Arbeiten mit Legacy Teams

Jens Schauder
T-Systems on site services GmbH
Wolfsburg

Schlüsselworte
Clean Code, Motivation, Team Building

Abstract
Die meiste Arbeit in der Softwareentwicklung findet nicht auf der grünen Wiese in frischem neuen Code statt, sondern in Legacy Code. In Anwendungen, die über Jahre, wenn nicht Jahrzehnte gewachsen sind, oder die zumindest so aussehen: Lange Methoden, mit irreführenden Namen, Kommentare die kaum noch zu entziffern sind.

Aber zu Legacy Code gehört nicht selten ein Legacy Team. Erfahrene Veteranen, die oft unter Einsatz Ihrer geistigen Gesundheit, diese Systeme am Leben erhalten. Die den Missstand zwar anprangern ihn aber gleichzeitig oft mit aufrecht erhalten, oder sogar seine Ursache sind.

In diesem Vortrag wird über Erfahrungen mit solchen Teams und Ihren Codebasen berichtet, über Erfolge und Niederlagen im Kampf gegen das Big Ball of Mud Pattern und seine Jünger. Über Ansätze die funktionieren und über Versuche, die man sich auch gleich sparen kann.

Vorwort

Die folgenden Erlebnisse stammen nicht wirklich aus **einem** Projekt, sondern sind ein Amalgam aus vielen verschiedenen Projekten, aber die Beispiele sind real, auch die, bei denen man es nicht glaubt.

Neuer Start in ein existierendes Projekt

Alles begann mit einem für mich neuen Projekt. Das Projekt selbst war allerdings schon viele Jahre alt und hatte diverse Teams gesehen, um nicht zu sagen verschlissen. Dementsprechend sah die Code Basis aus. Eine Mischung aus verschiedenen Architekturen, keine davon erkennbar oder angemessen dokumentiert. Der Code über weite Strecken ungetestet und von, sagen wir mal, zweifelhafter Qualität.

Aus der Vergangenheit hörte ich die dazu passenden Geschichten. Horror Releases mit Wochenend Schichten um irgendwie die Bugzahl auf ein erträgliches Maß zu reduzieren.

Ich war schon ein wenig geschockt, aber das war genau die Aufgabe für die man mich in das Projekt gebracht hat: Das Team dabei zu unterstützen besser zu werden und besseren Code zu schreiben, und zwar mit Tests.

Der erste Angriff

Gegen den dreckigen Code habe ich die üblichen Waffen zu Einsatz gebracht: Jenkins um nach jedem Commit den Code zu kompilieren, zu testen und Analysen darauf auszuführen, PMD und Checkstyle um nach verbesserungswürdigem Code zu fahnden, und um zu verhindern, dass noch mehr davon entsteht, Sonar um die Daten zusammenzuführen und zu visualisieren. In der Tat war all dies schon vorhanden, nur wurde es geflissentlich ignoriert.

Nach ein paar deutlichen Worten meinerseits hatten aber die meisten verstanden, dass sich dies nun ändern würde, und ich war guter Dinge, dass ich einen ersten Teilerfolg verbuchen konnte.

Rückschlag

Bis ich mir den schönen neuen Checkstyle-Warnung-freien Code mal etwas genauer anschaute. Zum Beispiel verbietet Checkstyle die direkte Verwendung von Zahlenliteralen größer zwei, außer in der Zuweisung zu Variablen und Feldern. Diese Zahlen sind als Magic Numbers bekannt.

Nun begab es sich, dass ein Entwickler versuchte Werte aus einem Resultset auszulesen:

```
firstName = rs.getString(1)

lastName = rs.getString(2)

age = rs.getInt(3)
```

In der letzten Zeile beschwert sich Checkstyle über die Magic Number. Der Entwickler, der sich nicht so recht zu helfen wusste, änderte seinen Code wie folgt:

```
firstName = rs.getString(1)

lastName = rs.getString(2)

age = rs.getInt(2+1)
```

Und da endete die Geschichte nicht, es ging weiter:

```
firstName = rs.getString(1)

lastName = rs.getString(2)

age = rs.getInt(2+1)

city = rs.getString(2+2)

zip = rs.getString(2+2+1)
```

Nicht wirklich das, was ich mir unter sauberem Code vorstelle.

Der zweite Angriff

Natürlich gibt es in einem Legacy Projekt mehr als nur schlimmen Code. In vielen Projekten sehe ich Testteams, die wenn sie denn wenigstens existieren, vollständig getrennt sind von der Entwicklung. Wenn dann die Anforderungen von den Analysten über zwei Zäune zu Testern und Entwicklern geworfen werden, wundern sich am Ende alle, dass die Interpretationen von Testern und Entwicklern unterschiedlich aussehen. Erst am Ende wird dann versucht die beiden Interpretationsvarianten übereinander zu bekommen, wenn man eigentlich dachte schon längst fertig zu sein.

Ein weitere Klassiker sind Datenbankentwickler/Administratoren, die vollständig unabhängig vom Rest des Teams arbeiten. Migrationsskripte werden in einer Verzeichnisstruktur verwaltet, die nur ein Mitarbeiter kennt. Datenbanken manuell synchronisiert (was de facto meistens heißt sie sind nur so in etwa synchron).

Solche und ähnliche Probleme gab es viele und ich habe sie alle aufgelistet, mit dem Team besprochen, Maßnahmen definiert, mit Zuständigen und Terminen. Wie man es gelernt hat.

Und dann kamen viele andere Aufgaben und nichts von all dem wurde kontrolliert. Und da die anderen Kollegen auch andere Aufgaben bekamen wurde nichts von all dem umgesetzt.

Mit anderen Worten: Wir hatten den Mund zu voll genommen.

Warum?

Warum funktionierte dies alles in diesem Projekt, obwohl wir in einem ähnlichen Projekt mit ganz ähnlichem Vorgehen sehr Erfolgreich waren? In dem Projekt hatten wir Code Coverage je nach Messverfahren zwischen 80% und 98%; 0 Checkstyle oder andere Warnungen; Im großen und ganzen sauberen Code. Warum hier nicht?

Und dann bin ich eines Tages über Conways Law gestolpert:

"*organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations*"

Und ich habe Schauders Corollary formuliert:

"If your team is a mess, so will be your code."

Oder wie es Jim McCarthy formulierte:

“You can’t have great software without a great team, and most software teams behave like dysfunctional families.”

Wenn man möchte, dass sich die Artefakte ändern, die ein Team produziert, kann man nicht an den Artefakten herum doktern, sondern man muss sich um das Team kümmern. D.h. man muss das Team motivieren, sich beziehungsweise sein Verhalten zu ändern. Und ja, man kann Motivation von außen beeinflussen. Wer es nicht glaubt sollte sich mal mit einem Marketingspezialisten oder Psychologen unterhalten.

Aber wie?

Es gibt Modelle zum Thema Motivation wie Sand am Meer. Mir persönlich haben die meisten relativ wenig geholfen, da sie zwar einen Sachverhalt beschreiben / modellieren. Aber ich konnte daraus keine Anregungen für Maßnahmen herleiten. Aber ich bin schließlich auf zwei Modelle gestoßen, dir mir weiter geholfen haben.

Das erste ist das "**Fogg Behavior Model**". Es besagt, dass man drei Dinge benötigt um zu veranlassen, dass jemand etwas tut:

- Motivation: Die Person muss motiviert sein etwas zu tun. Ich bin problemlos in der Lage mir mit einem Hammer auf den Daumen zu schlagen, aber ich will es nicht, also tue ich es nicht (oder zumindest nicht absichtlich)

- Fähigkeit: Ich fände es total Klasse einen Marathon zu laufen, aber ich würde allerspätestens nach 10km zusammenbrechen, wahrscheinlich eher nach 5km. Ich kann es einfach nicht. Die Fähigkeit fehlte offensichtlich in dem obigen Code Beispiel bezüglich Magic Numbers. Der Entwickler wollte sauberen Code gemäß Checkstyle schreiben, er wusste bloß nicht wie.

- Auslöser: Wenn jemand motiviert und fähig ist etwas zu tun, dann fehlt nur noch der Auslöser. Etwas dass den Anstoß dazu gibt.

Wenn jemand ein gewünschtes Verhalten nicht an den Tag legt fehlt wenigstens eins dieser Dinge, und wenn ich dafür sorge dass der Mangel behoben wird, kann sich das Verhalten einstellen. Umgekehrt kann ich einen der drei Faktoren entfernen und dadurch das Verhalten unterbinden. Zum Beispiel steckt man die Hände von sehr kleinen Kindern in weiche Fäustlinge, damit sie sich nicht an juckenden Wunden kratzen können -> Sie haben nicht mehr die Fähigkeit sich zu kratzen. Wenn ich mir das Essen von ungesundem Zeugs abgewöhnen möchte, kann ich dafür sorgen, dass ich so etwas nicht im Haus habe und damit den Trigger "Oh, da ist ja noch eine Tüte Chips" entschärfen.

Fähigkeiten können meist einfach verbessert werden, durch allseits bekannte Maßnahmen wie Schulungen, Coaching und gerade bezüglich Clean Code: Pair Programming und Code Reviews. Als Auslöser (und unter Umständen als Motivation) können Metriken helfen. Dafür hilft es wenn die Metriken gegen 0 gemessen werden. Wenn ich 39572 Checkstyle Warnungen habe, nehme ich es nicht war, wenn durch eine Änderung daraus 39578 Checkstye Warnungen werden. Habe ich aber 0 Checkstyle Warnungen sehe ich eine Änderung auf 8 sofort und dies kann als Trigger dienen.

ABER: Ich muss intensiv daran arbeiten, dass die Mitarbeiter den Hintergrund hinter diesen Warnungen verstehen und wissen wie diese in sinnvolle Korrekturen umgesetzt werden können.

UND: Wer Metriken als Motivation eingesetzt, in dem Sinne: Ihr bekommt für die Metrik X Y€ schadet seinem Projekt. Diese Art der Wenn-Dann Belohnung ist Kontraproduktiv und führt zu lokalen Optimierungen, die für die Gesamtlösung hinderlich sind.

Beispiele: Bekommen Entwickler Boni pro gefixten Bug, motiviert dies schlechten Code zu schreiben, damit viele Bugs gefixt werden.

Selbst wenn es jemand schafft eine Metrik zu konstruieren, die keine so offensichtlichen Nachteile hat, bleibt der Effekt, dass die starke Fokussierung auf die Zielerreichung die Kreativität einschränkt und zu schlechterer Leistung führt. Siehe zum Beispiel:
http://www.ted.com/talks/dan_pink_on_motivation.html

Aber wie kann man jemanden motivieren? Hier hilft das zweite Model, auf das ich gestoßen bin:

SCARF

Die Idee hinter SCARF ist, dass jeder Mensch gewisse Bedürfnisse hat, und jeder Mensch versucht Situationen herbeizuführen, in denen diese Bedürfnisse erfüllt werden und versucht Situationen zu vermeiden, die die Erfüllung dieser Bedürfnisse vermeiden. SCARF ist benannt nach diesen Bedürfnissen:

Status: Jeder mag es wenn er gesagt bekommt, oder das Gefühl hat toll zu sein. Wenn ich jemanden Vorschriften mache, oder kritisiere greife ich seinen Status an, denn er wird zum Befehlsempfänger. Wenn ich Aufgaben und Rollen dem Team anbiete und ein Teammitglied eine herausfordernde Aufgabe freiwillig übernimmt und erledigt, erhöht er seinen Status.

Certainty (Sicherheit): Ist bei einer Aufgabe nicht klar, wann diese erledigt ist, erzeugt dies Unsicherheit. Habe ich eine klare Definition was zu tun ist (Folgende Tests müssen funktionieren, Code muss sauber gemäß Checkstyle sein, Architekturdokumentation, Benutzerhandbuch und Administrationshandbuch müssen aktualisiert sein), wird die Aufgabe sehr viel attraktiver. Transparenz ist hier ein wichtiges Mittel.

Autonomy (Autonom sein): Schreibe ich genau vor, was, wann, wie zu tun ist, ist eine solche Aufgabe sehr unattraktiv. Gebe ich nur die essentiellen Randbedingungen vor: Diese Tests müssen funktionieren, es darf X Millisekunden dauern und es muss in 2 Wochen fertig sein und überlasse

denen die es tatsächlich machen, wie genau sie es tun, ist die gleiche Aufgabe plötzlich viel attraktiver und die Kollegen machen sich motivierter ans Werk.

Relatedness (Dazugehörigkeitsgefühl): Menschen wollen Teil einer Gruppe sein. Daher wehren sich Tester dagegen aus ihrem Tester Team herausgerissen zu werden, um zum Teil des Entwicklerteams zu werden. Dies erzeugt Unsicherheit und das Risiko plötzlich ohne Gruppe da zustehen, wenn das mit dem Entwicklerteam nicht funktioniert. Also die Zugehörigkeit zum Testerteam nicht auflösen, sondern das Entwickerteam als zusätzliche Gruppe anbieten. Neue Mitglieder auch wirklich dazu gehören lassen.

Fairness: Kann oft durch Transparenz erzeugt bzw. vermittelt werden. Unbeliebte Aufgaben können rotieren. Zum Beispiel hatten wir in einem Projekt den Tester der Woche: Jede Woche wurde ein anderer bestimmt, der die Änderungen aller anderer Teammitglieder testen musste. Bestimmen sie ein Kind einer Jugendgruppe den Tisch abzuräumen, während alle anderen raus gehen und spielen, bekommen sie viele Diskussionen. Machen dies alle gemeinsam, oder rotiert die Aufgabe, geht es meistens viel besser.

Positive Motivation, dass heißt durch Erfüllung dieser Bedürfnisse funktioniert meist und langfristig wesentlich besser, als die negative Motivation, da es viel mehr Wege gibt etwas nicht zu tun, als etwas zu tun: D.h. das öffentliche Lob für guten, sauberen Code funktioniert besser, als die Standpauke für den Bockmist.

Natürlich sind dies nur Modelle, und wie es so schön heißt: alle Modelle sind falsch, aber einige sind nützlich. Ich erlebe diese Modelle als nützlich.

Der lange Weg

Ein Team zu verändern ist eine langfristige Angelegenheit. Schlechte Angewohnheiten werden nicht von einem Tag auf den nächsten abgelegt. Es gibt aber Strategien, die einem helfen:

Die "**Broken Window**" Theorie besagt, dass der Verfall einer Gegend mit einem eingeschlagenen Fenster beginnt, welches nicht repariert wird. Dies sendet das Signal aus: "Es spielt keine Rolle wie es hier aussieht, jeder kann tun und lassen was er möchte." Und bald sind weitere Fenster eingeschlagen, Wände beschmiert und wer es sich leisten kann zieht weg. Aber wie hilft dies mit einer Code Basis die schon verfallen ist? Schafft Bereiche die sauber sind und lasst nicht zu, dass diese wieder verfallen. Verabschiedet euch dafür von der Idee alles auf einmal perfekt zu machen. Dies lässt sich sowohl auf Code, wie auf Verhalten im Team anwenden.

Benutzt die "**Mikado Methode**". Bei der Mikado Methode wird zunächst ein Ziel definiert. In der originalen Mikado Methode bezieht sich dies auf eine Code Veränderung, aber es kann auch eine Veränderung im Team sein, z.B. soll die Anzahl der Bugs beim Release reduziert werden. Nun kann man unterschiedliche Maßnahmen identifizieren, die beim Erreichen des Ziels helfen können. Davon wird eine probiert. Ist sie erfolgreich, behält man sie bei und nimmt die nächste Maßnahme in Angriff. Ist die Maßnahme nicht erfolgreich, analysiert man, woran dies liegt und macht die Maßnahme zunächst wieder rückgängig. Die Ursachen, definieren neue Maßnahmen, die man angehen kann, um die Situation zu verbessern.

Stellt Dinge unter **Quarantäne**. Dies kann Code sein, der nicht so aussieht wie er sein sollte, den man jetzt aber nicht reparieren kann. Dies können Beziehungen im Projekt sein: "Ja, die Zusammenarbeit mit den DBAs funktioniert nicht, wie wir uns das vorstellen, aber wir leben bis auf weiteres damit, weil wir uns erst um andere Baustellen kümmern." Dabei wird klar gemacht, dass eine Situation nicht

in Ordnung ist, und gleichzeitig wird vermieden, dass man sich mit zu vielen Baustellen ineffizient macht.

"If it hurts, do it more often" ein Release der Software ist schmerzhaft, weil es mit viel manueller Arbeit verbunden ist? Wie wäre es mit einem Release pro Monat, pro Woche, Tag oder nach jedem Commit? Was muss dafür passieren? Wenn diese Maßnahmen umgesetzt werden geht der Schmerz irgendwann weg.

Bei vielen Dingen die Teams tun handelt sich aber um **Cargo Cult**, d.h. um Dinge die so aussehen wie Dinge, die von anderen erfolgreichen Teams so gemacht werden, ohne aber zu verstehen, warum und wie diese Dinge wirken. Ein Beispiel sind Retrospektiven die keine Verbesserungen hervorbringen, oder deren Maßnahmen nicht umgesetzt werden. So etwas ist bestenfalls Zeitverschwendung. Daher also: bei allen Dingen die Ihr tut: Warum tut ihr es? Erreicht ihr dieses Ziel? Wenn nicht, was muss sich ändern?

Letztendlich läuft alles darauf hinaus, dass man **zu hören und klar kommunizieren** muss. Beobachtet, das Team. Warum verhält es sich so wie es sich verhält? Bestehen Konflikte? Insbesondere unterschwellige? Gibt es Stellen an denen Kommunikation nicht funktioniert? Das Militär hat dieses Problem identifiziert und einiges an Ideen entwickelt, wie man dies vermeiden kann: Befehle werden wiederholt. Dieses Vorgehen ist in der üblichen Form nicht wirklich praktikabel. Aber wenn ich einen Bug aufnehme, kann ich ihn dokumentieren und dem Reporter diese Dokumentation vorlegen um sicher zu gehen, dass ich ihn richtig verstanden habe. Wenn mir von einem Problem im Team erfahre, kann ich die Problembeschreibung neu formulieren um sicher zu stellen, dass ich es richtig verstanden habe. Ich kann anderen von diesem Vorgehen erzählen und damit den Anstoß geben, dass auch andere es versuchen (s.o.).

Teams gehen durch verschiedene **Phasen der Team Entwicklung**.

Forming: Das Team entsteht, die Teammitglieder beobachten sich vorsichtig um sich gegenseitig einschätzen zu können

Storming: Es entstehen Konflikte, darüber wer welche Rollen und Aufgaben inne hat und welche Regeln gelten. Diese Konflikte sind wichtig und nicht das Ende der Welt oder auch nur des Teams. Teams scheitern viel eher, weil diese Konflikte verdeckt gehalten werden, als an den Konflikten selbst.

Norming: Konflikte werden gelöst und Regeln definiert. Nicht notwendiger Weise formal und dokumentiert. Dies kann unterstütz werden, durch die Anregung Regen zu identifizieren. Regeln vorgeben ist aber eher kontraproduktiv.

Performing: Das Team wird produktiv, weil jeder seinen Platz gefunden hat, weiß was er zu tun hat und idealerweise sich blind auf seine Kollegen verlassen kann.

Ressourcen

- http://www.behaviormodel.org/
- http://www.davidrock.net/files/NLJ_SCARFUS.pdf
- http://mikadomethod.org/

.

Kontaktadresse:
Jens Schauder
T-Systems on site services GmbH

Alessandro-Volta-Straße 11
D-38440 Wolfsburg

E-Mail jens.schauder@t-systems.com; jens@schauderhaft.de
Internet: https://www.t-systems-onsite.de/; http://blog.schauderhaft.de