Clean Code

Refactoring, Patterns, Testen und Techniken für sauberen Code

Kapitel 1

Sauberer Code

»›Sauber‹ ist nicht als Angriff auf irgendjemanden gedacht. Unsere Vorstellung von ›unsauber‹ soll als Bezeichnung für Programmcode dienen, der vom durchschnittlichen Entwickler mehr Aufwand beim Lesen oder bei der Wartung verlangt.«

– Jeff Langr

Sie lesen dieses Buch aus zwei Gründen. Erstens, Sie sind Programmierer. Zweitens, Sie möchten ein besserer Programmierer werden. Gut. Wir brauchen bessere Programmierer.

Dieses Buch beschäftigt sich mit guter Programmierung. Seine Seiten sind voller Code. Code, den wir uns von allen Seiten ansehen werden. Wir werden diesen

Code auf links drehen, um kein Quäntchen davon zu übersehen. Wenn wir damit fertig sind, werden Sie viel über Code gelernt haben. Noch wichtiger: Sie werden in der Lage sein, den Unterschied zwischen gutem Code und schlechtem Code zu erkennen. Und Sie werden wissen, wie man guten Code schreibt und wie man schlechten Code in guten Code verwandelt.

1.1 Code, Code und nochmals Code

Vielleicht könnte man einwenden, ein Buch über Code wäre doch etwas altmodisch – Code wäre doch längst kein Thema mehr; stattdessen sollte man sich mit KI, Large Language Models (LLMs) und Anforderungen befassen. Tatsächlich vertreten einige Leute die Auffassung, die Ära des Codes ginge zu Ende.[1] Bald werde aller Code nicht mehr geschrieben, sondern generiert. Programmierer würden einfach überflüssig, weil Geschäftsentwickler Programme einfach aus Spezifikationen oder per KI-Prompt generieren würden.

Unsinn! Wir werden niemals ohne Code arbeiten können, weil der Code die Details der Anforderungen repräsentiert. Auf einer gewissen Ebene können diese Details nicht ignoriert oder abstrahiert werden; sie müssen spezifiziert werden. Und die Spezifikation von Anforderungen in einer Detailgenauigkeit, dass sie von einer Maschine ausgeführt werden können, ist *Programmierung*. Und eine solche Spezifikation ist *Code*.

Ich rechne damit, dass die Abstraktionsebene unserer Sprachen noch höher wird. Ich erwarte auch, dass die Anzahl der domänenspezifischen Sprachen und leistungsstarker KI weiterhin wachsen wird. Diese Entwicklung bringt Vorteile mit sich, aber sie wird den Code nicht eliminieren. Tatsächlich werden alle Spezifikationen, die auf diesen höheren Ebenen und in den domänenspezifischen Sprachen geschrieben werden, ebenso wie die Prompts zum Anleiten einer guten KI, nach wie vor aus Code bestehen! Sie müssen immer noch stringent, genau und so formal und detailliert sein, dass sie von einer Maschine verstanden und ausgeführt werden können.

Leute, die denken, Code werde eines Tages verschwinden, ähneln »Mathematikern«[2], die hoffen, eines Tages eine Mathematik zu entdecken, die nicht formal sein muss. Sie hoffen, dass wir eines Tages eine Methode entdecken werden, Maschinen zu erschaffen, die tun, was wir wollen, und nicht, was wir sagen. Diese Maschinen müssen in der Lage sein, uns so gut zu verstehen, dass sie unsere unscharf formulierten Bedürfnisse in perfekt ausgeführte Programme übersetzen können, die genau diese Bedürfnisse erfüllen.

---

1 Es ist urkomisch, dass ich diese Zeile 2008 geschrieben habe und sie auch heute, 2024, noch Bestand hat. Manche Dinge ändern sich einfach niemals.

2 In Anführungszeichen, da echte Mathematiker so etwas nicht hoffen.

Dies wird nie passieren. Nicht einmal Menschen mit all ihrer Intuition und Kreativität sind bis jetzt in der Lage gewesen, aus den schwammigen Anforderungen ihrer Kunden erfolgreiche Systeme abzuleiten. Wenn wir überhaupt etwas aus der Disziplin der Anforderungsspezifikation gelernt haben, ist es Folgendes: Wohlspezifizierte Anforderungen sind genauso formal wie Code und können als ausführbare Tests dieses Codes verwendet werden!

Vergessen Sie nicht, dass Code letztlich die Sprache ist, in der wir die Anforderungen ausdrücken. Wir können Sprachen konstruieren, die näher an den Anforderungen angesiedelt sind. Wir können Tools erstellen, die uns helfen, diese Anforderungen zu parsen und zu formalen Strukturen zusammenzusetzen. Aber wir werden niemals die erforderliche Präzision eliminieren können – und deshalb wird es immer Code geben.

1.2 Schlechter Code

Vor langer Zeit habe ich das Vorwort zu dem Buch *Implementation Patterns*[3] von Kent Beck gelesen. Darin schreibt er: »... dieses Buch basiert auf einer recht fragilen Prämisse: dass guter Code eine Rolle spiele ...« Eine fragile Prämisse? Dem kann ich nicht zustimmen! Ich glaube, dass diese Prämisse zu den robustesten, am besten unterstützten und meistdiskutierten Prämissen unserer Zunft gehört (und ich glaube, das weiß Kent Beck auch). Wir wissen, dass guter Code eine Rolle spielt, weil wir uns so lange mit dem Mangel hieran auseinandersetzen mussten.

Ich kenne ein Unternehmen, das in den späten 80er-Jahren eine Killerapplikation herausbrachte. Sie war sehr beliebt, und zahlreiche professionelle Anwender kauften und nutzten sie. Aber dann wurden die Release-Zyklen immer größer. Bugs

3 [IMP]

wurden von einem Release zum nächsten nicht mehr repariert. Die Startzeiten wurden länger und die Abstürze häufiger. Ich erinnere mich an den Tag, an dem ich das Produkt frustriert abschaltete und niemals wieder benutzte. Kurz danach verschwand das Unternehmen vom Markt.

Zwei Jahrzehnte später traf ich einen früheren Mitarbeiter dieses Unternehmens und fragte ihn, was damals passiert sei. Die Antwort bestätigte meine Befürchtungen. Das Unternehmen hatte das Produkt zu schnell auf den Markt gebracht und im Code ein riesiges Chaos angerichtet. Je mehr Features dem Code hinzugefügt wurden, desto schlechter wurde er, bis das Unternehmen ihn einfach nicht mehr verwalten konnte. Es war der schlechte Code, der das Unternehmen in den Abgrund trieb.

Sind Sie jemals erheblich von schlechtem Code beeinträchtigt worden? Wenn Sie als Programmierer auch nur ein bisschen Erfahrung haben, dann haben Sie ein solches Hindernis viele Male erlebt. Tatsächlich haben wir eine spezielle Bezeichnung dafür: *Wading* (Waten). Wir waten durch schlechten Code. Wir kämpfen uns durch einen Morast verschlungener Schlingpflanzen und verborgener Fallgruben. Wir mühen uns ab, den richtigen Weg zu finden, und hoffen auf irgendwelche Hinweise, die uns zeigen, was passiert; aber alles, was wir sehen, ist ein schier endloses Meer von sinnlosem Code.

Natürlich sind Sie von schlechtem Code behindert worden. Also – warum haben Sie ihn geschrieben?

Haben Sie zu schnell gearbeitet? Waren Sie unter Druck? Wahrscheinlich. Vielleicht hatten Sie das Gefühl, keine Zeit für gute Arbeit zu haben, glaubten, Ihr Chef würde ärgerlich, wenn Sie sich die Zeit nehmen, Ihren Code aufzuräumen. Vielleicht waren Sie es einfach leid, an diesem Programm zu arbeiten, und wollten endlich damit fertig werden. Oder vielleicht haben Sie Ihren Stapel Arbeit angeschaut, die Sie längst hätten erledigen müssen, und sind zu dem Schluss gekommen, Sie müssen dieses Modul zusammenschustern, um mit dem nächsten weitermachen zu können. Wir alle kennen das.

Wir alle haben uns das Chaos angeschaut, das wir gerade angerichtet hatten, und dann beschlossen, es an einem anderen Tag zu beseitigen. Wir alle haben die Erleichterung erlebt, zu sehen, dass unser chaotisches Programm lief, und beschlossen, dass ein laufendes Chaos besser wäre als nichts. Wir alle haben uns vorgenommen, später zurückzukommen und das Chaos zu beseitigen. Natürlich kannten wir damals das Gesetz von LeBlanc nicht: *Später gleich niemals.*

1.2.1 Einstellung

Sind Sie jemals durch einen Morast gewatet, der so dicht war, dass es Wochen dauerte, um zu tun, was nur einige Stunden hätte dauern sollen? Haben Sie er-

lebt, dass eine Änderung, die nur eine Zeile hätte erfordern sollen, in Hunderten verschiedener Module durchgeführt werden musste? Diese Symptome kommen leider allzu oft vor.

Warum passiert das mit Code? Warum verrottet guter Code so schnell zu schlechtem Code? Dafür haben wir viele Erklärungen. Wir beklagen uns, dass die Anforderungen in einer Weise geändert wurden, die dem ursprünglichen Design zuwiderläuft. Wir jammern, dass der Zeitplan zu eng bemessen war, um die Aufgaben richtig zu erledigen. Geben dummen Managern und den intoleranten Kunden und nutzlosen Marketingtypen und einem unzureichenden Telefonsupport die Schuld. Aber der Fehler, lieber Dilbert, liegt nicht in unseren Sternen, sondern in uns selbst. Wir sind unprofessionell.

Diese Pille zu schlucken, mag etwas bitter sein. Wie könnte dieses Chaos unsere Schuld sein? Was ist mit den Anforderungen? Was ist mit dem Zeitplan? Gibt es etwa keine dummen Manager und nutzlose Marketingtypen? Tragen sie nicht einen Teil der Schuld?

Nein. Die Manager und Marketingleute fragen *uns* nach den Informationen, die sie benötigen, um Versprechungen und Zusagen zu machen; und selbst wenn sie uns nicht fragen, sollten wir keine Hemmungen haben, ihnen zu sagen, was wir denken. Die Benutzer wenden sich an uns, damit wir ihnen zeigen, wie ihre Anforderungen durch das System erfüllt werden können. Die Projektmanager benutzen unsere Informationen, um ihre Zeitpläne aufzustellen. Wir sind intensiv in die Planung des Projekts eingebunden und tragen einen großen Teil der Verantwortung für auftretende Fehler, insbesondere wenn diese Fehler mit schlechtem Code zu tun haben!

»Doch halt!«, sagen Sie. »Wenn ich nicht tue, was mein Manager sagt, werde ich gefeuert.« Wahrscheinlich nicht. Die meisten Manager wollen die Wahrheit wissen, selbst wenn sie sich nicht immer entsprechend verhalten. Die meisten Manager wollen guten Code haben, selbst wenn sie von ihrem Zeitplan besessen sind. Vielleicht verteidigen sie leidenschaftlich den Zeitplan und die Anforderungen; aber das ist ihr Job. Dagegen ist es Ihr Job, den Code mit gleicher Leidenschaft zu verteidigen.

Betrachten wir eine Analogie: Was würden Sie als Arzt machen, wenn ein Patient Sie auffordern würde, dieses blödsinnige Händewaschen bei der Vorbereitung auf einen chirurgischen Eingriff zu unterlassen, weil es zu viel Zeit kostet?[4] Natürlich hat der Patient Vorrang. Dennoch sollte der Arzt in diesem Fall die Forderung kompromisslos zurückweisen. Warum? Weil der Arzt mehr über die Risiken einer

4 Als das Händewaschen 1847 den Ärzten erstmals von Ignaz Semmelweis empfohlen wurde, wurde es mit der Begründung zurückgewiesen, die Ärzte seien zu beschäftigt und hätten keine Zeit, sich die Hände zwischen ihren Patientenbesuchen zu waschen.

Erkrankung und Infektion weiß als der Patient. Es wäre unprofessionell (und in diesem Fall sogar kriminell), wenn der Arzt der Forderung des Patienten nachgeben würde.

Deshalb ist es auch unprofessionell, dass sich Programmierer dem Willen von Managern beugen, die die Risiken nicht verstehen, die mit dem Erzeugen von Chaos im Code verbunden sind.

1.2.2 Das grundlegende Problem

Programmierer werden mit einem grundlegenden Wertedilemma konfrontiert. Erfahrene Entwickler wissen, dass ihre Arbeit durch alten chaotischen Code erheblich beeinträchtigt wird. Dennoch fühlen alle Entwickler den Druck, chaotischen Code zu schreiben, um Termine einzuhalten. Kurz gesagt: Sie nehmen sich nicht die Zeit, es richtig zu machen!

Echte Profis wissen, dass der zweite Teil dieses Dilemmas falsch ist. Man erfüllt einen Termin eben nicht, indem man chaotischen Code produziert. Tatsächlich verlangsamt chaotischer Code Ihre Arbeit sofort und führt dazu, dass Sie Ihren Termin nicht einhalten können. Und er wird auch alle anderen verlangsamen, die sich nach Ihnen damit befassen müssen. Dieses Problem wird sich weiterentwickeln und fortpflanzen, bis Sie (und andere) eine Frist nach der anderen versäumen.

Die einzige Methode, den Termin einzuhalten, besteht darin, den Code jederzeit so sauber wie möglich zu halten.

Der beste Weg zu schneller Arbeit ist sorgfältige Arbeit.

1.3 Sauberen Code schreiben – eine Kunst?

Angenommen, Sie glaubten, chaotischer Code wäre eine beträchtliche Behinderung Ihrer Arbeit. Wenn Sie jetzt akzeptieren, dass die einzige Möglichkeit, schneller zu arbeiten, darin besteht, den eigenen Code sauber zu halten, müssen Sie sich zwangsläufig fragen: »Wie schreibe ich sauberen Code?« Es hat keinen Sinn, zu versuchen, sauberen Code zu schreiben, wenn Sie nicht wissen, wie sauberer Code aussieht!

Sauberen Code zu schreiben, erfordert den disziplinierten Einsatz zahlreicher kleiner Techniken, die mit einem sorgfältig erworbenen Gefühl für »Sauberkeit« angewendet werde, um so schlechten Code in sauberen Code zu transformieren.

Und es ist immer eine Transformation. Niemand schreibt sauberen Code. Wir alle schreiben schlechten (sprich: unsauberen) Code. Sauberer Code entsteht dann, wenn wir unseren Code nach dem Schreiben bereinigen. Hier kommt das Gesetz

ins Spiel, das Kent Beck formulierte und das Ihnen in diesem Buch wieder und wieder begegnen wird:

Zunächst bring es zum Laufen. Dann mach es richtig.

1.3.1 Was ist sauberer Code?

Es gibt wahrscheinlich so viele Definitionen wie Programmierer. Deshalb habe ich einige sehr bekannte und sehr erfahrene Programmierer nach ihrer Meinung gefragt. Die Antworten habe ich auf den folgenden Seiten zusammengefasst.

»Sauberer Code erledigt eine Aufgabe gut.«

– Bjarne Stroustrup, Erfinder von C++ und Autor von *The C++ Programming Language*

Dies ist eine alte und bewährte Maxime. In den 70ern haben wir immer gesagt, dass ein Modul eine Aufgabe erledigen sollte, sie gut erledigen sollte und nur sie erledigen sollte. Aber was bedeutet »eine Aufgabe«? Ende der 80er habe ich eine 3.000 Zeilen lange C-Funktion namens `gi` geschrieben. Das stand für *graphic interpreter*. Hätte mir damals jemand gesagt, dass meine ellenlange Funktion mehr als eine Aufgabe erledigt, hätte ich geantwortet: »Keinesfalls. Sie interpretiert Grafiken.«

Das Problem mit dem Begriff »eine Aufgabe« ist, dass jeder von uns den Begriff auf seine Weise definiert, er also subjektiv ist. Aber ich glaube, ich kenne einen Weg, um diesen Begriff *objektiv* zu definieren, worauf ich später zurückkommen werde. Ich bin überzeugt, dass meine Definition Sie überzeugen wird. Zumindest werden Sie zustimmen, dass sie so gut wie objektiv ist. Sie wird Ihnen vielleicht nicht gefallen, aber ich gehe jede Wette ein, dass Sie mir im Wesentlichen nicht widersprechen werden.

»Sauberer Code liest sich wie wohlgeschriebene Prosa.«

– Grady Booch, bekannter Autor vieler Bücher zum Thema Softwareentwicklung

Was für eine Aussage! Wann haben Sie zuletzt Code gesehen, der sich wie wohlgeschriebene Prosa gelesen hat? Ich werde Ihnen in diesem Buch zeigen, wie Sie mit Ihrem Code dieses Ziel erreichen oder ihm zumindest nahekommen können.

Verstehen Sie mich nicht falsch. Ihr Code liest sich dadurch nicht wie *ein Hemingway-Roman,* aber es ist durchaus möglich, Code zu konstruieren, der so lesbar ist, dass sogar Nichtprogrammierer ihn bis zu einem gewissen Grad nachvollziehen[5] können.

»Sauberer Code sieht immer so aus, als wäre er von jemandem geschrieben worden, dem dies wirklich wichtig war.«

– Michael Feathers, Autor von *Effektives Arbeiten mit Legacy Code*

5 Im Sinne einer gewissen Definition des Begriffs *nachvollziehbar.*

Vielleicht könnte man den letzten Teil von Michaels Zitat neu formulieren: »dem Sie wirklich wichtig waren.« Ein Programmierer wird seinen Code bereinigen und lesbar machen, damit der nächste Programmierer, der sich damit befassen muss, es einfacher hat. Die Sorgfalt, die er aufwendet, damit sein Code nicht in die Irre führt oder verwirrt, kommt jenen zugute, die diesen Code warten oder pflegen müssen. Tatsächlich ist *Sorgfalt* der Dreh- und Angelpunkt für sauberen Code.

Im Umkehrschluss bedeutet das, dass nicht bereinigter Code auf nachlässige Weise geschrieben und veröffentlicht wurde – ihm mangelt es an Sorgfalt.

»Sie wissen, dass Sie an sauberem Code arbeiten, wenn jede Routine, die Sie lesen, ziemlich genau so funktioniert, wie Sie es erwartet haben.«

– Ward Cunningham, Erfinder des Wikis, Erfinder von Fit, Miterfinder des eXtreme Programming. Treibende Kraft hinter den Design Patterns. Smalltalk- und OO-Vordenker. Der Pate aller Leute, denen ihr Code nicht egal ist.

Stellen Sie sich vor, Code zu lesen, bei dem Sie jeder Zeile zustimmen können. Sie blättern nickend durch den Quelltext. Ihre Augen gleiten über den Code wie ein heißes Messer durch Butter. Es gibt keine Stolpersteine, keine cleveren Tricks, keine Logiksprünge. Sie lesen den Code von Anfang bis Ende und bejahen die gesamte Schöpfung vor Ihren Augen.

Klingt utopisch? Das mag sein, aber Utopien sind Träume, und Träume sollte man erreichen wollen. In diesem Buch zeige ich Ihnen, wie Sie diesem Traum einen oder auch mehrere Schritte näherkommen.

»Sauberer Code lässt sich rasch begreifen. Jede Funktion nimmt höchstens eine Bildschirmseite ein und umfasst nur wenige bewegliche Teile – aber dafür aussagekräftige Namen. Andere Programmierer können sie auch Jahre später noch problemlos verstehen. In diesem Code gibt es keine Überraschungen. Er ist gut geschrieben und lässt sich ebenso einfach löschen.«

– Mark Seeman, Autor von *Code That Fits in Your Head,* beliebter Blogger über Software-Craftsmanship und funktionale Programmierung

Klingt das nicht wunderbar? In gewisser Weise ist dieses Zitat eine Zusammenfassung dieses Buchs.

1.4 Das große Ganze

Sauberer Code liest sich wie wohlgeschriebene Prosa, weil er sorgfältig in Funktionen und Module unterteilt wurde – wie Prosa in Sätze und Absätze unterteilt ist, die jeweils einen Aspekt behandeln, ein Thema, ein Kernkonzept. Wohlgeschriebene Prosa fließt von einem Thema zum nächsten und bildet so aus vielen kleinen Einzelteilen ein vollständiges Kunstwerk.

1.4.1 Was bringt uns sauberer Code?

Sehen Sie sich um. Wie viele Computer können Sie mit zwei oder drei Schritten erreichen? Haben Sie an Ihr Smartphone, eine Smartwatch, Ihre Noise-Cancelling-Kopfhörer und den Funkschlüssel für Ihr Auto gedacht? Tragen Sie vielleicht ein smartes medizinisches Gerät?

Wie sieht es in Ihrer Küche aus? In wie vielen Geräten steckt ein Computer? Mikrowelle, Backofen oder Herd mit Zeitsteuerung, Mixer, Kühlschrank mit smarten Funktionen?

Besitzen Sie einen Smart-TV? Müssen Sie an Ihren Heizkörperthermostaten drehen oder steckt darin bereits ein Computer? Wird Ihr Haus durch ein intelligentes Sicherheitssystem geschützt? Welche Funktionen bietet Ihre Hi-Fi-Anlage?

Besitzen Sie ein Auto? Wie viele Zeilen Code sorgen dafür, dass dieses Auto funktioniert? Steckt vielleicht ein Navigationssystem mit GPS-Empfang darin? Und in Ihrem Handy? Finden Sie noch ohne diese Helfer ans Ziel?

Denken Sie einen Moment darüber nach. Denken Sie noch einmal, dieses Mal etwas länger. Fast alles, was wir in dieser modernen Welt tun, hängt irgendwie von Software ab. Ohne Software könnten Sie kein Popcorn in der Mikrowelle machen. Sie könnten nicht fernsehen. Sie könnten nicht telefonieren. Sie könnten Ihr Auto nicht in Bewegung setzen. Sie könnten nicht im Laden bezahlen. Sie könnten nichts verkaufen. Es könnten keine neuen Gesetze erlassen werden. Es könnten keine bestehenden Gesetze durchgesetzt werden. Sie könnten keine Ansprüche an Ihre Versicherung stellen. Software ist die tragende Säule der modernen Gesellschaft.

Wie ich bereits in der Einleitung sagte, stünden ohne Software die Räder der Zivilisation schnell still.

Sie und ich, wir regieren die Welt.

Es mag andere Leute geben, die denken, sie würden die Welt regieren, aber in Wirklichkeit überlassen sie es uns Programmierern, die Regeln ihrer Regentschaft in eine Software für die Maschinen umzusetzen, die unseren Alltag bestimmen.

Unsere Zivilisation hängt auf kritische Weise von uns ab. Ohne uns kommt die Zivilisation über Nacht zum Stillstand. Niemand ist darauf vorbereitet, wieder auf Papierakten umzusteigen, um den Betrieb am Laufen zu halten.

Aber sind wir als Programmierer moralisch und professionell und verantwortlich genug, um die Zivilisation auf unseren Schultern zu tragen? Oder folgen wir einem Kurs, der geradewegs ins Verderben führt? Werden Sie und ich für eine Katastrophe verantwortlich sein?

Leider gibt es genug Negativbeispiele, auch aus der jüngsten Vergangenheit. Ein Softwarefehler hat dazu geführt, dass zwei voll besetzte Passagierflugzeuge nahezu mit Schallgeschwindigkeit auf der Erde zerschellt sind. 346 Menschen sind dabei gestorben. Softwarefehler in Kfz-Steuergeräten haben Dutzende Tote und Hunderte von Verletzten auf dem Gewissen, weil Gas- und Bremspedal nicht richtig funktionierten. Knight Capital hat wegen eines Softwarefehlers in nur 45 Minuten ganze 450 Millionen Dollar verloren. Die Website `healthcare.gov` legte aufgrund von Überlastung und Designfehlern einen grandiosen Fehlstart hin.

Es ist durchaus möglich, dass in nicht allzu ferner Zukunft Zehntausende Menschen aufgrund eines einzigen Softwarefehlers ihr Leben verlieren. Ich übertreibe

nicht. Und wissen Sie auch, wem man die Schuld dafür zuweisen wird? Uns, den Programmierern.

Nicht unseren Bossen. Nicht den CEOs. Natürlich werden auch sie Rede und Antwort stehen müssen. Aber alle Finger werden auf uns zeigen, denn schließlich haben wir den Code geschrieben, der die Katastrophe ermöglicht (oder zumindest nicht verhindert) hat.

Stellen Sie sich die Gerichtsverfahren vor, die folgen werden! Die Gutachten von Beratern und Fachleuten! Sie werden die Tausende globale Variablen, schlecht benannte Argumente, falsch verwaltete und fehlende Semaphore, Speicherlecks, unsachgemäß zugewiesene Stacks, fragmentierte Heaps, Qualitätskompromisse zugunsten der Geschwindigkeit, Funktionen mit einem Umfang von 3000 Zeilen, Module mit dupliziertem Code, der zum Teil (aber nicht in seiner Gesamtheit) gefixt wurde, als ursächlich benennen.

Auch das ist bereits geschehen. Mehrfach. Zu oft.

Wie werden die Regierungen reagieren, die den Verlust von Zehntausenden ihrer Bürger zu beklagen haben? Vermutlich mit neuen Gesetzen.

Welcher Art mögen diese Gesetze sein? Werden die Gesetze bestimmte Programmiersprachen, Plattformen und Frameworks vorschreiben? Zu befolgende Prozesse, einzuholende Genehmigungen, die zu erstellende Dokumentation, einzuhaltende Prüfverfahren und Tests, bestimmte Anforderungen an die Ausbildung, Qualifikation oder den Lese- und Lehrstoff?

Ich bin mir absolut sicher, dass diese neuen Gesetze weder Ihnen noch mir gefallen würden. Und ich bin mir ziemlich sicher, dass diese Gesetze wenig zur Verbesserung der Situation beitragen werden.

Die wichtigste Antwort auf die Frage, darauf, warum wir unseren Code bereinigen sollten, lautet daher: weil wir Profis sind. Und wenn wir unserer (Eigen-)Verantwortung nicht nachkommen, wird uns der Gesetzgeber unbequeme und ineffektive Regeln aufzwingen.

Aber es gibt einen weiteren Grund für sauberen Code. Einen Grund, der Sie sehr viel direkter betrifft.

1.4.2 Produktivität

Ich habe in diesem Buch die Frage gestellt, ob Sie jemals erheblich von schlechtem Code beeinträchtigt worden sind. Auf alle erfahrenen Programmierer trifft das ziemlich sicher zu. Und ich wiederhole eine andere Frage: Warum haben Sie diesen Code geschrieben?

Warum schreiben wir alle Code, von dem wir wissen, dass er uns erheblich behindern wird? Die Antwort ist stets dieselbe: weil wir schneller sein wollen. Ich

überlasse es Ihnen, sich mit der logischen Widersprüchlichkeit dieser Antwort zu befassen.

Ich kenne alle Ausreden. Sie alle laufen auf dasselbe hinaus: »Niemand gibt uns die Zeit, es richtig zu machen.« Das stimmt nicht. Der beste Weg zu schneller Arbeit ist sorgfältige Arbeit. Der beste Weg zur Einhaltung des Zeitplans und zur pünktlichen Lieferung besteht darin, gute Arbeit zu leisten. Wenn Sie schnell arbeiten möchten, sollten Sie alles vermeiden, was Sie verlangsamt! Halten Sie Ihren Code sauber!

Ein chaotisches Modul verlangsamt alle, die sich damit auseinandersetzen müssen, und zwar jedes Mal, wenn sie es tun müssen. Die Kosten für das Durcheinander fallen immer und immer wieder an, die Kosten für das Bereinigen dagegen nur einmalig.

Wenn ein Team Chaos im Code zulässt, wird es dadurch irgendwann auf einen Bruchteil seiner ursprünglichen Produktivität gebremst. Ich kenne Teams, die ihre Schätzungen von ursprünglich einem oder zwei Tage auf Wochen oder gar Monate revidieren mussten.

Je langsamer ein Team wird, desto stärker wird der Druck. Je stärker der Druck wird, desto weniger sorgfältig arbeitet das Team. Jeder neue Chaosfetzen trägt zu den Problemen für das gesamte Team bei und es wird immer langsamer. Das ist ein Teufelskreis. Programmierer mit ein paar Jahren Erfahrung haben ihn in aller Regel bereits mehrmals durchlaufen.

Aber es wird noch schlimmer: Führungskräfte versuchen krampfhaft, die Produktivität zu erhöhen, und heuern neue Leute an. Unglücklicherweise hat das die gegenteilige Wirkung. Die neuen Leute, die auf den chaotischen Code angesetzt werden, nehmen sich ein Beispiel am Rest des Teams und arbeiten auf dieselbe Weise. Jetzt richten also mehr Menschen als zuvor Chaos an, das Rad dreht sich immer schneller und die Produktivität sinkt erneut.

Führungskräfte fragen sich erstaunt, wieso die Produktivität trotz zusätzlichem Personal nicht zunimmt. Einige versuchen es mit noch mehr Neuanstellungen, denn <*Sarkasmus*> die Definition von Vernunft ist, immer wieder das Gleiche zu tun und andere Ergebnisse zu erwarten. </*Sarkasmus*> Am Ende bleibt den Führungskräften nichts übrig, als die Entwickler zurate zu ziehen.

Und die Entwickler können diese Frage beantworten. Sie kennen die Antwort seit Monaten, wenn nicht Jahren. Sie liefern sie nur zu gern: »Das gesamte System muss von Grund auf neu designt werden.«

Entsetzte Gesichter auf der anderen Seite! Haben die Entwickler gerade wirklich gesagt, dass ihre gesamte bisherige Arbeit eingestampft werden muss und wertlos ist? Die Entwickler schieben hinterher, dass dieses neue Design narrensicher ist.

Es wird nicht mehr zu einem Rückgang der Produktivität führen. Dieses Mal wird alles besser!

Das glauben die Führungskräfte nicht. Sie wehren sich auf jede erdenkliche Art gegen ein Redesign. Aber haben sie eine andere Wahl? Die Fachleute berichten, dass die einzige Hoffnung darin besteht, das System von Grund auf neu zu gestalten. Am Ende stimmt die Unternehmensführung zu. Denn sie hat keine andere Wahl.

Und die Entwickler feiern und jubeln: »Halleluja! Wir fangen mit einem leeren Bildschirm an, ganz ohne Chaos. Unser Leben wird endlich gut sein!«

Und es kommt anders. Denn man wählt die zehn besten Männer und Frauen aus, das Tiger-Team. Sie erhalten den Auftrag, das Redesign zu übernehmen. Das Tiger-Team? Moment! Sind das nicht dieselben Leute, die das bisherige Chaos verursacht haben? Egal. Das Team schließt sich in ein Besprechungszimmer ein und arbeitet daran, dem Projekt eine neue und schöne Richtung zu geben. Es stellt die Weichen für die Zukunft.

In der Zwischenzeit fällt dem Rest der Entwickler die unliebsame Aufgabe zu, das alte System am Laufen zu halten. Denn immerhin ruht auf diesem Fundament das gesamte Unternehmen. Es gilt, Bugs zu beseitigen. Neue Features werden benötigt.

Zwischenfrage: Woher bekommt das Tiger-Team seine Anforderungen? Gibt es ein Pflichtenheft? Nimmt es jemand ernst? Oder besteht das Pflichtenheft aus so vielen Überarbeitungen, Änderungen und hingekritzelten Notizen, dass sich niemand mehr darin zurechtfindet? Auch egal. Denn zum Glück gibt es ja eine zentrale Stelle, die alle Anforderungen enthält: den alten Programmcode.

Da sitzt das Tiger-Team nun im Besprechungszimmer, brütet über dem alten Code und versucht, herauszufinden, was in aller Welt das System eigentlich tut. Vor der Tür arbeitet der Rest des Teams an diesem Code, fixt Bugs und fügt Features hinzu.

Es ist wie bei Hase und Igel: Das Tiger-Team hinkt dem Wartungsteam immer einen Schritt hinterher. Wenn das Tiger-Team schließlich dort ankommt, wo das Wartungsteam war, ist das Wartungsteam schon wieder ein paar Meter weiter.

Es ist vielleicht wie Zenons Paradoxon von Achilles und der Schildkröte. Das Tier erhält anfangs einen Vorsprung. Der schnellere Achilles läuft los, aber als er den Startpunkt der Schildkröte erreicht, hat diese sich bereits weiterbewegt. Zenon wollte argumentieren, dass es nicht möglich war, die Schildkröte jemals einzuholen, ja, dass vielleicht sogar gar keine Bewegung möglich war.

Natürlich können wir mit ein wenig Mathematik nachweisen, dass Achilles die Schildkröte irgendwann überholt. Überraschung: Diese Art der Mathematik und Softwareprojekte sind keine dicken Freunde.

Ich erinnere mich an ein Projekt, das von Grund auf neu gestaltet wurde. Selbst nach zehn Jahren war das neue System noch nicht im Einsatz. Dabei hatte man den Kunden viele Jahre zuvor gesagt, dass es bald ein neues und viel besseres System gäbe. Doch jedes Warten war vergeblich. Denn wann immer das Unternehmen den Vorgänger ersetzen wollte, beschwerten sich die Kunden darüber, dass das neue System nicht alle Möglichkeiten des alten Systems bot. Ganz gleich, wie sehr es sich bemühten, das Tiger-Team hinkte dem Wartungsteam immer einen Schritt hinterher. Es gibt in dieser Konstellation ein weiteres Problem: Das Wartungsteam sorgte für Umsätze. Natürlich bekam das Tiger-Team nur ein begrenztes Budget, während man dem Wartungsteam jeden Wunsch von den Lippen ablas.

Über viele Monate und Jahre wurden die ursprünglichen Mitglieder des Tiger-Teams durch andere Entwickler ersetzt, befördert oder bekamen neue Aufgaben zugewiesen. Zehn Jahre später war keines seiner ursprünglichen Mitglieder mehr da.

Am Ende sprach das Unternehmen vor lauter Verzweiflung ein Machtwort: »Liebe Kunden, ihr müsst ab sofort das neue System verwenden. Das alte wird abgeschaltet.« Heulen und Wehklagen brach aus – aber die Kunden hatten keine andere Wahl. Das Unternehmen konnte nicht länger beide Entwicklungsprojekte finanzieren.

Das Tiger-Team wurde von der Ankündigung ebenso überrascht wie die Kunden. Mit entsetzten Gesichtern forderten sie ein Meeting mit der Führungsetage: »Das können wir nicht produktiv schalten! Dieses System muss ganz neu designt werden!«

Mag sein, dass diese Geschichte ein wenig überspitzt ist, aber tatsächlich finden sich darin meine eigenen Erfahrungen und Berichte von Kunden wieder. Ich bin sicher, dass der ein oder andere Teil meiner Leserschaft sich darin wiedererkannt hat.

Zurück zur Ausgangsfrage: Warum sollten wir unseren Code bereinigen? Weil sauberer Code die beste Grundlage für produktive Arbeit im Team ist. Wenn wir das Chaos wuchern lassen, bereiten wir der Katastrophe aus meiner Geschichte den Weg. Doch wenn wir für sauberen Code sorgen, bleibt die Produktivität erhalten. Die Entwickler fordern kein Neudesign, und die Unternehmensführung wird nicht sinnlos mehr und mehr Menschen auf das Problem ansetzen, um die Produktivität zu erhöhen.