9 Pragmatische Projekte

Wenn Ihr Projekt in Gang kommt, müssen wir uns von Themen wie individuelle Philosophie und Implementierung lösen und größere, projektweite Themen betrachten. Wir werden nicht auf die Einzelheiten des Projektmanagements eingehen, aber eine Handvoll kritischer Bereiche behandeln, die über Wohl und Wehe eines jeden Projekts entscheiden können.

Sobald mehrere Personen an einem Projekt arbeiten, muss man einige Grundregeln aufstellen und Teile des Projekts entsprechend delegieren. In *Topic 49*, *Pragmatische Teams*, werden wir zeigen, wie man dabei der pragmatischen Philosophie folgt.

Der Zweck einer Software-Entwicklungsmethode ist es, Menschen bei der Zusammenarbeit zu unterstützen. Tun Sie und Ihr Team das, was für Sie gut funktioniert, oder investieren Sie nur in triviale Oberflächen-Artefakte und erhalten nicht die echten Vorteile, die Sie verdienen? Wir werden sehen, was es mit *Kokosnüsse bringen's nicht* (*Topic 50*) auf sich hat, und das wahre Geheimnis des Erfolgs lüften.

Und natürlich spielt all das keine Rolle, wenn Sie Software nicht konsistent und zuverlässig liefern können. Die Grundlage des magischen Trios aus Versionsverwaltung, Testen und Automatisierung finden Sie in *Topic 51*, *Pragmatic Starter Kit*.

Letztlich misst sich der Erfolg im Auge des Betrachters, dem Auftraggeber des Projekts. Die Wahrnehmung des Erfolgs ist das, was zählt, und in *Topic 52*, *Erfreuen Sie die Anwender*, zeigen wir Ihnen einige Tricks, um jeden Auftraggeber zu erfreuen.

Der letzte Tipp in diesem Buch ist eine direkte Folgerung aus den übrigen. In *Topic 53*, *Stolz und Vorurteil*, ermutigen wir Sie dazu, Ihr Werk zu signieren und stolz auf Ihr Tun zu sein.

Topic 49: Pragmatische Teams

49

Stoffel führt Gruppe L – sechs erstklassige Programmierer.
Das ist eine Management-Herausforderung, die dem Hüten von Katzen entspricht.

The Washington Post Magazine, 9. Juni 1985

Schon 1985 war der Witz über das Hüten von Katzen alt. Zum Zeitpunkt der Erstausgabe um die Jahrtausendwende war sie geradezu steinalt. Dennoch besteht sie fort, weil sie einen wahren Beiklang hat. Programmierer sind ein bisschen wie Katzen: intelligent, willensstark, rechthaberisch, unabhängig und werden oft vom Netz angebetet.

Bisher haben wir in diesem Buch pragmatische Techniken betrachtet, die einem Einzelnen helfen, ein besserer Programmierer zu werden. Können diese Methoden auch für Teams funktionieren, selbst für Teams von willensstarken, unabhängigen Menschen? Die Antwort ist ein klares „Ja"! Es hat Vorteile, als Einzelner pragmatisch zu sein, und diese Vorteile vervielfachen sich, wenn der Einzelne in einem pragmatischen Team arbeitet.

Ein Team ist unserer Ansicht nach eine kleine, weitgehend stabile eigene Einheit. Fünfzig Leute sind kein Team, sie sind eine Horde.[1] Teams, in denen Mitglieder ständig für andere Aufträge abgezogen werden und keiner den anderen kennt, sind auch kein Team; sie sind lediglich Fremde, die sich vorübergehend im Regen an einer Bushaltestelle unterstellen.

Ein pragmatisches Team ist klein, weniger als etwa 10–12 Mitglieder. Mitglieder wechseln eher selten. Jeder kennt jeden gut und vertraut dem anderen, alle sind aufeinander angewiesen.

Tipp 84

Arbeiten Sie auf kleine, stabile Teams hin.

In diesem Abschnitt werden wir kurz betrachten, wie pragmatische Techniken auf ein Team als Ganzes angewendet werden können. Diese Notizen sind nur der Anfang. Sobald man eine Gruppe pragmatischer Software-Entwickler hat, die in einer aufgeschlossenen Umgebung arbeiten, werden sie schnell ihre eigene funktionierende Teamdynamik entwickeln und verbessern.

Lassen Sie uns die vorangegangenen Abschnitte im Zusammenhang mit Teams noch einmal betrachten.

Keine zerbrochenen Fensterscheiben

Qualität ist eine Team-Angelegenheit. Selbst der gewissenhafteste Programmierer wird es in einem ignoranten Team schwer haben, den notwendigen Enthusiasmus aufrechtzuerhalten, um die drängenden Aufgaben zu lösen. Das Problem wird noch verschärft, wenn das Team dem Programmierer den Mut nimmt, Zeit für diese Dinge aufzubringen.

Teams als Ganzes sollten keine zerbrochenen Fensterscheiben tolerieren – diese kleinen Unzulänglichkeiten, die niemand in Ordnung bringt. Das Team muss die Verantwortung für die Qualität des Produkts übernehmen. Es muss die Programmierer unterstützen, die hinter der „Keine zerbrochenen Fensterscheiben"-Philosophie stehen (siehe *Topic 3*, *Software-Entropie*), und auch diejenigen ermutigen, die sie noch nicht für sich entdeckt haben.

Manche Teammodelle haben einen Qualitätsbeauftragten. Jemand, an den das Team die Verantwortung für die Qualität ihrer Ergebnisse delegiert. Das ist absolut lächerlich: Qualität kann nur aus den individuellen Beiträgen aller Teammitglieder entstehen. Qualität ist eingebaut, nicht angeflanscht.

[1] Mit zunehmender Teamgröße wachsen die Kommunikationswege mit der Geschwindigkeit $O(n^2)$, wobei n die Anzahl der Teammitglieder ist. In größeren Teams beginnt die Kommunikation zusammenzubrechen und wird ineffektiv.

Gekochte Frösche

Erinnern Sie sich an den armen Frosch in dem Topf mit Wasser in *Topic 4*, *Steinsuppe und gekochte Frösche*? Er bemerkt die schrittweise Veränderung in seiner Umgebung nicht und wird gekocht. Dasselbe kann mit Einzelnen passieren, die nicht wachsam sind. Es ist manchmal ganz schön schwer, in der Hitze des Projekts die Augen für die eigene Umgebung offen zu halten.

Es ist sogar noch leichter, sich als ganzes Team zu verbrühen. Die Team-Mitglieder nehmen an, es werde sich schon jemand anders um das Problem kümmern oder dass der Teamleiter die angeforderte Änderung abgesegnet habe. Selbst Teams mit den besten Absichten können bedeutende Veränderungen in ihrer Umgebung übersehen.

Verhindern Sie das. Stellen Sie sicher, dass jeder seine Umgebung aktiv auf Veränderungen beobachtet. Bleiben Sie wachsam, falls sich der Umfang vergrößert, die Zeiträume verkürzen, zusätzliche Funktionen erwünscht sind, neue Umgebungen hinzukommen – alles, was nicht in der ursprünglichen Abstimmung vorhanden war. Führen Sie Metriken für neue Anforderungen.[2] Das Team muss Änderungen nicht einfach kurzerhand ausschlagen, Sie müssen sich nur darüber im Klaren sein, dass diese Veränderungen stattfinden. Andernfalls sind *Sie* derjenige, der im heißen Wasser sitzt.

Planen Sie Ihr Wissensportfolio

In *Topic 6*, *Ihr Wissensportfolio*, haben wir untersucht, wie Sie in Ihrer Freizeit in das persönliche Wissensportfolio investieren sollten. Teams, die Erfolg haben wollen, müssen auch ihre Investitionen in Wissen und Fähigkeiten berücksichtigen.

Wenn Ihr Team ernsthaft an Verbesserung und Innovation interessiert ist, müssen Sie einen Zeitplan dafür aufstellen. Der Versuch, das „immer dann zu erledigen, wenn man mal Zeit hat“, bedeutet, *dass es niemals geschieht*. Mit welcher Art von Backlog oder Aufgabenliste oder Flow Sie auch arbeiten, reservieren Sie dieses Arbeitsmittel nicht nur für die Entwicklung von Features. Das Team arbeitet an mehr als nur neuen Funktionen. Einige mögliche Beispiele sind diese:

Wartung alter Systeme. Während wir es lieben, an dem schicken neuen System zu arbeiten, gibt es wahrscheinlich auch Wartungsarbeiten, die am alten System durchgeführt werden müssen. Wir haben Teams getroffen, die versuchen, diese Arbeiten möglichst zurückzustellen. Wenn das Team mit diesen Aufgaben betraut ist, dann erledigen Sie sie auch wirklich.

Prozessreflexion und -verfeinerung. Kontinuierliche Verbesserung kann nur geschehen, wenn man sich die Zeit nimmt, sich umzusehen und herauszufinden, was funktioniert und was nicht. Dann geht man Änderungen an (siehe *Topic 48*, *Das Wesen der Agilität*). Zu viele Teams sind so sehr mit Wasser schöpfen beschäftigt, dass sie keine Zeit haben, das Leck zu reparieren. Planen Sie das. Bringen Sie es in Ordnung.

Neue Tech-Experimente. Nehmen Sie keine neuen Technologien, Frameworks oder Bibliotheken an, bloß weil „jeder das macht“ oder weil es auf etwas basiert, das Sie online gelesen oder auf einer Konferenz gesehen haben. Prüfen Sie gezielt potenzielle Kandidaten für neue Technologien anhand von Prototypen. Setzen Sie Aufgaben auf den Zeitplan, um die neuen Dinge auszuprobieren und die Ergebnisse zu analysieren.

---

[2] Ein *Burn-up-Diagramm* ist dafür besser geeignet als das üblichere *Burn-down-Diagramm*. Mit einem Burn-up-Diagramm können Sie deutlich erkennen, wie die zusätzlichen Merkmale die Zieltermine verschieben.

Verbesserung der Lernfähigkeiten und Skills. Persönliches Lernen und Verbesserungen sind ein guter Anfang, aber viele Skills sind effektiver, wenn sie teamweit verbreitet werden. Planen Sie das ein, sei es bei informellen Treffen wie der gemeinsamen Kaffeepause oder formelleren Trainingseinheiten.

Tipp 85

Planen Sie es ein, damit es passiert.

Teampräsenz kommunizieren

Es ist naheliegend, dass Entwickler in einem Team miteinander reden müssen. Wir haben einige Vorschläge dafür in *Topic 7*, *Kommuniziere!*, gemacht. Nur zu leicht vergisst man, dass auch jedes Team in der Organisation eigenständig auftritt. Das Team als Ganzes muss klar und deutlich mit der Außenwelt kommunizieren.

Für Außenstehende sind die unangenehmsten Projektteams diejenigen, die mürrisch und wortkarg auftreten. Besprechungen dieser Teams haben keine klare Linie, und niemand möchte etwas sagen. Ihre E-Mails und Projektdokumente sind eine Schande, denn keines sieht aus wie das andere und jedes verwendet eine andere Terminologie.

Großartige Projektteams haben Charakter. Man freut sich auf Besprechungen mit ihnen, weil man einen gut vorbereiteten Beitrag erwartet, der alle zufriedenstellt. Ihre erstellte Dokumentation ist knapp, präzise und konsistent. Das Team spricht mit einer Stimme.[3] Vielleicht hat es sogar Sinn für Humor.

Es gibt einen einfachen Marketing-Trick, der Teams hilft, als Einheit zu kommunizieren: Wenn Sie mit einem Projekt beginnen, sollten Sie einen Namen dafür erfinden, idealerweise einen seltsamen (in der Vergangenheit haben wir Projekte nach Killer-Papageien, die sich Schafe als Opfer aussuchen, optischen Täuschungen oder mystischen Städten benannt). Nehmen Sie sich 30 Minuten, um ein verrücktes Logo zu entwerfen, und verwenden Sie es in Mitteilungen und Berichten. Setzen Sie den Namen Ihres Teams bewusst ein, wenn Sie mit Leuten reden. Es mag albern klingen, gibt aber Ihrem Team eine Identität, auf die es bauen kann, und die Welt kann sich zu Ihrer Arbeit etwas Unverwechselbares einprägen.

Wiederhole dich nicht

In *Topic 9*, *Die Übel der Wiederholung*, haben wir über die Schwierigkeiten geredet, doppelte Arbeit unter Teammitgliedern zu vermeiden. Diese Wiederholung führt zu unnötigem Aufwand und vermutlich auch zu einem Wartungsalbtraum. „Ofenrohr"- oder „Silo"-Systeme sind in diesen Teams weit verbreitet, mit wenig gemeinsamer Nutzung und vielen doppelten Funktionen.

Gute Kommunikation ist der Schlüssel zur Vermeidung dieser Probleme. Und mit „gut" meinen wir *sofort* und *reibungslos*.

Sie sollten in der Lage sein, Teammitgliedern eine Frage zu stellen und zeitnah eine Antwort zu erhalten. Wenn das Team räumlich beieinander arbeitet, kann das z. B. so einfach sein wie mal über die Trennwand der Arbeitsplätze zu schauen oder sich auf dem Flur zu treffen.

[3] Das Team spricht nach außen hin mit einer Stimme. Nach innen wollen wir zu lebendigen, harten Debatten ermutigen. Gute Entwickler sind manchmal ziemlich leidenschaftlich bei ihrer Arbeit.

Bei Remote-Teams müssen Sie sich möglicherweise auf eine Messaging-App oder andere elektronische Mittel verlassen.

Wenn Sie eine Woche auf die Teambesprechung warten müssen, um Ihre Frage zu stellen oder Ihren Status mitzuteilen, ist das eine Menge Reibung.[4] Reibungslos bedeutet, dass einfach und ohne großen Aufwand Fragen zu stellen sind und dass man Fortschritt, Probleme, Einsichten und Erkenntnisse mitteilen kann und weiß, was die Teamkollegen tun.

Fördern Sie das Bewusstsein dafür, DRY zu bleiben.

Leuchtspurmunition fürs Team

Ein Projektteam hat viele verschiedene Aufgaben in verschiedenen Bereichen des Projekts zu erfüllen, wobei viele verschiedene Technologien ins Spiel kommen. Ein Verständnis der Anforderungen, für den Entwurf der Architektur, die Entwicklung von Frontend und Server, für das Testen, all das muss umgesetzt werden. Aber es ist ein weit verbreiteter Irrtum, dass diese Aktivitäten und Aufgaben getrennt voneinander und isoliert stattfinden können. Dem ist nicht so.

Einige Methoden befürworten, dass im Team selbst alle möglichen unterschiedlichen Rollen und Titel vorhanden sind, oder schaffen separate spezialisierte Teams. Das Problem bei diesem Ansatz ist jedoch, dass er *Schranken* und *Übergabepunkte* einführt. Statt eines reibungslosen Ablaufs vom Team zum Deployment haben Sie jetzt künstliche Tore, an denen die Arbeit aufhört. Übergaben, auf deren Akzeptanz man warten muss. Genehmigungen. Papierkram. Die Lean-Profis nennen das *Verschwendung* und bemühen sich, so etwas aktiv zu beseitigen.

All diese verschiedenen Rollen und Aktivitäten sind in Wirklichkeit unterschiedliche Ansichten desselben Problems, und ihre künstliche Trennung kann eine Menge Ärger verursachen. Beispielsweise ist es unwahrscheinlich, dass Programmierer, die zwei oder drei Ebenen von den tatsächlichen Anwendern ihres Codes entfernt sind, sich des Kontexts bewusst sind, in dem ihre Arbeit verwendet wird. Sie sind daher nicht in der Lage, fundierte Entscheidungen zu treffen.

In *Topic 12*, *Leuchtspurmunition*, empfehlen wir die Entwicklung einzelner, wenn auch zunächst kleiner und begrenzter Funktionen, die sich durch das gesamte System ziehen. Das bedeutet, dass Sie alle Fähigkeiten benötigen, um dies innerhalb des Teams zu tun: Frontend, UI/UX, Server, DBA, QS usw., und alle können reibungsarm und gut aufeinander abgestimmt miteinander daran arbeiten. Mit dem Leuchtspurmunition-Ansatz können Sie sehr kleine Funktionalitäten sehr schnell implementieren und erhalten sofortiges Feedback darüber, wie gut Ihr Team kommuniziert und liefert. Das schafft eine Umgebung, in der Sie Änderungen umsetzen und Feinabstimmungen für Ihr Team und Ihre Prozesse schnell und einfach vornehmen können.

Tipp 86

Organisieren Sie voll funktionsfähige Teams.

Stellen Sie Teams zusammen, damit Sie den Code durchgängig, inkrementell und iterativ erstellen können.

[4] Andy hat Teams getroffen, die ihre täglichen Scrum-Stand-ups am Freitag durchführten.

Automatisierung

Alle Tätigkeiten des Teams zu automatisieren, ist eine großartige Möglichkeit, um sowohl Konsistenz als auch Genauigkeit zu sichern. Warum sich mit Code-Formatierungsstandards abmühen, wenn Ihr Editor oder die IDE dies automatisch für Sie erledigen kann? Warum manuelle Tests durchführen, wenn der kontinuierliche Build Tests automatisch ausführen kann? Warum von Hand deployen, wenn die Automatisierung dies jedes Mal auf die gleiche Weise, wiederholbar und zuverlässig tun kann?

Automatisierung ist für jedes Projektteam unverzichtbar. Stellen Sie sicher, dass das Team über Fähigkeiten verfügt, um *Werkzeuge* zur automatischen Projektentwicklung und Produktionseinsatz zu konstruieren und einzusetzen.

Wissen, wann genug Farbe drauf ist

Denken Sie daran, dass die Teams aus Individuen bestehen. Geben Sie jedem Mitglied die Möglichkeit, auf seine oder ihre Art zu glänzen. Sie müssen ihnen nur so viel Struktur vorgeben, wie nötig ist, um sie zu unterstützen, damit das Projekt Werte liefert. Dann müssen Sie der Versuchung widerstehen, noch mehr Farbe drauf zu tun – wie der Maler in *Topic 5, Gut ist gut genug*.

Verwandte Topics

- Topic 2, *Der Hund hat meinen Quelltext gefressen*
- Topic 7, *Kommuniziere!*
- Topic 12, *Leuchtspurmunition*
- Topic 19, *Versionsverwaltung*
- Topic 50, *Kokosnüsse bringen's nicht*
- Topic 51, *Pragmatic Starter Kit*

Aufgaben

- Betrachten Sie erfolgreiche Teams außerhalb der Software-Entwicklung. Was macht sie erfolgreich? Verwenden sie irgendein Vorgehen aus diesem Topic?
- Versuchen Sie zu Beginn Ihres nächsten Projekts, die Leute davon zu überzeugen, es zu einer Marke zu machen. Geben Sie Ihrer Organisation Zeit, sich daran zu gewöhnen, und machen dann eine kurze Umfrage, welchen Unterschied es im Team und außerhalb bedeutet hat.
- Wahrscheinlich wurden Ihnen mal Aufgaben gestellt wie „Wenn 4 Arbeiter 6 Stunden brauchen, einen Graben auszuheben, wie lange würden dann 8 Arbeiter brauchen?". Im wirklichen Leben beeinflussen welche Faktoren die Antwort, wenn die Arbeiter stattdessen Code schreiben würden? In welchen Fällen wird die Zeit tatsächlich reduziert?
- Lesen Sie *Vom Mythos des Mann-Monats* [Bro96] von Frederick Brooks. Wollen Sie ein Extrasternchen, kaufen Sie zwei Exemplare, um es doppelt so schnell lesen zu können.

Topic 50: Kokosnüsse bringen's nicht

50

Die einheimischen Inselbewohner hatten noch nie zuvor ein Flugzeug gesehen oder Menschen wie diese Fremden getroffen. Als Gegenleistung für die Nutzung ihres Landes stellten die Fremden mechanische Vögel zur Verfügung, die den ganzen Tag über auf einer „Landebahn" ein- und ausflogen und unglaublichen materiellen Reichtum in ihre Inselheimat brachten. Die Fremden erwähnten irgendetwas über Krieg und Kämpfe. Eines Tages war es vorbei, und die Fremden verschwanden und nahmen alle ihre seltsamen Reichtümer mit.

Die Inselbewohner versuchten verzweifelt, ihr Glück wiederherzustellen, und bauten den Flughafen mit dem Kontrollturm und der Ausrüstung nach. Sie schufen diese Attrappe mit lokalen Materialien wie Weinreben, Kokosnussschalen oder Palmwedel. Aber aus irgendeinem Grund kamen keine Flugzeuge mehr, obwohl sie alles richtig vorbereitet hatten. Sie hatten die Form nachgeahmt, aber nicht den Inhalt. Anthropologen bezeichnen dies als „Cargo-Kult".

Allzu oft sind *wir* die Inselbewohner.

Es ist einfach und verlockend, in die Cargo-Kult-Falle zu tappen: Man investiert in die leicht sichtbaren Artefakte, erbaut sie und hofft so, die zugrunde liegende, wirkungsvolle Magie anzuziehen. Aber wie bei den originalen Cargo-Kulten Melanesiens[5] ist ein nachgebauter Flughafen aus Kokosnussschalen kein Ersatz für den echten.

Zum Beispiel haben wir selbst Teams getroffen, die behaupteten, Scrum zu verwenden. Bei genauerem Hinsehen stellte sich jedoch heraus, dass sie einmal pro Woche ein Daily Stand-up-Meeting mit vierwöchigen Iterationen abhielten, aus denen oft sechs- oder achtwöchige Iterationen wurden. Sie waren der Meinung, dass dies in Ordnung sei, weil sie ein beliebtes „agiles" Terminplanungsinstrument verwendeten. Sie investierten nur in die oberflächlichen Artefakte – oft nur dem Namen nach, als ob „Stand up" oder „Iteration" eine Art Beschwörungsformel für Abergläubische wäre. Es überrascht nicht, dass es auch ihnen nicht gelungen ist, die wahre Magie anzuziehen.

Kontext ist wichtig

Sind Sie oder Ihr Team in diese Falle geraten? Fragen Sie sich, warum Sie überhaupt diese spezielle Entwicklungsmethode anwenden. Oder dieses Framework? Oder jene Testtechnik? Eignet sie sich tatsächlich gut für die anstehende Aufgabe? Funktioniert es auch für Sie gut? Oder hat man es übernommen, nur weil es im Internet gerade die neueste Erfolgsgeschichte angetrieben hat?

Es gibt einen aktuellen Trend, die Richtlinien und Prozesse von erfolgreichen Unternehmen wie Spotify, Netflix, Stripe, GitLab und anderen zu übernehmen. Alle haben ihre eigenen, einzigartigen Auffassungen von Software-Entwicklung und -Management. Aber bedenken Sie den Kontext: Sind Sie auf dem gleichen Markt tätig, mit den gleichen Zwängen und Möglichkeiten, ähnlicher Expertise und Organisationsgröße, ähnlichem Management und ähnlicher Kultur? Haben Sie eine ähnliche Nutzerbasis und Anforderungen?

Fallen Sie nicht darauf herein. Besondere Artefakte, oberflächliche Strukturen, Richtlinien, Prozesse und Methoden reichen nicht aus.

---

[5] Siehe *https://de.wikipedia.org/wiki/Cargo-Kult*

Tipp 87

Machen Sie, was funktioniert, und nicht, was gerade Mode ist.

Woher wissen Sie, „was funktioniert“? Sie verlassen sich auf diese grundlegendste aller pragmatischen Techniken:

Probieren Sie es aus.

Versuchen Sie, die Idee mit einem kleinen Team oder einer Reihe von Teams zu erproben. Behalten Sie die Teile, die gut zu funktionieren scheinen, und entsorgen Sie alles andere als Abfall oder Overhead. Niemand wird Ihre Organisation herabstufen, weil sie anders arbeitet als Spotify oder Netflix, denn auch diese befolgten auf dem Weg des Wachstums ihre eigenen aktuellen Prozesse nicht. Und Jahre später, wenn diese Unternehmen reifen, umschwenken und weiter florieren, werden sie wieder etwas anderes tun.

Das ist das eigentliche Geheimnis ihres Erfolgs.

Die eine Größe passt niemandem

Der Zweck einer Software-Entwicklungsmethode ist es, Menschen bei der Zusammenarbeit zu unterstützen. Wie wir in *Topic 48*, *Das Wesen der Agilität*, ausführen, gibt es keinen einzigen Plan, dem man bei der Entwicklung von Software folgen kann, insbesondere keinen Plan, den sich jemand anderes in einer anderen Firma ausgedacht hat.

Viele Zertifizierungsprogramme sind sogar noch schlimmer als das: Sie setzen voraus, dass der Student in der Lage ist, die Regeln auswendig zu lernen und zu befolgen. Aber das liegt nicht in Ihrem Interesse. Sie brauchen die Fähigkeit, über bestehende Regeln hinauszusehen und Möglichkeiten für Vorteile zu nutzen. Das ist eine ganz andere Denkweise als ein „aber bei Scrum/Lean/Kanban/XP/agile macht man es so und so ...“ und so weiter.

Stattdessen wollen Sie die besten Elemente aus einer bestimmten Methodik nehmen und sie an Ihre Gegebenheiten anpassen. Wir haben keine Einheitsgröße, die allen passt, und die derzeitigen Methoden sind bei Weitem nicht vollständig, sodass Sie sich mehr als nur eine beliebte Methode ansehen müssen.

Zum Beispiel definiert Scrum einige Praktiken zum Projektmanagement, aber Scrum allein bietet nicht genügend Anleitung auf der technischen Ebene für Teams oder auf der Portfolio/Governance-Ebene für die Führung. Wo fangen Sie also an?

Nehmen Sie die als Vorbild!

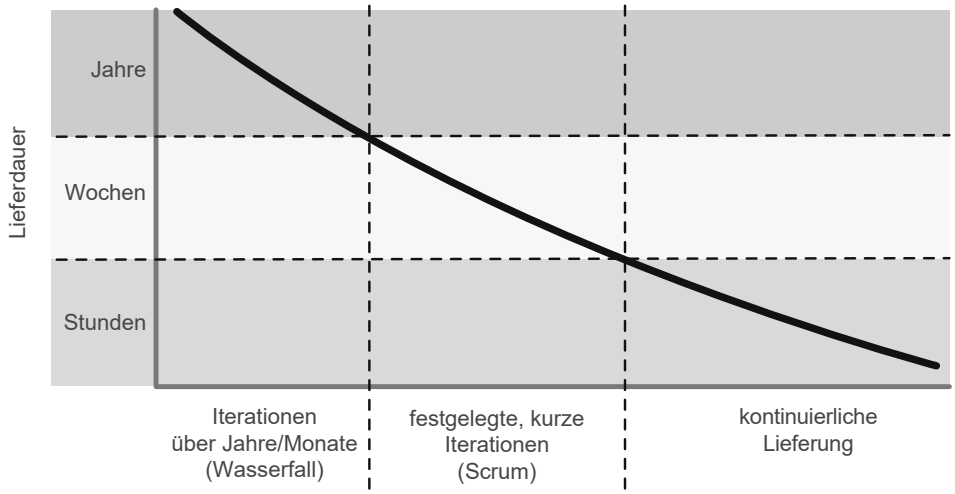
Wir hören häufig, wie Software-Entwicklungsleiter ihren Mitarbeitern sagen: „Wir sollten wie Netflix arbeiten“ (oder eines dieser anderen führenden Unternehmen). Natürlich *könnten* Sie das tun.

Zuerst besorgen Sie sich ein paar hunderttausend Server und zig Millionen Benutzer ...

Das eigentliche Ziel

Das Ziel besteht natürlich nicht darin, „Scrum umzusetzen“, „agil zu sein“, „nach Lean zu arbeiten“ oder was es da sonst noch gibt. Ziel ist es, in der Lage zu sein, funktionierende

Software zu liefern, die den Anwendern *jederzeit* neue Möglichkeiten bietet. Nicht erst in Wochen, Monaten oder Jahren, sondern *jetzt*. Für viele Teams und Organisationen fühlt sich eine kontinuierliche Lieferung wie ein hochgestecktes, unerreichbares Ziel an, vor allem, wenn Sie mit einem Prozess belastet sind, der die Lieferung auf Monate oder sogar Wochen beschränkt. Aber wie bei jedem Ziel kommt es darauf an, weiterhin in die richtige Richtung zu zielen.



Wenn Sie in jährlichen Releases liefern, versuchen Sie, den Zyklus auf Monate zu verkürzen. Von Monaten verkürzen Sie auf Wochen. Ausgehend von einem vierwöchigen Sprint versuchen Sie einen zweiwöchigen. Kochen Sie von einem zweiwöchigen Sprint auf einen einwöchigen herunter. Dann täglich. Dann schließlich auf Anfrage. Beachten Sie, dass die Fähigkeit, auf Anfrage zu liefern, nicht bedeutet, dass Sie gezwungen sind, in jeder Minute eines jeden Tages zu liefern. Sie liefern, wenn die Anwender es brauchen, wenn es geschäftlich sinnvoll ist, dies zu tun.

Tipp 88

Liefern Sie, wenn Anwender es brauchen.

Um zu dieser Art der kontinuierlichen Entwicklung überzugehen, benötigen Sie eine grundsolide Infrastruktur, die wir im nächsten *Topic 51*, *Pragmatic Starter Kit*, erörtern. Sie entwickeln im Hauptstamm Ihres Versionsverwaltungssystems, nicht in den Zweigen, und verwenden Techniken wie *Feature Switches*, um Testfunktionen selektiv an Anwender zu liefern.

Sobald Ihre Infrastruktur in Ordnung ist, müssen Sie entscheiden, wie Sie die Arbeit organisieren wollen. Anfänger sollten beim Projektmanagement vielleicht mit Scrum beginnen, plus die technischen Praktiken von eXtreme Programming (XP). Diszipliniertere und erfahrenere Teams könnten auf Kanban- und Lean-Techniken zurückgreifen, sowohl für das Team als auch vielleicht für größere Governance-Themen.

Aber folgen Sie uns hier nicht sklavisch, sondern untersuchen Sie alles und probieren Sie diese Ansätze selbst aus. Seien Sie aber vorsichtig, es nicht zu übertreiben. Eine übermäßige Investition in eine bestimmte Methodik kann dazu führen, dass man für Alternativen blind wird. Man gewöhnt sich daran. Bald wird es schwer, andere Wege zu sehen. Sie haben sich festgefahren, und jetzt können Sie sich nicht mehr schnell anpassen.

Man könnte genauso gut Kokosnüsse verwenden.

Verwandte Topics

- Topic 12, *Leuchtspurmunition*
- Topic 27, *Nicht schneller als die Scheinwerfer*
- Topic 48, *Die Essenz der Agilität*
- Topic 49, *Pragmatische Teams*
- Topic 51, *Pragmatic Starter Kit*

Topic 51: Pragmatic Starter Kit

Die Zivilisation schreitet dadurch fort, dass wir die Anzahl wichtiger Tätigkeiten steigern, die wir ohne Nachdenken verrichten können.

Alfred North Whitehead

Damals, als Autos noch eine Neuheit waren, war die Anleitung für das Starten eines Ford T mehr als zwei Seiten lang. Bei modernen Autos dreht man nur den Schlüssel herum, die Anlassprozedur läuft automatisch und idiotensicher. Ein Mensch, der einer Liste von Anweisungen folgt, könnte den Motor absaufen lassen, der automatische Starter nicht.

Obwohl die Software-Industrie immer noch auf dem Stand des Ford T ist, können wir es uns nicht leisten, bei immer wiederkehrenden Aufgaben jedes Mal zwei Seiten Anleitung durchzugehen. Ob es der Build-Prozess ist, die Freigabe von Programmversionen, Testen, der Papierkram für Projekte oder irgendeine andere wiederkehrende Aufgabe im Projekt, es muss automatisch geschehen.

Zusätzlich wollen wir die Konsistenz und Wiederholbarkeit im Projekt gewährleisten. Manuelle Vorgänge überlassen die Konsistenz dem Zufall und Wiederholbarkeit ist nicht garantiert, insbesondere wenn Teile des Vorgehens offen für Interpretationen sind.

Nachdem wir die erste Ausgabe von „Der Pragmatische Programmierer“ geschrieben hatten, wollten wir weitere Bücher erstellen, die Teams bei der Software-Entwicklung helfen. Wir dachten uns, wir sollten am Anfang beginnen: Was sind die grundlegendsten und wichtigsten Elemente, die *jedes* Team unabhängig von Methodik, Sprache oder Technologie benötigt? Und so entstand die Idee des Pragmatic Starter Kit, das diese drei kritischen und miteinander verbundenen Themen abdeckt:

- Versionsverwaltung
- Regressionstests
- Vollständige Automatisierung

Dies sind die drei Säulen, die jedes Projekt unterstützen. Und das geht so.

Setzen Sie stets eine Versionsverwaltung ein

Wie bereits in *Topic 19*, *Versionsverwaltung*, erwähnt, sollten wir alles unter Versionsverwaltung stellen, was für das Projekt benötigt wird. Diese Idee wird im Zusammenhang mit dem Projekt selbst noch wichtiger.

Erstens erlaubt es, dass Build-Maschinen vergänglich sind. Statt dieser geheiligten, ratternden Maschine in der Ecke des Büros, die keiner zu berühren wagt,[6] werden Build-Rechner und/oder Cluster bei Bedarf als Spot Instances in der Cloud erstellt. Auch die Deployment-Konfiguration unterliegt der Versionsverwaltung, sodass die Freigabe zur Produktion automatisch erfolgen kann.

Und das ist der wichtige Teil: Auf der Projektebene *steuert* die Versionsverwaltung den Build- und Release-Prozess.

Tipp 89

Nutzen Sie Versionsverwaltung zur Steuerung von Builds, Tests und Releases.

Das heißt, Builds, Tests und Deployment werden durch Commits oder Pushs an die Versionsverwaltung ausgelöst und in einem Container in der Cloud ausgeführt. Die Freigabe zur Bereitstellung oder Produktion wird durch die Verwendung eines Tags in Ihrem Versionskontrollsystem festgelegt. Releases werden dann zu einem viel weniger zeremoniellen Teil des täglichen Lebens – eine echte kontinuierliche Lieferung, die nicht an einen bestimmten Build- oder Entwicklungsrechner gebunden ist.

Schonungsloses Testen

Viele Entwickler testen sanft und wissen unbewusst, wo der Code versagen wird. Sie vermeiden diese Schwachstellen. Pragmatische Programmierer sind anders. Wir werden davon *angetrieben*, unsere Fehler *jetzt* zu finden. So müssen wir die Schande nicht ertragen, dass unsere Fehler später von anderen gefunden werden.

Fehler zu suchen, ist wie Fischen mit Netzen. Wir benutzen kleine, engmaschige Netze (Unit-Tests), um die kleinen Fische zu fangen, und große, grobe Netze (Integrationstests) für die Killer-Haie. Manchmal schaffen es die Fische zu entkommen. Dann flicken wir die sichtbaren Löcher in der Hoffnung, immer mehr von den glitschigen Defekten zu erwischen, die in unserem Projekt-Pool herumschwimmen.

Tipp 90

Testen Sie frühzeitig, häufig und automatisch.

[6] Wir haben das schon öfter live selbst gesehen, als Sie vielleicht glauben.

Wir beginnen mit dem Testen, sobald wir Quelltext haben. Diese mickrigen, kleinen Fische haben die üble Angewohnheit, ziemlich schnell zu gigantischen, menschenfressenden Haien zu werden, und Haie fangen ist ein ganzes Stück schwieriger. Also schreiben wir Unit-Tests. Ganz viele Unit-Tests.

Genau genommen wird in einem guten Projekt sogar *mehr* Quelltext für Tests geschrieben als für die eigentliche Anwendung. Der Aufwand für das Schreiben der Tests lohnt sich, denn auf lange Sicht kommt es Sie günstiger, und Sie haben damit eine reelle Chance, ein nahezu fehlerfreies Produkt abzuliefern.

Zusätzlich geben Ihnen die erfolgreichen Tests die Zuversicht, dass ein Stück Quelltext wirklich fertig ist.

Tipp 91

Das Programmieren ist erst fertig, wenn alle Tests erfolgreich waren.

Der automatische Build-Lauf führt alle verfügbaren Tests aus. Es ist wichtig, einen „Echttest" anzustreben, oder anders gesagt sollte die Testumgebung genau der Produktionsumgebung entsprechen. Lücken sind die Brutstätten von Bugs.

Der Build kann mehrere Haupttypen von Software-Tests abdecken: Unit-Tests, Integrationstests, Validierung und Verifizierung sowie Leistungstests.

Diese Liste ist keineswegs vollständig und einige spezialisierte Projekte werden auch verschiedene andere Arten von Tests erfordern. Aber damit haben wir eine gute Ausgangsbasis.

Unit-Tests

Ein *Unit-Test* ist Quelltext, der ein Modul auf die Probe stellt. Darum ging es in *Topic 41, Testen fürs Entwickeln*. Es ist die Basis aller anderen Testarten, die wir in diesem Topic behandeln. Wenn die Teile schon einzeln nicht funktionieren, werden sie sicher auch zusammen nicht besser arbeiten. Alle verwendeten Module müssen ihre eigenen Unit-Tests bestehen, bevor Sie weitermachen können.

Sobald alle relevanten Module ihre eigenen Unit-Tests bestanden haben, sind Sie bereit für den nächsten Schritt. Sie müssen testen, wie die Module im System interagieren.

Integrationstests

Integrationstests prüfen, ob die wesentlichen Teilsysteme des Projekts funktionieren und gut miteinander zusammenspielen. Mit vernünftigen Verträgen und sorgfältigen Tests können Integrationsprobleme leicht entdeckt werden. Anderenfalls wird die Integration ein fruchtbarer Nährboden für Fehler. Genau genommen sind Integrationsprobleme die allergrößte Fehlerquelle in einem System.

Integrationstests sind in der Tat nur eine Erweiterung der Unit-Tests. Man testet jetzt eben, ob ganze Subsysteme ihren Vertrag erfüllen.

Validierung und Verifikation

Sobald Sie eine ausführbare Benutzeroberfläche oder einen Prototyp haben, müssen Sie sich vor allem eine Frage stellen: Die Anwender haben Ihnen gesagt, was sie wollen, aber ist es auch das, was sie brauchen?

Erfüllt es die funktionalen Anforderungen des Systems? Auch das muss getestet werden. Ein fehlerfreies System, das die falschen Fragen beantwortet, ist nicht besonders sinnvoll. Finden Sie heraus, wie Anwender das System benutzen, und prüfen Sie, wie die Daten der Anwender von den Testdaten der Programmierer abweichen (siehe die Geschichte in Abschnitt *Wo Sie beginnen* (Kapitel 3) über das Linienziehen).

Performancetests

Performance- oder Lasttests sind sicherlich ein ebenfalls wichtiger Aspekt für das Projekt.

Fragen Sie sich selbst, ob die Software die Performance-Anforderungen unter realen Bedingungen erfüllt: mit der erwarteten Anzahl von Benutzern, Verbindungen oder Transaktionen pro Sekunde. Ist sie skalierbar?

Bei einigen Anwendungen benötigen Sie vielleicht sogar spezielle Test-Hardware und -Software, um die Last realistisch simulieren zu können.

Tests testen

Wenn niemand perfekte Software schreiben kann, sind auch perfekte Tests unmöglich. Wir müssen die Tests testen.

Betrachten Sie unsere Testreihen als aufwendiges Sicherheitssystem, das einen Alarm auslösen soll, wenn ein Fehler auftritt. Wie kann man ein Sicherheitssystem besser testen als mit einem Einbruchsversuch?

Haben Sie einen Test geschrieben, der einen bestimmten Fehler erkennen soll, müssen Sie den Fehler bewusst verursachen und sichergehen, dass der Testalarm anschlägt. Das gewährleistet, dass der Test den Fehler auch erkennt, wenn er wirklich auftritt.

Tipp 92

Testen Sie Ihre Tests durch Sabotage.

Wenn Sie es mit dem Testen *wirklich* ernst meinen, nehmen Sie einen gesonderten Zweig des Quelltexts und bauen dort absichtlich Fehler ein, die von den Tests entdeckt werden müssen. Auf einer höheren Ebene können Sie etwas wie *Chaos Monkey*[7] von Netflix nutzen, um Dienste zu unterbrechen (d. h. zu „killen“) und die Widerstandsfähigkeit Ihrer Anwendung zu testen.

Gehen Sie beim Schreiben von Tests sicher, dass der Alarm losgeht, wenn er soll.

Sorgfältig testen

Wenn Sie überzeugt sind, dass Ihre Tests korrekt sind und auftretende Fehler von Ihnen gefunden werden, woher wissen Sie dann, ob Sie Ihren Quelltext sorgfältig genug getestet haben?

Die Antwort lautet: „Sie wissen es nicht“ und werden es auch nie wissen. Versuchen Sie es auch einmal mit Tools zur *Abdeckungsanalyse*, die Ihren Code während des Testens beobachten und verfolgen, welche Codezeilen ausgeführt wurden und welche nicht. Diese Tools helfen Ihnen, ein allgemeines Gefühl dafür zu bekommen, wie umfassend Ihre Tests sind, aber erwarten Sie keine 100%ige Abdeckung.[8]

---

[7] *https://netflix.github.io/chaosmonkey*

[8] Eine interessante Studie über die Korrelation zwischen Testabdeckung und Defekten finden Sie in *Mythical Unit Test Coverage* [ADSS18].

Selbst wenn Sie jede Quelltextzeile erwischen, zeigt das nicht das ganze Bild. Wichtig ist die Anzahl der Zustände, die Ihr Programm annehmen kann. Zustände sind nicht gleichbedeutend mit Quelltextzeilen. Nehmen Sie beispielsweise an, Sie hätten eine Funktion, die zwei Integer erwartet, von denen jede eine Zahl zwischen 0 und 999 sein kann.

```
int test(int a, int b) {
  return a / (a + b);
}
```

In der Theorie hat dieser Dreizeiler 1.000.000 logische Zustände, von denen 999.999 korrekt funktionieren und einer nicht (wenn a + b = 0 ist). Allein die Tatsache, dass diese Quelltextzeile ausgeführt worden ist, zeigt das nicht. Sie müssen alle möglichen Zustände des Programms identifizieren. Dummerweise ist das im Allgemeinen ein *echt hartes* Problem. Hart im Sinne von: „Die Sonne wird ein kalter, harter Brocken sein, bevor Sie es gelöst haben."

Tipp 93

Testen Sie Zustandsabdeckung, nicht Quelltextabdeckung.

Property-Based Testing

Eine gute Möglichkeit zu untersuchen, wie Ihr Code mit unerwarteten Zuständen umgeht, besteht darin, diese Zustände von einem Computer generieren zu lassen.

Verwenden Sie Property-Based Testing, um Testdaten gemäß den Verträgen und Invarianten des zu prüfenden Codes zu erzeugen. Wir behandeln dieses Thema ausführlich in *Topic 42*, *Property-Based Testing*.

Das Netz enger ziehen

Schließlich wollen wir noch das allerwichtigste Konzept beim Testen enthüllen. Es ist naheliegend, und praktisch jedes Lehrbuch weist darauf hin, aber aus irgendeinem Grund wenden es die meisten Projekte nicht an.

Wenn ein Fehler durch das bestehende Netz von Tests schlüpft, müssen Sie einen neuen Test hinzufügen, um ihn beim nächsten Mal zu entdecken.

Tipp 94

Finden Sie Fehler nur einmal.

Wenn ein menschlicher Tester einen Fehler findet, sollte es das *letzte* Mal gewesen sein, dass ein Mensch darauf gestoßen ist. Die automatisierten Tests müssen so geändert werden, dass ab dann auf diesen speziellen Fehler hin geprüft wird. Die Prüfung erfolgt jedes Mal, ohne Ausnahme, egal wie trivial der Fehler auch sein mag und wie bestimmt der Programmierer auch behauptet: „Oh, das wird nie wieder passieren."

Es wird wieder passieren, und wir haben nicht die Zeit zur Jagd auf Fehler, die automatisierte Tests für uns finden könnten. Wir müssen unsere Zeit nutzen, neuen Quelltext zu schreiben – und neue Fehler.

Vollständige Automatisierung

Wie wir zu Beginn dieses Topics gesagt haben, beruht die moderne Entwicklung auf skriptgesteuerten, automatischen Verfahren. Egal, ob Sie etwas so Einfaches wie Shell-Skripte mit `rsync` und `ssh` oder voll funktionsfähige Lösungen wie Ansible, Puppet, Chef oder Salt verwenden, verlassen Sie sich einfach nicht auf manuelles Eingreifen.

Es war einmal ein Kundenstandort, an dem alle Entwickler dieselbe IDE benutzten. Der Systemadministrator gab jedem Anweisungen, welche zusätzlichen Pakete zur IDE installiert werden sollten. Die Anweisungen waren viele Seiten lang, voll von „Klicken Sie hier", „Scrollen Sie dahin", „Ziehen Sie dies", „Doppelklicken Sie das" und „Wiederholen Sie das".

Wenig überraschend: Die Rechner aller Software-Entwickler waren immer ein klein wenig anders konfiguriert. Wenn verschiedene Software-Entwickler dieselbe Anwendung laufen ließen, gab es feine Unterschiede. Auf manchen Rechnern traten Fehler auf, auf anderen aber nicht. Das Aufspüren der Versionsdifferenzen enthüllte bei jeder Komponente Überraschungen.

Tipp 95

Vermeiden Sie manuelle Vorgänge

Menschen können einfach nicht so gut wiederholen wie Computer, und wir sollten das auch nicht von ihnen erwarten. Ein Shell-Skript oder ein Programm führt dieselben Anweisungen aus, in derselben Reihenfolge und immer wieder. Es steht selbst unter Versionsverwaltung, sodass Sie Veränderungen an der Build-/Release-Prozedur im Lauf der Zeit feststellen können („aber es *hat* mal funktioniert ...").

Alles hängt von der Automatisierung ab. Sie können den Build für das Projekt nicht auf einem anonymen Cloud-Server erstellen, es sei denn, die Erstellung erfolgt vollautomatisch. Sie können kein automatisches Deployment durchführen, wenn manuelle Schritte erforderlich sind. Und sobald man manuelle Schritte einführt („nur für diesen einen Teil ..."), zerbricht man gerade ein sehr großes Fenster.[9]

Mit diesen drei Säulen der Versionsverwaltung, schonungslosem Testen und vollständiger Automatisierung hat Ihr Projekt die solide Grundlage, die Sie brauchen, um sich auf den schwierigen Teil konzentrieren zu können: die Anwender zu begeistern.

Verwandte Topics

- Topic 11, *Umkehrbarkeit*
- Topic 12, *Leuchtspurmunition*
- Topic 17, *Kommandospiele*
- Topic 19, *Versionsverwaltung*
- Topic 41, *Testen fürs Entwickeln*
- Topic 49, *Pragmatische Teams*
- Topic 50, *Kokosnüsse bringen's nicht*

[9] Denken Sie immer an *Topic 3*, *Software-Entropie*. Immer!

Aufgaben

- Sind Ihre nächtlichen oder kontinuierlichen Builds automatisch, aber das Deployment zur Produktion nicht? Warum? Was ist so besonders an diesem Server?
- Können Sie Ihr Projekt vollständig automatisch testen? Viele Teams sind gezwungen, das zu verneinen. Und warum? Ist es zu schwierig, die richtigen Ergebnisse festzulegen? Wäre es dann nicht auch schwierig, den Auftraggebern klar zu machen, dass das Projekt „fertig“ ist?
- Ist es zu schwierig, die Anwendungslogik unabhängig von der grafischen Benutzeroberfläche zu testen? Was sagt das über die grafische Benutzeroberfläche aus? Was über Kopplung?

52 Topic 52: Erfreuen Sie die Anwender

Wenn Sie Menschen verzaubern, ist es nicht Ihr Ziel, an ihnen Geld zu verdienen oder sie dazu zu bringen, das zu tun, was Sie wollen, sondern Ihr Ziel ist es, sie mit großer Freude zu erfüllen.

Guy Kawasaki

Unser Ziel als Entwickler ist es, die Anwender zu *erfreuen*. Deshalb sind wir hier. Nicht, um sie nach ihren Daten zu durchsuchen, ihre Aktivitäten abzusaugen oder ihre Brieftaschen zu leeren. Mal abgesehen von solch ruchlosen Zielen ist selbst die rechtzeitige Bereitstellung funktionierender Software nicht genug. Das allein wird sie nicht erfreuen.

Ihre Anwender fahren nicht eigentlich besonders auf Code ab. Stattdessen haben sie ein geschäftliches Problem, das im Rahmen ihrer Ziele und ihres Budgets gelöst werden muss. Sie sind davon überzeugt, dass sie das in Zusammenarbeit mit Ihrem Team erreichen können.

Ihre Erwartungen beziehen sich nicht auf Software. Sie sind nicht einmal implizit in irgendeiner Spezifikation enthalten, die Sie von ihnen bekommen (weil diese Spezifikationen unvollständig sind, bis Ihr Team sie mit den Anwendern mehrmals durchgesprochen hat).

Wie finden Sie dann ihre Erwartungen heraus? Stellen Sie diese einfache Frage:

Wie werden Sie einen Monat (oder ein Jahr oder was auch immer) nach Abschluss dieses Projekts wissen, dass wir alle erfolgreich waren?

Die Antwort wird Sie vielleicht überraschen. Ein Projekt zur Verbesserung von Produktempfehlungen könnte tatsächlich vom Aspekt der Kundenbindung her beurteilt werden; ein Projekt zur Konsolidierung zweier Datenbanken könnte unter dem Gesichtspunkt der Datenqualität beurteilt werden, oder es könnte um Kosteneinsparungen gehen. Aber es sind diese Erwartungen an den Geschäftswert, die wirklich zählen – nicht nur das Software-Projekt selbst. Die Software ist nur ein Mittel zu diesem Zweck.

Und jetzt, da Sie einige der dem Projekt zugrunde liegenden Werterwartungen ausgegraben haben, können Sie darüber nachdenken, wie Sie diese Erwartungen erfüllen können:

- Achten Sie darauf, dass jeder im Team sich über diese Erwartungen völlig im Klaren ist.
- Überlegen Sie bei Ihren Entscheidungen, welcher Weg Sie diesen Erwartungen näherbringt.

- Analysieren Sie kritisch die Anforderungen der Anwender im Hinblick auf die Erwartungen. Bei vielen Projekten haben wir festgestellt, dass die angegebene „Anforderung“ in Wirklichkeit nur eine Vermutung war, was man mithilfe der Technologie erreichen könnte: Es handelte sich eigentlich um einen als Anforderungsdokument verkleideten Amateur-Implementierungsplan. Scheuen Sie sich nicht, Vorschläge zu machen, die die Anforderungen ändern, wenn Sie nachweisen können, dass sie das Projekt dem Ziel näherbringen.
- Denken Sie weiterhin über diese Erwartungen nach, während Sie das Projekt vorantreiben.

Wir haben festgestellt, dass wir mit zunehmender Kenntnis des Bereichs besser in der Lage sind, Vorschläge für andere Dinge zu machen, die zur Lösung der zugrunde liegenden Geschäftsprobleme beitragen könnten. Wir sind der festen Überzeugung, dass Entwickler, die vielen verschiedenen Aspekten einer Organisation ausgesetzt sind, oft Wege sehen können, verschiedene Teile des Unternehmens miteinander zu verweben, die für einzelne Abteilungen nicht immer offensichtlich sind.

Tipp 96

Liefern Sie nicht bloß Code ab, sondern erfreuen Sie die Anwender.

Wenn Sie Ihren Kunden erfreuen wollen, bauen Sie eine Beziehung zu ihm auf, in der Sie aktiv zur Lösung seiner Probleme beitragen können. Auch wenn Sie irgendeine Variante von „Software-Entwickler“ oder „Software-Ingenieur“ in Ihrer Berufsbezeichnung herumtragen, sollte sie in Wahrheit „Problemlöser“ lauten. Denn genau das ist es, was wir tun, und genau das ist das Wesen eines Pragmatischen Programmierers.

Wir lösen Probleme.

Verwandte Topics

- Topic 12, *Leuchtspurmunition*
- Topic 13, *Prototypen und Post-it-Zettel*
- Topic 45, *Die Anforderungsgrube*

Topic 53: Stolz und Vorurteil

53

Sie haben uns lange genug erfreut.

Jane Austen, „Stolz und Vorurteil“

Pragmatische Programmierer entziehen sich ihrer Verantwortung nicht. Stattdessen genießen wir es, Herausforderungen anzunehmen und für unseren Sachverstand bekannt zu sein. Wenn wir für einen Entwurf oder ein Stück Quelltext verantwortlich sind, erfüllen wir eine Aufgabe, auf die wir stolz sein können.

Tipp 97

Signieren Sie Ihre Arbeit.

Früher waren Handwerker stolz, ihre Arbeiten zu signieren. Auch Sie sollten das sein.

Projektteams bestehen immer noch aus Menschen, und das kann zu Problemen führen. In manchen Projekten verursacht die Vorstellung vom *Eigentum am Quelltext* Probleme bei der Zusammenarbeit. Die Teammitglieder verteidigen ihr Territorium und arbeiten nicht mehr an gemeinsamen Fundamenten. Das Projekt endet eventuell in einem Haufen kleiner Machtinseln. Man betrachtet den eigenen Quelltext voreingenommen und stellt sich gegen den Quelltext der Kollegen.

Das ist nicht, was wir wollen. Sie sollten Ihren Quelltext nicht eifersüchtig gegen Eindringlinge verteidigen. Aus demselben Grund sollten Sie andere Programmierer mit Respekt behandeln. Die goldene Regel „Was du nicht willst, das man dir tu', das füg' auch keinem andern zu" und ein grundlegender gegenseitiger Respekt der Entwickler voreinander sind entscheidend, damit dieser Tipp funktioniert.

Anonymität kann insbesondere in großen Projekten ein Nährboden für Nachlässigkeit, Fehler, Faulheit und schlechten Quelltext sein. Zu leicht sieht man sich nur als kleines Rädchen im Getriebe und schreibt endlose Statusberichte mit billigen Ausreden statt guten Quelltext.

Quelltext braucht einen Eigentümer, aber es muss nicht unbedingt eine Einzelperson sein. Übrigens empfiehlt auch Kent Beck mit seinem eXtreme Programming[10] gemeinsames Eigentum am Quelltext (es erfordert aber auch noch weitere Praktiken wie Pair-Programming, um sich vor den Gefahren der Anonymität zu schützen).

Wir wollen stolze Eigentümer sehen („Ich habe das geschrieben, und ich stehe hinter meiner Arbeit"). Ihr Kürzel sollte sich zu einem anerkannten Zeichen für Qualität entwickeln. Man soll ein Stück Software für robust, gut geschrieben, gut getestet und gut dokumentiert halten, wenn man Ihren Namen darauf liest. Eine wirklich professionelle Arbeit, die ein echter Fachmann geschrieben hat.

Ein Pragmatischer Programmierer.

Wir danken Ihnen.

[10] *http://www.extremeprogramming.org*