Kapitel 1

Professionalität

»Lach, Curtin, alter Junge. Das ist ein toller Streich, den uns Gott, das Schicksal oder die Natur gespielt hat, wer immer dir lieber ist. Aber wer oder was ihn auch gespielt hat, hatte jedenfalls Sinn für Humor! Ha!«

Howard, in: Der Schatz der Sierra Madre

So, Sie wollen also professioneller Software-Entwickler werden, nicht wahr? Sie wollen der Welt hoch erhobenen Hauptes zurufen: »Ich bin ein *Profi*!« Sie wollen, dass man Sie respektvoll betrachtet und Ihnen mit Achtung begegnet. Sie wollen, dass Mütter auf Sie zeigen und ihren Kindern erzählen, dass sie wie Sie sein sollen. Sie wollen das ganze Paket. Richtig?

1.1 Seien Sie vorsichtig, wonach Ihnen verlangt

Professionalität ist ein belasteter Begriff. Sicherlich ist er ein Emblem von Ehre und Stolz, aber eben auch das Kennzeichen für Verantwortung und Haftung. Das geht natürlich beides Hand in Hand. Sie können nicht auf etwas stolz sein und mit Ehre tragen, für das Sie nicht verantwortlich gemacht werden können.

Es ist viel einfacher, unprofessionell zu sein. Wer unprofessionell ist, braucht sich nicht für seine Arbeit verantworten – das überlässt er seinen Arbeitgebern. Wenn so jemand einen Fehler macht, sorgt der Arbeitgeber dafür, dass der Schlamassel behoben wird. Aber wenn ein Profi einen Fehler begeht, richtet *er* den Schaden wieder her.

Was würde passieren, wenn Ihnen bei einem Modul ein Bug durch die Lappen geht, der Ihre Firma 10.000 Euro kostet? Ein unprofessioneller Mitarbeiter zuckt mit den Schultern, murmelt »Dumm gelaufen« und schreibt das nächste Modul. Der Profi würde der Firma einen Scheck über 10.000 Euro ausstellen[3]!

Genau, fühlt sich schon etwas anders an, wenn es Ihr eigenes Geld ist, nicht wahr? Aber dieses Gefühl begleitet den Profi die ganze Zeit. Tatsächlich ist dieses Gefühl die Essenz der Professionalität. Weil es, wie Sie sehen werden, bei der Professionalität um die Übernahme von Verantwortung geht.

3 Hoffentlich hat er eine gute Haftpflichtversicherung!

1.2 Verantwortung übernehmen

Sie haben die Einführung gelesen, oder? Falls nicht, blättern Sie bitte noch mal zurück. Darin wird der Kontext aufgestellt für alles, was in diesem Buch noch folgt.

Ich habe erfahren, was es bedeutet, Verantwortung zu übernehmen, als ich die Konsequenzen erlitt, es eben nicht zu tun.

1979 arbeitete ich für eine Firma namens Teradyne. Ich war als »verantwortlicher Ingenieur« für die Software zuständig, die ein mini- und mikrocomputerbasiertes System steuerte, das die Qualität von Telefonleitungen messen sollte. Der zentrale Minicomputer war über 300-Baud-Telefonleitungen (entweder extra dafür reserviert oder als Einwahlverbindung) mit Dutzenden Mikrocomputern als Satelliten verbunden, die die Messgeräte steuerten. Der Code war komplett in Assembler geschrieben.

Unsere Kunden waren die Servicemanager von großen Telefonfirmen. Jeder trug die Verantwortung für 100.000 Telefonanschlüsse oder mehr. Mein System half den Servicebereichsmanagern dabei, Fehlfunktionen und Probleme in den Leitungen zu finden und zu beheben, ehe sie von den Kunden bemerkt wurden. Das reduzierte die Anzahl der Kundenbeschwerden. Deren Zahl wurde behördlich gemessen und zur Regelung der Gebühren genutzt, die die Telefonfirmen für ihre Dienste berechnen durften. Kurz gesagt waren diese Systeme unglaublich wichtig.

Jede Nacht wurden diese Systeme routinemäßig durchgemessen, und der zentrale Minicomputer sorgte dafür, dass alle Satelliten-Mikrocomputer alle Telefonleitungen testeten, für die sie jeweils verantwortlich waren. Jeden Morgen bekam der Zentralcomputer die Liste der schadhaften Leitungen plus Hinweise auf die Art der Probleme. Die Servicebereichsmanager nutzten diese Berichte dann, um Zeitpläne und Reparaturaufträge für die Mechaniker zu erstellen, um die Defekte noch vor den Beschwerden der Kunden zu beheben.

Einmal lieferte ich ein neues Release an mehrere Dutzend Kunden aus. »Ausliefern« ist hier genau das richtige Wort. Ich schrieb die Software auf Bänder und verschickte sie an die Kunden. Diese setzen die Bänder in die jeweiligen Bandlaufwerke ein und starteten die Systeme neu.

Durch das neue Release wurden einige kleinere Mängel behoben und ein neues Feature hinzugefügt, das von unseren Kunden bestellt wurde. Wir hatten ihnen gesagt, dass dieses Feature zu einem bestimmten Datum erhältlich sein würde. Ich hatte es gerade noch geschafft, die Bänder per Express über Nacht zu versenden, damit sie am zugesagten Datum eintreffen konnten.

Zwei Tage später bekam ich einen Anruf von unserem Außendienstmanager Tom. Er berichtete mir, dass mehrere Kunden sich darüber beschwert hatten, dass der

»nächtliche Testlauf« nicht abgeschlossen worden war und sie keine Berichte bekommen hatten. Mein Herz rutschte mir in die Hose, denn um die Software rechtzeitig versenden zu können, hatte ich mir erspart, diese Routine zu testen. Ich hatte die restliche Funktionalität des Systems weitgehend getestet, aber es hätte Stunden gedauert, eben diese Routine zu testen, und ich musste die Software ausliefern. Keiner der Bugfixes befand sich im Routine-Code, also fühlte ich mich auf der sicheren Seite.

Dass einer dieser nächtlichen Berichte unter den Tisch gefallen war, war ein *großes* Problem. Es bedeutete, dass die Mechaniker weniger zu tun hatten und später überbucht sein würden. Manche Kunden würden einen Defekt bemerken und sich beschweren. Der Verlust dieser nächtlichen Daten würde ausreichen, damit ein Servicebereichsmanager Tom anrufen und ihn »auf den Pott« setzen würde.

Ich startete unser Laborsystem, lud die neue Software und startete dann eine Routine. Das dauerte mehrere Stunden, wurde dann aber abgebrochen: Die Routine schlug fehl. Hätte ich diesen Test vorm Ausliefern durchlaufen lassen, hätten die Servicebereiche keine Daten verloren, und die Servicebereichsmanager hätten Tom nicht zur Rede stellen müssen.

Ich rief Tom an, um ihm zu sagen, dass ich das Problem reproduzieren könne. Er sagte mir, dass die meisten der anderen Kunden ihn in gleicher Angelegenheit schon angerufen hätten. Dann wollte er wissen, bis wann ich das beheben könne. Das wisse ich noch nicht genau, entgegnete ich, aber ich würde dran arbeiten. In der Zwischenzeit, ergänzte ich noch, sollten die Kunden auf die alte Software zurückgreifen. Tom war sauer auf mich und meinte, dass dies die Kunden doppelt träfe, weil sie nicht nur die Daten einer ganzen Nacht verloren hätten, sondern überdies auch nicht mit dem versprochenen neuen Feature arbeiten konnten.

Der Bug war schwer zu finden, und die Testläufe dauerten mehrere Stunden. Der erste Fix funktionierte nicht. Der zweite auch nicht. Ich probierte alles Mögliche aus und brauchte deswegen mehrere Tage, bis ich herausfand, wo der Hase im Pfeffer lag. Die ganze Zeit rief Tom mich alle paar Stunden an und hakte nach, wann ich endlich das Problem gelöst hätte. Er sorgte auch dafür, dass ich möglichst viel davon mitbekam, mit welchen Klagen ihm die Servicebereichsmanager in den Ohren lagen, und wie peinlich es für ihn war, ihnen sagen zu müssen, dass sie die alten Bänder einlegen sollten.

Zum Schluss fand ich endlich den Defekt, lieferte die neuen Bänder aus, und alles lief wieder normal. Tom, der nicht mein Boss war, regte sich wieder ab, und wir konnten die ganze Episode hinter uns lassen. Als sich die Wogen gelegt hatten, kam mein Boss vorbei und meinte: »Ich wette, dieses kommt nicht noch einmal vor.« Dem konnte ich nur zustimmen.

Beim Nachdenken über diese Geschichte erkannte ich, wie unverantwortlich es gewesen war, ohne Testen der Routine die Software auszuliefern. Ich hatte den

Test natürlich deswegen vernachlässigt, um sagen zu können, dass meine Lieferung pünktlich war. Es ging für mich darum, nicht mein Gesicht zu verlieren. Weder hatte ich mir Gedanken um die Kunden gemacht noch Sorgen um meinen Arbeitgeber. Ich hatte nur auf meine eigene Reputation geachtet. Ich hätte bereits früh schon die Verantwortung übernehmen und Tom informieren sollen, dass die Tests nicht vollständig und zufriedenstellend abgeschlossen waren und ich nicht in der Lage war, die Software rechtzeitig auszuliefern. Das wäre nicht einfach gewesen, und Tom hätte sich sicherlich darüber aufgeregt. Aber kein Kunde hätte seine Daten verloren, und kein Servicemanager hätte angerufen.

1.3 Erstens: Richte keinen Schaden an

Wie übernimmt man also Verantwortung? Es gibt da einige Prinzipien. Vielleicht wirkt es arrogant, sich auf den Hippokratischen Eid zu beziehen, aber kann es eine bessere Quelle geben? Und tatsächlich: Ist es nicht wirklich sinnvoll, dass die erste Verantwortung und das erste Ziel eines angehenden Profis sein sollte, die eigene Macht für etwas Gutes einzusetzen?

Welche Schäden kann ein Software-Entwickler anrichten? Vom reinen Standpunkt der Software aus kann er (oder sie) sowohl die Funktion als auch die Struktur der Software beschädigen. Wir werden untersuchen, wie das genau zu vermeiden ist.

1.3.1 Beschädige nicht die Funktion

Wir wollen naturgemäß, dass unsere Software funktioniert. Tatsächlich sind die meisten von uns heute Programmierer, weil wir es mal geschafft haben, dass etwas funktionierte, und dieses Gefühl wollen wir gerne wieder haben. Aber wir sind nicht die Einzigen, die wollen, dass die Software funktioniert. Unsere Kunden und Arbeitgeber wollen das auch. Tatsächlich bezahlen sie uns dafür, dass wir Software erstellen, damit sie genauso wie gewünscht funktioniert.

Wir beschädigen die Funktion unserer Software, wenn wir Bugs schaffen. Somit dürfen wir keine Bugs produzieren, wenn wir professionell sein wollen.

»Aber Moment mal!«, höre ich Sie sagen. »Das ist unrealistisch. Software ist zu komplex, als dass man sie ohne Bugs erstellen könnte.«

Natürlich haben Sie recht. Software *ist* zu komplex, als dass man sie ohne Bugs erstellen kann. Bedauerlicherweise entlässt Sie das nicht aus der Verantwortung. Der menschliche Körper ist zu komplex, als dass man ihn in seiner Gesamtheit verstehen könnte, aber Ärzte legen trotzdem einen Eid ab, keine Schäden daran anzurichten. Wenn die sich schon bei einem solchen Thema dermaßen festlegen, wie könnten wir uns dann selbst vom Haken lassen?

»Soll das etwa heißen, dass wir perfekt sein müssen?« Höre ich da jemanden widersprechen?

Nein, ich will Ihnen nur sagen, dass Sie für Ihre Unzulänglichkeiten verantwortlich sind. Die Tatsache, dass in Ihrer Software mit Sicherheit Bugs drinstecken werden, bedeutet nicht, dass Sie keine Verantwortung dafür tragen. Die Tatsache, dass es praktisch unmöglich ist, perfekte Software zu schreiben, heißt im Gegenzug nicht, dass Sie die Verantwortung für die Unvollkommenheit ablehnen können.

Es gehört zum Los eines Profis, für Fehler verantwortlich zu sein, auch wenn diese Fehler praktisch mit Sicherheit geschehen werden. Also, mein angehender Profi, Sie müssen nun als Allererstes lernen, sich zu entschuldigen. Entschuldigungen sind notwendig, aber nicht ausreichend. Sie dürfen nicht einfach die gleichen Fehler immer wieder machen. Wenn Sie in Ihrer Profession reifen, sollte Ihre Fehlerrate schnellstmöglich gegen Null gehen. Sie wird nie auf Null landen, aber es obliegt Ihrer Verantwortung, dem so nahe wie möglich zu kommen.

Die Qualitätssicherung sollte nichts finden

Wenn Sie also Ihre Software freigeben, sollten Sie davon ausgehen, dass von der Qualitätssicherung keine Probleme entdeckt werden. Es ist außerordentlich unprofessionell, an die Qualitätssicherung Code zu schicken, von dem Sie *wissen*, dass er mangelhaft ist. Und bei welchem Code können Sie davon ausgehen, dass er mangelhaft ist? Das ist jeder Code, bei dem Sie *unsicher* sind!

Manche nutzen die Qualitätssicherung als Bug-Netz. Sie schicken der QS nicht gründlich geprüften Code und verlassen sich darauf, dass dort die Bugs gefunden und an die Entwickler zurückgemeldet werden. Tatsächlich vergüten manche Firmen ihre Qualitätssicherung anhand der aufgedeckten Bugs. Je mehr Bugs, desto besser die Entlohnung.

Es ist egal, dass dies ein absolut kostspieliges Verhalten ist, das Firma und Software beschädigt. Es ist egal, dass dieses Verhalten Zeitpläne ruiniert und das Vertrauen des Unternehmens ins Entwicklerteam untergräbt. Es ist egal, dass ein solches Verhalten einfach nur faul und unverantwortlich ist. Wenn man Code für die Qualitätssicherung freigibt, von dem man nicht weiß, ob er funktioniert, ist das einfach unprofessionell. Das verletzt die Regel »Richte keinen Schaden an«.

Wird die Qualitätssicherung Bugs finden? Wahrscheinlich, also sollte man sich schon mal ein paar Entschuldigungen zurechtlegen – und sich dann auf Weg machen herauszufinden, warum diese Bugs durchrutschen konnten, und etwas dafür tun, dass das nicht noch einmal passiert.

Jedes Mal, wenn die Qualitätssicherung – oder schlimmer noch: ein *User*! – ein Problem findet, sollten Sie überrascht und verärgert sein und entschlossen verhindern, dass das erneut passiert.

Sie *müssen* wissen, ob er funktioniert

Woher wissen Sie, dass Ihr Code funktioniert? Ganz einfach: Testen Sie den Code. Testen Sie ihn noch einmal. Testen Sie ihn von vorne. Testen Sie ihn von hinten. Testen Sie ihn hoch und runter!

Vielleicht machen Sie sich Sorgen darüber, dass das Testen des Codes so viel von Ihrer wertvollen Zeit frisst. Immerhin müssen Sie Zeitpläne befolgen und Termine einhalten. Wenn Sie die ganze Zeit nur testen, kriegen Sie nie irgendwas fertig geschrieben. Gutes Argument! Also sollten Sie Ihre Tests automatisieren. Schreiben Sie Unit-Tests, die Sie ganz kurzfristig ausführen können, und lassen Sie diese Tests so oft wie möglich laufen.

Wie viel von dem Code sollte mit diesen automatisierten Unit-Tests getestet werden? Muss ich diese Frage wirklich beantworten? Der gesamte Code! Der. Gesamte. Code.

Rate ich zu einer hundertprozentigen Testabdeckung? Nein, dazu *rate* ich nicht. Ich *fordere* sie! Jede einzelne Codezeile, die Sie schreiben, sollte getestet werden. Basta!

Ist das nicht unrealistisch? Natürlich nicht. Sie schreiben nur deswegen Code, weil Sie erwarten, dass er ausgeführt wird. Wenn er also Ihrer Erwartung nach ausgeführt werden wird, sollten Sie auch wissen, *dass* er funktioniert. Der einzige Weg, um das herauszubekommen, sind Tests.

Ich bin der Hauptentwickler und Committer für ein Open Source-Projekt namens FitNesse. Während ich dies hier schreibe, gibt es in FitNesse 60.000 Zeilen Quellcode. 26.000 davon enthalten die über 2.000 Unit-Tests. Laut Emma[4] liegt die Abdeckung dieser 2.000 Tests bei etwa 90 %.

Warum ist meine Code-Abdeckung nicht höher? Weil Emma nicht alle Codezeilen sehen kann, die ausgeführt werden! Ich gehe davon aus, dass die Abdeckung deutlich höher ist als 90 %. Beträgt die Abdeckung 100 %? Nein, 100 % ist nur eine Annäherungslinie.

Aber ist mancher Code nicht schwer zu testen? Ja, aber nur deswegen, weil dieser Code so designt wurde, dass er schwer zu testen ist. Die Lösung dafür ist, Ihren Code so designen, dass er *einfach* zu testen ist. Und der beste Weg dazu ist, dass Sie zuerst die Tests schreiben und dann erst den Code, der sie bestehen soll.

Diese Disziplin nennt man eine testgetriebene Entwicklung (Test Driven Development, TDD), über die wir mehr in einem späteren Kapitel erfahren.

4 `http://emma.sourceforge.net/`

Automatisierte Qualitätssicherung

Der gesamte Qualitätssicherungsprozess für FitNesse besteht in der Ausführung der Unit- und Akzeptanztests. Wenn diese Tests erfolgreich absolviert werden, dann liefere ich die Software aus. Das bedeutet, meine Prozesse zur Qualitätssicherung dauern etwa drei Minuten, und ich kann sie spontan ausführen, wann immer ich möchte.

Nun ist es wohl richtig, dass niemand ums Leben kommen wird, falls in FitNesse ein Bug steckt. Auch wird deswegen keiner mehrere Millionen Dollar verlieren. Andererseits hat FitNesse viele Tausend User und eine *sehr* kurze Bug-Liste.

Gewiss gibt es bestimmte Systeme, die so missionskritisch sind, dass ein kurzer automatisierter Test nicht ausreicht, um festzustellen, ob das System für die Auslieferung reif ist. Andererseits brauchen Sie als Entwickler einen schnellen und verlässlichen Mechanismus, um zu wissen, ob der von Ihnen verfasste Code funktioniert und sich nicht mit dem restlichen System verhakelt. Also sollten Ihre automatisierten Tests für Sie zumindest ergeben, dass das System *höchstwahrscheinlich* die Qualitätssicherung bestehen wird.

1.3.2 Beschädige nicht die Struktur

Der wahre Profi weiß, dass man dem Kunden einen Bärendienst erweist, wenn man Funktionen auf Kosten der Struktur abliefert. Es ist die Struktur Ihres Codes, durch die er so flexibel wird. Wenn Sie die Struktur kompromittieren, gefährden Sie seine Zukunft.

Die fundamentale, allen Software-Projekten zugrunde liegende Annahme lautet, dass Software einfach zu ändern ist. Wenn Sie diese Annahme verletzen, indem Sie unflexible Strukturen erstellen, dann untergraben Sie das ökonomische Modell, auf dem die gesamte Branche basiert.

Kurz gesagt: *Sie müssen in der Lage sein, Änderungen ohne exorbitante Kosten vornehmen zu können.*

Bedauerlicherweise versacken allzu viele Projekte in einer Teergrube schlechter Struktur. Aufgaben, für die man normalerweise Tage brauchte, dauern auf einmal Wochen und dann gar Monate. Das Management versucht verzweifelt, den verloren gegangenen Schwung wieder zu gewinnen, und stellt mehr Entwickler ein, um Dampf zu machen. Doch diese Entwickler erweitern den Sumpf nur noch, vertiefen die strukturelle Beschädigung und erhöhen die Hindernisse.

Über die Prinzipien und Patterns des Software-Designs, die flexibel und einfach zu wartende Strukturen unterstützen, ist schon viel geschrieben worden[5]. Profes-

5 [PPP2001]

sionelle Software-Entwickler prägen sich diese Dinge ins Gedächtnis ein und bemühen sich, dazu konforme Software zu entwickeln. Aber es gibt einen Trick dafür, den viel zu wenige Software-Entwickler befolgen: *Wenn Sie wollen, dass die Software flexibel ist, müssen Sie sie auch belasten und ausreizen!*

Der einzige Weg, um zu beweisen, dass Ihre Software einfach zu ändern ist, besteht darin, einfache Änderungen daran vorzunehmen. Und wenn Sie merken, dass die Änderungen nicht so einfach sind wie gedacht, dann überarbeiten Sie das Design, damit die nächste Änderung leichter geht.

Wann sollten Sie solch einfache Änderungen vornehmen? *Die ganze Zeit!* Jedes Mal, wenn Sie sich ein Modul anschauen, nehmen Sie daran kleine, leichte Änderungen vor, um dessen Struktur zu verbessern. Jedes Mal, wenn Sie den Code lesen, passen Sie die Struktur an.

Diese Philosophie wird manchmal *merciless refactoring* (etwa: gnadenlose Umgestaltung) genannt. Ich nenne es die »Pfadfinder-Regel«: Ein Modul sollte immer sauberer wieder eingecheckt werden, als man es zur Bearbeitung entnommen hat. Lassen Sie dem Code stets irgendeine gute Tat angedeihen, sobald Sie mit ihm zu tun bekommen.

Dies ist komplett konträr zu der Weise, wie meistens über Software gedacht wird. Man glaubt, dass eine fortgesetzte Serie von Änderungen bei funktionierender Software *gefährlich* ist. Nein! Was gefährlich ist: wenn man der Software erlaubt, statisch zu bleiben. Wenn Sie sie nicht ausreizen und anpassen, werden Sie merken, wie rigide die Software ist, falls dann doch einmal Änderungen nötig sind.

Warum fürchten die meisten Entwickler sich so davor, ihren Code fortwährend zu verändern? Sie haben Angst, dass er kaputt geht! Warum haben sie Angst, dass er kaputt geht? Weil sie keine Tests machen.

Alles läuft letzten Endes wieder auf Tests hinaus. Wenn Sie eine automatisierte Test-Suite haben, die praktisch 100 % des Codes abdeckt, und wenn diese Test-Suite spontan und mal eben schnell ausgeführt werden kann, *werden Sie einfach keine Angst mehr davor haben, den Code zu ändern*. Wie beweisen Sie, dass Sie keine Angst mehr haben, den Code zu ändern? Sie ändern ihn immer wieder.

Professionelle Entwickler sind sich ihres Codes und der Tests derart sicher, dass sie unfassbar locker damit umgehen, beliebige und anpassungsfähige Änderungen vorzunehmen. Sie ändern spontan den Namen einer Klasse. Wenn sie eine langatmige Methode bemerken, während sie ein Modul lesen, arbeiten sie sie ganz selbstverständlich um. Sie transformieren eine Switch-Anweisung in ein polymorphes Deployment oder wandeln eine Vererbungshierarchie in eine Befehlskette um. Kurz gesagt behandeln sie Software so, wie ein Bildhauer mit Lehm arbeitet: Sie formen und gestalten ständig um.

1.4 Arbeitsethik

Ihre Karriere obliegt *Ihrer* Verantwortung. Es ist nicht Sache Ihres Arbeitgebers, dafür zu sorgen, dass Sie marktfähig sind. Ihr Arbeitgeber ist nicht dafür verantwortlich, Sie zu schulen oder auf Konferenzen zu schicken oder Ihnen Bücher zu kaufen. Dafür sind *Sie selbst* verantwortlich. Wehe dem Software-Entwickler, der seine Karriere seinem Brötchengeber anvertraut.

Manche Arbeitgeber kaufen Ihnen bereitwillig Bücher und andere Schulungsunterlagen oder schicken Sie in Weiterbildungen und auf Konferenzen. Das ist prima, und sie tun Ihnen damit einen Gefallen. Aber tappen Sie niemals in die Falle zu glauben, dass dies zur Verantwortung Ihres Arbeitgebers gehört. Wenn er so etwas nicht für Sie macht, sollten Sie einen Weg finden, es eigenständig umzusetzen.

Es gehört auch nicht in den Verantwortungsbereich Ihres Arbeitgebers, Ihnen Zeit zum Lernen zu geben. Manche Firmen stellen Sie für solche Weiterbildungszeiten frei. Manche verlangen von Ihnen sogar, dass Sie sich die Zeit nehmen. Aber noch einmal: Man tut *Ihnen* damit einen Gefallen, und Sie sollten deswegen angemessen dankbar dafür sein. Solche Vorteile sollten für Sie nicht selbstverständlich sein.

Sie schulden Ihrem Arbeitgeber Zeit und Arbeitsaufwand in einem bestimmten Umfang. Für das Beispiel hier gehen wir einmal vom US-Standard von 40 wöchentlichen Arbeitsstunden aus. Diese 40 Stunden sollten Sie mit den Problemen *Ihres Arbeitgebers* verbringen und nicht mit *Ihren eigenen*.

Sie sollten einplanen, 60 Stunden die Woche zu arbeiten. Die ersten 40 gehören Ihrem Arbeitgeber. Die restlichen 20 gehören Ihnen. Während dieser 20 Stunden sollten Sie lesen, üben, lernen und Ihre Karriere anderweitig ausbauen.

Ich höre Sie schon denken: »Aber was ist mit meiner Familie? Was ist mit meinem Privatleben? Muss ich das alles meinem Arbeitgeber opfern?«

Ich spreche hier nicht von Ihrer *gesamten* Freizeit. Ich spreche von 20 Extrastunden pro Woche. Das sind etwa drei Stunden täglich. Wenn Sie die Mittagspause für Lektüre nutzen, sich auf dem Weg zur Arbeit und dem Heimweg Podcasts anhören und 90 Minuten täglich eine neue Programmiersprache lernen, haben Sie die drei Stunden bereits voll und das Pensum abgedeckt.

Rechnen Sie sich das einmal durch: Die Woche hat 168 Stunden. Geben Sie Ihrem Arbeitgeber davon 40 und Ihrer Karriere weitere 20. Dann bleiben noch 108. Wenn Sie 56 Stunden davon schlafen, bleiben Ihnen für alles andere noch 52.

Vielleicht wollen Sie ein solches Commitment nicht eingehen. Das ist okay, aber dann sollten Sie sich selbst nicht als Profi betrachten. Profis wenden viel Zeit dafür auf, sich um ihre Profession zu kümmern.

Vielleicht glauben Sie, dass man die Arbeit auf der Arbeit lassen und nicht mit nach Hause nehmen sollte. Dem stimme ich zu! Sie sollten in diesen 20 Stunden

nicht für Ihren Arbeitgeber tätig sein. Stattdessen sollten Sie an Ihrer Karriere arbeiten.

Manchmal ist beides deckungsgleich. Manchmal ist die Arbeit für Ihre Firma auch für Ihre Karriere sehr vorteilhaft. In diesem Fall ist es vernünftig, von diesen 20 Stunden etwas dafür aufzuwenden. Aber denken Sie immer daran: Diese 20 Stunden sind *für Sie*. Sie sollten dazu genutzt werden, Sie als Profi noch wertvoller zu machen.

Vielleicht glauben Sie, dass man mit diesem Rezept im Burnout landet. Im Gegenteil: Es ist ein Rezept, um ein Burnout zu *vermeiden*. Wahrscheinlich sind Sie Entwickler geworden, weil Sie sich leidenschaftlich für Software interessieren und Ihr Wunsch, Profi zu sein, von dieser Leidenschaft motiviert wird. Während dieser 20 Stunden sollten Sie jene Dinge machen, die diese Leidenschaft noch *verstärken*. Diese 20 Stunden sollten *Spaß* machen!

1.4.1 Sie sollten sich in Ihrem Bereich auskennen

Wissen Sie, was ein Nassi-Shneiderman-Diagramm ist? Falls nicht, wieso nicht? Kennen Sie den Unterschied zwischen einem Mealy- und einem Moore-Automaten? Das sollten Sie. Können Sie einen Quicksort schreiben, ohne irgendwo nachzuschauen? Wissen Sie, was mit dem Begriff »Ablaufdiagramm« gemeint ist? Können Sie mit Datenflussdiagrammen eine funktionale Zerlegung vornehmen? Was bedeutet der Ausdruck »Vagabundierende Daten«? Haben Sie das Wort »Connascence« schon mal gehört? Was ist eine Parnas-Tabelle?

Ein ganzes Füllhorn voller Ideen, Disziplinen, Techniken, Tools und Terminologien schmückt die letzten 50 Jahre unserer Fachrichtung. Wie viel davon kennen Sie? Wenn Sie ein Profi sein wollen, sollten Sie einen erheblichen Teil davon kennen und nicht nachlassen, Ihr Wissensgebiet ständig auszubauen.

Warum sollten Sie all diese Dinge wissen? Entwickelt sich unser Arbeitsfeld nicht dermaßen schnell weiter, dass all diese alten Ideen und Konzepte irrelevant werden? Der erste Teil dieser Frage erscheint oberflächlich offensichtlich. Natürlich erweitert sich unser Arbeitsfeld deutlich, und das in einem rasanten Tempo. Interessanterweise ist dieser Fortschritt in vielerlei Hinsicht peripher. Es stimmt, dass wir keine 24 Stunden mehr auf den Abschluss einer Kompilierung warten müssen. Es trifft zu, dass wir Systeme schreiben, deren Größe sich im Gigabyte-Bereich bewegt. Es ist richtig, dass wir inmitten eines weltumspannenden Netzwerks arbeiten, über das man sofort auf alle möglichen Informationen zugreifen kann. Andererseits schreiben wir die gleichen `if`- und `while`-Anweisungen wie noch vor 50 Jahren. Viel hat sich geändert. Doch vieles eben auch nicht.

Der zweite Teil der Frage stimmt ganz gewiss nicht. Nur sehr wenige Ideen aus den vergangenen 50 Jahren sind irrelevant geworden. Sicher, manche sind auf dem Abstellgleis gelandet. Die Entwicklung nach dem Wasserfallmodell ist sicher-

lich in Ungnade gefallen. Doch das bedeutet nicht, dass wir es uns leisten können, es nicht zu kennen, und welche guten und schlechten Aspekte es mit sich bringt.

Insgesamt jedoch ist die große Mehrheit der hart erarbeiteten Ideen aus den vergangenen 50 Jahren heute so wertvoll wie damals. Vielleicht sind sie sogar noch wertvoller.

Denken Sie an Santayanas Fluch: »Wer sich seiner Vergangenheit nicht erinnert, ist dazu verurteilt, sie zu wiederholen.«

Hier ist eine minimale Liste der Dinge, die jedem Software-Profi vertraut sein sollten:

- **Design Patterns.** Sie sollten in der Lage sein, alle 24 Entwurfsmuster aus dem Buch der Viererbande (*GOF Book*) zu beschreiben, und über praktische Erfahrungen mit möglichst vielen Entwurfsmustern aus den POSA-Büchern verfügen.
- **Design-Prinzipien.** Sie sollten die SOLID-Prinzipien kennen und sich die Komponentenprinzipien gründlich angeeignet haben.
- **Methoden.** Sie sollten die Methoden XP, Scrum, Lean, Kanban, Wasserfall, Strukturierte Analyse und Strukturiertes Design verstanden haben.
- **Disziplinen.** Sie sollten TDD, Objektorientiertes Design, Strukturierte Programmierung, Kontinuierliche Integration und Paarprogrammierung praktizieren.
- **Artefakte:** Sie sollten wissen, wie man mit den folgenden Dingen arbeitet: UML, DFDs, strukturierte Diagramme, Petri-Netze, Zustandsübergangsdiagramme und -tabellen, Flussdiagramme und Entscheidungstabellen.

1.4.2 Lebenslanges Lernen

In unserer Branche ist die Veränderungsrate derartig fieberhaft, dass Software-Entwickler dauerhaft große Mengen Stoff lernen müssen, einfach nur um aktuell zu bleiben. Wehe den Architekten, die mit dem Programmieren aufhören: Schnell werden sie merken, dass sie irrelevant sind. Wehe den Programmierer, die aufhören, neue Sprachen zu lernen: Sie dürfen zuschauen, wie die Branche sie rechts überholt. Wehe den Entwicklern, denen es nicht gelingt, neue Disziplinen und Techniken zu lernen: Sie werden von ihren Kollegen ausgestochen und in Bedeutungslosigkeit versinken.

Würden Sie einen Arzt konsultieren, der sich nicht durch medizinische Fachjournale in seinem Metier auf dem Laufenden hält? Würden Sie einen Steuerberater engagieren, der sich mit aktuellen Steuergesetzen und Präzedenzfällen nicht auskennt? Warum sollten Firmen Entwickler einstellen, die nicht dafür sorgen, dass ihr Wissenstand aktuell ist?

Lesen Sie Bücher, Artikel, Blogs und Tweets. Besuchen Sie Konferenzen und Tagungen. Gehen Sie zu User-Gruppen. Nehmen Sie an Lesezirkeln und Studiengruppen teil. Lernen Sie Dinge, die sich außerhalb Ihrer Komfortzone befinden. Wenn Sie ein .NET-Programmierer sind, lernen Sie Java. Wenn Sie Java-Programmierer sind, lernen Sie Ruby. Sind Sie C-Programmierer sind, lernen Sie Lisp. Wenn Sie Ihr Hirn wirklich auf Trab halten wollen, lernen Sie Prolog und Forth!

1.4.3 Praxis

Profis üben. Echte Profis arbeiten hart daran, ihre Fähigkeiten aktuell und einsatzbereit zu halten. Es reicht nicht, einfach den täglichen Job durchzuziehen und das dann Praxis zu nennen. Wenn man den Alltagsjob erledigt, gehört das zur Arbeitsleistung, kann aber nicht als Übung und Praxis bezeichnet werden. Praxis ist, wenn Sie Ihre Skills explizit *außerhalb* der normalen Jobanforderungen ausüben, einzig und allein dafür, um diese Skills zu verfeinern und zu erweitern.

Was könnte Praxis für einen Software-Entwickler möglicherweise bedeuten? Beim ersten Gedanken scheint dieses Konzept absurd zu sein. Aber denken Sie doch mal etwas weiter. Überlegen Sie, wie Musiker ihre Kunst meistern. Das machen sie nicht durch Auftritte, sondern durch Üben. Und wie üben sie? Neben anderen Dingen haben sie spezielle Übungen, die sie ausführen. Tonleitern und Etüden und bestimmte Tonfolgen. Das üben sie immer wieder und wieder, um ihre Finger und ihren Geist zu schulen und sich bei ihren Fähigkeiten immer stets die Meisterschaft zu bewahren.

Wie könnten also Software-Entwickler für ihre Praxis üben? Dieses Buch enthält ein ganzes Kapitel, das sich den verschiedenen Übungstechniken widmet. Also werde ich hier nicht ins Detail gehen. Eine Technik, die ich häufig einsetze, ist die Wiederholung einfacher Übungen wie das *Bowling Game* oder *Primfaktoren*. Ich nenne diese Übungen Kata. Es gibt viele Kata, unter denen man wählen kann.

Ein Kata erscheint normalerweise als einfaches Programmierproblem, das gelöst werden muss, z.B. die Funktion zu schreiben, mit der die Primfaktoren einer Ganzzahl berechnet werden können. Das Kata sollten Sie nun nicht deswegen machen, um die Lösung des Problems herauszufinden, denn wie das geht, wissen Sie bereits. Das Wichtige beim Kata ist, die eigenen Finger und das eigene Gehirn zu trainieren.

Ich mache jeden Tag ein oder zwei Kata. Das gehört für mich oft dazu, mich auf die Arbeit einzustimmen. Ich mache sie dann entweder in Java oder Ruby oder Clojure oder in einer anderen Sprache, in der ich meine Skills schulen will. Ich nutze das Kata, um eine bestimmte Fähigkeit zu schärfen, um z.B. meine Finger wieder daran zu gewöhnen, bestimmte Tastaturkürzel zu treffen oder mit bestimmten Refakturierungen zu arbeiten.

Stellen Sie sich das Kata vor wie eine zehnminütige Aufwärmübung morgens und eine zehnminütige Übung zum Abschalten abends.

1.4.4 Teamwork

Der zweitbeste Weg zum Lernen ist, mit anderen zusammenzuarbeiten. Professionelle Software-Entwickler geben sich besondere Mühe, gemeinsam zu programmieren und zusammen zu üben, zu designen und zu planen. Auf diese Weise lernen sie eine Menge voneinander und bekommen mit weniger Fehlern mehr geschafft.

Das bedeutet nicht, dass Sie 100 % Ihrer Zeit mit anderen zusammenarbeiten sollen. Die Zeit für sich alleine ist ebenfalls sehr wichtig. So sehr ich das Pair Programming auch liebe, macht es mich doch verrückt, wenn ich nicht immer wieder mal auf mich allein gestellt arbeiten kann.

1.4.5 Mentorenarbeit

Der beste Weg, etwas zu lernen, ist das Lehren. Nichts bringt Fakten und Werte schneller und dauerhafter in den eigenen Kopf als sie jenen Personen zu vermitteln, für die Sie verantwortlich sind. Also hat der Lehrer eindeutig die Vorteile des Lehren3s auf seiner Seite.

Ebenso gibt es keinen besseren Weg, um neue Leute in eine Organisation zu bringen, als sich mit ihnen hinzusetzen und sie einzuarbeiten. Profis fühlen sich für die Betreuung von Jüngeren persönlich verantwortlich. Sie lassen nicht zu, dass sich ein solcher Junior unbeaufsichtigt abstrampelt.

1.4.6 Sie sollten sich in Ihrem Arbeitsgebiet auskennen

Es obliegt der Verantwortung eines jeden Software-Profis, sich im jeweiligen Arbeitsgebiet auszukennen, für das er programmiert. Wenn Sie ein Abrechnungssystem schreiben, sollten Sie über die Modalitäten der Buchführung Bescheid wissen. Wenn Sie eine Reise-Applikation schreiben, sollten Sie sich in der Reisebranche auskennen. Sie müssen kein Experte in dieser Domäne sein, aber es gehört zu Ihrer Sorgfaltspflicht, sich kenntnisreich in ein Gebiet einzuarbeiten.

Wenn Sie ein Projekt in einer neuen Domäne beginnen, sollten Sie sich ein oder zwei themenbezogene Bücher zu Gemüte führen. Sprechen Sie mit Ihren Kunden und Anwendern über die Basis und Grundprinzipien dieser Domäne. Verbringen Sie Zeit mit Experten und versuchen Sie, deren Prinzipien und Werte zu verstehen.

Es ist die schlimmste Art unprofessionellen Verhaltens, wenn man einfach nach der Spezifikation programmiert, ohne zu begreifen, wie und warum diese Spezifikation für das jeweilige Business relevant ist. Stattdessen sollten Sie sich in der Domäne so gut auskennen, um die Spezifikation kritisch zu hinterfragen und Fehler darin erkennen zu können.

1.4.7 Identifizieren Sie sich mit Ihrem Arbeitgeber bzw. Kunden

Die Probleme Ihres Arbeitgebers sind *Ihre* Probleme. Sie müssen verstehen, worum es bei diesen Problemen geht, und auf die besten Lösungsmöglichkeiten hinarbeiten. Wenn Sie ein System entwickeln, müssen Sie buchstäblich in die Haut Ihres Arbeitgebers schlüpfen und darauf achten, dass die von Ihnen entwickelten Features sich wirklich um die Ansprüche und Bedürfnisse Ihres Arbeitgebers kümmern.

Entwickler identifizieren sich leicht miteinander. Man fällt beim eigenen Arbeitgeber ganz leicht in eine Haltung des *wir gegen sie*. Profis vermeiden so etwas unter allen Umständen.

1.4.8 Bescheidenheit

Programmieren ist ein schöpferischer Akt. Wenn wir Code schreiben, schaffen wir etwas aus dem Nichts. Wir führen mutig Ordnung ins Chaos ein. Wir befehligen selbstbewusst auf präzise und detaillierte Weise das Verhalten einer Maschine, die anderenfalls unkalkulierbare Schäden anrichten kann. Und somit ist Programmieren ein Akt höchster Arroganz.

Profis wissen, dass sie arrogant sind, und stellen keine falsche Bescheidenheit zur Schau. Ein Profi kennt seinen Auftrag und ist stolz auf seine Arbeit. Ein Profi vertraut seinen Fähigkeiten und geht aufgrund dieses Selbstvertrauens mutige und kalkulierte Risiken ein. Ein Profi ist nicht furchtsam.

Aber ein Profi weiß auch, dass er manchmal versagen wird, dass seine Risikoberechnungen falsch sind, dass seine Fähigkeiten nicht ausreichen. Dann blickt er in den Spiegel und sieht, wie ihn ein arroganter Narr anlächelt.

Wenn also ein Profi merkt, dass er Zielscheibe des Spotts ist, wird er als Erster lachen. Er wird niemals andere lächerlich machen, sondern den Spott akzeptieren, wenn er berechtigt ist, und einfach darüber schmunzeln, wenn das nicht der Fall ist. Er erniedrigt niemanden, weil der einen Fehler gemacht hat, denn er weiß, dass er vielleicht der Nächste ist, der einen Bock schießt.

Ein Profi weiß um seine eigene Ober-Arroganz und dass das Schicksal sich irgendwann gegen ihn richten und ihn zurechtstutzen wird. Wenn Sie also in die Schusslinie geraten, ist das Beste, was Sie machen können, Howards Ratschlag umzusetzen: Lachen Sie.

1.5 Bibliografie

[PPP2001]: Robert C. Martin, *Principles, Patterns, and Practices of Agile Software Development*, Upper Saddle River, NJ: Prentice Hall, 2002.