



Weniger schlecht programmieren

Kathrin Passig &
Johannes Jander

Weniger schlecht programmieren

Kathrin Passig & Johannes Jander

O'REILLY®

Beijing · Cambridge · Farnham · Köln · Sebastopol · Tokyo

Die Informationen in diesem Buch wurden mit größter Sorgfalt erarbeitet. Dennoch können Fehler nicht vollständig ausgeschlossen werden. Verlag, Autoren und Übersetzer übernehmen keine juristische Verantwortung oder irgendeine Haftung für eventuell verbliebene Fehler und deren Folgen.

Alle Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt und sind möglicherweise eingetragene Warenzeichen. Der Verlag richtet sich im Wesentlichen nach den Schreibweisen der Hersteller. Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Alle Rechte vorbehalten einschließlich der Vervielfältigung, Übersetzung, Mikroverfilmung sowie Einspeicherung und Verarbeitung in elektronischen Systemen.

Kommentare und Fragen können Sie gerne an uns richten:

O'Reilly Verlag

Balthasarstr. 81

50670 Köln

E-Mail: komentar@oreilly.de

Copyright:

© 2013 by O'Reilly Verlag GmbH & Co. KG

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

Vorwort	XIII
----------------------	-------------

Teil 1: ~~Hallo Wels~~ Hallo Welt

1 Bin ich hier richtig?	3
2 Zwischen Hybris und Demut	7
Schwächen als Stärken	9
Richtiges muss nicht schwierig sein	12

Teil 2: Programmieren als Verständigung

3 Du bist wie die andern	17
4 Konventionen	19
Englisch oder nicht?	20
Die Steinchen des Anstoßes	23
Konventionen im Team	26
5 Namensgebung	29
Namenskonventionen	29
Von Byzanz über Konstantinopel nach Istanbul	31
Was Namen können sollten	33
Der Stoff, aus dem die Namen sind	40
Boolesche Variablen	50

Objektorientierte Programmierung	52
Datenbanken	53
Falsche Freunde	55
Wie es weitergeht	58
6 Kommentare	61
Mehr ist manchmal mehr	63
Zur äußeren Form von Kommentaren	64
Dokumentationskommentare	66
Wann und was soll man kommentieren?	67
Anzeichen, dass ein Kommentar eine gute Idee wäre	69
Problematische Kommentare	74
7 Code lesen	77
Muss ich wirklich?	77
Zuerst die Dokumentation lesen	79
Sourcecode ausdrucken	80
Zeichnen Sie schematisch auf, was einzelne Programmteile tun	81
Von oben nach unten, von leicht nach schwer	82
Lernen Sie Spurenlesen	82
80/20 ist gut genug (meistens)	83
Vergessen Sie die Daten nicht	84
Der Beweis ist das Programm	84
Gemeinsames Code-Lesen	85
8 Hilfe suchen	87
Der richtige Zeitpunkt	88
An der richtigen Stelle fragen	91
Die Anfrage richtig strukturieren	91
An den Leser denken	94
Nicht zu viel erwarten	95
Keine unbewussten Fallen stellen	96
Höflich bleiben – egal, was passiert	96
9 Lizenz zum Helfen	99
Der falsche Anlass	99
Die eigennützige Motivation	101
Die fehlende Einfühlung	102
Zu viel auf einmal	103
Antworten auf konkrete Fragen	105
Wenn Sie selbst keine Antwort wissen	106

Wenn Sie mit schlechteren Programmierern zusammenarbeiten	107
Schlechten Code gefasst ertragen	108
10 Überleben im Team	111
Ich war's nicht!	113
Der Bus-Faktor	114
Zusammenarbeit mit Anwendern	116
Zusammenarbeit mit Freiwilligen	117
Aussprache von Begriffen	117

Teil 3: Umgang mit Fehlern

11 Unrecht haben für Anfänger	123
Im Irrtum zu Hause	124
Fehlerforschung im Alltag	125
Der Hund hat die Datenbank gefressen!	126
Der gepolsterte Helm	127
12 Debugging I: Fehlersuche als Wissenschaft	131
Systematische Fehlersuche	133
Beobachtung	135
Was das Beobachten erschwert	136
Analyse und Hypothesenbildung	138
Was das Bilden von Hypothesen erschwert	138
Test der Hypothesen	139
Was das Testen von Hypothesen erschwert	140
13 Debugging II: Finde den Fehler	143
Fehlermeldungen sind unsere Freunde	143
Wer will da was von mir?	144
Diagnosewerkzeuge und -strategien	147
Wenn sonst nichts hilft	160
Wenn auch das nicht hilft	162
Die häufigsten Fehlerursachen schlechter Programmierer	163
14 Schlechte Zeichen oder Braune M&Ms	165
Zu große Dateien	166
Sehr lange Funktionen	167
Zu breite Funktionen	167

Tief verschachtelte if/then-Bedingungen	168
Mitten im Code auftauchende Zahlen	170
Komplexe arithmetische Ausdrücke im Code	170
Globale Variablen	171
Reparaturcode	172
Eigene Implementierung vorhandener Funktionen	173
Sonderfälle	174
Inkonsistente Schreibweisen	174
Funktionen mit mehr als fünf Parametern	174
Code-Duplikation	175
Zweifelhafte Dateinamen	176
Leselabyrinth	176
Ratlose Kommentare	176
Sehr viele Basisklassen oder Interfaces	177
Sehr viele Methoden oder Member-Variablen	177
Auskommentierte Codeblöcke und Funktionen	178
Browservorschriften	178
Verdächtige Tastaturgeräusche	179
15 Refactoring	181
Neu schreiben oder nicht?	182
Wann sollte man refakturieren?	183
Eins nach dem anderen	186
Code auf mehrere Dateien verteilen	191
Ein Codemodul in kleinere aufspalten	191
Nebenwirkungen entfernen	194
Code zusammenfassen	195
Bedingungen verständlicher gestalten	198
Die richtige Schleife für den richtigen Zweck	201
Schleifen verständlicher gestalten	201
Variablen kritisch betrachten	203
Refactoring von Datenbanken	204
Was man nebenbei erledigen kann	206
Ist das jetzt wirklich besser?	208
Wann man auf Refactoring besser verzichtet	208
Ein Problem und seine Lösung	211
16 Testing	213
Warum testen?	213
Testverfahren	214
Datenvalidierungen	220

Performancetests	222
Richtig testen	225
17 Warnhinweise	227
GET und POST	228
Zeichenkodierung	229
Zeitangaben	230
Kommazahlen als String, Integer oder Decimal speichern	232
Variablen als Werte oder Referenzen übergeben	233
Der schwierige Umgang mit dem Nichts	236
Rekursion	237
Usability	238
18 Kompromisse	241
Trügerische Tugenden	243
Absolution: Wann Bad Practice okay ist	247

Teil 4: Wahl der Mittel

19 Mach es nicht selbst	255
Der Weg zur Lösung	257
Bibliotheken	258
Umgang mit Fremdcode	261
Was man nicht selbst zu machen braucht	262
20 Werkzeugkasten	273
Editoren	274
Welche Programmiersprache ist die richtige?	275
REPL	279
Diff und Patch	282
Paketmanager	284
Frameworks	286
Entwicklungsumgebungen	289
21 Versionskontrolle	297
Alternativen	299
Arbeiten mit einem VCS	300
Konflikte auflösen	302
Welches Versionskontrollsystem?	303
Gute Ideen beim Arbeiten mit Versionskontrolle	305

Schlechte Ideen beim Arbeiten mit Versionskontrolle	306
Versionskontrollsysteme als Softwarebausteine	307
22 Command and Conquer – vom Überleben auf der Kommandozeile	309
Mehr Effizienz durch Automatisierung	310
Unsere langbärtigen Vorfahren	312
Windows	313
Was jeder Programmierer wissen sollte	313
Navigation	318
Dateien	318
Betrachten	321
Suchen und Finden	322
Ressourcen schonen	325
Zusammenarbeit	326
Zeitsteuerung	326
Editieren auf dem Server	328
Internet	328
Muss ich mir das alles merken?	330
Not the whole Shebang!	330
23 Objektorientierte Programmierung	333
Vorteile der objektorientierten Programmierung	335
Die Prinzipien objektorientierter Programmierung	337
Sinnvoller Einsatz von OOP	344
Nachteile und Probleme	347
Unterschiedliche Objektmodelle, je nach Sprache	348
Objektorientierte Programmierung und Weltherrschaftspläne	348
24 Aufbewahrung von Daten	351
Dateien	352
Versionskontrollsysteme	357
Datenbanken	357
25 Sicherheit	365
Wichtige Konzepte	366
Vor- und Nachteile der Offenheit	368
Vom Umgang mit Passwörtern	370
Authentifizierungsverfahren	371
SQL Injection und XSS – die Gefahren in User-Content	375
Weißer Listen sind besser als schwarze	380
Alle Regler nach links	381

Auch die Hintertür abschließen	383
Penetration Testing	384
Die Fehler der anderen	385
Sicherheit ist ein Prozess	386
26 Nützliche Konzepte	389
Exceptions	389
Error Handling	392
State und Statelessness	396
IDs, GUIDs, UUIDs	397
Sprachfamilien	399
Variablentypen	401
Trennung von Inhalt und Präsentation	404
Trennung von Entwicklungs- und Produktivserver	405
Selektoren	406
Namespaces	408
Scope von Variablen	410
Assertions	411
Transaktionen und Rollbacks	414
Hashes, Digests, Fingerprints	415
CRUD und REST	417
27 Wie geht es weiter?	419
Was ist ein guter Programmierer?	420
Zum Weiterlesen	421
Danksagungen	422
Index	423

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 untalenter 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 Komplizierten, 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, gammelligen 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 Welpresse 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 auszukomentieren.

- 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 Ahnungslosigkeit.

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 Unrecht haben 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.

~~Hallo Wels~~ Hallo Welt

Kapitel 1, *Bin ich hier richtig?*

Kapitel 2, *Zwischen Hybris und Demut*

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.

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-Regel² 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 Badensee 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.