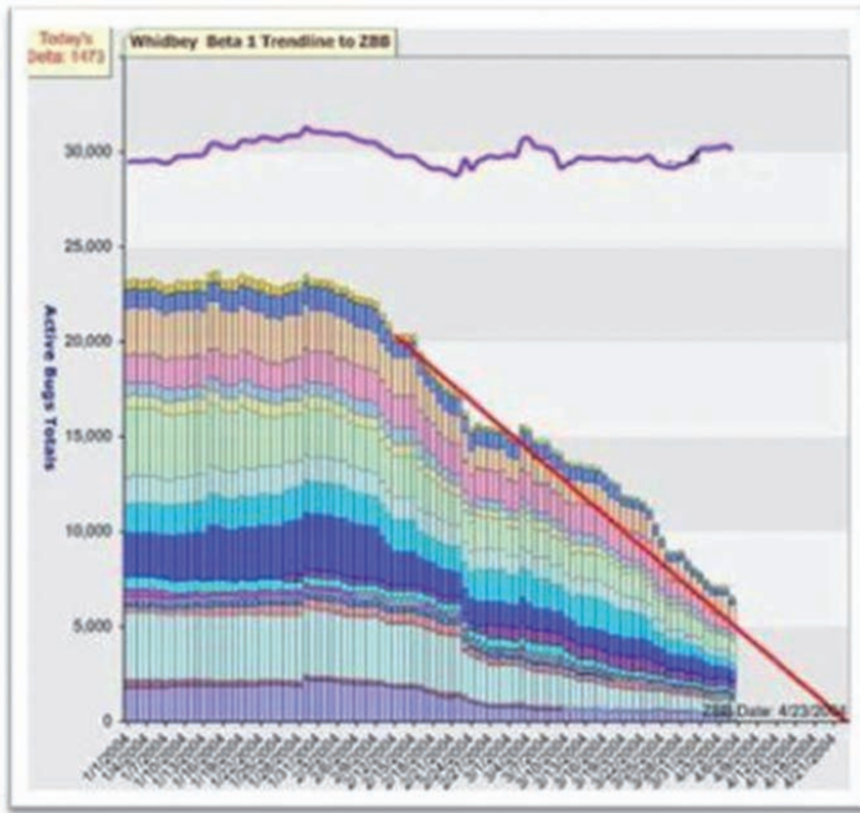


Abb. 1: *Anzahl offener Fehler zum Zeitpunkt Beta1 bei Visual Studio 2005.*

Studio 2005 – der Report ist aus dem Jahr 2004. Interessant ist vor allem die obere Linie, die zeigt, dass die „Deferred Bugs", also die auf unbestimmte Zeit verschobenen Fehler und Änderungsanträge, während der Projektlaufzeit nicht weniger werden. Eine tiefer gehende Analyse der Prozesse und Methoden zeigte folgende Gründe für die massive Anhäufung an aufgeschobener Arbeit:

- Es gab keine einheitlichen Prioritäten der Gesamtorganisation innerhalb der Developer Division.
- Ohne echte Vision, welcher Wert an den Kunden geliefert werden sollte, wurde viel Arbeit doppelt gemacht.
- Vorhandene Features wurden „verschlimmbessert".
- Es gab keine genaue Definition, wann eine Arbeit bzw. Funktionalität abgeschlossen war.
- Durch eine zerstückelte Werkzeuglandschaft war es nicht möglich, durchgängige Auswertungen zum Projektstand zu bekommen

Diese Symptome gab es natürlich nicht nur in der Developer Division, aber in dieser Abteilung wurden die Verbesserungsprozesse zuerst umgesetzt. Hauptziel aller Aktivitäten war die Eliminierung der Altlasten. Dazu wurden einige Prozessverbesserungen geplant und umgesetzt, die ich im Folgenden kurz vorstellen möchte.

Sauber werden

Als erstes ging es darum, vor einer Produktiteration alle Altlasten zu katalogisieren und zu erfassen. Klingt kompliziert, ist aber eigentlich eine grobe Katalogisierung aller aus vorherigen Versionen vorhanden Änderungsanträge und Fehler nach folgenden Kriterien:

- logische Fehler in den Testfällen
- Fehler in der Testautomatisierung, beispielsweise in den Automatisierungsskripten
- tatsächliche Codefehler in der Anwendung
- Fehler in der Entwicklungs- oder Testinfrastruktur
- Dokumentationsfehler

Anschließend konnte man die Altlasten hinsichtlich Kosten und Risiken bewerten. Dadurch entstand eine Priorisierungsliste, die dann abgearbeitet wurde. Mit diesem so genannten *MQ*-Ansatz (*Milestone Quality*) hat man idealerweise zu Beginn der eigentlichen Arbeit am neuen Produkt-Release keine Altlasten aus früheren Versionen mehr.

Sauber bleiben

Als nächstes ging es darum, bewährte, klassische und agile Software-Engineering-Methoden zu neuem Leben zu erwecken.

Iterative Planungen

Bei dem iterativen Planungsansatz werden zwei wichtige Meilensteine vor den Anfang der eigentlichen Entwicklungsaufgaben gestellt. Man beginnt mit dem Meilenstein *MQ*, der mit allen Altlasten aus der Vorversion aufräumt. Danach folgt der Meilenstein *M0*, bei dem definiert wird, an welcher Vision alle gemeinsam arbeiten und welcher Wert für den Endkunden – also den Benutzer – daraus entsteht. Es geht also nicht darum, goldene Wasserhähne zu bauen (was technologisch sicher spannend wäre), sondern die Anforderungen der Kunden und des weltweiten Marktes ganz klar zu verstehen.

Nach dem Meilenstein *M0* starten ca. fünfwöchige Entwicklungszyklen (Sprints). Am Ende eines jeden Sprints soll ein Stück potenziell auslieferbare Software stehen – „potenziell" deswegen, weil wir natürlich nicht alle paar Monate ein neues Release von Visual Studio oder Microsoft Office veröffentlichen können. Aber intern ist die Software als Ergebnis eines Sprints verfügbar. In jedem Sprint wird nicht nur entwickelt, sondern auch permanent getestet.

Diese Sprints gehen dann bis zum ersten *Customer Technology Preview (CTP)*, bei dem das Produkt zum ersten Mal auf eine gezielt ausgesuchte, strategische Kundenbasis trifft. Die CTPs dienen vor allem dazu, ein Feedback von ausgewählten Kunden zu bekommen. Nach den CTP-Iterationen gibt es dann die Beta1-Phase, die dazu dient, Feedback aus einer großen und breiten Kundenbasis zu erhalten. In der Beta2-Phase werden die letzten Änderungen validiert.

Darauf folgen die *RC*-Iterationen (*Release Candidate*), bei denen das Produkt bereits *Feature Complete* ist, d. h. die meisten Funktionsmerkmale sind ent-

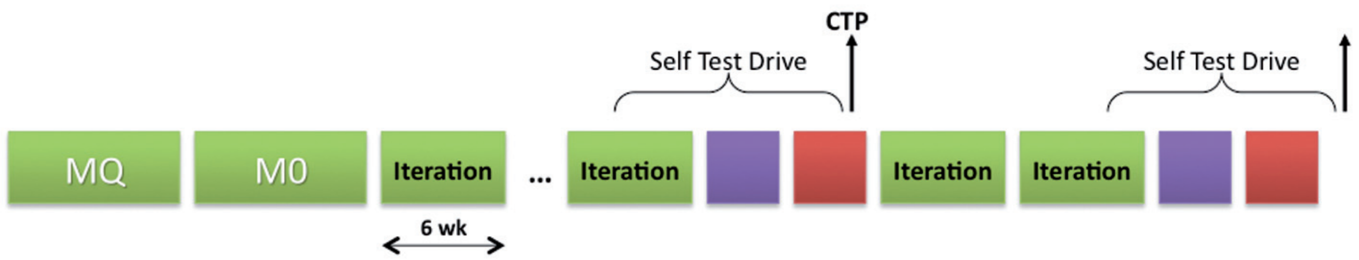


Abb. 2: Überblick über die Release- und Iterationsplanung.

halten, aber es ist auch noch viel Debug-Code in der Codebasis. In der RC-Phase wird weiterhin Kunden-Feedback eingesammelt und Fehler werden behoben, danach ist das Produkt *Ready To Manufactoring (RTM)*: Die Auslieferung wird vorbereitet und es folgen Arbeitsschritte, wie beispielsweise DVD-Master erstellen und die Auslieferungslogistik anstoßen, bis das Produkt dann in den Händen der Kunden bzw. im Download-Portal landet (siehe Abbildung 2).

Danach beginnt dann eine bis zu zehnjährige Support-Phase (je nach Produkt), in der dann im gleichen Rhythmus Service-Packs, R2-Versionen oder – seit Kurzem auch – so genannte Feature-Packs (vgl. [Har10]) „released" werden.

Klare Produktvisionen kommunizieren

Wie oben beschrieben, geht es im Meilenstein *M0*, darum, eine klare Produktvision sowohl an den Endkunden als auch an das Entwicklungsteam zu kommunizieren. Damit ist es aber nicht getan: Für die folgenden Sprints muss man für alle am Projekt beteiligten Rollen Arbeitsaufgaben planen. Wie kann das funktionieren, vor allem, wenn mehrere hundert Produktmerkmale durch mehrere tausend Entwicklungs- und Testaufgaben implementiert werden müssen? Sicher nur durch einen hierarchischen Ansatz.

Ganz oben in der Objekthierarchie stehen dabei die Szenarien. Aber woher kommen diese? Zum einen aus den *Software Design Reviews (SDRs)* – dazu später mehr. Zum anderen wird überlegt, wer der typische Benutzer einer Software ist und welche typischen Anwendungsfälle diese Nutzer brauchen. Dazu verwenden wir so genannte *Personas*, also fiktive Rollen mit den typischen Profilen unserer Kunden. Wie die Personas für das Thema „Testen" aussehen, ist in Abbildung 3 dargestellt.

Aus den Szenarien entwickeln wir dann *Value Propositions*, also die Sichtweise des Kunden. Ein Beispiel für eine solche Value Proposition ist: „Visual Studio 2010 soll Testmanagement-Funktionen enthalten, damit Tester und Entwickler besser miteinander kommunizieren können". Daraus werden dann in letzter Instanz Arbeitspakete gebildet, die von einer Feature-Crew abgearbeitet werden müssen (siehe Abbildung 4).

Das Feature-Crew-Modell

Eine Feature-Crew ist jeweils für ein ganz bestimmtes Feature zu einem Zeitpunkt zuständig. Erst wenn dieses Feature fertig ist (*Feature Complete*), kann die Crew für andere Features eingesetzt werden.

Im Prinzip entspricht es der Idee von Scrum, dass in einem Team alle wichtigen Rollen vorhanden sind. Als interdisziplinäres Team besteht eine Feature-Crew aus:

- einem Programm-Manager
- maximal fünf Entwicklern
- maximal fünf Testern

Bei Microsoft heißen diese *Software Development Engineer in Testing (SDET)*, d. h. es sind Tester, die mindestens Code lesen und verstehen können. Weitere Rollen können nach Bedarf hinzugezogen werden (z. B. UI-Designer und Sicherheitsfachleute). Eine Feature-Crew arbeitet dann gemeinsam an den Sprints mit dem Ziel *Feature Complete*. Im Unterschied zu *Code Complete* umfasst *Feature Complete* zusätzliche Aufgaben, die nicht mit dem reinen Codieren eines Features zusammenhängen, wie z. B. automatisierte Unit-Tests, vollständige Definition der automatisierten UI-Tests sowie Sicherheits- und Bedrohungsanalyse. Außerdem müssen alle definierten Tests auch durchgeführt und alle gefundenen Fehler behoben sein, erst dann ist das Feature fertig, also *Feature Complete*.

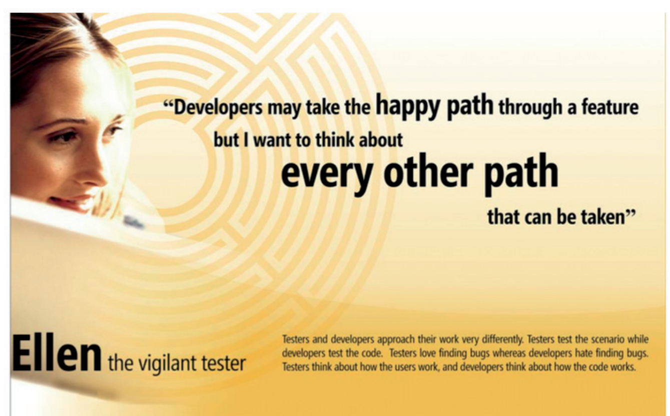


Abb. 3: Persona (Steckbrief) eines Testers (Vor- und Rückseite).

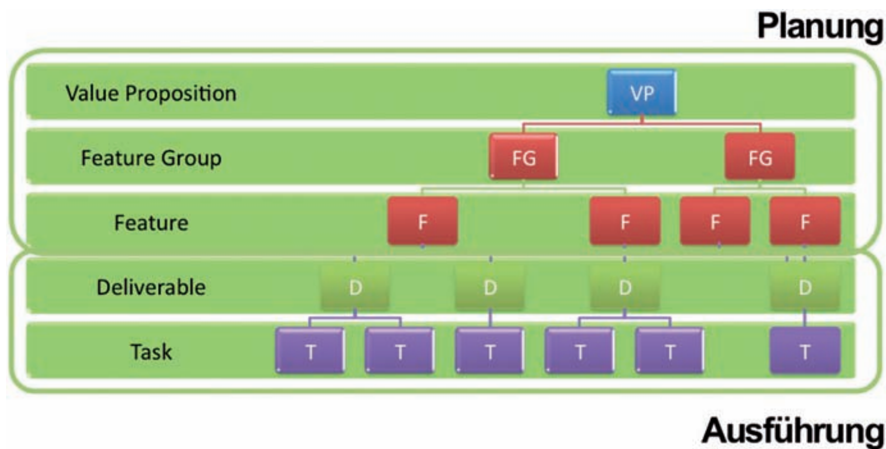


Abb. 4: Ableiten von Arbeitsaufgaben aus den Produktvisionen.

Die Feature-Crews sind übrigens weitestgehend frei in der Wahl des Prozesses, nach dem sie arbeiten: So gibt es von *Scrum* über *Feature Driven Development* bis hin zu *Test Driven Development* alle möglichen (meist agilen) Prozesse auf der Feature-Crew-Ebene. Allerdings gibt es auch Vorgaben, die für alle Feature-Crews verbindlich sind, beispielsweise den Rhythmus der Iterationen, die Verwendung desselben Reporting-Systems sowie die geltenden Qualitätskriterien, die so genannten Quality-Gates, um den Code des Features mit anderem Code zusammenzubringen (*Merging*).

Branching und Isolation des Quellcodes

Eine der Herausforderungen war die sehr große Anzahl an Feature-Crews, die gleichzeitig am einem Produkt-Release arbeiten müssen. Bei Visual Studio sind das etwa 1.200 individuelle Features, die am Ende ein konsistentes Produkt ergeben müssen. Mit all den detaillierten Aktivitäten und Arbeitsaufgaben ist es eine Herausforderung, die Codequalität für alle Features über den gesamten Lebenszyklus hinweg zu gewährleisten. Funktionieren kann das nur, wenn sich alle Teams an den Lebenszyklus für ein Feature (siehe Abbildung 5) halten, der im Wesentlichen auf sechs einfachen Prinzipien beruht. Aber wenn alle das tun, funktioniert das in der Summe sehr gut.

Wenn eine Feature-Crew die Aufgabe bekommt, ein Feature umzusetzen, laufen im Wesentlichen folgende Arbeitsgänge ab:

1. Die Feature-Crew erzeugt einen Code-Zweig aus dem Haupt-Codestrang. Ziel ist hier ganz klar, den neuen Code für die folgende Arbeit zu isolieren. Somit ist die Arbeit entkoppelt von den anderen Features. Hier legt die Crew bereits den Zeitraum für den ersten Feature-Meilenstein-Checkpoint fest und schätzt den Zeitpunkt des zweiten Meilensteines ab.
2. Während der Arbeit an den Features gibt es zwei weitere Meilensteine. Bei diesen Meilensteinen werden die Arbeitsergebnisse dem Management bzw. Stellvertreter des Kunden präsentiert, beispielsweise dem Produktmanager. Das erlaubt auch innerhalb einer Feature-Entwicklung frühzeitig Rückmeldungen und Änderungen. Meilenstein #1 betrifft dabei das Design und die Architektur sowie die Feinplanung der nächsten Arbeitsaufgaben. Danach findet die Umsetzung des Features statt. Bei Meilenstein #1 wird der Zeitraum für Meilenstein #2 festgelegt und das Fertigstellungsdatum abgeschätzt.
3. Meilenstein #2 zeigt dann das weitgehend implementierte Feature und, wie es tatsächlich umgesetzt wurde. Außerdem wird hier das Fertigstellungsdatum endgültig festgelegt
4. Danach holt sich das Team alle Änderungen, die bis dahin am Haupt-Codestrang erledigt wurden, in einen eigenen Feature-Zweig.
5. Bevor der neue Code in den Hauptzweig eingeflochten wird, müssen alle Quality-Gates bestanden werden. Dazu mehr im nächsten Abschnitt.
6. Nur wenn die Quality-Gates durchlaufen sind (mit positivem Ergebnis), darf der Code zurück in den Hauptcodezweig. Der Hauptcodezweig wird also weitestgehend durch die Quality-Gates vor zu viel fehlerhaftem Code geschützt. Damit ist der Hauptzweig fast immer in einem nahezu auslieferungsfähigem Zustand.

Quality Gates

Wann ist ein Feature eigentlich fertig entwickelt? Neben der Funktionalität, die weitestgehend durch automatisierte Tests geprüft wird, gibt es einige weitere Punkte,

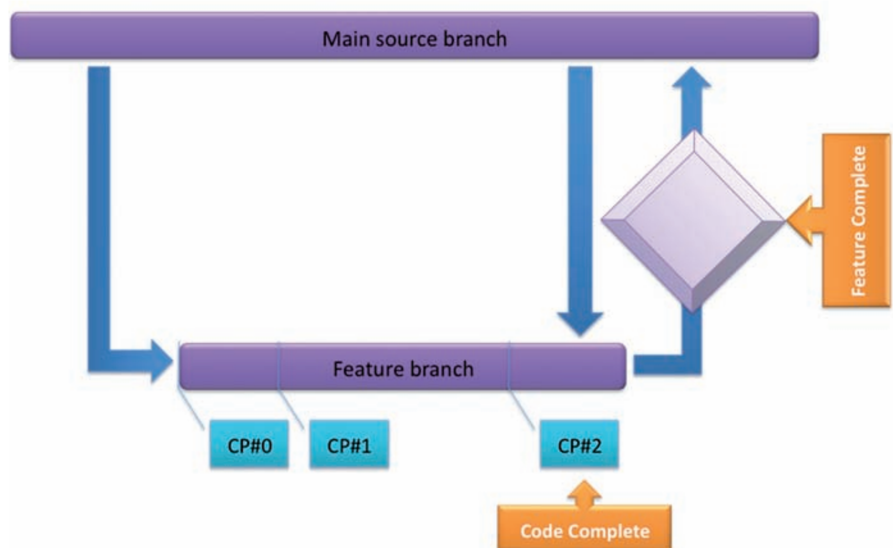


Abb. 5: Lebenszyklus eines Features inkl. der beschrieben Meilensteine (CP).

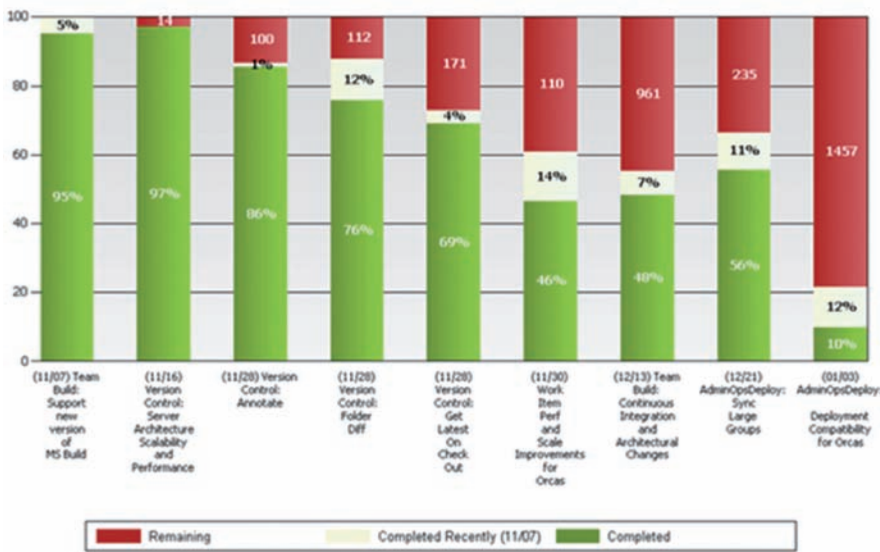


Abb. 6: Übersicht über erledigte, in der letzten Woche geschlossene und verbleibende Aufgaben.

die erledigt sein müssen. Der Code und das Feature müssen eine ganze Reihe so genannter *Quality-Gates* passieren, bevor der Feature Code mit dem Code anderer Feature zusammenkommen darf. Hier eine kurze Auswahl der wichtigsten Quality-Gates:

- 70 % Codeabdeckung bei automatisierten Tests.
- Sicherheitsplan ist OK.
- Statische Codeanalyse wurde durchgeführt.
- Codierungsrichtlinien sind eingehalten.
- Keine Performance-Regression (d. h. von einer Iteration auf die nächste darf ein Feature nicht langsamer werden).
- Lokalisierungstests bestanden, läuft auf allen unterstützten Sprachversionen.
- API-Reviews sind ohne Beanstandung.
- Alle bekannten (gefundenen) Bugs sind behoben.

Daneben gibt es weitere Quality-Gates, die einen eher nicht-technischen Charakter haben, wie z. B. Dokumentation und Einhaltung der Richtlinien bezüglich geistigen Eigentums.

Erst wenn wirklich alle Quality-Gates erfüllt sind, ist das Feature fertiggestellt und es kann mit anderen Features in Berührung kommen – das heißt, der Code wird zusammengeführt. Damit sind Quality-Gates die zentrale Methode, die sicherstellt, dass mehr als 3.000 Menschen an vielen Features gleichzeitig an mehreren Millionen Codefragmenten arbeiten können und dass trotzdem die Qualität des Gesamtsystems gewahrt bleibt.

Für Transparenz sorgen

Neben einer höheren Qualität der Produkte war es eines der Hauptziele bei der Änderung der Entwicklungsprozesse, eine bessere Transparenz vor allem in Bezug auf die Vorhersagbarkeit von Endterminen zu erhalten. Dazu ist ein Projektmanagement auf zwei Ebenen notwendig: auf der Ebene des Features und – natürlich für die Produktmanager, die Kunden und das Management – auf der Produktebene.

Auf der *Feature-Ebene* werden vor allem folgende Daten erfasst: Start- und Ende-Datum der Feature-Crews und die beiden Meilensteine sowie Grad der Fertigstellung – im Prinzip erledigte und geschätzte verbleibende Arbeit. Während die Feature-Crews in der Wahl ihrer Prozesse frei sind, müssen sie einen stabilen Weg finden, um das Verhältnis zwischen erledigter und verbleibender Arbeit abzuschätzen. Das Risikomanagement ist ein Ampelsystem, das auf mehreren Checkpoints basiert (grün = wir werden die Zeiten einhalten, gelb = es gibt Risiken, die Zeiten einzuhalten, rot = die Zeiten können nicht eingehalten werden). Dazu gibt es eine subjektive Einschätzung der Feature-Crew in Form von textuellen Beschreibungen über den Stand der Dinge.

Auf der *Produkt-Ebene* werden die vorhanden Feature-Daten automatisiert konsolidiert und in verschiedenen Reports dargestellt. Jede Woche gibt es ein kurzes Meeting, das von einem Programm-Manager geleitet wird. Dieser nutzt für das Meeting einen Report, der alle laufenden Feature-Crews anzeigt (siehe Abbildung 6). Die Farben in der Abbildung haben die folgende Bedeutung: grün = erledigte Arbeit im Verhältnis zum Gesamt-Feature, hellgrün = erledigte Arbeit seit dem letzten Meeting, in der Regel die erledigte Arbeit der letzten Woche, rot = verbleibende geschätzte Arbeitsstunden. Außerdem ist unter dem Chart das Fertigstellungsdatum

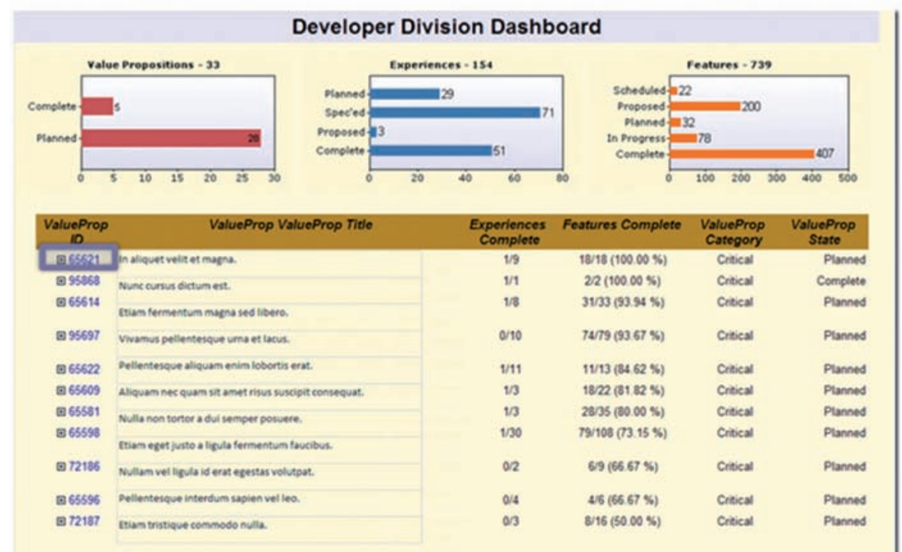
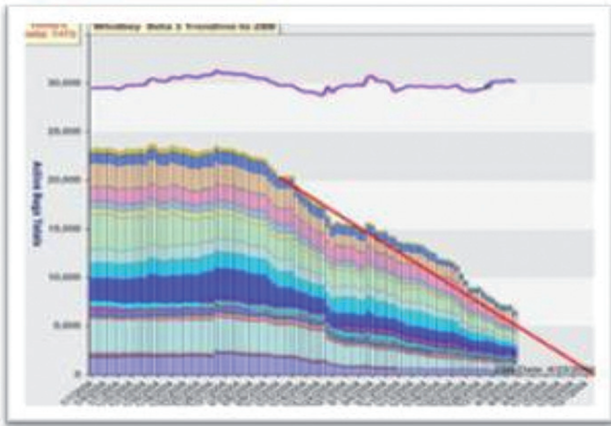


Abb. 7: Gesamtüberblick über den Produktstatus.



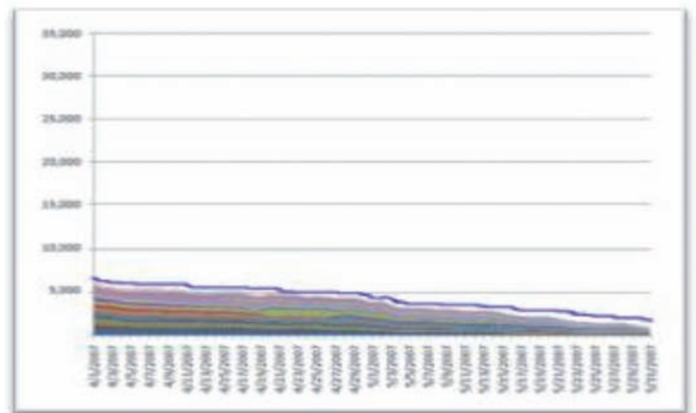


Abb. 8: *Vergleich der Produktqualität zum Zeitpunkt der Beta 1 bei Visual Studio 2005 (links) und Visual Studio 2008 (rechts).*

zu finden. Daraus lassen sich folgende Fragen an die Feature-Crews ableiten:

- Warum gab es letzte Woche keinen Fortschritt (zweite von links)?
- Warum wurde nur so wenig Arbeit erledigt (dritte von links)?
- Ihr habt noch einen Aufwand von 1.475 Stunden geschätzt, wollt aber in vier Wochen fertig sein – schafft ihr das noch?

Mit diesen Fragen soll eine Diskussion angestoßen werden. Es geht darum, Gründe zu finden, warum etwas nicht gut läuft, festzustellen, ob man die Schätzmethoden der Feature-Crew überprüfen muss usw. Es geht ausdrücklich nicht darum, Schuldzuweisungen vorzunehmen. Dazu ist es notwendig, dass alle immer auf die aktuellen Projektmetriken zugreifen können, die während der Entwicklung ständig aktualisiert werden. Risiken und Projekthindernisse und natürlich die Softwarequalität werden auf die gleiche Art und Weise aufbereitet und diskutiert. Das Ergebnis sind Dashboards (ein Beispiel zeigt Abbildung 7), die einen Gesamtüberblick über die Developer Division geben können.

Damit das funktionierte, musste natürlich ein Change-Management-Prozess angestoßen werden, der vor allem die Projekt- und Unternehmenskultur betrifft. In einer Kultur der Offenheit ist kein Platz für Managementmethoden, die beispielsweise vorschreiben: „Das Feature ist unter allen Umständen bis dahin fertigzustellen". Genauso wenig ist es akzeptabel, wenn Entwickler auf eine Statusanfrage beispielsweise regelmäßig wie folgt antworten: „Ich bin bald fertig, ca. 80 % sind erledigt". Beide Seiten mussten dazu lernen.

Das Ergebnis

Es stellt sich natürlich die Frage, ob sich die ganzen Anstrengungen gelohnt haben. Schließlich wurden nicht nur organisatorische Aspekte geändert, sondern auch Themen wie Reporting und Zeiterfassung angepasst bzw. ganz anders durchgeführt. Außerdem wurde die gesamte Tool-Landschaft aus zerstückelten Einzelwerkzeugen in eine integrierte Werkzeugkette („Team Foundation Server") überführt. Damit war ein disziplinübergreifendes Reporting möglich.

Aus einem Vergleich mit dem Vorgängerprodukt wurden ganz konkret folgende Verbesserungen erzielt:

- Die Altlasten wurden um 90 % verringert.
- Fehler und Änderungsanträge mit dem Status „auf später verschoben" wurden weitestgehend eliminiert.
- Die Vorhersagbarkeit der Entwicklungszeiten wurde wesentlich verbessert.
- Die Entwicklungszeit konnte stark gekürzt werden, hauptsächlich durch Einsparung von verschwendeter Arbeitszeit.
- Die Kundenzufriedenheit wurde erhöht.

Abbildung 8 zeigt einen Vergleich der Altlasten in Visual Studio 2005 (links, alter Prozess) und Visual Studio 2008 (rechts, neuer Prozess und neue Tools).

Fazit

Die folgende Checkliste kann Ihnen in eigenen Projekten – unabhängig von der Größenordnung – helfen, signifikante Verbesserungen zu erzielen. Dabei muss nicht alles sofort umgesetzt werden, sondern es empfiehlt sich, eine iterative Vorgehensweise zu nutzen. Außerdem muss jede Organisation für sich entscheiden, welche der folgenden Punkte wichtig sind:

- Altlasten vermeiden
- an *einer* Vision arbeiten
- Transparenz herstellen
- Features und deren Code isolieren
- Design von unten (*Feature Driven Design*)
- iterativ vorgehen (*Time Boxing*)
- ständig Software liefern (*Continuous Integration*)
- Quality-Gates und Tests weitestgehend automatisieren (Unit-Tests, GUI-Tests, Performance-Tests)
- richtige Teamgrößen etablieren (nicht zu groß, aber auch keine Einzelkämpfer)
- Kunden-Feedback frühzeitig einholen
- messbare Daten und Metriken definieren und damit für Transparenz sorgen

Meine Eingangsfrage lautete, wie Microsoft entwickelt. Es sollte deutlich werden, dass wir auf keinen Fall chaotisch unterwegs sind. In der kleinsten Einheit – der Feature Crew – arbeiten wir agil und nutzen zahlreiche agile *Best Practices*, wie kontinuier-

liche Integration, Unit-Testing, Time-Boxing und die enge Integration von Entwicklern und Testern in kleinen autarken Teams. Natürlich lässt sich eine Organisation von mehreren tausend Menschen nicht ohne Formalismen steuern. Diese spiegeln sich hauptsächlich in den Auswertungen und Reports wider, die für einen notwendigen Gesamtproduktüberblick sorgen.

Literatur & Links

[Har10] B. Harry, What on Earth is a Feature Pack?, 2010, siehe: http://blogs.msdn.com/b/bharry/archive/2010/06/07/what-on-earth-is-a-feature-pack.aspx

[Pag08] A. Page, K. Johnston, How we Test at Microsoft, Microsoft Press 2008

[Sin10] S. Sinofsky, One Strategy: Organization, Planning, and Decision Making, John Wiley & Sons 2010