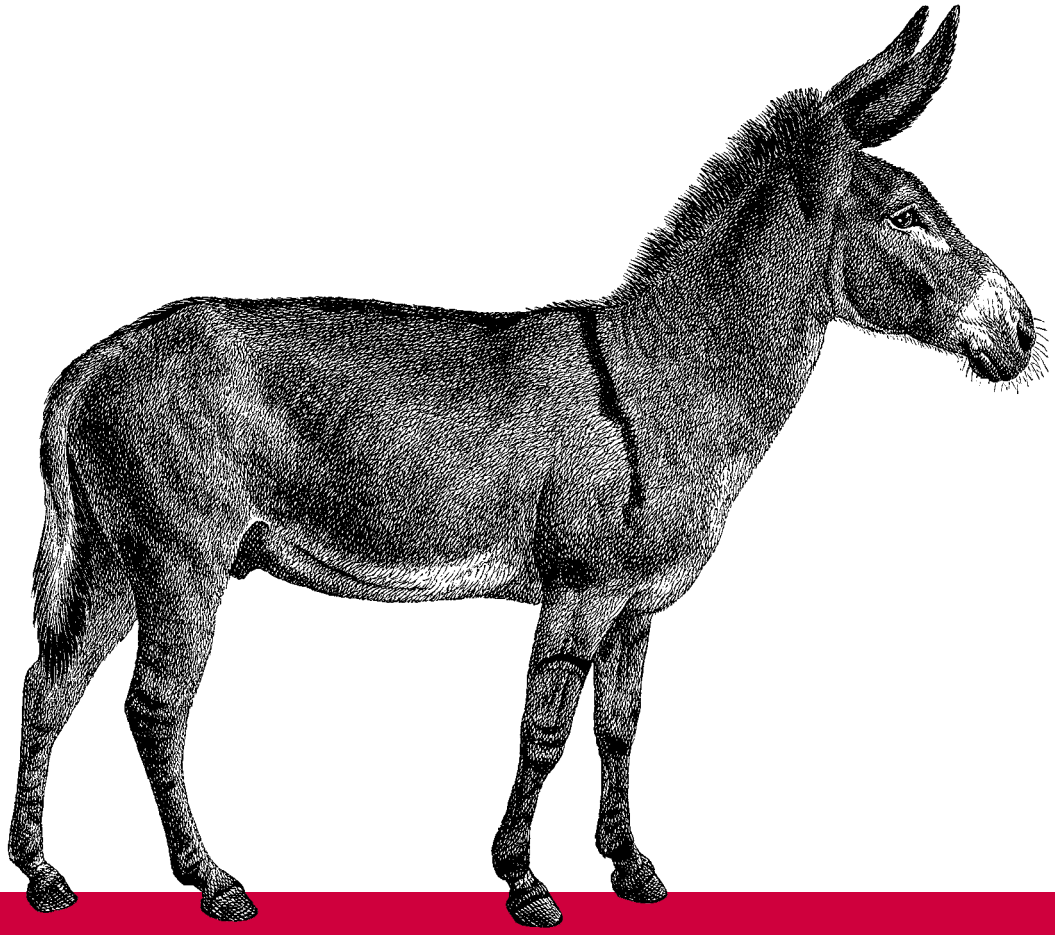
O'REILLY®



Weniger schlecht programmieren

Kathrin Passig &
Johannes Jander

Weniger schlecht programmieren

Kathrin Passig & Johannes Jander

O'REILLY®
Beijing · Cambridge · Farnham · Köln · Sebastopol · Tokyo

Kommentare und Fragen können Sie gerne an uns richten:
O'Reilly Verlag
Balthasarstr. 81
50670 Köln
E-Mail: kommentar@oreilly.de

1. Auflage 2013

Bibliografische Information der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.d-nb.de* abrufbar.

Lektorat: Inken Kiupel & Christine Haite, Köln
Fachliche Unterstützung: Dirk Gómez, München,
Jörg Staudemeyer, Berlin & Christian Trabold, Hamburg
Korrektorat: Eike Nitz, Köln
Satz: III-satz, Husby, www.drei-satz.de
Produktion: Karin Driesen, Köln
Belichtung, Druck und buchbinderische Verarbeitung:
Druckerei Kösel, Krugzell; www.koeselbuch.de

ISBN 978-3-89721-567-2

Dieses Buch ist auf 100% chlorfrei gebleichtem Papier gedruckt.

Inhalt

Teil 3: Umgang mit Fehlern

Teil 4: Wahl der Mittel

Vorwort

> »Von ungezählten Dingen weiß man, dass man sie nicht weiß. Die Anzahl der Dinge, von denen man weiß, dass man sie nicht weiß, soll mal als verwickeltes Beispiel dienen. Das ist nicht weiter schlimm, denn wenn man weiß, dass man etwas nicht weiß, fragt man einfach Google, fertig. Schwieriger wird's, wenn man nicht weiß, was man nicht weiß. Diese Sorte Nichtwissen ist leider eine offene Tür für heftige Überraschungen.«
>
> Kai Schreiber, »Riesenmaschine – Das brandneue Universum«

»Wir sollten alle danach streben, bessere Programmierer zu werden; wenn Ihnen dieser Ehrgeiz fehlt, ist dieses Buch nichts für Sie«, heißt es in der Einleitung zu Pete Goodliffes »Code Craft«. Unser Buch hingegen gibt es, weil es vielleicht genau dieser Ehrgeiz ist, der Ihnen fehlt. Beziehungsweise fehlt er Ihnen vermutlich nicht einmal. Sie haben andere Prioritäten, und ein besserer Programmierer zu werden, kommt auf der Liste Ihrer Ziele im Leben frühestens auf Platz 8. Wenn Sie sich trotzdem gern etwas weniger häufig ins Knie schießen würden als bisher, dann sind Sie hier richtig.

Sie möchten Ihre Programmierkenntnisse pragmatisch ausbauen, haben aber nicht das Bedürfnis, gleich als Referenz für guten Programmierstil zu gelten. Sie haben nicht vor, Softwarearchitekt oder Gruppenleiter für Software in einer Firma zu werden, sondern wollen ein paar alltägliche Probleme lösen, ohne jemanden dafür bezahlen zu müssen.

Eventuell sind Sie weit davon entfernt, sich »Programmierer« zu nennen. Weil Programmierung keine Geheimwissenschaft für Spezialisten mehr ist, haben Sie ein bisschen damit herumgespielt, ob nun zu Ihrem Privatvergnügen oder um sich eine ganz andere Arbeit zu erleichtern. Sie haben das eine oder andere Programm geschrieben und beherrschen mindestens eine Programmiersprache so lala. Für Probleme finden Sie Lösungen, haben aber das Gefühl, dass das bestimmt alles irgendwie besser ginge. Nachträglichen Anpassungen gehen Sie so lange wie möglich aus dem Weg, weil es Ihnen schwer fällt, Ihren eigenen Code aus dem Vorjahr zu verstehen. Sie fragen sich, ob das wohl allen so geht.

Oder aber Sie halten sich trotz geringer Erfahrung für einen ziemlich begabten Programmierer, der nur gelegentlich auf kleine Schwierigkeiten stößt. Das ist das Stadium der

»unbewussten Inkompetenz«, und laut den Interviews, die wir für dieses Buch geführt haben, kann dieser Zustand zehn bis fünfzehn Jahre lang anhalten:

> »Ich habe von den 19 Jahren, die ich programmiere, sicher 13 Jahre sehr schlecht programmiert. Es mussten erst mal Probleme einer bestimmten Größe entstehen, die mich dann zum Umdenken gezwungen haben. Wenn man anfängt, Sachen für andere zu machen, und dann immer noch schlecht ist oder die Abkürzungen nimmt, die man sich halt so angewöhnt hat, dann kann es sein, dass die Sachen sehr schnell zusammenbrechen. Weil die Leute die Software ganz anders benutzen, weil sie die ganzen Fehler nicht kennen und sie auch nicht automatisch umgehen. Da bin ich durch sehr sauren Regen gegangen und hatte viele schreiende Kunden am Telefon. Ich wusste nicht, dass ich ein schlechter Programmierer bin. Ich dachte, was ich kann, reicht völlig aus. Aber der persönliche Stress mit den Leuten ist mir auf die Nerven gegangen. Ein Projekt ist krass gescheitert, das war für mich eine persönliche Niederlage, da dachte ich: So geht's nicht weiter, wie kann man das besser machen? Danach ging's dann schnell bergauf.«
>
> Lukas Hartmann, Softwareentwickler

In der Entwicklungspsychologie gibt es seit den 1970ern ein Modell für die Entwicklung von Verständnis und Fähigkeiten in komplexen Wissensgebieten, die »Vier Stufen der Kompetenzentwicklung«[1]. Auch wenn diese Abstufung wissenschaftlich nur mäßig fundiert ist, illustriert sie ganz gut die Entwicklung eines Programmierers vom Anfänger zum Spezialisten.

Die erste Stufe ist die unbewusste Inkompetenz, bei der man weder die Konzepte des Wissensgebietes kennt noch weiß, dass man Defizite hat. Auf die unbewusste Inkompetenz folgt die bewusste Inkompetenz: Man hat zwar immer noch große Schwierigkeiten, die anfallenden Probleme zu lösen, ist sich aber dessen bewusst. Möglicherweise haben Sie diesen Schritt schon hinter sich; schließlich hätten Sie sonst keinen Grund, dieses Buch zu lesen. Vielleicht haben Sie Ihren Code anderen Menschen gezeigt und wurden ausgelacht, vielleicht haben Sie einmal zu oft versehentlich die projektentscheidende Datenbank gelöscht, oder vielleicht sind Sie einfach von Natur aus ein bescheidener Mensch. Auf die beiden Inkompetenzstufen wiederum folgt die »bewusste Kompetenz«: Man kann alles Nötige, muss sich aber noch stark auf die richtige Ausführung konzentrieren. Ganz zum Schluss erreicht man die Stufe der »unbewussten Kompetenz«. Jetzt läuft alles wie von allein. Nach der Lektüre dieses Buches sollten Sie sich in der bewussten Inkompetenz ganz zu Hause fühlen und hin und wieder einen Anflug bewusster Kompetenz verspüren.

Insgeheim halten Sie sich womöglich für ein bisschen langsamer, dümmer oder technisch untalentierter als andere Menschen. In Wirklichkeit aber ist die Welt voll mit hauptberuflichen Programmierern, die Schwierigkeiten haben, eine einfache Bierdeckelrechnung im Kopf durchzuführen. Ein schlechter Programmierer ist nicht schlecht, weil er noch nicht lange programmiert, weil sein IQ nicht hoch genug ist, weil er ein schlechtes Gedächtnis hat, weil er Autodidakt ist oder weil er erst spät im Leben mit dem Program-

1 Siehe *en.wikipedia.org/wiki/Four_stages_of_competence*.

mieren angefangen hat. All diese Faktoren spielen entweder keine große Rolle oder können einem, richtig eingesetzt, sogar zum Vorteil gereichen (dazu später mehr). Ein schlechter Programmierer ist man vor allem, weil man schlecht programmiert.

Warum dauert es so lange, bis der Mensch klüger wird?

Wahrscheinlich wissen Sie über viele Probleme Ihres Codes ganz gut Bescheid. Vielleicht haben Sie sogar eine Vorstellung davon, was Sie in Zukunft tun oder unterlassen müssten, um ein besserer Programmierer zu werden. »Es ist keine Schande, ein schlechter oder mittelmäßiger Programmierer zu sein«, schreibt Steve McConnell in »Code Complete«, »die Frage ist nur, wie lange man schlecht oder mittelmäßig bleibt, nachdem man erkannt hat, wie es besser ginge«[2]. Aber warum ist es oft so schwer, den Schritt von der Einsicht zur Problembehebung zu tun?

Die meisten Ursachen für die Beharrlichkeit, mit der wir auf falsche Lösungswege setzen, liegen nicht in der Natur des schlechten Programmierers, sondern in der des Menschen. An erster Stelle steht dabei der Konservatismus. Gegen den Wunsch, einfach immer so weiterzumachen wie bisher, ist erst einmal nichts einzuwenden. Das Gehirn muss mit seinen Kräften haushalten und tut deshalb gut daran, eine akzeptable Lösung nicht gleich wieder über Bord zu werfen, nur weil am Horizont irgendetwas anderes auftaucht. Und allen relevanten Entwicklungen neuer Technologien, Sprachen, Methoden und Frameworks zu folgen, ist eine so aufwendige Beschäftigung, dass kaum Zeit für vielleicht noch unterhaltsamere Aspekte des Lebens bleibt.

Während Kinder und Jugendliche große Teile des Tages damit verbringen, Neues herauszufinden, kann es vorkommen, dass man als erwachsener Mensch über lange Zeiträume fast gar nichts dazulernt. »Ach, das ging doch bisher auch so«, sagt sich der schlechte Programmierer, weil er keine Lust hat, sich eine bessere Welt vorzustellen, in der es weniger als einen Tag dauert, alle Kleinbuchstaben eines Textes in Großbuchstaben zu verwandeln – inklusive der Umlaute. Denn wir haben es nicht nur mit dem Hang zu tun, bei einer bewährten Lösung zu bleiben, sondern auch mit dem nachvollziehbaren Wunsch, möglichst wenig nachzudenken. Wer ohne großes Nachdenken vor sich hinbastelt, kommt zwar nur langsam ans Ziel, erzeugt aber mit jedem Schritt sichtbare Ergebnisse. Denkt man über ein Problem erst einmal nach, wird man es zwar am Ende in einem Bruchteil der Zeit lösen, muss aber damit leben, in den ersten Stunden, Tagen oder Wochen überhaupt nichts Vorzeigbares zu produzieren. Diese Unlust, ins Dazulernen zu investieren, fällt unter Trägheit – obwohl das resultierende Verhalten paradoxerweise nach Fleiß und Arbeit aussieht. Der nachdenkliche Programmierer ähnelt währenddessen über lange Zeiträume verdächtig einem Menschen, der auf dem Balkon sitzt und mit glasigem Blick in die Wolken starrt oder tagelang das Internet durchliest.

2 »Code Complete«, S. 825.

Ein Teil der Unwilligkeit, dazuzulernen, entspringt auch aus Angst: Angst vor dem Neuen, Angst vor dem Unbekannten, Angst vor dem Komplizierteren, Angst vor der Dequalifizierung. Wer das bisherige Framework, die bisherige Programmiersprache weiterverwendet, sieht vor sich selbst und anderen wenigstens halbwegs kompetent aus. Auf einem neuen Gebiet wäre man plötzlich wieder Anfänger. Im Vergleich zur Angst, in einer unbeleuchteten Höhle von Zombies aufgefressen zu werden, handelt es sich dabei zwar um eine sehr überschaubare Besorgnis, aber oft reicht auch sie schon aus, um uns von der Beschäftigung mit dem Unbekannten abzuhalten.

Spezialprobleme schlechter Programmierer

Schlechte Programmierer haben wie alle Menschen mit diesen Hindernissen beim Klügerwerden zu kämpfen. Aber für sie kommen noch ein paar eigene Probleme hinzu. Zunächst einmal überfordern sie sich häufig. Wenn man erst als Erwachsener mit dem Programmieren anfängt – und nicht wie viele hauptberufliche Programmierer im Alter von sieben Jahren –, wird man sich nicht damit zufriedengeben, ein Klötzchen auf dem Bildschirm hin- und herspringen zu lassen. Man hat wahrscheinlich ein konkretes Erwachsenenproblem vor Augen, das man lösen will, zum Beispiel Börsenkurse nach Auffälligkeiten zu durchforsten. Und man wird mit seinen beschränkten Fähigkeiten dabei das interessanteste Unheil anrichten.

Nebenbei-Programmierer neigen außerdem dazu, ihren Code vorsichtshalber niemandem zu zeigen. Dadurch entfällt nicht nur das motivierende Element Scham bzw. Angeberei, sondern auch eine der einfachsten Möglichkeiten, dazuzulernen, nämlich das Lernen von besseren Programmierern. Häufig ist eine kurze Erklärung eines erfahrenen Programmierers mehr wert als wochenlanges Nachlesen – nach Konzepten, die man nicht kennt, kann man nicht suchen. Aber jemand, der sie schon kennt, weiß möglicherweise, an welchen Stellen sie dem Anfänger das Leben erleichtern könnten, und kann sie ihm dann mitsamt ihren Vorteilen für die konkrete Fragestellung erklären.

Ein weiterer Punkt ist die Furcht vor dem eigenen Code: Einerseits erkennt man die Notwendigkeit, den Code neu zu strukturieren, um ihn verständlicher zu machen. Aber es dominiert die Angst, aus undurchschaubarem Code, der die richtige Lösung liefert, durchschaubaren Code zu machen, der aber womöglich ein falsches Ergebnis produziert. Dass diese Angst durchaus berechtigt sein kann, spiegelt sich in der Maxime »Never change a running system« wider. Wenn das System aber gar nicht so funktioniert, wie man es gerne hätte, wird aus der pragmatischen Ruhe eine problematische Scheu vor dem Code.

Schließlich ist auch die Vorstellung »Ach, das geht mich alles nichts an, ich programmiere ja nur so zum Spaß« der Weiterentwicklung nicht dienlich. Dahinter steckt die Vorstellung, das Dazulernen oder das bessere Programmieren mache weniger Spaß als das von Wissen unbelastete Herumbasteln. Dabei stimmt das gar nicht. Unerfreulich ist lediglich der Moment der Einsicht, dass man etwas falsch macht. Im schlimmsten Fall

folgt darauf noch etwas geistige Gymnastik bei der Einarbeitung in eine neue Technik. Die dabei investierte Zeit spart man an anderer Stelle mehrfach wieder ein – und zwar dann, wenn man nicht mehr ganze Nächte damit zubringt, den unverständlich gewordenen eigenen Code zu entwirren. Was nämlich, wenn man ehrlich ist, so viel Spaß auch wieder nicht macht.

Die sieben gebräuchlichsten Argumente schlechter Programmierer

Den Code sieht ja eh niemand außer mir.

- Schon morgen kann man jemandem begegnen, der sich nichts dringender wünscht, als an diesem herrlichen Projekt mitzuarbeiten. Bonuspunkte, wenn es sich dabei um jemanden handelt, der kein Deutsch kann, der Code aber voller deutscher Variablennamen und Kommentare ist.
- Aus »Den Code sieht niemand außer mir« wird schnell »Den Code darf niemand außer mir sehen«. Das führt dazu, dass man sich unnötig an Projekte klammert, mit denen man eigentlich überfordert ist oder die einen längst langweilen. Nach Jahren der guten Codeüberarbeitungsvorsätze verlässt man dann den Arbeitsplatz oder das Projekt schließlich doch einfach so. Die unbetreute und unbetreubare Software stirbt einen sinnlosen Tod.
- Dass man selbst den eigenen Code sehen muss – und zwar Jahre später, als klüger gewordener Mensch –, ist schlimm genug.

Die Software benutzt ja eh niemand außer mir.

- Schon kurze Zeit später wird man diesen Gedanken vergessen haben; man wird aber auch vergessen, die nötigen Sicherheitsfunktionen nachzurüsten.
- Auch Sie selbst werden demnächst ein anderer Mensch sein, der sich gar nicht mehr daran erinnert, was bei der Nutzung dieser Software alles zu beachten war.

Später mach ich das alles noch mal ordentlich.

- Eine Grundregel des Universums: Provisorien halten am längsten. So ist praktisch garantiert, dass man ausgerechnet den Code, den man am gedankenlosesten hingeschludert hat, bis ins hohe Alter wiederverwenden wird.
- In 90 Prozent aller Fälle kommt dieses »Später« nie, weil erstens neue Projekte auftauchen, die die ganze Zeit und Aufmerksamkeit in Anspruch nehmen, zweitens niemand Lust hat, gammeligen Code anzufassen, und man drittens gerne verdrängt, wie schlecht der Code ist. Manchmal fängt bereits vor diesem »Später« total überraschend ein neues Jahrtausend an und stellt unerwartet hohe Speicherplatzforderungen für Jahreszahlen, die sich auf die Schnelle nur schwer erfüllen lassen.

Das ist halt ein ganz kompliziertes Problem, da geht es nicht anders, ich muss acht ineinandergeschachtelte Schleifen verwenden.

- Auch bei einem komplizierten Problem sollte niemand acht ineinandergeschachtelte Schleifen verwenden.
- Gerade bei einem komplizierten Problem sollte niemand acht ineinandergeschachtelte Schleifen verwenden.
- Bei genauerem Hinsehen ist ein komplexes Problem lediglich ein Bündel mehrerer mittelkomplexer Probleme, und die wiederum bestehen aus wenig komplexen Problemen, die sich aus trivialen Problemen zusammensetzen. Man braucht also nur eine große Anzahl simpler Funktionen zu schreiben (noch besser: bereits von anderen Menschen geschriebene zu benutzen), und schon ist das komplexe Problem gelöst.

Ich merke mir einfach, dass ich bestimmte Eingaben nicht machen darf.

- Nein, tun Sie nicht. Gerade wenn es hektisch wird, Sie überraschend gebeten werden, der Weltpresse Ihre Erfindung zu präsentieren, oder Sie ein anderes, kompliziertes Problem debuggen, werden Sie genau die Eingaben machen, die Ihre Datenbank in Stücke fallen lassen.
- Andere Programmierer, die neuerdings an Ihrem Projekt mitarbeiten (s. o.), können nicht erahnen, dass bestimmte Eingaben tabu sind.

Ich denke schon dran, das rechtzeitig wieder auszukommentieren.

- Dieser feste Vorsatz hat schon zu romantischen Debug-Nächten im Scheine des Monitors geführt, als Sie noch gar nicht geboren waren. Warum sollte es Ihnen anders gehen? Ein Programmierprojekt im Kopf zu halten, ist schon schwer genug, wenn man sich nicht merken muss, dass irgendwo noch ein paar entsicherte Handgranaten rumliegen.

Es ist ja nur ein ganz kleines Projekt.

- Projekte haben mehrere Dimensionen. Code, der nur ein eng umrissenes Problem lösen soll, bleibt oft über viele Jahre hinweg im Einsatz. Das Projekt ist dann zwar nicht groß, aber lang.
- Am Anfang ist jedes Projekt klein. Bei schlechten Programmierern sorgt die Einstellung »Es ist ja nur ein ganz kleines Projekt« ganz von allein dafür, dass das Projekt niemals groß werden kann. Es erstickt lange vorher an seiner eigenen Unübersichtlichkeit.

Wenige Jahre später

Was braucht man also, um ein weniger schlechter Programmierer zu werden? Nicht viel. Neugier ist hilfreich, ebenso wie ein entspannter Umgang mit der eigenen Ahnungslosig-

keit – zum Beispiel dann, wenn man sich entschließen muss, andere um Rat zu bitten. Man braucht die Bereitschaft, geduldig Dinge nicht zu verstehen. Man muss Texte lesen, die man nicht durchschaut, und man muss hinnehmen, dass man sich noch kein funktionierendes Modell der Realität bilden kann. Man darf sich nicht darüber ärgern, dass man nach zwei Stunden Beschäftigung mit einem neuen Konzept immer noch nicht begreift, wovon der Autor von »C++ über Nacht« eigentlich schreibt.

Lernen ist ein langwieriger Prozess. Es dauert viele Jahre – manche sagen, ein Leben lang. Wesentliche Dazulernvorgänge lassen sich zwar etwas beschleunigen, aber sie brauchen trotzdem ihre Zeit. Joe Armstrong, der Erfinder der Programmiersprache Erlang, erklärt in »Coders at Work«, wie er im Laufe seines Lebens klüger wurde: Am Anfang seiner Laufbahn musste er ein Programm erst schreiben, um herauszufinden, ob es funktioniert. 20 Jahre später kann er sich dasselbe einfach denken. Zeit spart er dadurch allerdings nicht – die Ausprobiermethode dauert ein Jahr, die Nachdenkmethode auch –, er muss nur weniger tippen.[3]

Die Lektüre dieses Buchs wird Sie also keineswegs gleich klüger machen. Auch nicht zu einem guten Programmierer. Am Ende des Buchs werden Sie im günstigsten Fall ein weniger schlechter Programmierer geworden sein. Ein weniger schlechter Programmierer weiß, dass er das, was er heute anrichtet, morgen selbst ausbaden wird. Er erkennt, an welchen Stellen er sich das Leben kurzfristig leichter und langfristig dafür sehr viel schwerer macht. Er nimmt Rücksicht auf seine eigene Fehlbarkeit. Ein weniger schlechter Programmierer ist dazu bereit, seinen Code anderen zu zeigen und sich Kritik anzuhören. Er beginnt, Verantwortung für seine Taten zu übernehmen. Er sagt nicht »Die API ist unausgereift«, »In PHP geht es nicht besser«, »Das liegt an Fehlern in der Library, für die ich nichts kann« oder »Es musste halt schnell gehen«. Sogar bei wirklich unverschuldeten Problemen fühlt sich der weniger schlechte Programmierer selbst zuständig und bemüht sich um Verbesserungen. Er tut einfach nicht wider besseres Wissen das Falsche.

Oder wenigstens nicht mehr ganz so oft.

Die nächsten 422 Seiten

Dieses Buch enthält relativ viel Text und nur wenige Codebeispiele. Dass es konkrete Anleitungen für das Schreiben besseren Codes da draußen gibt, wissen Sie ja wahrscheinlich. Aber irgendetwas hat Sie bisher daran gehindert, eine solche Anleitung ausfindig zu machen und zu beherzigen. Unser Buch befasst sich mit den Gründen für diese Scheu vor dem Dazulernen und soll die schlimmsten Probleme auf eine Art lindern, die möglichst wenig Arbeit verursacht.

Der erste Teil, »~~Hallo Wels~~ Hallo Welt«, beschäftigt sich mit der Gratwanderung zwischen Selbstzweifeln und Selbstüberschätzung und dem Weg von der unbewussten zur

3 »Coders at Work« von Peter Seibel (Apress 2009), S. 215.

bewussten Inkompetenz. Im zweiten Teil, »Programmieren als Verständigung«, geht es darum, wie man sich selbst und anderen Menschen klarmacht, was man mit dem Code eigentlich bezweckt – oder in einer fernen Vergangenheit, also letzte Woche, einmal bezweckt haben könnte. Der dritte Teil, »Umgang mit Fehlern«, handelt vom Unrechthaben und seinen Problemen. Wie kann man mit demselben Kopf, der einen Fehler verursacht hat, diesen Fehler erkennen, beheben und vielleicht sogar in Zukunft vermeiden? Im vierten Teil, »Wahl der Mittel«, geht es um den Werkzeugkasten des Programmierers. Was braucht man nicht selbst zu machen, was sollte man auf keinen Fall selbst machen? Nach der Lektüre werden Sie vielleicht immer noch nicht objektorientiert programmieren oder eine Entwicklungsumgebung einsetzen wollen, aber zumindest wissen Sie dann, wozu diese Werkzeuge erfunden wurden und bei welchen Aufgaben Sie sich Arbeit sparen könnten, wenn Sie eines Tages Ihre Meinung ändern.

Am Ende einiger Kapitel und am Ende des Buchs geben wir Tipps zum Weiterlesen. Wenn es einen deutschen Wikipedia-Eintrag zu einem Thema gibt und er gut ist, verweisen wir darauf; wenn nicht, auf den englischen. Falls die Welt so viele schlechte Programmierer beherbergt, wie die inoffizielle Marktforschung der Autoren nahelegt, könnte sich das Buch gut genug für eine zweite Auflage verkaufen. Sie helfen uns und den zukünftigen Lesern, wenn Sie gefundene Fehler, Kritik und Verbesserungsvorschläge an *wenigerschlechtprogrammieren@kulturindustrie.com* schicken.

TEIL 1

~~Hallo Wels~~ Hallo Welt

KAPITEL 1

Bin ich hier richtig?

Um herauszufinden, ob dieses Buch für Sie geschrieben ist, nehmen Sie sich ein paar Augenblicke Zeit und lesen Sie sich die unten stehenden Fragen durch. Antworten Sie ehrlich und ohne lange zu überlegen. Wenn Sie eine Antwortmöglichkeit gar nicht verstehen, machen Sie sich keine Gedanken, sondern wählen eine andere.

Ich schreibe meine Programme ...

a) in Notepad.

b) im Browser.

c) in irgendwas anderem.

Wenn etwas nicht funktioniert ...

a) poste ich eine Fehlerbeschreibung mit dem Titel »Hilfe!!!« unter exakter Nennung aller verwendeten Hardwarekomponenten in einem passenden Forum.

b) baue ich viele »print«-Zeilen ein, die mir den Inhalt von Variablen ausgeben.

c) debugge ich mit GDB

Zur Versionskontrolle benutze ich ...

a) gar nichts. Wenn ich versehentlich was lösche, muss ich es neu schreiben. Deshalb passe ich immer sehr gut auf.

b) SVN.

c) Git oder mercurial.

Ich kommentiere meinen Code ...

a) nie, weil ich nicht so viel tippen will.

b) nie, weil ich meinen Code für selbsterklärend halte.

c) nie, weil mein Code selbsterklärend ist.

Wenn ich einen XML-Parser brauche ...

a) nehme ich mir ein Wochenende Zeit und schreibe einen, wie schwer kann das schon sein.

b) Ich brauche keinen XML-Parser.

c) Ich lese die Wikipedia-Einträge zu SAX- und DOM-Parsern durch, sehe mir verschiedene Bibliotheken und deren Bindings an meine verwendete Programmiersprache an, wäge ihre Vor- und Nachteile ab und finde heraus, ob es eine lebendige Community dazu gibt.

Um E-Mail-Adressen zu validieren ...

a) schreibe ich schnell zwei Zeilen hin, die ich an meiner eigenen Mailadresse überprüfe.

b) teste ich, ob ein @-Zeichen enthalten ist.

c) google ich nach einer Regular Expression, auf deren Korrektheit sich namhafte Projekte verlassen.

Mein Code und das Licht der Öffentlichkeit:

a) Ich halte meinen Code geheimer als ein Messie seine Wohnung.

b) Wenn jemand Code von mir sieht, der schon ein halbes Jahr alt ist, dann ist mir das ein bisschen peinlich.

c) Mein Code ist Teil des Linux-Kernels.

Ich teste meinen Code ...

a) gar nicht. Wenn etwas nicht mehr funktioniert, merke ich das schon früher oder später.

b) nach jeder Änderung an meinem Code.

c) gar nicht. Nach jeder Änderung an meinem Code prüfen automatisierte Unit-Tests, ob noch alles funktioniert.

Wie sieht ein geeignetes Datumsspeicherformat aus?

a) Wie schon, »T.M.JJ« natürlich!

b) Die vergangenen Sekunden seit 00:00 Uhr Koordinierter Weltzeit am 1. Januar 1970, wobei Schaltsekunden nicht mitgezählt werden, gespeichert in einem 64-Bit-Integer.

c) ISO 8601.

Optimierung ...

a) ist mir egal.

b) betreibe ich so lange, bis das Programm auf meinem 2,4 GHz Core 2 Duo mit 256MB 2nd Level Cache in 43.3485 Taktzyklen durch ist.

c) ist mir so lange egal, wie die User Experience nicht durch Wartezeit beeinträchtigt wird.

Alle anderen Programmierer ...

a) sind besser als ich.

b) sind schlechter als ich.

c) Mal so, mal so.

Ich überarbeite meinen Code ...

a) nie, ich nehme mir aber hin und wieder vor, alles noch mal neu und besser zu schreiben.

b) in einem winzigen Schritt nach dem anderen, wenn ich gerade viel Zeit habe.

c) in einem winzigen Schritt nach dem anderen, auch wenn ich gerade wenig Zeit habe.

Überschlagen Sie jetzt grob, ob Sie mehr als die Hälfte der Fragen mit a) oder b) beantwortet haben. Wenn ja, dann ist dieses Buch für Sie geschrieben. Haben Sie mehr als die Hälfte der Fragen mit c) beantwortet, dann lachen Sie bitte nicht höhnisch, sondern gehen weiter und programmieren Linux-Kerneltreiber. Oder was Sie sonst so tun.

KAPITEL 2

Zwischen Hybris und Demut

> »I regularly have to google the basic syntax of a language I've used every day for 10 years. #coderconfessions«
>
> @HackerNewsOnion / Twitter, 10. Juli 2013

In den letzten Jahren wurde in Geek-Kreisen gern der »Dunning-Kruger-Effekt«[1] zitiert, demzufolge ausgerechnet inkompetente Personen besonders stark dazu neigen, das eigene Können zu überschätzen. Nachfolgestudien deuten darauf hin, dass es sich in Wirklichkeit anders und einfacher verhält: Menschen sind ganz allgemein nur schlecht in der Lage, ihre eigene Kompetenz auch nur halbwegs zutreffend einzuschätzen.

Ungeübte Programmierer schwanken zwischen Selbstüberschätzung und dem Glauben, zu dumm für das Metier zu sein. In der Begeisterung, zu der die Planung eines neuen Projekts führt, werden die eigenen Fähigkeiten oft überschätzt, insbesondere macht man sich keine Vorstellung davon, wie langsam die Entwicklung von funktionierendem Code tatsächlich voranschreitet. Die Konfrontation mit der unangenehmen Wahrheit führt dann entweder zu Verzweiflung (bei Projekten mit Deadline) oder Lustlosigkeit (bei Hobbyprojekten).

Selbstüberschätzung hat mehrere Ursachen. Zum einen lernt man als Anfänger ständig hinzu, das schmeichelt dem Ego. Die Lernkurve ist in den ersten zwölf Monaten, in denen man etwas Neues lernt, so steil, dass man nicht anders kann, als sich für einen ausgemachten Topchecker zu halten. Was schwer zu erkennen ist: Trotz des vielen und schnellen Dazulernens ist man gerade erst bis zu den Knöcheln ins Nichtschwimmerbecken gewatet. Die Welt der Programmierung ist für den Anfänger, um es mit Donald Rumsfeld zu sagen, voller »unknown unknowns«, also voller Wissenslücken, die man gar nicht als Lücken erkennt.

Eine weitere Ursache für Selbstüberschätzung liegt darin, dass unerfahrene Programmierer dazu neigen, nur die ersten 80 Prozent eines Projekts überhaupt zu erledigen, weil sie auf-

1 *de.wikipedia.org/wiki/Dunning-Kruger-Effekt*.

hören, sobald die Aufgabenstellung weniger interessant wird. Der 80-20-Regel[2] zufolge verursachen die dann noch folgenden 20 Prozent allerdings 80 Prozent der Arbeit. Man baut also mit einem Bruchteil der Programmierkenntnisse eines Profis eine Applikation zusammen, die »praktisch dasselbe« ist wie ihr Vorbild. Die dabei durch Unkenntnis und Fehlentscheidungen verursachten Probleme fallen nicht weiter auf, weil sie erst innerhalb der letzten 20 Prozent der Aufgabe ihre hässliche Fratze zeigen.

Überschätzung der eigenen Fähigkeiten oder Unterschätzung einer Aufgabe kann auch Vorteile haben. Wenn jeder von Anfang an über das Ausmaß seines Unwissens Bescheid wüsste, lebte die Menschheit vermutlich immer noch in Erdhöhlen (»Hochbau? Was man da alles wissen müsste!«). Auch bei einzelnen Projekten kann es hilfreich sein, sich zu verschätzen. Als Donald Knuth 1977 – frustriert über den hässlichen Textsatz des zweiten Bandes seines Hauptwerks »The Art of Computer Programming« – beschloss, selbst ein besseres Satzprogramm zu schreiben, rechnete er mit etwa sechs Monaten Entwicklungszeit. Das Ergebnis hieß TeX und war schon knapp zwölf Jahre später fertig. Der Doktorand George Bernard Dantzig erschien 1939 zu spät zu seinem Statistikkurs an der University of California und fand zwei Probleme an der Tafel vor, die er für Hausaufgaben hielt. Einige Tage später entschuldigte er sich bei seinem Professor, dass er so lange gebraucht habe, um die Aufgaben zu lösen; sie seien etwas schwieriger gewesen als sonst. Allerdings handelte es sich gar nicht um Hausaufgaben, sondern um bis dahin unbewiesene Statistiktheoreme, die Dantzig damit versehentlich bewiesen hatte. In manchen Fällen sind Fehleinschätzungen der Lage also durchaus hilfreich, weil sie einen davor bewahren, sich vom Umfang oder der Komplexität einer Aufgabe abschrecken zu lassen.

Auf der anderen Seite des Wetterhäuschens wohnt der Selbstzweifel. Der eigene Code ist ein hässliches, unwartbares Dickicht. Sicher müssen andere, bessere Programmierer nicht jeden Befehl vor jeder Verwendung in der Dokumentation nachschlagen. Niemand sonst vergisst über Nacht, was er sich beim Schreiben des Codes gedacht hat. Man ist einfach unbegabt und faul und wird zeitlebens ein schlechter Programmierer bleiben.

Die gute Nachricht: Den anderen geht es genauso. Auch erfahrene Programmierer sind vergesslich, unkonzentriert, arbeitsscheu und halten den eigenen Code für den schlechtesten der Welt. Es gibt nur wenige Probleme, die ausschließlich unerfahrene, halbherzige oder eilige Programmierer betreffen:

Unwartbarer Code

Code wird ohne Rücksicht auf spätere Wartbarkeit (durch einen selbst oder andere) geschrieben. Das liegt in erster Linie an mangelnder Erfahrung. Erst wenn man oft genug vor einem selbst erzeugten, undurchdringlichen Codedschungel gestanden hat, fällt es etwas leichter, beim Schreiben an den künftigen Leser zu denken. Nur ungewöhnlich

2 *de.wikipedia.org/wiki/Paretoprinzip.*

phantasiebegabte oder zukunftsorientierte Programmierer erfassen diesen Sachverhalt gleich von Anfang an. Alle anderen müssen erst das Tal der Schmerzen durchschreiten.

Wahl ungünstiger Mittel

Wer nur eine halbe Programmiersprache oder Technik beherrscht, der wird sie zur Lösung sämtlicher Aufgaben heranziehen, die ihm einfallen. Oft ist das Problem dabei nicht totale Unkenntnis, sondern nur mangelnde Vertrautheit mit der nächstliegenden Herangehensweise.

Selbstüberschätzung

Auch gute Programmiererinnen überschätzen häufig ihre Produktivität. Das liegt zum einen daran, dass ihnen wie allen anderen Menschen die Fähigkeit zur realistischen Einschätzung fehlt, wie lange ein Projekt dauern wird. Zum anderen kristallisiert sich die eigentliche Problemstellung oft erst im Laufe der Arbeit heraus. Das ist bei schlechten Programmiererinnen nicht anders – nur um etwa drei Größenordnungen schlimmer.

Fehlendes Vorwissen

Jedes Konzept, dem der unerfahrene Programmierer begegnet, ist für ihn neu. Auch in der Softwareentwicklung gibt es jedoch relativ viele Ideen, die wieder und wieder in neuem Gewand Anwendung finden. Erfahrene Programmierer erkennen diese Muster, können sie benennen und tun sich dann leichter damit, sie in anderen Kontexten anzuwenden.

Schwächen als Stärken

Der Perl-Erfinder Larry Wall nennt im Standardwerk »Programmieren mit Perl« Faulheit, Ungeduld und Selbstüberschätzung als wichtige Programmiertugenden. Faulheit ist gut, weil sie die Programmiererin dazu motiviert, so viel Energie wie möglich einzusparen. Sie wird arbeitssparende Software schreiben, das Geschriebene gründlich dokumentieren und eine FAQ verfassen, um nicht so viele Fragen dazu beantworten zu müssen. Ungeduld bringt die Programmiererin dazu, Software zu schreiben, die ihre Bedürfnisse nicht nur erfüllt, sondern vorausahnt. Und Selbstüberschätzung lässt sie auch das Unmögliche anpacken. Aber auch andere Untugenden können Programmierern, richtig eingesetzt, zum Vorteil gereichen:

Dummheit

»Mir kommt es oft so vor, als käme der schlimmste Spaghetticode von den Leuten, die am meisten gleichzeitig im Kopf behalten können. Anders bringt man solchen Code ja gar nicht zustande«, vermutet der Softwareentwickler Peter Seibel.[3] Ein weniger intelli-

3 »Coders at Work«, S. 311.

genter Programmierer wird versuchen, eine möglichst einfache Lösung zu finden und dadurch mit höherer Wahrscheinlichkeit Code schreiben, den auch andere Menschen verstehen, warten und erweitern können.

Unwissenheit

Wissen um die Details einer Programmiersprache oder um abstraktere Konzepte ist eine feine Sache. Sobald es aber zu einem Paradigmenwechsel kommt – die Einführung der objektorientierten Programmierung war ein solcher –, werden viele bisher kompetente Programmierer durch ihr vorhandenes Wissen ausgebremst. Weitgehend ahnungslose Geschöpfe haben es in so einer Situation leichter, weil sie unvorbelastet an das Neue herangehen können.

Vergesslichkeit

Douglas Crockford, eine wichtige Figur in der Entwicklung von JavaScript, antwortet auf die Interviewfrage, ob Programmieren nur etwas für die Jugend sei: »Ich bin heute vielleicht ein bisschen besser als früher, weil ich gelernt habe, mich weniger auf mein Gedächtnis zu verlassen. Ich dokumentiere jetzt gründlicher, weil ich mir nicht mehr so sicher bin, dass ich nächste Woche noch weiß, warum ich etwas so und nicht anders gelöst habe.«[4] Außerdem wird ein Programmierer, der ständig Namen und Syntax der einfachsten Funktionen vergisst, stärker motiviert sein, sich eine Entwicklungsumgebung oder wenigstens einen Editor mit intelligenten Codevervollständigungsoptionen zuzulegen.

Fehlendes Durchhaltevermögen

In manchen Kreisen gilt es als heroisch, die ganze Nacht durchzuprogrammieren. Die Wahrscheinlichkeit, dass man den so entstandenen Code am nächsten Tag wegwirft, ist hoch. Wer es sich leisten kann, weil ihm kein Chef und keine Deadline im Nacken sitzt, tut besser daran, sich nicht zur Arbeit am Code zu zwingen. Oft kann eine gute Idee am Badesee ganze Wochen unsinniger Arbeit ersetzen.

Prokrastination

Gerade für schlechte Programmierer ist es nützlich, Verbesserungen am Code so lange wie möglich aufzuschieben. Falls es eines Tages wirklich unumgänglich ist, etwas zu ändern, wird man bis dahin ein besserer Programmierer geworden sein. Dann ist es gut, dass man nicht früher ans Werk gegangen ist. Außerdem werden Werkzeuge, Programmiersprachen, Codebibliotheken und Frameworks mit der Zeit immer besser. Die Lösung eines Problems wird also einfacher, je länger man wartet, denn andere, erfahrenere Programmierer erledigen dann einen Teil der Arbeit. Oft genug kann eine Aufgabe einfach ausgesessen werden, bis ein entsprechendes Open Source-Projekt das Problem

4 »Coders at Work«, S. 114.

gelöst hat. Durch aktive Beteiligung an so einem Projekt lässt sich dieser Prozess natürlich noch beschleunigen.

Ekel vor dem eigenen Code

Es ist gut und richtig, den eigenen Code zu hassen. An guten Code klammert man sich mit irrationaler Anhänglichkeit. Schlechten Code wird man bereitwillig wegwerfen, sobald Nutzer sich Änderungen wünschen oder ein neuer Lösungsweg auftaucht. Wer seinen Code für perfekt hält, lernt zu wenig dazu. Die Zusammenarbeit mit anderen ist produktiver, wenn die Beteiligten nicht in ihren eigenen Code verliebt sind. Und die übermäßige Beschäftigung mit der Codequalität kann Energie dort abziehen, wo sie dringender gebraucht wird: beim Nachdenken über Zweck und Ergebnis der Arbeit. Code ist lediglich ein Lösungsweg und – im Gegensatz zu Literatur – kein Selbstzweck.

Ehrgeizlosigkeit

Mittelmäßiger Code macht eventuell froher – vielleicht nicht unbedingt seine Nutzer, aber wenigstens seinen Urheber. Die Glücksforschung unterscheidet zwischen »Maximizers«, die stets das Beste anstreben, und »Satisficers«, die sich auch mit weniger zufriedengeben. Die Maximizer mögen bessere Ergebnisse erzielen, glücklicher aber sind die Satisficer.

Trägheit

Geschickt eingesetzte Trägheit kann zum Werkzeug werden. Ein Beispiel: Globale Variablen sollten nur dort verwendet werden, wo sie wirklich nötig sind – also ziemlich selten.[5] Weil es aber so bequem ist, Daten, auf die man an mehreren Stellen zugreifen möchte, in eine globale Variable zu stecken, neigen Anfänger dazu, dieses Mittel überzustrapazieren. Es kostet viel Mühe, eine solche global angelegte Variable später wieder loszuwerden. Umgekehrt kann man die Kraft der eigenen Trägheit in den Dienst der guten Sache stellen, indem man grundsätzlich alle Variablen lokal anlegt und erst dann global werden lässt, wenn es gar nicht mehr anders geht. Jetzt macht sich die Trägheit nützlich, indem sie einen daran hindert, den Code zu verschlechtern.

Das Hauptproblem des Glaubens an die eigenen Schwächen ist, dass er sich so gut als Ausrede benutzen lässt: »Ich muss gar nicht erst versuchen, es besser zu machen, weil ich es nicht besser machen kann. Ich bin einfach nicht schlau genug.« Diese Annahme ist nicht nur kontraproduktiv, sondern auch falsch. Die meisten persönlichen Schwächen und Unfähigkeiten erweisen sich bei näherer Betrachtung als so weit verbreitet, dass man sie sinnvoller als Normalzustand des menschlichen Geistes betrachten sollte. Irgendwo an der Peripherie des Bekanntenkreises mag es einen Superprogrammierer geben, der Fehler im Code auf den ersten Blick identifiziert, die Umgekehrt Polnische Notation

5 Die Nachteile globaler Variablen sind im Wikipedia-Eintrag *en.wikipedia.org/wiki/Global_variables* erläutert.

beherrscht und sämtliche Syntaxfinessen seiner zwölf Lieblingssprachen im Kopf hat. Aber die Existenz dieses Superprogrammierers bedeutet nur, dass Ausnahmen von der allgemeinen Unzulänglichkeit möglich sind.

Richtiges muss nicht schwierig sein

Der selbst geschriebene Code sieht womöglich aus wie im Kinderbuch: »Das ist der Hund. Der Hund spielt im Garten. Jetzt bellt der Hund.« Aber das ist kein Grund zur Demut. Nicht die Fähigkeit, möglichst komplexen Code zu schreiben, zeichnet gute Programmierer aus. Elegante und gute Ideen lassen sich auch in schlichtem Code ausdrücken, und umgekehrt kann sich hinter kompetentem Code ein schlecht durchdachtes Konzept verbergen.[6]

Ein aus dem Flugzeugbau stammendes und auch in anderen technischen Bereichen sehr populäres Motto lautet »KISS«, was für »Keep it simple, stupid« steht. Motivation dieses Designprinzips war ursprünglich, zu gewährleisten, dass amerikanische Bomber von einem einzelnen Mechaniker in zwei Minuten mit einem Schweizer Taschenmesser repariert werden können. Auf Softwareentwicklung angewendet, besagt es, dass Formulierungen im Kinderbuchstil nicht nur akzeptabel, sondern sogar vorteilhaft sind, weil sie unmissverständlich und leicht zu warten sind.

Häufig sind die Aufgaben, mit denen man als Programmierer zu tun hat, auch gar nicht so kompliziert wie zunächst gedacht. Bernie Cosell, einer der Programmierer hinter dem Ur-Internet Arpanet, erklärt: »Ich habe ein paar Grundregeln, die ich den Leuten beizubringen versuche. Das sind meistens Leute, die direkt vom College kommen und glauben, sie wüssten alles über Programmierung. Zum einen geht es mir dabei um die Einsicht, dass es nur ganz wenige an sich schwierige Programmierprobleme gibt. Wenn man Code betrachtet, der sehr schwierig aussieht – wenn man überhaupt nicht versteht, was das alles soll –, dann ist das fast immer ein Anzeichen dafür, dass dieser Code schlecht durchdacht ist. Dann krempelt man nicht die Ärmel hoch und versucht, den Code geradezuziehen, sondern man lehnt sich erst mal zurück und denkt nach. Wenn man lange genug nachgedacht hat, stellt sich heraus, dass alles ganz einfach ist. (...) Ich bin lange genug im Geschäft; ich weiß, dass es ein paar sehr komplexe Probleme gibt. Aber das sind nicht viele. Es ist immer wieder dasselbe: Wenn man gründlicher darüber nachdenkt, wird es einfacher, und die eigentliche Programmierung ist dann am Ende ganz leicht.«[7]

---

6 Ein Sonderfall ist schlechter Code, der ein schlechtes Konzept verbirgt: »In other words – and that is a rock-solid principle on which the whole of the Corporation's Galaxywide success is founded – their fundamental design flaws are completely hidden by their superficial design flaws.« Douglas Adams, »So Long, and Thanks For All the Fish«.

7 »Coders at Work«, S. 542.

Wir brauchen einen vernünftigen Umgang mit unserer eigenen Fehlbarkeit und der anderer Menschen. Alles kann immer falsch sein und ist es meistens auch (eine Regel, die man insbesondere beim Lesen von Codekommentaren im Kopf behalten sollte). Aber es ist unproduktiv, sich deshalb für einen Idioten zu halten. Ein Kleinkind, das sprechen lernt, hat kein Problem damit, wenn es »ich habe getrinkt« sagt. Kind und Eltern verstehen, was gemeint ist, und hinter diesem einfachen Satz stecken erhebliche Leistungen der Evolution und des Individuums. Wer in der Lage ist, sich irgendeiner Programmiersprache zu bedienen, um auch nur »Hallo Welt« zu sagen, der hat es weit gebracht und braucht sich nicht durch die Lektüre von Blogbeiträgen, die von »real programmers« handeln, entmutigen zu lassen. Alles, was jetzt noch fehlt, ist der Wille, die Sache halbwegs gut zu machen. Man muss Störendes und Falsches erkennen und bereit sein, es zu ändern – vielleicht nicht sofort, aber doch zumindest irgendwann.

TEIL 2

Programmieren als Verständigung

KAPITEL 3

Du bist wie die andern

> »Any fool can write code that a computer can understand. Good programmers write code that humans can understand.«
>
> Martin Fowler, »Refactoring«

Eine Programmiersprache ist in erster Linie eine künstliche Sprache zur Kommunikation von Menschen mit Maschinen. Im Unterschied zu natürlichen Sprachen, die der Kommunikation zwischen Menschen dienen, sind Programmiersprachen vollkommen eindeutig definiert: Es gibt keinerlei Interpretationsspielraum, was ein bestimmtes Sprachkonstrukt bedeutet. Diese Eigenschaft macht die Sprache maschinenlesbar.

Dass Programmiersprachen keine Mehrdeutigkeiten zulassen, bedeutet jedoch nicht, dass Missverständnisse ausgeschlossen sind, denn sie haben noch eine zweite wichtige Funktion: Sie sind ein Kommunikationsmittel zwischen dem ursprünglichen Programmierer, der das Programm geschrieben hat, und einem anderen, der es später liest. Da Menschen etwas weniger logisch denken als Computer, kann es dazu kommen, dass der Programmierer nicht schreibt, was er meint, oder dass ein zweiter Programmierer den Text des ersten Programmierers falsch versteht.

Unangenehm wird es, wenn dieser zweite Programmierer auf Grundlage seines falschen Verständnisses den Code verändert oder falsch aufruft, denn dann geht mit großer Wahrscheinlichkeit etwas kaputt. In vielen Fällen ist der zweite Programmierer mit dem ersten identisch – er ist nur ein paar Wochen oder Monate älter.

Diese Missverständnisse sind es, die Zeit kosten und, falls sie nicht erkannt und ausgemerzt werden, zu Abstürzen oder falschen Ergebnissen führen. Für gute Programmierer hat das Vermeiden von Missverständnissen daher allerhöchste Priorität.

Jede unserer Äußerungen, ob sie im Gespräch mit einem Menschen stattfindet, beim Schreiben von Büchern oder beim Programmieren, enthält nur einen Teil der zum Verständnis nötigen Informationen. Der andere Teil bleibt unausgesprochen im Kopf zurück. Was wir im ersten Versuch von uns geben, ist deshalb meistens miss- und oft ganz unverständlich. Wir merken das nur selten, weil sich kaum jemand die Mühe macht, nachzufragen. Der Gesprächspartner hat nicht so ganz zugehört, weil er darüber

nachdenkt, was er selbst Wirres sagen will, und Leser haben nicht immer die Möglichkeit, einen Autor so lange zu schütteln, bis der sich verständlich ausdrückt. Nicht zuletzt wegen dieses fehlenden Feedbacks neigen wir fast alle dazu, die Gedankenlesefähigkeiten von Lesern, Zuhörern und Computern zu überschätzen. »Aber das steht da doch!«, protestieren wir auf Nachfrage, und »Hab ich doch gesagt!« In Wirklichkeit haben wir es uns bestenfalls gedacht.

Mit dem Schreiben verständlichen Codes, dem Nachdenken über Namensgebung und dem Kommentieren verhält es ein bisschen wie mit Safer Sex: Jeder erkennt den Nutzen und ist dafür, und doch wird es zu selten praktiziert – insbesondere, wenn es wirklich drauf ankommt. Das Haupthindernis ist dabei psychologischer Natur: Im Moment des Schreibens hat man Verständnishilfen nicht nötig. Man weiß, was man tut, und alle nötigen Informationen kursieren noch frisch im Kurzzeitgedächtnis. Außerdem will man mit dem Programm weiterkommen und sich nicht durch das Schreiben von Kommentaren ablenken lassen.

Der Autor müsste sich also in fremde Leser seines Codes hineinversetzen und deren Bedürfnisse nachvollziehen, was zusätzliche Arbeit verursacht und deshalb gern unterlassen wird. Manchmal missachtet er diese Bedürfnisse sogar ganz bewusst, etwa wie er die andren Leser als Konkurrenten um seinen Arbeitsplatz wahrnimmt.

Ob man sich um lesbaren Code bemüht, hängt also auch wesentlich davon ab, welches Verhältnis man zu seinen imaginären Codelesern hat. Dieses Verhältnis lässt sich freundlicher gestalten mithilfe der von buddhistischen Weisen sowie den Fantastischen Vier beschriebenen Einsicht, »Du bist wie die andern und die andern sind wie du.« Die Leser, die jede Hilfe beim Verständnis des Codes gebrauchen können, sind wir selbst – heute beim Versuch, den Code anderer zu verstehen, und morgen beim Betrachten des eigenen Codes. Die Erkenntnis, dass der eigene Code einem sehr schnell sehr fremd wird, ist unter Programmierern so weit verbreitet, dass sie schon vor Jahrzehnten Gesetzesform angenommen hat. »Eagleson's Law« lautet: »Any code of your own that you haven't looked at for six or more months might as well have been written by someone else.« Eine Ergänzung: Eagleson war Optimist, in Wirklichkeit sind es eher drei Wochen.

Widerstehen Sie daher der Einflüsterung »Das merkst du dir doch auch so!« des Codeteufels. Es schadet nicht, sich beim Programmieren vorzustellen, man sei der Protagonist aus »Memento«, der alles, was er nicht sofort schriftlich festhält, wenige Minuten später wieder vergessen hat. Das hat auch mit Respekt vor der eigenen Arbeitszeit zu tun, denn dass man einmal Zeit auf das mühsame Verstehen eines Problems aufgewendet hat, heißt nicht, dass man es ab jetzt für immer verstehen wird. Auch Verständnis hat ein Verfallsdatum, und man kann heute schon die Zeit von morgen einsparen, indem man sich selbst das Verstandene aufschreibt und erklärt. Dass dadurch auch andere Menschen den Code leichter erfassen können, ist eine praktische Nebenwirkung.

KAPITEL 4

Konventionen

> »Ich habe in meiner Laufbahn mit vielen Programmierern zusammengearbeitet, und ich habe einige kennengelernt, die waren grauenhaft schlecht. Ganz, ganz furchtbar. Haarsträubend. Da war nicht mal der Code vernünftig eingerückt!«
>
> Lukas Hartmann, Softwareentwickler

Man hat gerade eigenhändig die ersten zwei, drei Projekte zum Laufen gebracht, ohne sich dabei um herkömmliche Praktiken zu scheren. Jetzt blickt man in den Spiegel und sieht dort eine coole Person mit den richtigen Prioritäten im Leben. Jeder, der einem erklären will, dass der so entstandene Code gegen irgendwelche etablierten Regeln verstößt, ist erst mal ein Blockwart und Erbsenzähler. Das ist eine übliche und in gewissen Grenzen auch sinnvolle Reaktion von Menschen, die auf einem bestimmten Gebiet ganz neu anfangen und sich gegen diejenigen behaupten müssen, die schon da sind.

Das Misstrauen des Anfängers gegenüber dem Etablierten ist nicht völlig unbegründet: Zu einem Teil steckt hinter diesen Konventionen wirklich die schäbige Tatsache, dass jede Gruppe bemüht ist, sich durch willkürliche Sprachregelungen und Praktiken vom Rest der Welt abzugrenzen: Hier sind wir, wir machen es richtig. Da sind die anderen, die machen es falsch, essen Hunde und rücken ihren Code nicht ordnungsgemäß ein. Aber dieser Abgrenzungswunsch ist eben nur ein Aspekt. Es gibt auch bessere Argumente für das Einhalten etablierter Bräuche beim Programmieren:

- Als Anfänger ignoriert man die Konventionen seiner Sprache oft gar nicht aus revolutionärer Überzeugung, sondern aus schierer Unkenntnis.
- Einen eigenen Programmierstil kann man erst entwickeln, wenn man seine Sprache beherrscht. Bis dahin ist die Wahrscheinlichkeit recht groß, dass die eigenen neuen Ideen schlechter als etablierte Konventionen sind. Menschen, die das Althergebrachte infrage stellen, arbeiten im Dienste der Weltverbesserung und haben auch die Softwareentwicklung wesentlich vorangebracht. Aber um sich auf diesem Gebiet nützlich zu machen, sollte man a) das Althergebrachte erst mal verstehen und b) es nicht nur anders machen wollen, sondern auch besser machen können. Die Wahr-

scheinlichkeit, dass man als eher mäßig geübter Programmierer über diese Voraussetzungen verfügt, ist sehr gering.

- Selbst wenn die eigenen Ideen nicht schlechter sind als die etablierten, behindert man sich durch Privatregelungen unnötig beim Lesen von Fremdcode und in der Zusammenarbeit mit anderen.

Für Anfänger ist es Zeitverschwendung, sich mit den relativen Vorteilen bestimmter Klammer- oder Einrückungsstile auseinanderzusetzen. Auf anderen Gebieten kann man viel größere Produktivitätsfortschritte bei geringerem Aufwand erzielen. Wer das Problem pragmatisch lösen möchte, sollte daher einfach in den ersten Jahren alles widerstandslos befolgen, was ihm von besseren Programmierern empfohlen wird, auch wenn es ihm seltsam vorkommt. Auch Standardbibliotheken, also jene Sammlungen von Funktionen und Klassen, die zusammen mit dem Compiler oder Interpreter einer Sprache ausgeliefert werden, sind oft eine gute Quelle, denn mit diesen Konventionen sind die meisten Entwickler bereits vertraut.

Wenn eine bestimmte Konvention Sie ärgert und Sie alles ganz anders machen wollen, dann bedenken Sie bitte, dass Sie indirekt mit ziemlich vielen Leuten zusammenarbeiten, selbst als einzelner Freizeitprogrammierer: Sie verwenden Codebeispiele und Bibliotheken, die von anderen geschrieben wurden, und werden sich wahrscheinlich irgendwann mit einem schwierigen Fehler hilfesuchend an andere Menschen im Netz wenden. Genau in diesem Moment werden Ihre ganz persönlichen Codekonventionen ein Kommunikationshindernis.

Englisch oder nicht?

Die kurze Antwort auf diese Frage lautet:

```
/**
* Kincskereso halalat vezenyli le.
*/
public void meghal()
{
    Game.jatekVege();
    this.mezo.setCellaElem(null);
}
```

Etwas ausführlicher: Programmiersprachen beruhen auf englischen Begriffen. Die Ausnahmen von dieser Regel lassen sich an relativ wenigen Händen abzählen und sind für Alltagsprogrammierer nicht von Bedeutung.[1] Manchmal wird von der Sprache oder einem verwendeten Framework eine bestimmte Benennung erzwungen: In C oder Java wird man den Startpunkt des Programms immer `main` nennen müssen, auch als deutschsprachiger Entwickler. Bedeutet das automatisch, dass man als Programmierer auch englische Namen vergeben und englische Kommentare schreiben muss?

1 Eine Liste findet sich unter *en.wikipedia.org/wiki/Non-English-based_programming_languages*.

Für den Einsatz der Muttersprache spricht:

- Es ist mühsamer, sich passende englische Namen auszudenken und verständliche englische Kommentare zu schreiben.
- Das Ergebnis ist je nach Stand der eigenen Englischkenntnisse manchmal wenig überzeugend; schlechtes Englisch führt zu Verwirrung und Missverständnissen.
- Man macht mehr Schreibfehler im Englischen, weil einem als Nicht-Muttersprachler nicht so geläufig ist, wie beispielsweise `height` und `width` geschrieben werden.

Für Englisch spricht:

- Deutsch im Code sagt dem Leser auf den ersten Blick: Hier hat jemand nur für sich selbst programmiert, ohne damit zu rechnen, dass sich jemals jemand anders für den Code interessieren könnte. Das tun überwiegend Anfänger, also ist der Code wahrscheinlich nicht besonders gut.
- Viele Programmierer finden es unbequem oder abstoßend, ein zweisprachiges Codegemisch lesen zu müssen: `if haus[zaehler] instanceof HochHaus ...` Besonders unbeliebt sind gemischte Methodennamen: `getKundenNummerFromDatenbank()`. Code in einer einzigen Sprache ist angenehmer zu lesen. Und da diese Sprache nun mal nicht Deutsch sein kann, bleibt nur Englisch.
- Wer sein Programmier-Englisch nicht übt, bleibt immer auf deutschsprachige Programmierer-Communities angewiesen. Oft ist die Lösung eines Problems aber nicht dort, sondern nur in einem internationalen Forum zu finden.
- Wenn man sich hilfesuchend an eine englischsprachige Community wendet, muss man vor dem Veröffentlichen eines halb-deutschen Codebeispiels erst alles ins Englische übersetzen.
- Dasselbe gilt, wenn man eigenen Code anderswo als gutes oder schlechtes Beispiel zur Verfügung stellen möchte.
- Englisch kann man immer mal brauchen. Nicht nur in der Programmierung.

Die beste Lösung ist durchgehend englischer, auf Englisch kommentierter Code. Englischer Code mit anderssprachigen Kommentaren ist ein verbreiteter, wenn auch unschöner Kompromiss:

```
// ne pas oublier de renseigner ces valeurs
// sinon l'addon ne pourras pas etre utilisée
_resource = "";
```

Irgendetwas darf der Nutzer des Codes hier nicht vergessen. Nur was, und wann? Wenn Sie sich für diese Variante entscheiden, gestalten Sie den Code aus Rücksicht auf Ihre Leser bitte so, dass er auch ohne Kommentare verständlich ist. Das ist generell eine ganz gute Idee (siehe Kapitel 6).

Angriff der Paamayim Nekudotayim

PHP-Nutzern kann es passieren, dass sie aus heiterem Himmel mit der Nachricht `Parse error: syntax error, unexpected T_PAAMAYIM_NEKUDOTAYIM` konfrontiert werden. Was aussieht wie eine Warnung vor der unmittelbar bevorstehenden Machtübernahme durch Außerirdische, ist eine hebräische Fehlermeldung, die ursprünglich aus der in Israel entwickelten Zend Engine 0.5 stammt. *Paamayim Nekudotayim* sind zwei aufeinanderfolgende Doppelpunkte. Die Fehlermeldung hat immerhin den Vorteil großer Eindeutigkeit, ihre Eingabe in Suchmaschinen führt viel schneller zur Erleuchtung als eine Suche nach »double colon«. Machen Sie es den Zend-Entwicklern trotzdem nicht nach.

Nützlich ist es außerdem, sich frühzeitig mit sich selbst oder anderen Programmierern darüber zu verständigen, ob man US-Englisch oder britisches Englisch verwenden will. Das versäumen nicht nur Programmieramöben, sondern beispielsweise auch die Urheber von Twitter, so dass jeder, der die Twitter-API einsetzte, sich einige Jahre lang merken musste, wann er *favorites* und wann *favourites* zu schreiben hatte. Da deutlich mehr Software US-Englisch verwendet, sollte man sich auf diese Variante festlegen, auch wenn es den Briten gegenüber historisch ungerecht ist.

Englisch lernen mit dem ZX81

»Als ich ungefähr acht war, brachte mein Vater eines Abends einen der ersten Homecomputer, einen ZX81 mit nach Hause. Ein schwarzes, türstopperartiges Ding mit einer kaum benutzbaren Folientastatur, die aber den für mich unschätzbaren Vorteil besaß, dass auf ihr sämtliche BASIC-Befehle aufgedruckt waren. So konnte ich trotz kompletter Unkenntnis der Bedeutung von Wörtern wie ›print‹, ›goto‹ und ›next‹ einfach ausprobieren, was passiert, wenn man diese Befehle in ein Programm schreibt und es dann startet. Während mein Vater bald das Interesse am Programmieren verlor, verbrachte ich jede freie Minute mit dieser vermutlich ineffizientesten Methode des Programmierenlernens durch Stochern im Dunkeln. Nach ein paar Wochen zog ich dann endlich das englischsprachige Anleitungsbuch als Informationsquelle hinzu. Besonders die Codebeispiele waren hilfreich für grundsätzliche Erkenntnisse wie zum Beispiel über die irgendwie geartete Zusammengehörigkeit von `for` und `next` sowie `gosub` und `return`. Als es in der sechsten Klasse dann endlich losging mit dem Englischunterricht, war mein Wortschatz bereits um die Schlüsselwörter der Programmiersprache C angewachsen. Kurze Zeit später konnte ich dann endlich auch Sätze sagen wie: ›My name is Jan. My pen is in my pencil-case. My pencil-case is black.‹«

Jan Bölsche

Die Steinchen des Anstoßes

Klammern, Einrückungen und Leerzeilen sind für Programmierer das, was für streitlustige Badbenutzer die Zahnpastatube ist: Letztlich ist es weitgehend egal, ob die Tube vom Anfang oder vom Ende her ausgedrückt, auf den Deckel gestellt oder hingelegt, zugeschraubt oder offen gelassen wird. Aber wenn zwei Anhänger unterschiedlicher Glaubensrichtungen aufeinandertreffen, gibt es Probleme.

Im Wesentlichen geht es bei Regeln für die Codeformatierung um folgende Fragen:

Wo kommt die öffnende geschweifte Klammer hin – ans Ende der Zeile oder an den Anfang der nächsten Zeile?

```
public boolean hasStableIds() {
    return true;
}
```

Oder so:

```
public boolean hasStableIds()
{
    return true;
}
```

Wo kommt die schließende geschweifte Klammer hin?

```
public boolean hasStableIds() {
   return true;
}
```

Oder so:

```
public boolean hasStableIds() {
    return true; }
```

Werden die Einrückungen mit dem Tabulator oder mit Leerzeichen erzeugt?

Ein leider unterspezifizierter Aspekt von Textdateien ist der Abstand von Tabulator-Stopps. Da Tab-Zeichen gerne zum Einrücken von Programmblöcken verwendet werden, kommt ihnen bei der Programmierung eine besondere Bedeutung zu. In der Programmiersprache Python ist die Einrückung sogar Teil der Semantik eines Programms: Falsch eingerückte Textblöcke können Anweisungen zum Beispiel aus einer Schleife hinausbefördern. Daher vermeiden viele Teams (insbesondere Python-Entwickler) Tab-Stopps komplett und weisen ihre Editoren an, sie durch eine festgelegte Anzahl von Leerzeichen zu ersetzen.

Wie breit hat eine Einrückung zu sein?

Zwei Leerzeichen:

```
public boolean hasStableIds() {
  return true;
}
```

Oder vier:

```
public boolean hasStableIds() {
    return true;
}
```

Oder acht (der Standard im Linux-Kernel):

```
public boolean hasStableIds() {
        return true;
}
```

Sollen Zeilen nach einer bestimmten Maximallänge hart umbrochen werden?

```
scroll.setLayoutParams(new LayoutParams(LayoutParams.FILL_PARENT, LayoutParams.
FILL_PARENT));
```

Oder so:

```
scroll.setLayoutParams(new LayoutParams(LayoutParams.FILL_PARENT,
LayoutParams.FILL_PARENT));
```

Vor und nach welchen Teilen des Codes sollen Leerzeilen stehen?

```
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.thread_list);
Bundle b = getIntent().getExtras();
byte[] boardS = b.getByteArray("board");
ByteArrayInputStream bitch = new ByteArrayInputStream(boardS);
ObjectInputStream in;
final View progWrapper = findViewById(R.id.threadlist_watcher_wrapper);
progress = (ProgressBar)findViewById(R.id.threadlist_watcher);
progress.setMax(100);
progress.setProgress(0);
new AlertDialog.Builder(KCThreadListActivity.this)
```

Oder so:

```
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.thread_list);

Bundle b = getIntent().getExtras();
byte[] boardS = b.getByteArray("board");
ByteArrayInputStream bitch = new ByteArrayInputStream(boardS);

ObjectInputStream in;

final View progWrapper = findViewById(R.id.threadlist_watcher_wrapper);
progress = (ProgressBar)findViewById(R.id.threadlist_watcher);
progress.setMax(100);
progress.setProgress(0);

new AlertDialog.Builder(KCThreadListActivity.this)
```

Sollen multiple Bedingungen jeweils auf einzelnen Zeilen stehen?

```
if ((null == fileName) || (null == content)) {
    return;
}
```

Oder so:

```
if ((null == fileName) ||
    (null == content)) {
    return;
}
```

Wie kennzeichnet der verwendete Editor die Zeilenenden?

Die Unterschiede zwischen den verschiedenen Betriebssystemen werden immer geringer. Wo vor ein paar Jahren die Übertragung von Textdateien über Betriebssystemgrenzen hinweg noch für verstümmelte Umlaute gesorgt hat, bringt Unicode jetzt Ruhe. Zumindest Mac OS X und Linux sind sich über Zeilenenden in Textdateien (line feed) und Trennzeichen in Dateipfaden (Schrägstrich /) mittlerweile einig. Windows geht hier jeweils eigene Wege. Zumindest die meisten Programmierwerkzeuge verstehen aber auch unter Windows Dateipfade mit einem normalen Schrägstrich statt einem Backslash.

Alle guten Texteditoren auf allen Plattformen unterstützen die gängigen Konventionen für Zeilenenden. Außer in reinen Windows-Projekten setzt sich hier die Unix-Konvention durch: ein Line-Feed-Character (LF) ASCII 10, hexadezimal: 0A, im Sourcecode vieler Sprachen als `\n` geschrieben. Unter Windows ist die Zeichenfolge Line-Feed gefolgt von Carriage-Return (LF CR), ASCII 10 gefolgt von ASCII 13, hexadezimal: 0A 0D, im Sourcecode: `\n\r`) üblich.

Sie sollten darauf achten, dass Ihr Texteditor die im Team verwendete Konvention für Zeilenenden befolgt, ansonsten erscheinen von Ihnen bearbeitete Sourcecode-Dateien bei Ihren Kollegen in einer einzigen, sehr langen Zeile und sorgen für Unmut.

Wie wichtig sind die unterschiedlichen Konventionen wirklich? Es gibt ein paar Untersuchungen, die Lesbarkeitsunterschiede zwischen verschiedenen Code-Layoutkonventionen finden. Im Vergleich zu den riesigen Lesbarkeitsunterschieden zwischen schlechtem und halbwegs brauchbarem Code sind diese Feinheiten aber vollkommen egal. Ob der Code 2, 4 oder 8 Leerzeichen weit eingerückt ist und ob Sie diese Leerzeichen mithilfe der Leertaste oder mit dem Tabulator erzeugen, sollte Sie als Leser dieses Buchs nicht länger als ein paar Minuten beschäftigen. Wichtig ist, dass Sie sich für eine dieser Möglichkeiten entscheiden und nicht mehrere Varianten nebeneinander verwenden. Denken Sie sich keinen eigenen Stil aus, sondern wählen Sie einen der existierenden. Er ist praxiserprobt, leichter zu verteidigen »und Ihre Leser kotzen weniger«[2]. Eine Übersicht finden Sie unter *de.wikipedia.org/wiki/Einrückungsstil*.

2 »Code Craft« von Pete Goodliffe (No Starch Press 2006), S. 26.

Konventionen im Team

Wenn man neu in einer Programmiersprache oder in Projekt ist, empfiehlt sich ein zweistufiges Verfahren:

1. Erst mal gucken, wie es die anderen machen.
2. Dann alles genauso machen.

Firmen- oder projektinterne Regelungen sind unter anderem sinnvoll, weil Programmierer sonst zu viel Zeit damit zubringen, den Code anderer Leute neu zu formatieren. Die Vorteile, die sich daraus ergeben, dass alle sich an dieselben Vorgaben halten, sind größer als der Umgewöhnungsärger der einzelnen Programmierer – obwohl dieser Ärger erheblich sein kann, wenn man in mehreren Projekten zugleich arbeitet, die alle ihre eigenen Konventionen haben. Jedenfalls bleibt das Gehirn flexibel, wenn man hin und wieder seine Formatierungsgewohnheiten ändert, die Ketchupmarke wechselt oder ein Land mit Linksverkehr aufsucht.

In der Informatik finden sich viele Menschen ein, die große Aufmerksamkeit für Details aufbringen. Das ist hilfreich beim Programmieren, kann aber zu einer gewissen Zwanghaftigkeit beim Umgang mit dem Code anderer Menschen führen. Wenn Sie selbst keinen unwiderstehlichen Drang dazu verspüren, Fremdcode umzuformatieren, tun Sie der Welt einen großen Gefallen, indem Sie es bleiben lassen. Und bringen Sie Geduld für Menschen auf, die nicht anders können.

Wenn fremder Code wirklich unbedingt umformatiert werden muss, darf das auf keinen Fall von Hand geschehen. Falls Ihr Texteditor keine automatische Formatkorrektur beherrscht, brauchen Sie ein separates Tool. Welches das sein könnte, hängt von der Programmiersprache ab, Stichwörter für die Suche wären »Code Beautifier« oder »Code Formatter«. Manche Texteditoren und Entwicklungsumgebungen haben einen Menüeintrag »Formatierung korrigieren« (oder ähnlich). Man kann dann einen Codeblock markieren und mithilfe dieses Menüs den Formatierungskonventionen entsprechend bereinigen lassen. Das klingt praktisch, birgt aber Sprengkraft, wenn das Programm nicht dieselben Vorstellungen von korrekter Formatierung hat wie die Kollegen.

Es ist weiterhin ungemein schlechter Stil, in einem unbeobachteten Moment dem Kollegen X per automatischer Korrektur an von ihm geschriebenen Code mitzuteilen, dass er sich nicht an Konventionen hält. Auch Versionskontrollsysteme und diff-Tools (siehe dazu den Abschnitt »Diff und Patch« in Kapitel 20) erkennen nicht unbedingt, dass zwei unterschiedlich formatierte Dateien inhaltlich gleich sind. Deshalb darf die automatische Formatierung in solchen Arbeitsumgebungen nur ganz am Anfang zum Einsatz kommen, bevor der Code zum ersten Mal ins Versionskontrollsystem eingecheckt wird. Und zu guter Letzt kann man die Regeln für die automatische Formatkorrektur häufig auch noch anpassen – wehe, wenn in einer Gruppe nicht alle mit den gleichen Einstellungen korrigieren.