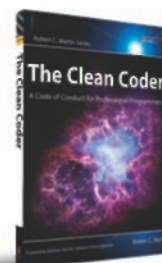


Clean Coder

➤ Verhaltensregeln für professionelle Programmierer



The Clean Coder:
A Code of Conduct for
Professional Programmers
(Robert C. Martin)

Clean Coder

Hier eine Auswahl:



Entwurfsmuster

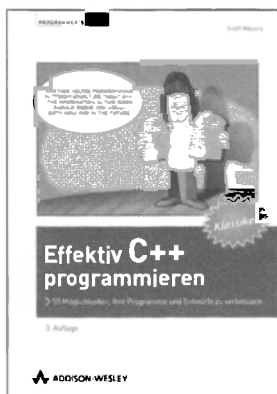
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

504 Seiten, 1 CD

€ 49,80 [D] € 51,20 [A]

ISBN 978-3-8273-3043-7

Die Autoren formulieren 23 Entwurfsmuster, benennen und beschreiben sie und erläutern ihre Verwendung. Diese Entwurfsmuster bieten einfache und prägnante Lösungen für häufig auftretende Programmieraufgaben. Sie erlauben die Wiederverwendung bewährter Lösungsstrategien und ermöglichen die Verständigung über die eigene Arbeit. Übersetzung aus dem Amerikanischen von Dirk Riehle. In der 6. Auflage erwartet Sie außerdem eine CD mit allen Quellen und zusätzlich 8 neuen Entwurfsmustern!



Effektiv C++ programmieren

Scott Meyers

336 Seiten

€ 34,80 [D] € 35,80 [A]

ISBN 978-3-8273-3078-9

Dieses Buch ist in 55 Themen gegliedert, die jeweils eine Maßnahme beschreiben, um besseren C++-Code zu schreiben. Jedes dieser Themen wird durch Beispiele illustriert. Mehr als die Hälfte des Inhalts dieser dritten Ausgabe ist neu, unter anderem die Kapitel über die Verwaltung von Ressourcen und die Verwendung von Templates. Die Themen aus der zweiten Ausgabe wurden sorgfältig überarbeitet, um die Anforderungen modernen Softwaredesigns widerzuspiegeln - darunter Ausnahmen, Entwurfsmuster und Multithreading.

Robert C. Martin



Clean Code

❏ Verhaltensregeln für professionelle Programmierer



ADDISON-WESLEY

An imprint of Pearson Education

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Bibliografische Information der deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <<http://dnb.d-nb.de>> abrufbar.

Die Informationen in diesem Produkt werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht.
Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt.
Bei der Zusammenstellung von Abbildungen und Texten wurde mit größter Sorgfalt vorgegangen.
Trotzdem können Fehler nicht vollständig ausgeschlossen werden.
Verlag, Herausgeber und Autoren können für fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen.
Für Verbesserungsvorschläge und Hinweise auf Fehler sind Verlag und Herausgeber dankbar.

Alle Rechte vorbehalten, auch die der fotomechanischen Wiedergabe und der Speicherung in elektronischen Medien.
Die gewerbliche Nutzung der in diesem Produkt gezeigten Modelle und Arbeiten ist nicht zulässig.

Fast alle Hardware- und Softwarebezeichnungen und weitere Stichworte und sonstige Angaben, die in diesem Buch verwendet werden, sind als eingetragene Marken geschützt.
Da es nicht möglich ist, in allen Fällen zeitnah zu ermitteln, ob ein Markenschutz besteht, wird das ®-Symbol in diesem Buch nicht verwendet.

Autorisierte Übersetzung der amerikanischen Originalausgabe: »The Clean Coder«.
Authorized translation from the english language edition, entitled The Clean Coder, by Robert C. Martin, published by Pearson Education publishing as Prentice Hall, Copyright © 2011.

10 9 8 7 6 5 4 3 2 1

13 12 11

ISBN 978-3-8273-3104-5

© 2011 by Addison-Wesley Verlag,
ein Imprint der Pearson Deutschland GmbH,
Martin-Kollar-Straße 10-12, D-81829 München/Germany
Alle Rechte vorbehalten
Lektorat: Brigitte Bauer Schiewek, bbauer@pearson.de
Übersetzung: Jürgen Dubau, www.dubau.net
Fachlektorat: Claudia Nölker, www.softpearls.de
Herstellung: Martha Kürzl-Harrison, mkuerzl@pearson.de
Korrektorat: Sandra Gottmann
Coverkonzeption und -gestaltung: Marco Lindenbeck, webwo GmbH, mlindenbeck@webwo.de
Satz: Reemers Publishing Services GmbH, Krefeld, www.reemers.de
Druck und Verarbeitung: Drukarnia Dimograf, Bielsko-Biala
Printed in Poland

Zwischen 1986 und 2000 arbeitete ich eng mit Jim Newkirk zusammen, einem Kollegen von Teradyne. Wir teilten die Leidenschaft fürs Programmieren und für sauberen Code. Wir verbrachten Abende, Nächte und ganze Wochenenden damit, mit unterschiedlichen Programmierstilen und Designtechniken herumzuspielen. Wir schmiedeten ständig neue Business-Ideen. Schließlich gründeten wir gemeinsam Object Mentor, Inc. Ich lernte vieles von Jim, während wir gemeinsam unsere Pläne ausheckten. Aber mit das Wichtigste war mir seine Einstellung zur *Arbeitsethik*. Darauf habe ich immer abgezielt, das wollte ich mir aneignen. Jim ist ein Profi. Ich bin stolz, mit ihm gearbeitet zu haben und ihn meinen Freund nennen zu dürfen.

Übersicht

Lob für den <i>Clean Coder</i>	17
Vorwort	19
Einführung	25
Danksagungen	29
Über den Autor	35
Auf dem Titelbild	37
Unverzichtbare Einführung	39
1 Professionalität	45
2 Nein sagen	61
3 Ja sagen	85
4 Programmieren	97
5 Test Driven Development	115
6 Praktizieren und Üben	123
7 Akzeptanztests	133
8 Teststrategien	151
9 Zeitmanagement	159
10 Schätzungen	171
11 Druck von außen	185
12 Teamwork	191
13 Teams und Projekte	201
14 Mentoring, Lehrzeiten und die Handwerkskunst	207
A Werkzeuge und Hilfsmittel	219
index	235

Inhalt

Lob für den <i>Clean Coder</i>	17
Vorwort	19
Einführung	25
Über dieses Buch	27
Bibliografie	28
Danksagungen	29
Über den Autor	35
Auf dem Titelbild	37
Unverzichtbare Einführung	39
1 Professionalität	45
1.1 Seien Sie vorsichtig, wonach Sie verlangen	46
1.2 Verantwortung übernehmen	46
1.3 Erstens: Richte keinen Schaden an.	48
1.3.1 Beschädige nicht die Funktion	49
1.3.2 Beschädige nicht die Struktur	51
1.4 Arbeitsethik	53
1.4.1 Sie sollten sich in Ihrem Bereich auskennen	54
1.4.2 Lebenslanges Lernen	56
1.4.3 Praxis	56
1.4.4 Teamwork	57
1.4.5 Mentorenarbeit	57
1.4.6 Sie sollten sich in Ihrem Arbeitsgebiet auskennen	58
1.4.7 Identifizieren Sie sich mit Ihrem Arbeitgeber bzw. Kunden.	58
1.4.8 Bescheidenheit	58
1.5 Bibliografie	59

2	Nein sagen	61
2.1	Feindliche Rollen	64
2.1.1	Was ist mit dem Warum?	66
2.2	Hoher Einsatz	67
2.3	Ein »Teamplayer« sein	68
2.3.1	Versuchen	70
2.3.2	Passive Aggression	72
2.4	Die Kosten eines Ja	74
2.5	Code unmöglich	81
3	Ja sagen	85
3.1	Verbindliche Sprache	87
3.1.1	So erkennt man mangelnde Selbstverpflichtung	88
3.1.2	Wie echte Selbstverpflichtung klingt	89
3.1.3	Zusammenfassung	91
3.2	Lernen, wie man »Ja« sagt	91
3.2.1	Die Kehrseite von »Ich versuch's mal«	92
3.2.2	Der Disziplin verpflichtet	92
3.3	Schlussfolgerung	95
4	Programmieren	97
4.1	Bereit sein	98
4.1.1	Code um drei Uhr früh	99
4.1.2	Sorgencode	100
4.2	Der Flow-Zustand	101
4.2.1	Musik	102
4.2.2	Unterbrechungen	103
4.3	Schreibblockaden	104
4.3.1	Kreativer Input	105
4.4	Debugging	105
4.4.1	Zeit zum Debuggen	108
4.5	Die eigene Energie einteilen	108
4.5.1	Wann man den Stift weglegen muss	109
4.5.2	Die Heimfahrt	109
4.5.3	Die Dusche	109

4.6	In Verzug sein	110
4.6.1	Hoffnung	110
4.6.2	Sich beeilen	110
4.6.3	Überstunden	111
4.6.4	Unlautere Ablieferung	111
4.6.5	Definieren Sie »fertig und erledigt«	112
4.7	Hilfe	112
4.7.1	Anderen helfen	112
4.7.2	Hilfe annehmen	113
4.7.3	Mentorenarbeit	114
4.8	Bibliografie	114
5	Test Driven Development	115
5.1	Die Geschworenen haben sich entschieden	117
5.2	Die drei Gesetze des TDD	117
5.2.1	Die Litanei der Vorteile	118
5.2.2	Die professionelle Option	121
5.3	Was TDD nicht ist	121
5.4	Bibliografie	122
6	Praktizieren und Üben	123
6.1	Etwas Hintergrund übers Üben	123
6.1.1	22 Nullen	124
6.1.2	Durchlaufzeiten	125
6.2	Das Coding Dojo	127
6.2.1	Kata	127
6.2.2	Waza	129
6.2.3	Randori	129
6.3	Die eigene Erfahrung ausbauen	130
6.3.1	Open Source	130
6.3.2	Ethisch handeln	130
6.4	Schlussfolgerung	131
6.5	Bibliografie	131

7 Akzeptanztests	133
7.1 Anforderungen der Kommunikation	133
7.1.1 Verfrühte Präzisierung	135
7.2 Akzeptanztests	138
7.2.1 Die »Definition of Done«	138
7.2.2 Kommunikation	141
7.2.3 Automatisierung	141
7.2.4 Zusätzliche Arbeit	143
7.2.5 Wer schreibt die Akzeptanztests und wann?	143
7.2.6 Die Rolle des Entwicklers	144
7.2.7 Verhandlungen über die Tests und passive Aggression	145
7.2.8 Akzeptanz- und Unit-Tests	146
7.2.9 GUIs und andere Komplikationen	147
7.2.10 Andauernde Integration	148
7.3 Schlussfolgerung	149
8 Teststrategien	151
8.1 Die Qualitätssicherung sollte keine Fehler finden	152
8.1.1 Die Qualitätssicherung gehört zum Team	152
8.2 Die Pyramide der Testautomatisierung	153
8.2.1 Unit-Tests	153
8.2.2 Komponententests	154
8.2.3 Integrationstests	155
8.2.4 Systemtests	156
8.2.5 Manuelle explorative Tests	156
8.3 Schlussfolgerung	157
8.4 Bibliografie	157
9 Zeitmanagement	159
9.1 Meetings	160
9.1.1 Absagen	161
9.1.2 Sich ausklinken	161
9.1.3 Tagesordnung und Ziel	162
9.1.4 Stand-up-Meetings	162
9.1.5 Planungstreffen zur Iteration	163
9.1.6 Retrospektive und Demo der Iteration	163
9.1.7 Auseinandersetzungen und Meinungsverschiedenheiten	163

9.2	Fokus-Manna	164
9.2.1	Schlaf	165
9.2.2	Koffein	165
9.2.3	Die Akkus aufladen	166
9.2.4	Muskelfokus.....	166
9.2.5	Input vs. Output	167
9.3	Zeitfenster und Tomaten	167
9.4	Vermeidung	168
9.4.1	Umkehrung der Prioritäten	168
9.5	Sackgassen	168
9.6	Morast, Moore, Sümpfe und andere Schlamassel.....	169
9.7	Schlussfolgerung	170
10	Schätzungen	171
10.1	Was eine Schätzung ist.....	173
10.1.1	Ein Commitment	173
10.1.2	Eine Schätzung	174
10.1.3	Implizierte Commitments.....	176
10.2	PERT	177
10.3	Aufgaben schätzen	179
10.3.1	Wideband Delphi	179
10.4	Das Gesetz der großen Zahlen.....	182
10.5	Schlussfolgerung	182
10.6	Bibliografie	183
11	Druck von außen	185
11.1	Druck vermeiden	187
11.1.1	Commitments	187
11.1.2	Sauber arbeiten	188
11.1.3	Verhalten in der Krise	188
11.2	Umgang mit Druck	189
11.2.1	Keine Panik	189
11.2.2	Kommunizieren Sie.....	189
11.2.3	Verlassen Sie sich auf Ihr Fachwissen.....	189
11.2.4	Hilfe holen	190
11.3	Schlussfolgerung	190

12 Teamwork	191
12.1 Programmierer kontra Menschen	193
12.1.1 Programmierer kontra Arbeitgeber	193
12.1.2 Programmierer kontra Programmierer	196
12.2 Kleinhirne	198
12.3 Schlussfolgerung	199
13 Teams und Projekte	201
13.1 Harmoniert es?	201
13.1.1 Das zusammengeschweißte Team	202
13.1.2 Aber wie managt man so etwas?	204
13.1.3 Das Dilemma des Product Owner	204
13.2 Schlussfolgerung	205
13.3 Bibliografie	205
14 Mentoring, Lehrzeiten und die Handwerkskunst	207
14.1 Der Grad des Versagens	207
14.2 Mentoring	208
14.2.1 Digi-Comp I – Mein erster Computer	208
14.2.2 Die ECP-18 in der Highschool	210
14.2.3 Unkonventionelles Mentoring	212
14.2.4 Schicksalsschläge	213
14.3 Die Lehrzeit	214
14.3.1 Die Lehrzeit bei der Software	215
14.3.2 Die Realität	216
14.4 Die Handwerkskunst	217
14.4.1 Menschen überzeugen	217
14.5 Schlussfolgerung	218
A Werkzeuge und Hilfsmittel	219
A.1 Tools	221
A.2 Quellcodekontrolle	221
A.2.1 Ein »Enterprise«-System der Quellcodekontrolle	221
A.2.2 Pessimistisches kontra optimistisches Locking	221
A.2.3 CVS/SVN	222
A.2.4 git	223

A.3	IDE/Editor	226
A.3.1	vi	226
A.3.2	Emacs	226
A.3.3	Eclipse/IntelliJ	226
A.3.4	TextMate	227
A.4	Issue-Tracking-Systeme	227
A.4.1	Bug-Zähler	228
A.5	Continuous Build	228
A.6	Tools für Unit-Tests	229
A.7	Tools für Komponententests	230
A.7.1	Die »Definition of Done«	230
A.7.2	FitNesse	230
A.7.3	Andere Tools	231
A.8	Tools für Integrationstests	231
A.9	UML/MDA	232
A.9.1	Die Details	232
A.9.2	Keine Hoffnung, keine Änderung	234
A.10	Schlussfolgerung	234
Index	235

Lob für den *Clean Coder*



»Mit seinem aktuellen Buch legt ‚Uncle Bob‘ Martin die Latte definitiv sehr hoch. Darin erläutert er, was er von einem professionellen Programmierer hinsichtlich der Verwaltung von Interaktionen, Zeitmanagement, Arbeiten unter Druck, Teamwork und der Wahl der eingesetzten Tools erwartet. Über TDD und ATDD hinaus erklärt Martin, was jeder Programmierer, der sich als Profi versteht, nicht nur wissen, sondern auch befolgen muss, um das junge Arbeitsfeld der Software-Entwicklung auszubauen.«

– Markus Gärtner
Senior Software Developer
it-agile GmbH
www.it-agile.de
www.shino.de

Lob für den Clean Coder

»Manche technischen Bücher inspirieren und informieren, manche erfreuen und unterhalten einen. Selten schafft es aber ein technisches Buch, all dies auf einmal zu machen. Die Bücher von Robert Martin haben das bei mir immer geschafft, und *Clean Coder* bildet dabei keine Ausnahme. Lesen Sie die Lektionen in diesem Buch, lernen und leben Sie sie, dann dürfen Sie sich mit Fug und Recht als Software-Profi bezeichnen.«

– George Bullock
Senior Program Manager
Microsoft Corp.

»Wird man nach dem Informatikstudium noch auf weitere Pflichtlektüre hingewiesen, dann müsste dieses Buch dazugehören. In der realen Welt verschwindet der eigene schlechte Code nicht, wenn das Semester vorüber ist, man bekommt keine 1 für einen Marathon-Programmiereinsatz in der Nacht vor dem Abgabetermin, und am schlimmsten: Man muss mit anderen Leuten klarkommen. So sind Coding-Gurus nicht notwendigerweise auch Profis. Der *Clean Coder* beschreibt die Reise zum Professionalismus ... und schafft das auf eine bemerkenswert unterhaltsame Weise.«

– Jeff Overbey
University of Illinois at Urbana-Champaign

»Der *Clean Coder* ist weitaus mehr als nur eine Sammlung von Regeln oder Richtlinien. Darin enthalten ist hart erarbeitete Weisheit und ein Wissen, das man normalerweise durch viele Jahre von Versuch und Irrtum erlangt, während man bei einem Meister seines Faches in die Lehre geht. Wenn Sie sich als Software-Profi bezeichnen, dann brauchen Sie dieses Buch.«

– R. L. Bogetti
Lead System Designer
Baxter Healthcare
www.RLBogetti.com

Vorwort



Sie haben sich für dieses Buch entschieden, also darf ich davon ausgehen, dass Sie ein Software-Profi sind. Das ist gut, das bin ich nämlich auch. Und da ich nun schon Ihre Aufmerksamkeit habe, will ich Ihnen berichten, warum ich mir dieses Buch vorgenommen bin.

Alles begann vor nicht allzu langer Zeit an einem gar nicht so weit entfernten Ort. Vorhang auf, Licht und Kamera an, Ton ab ...

Vor einigen Jahren arbeitete ich bei einer mittelgroßen Firma, die Produkte mit besonders strengen behördlichen Auflagen verkaufte. Das ist Ihnen sicherlich geläufig: Wir saßen in einem Großraumbüro in einem dreistöckigen Gebäude. Geschäftsführer und die höheren Ränge hatten eigene Büros, und es dauerte ungefähr eine Woche, um alle relevanten Personen für ein Meeting in den gleichen Raum zu bekommen.

Vorwort

Wir operierten in einem hart umkämpften Marktsegment, als die Regierung ein neues Produkt freigab.

Plötzlich hatten wir eine völlig neue Zielgruppe potenzieller Kunden. Wir brauchten nur dafür zu sorgen, dass sie unser Produkt kauften. Das hieß, wir mussten unsere Unterlagen bis zu einem bestimmten Datum bei der Behörde einreichen, zu einem anderen Termin ein Assessment-Audit bestehen und zu einem dritten Termin auf den Markt gehen.

Immer wieder betonte unser Management nachdrücklich, wie wichtig diese Termine seien. Ein kleiner Fehler, und die Regierung würde uns für ein ganzes Jahr vom Markt sperren, und wenn sich die Kunden nicht gleich vom ersten Tag an sich bei uns anmelden konnten, dann würden sie sich bei anderen Anbietern registrieren, und wir wären aus dem Rennen.

Es war die Art von Umgebung, über die sich manche Leute beschwerten und andere betonten, dass hier »der Druck herrscht, der aus Kohle Diamanten formt«.

Ich war der technische Projektmanager, aus der Entwicklungsabteilung heraus dazu ernannt. Meine Verantwortung bestand darin, die Website an besagtem Tag online zu bringen, damit sich die potenziellen Kunden Informationen und vor allem Registrierungsformulare herunterladen konnten. Mein Partner bei diesem Vorhaben war der fürs Business zuständige Projektmanager, den ich hier mal Joe nennen möchte. Joes Rolle war, auf der »anderen« Seite zu arbeiten, also mit der Verkaufs- und Marketingabteilung und den nichttechnischen Anforderungen. Er war auch der Typ, von dem der Kommentar über den Druck, »der aus Kohle Diamanten formt«, stammte.

Wenn Sie sich mit der unternehmerischen Welt Amerikas auskennen und darin gearbeitet haben, dann kennen Sie wahrscheinlich all die Schuldzuweisungen, die anderen die Verantwortung zuschieben, und den Widerwillen gegen Arbeit – alles völlig normal. Unsere Firma hatte für das Problem mit Joe und mir eine interessante Lösung parat.

Es war ein bisschen wie mit Batman und Robin, wie wir unsere Sachen erledigen sollten. In einer bestimmten Büroecke traf ich mich täglich mit dem technischen Team. Wir erstellten jeden Tag den Fahrplan neu, erarbeiteten den kritischen Pfad und räumten anschließend jedes mögliche Hindernis auf diesem kritischen Pfad weg. Wenn jemand bestimmte Software brauchte, besorgten wir sie. Wenn jemand »liebend gerne« die Firewall konfigurieren wollte, aber »ach du meine Güte, es ist ja schon wieder Zeit für die Mittagspause« rief, dann ließen wir ihm sein Mittagessen anliefern. Wenn jemand an unserem Configuration Ticket arbeiten wollte, aber andere Prioritäten aufgetragen bekommen hatte, dann wandten wir uns an seinen Vorgesetzten.

Dann an den Manager.

Dann an den Geschäftsführer.

Wir sorgten dafür, dass alles fluppte.

Es wäre etwas übertrieben zu sagen, dass wir mit Stühlen herumwarfen, brüllten und tobten, aber wir setzten aus unserer Werkzeugkiste jedes einzelne Instrument ein, um alles auf die Reihe zu bekommen. Wir erfanden nebenher ein paar neue Instrumente und Techniken und machten das auf eine ethische Weise, auf die ich noch heute stolz bin.

Ich betrachtete mich selbst als Teammitglied, das sich keinen Zacken aus der Krone bricht, im Notfall auf die Schnelle auch mal eine SQL-Anweisung zu schreiben oder mit Kollegen zu zweit zu programmieren, damit der Code ablieferbar wird. Zu jener Zeit dachte ich über Joe auch auf diese Weise: Ich betrachtete ihn als Mitglied des Teams und nicht darüberstehend.

Schließlich musste ich erkennen, dass Joe diese Meinung nicht teilte. Das war für mich ein sehr trauriger Tag.

Es war Freitag, ein Uhr nachts. Die Website sollte planmäßig sehr früh am folgenden Montag live gehen.

Wir waren fertig. *FERTIG*. Alle Systeme schnurrten wie Kätzchen, wir waren bereit. Ich ließ das gesamte Tech-Team für das finale Scrum-Meeting zusammenkommen, und nun musste nur noch der sprichwörtliche Schalter umgelegt werden. Mehr als einfach bloß das technische Team hatten wir außerdem auch die Leute aus der Marketing-Abteilung, also die Product Owner, bei uns.

Wir waren stolz. Es war ein toller Moment.

Dann kam Joe vorbei.

Er sagte etwas wie: »Schlechte Nachrichten. Die Rechtsabteilung hat die Registrierungsformulare noch nicht fertig. Also können wir noch nicht live gehen.«

Das war kein sonderlich großes Problem. Die ganze Projektlaufzeit schon waren wir immer wieder mal von der einen oder anderen Sache aufgehalten worden und zogen die Batman-und-Robin-Masche aus dem Effe durch. Ich war bereit, und meine Antwort lautete im Wesentlichen: »Okay, Partner, machen wir uns noch mal an die Arbeit. Die Rechtsabteilung ist im zweiten Stock, nicht wahr?«

Dann wurde die Geschichte merkwürdig.

Anstatt mir zuzustimmen, fragte Joe: »Wovon redest du da, Matt?«

Ich entgegnete: »Ach, du weißt schon. Unseren üblichen Auftritt. Wir reden hier über vier PDFs, oder? Die sind doch schon fertig, und die Rechtsfritzen müssen sie nur abnicken, oder? Komm, wir hängen ein bisschen an ihren Schreibtischen rum, schauen sie ein wenig böse an und bringen das Ding hier endlich in trockene Tücher!«

Vorwort

Joe stimmte meiner Einschätzung nicht zu, sondern antwortete: »Wir gehen einfach erst Ende nächster Woche live. Keine große Sache.«

Sie können sich wahrscheinlich ausmalen, wie die restliche Unterhaltung weiterging. Sie verlief etwa wie folgt:

Matt: »Aber warum denn? Die kriegen das doch in ein paar Stunden hin.«

Joe: »Vielleicht dauert das aber auch länger.«

Matt: »Aber die haben doch noch das *ganze Wochenende*! Das ist doch eine Menge Zeit. Komm, wir ziehen das durch!«

Joe: »Matt, das sind Profis. Wir können sie nicht einfach mit bösen Blicken dazu zwingen, dass sie ihr Privatleben für unser kleines Projekt opfern.«

Matt: (holt Luft) »Joe, was haben wir denn deiner Meinung nach in den vergangenen vier Monaten mit dem Entwicklerteam gemacht?«

Joe: »Sicher, aber die sind eben Profis.«

Pause.

Tief Luft holen.

Was. Hat. Joe. Gerade. Gesagt?

Zu jener Zeit war ich der Ansicht, dass das technische Team aus Profis besteht, und zwar im besten Sinne des Wortes.

Wenn ich es mir im Nachhinein noch einmal überlege, bin ich mir da nicht mehr so sicher.

Schauen wir uns diese Batman-und-Robin-Masche noch einmal an, und zwar aus einer anderen Perspektive. Ich dachte, dass ich das Team durch Ermahnungen zu Höchstleistungen anspornte, aber ich hegte den Verdacht, dass Joe ein Spiel spielte mit der impliziten Annahme, das technische Personal stelle seinen Gegner dar. Denken Sie mal drüber nach: Warum war es nötig, herumzurennen, gegen Stühle zu treten und Kollegen zu bearbeiten?

Hätten wir nicht eher das Team fragen sollen, wann es fertig ist, um eine zuverlässige Aussage zu bekommen? Dieser Antwort hätten wir dann Glauben geschenkt und uns bei diesem Glauben nicht die Finger verbrannt.

Sicher sollten wir als Profis das so machen ... und gleichzeitig auch wiederum nicht. Joe vertraute unseren Antworten nicht und fühlte sich beim Mikromanagement des Teams unwohl. Gleichzeitig vertraute er dem Team der Rechtsabteilung und war nicht gewillt, das Mikromanagement auf sie auszudehnen.

Worum geht es hier eigentlich?

Das Team der Rechtsabteilung hatte einen Professionalismus auf eine Weise demonstriert, wie es das technische Team nicht vermochte.

Irgendwie hatte eine andere Gruppe es geschafft, Joe davon zu überzeugen, dass sie keinen Babysitter brauchte, dass sie keine Spielchen spielte und erwartete, dass sie auf Augenhöhe zu behandeln sei und als Gleichberechtigte respektiert werden wollten.

Nein, ich glaube nicht, dass das etwas mit ein paar schicken Zertifikaten an der Wand zu tun hat oder ein paar Extraseminaren am College, obwohl diese zusätzlichen Jahre am College einem vielleicht implizit ein soziales Training ermöglichen, wie man sich zu verhalten hat.

Seit jenem Tag vor vielen Jahren habe ich mich gefragt, wie der technische Berufsstand sich ändern müsste, damit er als Profis betrachtet wird.

Oh, ich habe so meine paar Ideen. Ich habe ein bisschen gebloggt, viel gelesen, es geschafft, meine eigene Arbeits- und Lebenssituation zu verbessern, und ein paar anderen geholfen. Und doch kannte ich kein Buch, in dem man einen ganzen Plan vorgestellt bekommt und das einem die ganze Sache explizit erläutert.

Dann bekam ich eines Tages aus heiterem Himmel die Anfrage, den frühen Entwurf eines Buches zu prüfen – und zwar genau jenes, das Sie hier in Händen halten.

Dieses Buch erläutert Ihnen Schritt für Schritt, wie Sie sich als Profi präsentieren und mit anderen umgehen. Nicht mit banalen Klischees, ohne Appelle auf Schriftstücken, sondern was Sie machen können und wie Sie es machen können.

In manchen Fällen deklinieren die Beispiele alles Wort für Wort durch.

Manche dieser Beispiele haben Antworten, Gegenreden, Verdeutlichungen, manchmal sogar Ratschläge dafür, falls die andere Person versucht, Sie »einfach zu ignorieren«.

Achtung, aufgepasst: Joe kommt noch einmal auf die Bühne, diesmal von links.

Nun sind wir wieder bei der Firma BigCo, Joe und ich, noch einmal bei dem großen Projekt der Website-Konvertierung.

Nur diesmal stellen Sie sich vor, dass es ein wenig anders ist.

Anstatt sich davor zu drücken, sich für etwas zu »committen«, geht das Tech-Team wirklich diese Selbstverpflichtungen ein. Anstatt sich davor zu scheuen, Einschätzungen abzugeben oder anderen die Planung zu überlassen (und sich hinterher darüber zu beschweren), organisiert sich das Tech-Team tatsächlich selbst und macht echte Commitments.

Vorwort

Nun stellen Sie sich bitte vor, dass das Team wirklich zusammenarbeitet. Wenn die Programmierer von Operationen blockiert werden, hängen sie sich ans Telefon, und der Sysadmin fängt tatsächlich an zu arbeiten.

Wenn Joe vorbeikommen will, um Druck zu machen, damit am Ticket 14321 gearbeitet wird, braucht er das nicht: Er kann sehen, dass der Datenbankadministrator fleißig arbeitet und nicht im Web surft. Entsprechend belastbar sind die Einschätzungen, die er vom Team bekommt, klar und konsistent, und er hat nicht den Eindruck, das Projekt läge in seiner Priorität irgendwo zwischen Mittagessen und dem Abrufen der Mails. Alle Tricks und Versuche, den Zeitplan zu manipulieren, erbringen kein »Wir versuchen das mal«, sondern führen zu der Aussage: »Dazu haben wir uns committet. Wenn Sie sich eigene Ziele suchen wollen, können Sie das gerne machen.«

Nach einer Weile würde ich mal davon ausgehen, dass Joe beginnt, das technische Team als ... tja, Profis zu betrachten. Und er hätte recht damit.

Mit welchen Schritten Sie Ihr Verhalten vom Techniker zum Profi transformieren können? Die finden Sie im weiteren Verlauf dieses Buches.

Willkommen zum nächsten Schritt in Ihrer Karriere. Ich schätze mal, dass er Ihnen gefallen wird.

*– Matthew Heusser
Software Process Naturalist*

Einführung



Um 11:39 Uhr Ortszeit zerbrach am 28. Januar 1986, nur 73,124 Sekunden nach dem Start, die Raumfähre Challenger aufgrund mangelhafter Dichtungsringe an einer der seitlichen Feststoffraketen (Booster). Sieben tapfere Astronauten, darunter die High-school-Lehrerin Christa McAuliffe, kamen dabei ums Leben. Der Ausdruck auf dem Gesicht ihrer Mutter, als sie zuschauen musste, wie ihre Tochter 15 Kilometer über ihr verglühte, verfolgt mich bis zum heutigen Tag.

Die Challenger zerbrach, weil heiße Abgase der fehlerhaften Feststoffrakete zwischen den Segmenten des Rumpfs austraten und über den externen Treibstofftank strömten statt durch die große Düse am Heck. Der Hauptflüssigwasserstofftank schlug leck und entzündete den Treibstoff. Dadurch wurde der Tank nach vorne gejagt und krachte in den Flüssigwasserstofftank darüber. Gleichzeitig löste sich der Booster von seiner hinteren Verstrebung und rotierte um die Achse der Vorderstrebe. Die Spitze durchbrach den Flüssigwasserstofftank. Diese abweichenden Kraftvektoren sorgten dafür, dass

Einführung

sich die gesamte Raumfähre, die deutlich über Mach 1,5 beschleunigt hatte, gegen die Luftströmung drehte. Die aerodynamischen Kräfte zerrissen blitzartig alles in kleine Fetzen.

Zwischen den kreisförmigen Segmenten der Feststoffrakete befanden sich zwei konzentrische Kunststoffringe, die sogenannten O-Ringe. Beim Zusammenfügen der Segmente wurden die O-Ringe aufeinandergepresst und dichteten so alles ab, damit die Abgasflammen der Rakete nicht nach außen dringen konnten.

Doch am Abend vor dem Start sank die Temperatur auf der Startrampe auf minus 8 Grad C, also 6 Grad weniger als die für die O-Ringe festgelegte Minimaltemperatur und 18 Grad niedriger als bei allen früheren Startvorgängen. Infolgedessen reduzierte sich auch die Elastizität der O-Ringe, um die heißen Abgase noch ausreichend blockieren zu können. Beim Zünden der Booster gab es einen Druckimpuls, als sich die heißen Gase rapide sammelten. Die Segmente der Booster dehnten sich nach außen und verringerten den Druck auf die O-Ringe. Weil die O-Ringe so unelastisch waren, schlossen sie nicht mehr dicht ab, sodass heiße Gase durchschlagen konnten und die O-Ringe in einem kreisförmigen Bereich von 70 Grad verdampfen ließen.

Die Ingenieure von Morton Thiokol hatten die Feststoffraketen entworfen und wussten, dass es Probleme mit den O-Ringen gegeben hatte. Diese Probleme hatten sie den Managern bei Morton Thiokol und der NASA bereits vor sieben Jahren gemeldet. Tatsächlich waren die O-Ringe auch bei früheren Startvorgängen auf ähnliche Weise beschädigt worden, aber nicht so sehr, dass es zu einer Katastrophe geführt hätte. Je geringer die Außentemperaturen beim Startvorgang waren, desto größer waren die Beschädigungen. Die Ingenieure hatten eine Reparatur für das Problem konstruiert, aber dessen Implementierung war schon seit Langem verzögert worden.

Die Ingenieure hatten den Verdacht, dass die O-Ringe sich bei Kälte versteiften. Sie wussten auch, dass die Temperaturen beim Start der Challenger so gering waren wie noch nie bei irgendeinem anderen Start und deutlich im gefährlichen Bereich lagen. Auf den Punkt gebracht: Die Ingenieure *wussten*, dass das Risiko zu groß war. Die Ingenieure handelten dieser Erkenntnis zufolge: Sie schrieben Memos, in denen sie mit allem Nachdruck Alarm gaben. Sie ermahnten die Manager von Thiokol und der NASA, den Start unbedingt abzublasen. In einem elf Stunden dauernden Meeting, das wenige Stunden vor dem Start abgehalten wurde, präsentierten die Ingenieure ihre besten Daten. Sie tobten, redeten mit Engelszungen und protestierten. Doch am Ende wurden sie von den Managern ignoriert.

Als der Startzeitpunkt näher rückte, weigerten sich einige Ingenieure, die Übertragung der Bilder anzuschauen, weil sie eine Explosion auf der Startrampe befürchteten. Doch als die Challenger graziös in den Himmel stieg, begannen sie sich zu entspannen. Wenige Augenblicke vor der Katastrophe meinte einer, als man zusah, wie das Vehikel auf Mach 1 beschleunigte, man sei »noch einmal mit einem blauen Auge davongekommen«.

Trotz aller Proteste, Memos und Mahnungen der Ingenieure glaubten die Manager, dass sie es besser wussten. Sie meinten, dass die Ingenieure übertrieben reagierten. Sie trauten den Daten der Ingenieure und ihren Schlussfolgerungen nicht. Sie zogen den Start durch, weil sie sich unter immensem finanziellen und politischen Druck befanden. Sie *hofften* einfach, dass alles gut gehen würde.

Diese Manager waren nicht einfach nur töricht, sondern kriminell. Das Leben von sieben tapferen Männern und Frauen und die Hoffnungen einer Generation, die sich auf die Reisen ins All freute, wurden an diesem kalten Morgen zerschmettert, weil die Manager ihre eigenen Ängste, Hoffnungen und Intuitionen über die Meinung ihrer eigenen Experten gestellt hatten. Sie trafen eine Entscheidung, zu der sie nicht berechtigt waren. Sie bemächtigten sich der Autorität jener Menschen, die *tatsächlich* Bescheid wussten: und das waren die Ingenieure.

Aber was war mit den Ingenieuren? Sicherlich haben die Ingenieure gemacht, was ihre Aufgabe war. Sie haben ihre Manager informiert und hart für ihre Position gekämpft. Sie durchliefen die entsprechenden Kommunikationskanäle und hielten sich an alle richtigen Protokolle. Sie taten, was sie konnten, *innerhalb* des Systems – und trotzdem wurden sie von den Managern ausgeschaltet. So scheint es also, dass sich die Ingenieure keine Schuld zuschreiben lassen müssen.

Aber manchmal frage ich mich, ob einige dieser Ingenieure nachts wach liegen, weil sie das Bild der Mutter von Christa McAuliffe nicht schlafen lässt, und sich wünschen, sie hätten besser Dan Rather Bescheid gegeben, dem Nachrichtensprecher von *CBS News*.

Über dieses Buch

In diesem Buch geht es um professionelle Software-Entwicklung. Es enthält eine Menge pragmatischer Ratschläge, um verschiedene Fragen zu beantworten, z.B.:

- Was ist ein Software-Profi?
- Wie verhält sich ein Profi?
- Wie geht ein Profi mit Konflikten, knappen Zeitplänen und unvernünftigen Managern um?
- Wann und wie sollte ein Profi »Nein« sagen?
- Wie geht ein Profi mit Druck um?

Doch in den pragmatischen Ratschlägen dieses Buches versteckt werden Sie eine bestimmte Haltung entdecken. Es ist eine Haltung der Aufrichtigkeit, der Ehre, des Selbstrespekts und des Stolzes. Es ist die Bereitschaft, die schreckliche Verantwortung zu akzeptieren, Handwerker und Ingenieur gleichzeitig zu sein. Zu dieser Verantwortung gehört es, gut und sauber zu arbeiten. Dazu gehört, gut zu kommunizieren und

Einführung

zuverlässige Einschätzungen abzugeben. Dazu gehört, die eigene Zeit gut zu planen und zu managen sowie schwierige Chancen-Risiken-Abwägungen vorzunehmen.

Doch in dieser Verantwortung steckt noch etwas anderes, eine sehr beängstigende Sache: Als Ingenieur verfügen Sie über ein tiefgehendes Wissen über Ihre Systeme und Projekte, an die kein anderer Manager herankommt. Dieses Wissen wird von der *Verantwortung zum Handeln* begleitet!

Bibliografie

[McConnell87]: Malcolm McConnell, *Challenger »A Major Malfunction«*, New York, NY: Simon & Schuster, 1987

[Wiki-Challenger]: »Das Challenger-Unglück«, http://de.wikipedia.org/wiki/Challenger-Katastrophe#Das_Challenger-Ungl.C3.BCck



Meine Karriere besteht aus einer Serie von Gemeinschaftsproduktionen und Projekten. Obwohl ich viele private Träume und Bestrebungen hatte, fand ich offenbar stets jemanden, mit dem ich sie teilen konnte. In dieser Hinsicht fühle ich mich ein wenig wie ein Sith: »Immer zwei es sind.«

Die erste Zusammenarbeit, die ich als professionell betrachten könnte, bestand mit John Marchese. Wir waren 13 und hatten große Pläne, gemeinsam Computer zu bauen. Er hatte das Geschick und ich die Kenne. Ich zeigte ihm, wo ein Draht zu verlöten war, und er lötete ihn an. Ich zeigte ihm, wo ein Relais eingebaut werden musste, und er schraubte es fest. Das machte einen Riesenspaß, und wir verbrachten Hunderte von Stunden damit. Tatsächlich erschufen wir eine beachtliche Menge beeindruckend aussehender Objekte mit Relais, Schaltern, Lämpchen und sogar Fernschreiber. Natürlich hat keins von diesen Geräten tatsächlich irgendwas gemacht, aber sie waren sehr beeindruckend, und wir arbeiteten sehr hart daran. An John: Vielen Dank!

Danksagungen

In meinem ersten Jahr an der Highschool lernte ich in meiner Deutschstunde Tim Conrad kennen. Tim war *clever*. Als wir uns als Team vornahmen, einen Computer zu bauen, war er das Gehirn und ich der mit dem Lötkolben. Er brachte mir Elektronik bei und stellte mir einen PDP-8 vor. Wir bauten dann tatsächlich aus Basiskomponenten einen funktionierenden elektronischen 18-Bit-Binärrechner zusammen. Damit konnte man addieren, subtrahieren, multiplizieren und dividieren. Damit brachten wir ein Jahr zu und werkten jedes Wochenende und in den ganzen Ferien. Wir arbeiteten wie wild. Am Ende funktionierte er wirklich toll. An Tim: Vielen Dank!

Gemeinsam mit Tim lernte ich, wie man Computer programmiert. Das war 1968 kein Kinderspiel, aber wir schafften es. Wir besorgten uns Bücher über PDP-8 Assembler, Fortran, Cobol, PL/1 usw. Wir verschlangen sie. Wir schrieben Programme, bei denen wir nie die Hoffnung hegen konnten, sie einmal laufen zu lassen, da wir keinen Zugang zu einem Computer hatten. Aber wir schrieben sie einfach trotzdem – aus reinem Spaß an der Freud.

An unserer Highschool gab es in der zehnten Klasse einen Computerkurs. Dort wurde eine ASR-33 Teletype an ein Einwahlmodem mit 110 Baud angeschlossen. Die Schule hatte einen Account auf dem mit Time-Sharing arbeitenden System Univac 1108 beim Illinois Institute of Technology. Tim und ich wurden sofort die De-facto-Operatoren dieses Rechners. Kein anderer reichte an uns heran.

Die Verbindung mit dem Modem wurde aufgebaut, indem man den Hörer abnahm und die Nummer wählte. Wenn man das andere Modem quiekend antworten hörte, musste man den »ORIG«-Knopf auf dem Fernschreiber drücken, damit das Ausgangsmodem mit seinem eigenen Quieken antwortete. Dann konnte man den Hörer auflegen, und die Datenverbindung stand.

Beim Telefon war an der Wählscheibe eine Sperre. Nur die Lehrer hatten den Schlüssel dazu. Aber das war egal, weil wir herausbekommen hatten, dass man bei einem Telefon auch durch Drücken der Telefongabel die Nummer wählen konnte. Ich war Schlagzeuger, hatte also gute Reflexe und gutes Timing. Ich konnte dieses Modem trotz der Sperre an der Wählscheibe in weniger als zehn Sekunden anwählen.

Im Computerlabor hatten wir zwei Teletype-Fernschreiber. Einer war online, der andere offline. Die Studierenden nutzten beide, um ihre Programme zu schreiben. Die Studenten tippen ihre Programme auf den Teletypes, während der Papierlocher angeschlossen war. Jede Tastatureingabe wurde auf Band gelocht. Die Studenten schrieben ihre Programme in IITran, einer bemerkenswert leistungsfähigen Interpretersprache. Hinterher warfen sie die Papierbänder in einen Abfallkorb neben den Geräten.

Nach der Schule wählten Tim und ich den Computer an (natürlich durch Drücken der Gabel), luden die Bänder in das IITran-Batchsystem und legten dann auf. Bei zehn Zeichen pro Sekunde ging diese Prozedur nicht sonderlich flott vonstatten. Etwa eine Stunde später riefen wir erneut an und bekamen die Ausgabe, wieder mit zehn Zeichen pro Sekunde. Das Teletype-Gerät trennte die Listings der Studenten nicht durch Auswerfen der Seiten. Es druckte einfach eine nach der anderen aus, und wir mussten sie mit der Schere zerschneiden, das Eingabepapierband mit einer Büroklammer an die Listings klemmen und sie dann in den Ausgabekorb legen.

Tim und ich waren Herren und Götter dieses Prozesses. Sogar die Lehrer ließen uns in Ruhe, wenn wir allein in diesem Raum arbeiteten. Wir erledigten ihren Job, und das war ihnen klar. Sie haben uns niemals darum gebeten, sie haben es uns auch nie erlaubt. Sie gaben uns nie den Schlüssel zum Telefon. Wir kamen einfach rein, und sie gingen raus – und ließen uns an einer sehr langen Leine laufen. An meine Mathelehrer Mr. McDermit, Mr. Fogel und Mr. Robien: Vielen Dank!

Dann, nachdem die ganzen Hausaufgaben erledigt waren, konnten wir spielen. Wir schrieben ein Programm nach dem anderen, um alle möglichen verrückten und komischen Sachen zu machen. Wir schrieben Programme, die Kreise und Parabeln in ASCII auf eine Teletype zeichneten. Wir schrieben Programme für Zufallsbewegungen und Zufallsgeneratoren für Wörter. Wir berechneten 50 Fakultäten bis zur letzten Zahl. Wir verbrachten viele, viele Stunden damit, Programme zu schreiben und sie ans Laufen zu bringen.

Zwei Jahre später wurden Tim, unser *compadre* Richard Lloyd und ich als Programmierer bei ASC Tabulating in Lake Bluff, Illinois, eingestellt. Damals waren Tim und ich 18. Wir beschlossen, dass das College reine Zeitverschwendung sei und wir sofort mit unserer Laufbahn beginnen sollten. Dort trafen wir Bill Hohri, Frank Ryder, Big Jim Carlin und John Miller. Sie gaben uns Jungspunden die Chance zu lernen, worum es beim professionellen Programmieren ging. Diese Erfahrung war weder positiv noch negativ, aber definitiv sehr aufschlussreich und bildend. An euch alle und an Richard, der einen Großteil dieses Prozesses katalysierte und vorantrieb: Vielen Dank!

Als ich mit 20 die Firma verlassen musste und zusammenklappte, hatte ich einen Einsatz bei meinem Schwager, wo ich Rasenmäher reparierte. Ich war dabei so mies, dass er mich rauswerfen musste. Danke, Wes!

Etwa ein Jahr später landete ich arbeitstechnisch bei der Outboard Marine Corporation. Zu jener Zeit war ich verheiratet, und wir erwarteten unser Baby. Sie feuerten mich ebenfalls. Danke, John, Ralph und Tom!

Danksagungen

Dann fing ich bei Teradyne an, wo ich Russ Ashdown, Ken Finder, Bob Copithorne, Chuck Studee und CK Srithran (jetzt Kris Iyer) traf. Ken war mein Chef. Chuck und CK waren meine Kollegen. Ich habe von allen total viel gelernt. Danke, Leute!

Dann kam Mike Carew. Bei Teradyne wurden wir zwei beide das dynamische Duo. Wir schrieben gemeinsam mehrere Systeme. Wenn etwas zu erledigen war und schnell umgesetzt werden musste, bekamen Mike und ich den Auftrag. Wir hatten eine Menge Spaß zusammen. Danke, Mike!

Jerry Fitzpatrick arbeitete ebenfalls bei Teradyne. Wir lernten uns beim Spielen von *Dungeons & Dragons* kennen, doch schon bald bildeten wir ein Team. Wir schrieben Software auf einem Commodore 64, um die User von D&D zu unterstützen. Wir riefen bei Teradyne ein neues Projekt namens »Der elektronische Rezeptionist« ins Leben. Wir arbeiteten mehrere Jahre zusammen, er wurde ein großartiger Freund – und das ist er immer noch. Danke, Jerry!

Ich verbrachte während meiner Arbeit für Teradyne ein Jahr in England. Dort arbeitete ich mit Mike Kergozou zusammen. Als Team heckten wir alle möglichen Sachen aus, obwohl die meisten dieser Geschichten mit Fahrrädern und Pubs zu tun hatten. Aber er war ein engagierter Programmierer, sehr auf Qualität und Disziplin konzentriert (dem würde er wahrscheinlich widersprechen). Danke, Mike!

Nach meiner Rückkehr 1987 aus England tat ich mich mit Jim Newkirk zusammen. Wir verließen beide (mehrere Monate nacheinander) Teradyne und kamen bei einem Start-up namens Clear Communications unter. Wir verbrachten mehrere Jahre zusammen und schufteten in Erwartung der vielen Millionen, die niemals kamen. Aber mit unserem Pläneschmieden hörten wir nie auf. Danke, Jim!

Am Ende gründeten wir gemeinsam Object Mentor. Jim ist ein Mensch, der sehr direkt, diszipliniert und konzentriert ist – und diese Eigenschaften habe ich in einer solchen Fülle bei anderen nie gefunden. Ich darf das Privileg mein Eigen nennen, mit ihm arbeiten zu dürfen. Er lehrte mich so viele Dinge, dass ich sie hier nicht aufzählen kann. Stattdessen habe ich ihm dieses Buch gewidmet.

Es gibt so viele andere, mit denen ich gemeinsame Projekte durchgeführt habe, mit denen ich zusammengearbeitet habe, so viele Menschen, die mein Berufsleben geprägt haben: Lowell Lindstrom, Dave Thomas, Michael Feathers, Bob Koss, Brett Schuchert, Dean Wampler, Pascal Roy, Jeff Langr, James Grenning, Brian Button, Alan Francis, Mike Hill, Eric Meade, Ron Jeffries, Kent Beck, Martin Fowler, Grady Booch und eine endlose Liste anderer Menschen. Euch allen gebührt mein herzlicher Dank.

Der großartigste Kollaborateur meines Lebens ist natürlich meine liebe Frau Ann Marie. Ich habe sie geheiratet, als ich 20 war, drei Tage nach ihrem 18. Geburtstag. Seit 38 Jahren ist sie nun meine stete Begleiterin, mein Ruder und Segel, meine Liebe und mein Leben. Ich freue mich so auf die nächsten vier Jahrzehnte mit ihr.

Und nun sind meine Kinder meine Kollaborateure und Projektpartner. Ich arbeite eng mit meiner ältesten Tochter Angela zusammen, meine reizende Glücke und unerschrockene Assistentin. Sie sorgt dafür, dass ich auf dem Pfad der Tugend bleibe und nie einen Termin oder eine Zusage vergesse. Mit meinem Sohn Micah, dem Gründer von 8thlight.com, kümmere ich mich um Businesspläne. Sein Geschäftssinn ist deutlich besser, als meiner je war. Unser neuestes Unternehmen cleancoders.com ist sehr spannend!

Mein jüngerer Sohn Justin ist gerade bei Micah in 8th Light mit eingestiegen. Meine jüngste Tochter Gina arbeitet als Chemieingenieur für Honeywell. Mit diesen beiden hat das ernsthafte Projektieren gerade erst angefangen!

Niemand bringt einem im Leben mehr bei als die eigenen Nachkommen. Danke, Kinder!



Robert C. Martin (»Uncle Bob«) ist seit 1970 Programmierer. Er ist Gründer und Präsident von Object Mentor, Inc., einer internationalen Firma mit höchst erfahrenen Software-Entwicklern und Managern, die sich darauf spezialisiert haben, anderen Unternehmen bei der Erledigung ihrer Projekte zu helfen. Object Mentor bietet Beratung zur Verbesserung von Prozessen und objektorientiertes Software-Design, Training und Dienstleistungen zur Weiterbildung für große, weltweit tätige Unternehmen.

Martin hat Dutzende Artikel in verschiedenen Branchenfachblättern veröffentlicht und hält regelmäßig Vorträge auf internationalen Konferenzen und Messen.

Über den Autor

Er hat viele Bücher verfasst und herausgegeben, z.B.:

- *Designing Object Oriented C++ Applications Using the Booch Method*
- *Patterns Languages of Program Design 3*
- *More C++ Gems*
- *Extreme Programming in Practice*
- *Agile Software Development: Principles, Patterns, and Practices*
- *UML for Java Programmers*
- *Clean Code*

Als Leitfigur in der Branche der Software-Entwicklung arbeitete Martin drei Jahre als verantwortlicher Chefredakteur beim *C++ Report* und war erster Vorsitzender der Agile Alliance.

Robert ist auch Gründer von Uncle Bob Consulting, LLC, und mit seinem Sohn Micah Martin Mitbegründer von The Clean Coders LLC.



Das erstaunliche Bild auf dem Umschlag erinnert an das Auge von Sauron, gehört aber M1, dem Krebsnebel. M1 befindet sich im Sternbild des Stieres, etwa ein Grad rechts von Zeta Tauri, dem Stern an der Spitze des linken Stierhorns. Der Krebsnebel ist das Überbleibsel einer Supernova, die am 4. Juli 1054 n. Chr. am ganzen Himmel sichtbar explodierte. Bei einer Entfernung von 6.500 Lichtjahren erschien diese Explosion den chinesischen Beobachtern wie ein neuer Stern, etwa so leuchtend wie Jupiter. Tatsächlich war sie auch *tagsüber* sichtbar! Im Laufe der folgenden sechs Monate verblasste er langsam und war dann mit bloßem Auge nicht mehr erkennbar.

Das Umschlagbild setzt sich aus sichtbarem Licht und Röntgenstrahlen zusammen. Das sichtbare Bild wurde vom Hubble-Teleskop aufgenommen und formt den äußeren Rand. Das innere Objekt, das wie ein blaues Ziel für Bogenschützen aussieht, wurde vom Röntgenteleskop Chandra aufgenommen.

Auf dem US-Titelbild

Das sichtbare Bild stellt eine sich schnell ausdehnende Wolke aus Staub und Gas dar, durchdrungen von schweren Elementen, die von der Explosion der Supernova übrig geblieben waren. Diese Wolke hat nun einen Durchmesser von elf Lichtjahren, wiegt etwa das 4,5-Fache der Sonnenmasse und dehnt sich mit dem atemberaubenden Tempo von 1.500 Kilometern pro Sekunde aus. Die kinetische Energie dieser alten Explosion ist, gelinde gesagt, beeindruckend.

Genau in der Mitte des Ziels befindet sich ein leuchtend blauer Punkt. Dort befindet sich der *Pulsar*. Die Bildung des Pulsars hat überhaupt erst dafür gesorgt, dass der Stern in die Luft geflogen ist. Etwa eine Sonnenmasse von Material im Kern des todgeweihten Sterns implodierte in eine Sphäre von Neutronen mit etwa 30 Kilometern im Durchmesser. Die kinetische Energie dieser Implosion, gekoppelt mit dem unglaublichen Trommelfeuer der Neutrinos, die entstanden, als sich all diese Neutronen bildeten, zerriss den Stern und jagte ihn ins Jenseits.

Der Pulsar dreht sich etwa 30 Mal pro Sekunde und blinkt beim Drehen. Wir können in unseren Teleskopen sehen, wie er blinkt. Dieses pulsierende Licht ist der Grund, warum wir ihn als *Pulsar* bezeichnen: Das bedeutet pulsierender Stern.



(Überspringen Sie dies nicht, Sie werden es brauchen.)

Ich gehe davon aus, dass Sie dieses Buch in die Hand nehmen, weil Sie Computerprogrammierer sind und Sie die Idee der Professionalität fasziniert. Das sollte auch so sein. Professionalität ist etwas, das unser Berufsstand unbedingt benötigt.

Auch ich bin Programmierer. Ich bin seit 42¹ Jahren Programmierer, und in dieser langen Zeit – glauben Sie mir – habe ich *alles* gesehen. Ich wurde gefeuert. Ich wurde belobigt. Ich war Teamleiter, Manager, Arbeitsknecht und sogar CEO. Ich habe mit brillanten Programmierern und mit Faulpelzen gearbeitet. Ich habe auf Hightech-Systemen absolut bahnbrechender eingebetteter Soft- und Hardware gearbeitet und mit Firmensystemen zur Gehaltsabrechnung. Ich habe in COBOL, Fortran, BAL, PDP-8, PDP-11, C, C++, Java, Ruby, Smalltalk und einer Vielzahl anderer Sprachen und Systeme programmiert. Ich

1 Keine Panik.

Unverzichtbare Einführung

habe mit unzuverlässigen Gehaltsscheckdieben gearbeitet und auch mit vollendeten Profis. Und diese letzte Klassifikation ist Thema dieses Buches.

Auf diesen Seiten werde ich versuchen zu definieren, was es heißt, ein professioneller Programmierer zu sein. Ich werde die Haltungen, Disziplinen und Aktionen beschreiben, die ich für essenziell professionell halte.

Woher weiß ich, was zu diesen Haltungen, Disziplinen und Aktionen gehört? Weil ich sie auf die harte Tour lernen musste. Wissen Sie, als ich meinen ersten Job als Programmierer bekam, war »professionell« das letzte Wort, das Sie benutzt hätten, um mich zu beschreiben.

Das war im Jahre 1969. Ich war 17. Mein Vater hatte bei einer Firma vor Ort namens ASC erst dann Ruhe gegeben, als sie mich als Aushilfsprogrammierer in Teilzeit einstellten. (Mein Vater bekam solche Sachen hin. Ich schaute mal zu, wie er sich vor einen zu schnell fahrenden Wagen stellte und ihn mit erhobener Hand zum Stoppen zwang. Das Auto hielt an. Keiner sagt »Nein« zu meinem Dad.) Die Firma ließ mich in dem Raum arbeiten, wo all die Handbücher für die IBM-Computer aufbewahrt wurden. Man ließ mich viele Jahrgänge von Updates in die Manuals einsortieren. Hier las ich zum ersten Mal die Redewendung: »Diese Seite wurde absichtlich leer gelassen.«

Nach einigen Tagen der Aktualisierung von Handbüchern ließ mich mein Chef ein einfaches Easycoder-Programm schreiben². Ich war entzückt, dass ich gefragt wurde. Ich hatte noch nie zuvor ein Programm für einen echten Computer geschrieben. Ich hatte allerdings die Autocoder-Bücher inhaliert und eine ungefähre Vorstellung, wie ich anfangen musste.

Das Programm sollte einfach Aufzeichnungen von Bändern auslesen und die IDs dieser Einträge durch neue IDs ersetzen. Die neuen IDs begannen mit 1 und wurden für jeden neuen Eintrag um 1 inkrementiert. Die Aufzeichnungen mit den neuen IDs sollten auf ein neues Band geschrieben werden.

Mein Chef zeigte mir ein Regal, in dem viele Stapel roter und blauer Lochkarten lagerten. Stellen Sie sich einmal vor, Sie haben 50 Kartenspiele gekauft: 25 rote und 25 blaue. Dann stapeln Sie diese Kartenspiele aufeinander. So ähnlich sahen diese Kartenstapel aus. Sie waren rot und blau gestreift, und die Streifen waren jeweils auf etwa 200 Karten. Jeder dieser Streifen enthielt den Quellcode einer Subroutine-Bibliothek, die üblicherweise von den Programmierern verwendet wurde. Die Programmierer nahmen einfach die oberste Karte vom Stapel, achteten darauf, dass sie nur rote oder blaue Karten nahmen, und steckten sie dann ans Ende des Programmstapels.

Ich schrieb mein Programm auf einige Programmvordrucke. Das waren große rechteckige Papierbögen, die in 25 Zeilen mit 80 Spalten aufgeteilt waren. Jede Zeile repräsentierte eine Karte. Man schrieb sein Programm in großen Blockbuchstaben und

2 Easycoder war Assembler für den Computer Honeywell H200 und glich Autocoder für den IBM-Computer 1401.

einem mittelharten Bleistift auf den Programmvordruck. In den letzten sechs Spalten jeder Zeile schrieb man mit dem Bleistift eine Sequenznummer. Üblicherweise wurde die Sequenznummer immer um 10 inkrementiert, damit man auch später noch Karten einfügen konnte.

Der Programmvordruck wanderte dann zu den Lochkartenfirmen. Diese Firmen beschäftigten mehrere Dutzend Frauen, die diese Programmvordrucke aus einem großen Eingangskorb nahmen und sie dann in Lochkartenmaschinen »eintippten«. Diese Maschinen glichen Schreibmaschinen, außer dass die Zeichen in Karten gelocht statt auf Papier gedruckt wurden.

Am nächsten Tag lieferten die Lochkartenstanzer das Programm per Betriebspost an mich zurück. Auf meinem kleinen Schreibtisch stapelten sich die Lochkarten, die Programmvordrucke und ein Gummiband. Ich prüfte die Karten nach Stanzfehlern. Es gab keine. Also legte ich den Stapel mit der Subroutinen-Bibliothek auf meinen Programmstapel und brachte diesen hoch zu den Computeroperatoren.

Deren Computer befanden sich hinter verschlossenen Türen in einer klimatisierten Umgebung, unter deren Fußboden all die Kabel verlegt waren. Ich klopfte an die Tür. Ein Operator nahm mir mit strengem Blick den Stapel ab und legte ihn in einen weiteren Eingangskorb im Computerraum. Wenn sie soweit waren, ließen sie meinen Stapel durchlaufen.

Am nächsten Tag bekam ich meinen Stapel zurück. Es war in eine Auflistung mit den Ergebnissen dieses Durchlaufs eingepackt und wurde durch ein Gummiband zusammengehalten (damals verbrauchten wir *massig* Gummibänder!).

Ich öffnete die Auflistung und sah, dass meine Kompilierung misslungen war. Die Fehlermeldungen im Listing waren für mich nur schwer verständlich, also brachte ich sie zu meinem Chef. Er sah sie durch, murmelte etwas in seinen Bart, notierte schnell einige Dinge am Rand des Ausdrucks, schnappte sich meinen Stapel und wies mich an, ihm zu folgen.

Er nahm mich mit in den Raum mit den Lochstanzern und setzte sich an eine freie Maschine. Er korrigierte eine fehlerhafte Karte nach der anderen und fügte ein oder zwei neue Karten ein. Schnell erklärte er mir währenddessen, was er da eigentlich gerade machte, aber es rauschte bei mir nur so durch.

Er nahm den neuen Stapel mit zum Computerraum und klopfte an die Tür. Zu einem der Operatoren sprach er ein paar Zauberworte und trat dann in den Computerraum hinter ihm. Er winkte mir zu, dass ich ihm folgen solle. Der Operator richtete die Bandlaufwerke ein und lud den Kartenstapel, während wir zuschauten. Die Bänder drehten sich, der Drucker rattete los, und dann war es vorbei. Das Programm hatte funktioniert.

Am nächsten Tag dankte mein Chef mir für meine Hilfe und beendete meine Anstellung. Offenbar war bei ASC der Eindruck entstanden, man habe keine Zeit, einen 17-Jährigen zu hegen und zu pflegen.

Unverzichtbare Einführung

Doch meine Verbindung mit ASC war keineswegs vorüber. Wenige Monate später bekam ich einen Vollzeitjob bei ASC und bediente Offline-Drucker. Diese Drucker gaben die Junk-Mail von Print-Images aus, die auf Band gespeichert waren. Mein Job bestand darin, bei den Druckern Papier nachzufüllen, die Bänder in die Laufwerke zu legen, Papierstaus zu beheben und ansonsten nur darauf zu achten, dass die Geräte arbeiteten.

Das war im Jahre 1970. College war für mich keine Option und barg für mich auch keine speziellen Verlockungen. Der Vietnam-Krieg tobte immer noch, und auf den Campus war es chaotisch. Ich verschlang weiterhin Bücher über COBOL, Fortran, PL/1, PDP-8 und IBM 360 Assembler. Ich hatte mir vorgenommen, die Schule zu umgehen und mich so sehr anzustrengen, dass ich möglichst irgendwo einen Programmiererjob bekam.

Zwölf Monate später erreichte ich mein Ziel. Ich wurde bei ASC zum Vollzeitprogrammierer befördert. Mit Richard und Tim, zwei meiner besten Freunde, ebenfalls 19, arbeitete ich in einem Team mit drei anderen Programmierern zusammen an einem Echtzeitarbeitungssystem für eine Transportarbeitergewerkschaft. Die Maschine war eine Varian 620i. Das war ein einfacher Minicomputer, der von der Architektur her einer PDP-8 glich, außer dass er mit 16 Bit und zwei Registern arbeitete. Die Sprache war Assembler.

Wir schrieben jede Zeile Code in diesem System. Und damit meine ich *jede* Zeile. Wir schrieben das Betriebssystem, die Interrupt-Heads, die IO-Treiber, das *Dateisystem* für die Festplatten, den Overlay-Swapper und sogar den Relocatable Linker. Vom gesamten Applikationscode ganz zu schweigen. Das alles schrieben wir in acht Monaten und arbeiteten dafür zwischen 70 und 80 Stunden die Woche, um eine höllische Deadline zu erfüllen. Meine Vergütung war 7.200 Dollar pro Jahr.

Wir lieferten dieses System ab. Und dann kündigten wir.

Wir kündigten plötzlich und boshaft. Wissen Sie, nach all dieser Arbeit und nachdem wir ein erfolgreiches System abgeliefert hatten, erhielten wir von der Firma eine Gehaltserhöhung von 2 Prozent. Wir fühlten uns betrogen und ausgenutzt. Mehrere von uns bekamen anderswo Jobs und warfen einfach die Arbeit hin.

Ich hingegen ging einen anderen und sehr unglückseligen Weg. Mit einem Kumpel zusammen stürmte ich ins Büro des Chefs, wo wir ihm lautstark unsere Kündigung auf den Tisch knallten. Das war emotional sehr befriedigend – und hielt einen Tag lang an.

Am nächsten Tag traf mich die Erkenntnis wie ein Schlag, dass ich nun arbeitslos war. Ich war 19 und besaß weder eine Beschäftigung noch einen Abschluss. Ich hatte verschiedene Bewerbungsgespräche wegen einiger Jobs als Programmierer, aber die liefen nicht so gut. Also stieg ich für vier Monate beim Rasenmäherreparaturdienst meines Schwagers ein. Leider war ich ein lausiger Rasenmähermechaniker. Er musste mich schließlich entlassen. Ich fiel in ein fieses Loch.

Ich blieb jede Nacht bis 3 Uhr früh auf, futterte Pizza und schaute mir auf dem alten Schwarzweißfernseher meiner Eltern mit Zimmerantenne alte Monsterfilme an. Nicht alle Gespenster tummelten sich auf dem Bildschirm. Ich blieb bis nachmittags im Bett liegen, weil ich mich meinen trostlosen Tagen nicht stellen wollte. Ich belegte bei der Volkshochschule einen Kurs in Analysis und vergeigte ihn. Ich war ein Wrack.

Meine Mutter nahm mich beiseite und sagte mir, dass mein Leben vollkommen in Unordnung sei, dass ich ein Idiot gewesen sei, ohne was Neues einfach den alten Job zu kündigen und auch noch emotional derart aufgeladen alles hingeworfen zu haben, und dann auch noch zusammen mit meinem Kumpel. Sie sagte zu mir, dass man nie kündigt, ohne einen neuen Job zu haben, und dass man immer ganz ruhig und gelassen und für sich alleine kündigt. Sie forderte mich auf, meinen Ex-Chef anzurufen und darum zu bitten, meinen alten Job zurückzubekommen. Sie meinte: »Da musst du einfach mal zu Kreuze kriechen.«

19-jährige Jungs sind auf diese sportliche Übung nicht sonderlich erpicht, und da war ich keine Ausnahme. Aber die Umstände hatten meinen Stolz untergraben. Am Ende rief ich meinen Boss an und kroch ziemlich zu Kreuze. Und es funktionierte. Er stellte mich für 6.800 Dollar jährlich gerne wieder ein, und ich war glücklich, das Angebot anzunehmen.

Ich arbeitete weitere anderthalb Jahre dort und legte meine besten Manieren an den Tag. Ich bemühte mich, ein möglichst wertvoller Mitarbeiter zu sein. Ich wurde mit Beförderungen und Gehaltserhöhungen belohnt und bekam meinen Gehaltsscheck regelmäßig. Das Leben war gut. Als ich diese Firma verließ, war das zu guten Bedingungen, und ich hatte das Angebot eines besseren Jobs in der Tasche.

Wahrscheinlich glauben Sie, dass ich meine Lektion gelernt hatte und nun ein Profi war. Weit gefehlt. Das war nur die erste der vielen Lektionen, die ich noch zu lernen hatte. In den folgenden Jahren sollte ich bei einem Job rausfliegen, weil ich unachtsamerweise wesentliche Daten weggelassen hatte, und bei einem anderen beinahe gefeuert werden, weil ich einem Kunden versehentlich vertrauliche Informationen zugänglich gemacht hatte. Ich sollte bei einem zum Scheitern verurteilten Projekt die Leitung übernehmen und es vor die Wand fahren, ohne die Hilfe zu rufen, von der ich selbst wusste, dass ich sie brauchen würde. Ich sollte meine technischen Entscheidungen aggressiv verteidigen, obwohl sie diametral den Anforderungen des Kunden entgegenstanden. Ich sollte später eine völlig unqualifizierte Person einstellen und damit meinem Arbeitgeber eine Riesenverantwortung auf den Buckel laden. Und am schlimmsten von allem: Wegen mir sollten zwei Leute gefeuert werden, weil ich Führungsschwäche zeigte.

Also lesen Sie dieses Buch als Katalog meiner eigenen Fehler, als Protokoll meiner eigenen Vergehen und als Richtlinie für Sie, um zu vermeiden, dass Sie in meine Fußstapfen treten.



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

– Howard in: Der Schatz der Sierra Madre

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

1.1 Seien Sie vorsichtig, wonach Sie verlangen

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

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

Was würde passieren, wenn Ihnen bei einem Modul ein Bug durch die Lappen geht, der Ihre Firma 10.000 Euro kostet? Ein unprofessioneller Mitarbeiter zuckt mit den Schultern, murmelt »Dumm gelaufen« und schreibt das nächste Modul. Der Profi würde der Firma einen Scheck über 10.000 Euro ausstellen!¹

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

1.2 Verantwortung übernehmen

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

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

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

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

¹ Hoffentlich hat er eine gute Haftpflichtversicherung!

der Gebühren genutzt, die die Telefonfirmen für ihre Dienste berechnen durften. Kurz gesagt waren diese Systeme unglaublich wichtig.

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

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

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

Zwei Tage später bekam ich einen Anruf von unserem Außendienstmanager Tom. Er berichtete mir, dass mehrere Kunden sich darüber beschwert hatten, dass der »nächtlige Testlauf« nicht abgeschlossen worden war und sie keine Berichte bekommen hatten. Mein Herz rutschte mir in die Hose, denn um die Software rechtzeitig versenden zu können, hatte ich mir erspart, diese Routine zu testen. Ich hatte die restliche Funktionalität des Systems weitgehend getestet, aber es hätte Stunden gedauert, eben diese Routine zu testen, und ich musste die Software ausliefern. Keiner der Bugfixes befand sich im Routine-Code, also fühlte ich mich auf der sicheren Seite.

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

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

Ich rief Tom an, um ihm zu sagen, dass ich das Problem reproduzieren könne. Er sagte mir, dass die meisten der anderen Kunden ihn in gleicher Angelegenheit schon angeru-

fen hätten. Dann wollte er wissen, bis wann ich das beheben könne. Das wisse ich noch nicht genau, entgegnete ich, aber ich würde dran arbeiten. In der Zwischenzeit, ergänzte ich noch, sollten die Kunden auf die alte Software zurückgreifen. Tom war sauer auf mich und meinte, dass dies die Kunden doppelt träfe, weil sie nicht nur die Daten einer ganzen Nacht verloren hätten, sondern überdies auch nicht mit dem versprochenen neuen Feature arbeiten konnten.

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

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

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

1.3 Erstens: Richte keinen Schaden an

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

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

1.3.1 Beschädige nicht die Funktion

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

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

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

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

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

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

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

Die Qualitätssicherung sollte nichts finden

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

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

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

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

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

Sie müssen wissen, ob er funktioniert

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

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

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

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

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

Ich bin der Hauptentwickler und Committer für ein Open-Source-Projekt namens FitNesse. Während ich dies hier schreibe, gibt es in FitNesse 60.000 Zeilen Quellcode. 26.000 davon enthalten die über 2.000 Unit-Tests. Laut Emma² liegt die Abdeckung dieser 2.000 Tests bei etwa 90 %.

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

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

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

Automatisierte Qualitätssicherung

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

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

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

1.3.2 Beschädige nicht die Struktur

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

2 <http://emma.sourceforge.net/>

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

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

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

Über die Prinzipien und Patterns des Software-Designs, die flexibel und einfach zu wartende Strukturen unterstützen, ist schon viel geschrieben worden³. Professionelle Software-Entwickler prägen sich diese Dinge ins Gedächtnis ein und bemühen sich, dazu konforme Software zu entwickeln. Aber es gibt einen Trick dafür, den viel zu wenige Software-Entwickler befolgen: *Wenn Sie wollen, dass die Software flexibel ist, müssen Sie sie auch belasten und ausreizen!*

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

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

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

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

3 [PPP2001]

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

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

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

1.4 Arbeitsethik

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

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

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

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

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

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

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

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

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

Vielleicht glauben Sie, dass man die Arbeit auf der Arbeit lassen und nicht mit nach Hause nehmen sollte. Dem stimme ich zu! Sie sollten in diesen 20 Stunden nicht für Ihren Arbeitgeber tätig sein. Stattdessen sollten Sie an Ihrer Karriere arbeiten.

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

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

1.4.1 Sie sollten sich in Ihrem Bereich auskennen

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

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

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

Der zweite Teil der Frage stimmt ganz gewiss nicht. Nur sehr wenige Ideen aus den vergangenen 50 Jahren sind irrelevant geworden. Sicher, manche sind auf dem Abstellgleis gelandet. Die Entwicklung nach dem Wasserfallmodell ist sicherlich in Ungnade gefallen. Doch das bedeutet nicht, dass wir es uns leisten können, es nicht zu kennen und nicht zu wissen, welche guten und schlechten Aspekte es mit sich bringt.

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

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

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

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

1.4.2 Lebenslanges Lernen

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

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

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

1.4.3 Praxis

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

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

Wie könnten also Software-Entwickler für ihre Praxis üben? Dieses Buch enthält ein ganzes Kapitel, das sich den verschiedenen Übungstechniken widmet. Also werde ich hier nicht ins Detail gehen. Eine Technik, die ich häufig einsetze, ist die Wiederholung

einfacher Übungen wie das *Bowling Game* oder *Primfaktoren*. Ich nenne diese Übungen Kata. Es gibt viele Kata, unter denen man wählen kann.

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

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

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

1.4.4 Teamwork

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

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

1.4.5 Mentorenarbeit

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

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

1.4.6 Sie sollten sich in Ihrem Arbeitsgebiet auskennen

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

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

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

1.4.7 Identifizieren Sie sich mit Ihrem Arbeitgeber bzw. Kunden

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

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

1.4.8 Bescheidenheit

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

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

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

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

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

1.5 Bibliografie

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



Mach es oder lass es. Es gibt kein Versuchen.

– Yoda

Anfang der 1970er-Jahre arbeitete ich mit zwei Freunden, beide ebenfalls 19 Jahre alt, bei einer Firma namens ASC an einem Echtzeitabrechnungssystem für die Chicagoer Transportarbeitergewerkschaft. Wenn Ihnen da ein Name wie Jimmy Hoffa in den Sinn kommt, dann trifft das zu. Mit der Gewerkschaft der Transportarbeiter legte man sich damals 1971 nicht an.

Unser System sollte zu einem bestimmten Zeitpunkt live gehen. Ein Menge Geld hing von diesem Datum ab! Um diesen Zeitplan einzuhalten, hatte unser Team pro Woche 60, 70 und 80 Stunden gearbeitet.

Eine Woche, bevor das System live gehen sollte, gelang es uns schließlich, das System komplett zusammenzusetzen. Es gab eine Menge Bugs und andere Probleme, um die wir uns zu kümmern hatten, und wir arbeiteten uns fieberhaft durch die Liste. Es gab kaum Zeit zum Essen und Schlafen, geschweige denn zum Nachdenken.

Frank, der Manager von ASC, war ein ehemaliger Colonel der Air Force. Er war einer jener lauten, provokativen Manager. Bei ihm hieß es stets »Friss, Vogel, oder stirb«, und um den zweiten Teil kümmerte er sich gerne, indem er anbot, einen ohne Fallschirm aus 10.000 Fuß Höhe abzuwerfen. Gerade mal 19 Jahre alt, waren wir kaum in der Lage, seinem Blick standzuhalten.

Frank sagte, es müsse alles bis zu diesem Datum fertig sein. Mehr gab es dazu nicht zu sagen. Wenn dieser Tag anbricht, dann wären wir fertig. Basta. Keine Diskussionen. Ende und aus.

Mein Chef Bill war ein liebenswürdiger Kerl. Er hatte schon ein paar Jahre mit Frank gearbeitet und wusste, was bei Frank möglich war und was nicht. Er meinte zu uns, dass wir das System an diesem Tag freizugeben hatten, egal wie.

Also gingen wir an diesem Tag mit dem System live. Und es war eine absolute Katastrophe.

Es gab ein Dutzend Halbduplex-Terminals mit 300 Baud, die das Hauptquartier der Gewerkschaft in Chicago mit unserem Rechner 40 Kilometer nördlich in den Vororten verbanden. Jedes dieser Terminals blockierte etwa alle 30 Minuten. Wir kannten dieses Problem bereits, hatten aber den Traffic noch nicht simuliert, den die Angestellten der Gewerkschaft für die Dateneingabe plötzlich in unser System jagten.

Um alles noch schlimmer zu machen: Wenn auf die ASR35-Teletypes, die mit 110-Baud-Telefonleitungen ebenfalls mit unserem System verbunden waren, gedruckt wurde, hängte sich der Druckvorgang mittendrin auf, und das Abreißpapier blieb stecken.

Die Lösung für dieses Einfrieren war ein Neustart. Also mussten alle, deren Terminals noch funktionierten, immer ihren jeweiligen Arbeitsvorgang beenden und dann aufhören. Wenn alle fertig waren, wurden wir angerufen, um den Neustart auszulösen. Die Leute, bei denen sich die Verbindung aufgehängt hatte, mussten von vorne anfangen. Und das geschah mindestens einmal pro Stunde.

Nachdem das einen halben Tag lang so gelaufen war, wies uns der Büroleiter der Gewerkschaft an, das System herunterzufahren und erst dann wieder einzuschalten, wenn wir es ans Laufen gebracht hatten. Inzwischen hatten sie einen halben Tag Arbeit verloren und mussten alles über das alte System erneut eingeben.

Wir hörten im ganzen Gebäude, wie Frank am Toben war. Das ging eine sehr lange Zeit so. Dann kam Bill mit Jalil, unserem Systemanalytiker, zu uns und fragte, wann wir in etwa das System stabil haben konnten. Ich antwortete: »Dauert vier Wochen.«

Auf ihren Gesichtern zeichneten sich zuerst der reine Schrecken und dann Entschlossenheit ab. »Nein«, sagten sie, »es muss am Freitag wieder laufen.«

Also entgegnete ich: »Hört mal, wir haben dieses System letzte Woche so gerade eben ans Laufen bekommen. Wir müssen ihm die Probleme und Defekte austreiben. Wir brauchen vier Wochen.«

Aber Bill und Jalil blieben unbittlich. »Nein, Freitag muss es definitiv wieder einsatzbereit sein. Könnt ihr das nicht wenigstens versuchen?«

Dann sagte unser Teamleiter: »Okay, versuchen wir's.«

Freitag war eine gute Wahl. Die Arbeitsbelastung war am Wochenende deutlich geringer. Wir konnten vor dem nächsten Montag viele Probleme finden und korrigieren. Und trotzdem stürzte unser ganzes Kartenhaus beinahe wieder erneut ein. Das System hängte sich immer noch ein oder zwei Mal täglich auf. Es gab auch noch andere Probleme. Aber schließlich kamen wir nach weiteren Wochen mit dem System Stück für Stück an einen Punkt, an dem die Beschwerden nachließen und es fast wieder möglich erschien, ein normales Leben zu führen.

Und dann, wie ich Ihnen in der Einführung schon verraten habe, warfen wir alles hin. Und ließen die Kollegen mit einer echten Krise am Hals stehen. Sie mussten einen neuen Trupp Programmierer einstellen, um mit der riesigen Menge von Problemen fertig zu werden, die der Kunde meldete.

Wem können wir dieses Debakel ans Bein binden? Franks Arbeitsstil gehörte eindeutig zum Problem. Durch seine Einschüchterungen war es schwer, ihm die Wahrheit ins Gesicht zu sagen. Bill und Jalil hätten bei Frank ganz sicher deutlich mehr gegenhalten sollen, als sie es gemacht haben. Sicherlich hätte unser Teamleiter bei der zeitlichen Anforderung bis Freitag nicht einknicken dürfen. Und ganz bestimmt hätte ich weiterhin »Nein« sagen sollen, anstatt mich hinter der Teamleitung zu verstecken.

Profis sagen angesichts der Macht die Wahrheit. Profis haben den Mumm, ihren Managern gegenüber Nein zu sagen.

Wie sagt man Nein zu seinem Chef? Immerhin ist er Ihr *Boss*! Muss man nicht das machen, was einem der Boss sagt?

Nein. Nicht, wenn man Profi ist.

Sklaven dürfen nichts ablehnen. Arbeiter könnten dabei zögern, etwas zu verweigern. Aber von Profis *erwartet* man, dass sie Nein sagen. Tatsächlich sehnen sich gute Manager nach jemandem, der den Mumm hat, Nein zu sagen. Das ist der einzige Weg, wie man wirklich etwas erledigt bekommt.

2.1 Feindliche Rollen

Einer der Rezensenten dieses Buches hat dieses Kapitel wirklich gehasst. Er meinte, dass er deswegen beinahe das Buch ganz weggelegt hätte. Er hatte Teams aufgebaut, wo es keine feindlichen Beziehungen gab. Die Teams arbeiteten harmonisch und ohne Konfrontationen zusammen.

Ich freue mich für diesen Rezensenten, aber ich frage mich doch, ob seine Teams wirklich so frei von Konfrontationen sind, wie er annimmt. Und wenn sie es sind, stellt sich für mich die Frage, ob sie so effektiv sind, wie sie es sein könnten. Meiner eigenen Erfahrung nach wurden die harten Entscheidungen am besten dadurch getroffen, dass man gegensätzliche Rollen konfrontativ zusammenführt.

Manager sind Leute, die einen Job zu erledigen haben, und die meisten Manager wissen sehr genau, *wie* sie diesen Job zu machen haben. Zu diesem Job gehört, ihre Zielvorgaben so aggressiv wie möglich zu verfolgen und zu verteidigen.

Programmierer sind ebenso Menschen, die einen Job zu machen haben, und die meisten wissen ziemlich genau, wie sie ihre Arbeit gut machen. Wenn sie Profis sind, werden sie *ihre* Zielvorgaben genauso aggressiv verfolgen und verteidigen, wie *sie* können.

Wenn Ihr Manager Ihnen sagt, dass die Login-Seite morgen fertig sein muss, dann verfolgt und verteidigt er eine seiner Zielvorgaben. Er macht seinen Job. Wenn Sie absolut sicher sind, dass die Login-Seite bis morgen unmöglich fertig sein kann, dann machen Sie Ihren Job nicht, wenn Sie sagen: »Okay, ich probiere es.« Der einzige Weg, um an diesem Punkt Ihren Job zu erfüllen, ist die Aussage: »Nein, das ist unmöglich.«

Aber müssen Sie nicht das befolgen, was Ihr Manager Ihnen sagt? Nein, Ihr Manager zählt darauf, dass Sie Ihre Planziele ebenso aggressiv verteidigen wie er selbst. So werden Sie beide das *bestmögliche Ergebnis* bekommen.

Das bestmögliche Ergebnis ist das Ziel, das Sie und Ihr Manager gemeinsam teilen. Der Trick besteht darin, dieses Ziel zu finden, und dazu gehören normalerweise Verhandlungen.

Verhandlungen können manchmal ganz angenehm sein.

Mike: »Paula, ich brauche die Login-Seite, die muss also bis morgen fertig sein.«

Paula: »Tatsächlich – so bald schon? Tja, okay, ich probier's.«

Mike: »Okay, das ist super. Vielen Dank!«

Das war eine nette, kleine Konversation. Jegliche Konfrontation wurde vermieden. Beide Parteien ziehen sich lächelnd zurück. Sehr schön.

Aber beide Parteien haben sich unprofessionell verhalten. Paula hat ganz genau gesagt, dass sie für die Login-Seite länger als einen Tag braucht, und hat somit gelogen. Vielleicht hält sie es nicht für eine Lüge. Vielleicht denkt sie, dass sie es *wirklich versuchen wird*, und vielleicht hegt sie auch eine dürre Hoffnung, dass sie damit tatsächlich bis morgen fertig sein könnte. Aber letzten Endes bleibt es doch eine Lüge.

Mike seinerseits hat das »Ich probier's« als »Ja« akzeptiert. Da hat er wirklich was Dummes gemacht. Er hätte wissen müssen, dass Paula einfach nur die Konfrontation vermeiden wollte. Also hätte er nachhaken müssen, indem er sagt: »Du wirkst so zögerlich. Bist du sicher, dass du das bis morgen hinkriegst?«

Es ist eine weitere angenehme Konversation.

Mike: »Paula, ich brauche die Login-Seite, die muss also bis morgen fertig sein.«

Paula: »Oh, tut mir leid, Mike, aber ich brauche dafür mehr Zeit.«

Mike: »Wann hast du das fertig?«

Paula: »Sollte ich in zwei Wochen fertig haben.«

Mike: (kritzelt etwas in seinen Kalender) »Okay, danke.«

So angenehm, wie dieses Gespräch auch wirken mag, hat es nicht die gewünschte Wirkung und ist dysfunktional und absolut unprofessionell. Beide Seiten haben in ihrer Suche nach dem bestmöglichen Ergebnis versagt. Anstatt zu fragen, ob zwei Wochen in Ordnung seien, hätte Paula sich bestimmter äußern sollen: »Dafür brauche ich zwei Wochen, Mike.«

Mike seinerseits hat das Datum einfach ohne Gegenfrage akzeptiert, als seien seine eigenen Planvorgaben egal. Man fragt sich, ob er nun einfach seinem Vorgesetzten berichten wird, dass die Demo für den Kunden wegen Paula verschoben werden muss. Diese Art von passiv-aggressivem Verhalten ist moralisch verwerflich.

In all diesen Fällen hat sich keine Seite für ein gegenseitig akzeptables Ziel eingesetzt. Keine hat sich für das bestmögliche Ergebnis engagiert. Versuchen wir es noch einmal.

Mike: »Paula, ich brauche die Login-Seite, die muss also bis morgen fertig sein.«

Paula: »Nein, Mike, für diese Arbeit brauche ich zwei Wochen.«

Mike: »Zwei Wochen? Die Architekten haben dafür drei Tage eingeplant, und fünf sind nun schon vergangen!«

Paula: »Die Architekten hatten unrecht, Mike. Sie haben ihre Schätzungen vorgenommen, bevor das Produktmarketing die Anforderungen in die

Finger bekommen hat. Ich brauche mindestens noch zehn Arbeitstage dafür. Hast du meine aktualisierte Kalkulation im Wiki nicht gesehen?»

Mike: (schaut streng und zittert vor Frust und Enttäuschung) »Das ist inakzeptabel, Paula. Die Kunden kommen morgen wegen der Demo, und ich muss denen zeigen, dass die Login-Seite funktioniert.«

Paula: »Welchen Teil der Login-Seite musst du bis morgen funktionsfähig haben?»

Mike: »Ich brauche *die Login-Seite*! Ich muss mich *einloggen können*.«

Paula: »Mike, ich kann dir einen Dummy für die Login-Seite geben, auf der du dich einloggen kannst. Das funktioniert bei mir schon. Da werden zwar Username und Passwort nicht überprüft, und man kriegt auch keine Mail, wenn man das Passwort vergessen hat. Die News-Banner der Firma sind oben noch nicht eingebaut, und der Hilfe-Button und der Hover-Text funktionieren auch noch nicht. Cookies werden noch nicht gespeichert, falls man auf die Site zurückkommt, und man hat auch noch keine Zulassungseinschränkungen. Aber du kannst dich einloggen. Reicht das?»

Mike: »Ich kann mich dann einloggen?»

Paula: »Ja, kannst du.«

Mike: »Das ist klasse, Paula. Du bist echt eine Lebensretterin!« (Geht weg, stößt die Faust in die Luft und ruft »Ja!!«)

Sie haben das bestmögliche Ergebnis erreicht. Das haben sie bekommen, indem sie beide Nein gesagt und dann eine Lösung erarbeitet haben, die für beide tragfähig war. Sie verhielten sich wie Profis. Das Gespräch war ein wenig konfrontativ, und es gab ein paar unangenehme Augenblicke, aber das kann man erwarten, wenn zwei Personen starrsinnig Ziele verfolgen, die nicht perfekt aufeinander abgestimmt sind.

2.1.1 Was ist mit dem Warum?

Vielleicht sind Sie der Ansicht, Paula hätte erklären sollen, *warum* die Login-Seite so viel länger dauern würde. Meiner Erfahrung nach ist das *Warum* weitaus weniger wichtig als die *Tatsache*. Die Tatsache lautet hier, dass die Login-Seite noch zwei Wochen Arbeit braucht. Warum das so lange dauern soll, ist nur ein vernachlässigbares Detail.

Doch wenn er den Grund kennt, würde Mike das helfen, die Tatsache besser zu verstehen und damit auch zu akzeptieren. Das ist nur fair. Und in Situationen, wo Mike die technische Sachkenntnis hat und in der Verfassung fürs Verstehen ist, können solche

Erklärungen auch hilfreich sein. Andererseits könnte Mike der Schlussfolgerung auch nicht zustimmen. Mike könnte beschließen, dass Paula alles verkehrt macht. Er könnte dann sagen, dass sie diesen ganzen Testkram oder das Reviewing nicht zu machen braucht oder dass man Schritt 12 auch weglassen kann. Wenn man zu viele Details liefert, kann das eine Einladung zum Mikromanagement sein.

2.2 Hoher Einsatz

Die wichtigste Zeit für ein Nein ist, wenn der Einsatz am höchsten ist. Je höher der Einsatz ist, desto wertvoller wird ein Nein.

Das sollte offensichtlich sein. Wenn die Kosten eines Fehlschlags so hoch sind, dass das Überleben Ihrer Firma davon abhängt, müssen Sie absolut fest entschlossen sein, Ihren Managern die besten Informationen zu geben, die Sie zur Verfügung haben. Und das bedeutet oft, Nein zu sagen.

Don (Leiter der Entwicklungsabteilung): »So, unsere aktuelle Schätzung für den Abschluss des Projekts *Goldene Gans* liegt bei zwölf Wochen ab heute plus/minus etwa fünf Wochen.«

Charles (CEO, sitzt 15 Sekunden reglos mit funkelnden Augen und wird immer röter): »Ist das Ihr Ernst, sich hier hinzusetzen und mir mitzuteilen, dass wir unter Umständen noch 17 Wochen bis Abgabe brauchen?«

Don: »Das ist möglich, ja.«

Charles: [steht auf, Don steht eine Sekunde später auf] »Verdammt noch mal, Don! Das sollte eigentlich schon vor drei Wochen fertig sein! Galitron ruft mich jeden Tag an und will unbedingt wissen, wo ihr verflixtes System bleibt. Ich werde denen nicht sagen müssen, dass sie noch weitere vier Monate warten müssen ...! Da erwarte ich mehr von Ihnen.«

Don: »Charles, ich habe Ihnen *vor drei Monaten* nach all den Entlassungen gesagt, dass wir vier weitere Monate brauchen würden. Herrgott noch mal, Charles, ich will damit sagen, Sie haben mein Team um ein Fünftel eingedampft! Haben Sie Galitron damals gesagt, dass wir uns verspäten?«

Charles: »Sie wissen ganz genau, dass ich das nicht gemacht habe. Wir können es uns nicht leisten, diesen Auftrag zu verlieren, Don.« [Hält inne, sein Gesicht wird weiß.] »Ohne Galitron stecken wir verdammt tief im Schlamassel. Das wissen Sie, oder? Und jetzt, mit dieser Verzögerung, muss ich fürchten ... Was soll ich dem Vorstand sagen?« [Er sinkt

langsam wieder auf den Sitz zurück und versucht, nicht zusammenzuklappen.) »Don, das müssen Sie besser hinkriegen.«

Don: »Da kann ich einfach nichts machen, Charles. Das sind wir doch schon mal durchgegangen. Galitron wird seine Ansprüche nicht zurückschrauben und auch keine vorläufigen Releases akzeptieren. Sie wollen die Installation nur einmal durchführen und dann alles erledigt haben. Ich kann das einfach nicht schneller hinkriegen. Das ist einfach *nicht* möglich!«

Charles: »Verdammt. Ich nehme mal an, es wäre egal, wenn ich Ihnen sage, dass Ihr Job auf dem Spiel steht.«

Don: »Wenn Sie mich feuern, Charles, wird das den Zeitrahmen auch nicht ändern.«

Charles: »Wir sind fertig hier. Gehen Sie zu Ihrem Team zurück und halten Sie das Projekt am Laufen. Ich muss jetzt ein paar ziemlich heftige Telefonate machen.«

Natürlich hätte Charles Galitron vor drei Monaten informieren sollen, als er zum ersten Mal von der neuen Zeitplanung erfahren hatte. Zumindest jetzt macht er das Richtige, indem er Galitron (und seinen Vorstand) anruft. Aber zum Glück hat Don nicht locker gelassen, denn sonst wären diese Telefonate noch länger aufgeschoben worden.

2.3 Ein »Teampayer« sein

Wir alle kennen den Spruch, wie wichtig es ist, ein »Teampayer« zu sein. Als Teamplayer spielt man auf einer bestimmten Position so gut wie möglich und hilft den Teamkollegen, wenn sie in eine Misere geraten. Ein Teamplayer kommuniziert regelmäßig und häufig und behält seine Teamkollegen im Auge. Außerdem setzt er die Dinge in seinem Verantwortungsbereich so gut wie möglich um.

Ein Teamplayer sagt nicht zu allem Ja und Amen. Nehmen wir folgendes Szenario:

Paula: »Mike, ich habe hier die Kalkulation für dich. Das Team stimmt zu, dass wir in etwa acht Wochen eine Demo zeigen können, plus/minus eine Woche.«

Mike: »Paula, wir haben in sechs Wochen schon einen Termin für die Demo anberaumt.«

Paula: »Ohne uns vorher zu fragen? Komm schon, Mike, das kannst du uns nicht einbrocken.«

Mike: »Ist schon eingestellt.«

- Paula: (seufzt) »Okay. Hör mal, ich setze mich gleich mit dem Team zusammen und finde heraus, was wir in sechs Wochen ganz sicher abliefern können, aber das komplette System wird's nicht sein. Da werden ein paar Features fehlen, und die Datenmenge wird unvollständig sein.«
- Mike: »Paula, der Kunde erwartet, die Demo komplett zu sehen.«
- Paula: »Das ist einfach *nicht* möglich, Mike.«
- Mike: »Verdammt. Okay, bitte arbeite den bestmöglichen Plan heraus und informiere mich morgen darüber.«
- Paula: »Das kann ich machen.«
- Mike: »Gibt's da nicht irgendwas, was du machen kannst, um diesen Termin zu halten? Vielleicht gibt es einen Weg, irgendwie schlauer zu arbeiten und kreativ zu werden.«
- Paula: »Mike, wir sind hier alle ziemlich kreativ. Wir haben das Problem ganz gut im Griff, und das Projekt wird in acht oder neun Wochen beendet sein und nicht in sechs.«
- Mike: »Ihr könntet Überstunden machen.«
- Paula: »Das bremst uns bloß aus, Mike. Weißt du noch, was für ein Chaos das beim letzten Mal gegeben hat, als wir Überstunden angeordnet haben?«
- Mike: »Sicher, aber das muss dieses Mal doch nicht wieder so sein.«
- Paula: »Es wird genauso sein wie letztes Mal, Mike. Glaub mir. Es werden acht oder neun Wochen, keine sechs.«
- Mike: »Okay, reich mir den bestmöglichen Plan rein, aber denk bitte die ganze Zeit daran, wie man das in sechs Wochen hinkriegen kann. Ich weiß einfach, dass ihr so klasse seid, dass euch schon was einfallen wird.«
- Paula: »Nein, Mike, es wird uns nichts einfallen. Ich werde dir einen Plan für sechs Wochen geben können, aber da wird eine Menge Features und Daten nicht mit drin sein. So ist es dann eben.«
- Mike: »Okay, Paula, aber ich wette, dass dein Team wirklich Wunder hinkriegen kann, wenn ihr das nur versucht.«

(Paula geht kopfschüttelnd weg.)

Später, im Strategiemeeting des Vorstands ...

- Don: »Okay, Mike, wie Sie wissen, kommt der Kunde in sechs Wochen wegen der Demo zu uns. Da wird erwartet, dass alles funktioniert.«

- Mike: »Genau, und wir werden bis dahin auch fertig sein. Mein Team reißt sich dafür wirklich den A... auf, und wir werden das definitiv umsetzen. Wir werden Überstunden machen und ziemlich einfallsreich werden müssen, aber wir kriegen das hin!«
- Don: »Ganz großartige Sache, dass Ihre Leute und Sie solche tollen Teamplayer seid.«

Wer waren in diesem Szenario die echten *Teamplayer*? Paula hat für ihr Team gespielt, weil sie für das einstand, was unter Einsatz aller Kräfte geschafft werden konnte und was eben nicht. Sie hat ihre Position aggressiv verteidigt – trotz all der Betteleien und Schmeicheleien von Mike. Mike hat in einem Team mit nur einem Mitglied gespielt. Mike gehört zu Mike. Er ist eindeutig nicht in Paulas Team, weil er ihr einfach etwas aufgebürdet hat, über das sie explizit gesagt hat, es sei ihr nicht möglich. Er ist auch nicht in Dons Team (obwohl er dieser Aussage nicht zustimmen würde), weil er nach Strich und Faden gelogen hat.

Warum hat Mike also so gehandelt? Er wollte, dass Don ihn als Teamplayer betrachtet, und vertraute auf seine Fähigkeit, mit Betteln und Schmeicheln Paula so weit zu manipulieren, dass sie die Deadline in sechs Wochen *versuchen* würde. Mike ist nicht böse, er legt einfach nur zu viel Vertrauen in seine Fähigkeit, andere dazu zu bringen, das zu machen, was er will.

2.3.1 Versuchen

Das Schlimmste, was Paula als Reaktion auf Mikes Manipulationen machen konnte, war die Aussage: »Okay, wir versuchen es.« Ich schleuse hier nur ungern Yoda ein, aber in diesem Fall hat er recht. *Es gibt kein Versuchen*.

Vielleicht gefällt Ihnen diese Idee nicht? Vielleicht glauben Sie, dass es positiv ist, etwas zumindest zu *versuchen*. Schließlich hätte Kolumbus Amerika auch nicht entdeckt, wenn er es nicht wenigstens versucht hätte, oder?

Das Wort *versuchen* hat viele Definitionen. Die Definition, mit der ich hier nicht übereinstimme, lautet: »besondere Mühe aufwenden«. Welche besondere Mühe könnte Paula aufwenden, um die Demo rechtzeitig fertigzukriegen? Falls es noch besondere Mühe *gibt*, die sie einbringen kann, dann hat sie mit ihrem Team bisher nicht all ihre Kraft eingesetzt. Also hat man noch ein paar Reserven zurückbehalten.¹

Wenn man verspricht, etwas zu versuchen, gibt man damit zu, dass man etwas zurückgehalten hat, dass es noch Reserven gibt, die man abrufen kann, und man also nicht alles gegeben hat. Das Versprechen, es zu versuchen, ist das Eingeständnis, dass das Ziel durch Erbringen dieser besonderen Anstrengung erreichbar ist. Überdies ist es ein

¹ So wie Foghorn Leghorn: »Für einen solchen Notfall habe ich immer meine Federn durchnummeriert.«

Commitment, sich besonders anzustrengen, um das Ziel zu erreichen. Somit gehen Sie die Verpflichtung ein, erfolgreich zu sein, wenn Sie versprechen, etwas zu versuchen. Damit bürdet Sie sich die Last auf die eigenen Schultern. Wenn Ihr »Versuchen« nicht zum gewünschten Ergebnis führt, haben Sie versagt.

Haben Sie noch ein zusätzliches Energiereservoir, das Sie bisher zurückgehalten haben? Wenn Sie diese Reserven einsetzen, werden Sie dann das Ziel erreichen? Oder rechnen Sie bereits mit einem Fehlschlag, wenn Sie versprechen, es wenigstens zu versuchen?

Wenn Sie versprechen, etwas zu versuchen, versprechen Sie, Ihre Pläne zu ändern. Immerhin reichten Ihre bisherigen Pläne nicht aus. Wenn Sie es zu versuchen versprechen, machen Sie damit deutlich, dass Sie einen neuen Plan haben. Worin besteht dieser neue Plan? Welche Änderung wird das in Ihrem Verhalten bewirken? Was machen Sie nun anders, wenn Sie es ... »versuchen«?

Wenn Sie keinen neuen Plan haben, wenn Sie an Ihrem Verhalten nichts verändern, wenn Sie alles genauso weitermachen wie bisher, bevor Sie das mit dem »Versuchen« versprochen – was soll dann dieses Versuchen bedeuten?

Wenn Sie keine weiteren Energiereserven zurückhalten, wenn Sie keinen neuen Plan haben, wenn Sie Ihr Verhalten nicht ändern werden und wenn Sie auf vernünftige Weise Ihrer ursprünglichen Schätzung vertrauen, dann ist dieses Versprechen, etwas zu versuchen, zutiefst unehrlich. Sie *lügen*! Und das machen Sie wahrscheinlich, um Ihr Gesicht zu wahren und eine Konfrontation zu vermeiden.

Paulas Vorgehen war deutlich besser. Sie blieb dabei, Mike daran zu erinnern, dass die Kalkulation des Teams ungewiss war. Sie hat immer von »acht oder neun Wochen« gesprochen. Sie hat stets die Unsicherheit betont und ist nicht eingeknickt. Sie hat nie eingeräumt, dass man sich besondere Mühe geben könnte oder dass es einen neuen Plan gebe oder man sich anders verhalten würde, um diese Unsicherheit zu reduzieren.

Drei Wochen später ...

Mike: »Paula, in drei Wochen ist die Demo, und die Kunden verlangen zu sehen, dass der DATEI-UPLOAD funktioniert.«

Paula: »Das gehört aber nicht zu der Feature-Liste, auf die wir uns verständigt haben, Mike.«

Mike: »Das weiß ich, aber die verlangen es halt.«

Paula: »Okay, das heißt aber, dass entweder das SINGLE SIGN-ON oder das BACK-UP bei der Demo unter den Tisch fällt.«

Mike: »Definitiv nicht! Die erwarten, auch die Funktion dieser Features demonstriert zu bekommen!«

Paula: »So, dann gehen sie also davon aus, dass jedes Feature funktionieren soll? Ist es das, was du mir sagen willst? Ich habe dir doch schon gesagt, dass wir das nicht hinkriegen.«

Mike: »Paula, es tut mir leid, aber der Kunde gibt bei diesem Thema kein Stück nach. Die wollen einfach alles sehen.«

Paula: »Das ist einfach *nicht* zu machen, Mike. Definitiv nicht.«

Mike: »Komm schon, Paula, könnt ihr das nicht wenigstens *versuchen*?«

Paula: »Mike, ich könnte hier auch *versuchen* zu schweben. Ich könnte auch *versuchen*, Blei in Gold zu verwandeln. Ich könnte *versuchen*, über den Atlantik zu schwimmen. Meinst du, dass mir das gelingen würde?«

Mike: »Jetzt übertreibst du. Ich erwarte doch nichts *Unmögliches*.«

Paula: »Doch, Mike, das machst du.«

(Mike lächelt süffisant, nickt und wendet sich zum Gehen.)

Mike: »Ich vertraue dir voll und ganz, Paula. Ich weiß, dass ihr mich nicht hängen lasst.«

Paula: (spricht zu Mikes Rücken) »Mike, du träumst. Das wird alles böse enden.«

(Mike winkt bloß, ohne sich umzudrehen.)

2.3.2 Passive Aggression

Paula hat eine interessante Entscheidung zu treffen. Sie hat den Verdacht, dass Mike Don nichts über ihre Kalkulation berichtet. Sie könnte einfach zulassen, dass Mike ins Messer läuft. Sie könnte darauf achten, dass Kopien aller entsprechenden Notizen und Memos archiviert sind. Wenn dann die Katastrophe ausbricht, kann sie beweisen, *was* sie zu Mike gesagt hat und *wann*. Das ist passive Aggression. Sie würde dafür sorgen, dass Mike sich selbst zur Strecke bringt.

Oder sie könnte das Desaster abfangen, indem sie direkt mit Don kommuniziert. Das ist ein riskantes Spiel, gehört aber auch dazu, ein Teamplayer zu sein. Wenn ein Frachtzug auf einen zuhält und man ist der Einzige, der das wahrnimmt, kann man sich entscheiden, ob man still und leise von den Gleisen verschwindet und zuschaut, wie alle anderen überfahren werden, oder man kann laut brüllen: »Achtung – Zug! Runter von den Gleisen!«

Zwei Tage später ...

- Paula: »Mike, hast du Don von meiner Kalkulation berichtet? Hat er den Kunden informiert, dass bei der Demo das DATEI-UPLOAD-Feature noch nicht funktioniert?«
- Mike: »Paula, du hast mir gesagt, dass du das bis dahin für mich einbauen wirst.«
- Paula: »Nein, Mike, habe ich nicht. Ich habe dir gesagt, das sei unmöglich. Hier ist eine Kopie des Memos, das ich dir nach unserem Gespräch geschickt habe.«
- Mike: »Ja, aber du wolltest es doch trotzdem probieren, nicht wahr?«
- Paula: »Wir sind diese Diskussion doch schon durchgegangen, Mike. Weißt du noch? Gold und Blei?«
- Mike: (seufzend) »Schau mal, Paula, du musst es einfach schaffen. Wirklich, das musst du. Bitte unternimm alles, was dafür nötig ist, aber du musst das einfach für mich umsetzen.«
- Paula: »Mike, da liegst du verkehrt. Ich *mus*s das nicht für dich hinkriegen. Das Einzige, was ich machen *mus*s, wenn du es nicht tust, ist, Don zu informieren.«
- Mike: »Damit würdest du mich übergehen, und das würdest du nicht bringen.«
- Paula: »Das will ich auch nicht, Mike, aber ich tue es, wenn du mich dazu zwingst.«
- Mike: »Oh, Paula ...«
- Paula: »Hör mal, Mike, die Features werden *nicht* rechtzeitig für die Demo fertig. Das muss endlich in deinen Kopf. Hör auf damit, mich zu mehr Arbeit zu überreden. Hör auf, dich damit zu täuschen, dass ich irgendwie ein Kaninchen aus dem Zylinder zaubern werde. Stell dich der Tatsache, dass du Don informieren musst und dass du es *heute* machen musst.«
- Mike: (mit großen Augen) »Heute?«
- Paula: »Ja, Mike, heute. Weil ich davon ausgehe, dass ich morgen ein Meeting mit dir und Don haben werde, wo es um die Features geht, die in der Demo gezeigt werden sollen. Wenn dieses Meeting morgen nicht stattfindet, bin ich dazu gezwungen, selbst zu Don zu gehen. Hier ist eine Kopie des Memos, in der genau das erklärt wird.«

Mike: »Du willst bloß deinen Arsch retten!«

Paula: »Mike, ich versuche, unser aller Ärsche zu retten. Kannst du dir das Debakel vorstellen, wenn der Kunde herkommt und eine komplette Demo erwartet, und wir können die nicht liefern?«

Was passiert Paula und Mike am Ende? Ich überlasse es Ihnen, sich die verschiedenen Möglichkeiten auszumalen. Der Punkt ist, dass Paula sich sehr professionell verhalten hat. Sie ist bei ihrem Nein geblieben – in jedem richtigen Moment und stets auf die richtige Weise. Sie hat abgelehnt, als sie unter Druck gesetzt wurde, ihre Kalkulation anzupassen. Sie hat Nein gesagt, als sie durch Schmeicheln und Betteln manipuliert und überredet werden sollte. Und am Wichtigsten: Sie ist bei ihrem Nein geblieben, als ihr Mikes Selbsttäuschung und Inaktivität klar wurden. Paula hat für das Team gespielt. Mike brauchte Hilfe, und sie hat alles in ihrer Macht Stehende getan, um ihm zu helfen.

2.4 Die Kosten eines Ja

Die meiste Zeit wollen wir gerne Ja sagen. Tatsächlich legen es intakte Teams darauf an, Wege zu finden, um Ja zu sagen. Manager und Entwickler in gut geführten Teams werden so lange miteinander verhandeln, bis sie am Ende einen Aktionsplan haben, dem beide Seiten zustimmen können.

Aber wie wir gesehen haben, ist es manchmal der einzige Weg, um zum *richtigen* Ja zu kommen, keine Angst vor einem Nein zu haben.

Nehmen wir die folgende Geschichte, die John Blanco in seinem Blog² gepostet hat. Sie wird hier mit freundlicher Genehmigung abgedruckt. Bei der Lektüre sollten Sie sich fragen, wann und wie er hätte Nein sagen sollen.

Ist guter Code unmöglich?

Als Teenager beschlossen Sie, Software-Entwickler zu werden. Während der High-school lernten Sie, wie man anhand von objektorientierten Prinzipien Software schreibt. Als Sie Ihren Abschluss machten, haben Sie all die erlernten Prinzipien auf solche Bereiche wie künstliche Intelligenz oder 3D-Grafiken angewendet.

Und als Sie in die professionelle Laufbahn einstiegen, fing für Sie die niemals endende Suche an, Code in kommerzieller Qualität zu schreiben, der gut zu warten und einfach »perfekt« ist und sich darum im Laufe der Zeit bewährt.

Kommerzielle Qualität. Ha. Noch so einen Witz.

² <http://raptureinvenice.com/?p=63>

Ich halte mich für glücklich, weil ich Design-Pattern *liebe*. Ich liebe es, die Theorie der Perfektionierung von Code zu studieren. Ich habe kein Problem damit, eine stundenlange Diskussion darüber einzuleiten, warum die Entscheidung meines XP-Partners bei der Vererbungshierarchie falsch ist – warum es also in so vielen Fällen besser ist, mit HAS-A zu arbeiten statt mit IS-A. Aber seit einiger Zeit zerbreche ich mir immer wieder den Kopf und frage mich dann ...

... Ist guter Code in der modernen Software-Entwicklung unmöglich?

Der typische Projektvorschlag

Als Auftragsentwickler in Vollzeit (und Teilzeit) verbringe ich meine Tage (und Nächte) damit, mobile Applikationen für Kunden zu entwickeln. Und in den vielen Jahren dieser Tätigkeit habe ich gelernt, dass die Anforderungen der Arbeit für Kunden es für mich unmöglich machen, die qualitativ wirklich guten Apps zu schreiben, die mir gefallen.

Bevor ich beginne, will ich schon mal darauf hinweisen, dass es keinen Mangel an Versuchen gegeben hat. Ich liebe das Thema sauberer Code. Ich kenne niemanden, der sich so sehr bei perfektem Software-Design reinhängt wie ich. Es ist die Ausführung, die für mich schwerer zu fassen ist, aber nicht aus dem Grund, den Sie vielleicht annehmen.

Lassen Sie mich dazu eine Geschichte erzählen.

Gegen Ende letzten Jahres brachte eine ziemlich bekannte Firma ein RFP (Request for Proposal) heraus, eine App produzieren zu lassen. Dabei handelt es sich um eine riesige Einzelhandelskette, aber aus Gründen des Datenschutzes wollen wir sie hier Gorilla Mart nennen. Sie meinten, sie müssten eine iPhone-Präsenz erstellen, und bräuchten dafür eine App, die bis zum Black Friday (in den USA der Freitag nach Thanksgiving, traditionell ein Familienwochenende und Beginn der Weihnachtssaison) produziert sein müsste. Der Haken? An dem Tag war schon der 1. November. Damit bleiben noch vier Wochen, um die App zu erstellen. Ach, und in dieser Zeit braucht Apple immer noch zwei Wochen, um die Apps zu akzeptieren (ach, was waren das früher für schöne Tage ...). Okay, fürs Programmieren dieser App haben wir also ... ZWEI WOCHEN??!!

Ja. Wir haben zwei Wochen, um diese App zu schreiben. Und leider haben wir den Zuschlag für diesen Auftrag bekommen (im Business zählt nur die Relevanz des Kunden). Also muss das umgesetzt werden.

»Aber das ist okay«, sagt der Gorilla Mart-Manager #1. »Die App ist ganz einfach. Sie braucht dem User bloß ein paar Produkte aus unserem Katalog zu zeigen, und man muss nach Filialen suchen können. Das machen wir bereits auf unserer Website. Sie kriegen von uns auch die Grafiken. Das können Sie dann höchstwahrscheinlich ... wie heißt das doch gleich? Ach ja, hartkodieren!«

Gorilla Mart-Manager #2 ergreift das Wort: »Und wir brauchen dann nur noch ein paar Coupons, die der User an der Kasse vorzeigen kann. Die App ist quasi Wegwerfware. Bringen wir das schnell unter Dach und Fach, und dann machen wir für Phase II was von Grund auf Neues, das dann viel größer und besser wird.«

Und dann passiert es. Trotz all der jahrelang eintrudelnden Denkkärtchen, dass jedes Feature, das der Kunde haben will, immer deutlich komplexer zu schreiben ist, als es erklärt wird, greifen Sie zu. Sie glauben wirklich, dass es dieses Mal tatsächlich in zwei Wochen geschafft sein kann. Ja! Wir schaffen das! Dieses Mal ist es ganz anders! Das sind bloß ein paar Grafiken und ein Service-Aufruf, um den Standort von Filialen zu kriegen. XML! Kein Stress! Wir schaffen das! Ich steh schon voll unter Strom! Ran an die Tastatur!

Es dauert bloß einen Tag, bis Sie und die Realität mal wieder aufeinandertreffen.

Ich: Können Sie mir also bitte die Infos geben, die ich brauche, um den Webservice für Ihre Filialenstandorte aufzurufen?

Der Kunde: Was ist ein Webservice?

Ich:

Und genauso ist das auch wirklich passiert. Der Dienst für die Filialsuche befindet sich, wo er sein soll, auf der Website rechts oben in der Ecke – ist aber kein Webservice. Er wird von Java-Code generiert. Eine API? Fehlanzeige ... Und obendrein wird die Website von einem strategischen Partner von Gorilla Mart gehostet.

Es folgt der Auftritt des ominösen »Drittanbieters«.

Was den Kunden angeht, ähnelt der »Drittanbieter« sehr der wunderbaren Angelina Jolie. Trotz des Versprechens, dass man eine sehr anregende Unterhaltung bei einem wunderbaren Essen haben wird und sich hinterher noch was ergeben könnte ... tja, sorry, da läuft nichts. Man darf darüber einfach seinen Fantasien nachhängen, während man sich selbst um alles kümmert.

In meinem Fall war das Einzige, was ich Gorilla Mart abringen konnte, ein aktueller Snapshot ihrer momentanen Filialen in einer Excel-Datei. Den Code für die Filialsuche durfte ich von Grund auf neu schreiben.

Das doppelte Pech kam an diesem Tag noch etwas später: Sie wollten, dass die Daten über Produkte und Coupons online stehen, damit es wöchentlich geändert werden kann. Und Tschüss fürs Hartkodieren! Die beiden Wochen, die fürs Schreiben einer iPhone-App zur Verfügung standen, haben sich nun in zwei Wochen verwandelt, in denen eine iPhone-App geschrieben werden soll, dazu noch ein PHP-Backend, und beides muss miteinander integriert werden ... Was? Ich soll mich auch noch um die Qualitätssicherung kümmern?

Um die zusätzliche Arbeit auszugleichen, muss das Programmieren einfach noch ein wenig schneller laufen. Vergiss diese abstrakte Factory. Statt eines Composites nimm einfach eine große, fette Schleife – mehr Zeit bleibt nicht!

Guter Code wird so unmöglich.

Zwei Wochen bis Fertigstellung

Ich will Ihnen eines sagen: Diese beiden Wochen waren wirklich fürchterlich. Zum einen fielen zwei von diesen Tagen wegen ganztägiger Meetings für mein nächstes Projekt unter den Tisch (damit wurde das sowieso schon kleine Zeitfenster noch enger). Schließlich blieben mir de facto noch acht Tage, um alles zu erledigen. In der ersten Woche arbeitete ich 74 Stunden und in der nächsten ... Gott ... das weiß ich nicht mal mehr genau, weil es aus meinen Synapsen ausradiert ist. Wahrscheinlich auch besser so.

Diese acht Tage verbrachte ich damit, wie ein Irrer zu programmieren. Ich fuhr das ganze Instrumentarium auf, um fertigzuwerden: Copy & Paste (alias wiederverwendbarer Code), magische Zahlen (erspart mir die Duplizierung von definierenden Konstanten und dann auch das *keuch* erneute Eintippen) und absolut *keine* Unit-Tests! (Wer kann in einer solchen Zeit schon rote Balken gebrauchen? Hätte mich bloß demotiviert!)

Es war ziemlich schlechter Code, und zum Refakturieren blieb mir keine Zeit. Wenn man das Zeitfenster betrachtet, war er aber eigentlich ziemlich brilliant, und immerhin sollte es doch »Wegwerf«-Code sein, oder nicht? Hört sich das irgendwie vertraut an? Tja, warten Sie nur ab, es kommt noch besser.

Als ich bei der App den letzten Feinschliff vornahm (Feinschliff bedeutet hier, den gesamten Server-Code zu schreiben), schaute ich mir mal die Codebasis an und fragte mich, ob sich das nicht vielleicht auch gelohnt habe. Immerhin war die App fertig. Ich hatte es überlebt!

»Hey, gerade haben wir Bob angeheuert, und er ist sehr beschäftigt und konnte nicht zurückrufen, aber er sagt, wir sollten von den Usern die E-Mail-Adressen abfragen, damit sie die Coupons kriegen. Er hat die App noch nicht gesehen, aber er glaubt, das wäre eine tolle Idee! Wir wollen auch ein Berichtssystem, um diese E-Mails vom Server zu kriegen. Eines, das nett ist und nicht zu teuer. (Warte mal, der letzte Teil war Monty Python.) Da wir gerade von Coupons reden: Die sollen unbedingt nach einer bestimmten Zahl von Tagen verfallen, die wir jeweils angeben. Ach, und ...«

Gehen wir mal einen Schritt zurück. Was wissen wir darüber, was guten Code ausmacht? Guter Code sollte erweiterbar, gut zu warten sein. Er sollte sich für Modifikationen eignen. Er sollte sich wie Prosa lesen lassen. Tja, das hier war kein guter Code.

Noch eines: Wenn Sie ein besserer Entwickler sein wollen, müssen Sie sich Folgendes unauslöschlich einprägen: Der Kunde wird die Deadline immer nach hinten schieben. Er wird immer mehr Features haben wollen. Er wird immer noch was geändert haben wollen – *ganz kurzfristig*. Und hier ist die Formel für das, was Sie da erwarten können:

(Zahl der Manager)²

+ 2 * Zahl der neuen Manager

+ Zahl von Bobs Kindern

= TAGE, DIE IM LETZTEN MOMENT HINZUGEFGÜGT WERDEN

Nun, Manager sind anständige Leute. Davon gehe ich jedenfalls aus. Sie kümmern sich um ihre Familie (vorausgesetzt, Satan hat zugelassen, dass sie eine haben). Sie wollen, dass die App erfolgreich ist (schlägt für die Beförderung positive zu Buche!). Das Problem ist, dass alle direkt vom Erfolg des Projekts profitieren wollen. Sie wollen letzten Endes auf jeden Fall auf ein bestimmtes Feature oder eine Designentscheidung zeigen und sagen können, die stamme direkt von ihnen.

Also zurück zu unserer Geschichte: Wir hängten noch ein paar Tage ans Projekt und bauten das E-Mail-Feature mit ein. Und dann brach ich vor Erschöpfung zusammen.

Den Kunden ist das alles nie so wichtig wie Ihnen

Den Kunden ist es trotz all ihrer Proteste und trotz der scheinbaren Dringlichkeit nie so wichtig wie Ihnen, dass die App rechtzeitig fertig ist. An dem Nachmittag, als ich die App als vollendet bezeichnen durfte, schickte ich allen Stakeholdern, Geschäftsführern (*fauch!*), Managern usw. eine E-Mail mit dem finalen Build. »ES IST VOLLBRACHT! ICH BRINGE EUCH V1.0! GEPRIESEN SEI DEIN NAME.« Ich klickte auf *Senden*, lehnte mich in meinem Stuhl zurück und begann selbstgefällig grinsend darüber zu fantasieren, wie mich die Firmenleute auf ihren Schultern tragen und mir zu Ehren eine Prozession die 42. Straße hinunter führen würden, bei der ich zum »allergrößten Entwickler überhaupt« gekrönt werde. Zumindest würde nun mein Gesicht in der gesamten Firmenwerbung erscheinen, nicht wahr?

Seltsamerweise fand das scheinbar nicht so deren Zustimmung. Tatsächlich war ich gar nicht mehr so sicher, was sie dachten. Ich hörte nichts. Keinen Mucks. Dann stellte sich heraus, dass sich die Leute vom Gorilla Mart bereits auf das nächste Projekt gestürzt haben.

Sie glauben, ich lüge? Prüfen Sie es selbst nach. Ich hatte die App in den Apple Store hochgeladen, ohne eine App-Beschreibung beizulegen. Ich hatte eine vom Gorilla Mart angefordert, aber man hatte sich nicht bei mir zurückgemeldet, und fürs War-

ten blieb keine Zeit mehr (siehe voriger Absatz). Ich schrieb ihnen erneut. Und dann noch mal. Ich setzte jemanden von unserem Management darauf an. Zweimal meldete sich einer, und beide Mal hörte ich: »Was wollten Sie noch gleich haben?« ICH BRAUCHE DIE APP-BESCHREIBUNG!

Eine Woche später begann Apple mit dem Testen der App. Das ist normalerweise eine Zeit der Freude, aber diesmal eine Phase tödlicher Furcht. Wie erwartet wurde die App später noch am gleichen Tag abgelehnt. Es war die traurigste, ärmlichste Begründung, die ich mir für eine Ablehnung vorstellen kann: »Der App fehlt die Beschreibung.« Funktioniert perfekt, hat aber keine App-Beschreibung. Und aus diesem Grund war die App für Gorilla Mart am Black Friday nicht bereit. Ich war erschüttert.

Ich hatte mein Familienleben für einen zweiwöchigen Supersprint geopfert, und niemand bei Gorilla Mart kümmerte sich irgendwie darum, dass binnen einer Woche eine App-Beschreibung geliefert wird. Sie traf bei uns eine Stunde nach der Ablehnung ein – offenbar war dies das Signal, sich wieder ans Geschäft zu machen.

Wenn man meinen Zustand vorher als erschüttert bezeichnen konnte, war ich danach anderthalb Wochen lang außer mir vor Wut. Wissen Sie, die hatten uns immer noch keine echten Daten gegeben. Die Produkte und Coupons auf dem Server waren alle gefakt. Erfunden. Der Couponcode lautete 1234567890. Sie wissen schon, einfach total erfundener Quatsch!

Und an diesem schicksalsträchtigen Morgen prüfte ich das Portal und DIE APP WAR ERHÄLTlich! Mit gefakten Daten und dem ganzen Rest! In meinem hoffnungslosen Horror schrie ich auf und rief einfach irgendwen an, der den Hörer abnahm, und brüllte ihm ins Ohr: »ICH BRAUCHE DIE DATEN!« Die Frau am anderen Ende fragte mich, ob ich die Feuerwehr oder die Polizei brauche. Also unterbrach ich den Anruf bei 911. Aber ich rief dann Gorilla Mart an, und ich rief dann noch mal: »ICH BRAUCHE DIE DATEN!« Und die Antwort werde ich nie im Leben vergessen:

»Ah, hallo John. Wir haben einen neuen Vizepräsidenten und uns gegen die Veröffentlichung entschlossen. Nehmen Sie es doch bitte schön aus dem App Store, okay?«

Am Ende stellte sich heraus, dass sich ganze elf Personen in der Datenbank mit ihren E-Mail-Adressen registriert hatten. Das bedeutete, es konnten potenziell elf Leute in einen Gorilla Mart spazieren und dort einen gefakten iPhone-Coupon vorzeigen. Junge, das könnte aber hässlich werden.

Als dann letzten Endes alles vorbei war, hatte der Kunde tatsächlich eine Sache wirklich richtig festgestellt: Der Code war zum Wegwerfen. Das einzige Problem war, dass er überhaupt nie veröffentlicht worden war.

Resultat? Hektik beim Fertigstellen, Schneckentempo auf dem Weg zum Markt

Die Lektion aus dieser Geschichte lautet, dass Ihre Stakeholder (egal ob es sich um externe Kunden oder das interne Management handelt) herausgeknien haben, wie man Entwickler dazu bringt, Code möglichst schnell zu schreiben. Effektiv? Nein. Schnell? Definitiv. So funktioniert das:

- **Erzähl dem Entwickler, dass die App simpel ist.** Das dient dazu, das Entwicklungsteam druckvoll in eine falsche Einstellung zu drängen. Es sorgt auch dafür, dass die Entwickler früher mit der Arbeit beginnen, wodurch sie ...
- **Features einfügen, indem man dem Team vorwirft, die Notwendigkeit nicht erkannt zu haben.** In diesem Fall konnte der festkodierte Inhalt nur über App-Updates geändert werden. Wie konnte ich das nur übersehen? Ich hatte es nicht übersehen, aber mir war vorher ein falsches Versprechen gemacht worden, das war der Grund. Oder ein Kunde stellt einen »neuen Mitarbeiter« ein, der erkennt, dass da offenbar etwas übersehen wurde. Eines Tages verkündet ein Kunde, dass man gerade Steve Jobs eingestellt habe und ob man mal eben Alchemie in die App einbauen könne? Dann werden sie ...
- **Die Deadline verschieben. Immer wieder.** Entwickler arbeiten am schnellsten und intensivsten (und sind dann übrigens auch am fehlerträchtigsten, aber wen kümmert das schon, nicht wahr?), wenn sie nur noch wenige Tage vor einer Deadline stehen. Warum sollte man ihnen dann auch sagen, dass sich der Termin nach hinten verschoben hat, wenn sie gerade so produktiv sind? Das sollte man doch eher ausnutzen! Und das läuft dann so ab: Ein paar Tage werden angehängt, dann eine Woche, gerade als Sie nach einer 20-Stunden-Schicht endlich froh waren, alles richtig hingekriegt zu haben. Das ist so wie bei einem Esel, dem man eine Möhre vors Maul hält – außer dass Sie nicht so gut behandelt werden wie ein Esel.

Es ist ein brillantes Szenario. Kann man ihnen ernsthaft vorwerfen zu glauben, dass es funktioniert? Aber sie kriegen den höllisch schlechten Code nicht zu sehen. Und so passiert das immer und immer wieder trotz der Resultate.

In einer globalisierten Ökonomie, wo die Unternehmen dem allmächtigen Dollar unterworfen sind und zur Steigerung des Aktienkurses Entlassungen, überarbeitetes Personal und Verlagerung ins Ausland gehören, wird guter Code durch die von mir hier vorgestellte Strategie, um Entwicklungskosten zu reduzieren, obsolet. Als Entwickler werden wir gebeten/aufgefordert/hereingelegt, um den doppelten Code in der halben Zeit zu schreiben, wenn wir nicht vorsichtig sind.

2.5 Code unmöglich

Als John in seiner Geschichte fragt: »Ist guter Code unmöglich?«, meint er eigentlich: »Ist Professionalität unmöglich?« Immerhin hat in seiner Geschichte über Dysfunktion nicht nur der Code gelitten. Es waren seine Familie, sein Arbeitgeber, sein Kunde und die User. Jeder hat bei diesem Abenteuer verloren³. Und sie haben wegen der Unprofessionalität verloren.

Wer hat sich denn nun unprofessionell verhalten? John macht deutlich, dass er glaubt, es seien die Geschäftsführer des Gorilla Mart. Immerhin ist sein Szenario eine ziemlich deutliche Anklage ihres schlechten Verhaltens. Aber war ihr Verhalten so schlecht? Ich glaube nicht.

Die Leute vom Gorilla Mart wollten die Option, am Black Friday eine iPhone-App zu haben. Sie waren bereit, für diese Option zu zahlen. Sie fanden jemanden, der bereit war, diese Option zu bieten. Was gibt es daran auszusetzen?

Ja, es stimmt, es hat einige Kommunikationsfehler gegeben. Offensichtlich hatten die Geschäftsführer keine Ahnung, was ein Webservice wirklich ist, und es gab all die normalen Probleme damit, dass die eine Abteilung in einer großen Firma keine Ahnung hat, was die andere macht. Aber all das hätte man erwarten können. John gibt das sogar zu, wenn er sagt: »Trotz all der jahrelang eintrudelnden Denkkärtchen, dass jedes Feature, das der Kunde haben will, immer deutlich komplexer zu schreiben ist, als es erklärt wird ...«

Wenn der Übeltäter also nicht Gorilla Mart ist, wer dann?

Möglicherweise war es Johns direkter Arbeitgeber. John hat dies nicht explizit gesagt, aber es gab einen Hinweis darauf, als er in Klammern einfügt: »Im Business zählt nur die Relevanz des Kunden.« Hat also Johns Arbeitgeber bei Gorilla Mart unvernünftige Versprechen abgegeben? Haben sie Druck auf John ausgeübt, direkt oder indirekt, um diese Versprechen umzusetzen? John sagt nichts dazu, also können wir nur mutmaßen.

Und trotzdem: Wo bleibt Johns Verantwortung bei alledem? Ichbürde die Schuld voll und ganz John auf. John ist derjenige, der anfangs die Deadline in zwei Wochen akzeptierte, obwohl er eigentlich wusste, dass Projekte normalerweise viel komplexer sind, als sie klingen. John ist derjenige, der die Notwendigkeit akzeptierte, einen PHP-Server zu schreiben. John ist derjenige, der die E-Mail-Registrierung sowie das Verfallsdatum der Coupons akzeptierte. John ist derjenige, der 20 Stunden täglich und 90 Stunden pro Woche arbeitet. John ist derjenige, der sich von seiner Familie und seinem Privatleben abkoppelte, um diese Deadline zu schaffen.

³ Möglicherweise stellt Johns Arbeitgeber hier eine Ausnahme dar ... aber ich gehe jede Wette ein, dass der auch verloren hat.

Und warum hat John so gehandelt? Er formuliert es in diesen deutlichen Sätzen: »Ich klickte auf *Senden*, lehnte mich in meinem Stuhl zurück und begann selbstgefällig grinsend darüber zu fantasieren, wie mich die Firmenleute auf ihren Schultern tragen und mir zu Ehren eine Prozession die 42. Straße hinunter führen würden, bei der ich zum »allergrößten Entwickler überhaupt« gekrönt werde.« Kurz gesagt: John versuchte, ein Held zu sein. Er sah seine Chance für Ruhm und Ehre und legte es darauf an. Er beugte sich vor und versuchte, seine Chance zu erhaschen.

Profis sind oft Helden, aber nicht, weil sie es darauf anlegen. Profis werden zu Helden, wenn sie einen Auftrag gut, rechtzeitig und im budgetierten Rahmen erledigen. Indem John versuchte, der Mann der Stunde und der Held des Tages zu sein, verhielt er sich nicht wie ein Profi.

John hätte schon das Zeitfenster von zwei Wochen ablehnen sollen. Oder wenn nicht, dann hätte er Nein sagen sollen, als er herausfand, dass es keinen Webservice gibt. Er hätte die Anfrage nach der E-Mail-Registrierung und das Ablaufdatum bei den Coupons ablehnen sollen. Er hätte alles ablehnen sollen, was horrende Überstunden und Aufopferung erfordert.

Aber am wichtigsten ist, dass John zu seiner eigenen internen Entscheidung hätte Nein sagen sollen, dass man diesen Job einzig und allein dadurch rechtzeitig schaffen kann, dass man ein großes Chaos anrichtet. Achten Sie darauf, was John über guten Code und Unit-Tests sagt:

»Um die zusätzliche Arbeit auszugleichen, muss das Programmieren einfach noch ein wenig schneller laufen. Vergiss diese abstrakte Factory. Statt eines Composites nimm einfach eine große, fette Schleife – mehr Zeit bleibt nicht!«

Und dann noch mal:

»Diese acht Tage verbrachte ich damit, wie ein Irrer zu programmieren. Ich fuhr das ganze Instrumentarium auf, um fertigzuwerden: Copy & Paste (alias wiederverwendbarer Code), magische Zahlen (erspart mir die Duplizierung von definierenden Konstanten und dann auch das *keuch* erneute Eintippen) und absolut *keine* Unit-Tests! (Wer kann in einer solchen Zeit schon rote Balken gebrauchen? Hätte mich bloß demotiviert!)«

Dass er zu diesen Entscheidungen Ja gesagt hat, war die wahre Krux des Misserfolgs. John akzeptierte, dass der einzige Weg, erfolgreich zu sein, über ein unprofessionelles Verhalten führt, also bekommt er einzig und allein den verdienten Lohn.

Das klingt vielleicht hart. Es ist nicht so gedacht. In den vorigen Kapiteln habe ich beschrieben, wie ich – mehr als einmal – den gleichen Fehler auch in meiner Karriere gemacht habe. Die Versuchung, den Helden zu spielen und »das Problem zu lösen«, ist riesengroß. Wir alle müssen realisieren, dass es nicht der Lösung unserer Probleme dient, wenn wir bereitwillig unsere professionelle Disziplin über Bord werfen. Auf diese Weise schafft man überhaupt erst Probleme.

Damit kann ich schließlich Johns Frage vom Anfang beantworten:

■ Ist guter Code unmöglich? Ist Professionalität unmöglich?«

■ Antwort: Ich sage *Nein*.



Wussten Sie, dass ich Voicemail erfunden habe? Das stimmt wirklich. Tatsächlich besitzen wir das Patent für Voicemail zu dritt: Ken Finder, Jerry Fitzpatrick und ich. Das war ganz zu Anfang der 1980er-Jahre, und wir arbeiteten damals für eine Firma namens Teradyne. Unser CEO hatte uns beauftragt, uns ein neues Produkt auszudenken, und wir erfanden den »Elektronischen Rezeptionisten«, kurz ER.

Sie alle wissen, wer dieser ER ist. ER ist eine jener furchtbaren Maschinen, die bei den Firmen die Telefonate beantworten und Ihnen alle möglichen hirnrissigen Fragen stellt, die Sie durch Drücken der Tasten beantworten müssen (»Für Deutsch drücken Sie die 1«).

Unser ER sollte für eine Firma die Telefonate beantworten und den Anrufer auffordern, den Namen der gewünschten Person über die Tastatur zu wählen. ER sollte Sie bitten, Ihren Namen auszusprechen, und dann die gewünschte Person anrufen. ER sollte die

Verbindung aufbauen und fragen, ob der Anruf akzeptiert wird. Falls ja, sollte ER die Verbindung herstellen und sich dann ausklinken.

Man konnte ER sagen, wo man gerade war. Man konnte ihm mehrere Telefonnummern zum Ausprobieren geben. Wenn man sich also in einem anderen Büro befand, konnte ER einen finden. War man zu Hause, konnte ER einen auch dort finden. Wenn man sich in einer anderen Stadt aufhielt, konnte ER einen auch dort finden. Und wenn ER einen am Ende doch nicht gefunden hatte, konnte das System eine Nachricht aufnehmen. An dieser Stelle kam Voicemail ins Spiel.

Eigenartigerweise hatte Teradyne keine Ahnung, wie man ER verkaufen könnte. Dem Projekt ging das Geld aus, und es wurde in etwas umgewandelt, von dem wir wussten, wie wir das an den Mann bringen konnten: CDS, das *Craft Dispatch System*, das die Aufträge der Mechaniker für Telefonreparaturen verteilte und organisierte. Teradyne ließ außerdem das Patent auslaufen, ohne uns zu informieren (!). Der aktuelle Patentinhaber meldete seines drei Monate nach uns an. (!!!)¹

Das war lange, nachdem ER in CDS umgewandelt wurde, aber deutlich bevor ich herausfand, dass das Patent ausgelaufen war. Ich wartete auf einem Baum auf den CEO der Firma. Vor unserem Gebäude stand eine große Eiche. Dort stieg ich hinauf und wartete, bis sein Jaguar auf den Parkplatz einbog. Ich traf ihn an der Tür und bat um ein paar Minuten. Er tat mir den Gefallen.

Ich sagte ihm, dass wir das ER-Projekt unbedingt wieder neu beleben müssten. Ich sagte ihm, ich sei total sicher, dass man damit Geld verdienen könne. Er überraschte mich, indem er sagte: »Okay, Bob, arbeiten Sie einen Plan aus. Zeigen Sie mir, wie ich damit Geld verdienen kann. Wenn Ihnen das gelingt und ich Ihnen glaube, greife ich ER wieder neu auf.«

Das hatte ich nicht erwartet. Ich war davon ausgegangen, dass er sagt: »Sie haben recht, Bob. Ich werde das Projekt neu starten und herausfinden, wie man damit Geld verdienen kann.« Aber nein, er bürdete mir die Last auf. Und bei dieser Last war ich eher ambivalent. Schließlich war ich für Software zuständig und nicht fürs Geldverdienen. Ich wollte im ER-Projekt arbeiten, aber nicht für Profit und Verluste verantwortlich sein. Aber ich wollte ihm meine Ambivalenz nicht zeigen. Also bedankte ich mich und verließ sein Büro mit den Worten:

»Danke, Russ. Dann hänge ich mich da mal rein ... glaube ich.«

Nun will ich Ihnen Roy Osherove vorstellen, der Ihnen sagen wird, wie erbärmlich diese Aussage war.

¹ Nicht, dass das Patent mir irgendwie Geld eingebracht hatte. Ich hatte es im Rahmen meines Arbeitsvertrages für 1 Dollar an Teradyne verkauft (und den habe ich auch nicht bekommen).

3.1 Verbindliche Sprache

Von Roy Osherove

Sagen. Meinen. Machen.

Eine verbindliche Aussage, also ein Commitment, besteht aus drei Teilen:

1. Sie *sagen*, dass Sie es machen werden.
2. Sie *meinen* es.
3. Sie *machen* es dann auch tatsächlich.

Doch wie oft begegnen wir Menschen (uns natürlich ausgenommen!), die nie all diese drei Stufen durchlaufen?

- **Sie fragen den IT-Kollegen**, warum das Netzwerk so langsam ist, und er antwortet: »Tja, wir brauchen echt ein paar neue Router.« Und Sie *wissen*, dass in dieser Kategorie niemals irgendwas passieren wird.
- **Sie bitten einen Teamkollegen**, ein paar manuelle Tests durchzuführen, bevor er den Quellcode eincheckt, und er erwidert: »Sicher, dazu komme ich wahrscheinlich bis zum Feierabend.« Und irgendwie *spüren* Sie sehr deutlich, dass Sie morgen noch einmal danach fragen müssen, ob vor dem Einchecken tatsächlich irgendwelche Tests gelaufen sind.
- **Ihr Chef** wandert beiläufig ins Zimmer und murmelt: »Wir müssen uns schneller bewegen.« Und Sie *wissen*, dass er in Wirklichkeit meint, dass SIE sich schneller bewegen müssen. *Er* wird keinen Handschlag dafür machen.

Es gibt nur sehr wenige Leute, die, wenn sie etwas sagen, das dann auch wirklich meinen und es tatsächlich umsetzen. Manche sagen etwas und meinen es auch, aber kriegen es nicht umgesetzt. Und es gibt noch weitaus mehr, die etwas versprechen und nicht einmal *meinen*, dass sie das machen werden. Haben Sie schon mal jemanden sagen hören: »Junge, Junge, ich muss wirklich abnehmen«, und Sie wissen, dass er dafür kein Stück machen wird? So etwas passiert dauernd.

Warum bekommen wir dieses seltsame Gefühl, dass sich die Leute meistens nicht wirklich verpflichtet fühlen, etwas umzusetzen?

Schlimmer noch: Oft verlässt uns auch unsere Intuition. Manchmal möchten wir nur zu gerne glauben, dass jemand meint, was er sagt, obwohl das gar nicht stimmt. Wir *wollen* gerne dem Entwickler glauben, wenn er – mit dem Rücken an die Wand gedrängt – sagt, dass er diese Aufgabe, für die zwei Wochen nötig sind, auch in einer hinkriegt, aber das sollten wir nicht.

Anstatt nach unserem Bauchgefühl zu gehen, können wir ein paar auf die Sprache bezogene Tricks nutzen, um tatsächlich herauszufinden, ob jemand auch wirklich meint, was er sagt. Und indem wir unsere eigenen Formulierungen ändern, können wir uns auf eigene Faust um die Schritte 1 und 2 in der obigen Liste kümmern, wo wir *sagen*, dass wir uns für etwas verbindlich verpflichten, und das dann auch wirklich *meinen*.

3.1.1 So erkennt man mangelnde Selbstverpflichtung

Wir sollten uns die Sprache anschauen, die wir einsetzen, wenn wir uns für etwas »committen«, denn »an ihrer Sprache sollt ihr sie erkennen«, um das mal etwas biblisch auszudrücken. Tatsächlich geht es eher darum, auf *bestimmte Wörter* in dem zu achten, was wir sagen. Wenn Sie diese kleinen magischen Wörter nicht finden, ist die Wahrscheinlichkeit groß, dass wir nicht meinen, was wir sagen, oder es eigentlich nicht für umsetzbar halten.

Hier sind einige Beispiele für Wörter und Redewendungen, in denen man nach verräterischen Anzeichen suchen kann, dass jemand sich nicht festlegen will.

- **Sollte/müsste.** »Wir sollten das fertigkriegen.« »Ich müsste abnehmen.« »Jemand sollte sich darum kümmern.«
- **Hoffe/wünsche.** »Ich hoffe, das bis morgen erledigt zu haben.« »Ich hoffe, wir können uns eines Tages mal wieder treffen.« »Ich wünschte, ich hätte Zeit dafür.« »Ich wünschte, dieser Computer wäre schneller.«
- **Wir können** [allgemein formuliert] ... »Wir können uns ja mal wieder treffen.« »Wir können das ja mal fertigstellen.« »Das können wir im Auge behalten.«

Wenn Sie damit anfangen, nach diesen Wörter zu suchen und darauf zu achten, werden Sie merken, dass sie Ihnen immer wieder über den Weg laufen, sogar in Ihren eigenen Aussagen anderen gegenüber.

Sie werden merken, dass wir uns sehr darum kümmern, keine Verantwortung für irgendwas zu übernehmen.

Und das ist *nicht* okay, wenn Sie oder jemand anderes sich auf diese Versprechen verlässt als Teil eines Jobs. Sie haben allerdings schon den ersten Schritt gemacht: Sie fangen an, auf die mangelnde Selbstverpflichtung um Sie herum und bei sich selbst zu achten.

Wir haben gehört, wie es klingt, wenn sich jemand nicht festlegen will. Aber wie erkennen wir echte Verbindlichkeit, ein echtes Commitment?

3.1.2 Wie echte Selbstverpflichtung klingt

Was die Formulierungen aus dem vorigen Abschnitt gemeinsam haben, ist, dass sie entweder von Dingen ausgehen, die nicht in »der eigenen Hand« liegen oder keine persönliche Verantwortung übernehmen. In all diesen Fällen verhalten sich die Leute, als wären sie *Opfer* einer Situation anstatt sie zu kontrollieren.

Die Wahrheit ist, dass *Sie ganz persönlich* IMMER etwas haben, das Sie selbst steuern und beeinflussen. Also gibt es *immer* etwas, für dessen Umsetzung Sie verbindliche Zusagen treffen können.

Das Geheimrezept zum Erkennen eines echten Commitments ist, auf solche Formulierungen wie folgt zu achten: »Ich werde ... bis ... « (z.B.: Ich werde das bis Dienstag fertig haben.)

Was ist an diesem Satz so bedeutsam? *Sie benennen eine Tatsache über etwas, das Sie selbst machen werden, mit einem klaren Enddatum.* Sie sprechen über *niemand anderen* als nur von sich selbst. Sie sprechen über eine *Handlung*, die Sie unternehmen *werden*. Sie werden das nicht »wahrscheinlich« umsetzen oder »möglicherweise dazu kommen«, sondern es *definitiv* umsetzen.

Es gibt (technisch gesehen) keine Möglichkeit, sich um dieses verbale Commitment herumzumogeln. Sie haben gesagt, dass Sie es tun werden, und nun bleibt nur noch ein binäres Resultat übrig: Entweder schaffen Sie es oder eben nicht. Wenn Sie es nicht machen, kann man Sie auf Ihre Zusage festnageln. Sie werden ein *schlechtes Gewissen* haben, wenn Sie es nicht erledigt haben. Es wird Ihnen *peinlich* sein, jemandem sagen zu müssen, dass Sie es nicht geschafft haben (wenn dieser Jemand gehört hat, dass Sie es versprochen hatten).

Gruselig, nicht wahr?

Sie übernehmen die volle Verantwortung für etwas, und zwar vor einem Publikum, das aus mindestens einer Person besteht. Sie stehen da nicht alleine vor dem Spiegel oder sitzen vor dem Monitor. Es sind Sie selbst: Sie schauen jemandem ins Gesicht und sagen ihm, Sie machen es. Damit beginnt die Selbstverpflichtung. Sie bringen sich in eine Situation, die Sie zum Handeln zwingt.

Sie haben Ihre Sprache verändert, um eine verbindliche Sprache zu nutzen, und das wird Ihnen dabei helfen, die nächsten beiden Stufen durchzustehen: es zu meinen und dann auch umzusetzen.

Hier sind ein paar Gründe, warum Sie etwas nicht *meinen* könnten oder umsetzen wollen, und dazu ein paar passende Lösungen.

Es funktioniert nicht, weil ich mich dafür auf Person X verlassen muss.

Sie können sich nur für Dinge committen, die Sie *komplett selbst* kontrollieren. Wenn es beispielsweise Ihr Ziel ist, ein Modul fertigzustellen, das außerdem von der Arbeit eines anderen Teams abhängt, können Sie sich nicht committen, das Modul zu vollenden, ohne das andere Team zu integrieren. Aber Sie *können* sich für bestimmte Handlungen committen, die Sie an Ihr Ziel bringen. Sie können z.B.:

- sich mit Gary aus dem Infrastrukturteam eine Stunde Zeit nehmen, um die jeweiligen Abhängigkeiten zu verstehen, nach denen Sie sich zu richten haben.
- ein Interface erstellen, das die Abhängigkeiten Ihres Moduls von der Infrastruktur des anderen Teams abstrahiert.
- sich mindesten dreimal diese Woche mit dem für den Build zuständigen Kollegen treffen, um sicherzustellen, dass Ihre Änderungen im Build-System der Firma gut funktionieren.
- Ihr eigenes, ganz persönliches Build erstellen, mit dem Sie Ihre Integrationstests für das Modul laufen lassen.

Erkennen Sie den Unterschied?

Wenn das Endziel von jemand anderem abhängt, könnten Sie sich für spezielle Aktionen committen, die Sie näher an dieses Ziel bringen.

Es wird nicht funktionieren, weil ich nicht genau weiß, ob man es machen kann.

Wenn man es nicht machen kann, können Sie sich immer noch für Aktionen committen, die Sie näher ans Ziel bringen. Herauszufinden, ob es umsetzbar ist, könnte eine der Aktionen sein, für die man sich committen kann!

Anstatt verbindlich zu erklären, vor dem Release noch all die 25 restlichen Bugs zu beheben (was vielleicht gar nicht möglich ist), können Sie sich für diese speziellen Aktionen committen, die Sie diesem Ziel näher bringen:

- Gehen Sie all die 25 Bugs durch und versuchen Sie, sie zu reproduzieren.
- Treffen Sie sich mit der Qualitätssicherung, die jeden Bug gefunden hat, um eine Reproduktion dieses Bugs zu sehen.
- Verbringen Sie in dieser Woche möglichst viel Zeit damit, jeden Bug zu fixen.

Es wird nicht funktionieren, weil ich es manchmal einfach nicht schaffe.

Das kann sein. Manchmal passieren unerwartete Sachen, so ist das eben im Leben. Aber Sie sollten immer noch die Erwartungen erfüllen wollen. In diesem Fall wird es Zeit, die Erwartungen zu ändern – *so schnell wie möglich*.

Wenn Sie Ihr Commitment nicht schaffen können, ist das Wichtigste, die Person zu alarmieren, der gegenüber Sie sich verpflichtet haben.

Je eher Sie alle Stakeholder alarmieren, desto eher kann das Team innehalten, die aktuell zu erledigenden Aktionen neu beurteilen und beschließen, ob etwas gemacht oder geändert werden kann (z.B. hinsichtlich der Prioritäten). Wenn Sie das machen, kann Ihr Commitment immer noch erfüllt werden, oder Sie können es in ein anderes umwandeln.

Hier einige Beispiele dafür:

- Sie haben sich mit einem Kollegen in einem Cafe in der Innenstadt verabredet, und die Straßen sind so verstopft, dass fraglich ist, ob Sie Ihre Zusage noch pünktlich einhalten können. Sie können Ihren Kollegen anrufen, sobald Sie die Verspätung abschätzen können, und ihn informieren. Vielleicht finden Sie einen näher gelegenen Treffpunkt oder verschieben das Meeting.
- Wenn Sie zugesagt haben, einen Bug zu fixen, den Sie für behebbar hielten, und erkennen müssen, dass dieser Bug deutlich fieser ist als angenommen, können Sie Alarm schlagen. Das Team kann sich dann für einen Aktionsplan entscheiden, um dieses Commitment zu halten (Pairing, sich von potenziellen Lösungen Anregungen holen, Brainstorming), oder die Priorität ändern, damit Sie sich einen anderen, einfacheren Bug vornehmen.

Ein wichtiger Punkt hier lautet: Wenn Sie nicht jemanden möglichst frühzeitig über das potenzielle Problem informieren, unterbinden Sie die Chance, dass Ihnen jemand dabei helfen kann, Ihr Commitment einzuhalten.

3.1.3 Zusammenfassung

Eine verbindliche Sprache zu schaffen, mag einen erst einmal etwas erschrecken, aber es kann dabei helfen, viele der Kommunikationsprobleme zu lösen, denen sich Programmierer heutzutage stellen müssen: Kalkulationen, Deadlines und Kommunikationsspannen im persönlichen Gespräch. Sie werden als ernst zu nehmender Entwickler angesehen, dessen Wort etwas gilt, und das ist in unserer Branche eines der besten Dinge, auf die man hoffen kann.

3.2 Lernen, wie man »Ja« sagt

Ich bat Roy darum, diesen Artikel für das Buch beizusteuern, weil er in mir eine Saite zum Klingen brachte. Ich predige schon eine ganze Zeit, wie man lernt, Nein zu sagen. Aber genauso wichtig ist es zu lernen, wie man Ja sagt.

3.2.1 Die Kehrseite von »Ich versuch's mal«

Stellen wir uns einmal vor, dass Peter für einige Modifikationen an der Rating-Engine verantwortlich ist. Für sich persönlich hat er kalkuliert, dass er für diese Modifikationen fünf oder sechs Tage braucht. Er schätzt außerdem, dass das Erstellen der Dokumentation für diese Modifikationen ein paar Stunden dauert. Am Montagmorgen fragt ihn seine Managerin Marge nach dem Stand der Dinge.

Marge: »Peter, werden Sie es bis Freitag geschafft haben, diese Rating-Engine zu überarbeiten?«

Peter: »Ich glaube, das ist machbar.«

Marge: »Ist da auch die Dokumentation mit drin?«

Peter: »Ich werde versuchen, das ebenfalls fertigzukriegen.«

Vielleicht hört Marge das Zaudern in Peters Aussagen nicht, aber er geht definitiv keine sonderlich verbindliche Festlegung ein. Marge stellt Fragen, die boolesche Antworten verlangen, aber Peters boolesche Reaktion ist eher *fuzzy*.

Beachten Sie den Missbrauch des Wortes »versuchen«. Im vorigen Kapitel haben wir für »versuchen« die Definition »besondere Mühe aufwenden« verwendet. Hier arbeitet Peter mit der Definition »vielleicht ja, vielleicht nein«.

Peter wäre besser dran, wenn er wie folgt reagieren würde:

Marge: »Peter, werden Sie es bis Freitag geschafft haben, diese Rating-Engine zu überarbeiten?«

Peter: »Möglich, kann aber auch Montag werden.«

Marge: »Ist da auch die Dokumentation mit drin?«

Peter: »Für die Dokumentation brauche ich noch ein paar Stunden zusätzlich. Also ist Montag drin, könnte aber auch bis Dienstag dauern.«

In diesem Fall ist Peters Formulierung ehrlicher. Er beschreibt Marge seine eigene Ungewissheit. Marge könnte dann in der Lage sein, mit dieser Ungewissheit umzugehen. Andererseits ist sie das vielleicht auch nicht.

3.2.2 Der Disziplin verpflichtet

Marge: »Peter, ich brauche ein definitives Ja oder Nein. Werden Sie Freitag mit der Rating-Engine und der Dokumentation dazu fertig sein oder nicht?«

Es ist vollkommen fair, dass Marge eine solche Frage stellt. Sie muss einen Zeitplan einhalten und braucht zum Thema Freitag eine binäre Antwort. Wie soll Peter reagieren?

Peter: »In diesem Fall muss ich leider Nein sagen, Marge. Ich kann ganz sicher sagen, dass ich mit den Änderungen und der Doku am Dienstag fertig sein werde – eher nicht.«

Marge: »Dann ist der Dienstag für Sie also absolut verbindlich?«

Peter: »Ja, am Dienstag werde ich alles definitiv fertig haben.«

Aber was ist, wenn Marge die Modifikationen zusammen mit der Dokumentation wirklich schon am Freitag braucht?

Marge: »Peter, mit Dienstag kriege ich ein echtes Problem. Willy, unser technischer Autor, steht uns am Montag zur Verfügung. Er hat dann fünf Tage, um den Benutzerleitfaden fertigzustellen. Wenn ich die Dokumentation für die Rating-Engine nicht bis Montagmorgen habe, kriegt er das Handbuch nicht rechtzeitig fertig. Können Sie die Doku zuerst machen?«

Peter: »Nein, die Änderungen müssen zuerst fertig sein, weil wir die Doku aus dem Output der Testläufe generieren.«

Marge: »Tja, gibt es irgendeinen Weg, wie Sie vor Montagmorgen mit den Änderungen und der Doku fertigwerden?«

Nun ist Peters Entscheidung gefordert. Es ist recht wahrscheinlich, dass er mit den Modifikationen der Rating-Engine schon Freitag durch ist, und er könnte auch die Doku schaffen, bevor er sich ins Wochenende verabschiedet. Er *könnte* auch am Samstag noch ein paar Stunden dranhängen, wenn sich das länger als gedacht hinzieht. Was soll er nun also Marge sagen?

Peter: »Schauen Sie, Marge, es ist recht wahrscheinlich, dass ich bis Montagmorgen alles fertigkriege, wenn ich am Samstag noch ein paar Stunden dranhänge.«

Ist damit das Problem von Marge gelöst? Nein, nur die Wahrscheinlichkeit verändert, und genau das muss Peter ihr sagen.

Marge: »Kann ich dann auf Montagmorgen zählen?«

Peter: »Wahrscheinlich, aber nicht definitiv.«

Das könnte für Marge nicht genug sein.

Marge: »Schauen Sie, Peter, ich brauche eine definitive Aussage. Gibt es irgendeinen Weg, verbindlich zuzusagen, dass Sie alles bis Montagmorgen erledigt haben?«

Peter könnte an diesem Punkt versucht sein, die Regeln seiner Disziplin zu brechen. Er könnte in der Lage sein, schneller abzuschließen, wenn er keine Tests schreibt. Er könnte schneller abschließen, wenn er kein Refactoring macht. Er könnte schneller abschließen, wenn er nicht die komplette Regression-Suite durchlaufen lässt.

Das ist der Moment, wo für den Profi die Grenze erreicht ist. Zuerst einmal liegt Peter schlicht und einfach falsch mit seinen Annahmen. Er wird *nicht* schneller abschließen können, wenn er keine Tests schreibt. Er wird *nicht* schneller abschließen können, wenn er das Refactoring unterschlägt. Er wird nicht schneller abschließen können, wenn er die komplette Regression-Suite weglässt. Jahrelange Erfahrungen haben uns gelehrt, dass es uns nur verlangsamt, nicht den Regeln unserer Disziplin zu folgen.

Aber zweitens hat er als Profi die Verantwortung, bestimmte Standards zu bewahren. Sein Code muss getestet werden und braucht diese Tests. Sein Code muss sauber sein. Und er muss sicher sein, dass er im System nicht irgendetwas anderes kaputt gemacht hat.

Peter hat sich als Profi bereits verpflichtet, diese Standards einzuhalten. Alle anderen seiner Commitments sollten sich dem unterordnen. Also muss die ganze Argumentationsreihe abgebrochen werden:

- Peter: »Nein, Marge, es bleibt als todsicherer Zeitpunkt wirklich nur der Dienstag. Es tut mir leid, wenn das Ihren Zeitplan durcheinanderbringt, aber das ist eben einfach die Realität, mit der wir uns abfinden müssen.«
- Marge: »Verdammt. Ich hatte wirklich darauf gezählt, dass wir dies schneller in trockenen Tüchern haben. Sind Sie wirklich sicher?«
- Peter: »Ich bin sicher, dass es möglicherweise bis Dienstag dauern wird, ja.«
- Marge: »Okay, dann sollte ich mal mit Willy reden, um zu schauen, ob er seine Termine umstellen kann.«

In diesem Fall hat Marge Peters Antwort akzeptiert und macht sich daran, andere Optionen aufzutun. Aber was ist, wenn alle Optionen von Marge ausgereizt sind? Was ist, wenn Peter die letzte Hoffnung ist?

- Marge: »Hören Sie, Peter, ich weiß, dass dies eine ziemliche Zumutung ist, aber ich muss Sie wirklich dringend bitten, es möglich zu machen, alles bis Montagmorgen fertigzuhaben. Das ist wirklich kritisch. Gibt es da nichts, was Sie machen können?«

Nun beginnt Peter, darüber nachzudenken, ob er beträchtliche Überstunden machen und wahrscheinlich fast das ganze Wochenende arbeiten soll. Er muss hinsichtlich seiner Ausdauer und seiner Reserven sehr ehrlich mit sich selbst sein. Es geht einem *leicht* von den Lippen, dass man am Wochenende eine Menge geschafft bekommt, aber

es ist viel schwerer, tatsächlich so viel Energie aufzubringen, eine Arbeit von hoher Qualität abzuliefern.

Profis kennen ihre Grenzen. Sie wissen, wie viele Überstunden sie effektiv einbringen können, und kennen sich damit aus, um welchen Preis das geschieht.

In diesem Fall ist Peter sich ziemlich sicher, dass ein paar Überstunden in der Woche und etwas Zeit am Wochenende reichen werden.

Peter: »Okay, Marge, ich sage Ihnen Folgendes: Ich rufe zu Hause an und kläre mit der Familie, wie es mit Überstunden aussieht. Wenn das für die okay ist, kriege ich diese Aufgabe bis Montagmorgen fertig. Ich werde sogar Montagmorgen ins Büro kommen, damit wir sicher sein können, dass mit Willy alles glatt geht. Aber dann gehe ich nach Hause und komme erst Mittwoch wieder. Okay?«

Ein faires Angebot. Peter weiß, dass er die Modifikationen und Dokumentation hinkriegen wird, wenn er Überstunden macht. Er weiß aber auch, dass er ein paar Tage danach zu nichts zu gebrauchen sein wird.

3.3 Schlussfolgerung

Profis müssen nicht zu allem Ja sagen, was ihnen abverlangt wird. Allerdings sollten sie sich richtig ins Zeug legen, um kreative Wege zu finden, damit ein »Ja« möglich wird. Wenn Profis Ja sagen, formulieren sie das in verbindlicher Sprache, damit es keinen Zweifel an dem gibt, was sie versprechen.

4

Programmieren



In einem früheren Buch¹ habe ich sehr viel über die Struktur und Natur von *Clean Code* geschrieben. In diesem Kapitel wird es um den Akt des Programmierens gehen und den Kontext, in dem er stattfindet.

Als ich 18 war, konnte ich ganz gut tippen, aber ich musste immer auf die Tasten schauen. Blindschreiben war nicht drin. Also verbrachte ich mal einen Abend viele lange Stunden am Lochkartenlocher IBM 029 mit Tippen, ohne auf meine Finger zu schauen, während ich ein Programm eintippte, das ich auf mehrere Programmvordrucke geschrieben hatte. Ich untersuchte nach dem Tippen jede Karte und sortierte alle mit Tippfehlern aus.

¹ [Martin09]

Zuerst tippte ich eine ganze Menge verkehrt. Am Ende des Abends tippte ich alle fast perfekt, ohne hinzusehen. Ich erkannte an diesem langen Abend, dass es beim Blind-schreiben vor allem um *Vertrauen* geht. Meine Finger wussten, wo die Tasten waren. Ich musste mir nur das Selbstvertrauen zulegen, dass ich keine Fehler machte. Bei diesem Vertrauen war es u.a. hilfreich, dass ich es irgendwie *spüren* konnte, wenn ich mich vertippte. Am Ende des Abends wusste ich es beinahe sofort, wenn ich einen Fehler gemacht hatte, und warf die Karte einfach aus, ohne sie überhaupt anzusehen.

Es ist wirklich wichtig, ein Gespür für die eigenen Fehler zu entwickeln. Nicht nur beim Tippen, sondern auch sonst. Wenn man dieses Gespür für Fehler hat, schließt man die Feedback-Schleife sehr schnell und lernt umso schneller aus den eigenen Fehlern. Ich habe seit jenem Tag an der IBM 029 mehrere Fachrichtungen studiert und gemeistert. Dabei fand ich heraus, dass in jedem Fall der Schlüssel zur Meisterschaft im Selbstvertrauen und diesem Gespür für Fehler liegt.

Dieses Kapitel beschreibt meine persönlichen Regeln und Prinzipien beim Programmieren. Bei diesen Regeln und Prinzipien geht es nicht um meinen eigenen Code, sondern um mein Verhalten, meine Stimmung und meine innere Einstellung, wenn ich programmiere. Sie beschreiben meinen eigenen mentalen, moralischen und emotionalen Kontext beim Schreiben von Code. Dies sind die Wurzeln meines Selbstvertrauens und meines Gespürs für Fehler.

Sie werden wahrscheinlich nicht alledem zustimmen, was ich hier zu sagen habe. Immerhin sind das vor allem höchst persönliche Sachen. Womöglich widersprechen Sie auch massiv einigen meiner Einstellungen und Prinzipien. Das ist okay – sie sollen auch keine absoluten Wahrheiten für andere außer mir sein. Was sie aber sind: der ganz persönliche Ansatz eines Mannes, um ein professioneller Programmierer zu sein.

Vielleicht lernen Sie, mir den Kieselstein von der Hand zu schnappen, wenn Sie mein eigenes persönliches Programmiermilieu studieren und reflektieren.

4.1 Bereit sein

Programmieren ist eine intellektuell herausfordernde und anstrengende Aktivität. Es erfordert ein Maß an Konzentration und Fokus, den nur wenige andere Disziplinen einem abverlangen. Der Grund dafür ist, dass man beim Programmieren mit mehreren, einander widersprechenden Faktoren gleichzeitig jonglieren muss.

1. Erstens muss Ihr Code funktionieren. Sie müssen verstanden haben, welches Problem Sie lösen wollen, und auch, wie man dieses Problem lösen kann. Sie müssen darauf achten, dass der von Ihnen geschriebene Code eine gewissenhafte Repräsentation dieser Lösung ist. Sie müssen jedes Detail dieser Lösung im Blick haben und dabei gleichzeitig innerhalb der Sprache, Plattform, aktuellen Architektur und der ganzen Macken und Mucken des aktuellen Systems konsistent bleiben.

2. Ihr Code muss das Problem lösen, das Ihr Kunde Ihnen gestellt hat. Oft lösen die Anforderungen des Kunden dessen wahre Probleme nicht wirklich. Es obliegt Ihnen, das zu erkennen und mit dem Kunden zu verhandeln, damit gewährleistet bleibt, dass die wahren Bedürfnisse des Kunden erfüllt werden.
3. Ihr Code muss gut ins existierende System passen. Es sollte die Rigidität, Fragilität oder Opazität dieses Systems nicht steigern. Die Abhängigkeiten müssen gut verwaltet werden. Kurz gesagt: Ihr Code muss soliden Engineering-Prinzipien folgen².
4. Ihr Code muss von anderen Programmierern lesbar sein. Da reicht es nicht, einfach nur nette Kommentare einzustreuen. Es erfordert vielmehr, dass Sie den Code so formen und gestalten, dass damit Ihre Absicht deutlich wird. Das ist schwer umzusetzen. Tatsächlich ist es wohl das Schwierigste, was ein Programmierer meistern kann.

Es ist verflucht schwer, mit all diesen Aspekten zu jonglieren. Es ist physiologisch schwierig, für einen längeren Zeitraum die nötige Konzentration und den Fokus aufzubringen. Erschwert wird das noch durch die Probleme und Ablenkungen der Arbeit in einem Team, in einer Organisation und dem ganz alltäglichen Ablauf des normalen Lebens. Daraus lässt sich schließen, dass die Möglichkeiten der Ablenkung ziemlich groß sind.

Wenn Sie sich nicht konzentrieren und ausreichend fokussieren, wird Ihr Code fehlerhaft. Er wird Bugs haben. Er bekommt die falsche Struktur. Er wird unklar und kompliziert. Er wird die Alltagsprobleme des Kunden nicht beheben. Auf den Punkt gebracht: Ein solcher Code wird überarbeitet oder neu geschrieben werden müssen. Wenn man bei der Arbeit abgelenkt ist, produziert man Müll.

Wenn Sie müde oder abgelenkt sind, *sollten Sie nicht programmieren*. Das läuft dann über kurz oder lang darauf hinaus, dass Sie alles noch einmal machen müssen. Finden Sie stattdessen Wege, um die Ablenkungen zu eliminieren und Ihren Geist zu beruhigen.

4.1.1 Code um drei Uhr früh

Der schlimmste Code, den ich je geschrieben habe, war um 3 Uhr früh. Das war im Jahre 1988, und ich arbeitete in einem Start-up-Unternehmen in der Telekommunikationsbranche namens Clear Communications. Wir alle hatten Überstunden gemacht, um die Belastung gleichermaßen zu verteilen. Wir träumten natürlich alle davon, reich zu werden.

Eines sehr späten Abends – oder man kann auch sagen, ganz früh am Morgen – ließ ich meinen Code durch das Event-Dispatch-System eine Botschaft an sich selbst schicken, um ein Timing-Problem zu lösen (wir nannten das »Mails versenden«). Das war die falsche Lösung, aber um 3 Uhr früh wirkte das verdammt pfiffig. Tatsächlich fiel mir nach

² [Martin03]

18 Stunden durchgehendem Programmieren (ganz zu schweigen von der Woche vorher mit 60 oder 70 Stunden) einfach gar nichts anderes mehr ein.

Ich weiß noch, wie toll ich mich fand, dass ich so lange bei der Arbeit durchhalten konnte. Ich erinnere mich, wie engagiert ich mich fühlte. Ich weiß noch, dass ich dachte, für ernsthafte Profis gehört es eben dazu, um 3 Uhr früh noch zu arbeiten. Wie verkehrt ich damit lag!

Dieser Code war für uns ein ständig wiederkehrender Alptraum – immer wieder mussten wir uns damit auseinandersetzen. Er verfestigte eine fehlerhafte Designstruktur, die dann alle verwendeten, um die aber immer herumgearbeitet werden musste. Er sorgte für alle möglichen Timing-Fehler und merkwürdigen Feedback-Schleifen. Wir gerieten in Endlosschleifen bei den Mails, weil eine Nachricht dafür sorgte, dass die nächste verschickt wurde und dann wieder eine – ohne Ende. Wir hatten einfach keine Zeit mehr, diesen Kram neu zu schreiben (glaubten wir), aber scheinbar hatten wir immer Zeit, eine neuerliche Blase oder einen Patch zu schreiben, um die Hindernisse zu umgehen. Dieser Quellcode-Müll wuchs immer mehr und belastete diesen Code von 3 Uhr früh mit immer mehr Kram und Nebeneffekten. Jahre später wurde das im Team zu einem Running Gag. Immer, wenn ich müde oder frustriert war, meinten meine Kollegen: »Passt auf! Gleich schickt Bob wieder Mails an sich selbst!«

Die Moral von dieser Geschichte: Schreiben Sie keinen Code, wenn Sie müde sind. Bei Engagement und Professionalität geht es mehr um Disziplin als um Uhrzeit. Achten Sie darauf, dass Sie den Schlaf, die körperliche Gesundheit und Ihren Lebensstil ausbalancieren, damit Sie acht gute Stunden pro Tag einbringen können.

4.1.2 Sorgencode

Haben Sie sich schon mal nach einer heftigen Auseinandersetzung mit Ihrem Ehepartner oder Freunden an die Tastatur gesetzt und zu programmieren versucht? Ist Ihnen da auch aufgefallen, dass ein Hintergrundprozess in Ihrem Kopf ablief, der versuchte, diesen Kampf zu lösen oder zumindest noch einmal durchzuspielen? Manchmal spüren Sie den Stress dieses Hintergrundprozesses in Ihrem Brustkorb oder in der Magengrube. Er kann dafür sorgen, dass Sie unruhig und angespannt sind, als hätten Sie zu viel Kaffee oder Cola getrunken. Das lenkt ganz schön ab.

Wenn ich mir Sorgen wegen einer Auseinandersetzung mit meiner Frau mache oder wegen einer Krise beim Kunden oder wegen meines kranken Kindes, kann ich den Fokus nicht aufrechterhalten. Meine Konzentration schwankt und flattert. Ich merke, dass ich zwar auf den Bildschirm schaue und die Finger auf dem Keyboard liegen, aber ich mache nichts. Ich bin katatonisch. Wie paralysiert. Eine Million Meilen weg und arbeite durch das Problem im Hintergrund, anstatt tatsächlich das Programmierproblem vor meiner Nase zu lösen.

Manchmal zwingt ich mich dazu, über den Code nachzudenken. Vielleicht bringe ich es auch zustande, eine oder zwei Zeilen zu schreiben. Ich reiße mich so zusammen, dass ich vielleicht sogar einen oder zwei Tests durchführe. Aber ich halte das nicht durch. Ich merke, wie ich unausweichlich in eine Art erstarrter Gefühllosigkeit abrutsche und offenen Auges nichts mehr sehe, während ich im Hintergrund innerlich ständig um mein Anliegen kreise.

Ich habe daraus gelernt, dass man in dieser Zeit die Finger vom Code lassen sollte. Alles, was ich programmiere, wird dann einfach nur Müll. Also muss ich diese Sorge lösen anstatt zu programmieren.

Natürlich gibt es viele Sorgen und Nöte, die man nicht in ein oder zwei Stunden lösen kann. Außerdem werden unsere Arbeitgeber es wahrscheinlich nicht lange tolerieren, dass wir arbeitsunfähig sind, weil wir unsere privaten Probleme zu lösen versuchen. Der Trick besteht darin zu lernen, wie man den Hintergrundprozess abschaltet oder zumindest dessen Priorität reduziert, damit er einen nicht dauerhaft ablenkt.

Ich mache das, indem ich meine Zeit aufteile. Anstatt mich zum Programmieren zu zwingen, während dieser Hintergrundprozess an mir nagt, stelle ich mich darauf ein, einen bestimmten Zeitraum (z.B. eine Stunde) damit zu verbringen, an dem Problem zu arbeiten, das mich bedrückt. Wenn mein Kind krank ist, rufe ich zu Hause an und frage nach. Wenn ich einen Streit mit meiner Frau hatte, rufe ich sie an und spreche die Sache durch. Wenn mich finanzielle Probleme drücken, richte ich mir Zeit ein, um darüber nachzudenken, wie ich die Geldangelegenheiten regeln kann. Ich weiß, dass ich die Probleme wahrscheinlich nicht in dieser Stunde lösen werde, aber sehr wahrscheinlich kann ich den Druck und die Unruhe reduzieren und den Hintergrundprozess beruhigen.

Idealerweise sollte die Zeit, in der man mit persönlichen Themen beschäftigt ist, in der privaten Zeit liegen. Es wäre eine Schande, eine Stunde im Büro auf diese Weise zuzubringen. Professionelle Entwickler kontingentieren ihre private Zeit, um zu gewährleisten, dass die im Büro verbrachte Zeit so produktiv wie möglich ist. Das bedeutet, Sie sollten sich darum kümmern, außerhalb der Arbeit Zeit dafür einzuräumen, um Ihre Sorgen und Nöte anzugehen, damit Sie sie nicht mit ins Büro bringen müssen.

Wenn Sie andererseits im Büro sitzen und merken, dass im Hintergrund Sorgen und Unruhe an Ihrer Produktivität nagen, ist es besser, eine Stunde dafür einzuräumen, sie zu beruhigen, als sich mit roher Gewalt zum Programmieren zu zwingen und diesen Code dann später einfach wegwerfen (oder schlimmer noch: damit leben) zu müssen.

4.2 Der Flow-Zustand

Über den hyperproduktiven Zustand, den man »Flow« nennt, ist schon viel geschrieben worden. Manche Programmierer nennen ihn »die Zone«. Egal wie man ihn nennt, Sie sind wahrscheinlich schon vertraut damit. Damit ist der höchst konzentrierte Bewusst-

seinszustand mit Tunnelblick gemeint, in den Programmierer kommen können, wenn sie Code schreiben. In diesem Zustand fühlen sie sich *produktiv*. In diesem Zustand fühlen sie sich *unfehlbar*. Also ist es für sie erstrebenswert, in diesen Zustand zu gelangen, und oft messen sie ihren Selbstwert daran, wie lange sie darin verbringen können.

Hier kommt ein kleiner Tipp von jemandem, der dort war und zurückkam: *Vermeiden Sie die Zone*. Dieser Bewusstseinszustand ist nicht wirklich hyperproduktiv und ganz gewiss nicht unfehlbar. In Wirklichkeit handelt es sich um einen leicht meditativen Status, in dem bestimmte rationale Fähigkeiten vermindert sind zugunsten eines gewissen Gefühls der Geschwindigkeit.

Lassen Sie mich das deutlich formulieren: Ihnen wird es in der Zone *tatsächlich* gelingen, mehr Code zu schreiben. Wenn Sie TDD praktizieren, werden Sie schneller durch die Red-/Green-/Refactor-Schleife kommen. Und Sie werden *definitiv* eine gewisse Euphorie spüren oder ein Kampf- und Siegesgefühl bekommen. Das Problem ist, dass Sie den größeren Zusammenhang etwas aus dem Blick verlieren, während Sie im Flow sind. Also werden Sie wahrscheinlich Entscheidungen treffen, zu denen Sie später zurückgehen müssen, um sie ungeschehen zu machen. Code, der im Flow geschrieben wurde, wird schneller produziert, aber später müssen Sie sich den dann öfter wieder vornehmen.

Wenn ich heutzutage merke, dass ich in die Zone rutsche, gehe ich ein paar Minuten weg. Ich lüfte meinen Kopf durch, indem ich ein paar E-Mails beantworte und Tweets lese. Wenn schon bald Mittag ist, mache ich meine Essenspause. Wenn ich in einem Team arbeite, suche ich mir einen Partner fürs Pair Programming.

Einer der großen Vorteile des Pair Programmings ist, dass es für ein Paar praktisch unmöglich ist, gemeinsam in die Zone zu geraten. Die Zone ist ein unkommunikativer Zustand, während das Pair Programming intensive und konstante Kommunikation braucht. Tatsächlich gehört es zu den Beschwerden übers Pairing, dass damit der Eintritt in die Zone blockiert wird. Aber das ist gut! Sie sollten sich *nicht* in der Zone befinden.

Tja, *ganz* stimmt das nicht. Es gibt Zeiten, wo die Zone genau der richtige Ort ist. Nämlich wenn Sie üben. Aber darüber sprechen wir in einem anderen Kapitel.

4.2.1 Musik

Bei Teradyne hatte ich Ende der 1970er-Jahre mein eigenes Büro. Ich war Systemadministrator unserer PDP 11/60, und so war ich einer der wenigen Programmierer, denen ein privates Terminal erlaubt war. Dieses Terminal war ein VT100, das eine Baudrate von 9600 Baud hatte und mit der PDP 11 über ein 25 Meter langes RS232-Kabel verbunden war. Diese Strippe hatte ich über die Deckenverkleidung meines Büros bis zum Computerraum gezogen.

In meinem Büro stand eine Musikanlage. Das war ein alter Plattenspieler mit Verstärker und Boxen. Ich hatte eine ansehnliche Sammlung mit Platten von Led Zeppelin, Pink Floyd und ... Nun, Sie wissen schon.

Meist zog ich die Regler ziemlich hoch und schrieb dabei meinen Code. Ich dachte, das würde meiner Konzentration helfen. Aber da lag ich falsch.

Eines Tages nahm ich mir wieder ein Modul vor, das ich bearbeitet hatte, während im Hintergrund der Anfang von *The Wall* gedudelt hatte. Die Kommentare in diesem Code enthielten Textzeilen aus diesem Stück und redaktionelle Anmerkungen über Sturzbomben und schreiende Babys.

Da durchfuhr es mich: Der Leser dieses Codes erfuhr mehr über die Musiksammlung des Autors (ich), als dass er über das Problem herausbekam, das der Code zu lösen versuchte.

Ich erkannte, dass ich einfach nicht gut programmiere, während ich Musik zuhöre. Die Musik unterstützt mich nicht beim Fokussieren. Vielmehr scheint der Vorgang, Musik zu hören, an einer vitalen Ressource zu nagen, die mein Geist benötigt, um sauberen und wohlgeformten Code zu schreiben.

Vielleicht geht Ihnen das ganz anders. Vielleicht *hilft* Ihnen Musik dabei, Code zu schreiben. Ich kenne viele, die beim Programmieren Kopfhörer tragen. Ich akzeptiere, dass die Musik sie vielleicht unterstützt, hege aber auch den Verdacht, dass Musik ihnen eher dabei hilft, in die Zone zu kommen.

4.2.2 Unterbrechungen

Stellen Sie sich einmal bildlich vor, wie Sie an Ihrer Workstation sitzen und arbeiten. Wie reagieren Sie, wenn jemand Ihnen eine Frage stellt? Blaffen Sie ihn an? Starren Sie ihn mit funkelnden Augen an? Kann man Ihrer Körpersprache entnehmen, dass der Fragesteller weggehen soll, weil Sie beschäftigt sind? Kurz gesagt, sind Sie unhöflich?

Oder unterbrechen Sie, was Sie gerade machen, und helfen höflich jemandem, der irgendwie festsitzt? Behandeln Sie ihn so, wie Sie selbst gerne behandelt werden möchten, wenn Sie festhängen?

Die unhöfliche Reaktion stammt oft aus der Zone. Vielleicht haben Sie etwas dagegen, aus der Zone gezogen zu werden, oder es stört Sie, wenn Ihnen jemand dabei querschießt, wenn Sie gerade versuchen, in die Zone zu kommen. Wie dem auch sei: Die Unhöflichkeit rührt oft aus Ihrer Beziehung zur Zone her.

Manchmal ist allerdings nicht die Zone schuld, sondern Sie versuchen bloß inständig, etwas Kompliziertes zu verstehen, auf das man sich sehr konzentrieren muss. Dafür gibt es mehrere Lösungen.

Pairing kann sehr hilfreich sein, um mit Unterbrechungen und Störungen umgehen zu lernen. Ihr Partner kann den Kontext des aktuellen Problems weiter aufrechterhalten, während Sie einen Anruf abwickeln oder sich mit der Frage eines Kollegen beschäftigen. Wenden Sie sich dann wieder Ihrem Partner zu, hilft er Ihnen schnell dabei, den mentalen Kontext zu rekonstruieren, in dem Sie sich vor der Unterbrechung befanden.

TDD ist eine weitere große Hilfe. Wenn Sie an einem fehlgeschlagenen Test arbeiten, bewahrt der Test den Kontext, wo Sie sich gerade befinden. Sie können nach der Unterbrechung dorthin zurückkehren und damit weitermachen, dass dieser misslungene Test bestanden wird.

Letzten Endes wird es natürlich Unterbrechungen geben, die Sie ablenken und Zeit kosten. Wenn das geschieht, sollten Sie sich das in Erinnerung rufen, wenn Sie das nächste Mal derjenige sind, der einen anderen unterbrechen muss. Also gehört zur professionellen Einstellung die höfliche Bereitschaft, hilfsbereit zu sein.

4.3 Schreibblockaden

Manchmal kriegt man keine Codezeile herausgedrückt. Das ist mir selbst so gegangen, und ich habe auch andere gesehen, denen das widerfahren ist: Man sitzt an der Workstation herum, und nichts passiert.

Oft findet man andere Dinge, die erledigt werden wollen. Man studiert seine E-Mails. Man liest Tweets. Man blättert Bücher oder Zeitpläne oder Dokumente durch. Beruft Meetings ein. Trifft sich zu Gesprächen mit anderen. Man stellt einfach *alles Mögliche* an, um sich nicht an die Workstation setzen und merken zu müssen, dass der Code einfach nicht erscheinen will.

Was verursacht solche Blockaden? Wir haben über viele dieser Faktoren bereits gesprochen. Für mich ist Schlaf ein weiterer wichtiger Faktor. Wenn ich nicht genug schlafe, kann ich schlicht und einfach nicht programmieren. Andere Einflüsse sind Sorgen, Angst und Depressionen.

Komischerweise gibt es eine sehr einfache Lösung, die praktisch immer funktioniert. Sie ist ganz einfach und kann dafür sorgen, dass Sie wieder viel Schwung bekommen, um eine Menge Code fertigzukriegen.

Die Lösung lautet: Suchen Sie sich einen (guten) Pairing-Partner.

Es ist regelrecht unheimlich, wie gut das funktioniert. Sobald Sie sich neben jemanden setzen, zerfließen die Probleme, die Sie blockiert haben. Irgendwie findet da eine *physiologische* Veränderung statt, wenn Sie mit jemandem zusammenarbeiten. Ich weiß nicht, was das genau ist, aber ich kann es definitiv fühlen. Es ist irgendeine chemische Veränderung in meinem Gehirn oder Körper, durch die ich die Blockade durchbreche und wieder in Fluss komme.

Das ist keine perfekte Lösung. Manchmal dauert diese Veränderung nur ein oder zwei Stunden an, und dann kommt eine derart heftige Erschöpfung, dass ich mich von meinem Pairing-Partner zurückziehen und sozusagen erst mal in die Höhle muss, um mich wieder zu erholen. Manchmal ist es beim Pairing sogar so, dass ich kaum mehr machen kann als allem zuzustimmen, was mein Partner macht. Doch für mich ist die typische Reaktion aufs Pairing die Wiederherstellung meines Schwungs.

4.3.1 Kreativer Input

Es gibt noch weitere Dinge, die ich gegen Blockaden unternehme. Ich habe schon vor längerer Zeit gelernt, dass kreativer Output von kreativem Input abhängt.

Ich lese sehr viel und auch alles Mögliche. Ich lese Material über Software, Politik, Biologie, Astronomie, Physik, Chemie, Mathematik und vieles mehr. Doch was bei mir am besten funktioniert, wo mein kreativer Output den größten Kick kriegt, ist Science-Fiction.

Für Sie ist das eventuell etwas anderes: vielleicht ein guter Krimi oder Thriller, vielleicht Poesie oder sogar ein romantischer Roman. Ich denke, dass es hier in Wirklichkeit darum geht, dass Kreativität die Kreativität schafft. Auch wohnt dieser Lösung ein eskapistisches Element inne. Die Stunden, die ich ohne meine üblichen Probleme verbringe, während ich aktiv von herausfordernden und kreativen Ideen stimuliert werde, führen einen in den beinahe unwiderstehlichen Druck, selbst etwas Kreatives zu schaffen.

Nicht alle Formen von kreativem Input funktionieren bei mir. Wenn ich fernsehe, macht mich das normalerweise nicht kreativ. Ins Kino gehen ist besser, aber nur ein bisschen. Wenn ich Musik höre, hilft mir das beim Erschaffen von Code nicht weiter, aber bei Präsentationen, Vorträgen und Videos. Von allen Formen des kreativen Inputs funktioniert bei mir nichts besser als die gute alte *Weltraumoper*.

4.4 Debugging

Eine der schlechtesten Debugging-Sessions meiner Karriere hatte ich 1972. Die Terminals, die mit dem Abrechnungssystem der Gewerkschaft verbunden waren, hängten sich täglich ein- oder zweimal auf. Man konnte diesen Fehler nicht reproduzieren. Der Fehler tauchte nicht bevorzugt bei einem bestimmten Terminal oder einer Anwendung auf. Es war ganz egal, was der User vorher gerade gemacht hatte. In der einen Minute funktionierte das Terminal prima, und im nächsten Augenblick hängte es sich unweigerlich auf.

Es dauerte Wochen, dieses Problem zu diagnostizieren. In der Zwischenzeit gerieten die Gewerkschaftler immer mehr in Wallung. Jedes Mal, wenn sich ein Terminal aufhängte, musste die Person, die daran arbeitete, ihre Arbeit unterbrechen und warten,

bis alle anderen User koordiniert ihre Aufgaben abgeschlossen hatten. Dann wurden wir angerufen, und wir kümmerten uns um den Neustart. Es war ein Alptraum.

Wir verbrachten die ersten paar Wochen nur damit, Daten zu sammeln, indem wir die Leute interviewten, bei denen sich die Terminals aufhängten. Wir fragten sie, was sie just in dem Moment des Einfrierens und in der Zeit davor gemacht hatten. Wir baten die anderen User um Auskunft, ob ihnen etwas an ihren Terminals aufgefallen ist, als eines der Terminals eingefroren war. Diese Interviews führten wir alle telefonisch durch, weil sich die Terminals in der Stadtmitte von Chicago befanden, während wir etwa 50 km nördlich zwischen Maisfeldern arbeiteten.

Wir hatten keine Logs, keine Counter, keine Debugger. Unser einziger Zugang zu den Innereien des Systems waren Ein-/Ausschalter sowie Umschalter auf dem Frontpanel. Wir konnten den Computer anhalten und uns dann im Speicher umschauchen – und zwar buchstäblich Wort für Wort. Aber das durften wir längstens fünf Minuten machen, weil die Gewerkschaftler ihr System wieder hochgefahren brauchten.

Wir brauchten einige Tage, bis wir einen einfachen Inspektor in Echtzeit fertig hatten, der vom ASR-33 Teletype bedient werden konnte, das als unsere Konsole diente. Damit konnten wir uns im Speicher etwas umschauchen und herumstochern, während das System lief. Wir fügten Protokollnachrichten ein, die in kritischen Momenten auf der Teletype ausgegeben wurden. Wir erstellten speicherinterne Counter, die Events zählten und die Zustandshistorie speicherten, damit wir den Inspektor untersuchen konnten. Und natürlich musste das alles von Grund auf in Assembler geschrieben und abends oder nachts getestet werden, wenn das System nicht verwendet wurde.

Die Terminals waren interrupt-gesteuert. Die an die Terminals gesendeten Zeichen wurden in zirkulären Puffern abgelegt. Jedes Mal, wenn ein serieller Port ein Zeichen schickte, feuerte ein Interrupt, und in diesem zirkulären Puffer wurde das nächste Zeichen für den Versand vorbereitet.

Wir fanden schließlich heraus, dass ein Terminal dann einfro, wenn die drei Variablen, die den zirkulären Puffer verwalteten, nicht synchron waren. Wir hatten keine Ahnung, warum das passierte, aber zumindest war es ein erster Hinweis. Irgendwo in den 5.000 Zeilen des Überwachungscode befand sich ein Bug, der einen dieser Zeiger falsch behandelte.

Diese neue Erkenntnis erlaubte uns auch, die eingefrorenen Terminals manuell wieder zu starten! Wir konnten anhand des Inspektors Standardwerte in diese drei Variablen stecken, und die Terminals starteten wie durch Zauberhand erneut. Schließlich schrieben wir einen kleinen Hack, der immer alle Counter durchging und schaute, ob sie irgendwie verkehrt ausgerichtet waren. Dann wurden sie bei Bedarf wieder repariert. Zuerst riefen wir diesen Hack auf, indem wir auf dem Frontpanel einen speziellen User-Schalter für Interrupts umlegten, sobald einer der Gewerkschaftler telefonisch wieder ein Einfrieren meldete. Später ließen wir dieses Reparatur-Utility einmal pro Sekunde laufen.

Etwa einen Monat später hatte sich der Ärger mit den eingefrorenen Terminals erledigt – zumindest was die Gewerkschaftler anging. Gelegentlich pausierte eines ihrer Terminals noch mal für etwa eine halbe Sekunde, aber bei einer Basisrate von 30 Zeichen pro Sekunde schien das niemandem aufzufallen.

Aber warum kamen die Zähler immer wieder aus dem Takt? Ich war 19 und entschlossen, das herauszufinden.

Den Überwachungscode hatte Richard geschrieben, der anschließend zum College gewechselt war. Keiner von uns Verbliebenen war mit diesem Code vertraut, denn Richard hatte ihn eifersüchtig gehütet. Dieser Code war *seiner*, und wir durften darüber nichts wissen. Aber nun war Richard weg, also holte ich das mehrere Zentimeter dicke Listing hervor und begann, alles Seite für Seite durchzusehen.

Bei den zirkulären Queues in diesem System handelte es sich einfach um FIFO-Datenstrukturen, d.h. Queues. Die Anwendungsprogramme schoben Zeichen am einen Ende der Queues hinein, bis die Queue voll war. Die Interrupt-Heads poppten die Zeichen vom anderen Ende der Queues, wenn der Drucker für sie bereit war. Wenn die Queue leer war, stoppte der Drucker. Unser Bug sorgte dafür, dass die Anwendungen meinten, die Queue sei voll. Gleichzeitig gingen die Interrupt-Heads davon aus, dass die Queue leer sei.

Interrupt-Heads laufen in einem anderen »Thread« als der restliche Code. Also müssen Counter und Variablen, die sowohl von Interrupt-Heads als auch anderem Code manipuliert werden, vor gleichzeitigen Updates geschützt werden. In unserem Fall bedeutete das, die Interrupts in der jeweiligen Code-Umgebung abzuschalten, die diese drei Variablen manipulierten. Als ich mich an den Code setzte, wusste ich, dass ich nach irgendeiner Stelle im Code suchen musste, die sich mit den Variablen befasste, aber nicht vorher die Interrupts deaktiviert hatte.

Heutzutage haben wir natürlich ein ganzes Bündel leistungsfähiger Tools zur Hand, um jede Stelle zu finden, wo der Code diese Variablen anfasst. Innerhalb weniger Sekunden wüssten wir jede Codezeile, wo das passiert. Innerhalb von Minuten wäre klar, an welcher Stelle die Interrupts nicht deaktiviert wurden. Aber wir schrieben das Jahr 1972, und solche Tools standen mir einfach nicht zur Verfügung. Ich hatte bloß meine Adlraugen.

Ich grübelte über jeder Seite dieses Codes und suchte nach den Variablen. Bedauerlicherweise kamen die Variablen *überall* zum Einsatz. Beinahe auf jeder Seite wurden sie auf die eine oder andere Weise bearbeitet. Viele dieser Referenzen deaktivierten die Interrupts nicht, weil es Nur-lesen-Referenzen waren und somit harmlos. Das Problem war, dass es in diesem speziellen Assembler keine praktische Art und Weise gab herauszufinden, ob ein Verweis *read only* war, ohne der Logik des Codes zu folgen. Jedes Mal, wenn eine Variable gelesen wurde, konnte sie später auch mal aktualisiert und gespeichert werden. Wenn dann die Interrupts noch aktiviert waren, konnten die Variablen korrumpiert sein.

Ich musste tagelang intensiv forschen, aber am Ende fand ich es heraus: Tief im Code vergraben gab es eine Stelle, bei der eine der drei Variablen aktualisiert wurde, während die Interrupts aktiviert blieben.

Ich rechnete das mal durch: Die Schwachstelle dauerte etwa zwei Mikrosekunden. Es gab ein Dutzend Terminals, die alle mit 30 cps liefen. Also geschah etwa alle 3 ms ein Interrupt. Wenn man die Größe des Programms und die Taktrate der CPU berücksichtigt, war durch diese Schwachstelle etwa ein bis zwei Mal täglich ein Aufhängen zu erwarten. Bingo!

Ich behob das Problem natürlich, fand aber nie den Mut, den automatischen Hack abzuschalten, mit dem die Counter inspiziert und gefixt wurden. Bis zum heutigen Tag bin ich nicht überzeugt, ob es nicht noch ein anderes Schlupfloch gab.

4.4.1 Zeit zum Debuggen

Aus irgendwelchen Gründen betrachten Software-Entwickler die Debugging-Zeit nicht als Coding-Zeit. Sie stellen sich die Zeit fürs Debuggen als dringliches Bedürfnis vor – etwas, was einfach erledigt werden *muss*. Aber Debugging-Zeit ist fürs Business genauso kostenträchtig wie Coding-Zeit, und darum ist alles gut, was wir tun können, um diese Zeit zu vermeiden oder zu verkürzen.

Heutzutage muss ich viel weniger Zeit fürs Debugging aufwenden als noch vor zehn Jahren. Ich habe den Unterschied nicht gemessen, aber gefühlt geht es hier um den Faktor 10. Diese wirklich radikale Reduktion bei der Debugging-Zeit ist mir dadurch gelungen, dass ich die Praxis der testgetriebenen Entwicklung (Test Driven Development, TDD) befolge, die wir in einem anderen Kapitel besprechen werden.

Egal ob Sie sich TDD oder eine andere Disziplin mit ähnlicher Wirksamkeit aneignen³, obliegt es Ihnen als Profi, Ihre Debugging-Zeit so nahe wie möglich gegen null zu bringen. Natürlich ist null ein asymptotisches Ziel, aber nichtsdestotrotz das Ziel.

Ärzte schneiden ihre Patienten auch nicht gerne wieder auf, um zu reparieren, wo sie gepatzt haben. Rechtsanwälte greifen auch nicht gerne wieder Fälle auf, die sie vermurkst haben. Einen Arzt oder Anwalt, der das zu oft macht, betrachtet man nicht als Profi. Entsprechend unprofessionell verhält sich ein Software-Entwickler, der viele Bugs produziert.

4.5 Die eigene Energie einteilen

Software-Entwicklung ist ein Marathon, kein Sprint. Sie können das Rennen nicht gewinnen, wenn Sie von Anfang an so schnell wie möglich laufen. Sie gewinnen, indem Sie

3 Ich kenne keine Disziplin, die so effektiv wie TDD ist, aber vielleicht fällt Ihnen ja eine ein.

Ihre Ressourcen schonen und sich das Tempo einteilen. Ein Marathonläufer kümmert sich um seinen Körper *vor und während* des Rennens. Mit der gleichen Sorgfalt teilen sich professionelle Programmierer ihre Energie und Kreativität ein.

4.5.1 Wann man den Stift weglegen muss

Sie können einfach nicht Feierabend machen, bevor Sie dieses Problem nicht gelöst haben? O doch, das können Sie, und Sie sollten es wahrscheinlich auch! Kreativität und Intelligenz sind flüchtige Geisteszustände, die sich aus dem Staube machen, wenn Sie müde sind. Wenn Sie dann Ihr nicht funktionierendes Hirn stundenlang bis spät in die Nacht vorantreiben, um ein Problem zu lösen, werden Sie selbst immer träger und müder und reduzieren die Chance, dass Dusche oder Heimfahrt im Auto Ihnen bei der Lösung des Problems helfen.

Wenn Sie feststecken oder müde sind, sollten Sie sich für eine Weile von der Arbeit lösen. Geben Sie Ihrem kreativen Unterbewusstsein die Chance, an diesem Problem zu arbeiten. Sie schaffen mehr in weniger Zeit und brauchen weniger Mühe aufzuwenden, wenn Sie mit Ihren Ressourcen haushalten. Teilen Sie das Tempo für sich, aber auch für Ihr Team ein. Werden Sie sich über Ihre eigenen Muster für Kreativität und Brillanz klar und nutzen Sie diese aus anstatt dagegen zu arbeiten.

4.5.2 Die Heimfahrt

Ein Ort, wo ich eine ganze Reihe von Problemen lösen konnte, ist in meinem Wagen auf dem Heimweg von der Arbeit. Fürs Autofahren braucht man viele nichtkreative mentale Ressourcen. Sie müssen Ihre Augen, Hände und Bereiche Ihres Geistes auf die Aufgabe konzentrieren, also müssen Sie sich von den Arbeitsproblemen abkoppeln. Irgendetwas an dieser Art der *Abkopplung* erlaubt es Ihrem Geist, sich auf eine andere und kreativere Weise auf die Suche nach Lösungen zu machen.

4.5.3 Die Dusche

Unter der Dusche habe ich bereits unglaublich viele Probleme lösen können. Vielleicht erweckt mich dieser morgendliche Wasserschauer derart, dass ich all die Lösungen prüfen kann, die meinem Hirn während meines Schlafes eingefallen sind.

Wenn Sie an einem Problem arbeiten, stehen Sie manchmal mit der Nase so dicht davor, dass Sie nicht mehr alle Optionen überschauen können. Sie verpassen elegante Lösungen, weil der kreative Teil Ihres Gehirns von der Intensität Ihrer Konzentration unterdrückt wird. Manchmal ist es für die Problemlösung am besten, wenn Sie nach Hause gehen, zu Abend essen, in die Glotze schauen, ins Bett gehen und dann am nächsten Morgen nach dem Aufstehen eine schöne Dusche nehmen.

4.6 In Verzug sein

Sie *werden* sich verspäten. Das geschieht auch den Besten und Engagiertesten von uns. Manchmal haben wir uns komplett verkalkuliert und kommen einfach in Verzug.

Der Trick, um mit einer solchen Verzögerung umzugehen, ist frühzeitiges Erkennen und Transparenz. Das Worst-Case-Szenario tritt ein, wenn Sie fortlaufend bis zur letzten Minute allen erzählen, dass Sie pünktlich fertigwerden ... und dann alle hängen lassen. Machen Sie das bloß nicht! Messen Sie vielmehr *regelmäßig* Ihren Fortschritt anhand Ihres Ziels und erstellen Sie drei⁴ faktenbasierte Enddaten: optimal, Normalfall und schlimmstenfalls. Seien Sie bei all diesen drei Zeitpunkten so ehrlich wie möglich. *Bauen Sie keine hoffnungsvollen Zeitschätzungen ein!* Präsentieren Sie Ihrem Team und Ihren Stakeholdern alle drei Zahlen. Aktualisieren Sie diese Zahlen täglich.

4.6.1 Hoffnung

Was ist, wenn die Zahlen nahelegen, dass Sie eine Deadline verpassen *könnten*? Nehmen wir beispielsweise an, es fände in zehn Tagen eine Messe statt, und Sie müssten dort Ihr Produkt zeigen können. Aber nehmen wir ebenfalls an, dass die drei Zahlen für das Feature, an dem Sie arbeiten, in der Kalkulation 8/12/20 lauten.

Geben Sie sich nicht der Hoffnung hin, alles auch in zehn Tagen zu schaffen! Hoffnung ist der Killer des Projekts. Hoffnung zerstört Zeitpläne und ruiniert Reputationen. Mit der Hoffnung geraten Sie tief in Ärger. Wenn die Messe in zehn Tagen ist und das Soll-Datum 12 besagt, werden Sie es *nicht* schaffen! Achten Sie darauf, dass Team und Stakeholder die Situation begreifen, und lassen Sie nicht locker, bis es einen Notfallplan gibt. Lassen Sie nicht zu, dass jemand sich in Hoffnung ergeht.

4.6.2 Sich beeilen

Was ist, wenn Ihr Manager Sie bittet, Platz zu nehmen, und Sie auffordert, die Deadline einzuhalten zu versuchen? Was ist, wenn Ihr Manager darauf besteht, dass Sie »alle nötigen Maßnahmen ergreifen«? *Bleiben Sie bei Ihrer Kalkulation!* Ihre ursprüngliche Kalkulation ist genauer als jede Änderung, die Sie vornehmen, während der Chef Ihnen Druck macht. Sagen Sie ihm, dass Sie die Optionen bereits abgewogen haben (denn das stimmt) und dass die einzige Möglichkeit, den Zeitplan einzuhalten, darin besteht, den Umfang zu reduzieren. *Lassen Sie sich nicht zur Eile verführen.*

Wehe dem armen Entwickler, der unter Druck einknickt und einwilligt zu versuchen, die Deadline zu halten. Dieser Entwickler wird anfangen, Abkürzungen zu nehmen und Überstunden einzulegen in der vergeblichen Hoffnung, ein Wunder zu vollbringen. Das ist eine hervorragende Rezeptur für Katastrophen, weil es Sie, Ihr Team und Ihre Stake-

⁴ Mehr darüber im Kapitel über die Kalkulation.

Der in falscher Hoffnung wiegt. So dürfen sich alle darum drücken, sich der Problematik zu stellen, und die nötigen harten Entscheidungen werden aufgeschoben.

Sich beeilen ist nicht der Weg. Sie können sich nicht dazu bringen, schneller zu programmieren. Sie können sich nicht dazu bringen, schneller die Probleme zu lösen. Wenn Sie das versuchen, werden Sie nur langsamer und richten ein Chaos an, das auch alle anderen ausbremst.

Also müssen Sie auf Ihren Chef, Ihr Team und Ihre Stakeholder reagieren, indem Sie sie der Hoffnung berauben.

4.6.3 Überstunden

Nun sagt Ihr Chef: »Wie wäre es, wenn Sie täglich zwei Stunden extra arbeiten? Kommen Sie doch am Samstag auch noch mal rein. Na los, es muss doch irgendeinen Weg geben, um genug Stunden zusammenzukriegen, damit das Feature rechtzeitig fertig wird.«

Manchmal funktionieren Überstunden, und manchmal sind sie auch notwendig. Manchmal können Sie ein ansonsten unmögliches Datum schaffen, indem Sie ein paar Arbeitstage mit zehn Stunden einlegen und einen Samstag oder zwei opfern. Aber das ist sehr riskant. Sie werden wahrscheinlich nicht 20 % mehr Leistung bringen, indem Sie 20 % länger arbeiten. Überdies geht das mit Überstunden fast mit *Sicherheit* nach hinten los, wenn sie mehr als zwei oder drei Wochen andauern.

Also sollten Sie den Überstunden nicht zustimmen, außer (1) Sie können es sich persönlich leisten, (2) es betrifft nur einen kurzen Zeitraum, also zwei Wochen oder weniger, und (3) *Ihr Chef hat einen Notfallplan*, falls der Überstundeneinsatz nicht ausreicht.

Dieses letzte Kriterium ist das ausschlaggebende, an dem die Einigung scheitern kann. Wenn der Chef Ihnen gegenüber nicht in der Lage ist zu benennen, was er machen wird, falls die Überstunden nicht den gewünschten Erfolg bringen, sollten Sie den Überstunden nicht zustimmen.

4.6.4 Unlautere Ablieferung

Von allem unprofessionellen Verhalten, dem ein Programmierer frönen kann, ist vielleicht das Schlimmste zu behaupten, man sei fertig, wenn das gar nicht stimmt. Manchmal ist es einfach eine unverhohlene Lüge, und das ist schlimm genug. Doch der weitaus heimtückischere Fall ist, wenn es uns gelingt, eine neue »Definition of Done« auszuhecken. Wir überzeugen uns selbst, wir seien *fertig genug*, und nehmen die nächste Aufgabe in Angriff. Wir reden uns ein, dass wir uns um eventuell noch übrig gebliebene Arbeit später kümmern können, wenn mehr Zeit ist.

Diese Praxis ist ansteckend. Wenn ein Programmierer damit anfängt, werden andere das merken und es übernehmen. Einer dehnt die »Definition of Done« noch weiter aus, und alle anderen akzeptieren dann diese neue Definition. Ich habe erlebt, wie das ganz schreckliche Ausmaße annehmen kann. Einer meiner Kunden definierte »erledigt und fertig« als »eingescheckt«. Der Code brauchte noch nicht mal kompiliert zu sein! Es ist sehr einfach, »fertig« zu sein, wenn nichts funktionieren muss!

Wenn ein Team in diese Falle tappt, hören die Manager, dass alles ganz prima läuft und gut wird. Alle Statusberichte zeigen, dass alle im Zeitplan liegen. Das gleicht blinden Männern, die auf den Bahngleisen ein Picknick veranstalten: Niemand sieht den Zug der unerledigten Arbeit herandonnern, bis es zu spät ist.

4.6.5 Definieren Sie »fertig und erledigt«

Sie vermeiden das Problem der falschen Lieferung, indem Sie eine unabhängige »Definition of Done« erstellen. Das gelingt Ihnen am besten, wenn Ihre Business-Analysten und Tester automatisierte Akzeptanztests⁵ erstellen, die bestanden werden müssen, bevor Sie sagen dürfen, dass Sie fertig sind. Diese Tests sollten in einer Testsprache wie FitNesse, Selenium, RobotFX, Cucumber usw. geschrieben sein. Diese Tests sollten für Stakeholder und Business-Leute verständlich sein und regelmäßig durchgeführt werden.

4.7 Hilfe

Programmieren ist *schwer*. Je jünger Sie sind, desto weniger werden Sie das glauben. Immerhin geht's ja bloß um einen Haufen `if`- und `while`-Anweisungen. Aber wenn Sie im Laufe der Zeit Ihre Erfahrungen sammeln, beginnen Sie zu erkennen, dass die Art, wie Sie diese `if`- und `while`-Anweisungen kombinieren, von zentraler Bedeutung ist. Sie können die nicht einfach zusammenkloppen und das Beste hoffen. Sie müssen vielmehr das System in kleine, verständliche Einheit aufteilen, die so wenig wie möglich miteinander zu tun haben, und das ist wirklich schwer.

Programmieren ist tatsächlich dermaßen schwer, dass es die Fähigkeiten einer Person übersteigt, das gut zu machen. Egal wie geschickt und bewandert Sie sind, Sie werden mit Sicherheit von den Gedanken und Ideen eines anderen Programmierers profitieren.

4.7.1 Anderen helfen

Deswegen obliegt es der Verantwortung von Programmierern, sich gegenseitig zur Verfügung zu stehen und einander zu helfen. Es ist eine Verletzung der professionellen Ethik, sich im Büro in einer Ecke abzukapseln und Kollegenanfragen von sich abtropfen zu lassen.

⁵ Siehe Kapitel 7, »Akzeptanztests«.

ten zu lassen. Ihre Arbeit ist nicht so wichtig, dass Sie nicht etwas von Ihrer Zeit dafür abzwiegen können, anderen zu helfen. Als Profi muss es für Sie vielmehr Ehrensache sein, diese Hilfe immer anzubieten, wenn sie benötigt wird.

Das heißt nicht, dass Sie keine Zeit für die Arbeit alleine beanspruchen dürften. Natürlich brauchen Sie die. Aber Sie müssen fair und höflich dabei sein. Sie könnten z.B. allseits verlauten lassen, dass Sie zwischen 10 Uhr und Mittag nicht gestört zu werden wünschen, aber Ihre Türe zwischen 13 und 15 Uhr offen steht.

Sie sollten sich über die Situation Ihrer Teamkollegen im Klaren sein und darauf achten. Wenn Sie sehen, dass sich jemand offenkundig mit Problemen abkämpft, sollten Sie Hilfe anbieten. Sie werden wahrscheinlich recht überrascht sein, welchen tief greifenden Effekt Ihre Hilfe haben kann. Es ist nicht so, dass Sie so viel schlauer wären als der andere, sondern Sie bieten eine neue und andere Perspektive, und die kann ein ungeheurer Katalysator für die Problemlösung sein.

Wenn Sie jemandem helfen, setzen Sie sich gemeinsam hin und schreiben zusammen Code. Planen Sie mindestens eine gute Stunde ein, besser mehr. Es könnte schneller gehen, aber Sie sollten nicht den Eindruck erwecken, unter Zeitdruck zu stehen, und hektisch werden. Finden Sie sich in die Aufgabe und hängen Sie sich ehrlich in die Lösung rein. Sie werden hinterher wahrscheinlich mehr gelernt gegeben haben.

4.7.2 Hilfe annehmen

Wenn jemand Ihnen Hilfe anbietet, gehen Sie freundlich damit um. Akzeptieren Sie das Hilfsangebot dankbar und lassen Sie sich helfen. *Versuchen Sie nicht, Ihr Revier zu verteidigen.* Wischen Sie die Hilfe nicht beiseite, weil Sie unter Druck stehen. Räumen Sie diesem Angebot etwa eine halbe Stunde ein. Wenn die Person bis dahin nicht sonderlich viel geholfen hat, entschuldigen Sie sich höflich und beenden die Sitzung mit einem Dank. Denken Sie daran, dass es ebenso eine Ehrensache ist, Hilfe anzubieten, wie auch eine Ehrensache, Hilfe anzunehmen.

Lernen Sie, wie man um Hilfe *bittet*. Wenn Sie irgendwie festhängen oder konfus sind oder ein Problem geistig einfach nicht in den Griff kriegen, bitten Sie jemanden um Hilfe. Wenn Sie in einem Teamraum sitzen, können Sie sich einfach zurücklehnen und sagen: »Ich brauche hier mal Hilfe.« Anderenfalls nutzen Sie Yammer, Twitter oder E-Mail oder das Telefon auf Ihrem Tisch. Melden Sie anderen Ihren Hilfebedarf. Auch dies ist wiederum eine Sache der professionellen Ethik. Es ist unprofessionell, sich damit abzufinden, dass man feststeckt, wenn Hilfe leicht erreichbar ist.

Mittlerweile glauben Sie wohl, dass ich bald fröhlich *Kumbaya* anstimmen werde, während flauschige Häschen Einhörnern auf den Rücken hüpfen und wir alle glücklich und zufrieden über den Regenbogen der Hoffnung und Veränderung fliegen. Nein, so nicht ganz. Wissen Sie, Programmierer neigen dazu, arrogante, mit sich selbst beschäftigte,

introvertierte Leute zu sein. Wir haben uns dieser Branche nicht verschrieben, weil wir *Leute* gerne haben. Die meisten von uns sind zum Programmieren gekommen, weil wir es vorziehen, uns intensiv auf sterile Details zu konzentrieren, mit vielen verschiedenen Konzepten gleichzeitig zu jonglieren und uns generell zu beweisen, dass unsere Hirne so groß wie der Planet sind, alldieweil wir uns nicht mit den chaotischen Komplexitäten *anderer Leute* abgeben wollen.

Ja, dies ist ein Stereotyp. Richtig, es handelt sich um eine Generalisierung mit vielen Ausnahmen. Aber die Realität ist, dass Programmierer zuerst einmal nicht gerne gemeinsam mit anderen arbeiten⁶. Und doch ist die Zusammenarbeit für ein effektives Programmieren absolut wesentlich. Und damit benötigen wir Disziplinen, die uns zur Kollaboration bringen, da vielen von uns eine Zusammenarbeit nicht instinktiv gelingt.

4.7.3 Mentorenarbeit

Diesem Thema widme ich später im Buch ein ganzes Kapitel. Hier will ich einfach nur feststellen, dass das Training weniger erfahrener Programmierer in der Verantwortung jener liegt, die mehr Erfahrung mitbringen. Seminare bringen es nicht, auch Bücher nicht. Nichts bringt einen jungen Software-Entwickler schneller zu Höchstleistungen als sein eigener Antrieb und eine effektive Mentorenarbeit durch erfahrenere Mitarbeiter. Also sei es hier noch einmal gesagt: Es gehört für erfahrene Programmierer zur professionellen Ethik, Zeit dafür einzurichten, die weniger erfahrenen Programmierer unter ihre Fittiche zu nehmen und sie einzuweisen. In gleicher Weise obliegt diesen jüngeren Programmierern die professionelle Pflicht, eine solche Mentorenschaft durch erfahrene Kollegen zu suchen.

4.8 Bibliografie

[Martin09]: Robert C. Martin, *Clean Code*, Upper Saddle River, NJ: Prentice Hall, 2009.

[Martin03]: Robert C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Upper Saddle River, NJ: Prentice Hall, 2003

⁶ Das gilt noch viel eher für Männer als für Frauen. Ich hatte ein wunderbares Gespräch mit @desi (Desi McAdam, der Gründerin von DevChix) darüber, was Frauen motiviert, Programmiererinnen zu werden. Ich erzählte ihr, wenn ich ein Programm ans Laufen bekommen habe, dann wäre das so, als habe ich ein großes, gefährliches Tier erlegt. Sie entgegnete, dass für sie und andere Frauen, mit denen sie gesprochen habe, der Akt des Programmierens ein fürsorglicher Schöpfungsakt sei.

5

Test Driven Development



Es ist schon über zehn Jahre her, dass die testgetriebene Entwicklung (TDD) in der Branche ihr Debüt hatte. Sie gehörte zur Welle des Extreme Programming (XP), wurde aber seitdem von Scrum und praktisch allen anderen agilen Methoden übernommen. Sogar nichtagile Teams praktizieren TDD.

Als ich 1998 zum ersten Mal vom »Test First Programming« hörte, war ich skeptisch. Wer wäre das nicht? Die Unit-Tests *vorher* schreiben? Wer würde denn bloß so etwas Dämliches machen?

Aber da war ich schon seit über dreißig Jahren professioneller Programmierer und hatte in der Branche schon eine Menge kommen und gehen sehen. Mir war klar, dass man das nicht einfach von der Hand wischen konnte, vor allem, wenn jemand wie Kent Beck davon spricht.

Also reiste ich 1999 nach Medford, Oregon, um Kent zu treffen und diese Disziplin von ihm zu lernen. Das war eine echt schockierende Erfahrung!

Ich setzte mich mit Ken in sein Büro, und wir begannen, ein kleines einfaches Problem in Java zu programmieren. Ich wollte dieses blöde Teil einfach nur fertig schreiben. Aber Kent bremste mich und führte mich Schritt für Schritt durch den Prozess. Zuerst schrieb er den ersten kleinen Teil eines Unit-Tests, den man knapp überhaupt als Code bezeichnen konnte. Dann schrieb er gerade so viel Code, damit dieser Test kompilieren konnte. Dann schrieb er ein wenig mehr Testcode, dann noch weiteren Produktionscode.

Die Zykluszeit sprengte meine bisherige Erfahrung komplett. Ich war daran gewöhnt, erst einmal eine Stunde ordentlich zu programmieren, bevor ich versuchte, es zu kompilieren oder zu starten. Aber Kent führte seinen Code etwa alle dreißig Sekunden aus. Ich war baff!

Aber darüber hinaus erkannte ich die Zykluszeit wieder! Es war jene Art Zykluszeit, die ich vor vielen Jahren als Kind¹ beim Programmieren von Spielen in interpretierten Sprachen wie Basic oder Logo verwendet hatte. In diesen Sprachen gibt es keine Build-Zeit, also ergänzt man einfach noch eine Codezeile und führt das Programm dann aus. Man durchläuft diesen Zyklus sehr schnell. Und deswegen ist man in diesen Sprachen *sehr* produktiv.

Doch beim *echten* Programmieren war eine solche Zykluszeit eher absurd. Beim *echten* Programmieren musste man viel Zeit damit verbringen, Code zu schreiben, und dann mehr Zeit damit, ihn zu kompilieren. Und noch viel mehr Zeit, um ihn zu debuggen. *Ich war C++-Programmierer, verdammt noch mal!* Und in C++ hatten wir Build- und Link-Zeiten, die Minuten dauerten und manchmal Stunden. Dreißigsekündige Zykluszeiten? Unvorstellbar.

Und doch gab es da einen Kerl namens Kent, der in dreißigsekündigen Zyklen kontinuierlich an seinem Java-Programm feilte, und nichts an ihm wies darauf hin, dass er sich in nächster Zeit irgendwie verlangsamen würde. Also dämmerte es mir, während ich da in Kents Büro hockte, dass ich mit dieser einfachen Disziplin in echten Sprachen programmieren konnte und dabei die Zykluszeit von Logo hätte! Ich sprang sofort darauf an!

¹ Aus meinem damaligen Blickwinkel war ein Kind jemand, der jünger als 35 war. Während meiner zwanziger Jahre wandte ich beträchtliche Zeit dafür auf, dumme kleine Spiele in interpretierten Sprachen zu schreiben. Ich schrieb Weltraumkampfspiele, Adventures, Pferderennspiele, Snake-Spiele, Zockerspiele – alles, was Ihnen so einfällt.

5.1 Die Geschworenen haben sich entschieden

Seit jenen Tagen habe ich gelernt, dass TDD viel mehr ist als einfach nur ein Trick, um meine Zykluszeiten zu verkürzen. Diese Disziplin weist ein ganzes Repertoire an Vorteilen auf, die ich in den folgenden Absätzen beschreiben werde.

Aber zuerst muss ich hier noch was loswerden:

- Die Geschworenen haben zu einer Entscheidung gefunden!
- Die Kontroverse ist vorbei.
- GOTO ist schädlich.
- Und TDD funktioniert

Richtig, es gab eine Menge kontroverser Blogs und Artikel, die im Laufe der Jahre über TDD geschrieben wurden, und weitere werden kommen. In den frühen Tagen waren es ernsthafte Versuche der Kritik und des Verstehens. Heutzutage handelt es sich einfach nur um irgendwelche Tiraden. Letzten Endes funktioniert TDD, und alle sollten sich damit abfinden.

Ich weiß, das klingt hart und einseitig, aber mit einem Blick auf die Erfolge glaube ich nicht, dass Chirurgen das Händewaschen verteidigen müssten, und ich glaube nicht, dass Programmierer TDD zu verteidigen brauchen.

Wie können Sie sich als Profi betrachten, wenn Sie nicht wirklich *wissen*, dass Ihr gesamter Code funktioniert? Wie können Sie wissen, ob Ihr Code insgesamt funktioniert, wenn Sie ihn nicht jedes Mal testen, sobald Sie eine Änderung vornehmen? Wie können Sie ihn jedes Mal bei einer Änderung testen, wenn Sie keine automatisierten Unit-Tests mit sehr hoher Abdeckung durchführen? Wie können Sie automatisierte Unit-Tests mit einer sehr hohen Abdeckung bekommen, ohne TDD zu praktizieren?

Zu diesem letzten Satz sind wohl noch einige Ausführungen nötig. Was ist denn nun TDD?

5.2 Die drei Gesetze des TDD

1. Sie dürfen erst dann Produktivcode schreiben, wenn Sie vorher einen scheiternden Unit-Test geschrieben haben.
2. Sie dürfen nicht mehr von einem Unit-Test schreiben, als man für ein Scheitern braucht – und mit Scheitern ist »nicht kompilieren« gemeint.
3. Sie dürfen nicht mehr Produktivcode schreiben, als nötig ist, um den aktuell misslingenden Unit-Test zu bestehen.

Diese drei Gesetze sperren Sie in einen Zyklus ein, der vielleicht dreißig Sekunden lang ist. Sie beginnen damit, einen kleinen Teil eines Unit-Tests zu schreiben. Doch schon nach wenigen Sekunden müssen Sie den Namen einer Klasse oder Funktion nennen, die Sie noch gar nicht geschrieben haben, und sorgen damit dafür, dass der Unit-Test nicht kompilieren kann. Also müssen Sie Produktionscode schreiben, durch den der Test kompilieren kann. Aber mehr können Sie dann auch nicht schreiben, und so schreiben Sie dann den nächsten Code für die Unit-Tests.

Diesen Zyklus durchlaufen Sie immer wieder. Sie ergänzen den Testcode ein wenig. Dann ergänzen Sie den Produktionscode ein bisschen. Die beiden Code-Ströme wachsen simultan zu einander entsprechenden Komponenten. Die Tests passen zum Produktionscode wie ein Antikörper zu einem Antigen.

5.2.1 Die Litanei der Vorteile

Gewissheit

Wenn Sie sich TDD als professionelle Disziplin aneignen, werden Sie täglich Dutzende Tests schreiben, pro Woche sind das Hunderte, und in einem Jahr können es Tausende werden. Und all diese Tests haben Sie griffbereit und starten sie jedes Mal, wenn Sie etwas am Code ändern.

Ich bin Hauptentwickler von FitNesse², einem auf Java basierenden Tool für Akzeptanztests, und pflege dieses Tool auch. Während ich dies hier schreibe, gibt es in FitNesse 64.000 Zeilen Quellcode, von denen 28.000 in etwas über 2.200 einzelnen Unit-Tests enthalten sind. Diese Tests decken mindestens 90 % des Produktionscodes³ ab und benötigen 90 Sekunden für einen Durchlauf.

Sobald ich irgendeinen Bereich von FitNesse verändere, starte ich einfach die Unit-Tests. Wenn die erfolgreich abgeschlossen werden, bin ich beinahe sicher, dass meine Überarbeitungen nichts zerschossen haben. Wie sicher ist »beinahe sicher«? Sicher genug, um die Software abzuliefern!

Der Qualitätssicherungsprozess für FitNesse besteht in dem Befehl: `ant release`. Dieser Befehl erstellt das Build für FitNesse von Grund auf und startet dann alle Unit- und Akzeptanztests. Wenn diese Tests erfolgreich absolviert wurden, liefere ich die Software aus.

² <http://fitnesse.org>

³ 90 % ist Minimum. Eigentlich ist die Zahl noch größer. Der genaue Betrag ist schwer zu berechnen, weil die Tools für die Codeabdeckung keinen Code sehen können, der in externen Prozessen oder in Catch-Blöcken läuft.

Injektionsrate für Defekte

Bei FitNesse handelt es sich um keine missionskritische Anwendung. Wenn hier ein Bug vorkommt, wird niemand sterben, und keiner verliert Millionen Dollar. Also kann ich es mir leisten, allein aufgrund der bestandenen Tests die Software abzuliefern. Andererseits hat FitNesse Tausende User, und trotz der 20.000 ergänzten neuen Quellcodezeilen im letzten Jahr weist meine Bug-Liste nur 17 Bugs auf (von denen viele rein kosmetischer Natur sind). Also weiß ich, dass meine Injektionsrate für Defekte sehr gering ist.

Das ist kein isolierter Effekt. Es gab mehrere Berichte⁴ und Studien⁵, die eine signifikante Defektreduzierung beschreiben. Ein Unternehmen nach dem anderen – von IBM über Microsoft bis zu Sabre und Symantec – und ein Team nach dem anderen hat Defektreduktionen mit dem Faktor 2, 5 und sogar 10 berichtet. Das sind Zahlen, die kein Profi ignorieren sollte.

Courage

Warum fixen Sie schlechten Code nicht genau dann, wenn Sie ihn sehen? Wenn Sie eine unordentliche Funktion sehen, wird Ihre erste Reaktion sein: »Was für ein Chaos, das sollte mal aufgeräumt werden.« Und die zweite Reaktion? »Davon lass ich mal die Finger!« Warum? Weil Sie wissen, wenn Sie diesen Code anrühren, könnte er kaputtgehen, und wenn Sie ihn kaputt gemacht haben, haben Sie ihn an den Hacken.

Aber wie wäre es, wenn Sie *sicher* sein könnten, dass durch Ihre Reparaturen nichts kaputtgegangen ist? Wie wäre es, wenn Sie jene Sorte Gewissheit haben könnten, die ich vorhin erwähnt habe? Wie würden Sie das finden, wenn Sie auf einen Button klicken und innerhalb von 90 Sekunden *wissen*, dass Ihre Bearbeitung nichts beschädigt und *nur Gutes* bewirkt hat?

Das ist einer der mächtigsten Vorteile von TDD. Wenn Sie eine Test-Suite haben, der Sie vertrauen, brauchen Sie nie mehr die Überarbeitungen zu fürchten. Wenn Sie schlechten Code sehen, dann beheben Sie das an Ort und Stelle. Der Code wird zu Lehm, den Sie einfach und sicher in simple und wohlgeformte Strukturen formen.

Wenn Programmierer die Angst vorm Aufräumen und Säubern verlieren, dann packen sie es an! Und sauberer Code ist einfacher zu verstehen, leichter zu ändern und besser zu erweitern. Sogar Defekte werden unwahrscheinlicher, weil der Code einfacher ist. Und die Code-Basis *verbessert* sich kontinuierlich, anstatt dass sie auf diese normale Weise vergammelt, an die sich unsere Branche so gewöhnt hat.

Welcher professionelle Programmierer würde zulassen, dass dieser Verwesungsvorgang andauert?

⁴ http://www.objectmentor.com/omSolutions/agile_customers.html

⁵ [Maximilien], [George2003], [Janzen2005], [Nagappan2008]

Dokumentation

Haben Sie schon mal mit einem Framework eines Drittanbieters gearbeitet? Oft schickt der Drittanbieter Ihnen ein schön formatiertes Handbuch, das von technischen Autoren verfasst wurde. Ein typisches Handbuch nutzt 27 8x10-Zoll-Hochglanzfotos mit Kreisen und Pfeilen und einem Text auf jeder Rückseite, auf der erklärt wird, wie man dieses Framework konfiguriert, deployed, manipuliert oder sonst wie verwendet. Am Ende gibt es meist einen Anhang mit einem hässlichen kleinen Abschnitt, der alle Code-Beispiele enthält.

Was schlagen Sie in einem solchen Handbuch zuerst auf? Wenn Sie Programmierer sind, springen Sie gleich zu den Code-Beispielen. Sie gehen zum Code, weil Sie wissen, dass der Code Ihnen die Wahrheit sagt. Die 27 8x10-Hochglanzfotos mit Kreisen und Pfeilen und einem Text auf der Rückseite sehen vielleicht schön aus, aber wenn Sie wissen wollen, wie man mit Code arbeitet, müssen Sie den Code auch lesen.

Wenn Sie die drei Gesetze befolgen, ist jeder von Ihnen geschriebene Unit-Test ein in Code geschriebenes Beispiel, das beschreibt, wie das System eingesetzt werden soll. Wenn Sie diese drei Gesetze befolgen, dann wird es einen Unit-Test geben, der beschreibt, wie man jedes Objekt im System erstellt und alle Arten, wie diese Objekte erstellt werden können. Es wird einen Unit-Test geben, der beschreibt, wie man jede Funktion des Systems aufruft, und zwar auf jede Weise, wie diese Funktionen sinnvoll aufgerufen werden. Für alles, dessen Handhabung Sie kennen müssen, wird es einen Unit-Test geben, der das detailliert beschreibt.

Die Unit-Tests sind Dokumente. Sie beschreiben das Design des Systems auf der untersten Ebene. Sie sind unzweideutig, akkurat und in einer Sprache geschrieben, die von der Zielgruppe verstanden wird, und so formal, dass sie ausgeführt werden können. Sie sind die beste Art einer Dokumentation auf niedrigster Ebene, die möglich ist. Welcher Profi würde eine solche Dokumentation nicht liefern wollen?

Design

Wenn Sie die drei Gesetze befolgen und zuerst Ihre Tests schreiben, stehen Sie vor einem Dilemma. Oft wissen Sie ganz genau, was für einen Code Sie schreiben wollen, aber die drei Gesetze verlangen von Ihnen, dass Sie einen Unit-Test schreiben sollen, der misslingt, weil dieser Code nicht existiert! Das bedeutet, Sie müssen den Code testen, den Sie gerade schreiben wollen.

Das Problem beim Testen von Code ist, dass Sie diesen Code isolieren müssen. Es ist oft schwierig, eine Funktion zu testen, wenn diese andere Funktionen aufruft. Um diesen Test zu schreiben, müssen Sie einen Weg finden, um die Funktion von allen anderen zu entkoppeln. Anders gesagt: Dass Sie zuerst testen müssen, zwingt Sie dazu, über *gutes Design* nachzudenken.

Wenn Sie nicht zuerst Ihre Tests schreiben, bewahrt Sie kein Zwang davor, die Funktionen zu einer untestbaren Masse zu verkoppeln. Wenn Sie die Tests später schreiben, können Sie möglicherweise den Input und Output dieser Masse insgesamt testen, aber wahrscheinlich wird es sehr schwer, einzelne Funktionen zu testen.

Somit schafft das Befolgen der drei Gesetze und das Schreiben von Tests zuerst einen Druck, der Sie zu einem besseren, zu einem entkoppelten Design führt. Welcher Profi würde solche Tools nicht einsetzen, die ihn in Richtung eines besseren Designs vorantreiben?

«Aber ich kann doch meine Tests später schreiben», sagen Sie. Nein, das können Sie nicht. Nicht wirklich. Oh, *manche* Tests können Sie natürlich auch später schreiben. Sie können sogar eine hohe Abdeckung erzielen, wenn Sie sorgfältig darauf achten, was zu messen. Aber Tests, die Sie hinterher schreiben, sind *Rückwärtsverteidigung*. Die Tests, die Sie vorher schreiben, sind *Vorwärtsangriff*. Nachher-Tests werden von jemandem geschrieben, der sich bereits im Code auskennt und weiß, wie das Problem gelöst wurde. Solche Tests können rein gar nicht so treffend sein wie Tests, die vorher geschrieben wurden.

5.2.2 Die professionelle Option

Das Fazit aus alledem lautet, dass TDD die professionelle Option ist. Es handelt sich um eine Disziplin, die Gewissheit, Courage, Defektreduktion, Dokumentation und Design verbessert. Bei alledem, was dafür spricht, sollte man es als unprofessionell betrachten, sie nicht einzusetzen.

5.3 Was TDD nicht ist

Bei all seinen guten Aspekten ist TDD weder Religion noch magische Zauberformel. Bei Befolgen der drei Gesetze ist keiner dieser Vorteile garantiert. Sie können immer noch schlechten Code verfassen, auch wenn Sie Ihre Tests vorher schreiben. Tatsächlich können Sie sogar schlechte Tests schreiben.

Ebenso kann es Zeiten geben, wenn das Befolgen der drei Gesetze einfach unpraktisch oder unangemessen ist. Solche Situationen sind selten, aber sie existieren. Kein professioneller Entwickler sollte jemals eine Disziplin befolgen, die mehr Schaden als Nutzen bewirkt.

5.4 Bibliografie

- [Maximilien]:** E. Michael Maximilien, Laurie Williams: »*Assessing Test-Driven Development at IBM*,« http://collaboration.csc.ncsu.edu/laurie/Papers/MAXIMILIEN_WILLIAMS.PDF
- [George2003]:** B. George, L. Williams: »*An Initial Investigation of Test-Driven Development in Industry*,« <http://collaboration.csc.ncsu.edu/laurie/Papers/TDDpaperv8.pdf>
- [Janzen2005]:** D. Janzen, H. Saiedian: »*Test-driven development concepts, taxonomy, and future direction*,« IEEE Computer, Volume 38, Issue 9, pp. 43–50.
- [Nagappan2008]:** Nachiappan Nagappan, E. Michael Maximilien, Thirumalesh Bhat, Laurie Williams: »*Realizing quality improvement through test driven development: results and experiences of four industrial teams*« Springer Science + Business Media, LLC 2008: http://research.microsoft.com/en-us/projects/esm/nagappan_tdd.pdf



Alle Profis verbessern sich in ihrer Kunst, indem sie Übungen durchführen, um ihr eigenes Geschick zu verbessern. Musiker üben Tonleitern, Football-Spieler rennen durch Reifen, Ärzte üben Nähte und chirurgische Techniken und Anwälte Plädoyers. Soldaten führen Übungsmissionen durch. Wenn die Leistung wichtig ist, machen Profis ihre Exerzitien. In diesem Kapitel geht es vor allem darum, wie Programmierer ihre Kunst üben.

6.1 Etwas Hintergrund übers Üben

Üben und Praktizieren ist bei Software-Entwicklung kein neues Konzept, aber wir erkannten es erst kurz nach der Jahrtausendwende. Vielleicht die erste formale Instanz eines Übungsprogramms erschien auf Seite 6 von [K&R-C].


```
main()  
{  
    printf("hello, world\n");  
}
```

Gibt es jemanden unter uns, der dieses Programm nicht schon einmal in der einen oder anderen Form geschrieben hat? Wir nutzen es dazu, um eine neue Umgebung oder Sprache zu beweisen. Wenn wir dieses Programm schreiben und ausführen, ist das ein Beleg dafür, dass wir *jedes* Programm schreiben und ausführen können.

Als ich noch viel jünger war, war SQINT (die Quadratur von Ganzzahlen) eines der ersten Programme, das ich auf einem neuen Computer geschrieben habe. Ich habe das in Assembler, BASIC, Fortran, COBOL und zig anderen Sprachen geschrieben. Auch hier konnte ich über diesen Weg beweisen, dass ich mit dem Computer all das anstellen konnte, was ich wollte.

Anfang der 1980er-Jahre tauchten die ersten Personal Computer in Kaufhäusern auf. Immer wenn ich an einem vorbeikam, z.B. einem VIC20, einem Commodore 64 oder einem TRS-80, schrieb ich ein kleines Programm, das einen endlosen Strom von ›\<- und ›/<-Zeichen auf dem Bildschirm ausgab. Die von diesem Programm produzierten Muster sahen hübsch aus und wirkten deutlich komplexer als das kleine Programm, mit dem sie generiert wurden.

Obwohl diese kleinen Programme sicherlich Übungsprogramme waren, *übten* Programmierer im Allgemeinen nicht. Offen gestanden sind wir nie auf diesen Gedanken gekommen. Wir hatten genug damit zu tun, unseren Code zu schreiben, als dass wir einen Gedanken darauf verschwendet hätten, wie wir unsere Skills verbessern könnten. Und was hätte das denn überhaupt gebracht? In jenen Jahren brauchte man fürs Programmieren keine schnellen Reaktionen oder flinke Finger. Wir arbeiteten erst seit Ende der 1970er mit Bildschirmeditoren. Einen Großteil unserer Zeit verbrachten wir damit, auf die Kompilierung zu warten oder lange, grässliche Code-Abschnitte zu debuggen. Die Erfindung der kurzen Entwicklungszyklen beim TDD lag noch in ferner Zukunft, also brauchten wir das Finetuning noch nicht, das durch Übung entsteht.

6.1.1 22 Nullen

Aber seit den frühen Tagen des Programmierens hat sich einiges geändert. Manches hat sich *sehr* geändert, anderes hingegen fast gar nicht.

Eine der ersten Maschinen, für die ich überhaupt programmierte, war eine PDP-8/I. Diese Maschine hatte eine Zykluszeit von 1,5 Mikrosekunden. Der Arbeitsspeicher fasste 4.096 12-Bit-Wörter. Sie war so groß wie ein Kühlschrank und verbrauchte nicht unbeträchtliche Mengen Strom. Darin war eine Festplatte verbaut, die 32K 12-Bit-Wörter speichern konnte, und wir sprachen mit ihr über eine Teletype mit zehn Zeichen pro

Sekunde. Wir hielten sie für eine *leistungsfähige* Maschine und nutzten sie, um Wunder zu bewirken.

Gerade habe ich mir einen neuen Laptop geleistet, ein Macbook Pro. Der hat einen 2,8-GHz-Dual-Core-Prozessor, 8 GB RAM, eine 512-GB-SSD-Festplatte und einen 17-Zoll-LED-Monitor mit 1920 x 1200 Bildpunkten. Ich transportiere ihn in meinem Rucksack und stelle ihn mir auf den Schoß. Er verbraucht weniger als 85 Watt.

Mein Laptop ist achttausend Mal so schnell wie die PDP-8/I, hat 2 Millionen Mal mehr Speicher, 16 Millionen Mal mehr Offline-Speicherplatz, verbraucht 1 % der Energie, benötigt nur 1 % ihres Platzbedarfes und kostet ein Fünfundzwanzigstel ihres Preises. Rechnen wir das mal durch:

$$8.000 \times 2.000.000 \times 16.000.000 \times 100 \times 100 \times 25 = 6.4 \times 10^{22}$$

Das ist eine verdammt große Zahl. Wir reden hier von einer *22-fachen Größenordnung!* Das sind so viele Angström wie von hier nach Alpha Centauri. So viele Elektronen gibt es in einem Silberdollar. Das ist die Masse der Erde, wenn man Michael Moore als Einheit nimmt. Das ist eine ganz, ganz große Zahl. Und ich habe so ein Ding auf dem Schoß und Sie wahrscheinlich auch!

Und was mache ich mit dieser Leistungssteigerung um das 22-Fache? Ziemlich genau das Gleiche wie mit dieser PDP-8/I: Ich schreibe *if*-Anweisungen, *while*-Schleifen und *Zuweisungen*.

Oh, ich habe noch bessere Tools, um damit all diese Anweisungen zu schreiben. Und mir stehen dafür auch noch bessere Sprachen zur Verfügung. Aber die Natur dieser Anweisungen hat sich in der ganzen Zeit nicht geändert. Code des Jahres 2011 würde auch ein Programmierer aus den 1960er-Jahren verstehen können. Der Lehm, den wir manipulieren, hat sich in diesen vier Dekaden nicht sonderlich geändert.

6.1.2 Durchlaufzeiten

Aber die Art und Weise, *wie* wir arbeiten, hat sich dramatisch verändert. In den 1960ern durfte ich einen oder zwei Tage darauf warten, bis ich ein Kompilierungsresultat zu sehen bekam. Ende 1970 brauchte ein Programm mit 50.000 Zeilen etwa 45 Minuten zum Kompilieren. Noch in den 1990ern waren lange Build-Zeiten die Norm.

Programmierer warten heutzutage nicht mehr auf Kompilierungen¹. Heutigen Programmierern stehen solche Kapazitäten zur Verfügung, dass sie die Red-/Green-/Re-factor-Schleife in Sekundenschnelle durchlaufen können.

¹ Die Tatsache, dass einige Programmierer wirklich auf ihre Builds warten, ist tragisch und zeugt von Nachlässigkeit. In der heutigen Welt sollte man die Build-Zeit in Sekunden messen, nicht in Minuten und gewiss nicht in Stunden.

Ich arbeite beispielsweise an einem Java-Projekt mit 64.000 Zeilen namens FitNesse. Ein kompletter Build einschließlich aller Unit- und Integrationstests braucht weniger als vier Minuten. Wenn diese Tests bestanden sind, kann ich das Produkt ausliefern. *Also braucht der gesamte Prozess der Qualitätssicherung vom Quellcode bis zum Deployment weniger als vier Minuten.* Für die Kompilierung ist die notwendige Zeit beinahe kaum messbar. Partielle Tests erfordern *Sekunden*. Also kann ich die Schleife Kompilieren/Testen buchstäblich *zehn Mal pro Minute* durchlaufen!

Es ist nicht immer schlau, so schnell zu arbeiten. Oft ist es besser, das Tempo herunterzuschrauben und einfach mal *nachzudenken*². Aber zu anderen Zeiten ist es *höchst produktiv*, diese Schleife so schnell wie möglich durchlaufen zu können.

Alles, was man schnell machen will, muss man üben. Wenn die Code-/Test-Schleife schnell durchlaufen werden soll, müssen Sie dabei Ihre Entscheidungen sehr schnell treffen. Wenn man schnelle Entscheidungen trifft, bedeutet das, eine große Zahl von Situationen und Problemen zu erkennen und einfach zu *wissen*, was man dabei jeweils zu machen hat.

Stellen Sie sich mal zwei Kampfsportler im Wettkampf vor. Jeder muss erkennen, was der andere im Schilde führt, und innerhalb von Millisekunden angemessen reagieren. In einer Kampfsituation bleibt Ihnen nicht der Luxus, die Zeit einzufrieren, die Positionen zu studieren und sich eine angemessene Reaktion auszudenken. In einer Kampfsituation müssen Sie einfach *reagieren*. Tatsächlich ist es Ihr Körper, der reagiert, während der Geist an einer Strategie auf höherer Ebene arbeitet.

Wenn Sie diese Code-/Test-Schleife mehrmals pro Minute durchlaufen, weiß Ihr *Körper* genau, welche Tasten gedrückt werden müssen. Ein wichtiger Bereich Ihres Geistes erkennt die Situation und reagiert innerhalb von Millisekunden mit der passenden Lösung. Dann ist Ihr Geist frei dafür, sich auf höherer Ebene mit dem Problem zu beschäftigen.

Sowohl im Falle der Kampfkunst als auch beim Programmieren hängt die Geschwindigkeit von *Übung und Praxis* ab. Und in beiden Fällen ist die Praxis ähnlich. Wir wählen aus einem Repertoire von Problem-Lösung-Paaren und führen sie immer wieder aus, bis wir sie aus dem Effeff kennen.

Nehmen wir einen Gitarristen wie Carlos Santana. Die Musik in seinem Kopf strömt einfach aus seinen Fingern. Er konzentriert sich nicht auf Fingerpositionen oder Picking-Techniken. Sein Geist ist frei dafür, Melodien und Harmonien auf höherer Ebene zu ersinnen, während auf niedrigerer Ebene sein Körper diese Pläne in Fingerbewegungen umsetzt.

Aber um an diese Art von Leichtigkeit des Spielens zu gelangen, ist *Praxis* unverzichtbar. Musiker üben Tonleitern, Etüden und Riffs immer wieder, bis sie sie im Schlaf kennen.

2 Diese Technik wird von Rich Hickey als HDD bezeichnet: Hammock-Driven Development (etwa: hängemattengetriebene Entwicklung).

6.2 Das Coding Dojo

Seit 2001 führe ich eine TDD-Demo durch, die ich das »Bowling Game³« nenne. Es ist eine hübsche kleine Übung, die etwa dreißig Minuten dauert. Vom Design her wird ein Konflikt aufgebaut, der sich zu einem Höhepunkt steigert und mit einer Überraschung endet. Über dieses Beispiel habe ich in [PPP2003] ein ganzes Kapitel geschrieben.

Im Laufe der Jahre habe ich diese Demonstration viele Hundert, vielleicht auch schon Tausende Male durchgeführt. Ich bin darin *sehr gut* geworden! Ich könnte sie auch im Schlaf machen. Ich habe die Tasteneingaben minimiert, die Variablennamen angepasst und die Algorithmenstruktur verbessert, bis alles genau richtig war. Obwohl ich es damals noch nicht wusste, war dies mein erstes Kata.

2005 nahm ich an der XP2005 Conference im britischen Sheffield teil. Ich besuchte eine Sitzung mit dem Namen *Coding Dojo*, geleitet von Laurent Bossavit und Emmanuel Gaillet. Sie forderten alle auf, ihre Laptops aufzuklappen und mit ihnen zu programmieren, als sie mit TDD das *Game of Life* von Conway schrieben. Sie bezeichneten es als »Kata« und nannten »Pragmatic⁴« Dave Thomas als Urheber der ursprünglichen Idee⁵.

Seitdem haben viele Programmierer für ihre Übungssessions Metaphern aus der Kampfkunst übernommen. Der Name Coding Dojo⁶ scheint sich durchgesetzt zu haben. Manchmal trifft sich eine Gruppe Programmierer und übt gemeinsam, wie es auch Kampfsportler machen würden. Zu anderen Zeiten üben die Programmierer solo – genau wie Kampfsportler.

Ungefähr vor einem Jahr unterrichtete ich eine Gruppe Entwickler in Omaha. Während des Mittagessens luden sie mich ein, an ihrem Coding Dojo teilzunehmen. Ich beobachtete, wie zwanzig Entwickler ihre Laptops öffneten und Taste für Taste ihrem Anleiter nachfolgten, der das Kata *Bowling Game* durchführte.

Es gibt verschiedene Aktivitäten, die in einem Dojo stattfinden können. Ein paar stellen wir hier vor.

6.2.1 Kata

Im Kampfsport ist ein *Kata* eine genau festgelegte Gruppe choreografierter Bewegungen, die eine Seite eines Kampfes simulieren. Das Ziel, auf das asymptotisch hingearbeitet wird, ist Perfektion. Der Künstler ist bestrebt, dass er mit seinem Körper jede Bewegung perfekt ausführt und diese Bewegungen in einer flüssigen Anordnung zusammenstellt. Ein gut ausgeführtes Kata ist eine Augenweide.

3 Dies ist zu einem sehr populären Kata geworden, und mit einer Google-Suche findet man viele Instanzen davon. Das Original befindet sich hier: <http://butunclebob.com/ArticleS.UncleBob.TheBowlingGameKata>.

4 Mit dem vorangestellten »Pragmatic« unterscheiden wir ihn von »Big« Dave Thomas von OTI.

5 <http://codekata.pragprog.com>

6 <http://codingdojo.org/>

Obwohl sie so schön anzuschauen sind, geht es beim Erlernen eines Kata nicht darum, ihn auf der Bühne vorzuführen. Der Zweck besteht darin, den eigenen Geist und Körper dafür zu trainieren, wie man in einer bestimmten Kampfsituation reagieren soll. Das Ziel ist, diese perfektionierten Bewegungen automatisch und instinktiv zu machen, damit man darauf zurückgreifen kann, wenn man sie braucht.

Ein Programmier-Kata ist eine genau festgelegte Gruppe von choreografierten Tasteneingaben und Mausbewegungen, die die Lösung eines Programmierproblems simulieren. Man löst nicht wirklich das Problem, weil man die Lösung bereits kennt. Man übt vielmehr die Bewegungen und Entscheidungen, die zur Problemlösung gehören.

Die Asymptote der Perfektion ist wiederum das Ziel. Sie wiederholen die Übung immer wieder, um Gehirn und Finger zu trainieren, wie sie sich bewegen und reagieren sollen. Wenn Sie üben, werden Sie womöglich subtile Verbesserungen und Wirksamkeiten in Ihren Bewegungen oder der Lösung selbst entdecken.

Eine Suite mit Kata zu praktizieren, ist ein guter Weg, um sich Hotkeys und Navigationsidiome anzueignen. Es ist auch eine gute Möglichkeit, solche Disziplinen wie TDD und CI zu lernen. Doch am wichtigsten ist, dass man sich damit sehr gut übliche Problemlösung-Kombinationen ins Unterbewusstsein einprägen kann, um einfach und schnell darauf zurückgreifen zu können, wenn man im Alltag beim Programmieren darauf stößt.

Wie jeder Kampfsportler sollte auch ein Programmierer mehrere unterschiedliche Kata kennen und sie regelmäßig üben, damit sie in der Erinnerung nicht verblassen. Viele Kata sind unter <http://katas.softwarecraftsmanship.org> gesammelt. Andere kann man bei <http://codekata.pragprog.com> finden. Zu meinen Favoriten gehören:

- *Das Bowling Game*: <http://butunclebob.com/ArticleS.UncleBob.TheBowling-Game-Kata>
- *Prime Factors*: <http://butunclebob.com/ArticleS.UncleBob.ThePrimeFactors-Kata>
- *Word Wrap*: <http://thecleanocoder.blogspot.com/2010/10/craftsman-62-dark-path.html>

Um mal eine echte Herausforderung zu meistern, sollten Sie versuchen, ein Kata so gut zu verinnerlichen, dass Sie es im Takt zu einer Musik machen können. Das gut hinzukriegen, ist wirklich *sehr* schwer⁷.

⁷ <http://katas.softwarecraftsmanship.org/?p=71>

6.2.2 Waza

Als ich Ju-Jutsu trainierte, verbrachten wir viel Zeit im Dojo damit, paarweise unsere *Waza* zu üben. *Waza* könnte man als Kata zu zweit bezeichnen. Man prägt sich die Abläufe präzise ein und spult sie dann wieder ab. Ein Partner spielt die Rolle des Angreifers, und der andere verteidigt sich. Die Bewegungen werden ständig wiederholt, und die Übenden tauschen andauernd die Rollen.

Programmierer können beim Spiel *Pingpong*⁸ ähnlich üben. Die beiden Partner wählen ein Kata oder ein einfaches Problem. Ein Programmierer schreibt einen Unit-Test, und der andere hat die Aufgabe, ihn zu bestehen. Dann tauschen sie die Rollen.

Wenn die Partner ein Standard-Kata wählen, ist das Ergebnis bekannt, und die Programmierer üben und kritisieren gegenseitig die Techniken an Maus und Tastatur und wie gut man sich das Kata jeweils eingeprägt hat. Andererseits wird das Spiel interessanter, wenn die Partner sich entscheiden, ein neues Problem zu lösen. Der Programmierer, der einen Test schreibt, kann weitgehend kontrollieren, wie das Problem gelöst wird. Er besitzt auch die wesentliche Macht, welche Constraints eingesetzt werden. Wenn die Programmierer beispielsweise einen Sortieralgorithmus implementieren wollen, kann der Testautor ganz einfach die Constraints für Geschwindigkeit und Speicherplatz festlegen, mit denen er seinen Partner herausfordert. Das macht das ganze Spiel sehr wettbewerbsorientiert ... und unterhaltsam.

6.2.3 Randori

Randori ist Freistilkampf. In unserem Ju-Jutsu-Dojo legten wir verschiedene Kampfszenarien fest und setzten sie dann um. Manchmal sollte sich einer verteidigen, während wir anderen ihn der Reihe nach angriffen. Manchmal standen zwei oder mehr Angreifer einem einzelnen Verteidiger gegenüber (das war meist der *Sensei*, der fast immer gewann). Manchmal kämpften zwei gegen zwei usw.

Einen simulierten Kampf kann man nicht so gut aufs Programmieren abbilden. Doch gibt es ein Spiel namens *Randori*, das in vielen Coding Dojos gespielt wird. Es ähnelt sehr dem *Waza* zu zweit, bei dem die Partner ein Problem lösen. Allerdings nehmen viele an diesem Spiel teil, und zu den Regeln gehört noch eine überraschende Wende. Der Bildschirm wird an die Wand projiziert. Nun schreibt jemand einen Test und setzt sich wieder hin. Der Nächste sorgt dafür, dass der Test bestanden wird, und schreibt dann den nächsten Test. Das geht dann am Tisch reihum so weiter, oder man stellt sich einfach in einer Schlange an, wenn man so weit ist. In jedem Fall sind diese Übungen sehr unterhaltsam.

8 <http://c2.com/cgi/wiki?PairProgrammingPingPongPattern>

Es ist erstaunlich, wie viel man aus diesen Sessions lernen kann. Es sorgt für immense Erkenntnisse, wie andere Leute Probleme lösen. Diese Einsichten führen unweigerlich dazu, dass Sie Ihren eigenen Ansatz erweitern und Ihre Skills verbessern.

6.3 Die eigene Erfahrung ausbauen

Professionelle Programmierer leiden oft unter mangelnder Vielfalt in der Art, wie sie Probleme lösen. An ihrem Arbeitsplatz werden sie vielfach gezwungen, nur mit einer Sprache, Plattform oder Domäne zu arbeiten. Ohne Einflüsse, die für einen erweiterten Horizont sorgen, verengt das auf sehr ungesunde Weise den Lebenslauf und die Mentalität. Es ist nicht unüblich, dass solche Programmierer auf die Änderungen, die periodisch die ganze Branche umstülpen, gänzlich unvorbereitet sind.

6.3.1 Open Source

Um hier bei den Entwicklungen auf dem Laufenden zu bleiben, sollte man etwas tun, was beispielsweise auch Anwälte und Ärzte machen: Übernehmen Sie ehrenamtliche Arbeit, indem Sie z.B. an einem Open-Source-Projekt mitarbeiten. Die gibt es wie Sand am Meer, und wahrscheinlich gibt es keinen besseren Weg, um das eigene Repertoire zu erweitern, als tatsächlich an etwas zu arbeiten, das für jemand anderes wichtig ist.

Wenn Sie z.B. Java-Programmierer sind, machen Sie bei einem Rails-Projekt mit. Wenn Sie für Ihre Firma sehr viel C++ schreiben, steigen Sie in ein Python-Projekt ein.

6.3.2 Ethisch handeln

Professionelle Programmierer üben in ihrer Freizeit. Es ist nicht Aufgabe Ihres Arbeitgebers, dafür zu sorgen, dass Ihre Skills *up to date* bleiben. Auch muss ein Arbeitgeber sich nicht darum kümmern, dass Ihr Lebenslauf auf der Höhe der Zeit bleibt. Patienten bezahlen ihre Ärzte nicht dafür, dass sie Nähte üben. Football-Fans zahlen (normalerweise) nicht dafür, damit sie zusehen dürfen, wie Spieler beim Training durch Reifen laufen. Konzertbesucher geben kein Geld dafür aus, um Musikern beim Üben von Tonleitern zuzuhören. Und die Arbeitgeber müssen ihren Programmierern keine Vergütung für deren Übungszeit zahlen.

Weil Übungszeit in Ihrer Freizeit stattfindet, müssen Sie nicht die gleichen Sprachen oder Plattformen wie bei Ihrem Arbeitgeber einsetzen. Wählen Sie eine beliebige Sprache und halten Sie Ihre polyglotten Sprachkenntnisse aktuell. Wenn Sie in einem .NET-Laden arbeiten, üben Sie in der Mittagspause oder zu Hause ein bisschen Java oder Ruby.

6.4 Schlussfolgerung

Auf die eine oder andere Weise üben und praktizieren *alle* Profis. Das machen sie, weil ihnen daran liegt, ihren Job so gut wie irgend möglich zu machen. Darüber hinaus üben sie in ihrer Freizeit, weil sie erkennen, dass es ihrer eigenen Verantwortung obliegt – und nicht der ihres Arbeitgebers –, ihre Skills gut geschärft zu halten. Üben ist das, was Sie tun, wenn Sie *nicht* dafür bezahlt werden. Das machen Sie, damit Sie *bezahlt werden*, und zwar gut!

6.5 Bibliografie

[K&R-C]: Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, Upper Saddle River, NJ: Prentice Hall, 1975.

[PPP2003]: Robert C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Upper Saddle River, NJ: Prentice Hall, 2003.



Ein professioneller Entwickler muss in seiner Rolle sowohl kommunizieren als auch programmieren. Erinnern Sie sich daran, dass *Garbage in/Garbage out* auch für Programmierer gilt. Also achten Profi-Programmierer sorgfältig darauf, dass ihre Kommunikation mit anderen Teamangehörigen und dem Business präzise und intakt ist.

7.1 Anforderungen der Kommunikation

Eines der üblichsten Kommunikationsprobleme zwischen Programmierern und Business-Leuten liegt in den Anforderungen. Die Business-Leute geben an, was sie ihrer Meinung nach brauchen, und dann erstellen die Programmierer, was sie durch die Beschreibungen vom Business verstanden haben. Zumindest sollte es so funktionieren. In der Realität ist die Kommunikation von Anforderungen extrem schwierig und der Prozess sehr fehleranfällig.

Als ich 1979 bei Teradyne arbeitete, bekam ich Besuch von Tom, dem Manager des Installations- und Außendienstes. Er bat mich darum, ihm zu zeigen, wie man den Texteditor ED-402 nutzt, um ein einfaches Troubleticket-System zu erstellen.

Der ED-402 war ein für den M365-Computer (Teradynes PDP-8-Klon) geschriebener proprietärer Editor. Als Texteditor war er sehr leistungsfähig. Er besaß eine integrierte Scripting-Sprache, die wir für alle möglichen einfachen Textapplikationen einsetzten.

Tom war kein Programmierer. Doch die Applikation, die ihm vorschwebte, war simpel. Also dachte er, ich könne ihn schnell einweisen und er könne die Applikation dann selbst schreiben. In meiner Naivität dachte ich ebenso. Schließlich war die Scripting-Sprache kaum mehr als eine Makrosprache für die Bearbeitungsbefehle mit sehr rudimentären Entscheidungs- und Schleifenkonstrukten.

Also setzten wir uns gemeinsam hin, und ich fragte ihn, was seine Applikation können sollte. Er begann mit dem Startbildschirm. Ich zeigte ihm, wie man eine Textdatei erstellt, in der die Script-Statements enthalten waren, und wie man die symbolische Repräsentation der Bearbeitungsbefehle in dieses Skript eintippt. Aber als ich ihn anschaute, kam ein völlig leerer Blick zurück. Meine Erklärungen ergaben für ihn einfach keinen Sinn.

Das war das erste Mal, dass ich so etwas erlebte. Für mich war es eine ganz einfache Sache, dass der Editor Befehle symbolisch repräsentiert. Um beispielsweise einen STRG-B-Befehl zu repräsentieren (der Befehl, mit dem der Cursor an den Anfang der aktuellen Zeile gesetzt wird), tippt man einfach ^B in die Skript-Datei. Aber Tom konnte das nicht nachvollziehen. Er war nicht in der Lage, den Wechsel von der Bearbeitung einer Datei hin zur Bearbeitung einer Datei, mit der eine Datei bearbeitet wird, vorzunehmen.

Tom war nicht dumm. Ich glaube, er hatte einfach erkannt, dass hierzu weitaus mehr gehörte, als er anfänglich angenommen hatte. Er wollte weder die notwendige Zeit noch mentale Energie investieren, um etwas so schrecklich Verworrenes zu erlernen wie einen Editor zu nutzen, um einen Editor zu steuern.

Also fügte ich diese Applikation Stück für Stück selber zusammen, während er neben mir hockte und zuschaute. Innerhalb der ersten zwanzig Minuten wurde klar, dass er seinen Fokus vom Lernen, wie man das selbst macht, dazu verlagert hatte aufzupassen, dass ich genau *das* machte, was *er* wollte.

Das dauerte dann den ganzen Tag. Er beschrieb ein Feature, und ich implementierte es unter seiner Beobachtung. Die Zykluszeit betrug etwa fünf Minuten oder weniger, also gab es keinen Grund für ihn, aufzustehen und etwas anderes zu machen. Er sagte mir jeweils, was ich machen sollte, und in fünf Minuten brachte ich es ans Laufen.

Oft zeichnete er das, was er meinte, auf einen Zettel. Einige seiner Wünsche konnte man mit ED-402 nur schwerlich hinkriegen, also schlug ich etwas anderes vor. Wir einigten uns dann auf etwas, was funktionierte, und ich setzte das dann um.

Aber dann probierten wir es aus, und er warf seine Ansicht um. Er sagte dann etwas in der Art wie: »Tja, das hat leider nicht gerade den Fluss, nach dem ich suche. Probieren wir doch was anderes aus.«

Stunde um Stunde fummelten und bastelten wir herum, und langsam nahm diese Applikation Formen an. Wir probierten erst dies aus, dann etwas anderes und schließlich etwas ganz Neues. Mir wurde sehr deutlich, dass er der Bildhauer war und ich das Instrument, das er schwang.

Am Ende bekam er die Applikation, die er gewollt hatte, hatte aber keine Ahnung, wie er das beim nächsten Mal selbst machen konnte. Ich hingegen hatte eine sehr lehrreiche Lektion darüber gelernt, wie Kunden eigentlich merken, was sie brauchen. Ich erfuhr, dass ihre Vision der Features den realen Kontakt mit dem Computer oft nicht überlebt.

7.1.1 Verfrühte Präzisierung

Sowohl Business-Leute als auch Programmierer unterliegen der Versuchung, in die Falle einer verfrühten Präzisierung zu tappen. Die vom Business wollen immer vorher exakt wissen, was sie kriegen, bevor sie ein Projekt autorisieren. Entwickler wollen genau wissen, was von ihnen bei Ablieferung erwartet wird, bevor sie eine Kalkulation des Projekts abgeben. Beide Seiten erwarten eine Präzisierung, die einfach nicht zu bekommen ist, und sind oft bereit, ein Vermögen zu verschwenden, um so etwas zu erlangen.

Das Prinzip der Ungewissheit

Das Problem ist, dass etwas auf Papier anders wirkt als auf einem funktionierenden System. Wenn die Leute vom Business tatsächlich live auf einem System vorgeführt bekommen, was sie als Spezifikation abgegeben haben, erkennen sie, dass sich das gar nicht mit ihren Wünschen deckt. Wenn sie die Anforderung erst einmal live und in Farbe sehen, dann haben sie eine bessere Vorstellung davon, was sie wollen – und das passt meist nicht zu dem, was sie sehen.

Dabei spielt eine Art Beobachtereffekt oder ein Prinzip der Ungewissheit mit hinein. Wenn Sie den Business-Leuten ein Feature demonstrieren, bekommen die mehr Informationen als vorher, und diese neue Info wirkt sich darauf aus, wie sie das ganze System betrachten.

Am Ende läuft es darauf hinaus, dass die Anforderungen umso irrelevanter werden, wenn das System implementiert ist, je präziser Sie die Anforderungen formulieren.

Angst vorm Einschätzen

Entwickler können ebenfalls in die Falle der Präzisierung geraten. Sie wissen, dass sie das System kalkulieren und abschätzen müssen, und glauben meist, dass dafür Präzision erforderlich ist – was aber nicht stimmt.

Zum einen wird Ihre Kalkulation eine riesige Streuung aufweisen, auch wenn Ihnen perfekte Informationen verfügbar sind. Zum anderen macht das Prinzip der Ungewissheit aus der zu frühen Präzisierung Hackfleisch. Die Anforderungen werden sich verändern, und das stellt die Präzisierung infrage.

Professionelle Entwickler verstehen, dass Kalkulationen auf weniger präzisen Anforderungen erstellt werden können und sollten, und akzeptieren, dass diese Kalkulationen eben *Schätzwerte* sind. Um dies noch zu bekräftigen, bauen professionelle Entwickler stets Fehlerbalken in ihre Kalkulationen ein, damit die Business-Leute die Ungewissheit verstehen (siehe Kapitel 10, »Kalkulationen«).

Späte Mehrdeutigkeit

Die Lösung für eine vorzeitige Präzisierung ist, sie möglichst lange hinauszuschieben. Professionelle Entwickler arbeiten eine Anforderung erst dann weiter aus, kurz bevor sie sie direkt entwickeln. Das kann allerdings zu einem anderen Übel führen: die späte Mehrdeutigkeit.

Oft sind die Stakeholder unterschiedlicher Meinung. Wenn das vorkommt, dann finden sie es meist einfacher, dieser Meinungsverschiedenheit mit blumigen Worten aus dem Wege zu gehen anstatt sie zu lösen. Sie werden einen Weg finden, die Anforderung so zu formulieren, dass alle zustimmen können, ohne den eigentlichen Disput aufzulösen. Tom DeMarco habe ich mal sagen hören: »Eine Mehrdeutigkeit in einem Anforderungsdokument steht für einen Konflikt bei den Stakeholdern¹.«

Natürlich braucht es keine Auseinandersetzung oder Meinungsverschiedenheit, um für Mehrdeutigkeit zu sorgen. Manchmal gehen die Stakeholder einfach davon aus, dass ihre Leser wissen, was sie meinen. In deren Kontext ist alles völlig klar, aber für einen Programmierer, der es liest, kann es etwas völlig anderes bedeuten. Diese Art kontextabhängiger Mehrdeutigkeit passiert auch, wenn Kunden und Programmierer direkt miteinander sprechen.

Sam (Stakeholder): »Okay, diese Log-Dateien müssen gesichert werden.«

Paula: »In Ordnung. Wie oft?«

Sam: »Täglich.«

Paula: »Gut. Und wo sollen die gespeichert werden?«

¹ XP Immersion 3. Mai 2000, <http://c2.com/cgi/wiki?TomsTalkAtXplmersionThree>

- Sam: »Was meinen Sie damit?«
- Paula: »Nun, sollen sie in einem bestimmten Unterverzeichnis abgelegt werden?«
- Sam: »Ja, das wäre gut.«
- Paula: »Wie soll das heißen?«
- Sam: »Man könnte es doch ›Backup‹ nennen.«
- Paula: »Das ist in Ordnung. Also schreiben wir täglich die Log-Datei ins Verzeichnis Backup. Um wie viel Uhr?«
- Sam: »Jeden Tag.«
- Paula: »Nein, ich meine, zu welcher Zeit am Tag soll das geschrieben werden?«
- Sam: »Egal wann.«
- Paula: »Mittags?«
- Sam: »Nein, nicht während der Bürozeiten. Mitternacht wäre besser.«
- Paula: »Okay, dann eben um Mitternacht.«
- Sam: »Toll, vielen Dank!«
- Paula: »Gern geschehen.«

Später berichtet Paula ihrem Teamkollegen Peter über die Aufgabe.

- Paula: »Wir sollen also täglich um Mitternacht die Log-Datei in ein Unterverzeichnis namens Backup kopieren.«
- Peter: »Und welchen Dateinamen sollen wir dafür nehmen?«
- Paula: »log.backup reicht doch.«
- Peter: »Okay, wird erledigt.«

In einem anderen Büro telefoniert Sam mit seinem Kunden.

- Sam: »Ja, sicher, die Log-Dateien werden gespeichert.«
- Carl: »Gut, denn es ist absolut entscheidend, dass wir keine Logs verlieren. Wir müssen diese Log-Dateien immer durchsehen können, auch noch nach Monaten oder Jahren, sobald es einen Ausfall, einen Vorfall oder einen Streit gibt.«

Sam: »Keine Sorge, ich habe gerade mit Paula gesprochen. Sie speichert die Logs jeden Tag um Mitternacht in einem Verzeichnis namens Backup.«

Carl: »Okay, klingt gut.«

Ich nehme mal an, dass Ihnen die Mehrdeutigkeit aufgefallen ist. Der Kunde erwartet, dass alle Log-Dateien gespeichert werden, und Paula nahm einfach an, dass nur die Log-Datei des letzten Tages zu sichern ist. Wenn der Kunde dann nach dem monatlichen Bestand seiner Log-Datei-Sicherungen sucht, findet er nur die von gestern.

In diesem Fall haben Paula und Sam die Sache gemeinsam versemmt. Es obliegt der Verantwortung der professionellen Entwickler (und Stakeholder), darauf zu achten, dass jegliche Mehrdeutigkeit aus den Anforderungen entfernt wird.

Das ist schwer, und es gibt meines Wissens nur einen Weg, wie man das machen kann.

7.2 Akzeptanztests

Der Begriff *Akzeptanztest* ist überladen und überstrapaziert. Manche nehmen an, dass damit die Tests gemeint sind, die User ausführen, bevor sie ein Release akzeptieren. Andere halten es für Tests der Qualitätssicherung. In diesem Kapitel definieren wir Akzeptanztests als Tests, die gemeinsam von Stakeholdern und Programmierern geschrieben werden, um zu definieren, *ob eine Anforderung erfüllt worden ist*.

7.2.1 Die »Definition of Done«

Eine der häufigsten Mehrdeutigkeiten, mit denen wir als Software-Profis zu tun haben, ist die, was mit »done«, also »erfüllt« gemeint ist. Wenn ein Entwickler sagt, er hat eine Aufgabe erfüllt, was bedeutet das eigentlich? Ist der Entwickler fertig in dem Sinne, dass er voller Zutrauen bereit ist, das Feature zu deployen? Oder heißt das, er sei bereit für die Qualitätssicherung? Oder er ist mit dem Schreiben fertig und hat es einmal starten können, aber noch nicht wirklich getestet.

Ich habe mit Teams gearbeitet, die die Wörter »erfüllt« und »komplett« unterschiedlich definierten. Ein bestimmtes Team arbeitete gar mit den Begriffen »done« und »done-done«.

Professionelle Entwickler haben eine einzige »Definition of Done«: *Done* bedeutet *done*, also fertig und abgeschlossen. Das bedeutet: Der gesamte Code ist geschrieben, hat alle Tests bestanden, und die Qualitätssicherung sowie die Stakeholder haben ihn akzeptiert. Also alles »fertig«.

Aber wie können Sie diesen Erfüllungsgrad bekommen und immer noch von einer Iteration zur nächsten schnelle Fortschritte machen? Sie erstellen eine Gruppe automatisierter Tests, und wenn alle bestanden werden, sind die obigen Kriterien erfüllt! Wenn die Abnahmeprüfungen für Ihr Feature bestanden wurden, sind Sie »done«.

Professionelle Entwickler treiben die Definition ihrer Anforderungen bis hin zu den automatisierten Abnahmeprüfungen. Sie arbeiten mit Stakeholdern und Qualitätssicherung zusammen, damit gewährleistet bleibt, dass diese automatisierten Tests eine komplette Spezifikation von »done« darstellen.

- Sam: »Okay, diese Log-Dateien müssen gesichert werden.«
- Paula: »In Ordnung. Wie oft?«
- Sam: »Täglich.«
- Paula: »Gut. Und wo sollen die gespeichert werden?«
- Sam: »Was meinen Sie damit?«
- Paula: »Nun, sollen sie in einem bestimmten Unterverzeichnis abgelegt werden?«
- Sam: »Ja, das wäre gut.«
- Paula: »Wie sollen wir das nennen?«
- Sam: »Man könnte das doch »Backup« nennen.«
- Tom(Tester): »Moment mal, Backup ist ein zu allgemeiner Name. Was soll denn wirklich in diesem Verzeichnis gespeichert werden?«
- Sam: »Die Backups.«
- Tom: »Backups wovon?«
- Sam: »Der Log-Dateien.«
- Paula: »Aber es gibt doch nur eine Log-Datei.«
- Sam: »Nein, es soll mehrere geben. Eine für jeden Tag.«
- Tom: »Sie meinen, dass es eine *aktive* Log-Datei geben soll und viele Log-Datei-Backups?«
- Sam: »Natürlich.«
- Paula: »Oh, und ich dachte, Sie wollten einfach nur ein temporäres Backup.«
- Sam: »Nein, der Kunde will sie allesamt aufbewahren können.«
- Paula: »Ah, das war mir nicht klar. Gut, dass wir das geklärt haben.«

- Tom: »Also sollte der Name des Unterverzeichnisses genau darauf verweisen, was sich darin befindet.«
- Sam: »Es enthält alle alten inaktiven Logs.«
- Tom: »Also nennen wir es `alte_inaktive_logs`.«
- Sam: »Genau, richtig.«
- Tom: »Und wann soll dieses Verzeichnis erstellt werden?«
- Sam: »Wie bitte?«
- Paula: »Wir sollten das Verzeichnis erstellen, wenn das System startet, aber nur, falls das Verzeichnis noch nicht existiert.«
- Tom: »Okay, da haben wir unseren ersten Test. Ich starte das System und schaue, ob das Verzeichnis `alte_inaktive_logs` erstellt wurde. Dann füge ich in dieses Verzeichnis eine Datei ein. Dann fahre ich alles runter und starte neu und achte darauf, dass sowohl das Verzeichnis als auch die Datei immer noch vorhanden sind.«
- Paula: »Dieser Test wird ziemlich lange dauern. Das System braucht zum Hochfahren schon jetzt 20 Sekunden, und das wird mehr werden. Außerdem habe ich keine Lust, jedes Mal das ganze System neu zu erstellen, sobald ich den Akzeptanztest starte.«
- Tom: »Was schlagen Sie also vor?«
- Paula: »Wir erstellen eine `SystemStarter`-Klasse. Das Hauptprogramm wird diese Klasse mit einer Gruppe von `StartupCommand`-Objekten laden, die das `COMMAND`-Muster befolgen. Während das System hochfährt, sagt der `SystemStarter` allen `StartupCommand`-Objekten einfach, dass sie starten sollen. Eines jener `StartupCommand`-Derivate erstellt das Verzeichnis `alte_inaktive_logs`, aber nur, falls es noch nicht existiert.«
- Tom: »Okay, dann brauche ich ja nur dieses `StartupCommand`-Derivat zu testen. Ich kann einen einfachen *FitNesse*-Test dafür schreiben.«

Tom geht an die Tafel.

»Der erste Teil wird etwa wie folgt aussehen:«

Gegeben sei der Befehl `LogFileDirectoryStartupCommand`
Gegeben sei, dass das Verzeichnis `alte_inaktive_logs` nicht existiert, wenn der Befehl ausgeführt wird.
Dann soll das Verzeichnis `alte_inaktive_logs` existieren,
und es soll leer sein

»Der zweite Teil wird etwa wie folgt aussehen:«

Gegeben sei der Befehl `LogFileDirectoryStartupCommand`
 Gegeben sei, dass das Verzeichnis `alte_inaktive_logs` existiert und eine
 Datei namens `x` enthält, wenn der Befehl ausgeführt wird.
 Dann soll das Verzeichnis `alte_inaktive_logs` immer noch existieren,
 und es soll immer noch eine Datei namens `x` enthalten

Paula: »Genau, damit hätten wir das abgedeckt.«

Sam: »Meine Güte, ist das denn alles nötig?«

Paula: »Sam, welches dieser beiden Statements muss man nicht näher beschreiben, weil es weniger wichtig ist?«

Sam: »Ich meine nur, dass das nach einer Menge Arbeit aussieht, sich all diese Tests auszudenken und zu schreiben.«

Tom: »Das ist es auch, aber es ist nicht mehr Arbeit, als einen manuellen Testplan zu verfassen. Und es ist *deutlich* mehr Arbeit, einen manuellen Test wiederholt auszuführen.«

7.2.2 Kommunikation

Der Zweck solcher Akzeptanztests ist Kommunikation, Klarheit und Präzision. Indem sie sich einigen, verstehen die Entwickler, Stakeholder und Tester alle, wie das Systemverhalten geplant ist. Diese Art von Klarheit zu erreichen, liegt in der Verantwortung aller Parteien. Professionelle Entwickler machen es zu ihrer Verantwortung, mit Stakeholdern und Testern zu arbeiten, damit gewährleistet bleibt, dass alle Seiten genau wissen, was erstellt wird.

7.2.3 Automatisierung

Akzeptanztests sollten *immer* automatisiert werden. Im Lifecycle einer Software gehören manuelle Tests an eine andere Stelle, aber diese Art von Tests sollten nie manuell durchgeführt werden. Das hat einen einfachen Grund: die Kosten.

Schauen Sie sich mal die Abbildung 7.1 an. Die Hände, die Sie da sehen, gehören dem Manager für Qualitätssicherung einer großen Internet-Firma. Das Dokument, das er zeigt, ist das *Inhaltsverzeichnis* für seinen *manuellen* Testplan. Er hat eine Armee manueller Tester in Billiglohnländern, die diesen Plan alle sechs Wochen einmal ausführen. Das kostet ihn jedes Mal über eine Million Dollar. Er hält das für mich hin, weil er gerade von einem Meeting zurückkommt, bei dem sein Manager ihm gesagt hat, dass man sein Budget um 50 % reduzieren müsse. Seine Frage an mich lautete: »Welche Hälfte dieser Tests sollte ich nicht durchführen?«

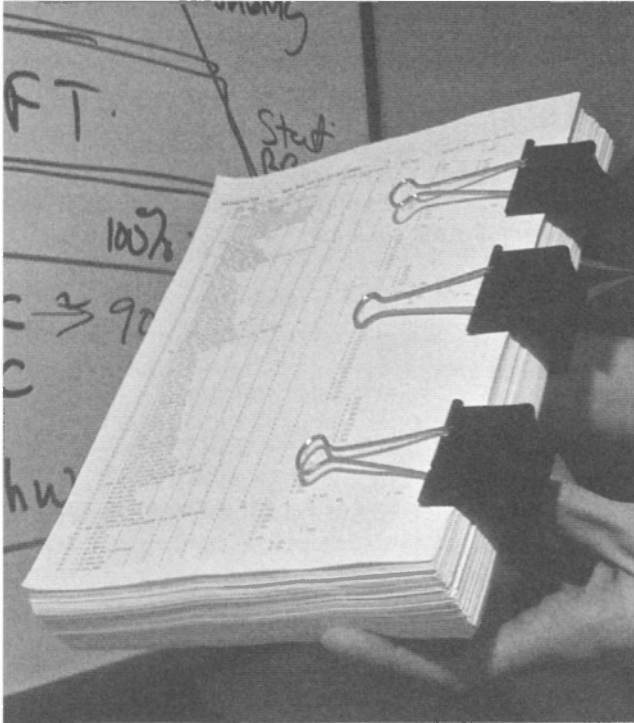


Abbildung 7.1: Manueller Testplan

Dies als Desaster zu bezeichnen, wäre eine grobe Untertreibung. Die Kosten für die Durchführung dieses manuellen Testplans sind derart enorm, dass einfach beschlossen wurde, sie zu opfern, und man gibt sich einfach mit der Tatsache zufrieden, dass man nicht weiß, *ob eine Hälfte des Produkts überhaupt funktioniert!*

Professionelle Entwickler lassen eine solche Situation gar nicht erst zu. Die Kosten für die Automatisierung von Abnahmeprüfungen sind im Vergleich zu den Kosten für manuelle Testpläne dermaßen gering, dass es ökonomisch nicht sinnvoll ist, Skripte zu schreiben, die von Menschen ausgeführt werden. Professionelle Entwickler übernehmen die Verantwortung für ihren Part insofern, dass sie darauf achten, dass Akzeptanztests automatisiert werden.

Es gibt viele Open-Source- und kommerzielle Tools, die die Automatisierung solcher Akzeptanztests erleichtern. FitNesse, Cucumber, cuke4duke, robot framework und Selenium sind nur einige davon. Mit all diesen Tools können Sie automatisierte Tests in einer Form festlegen, die auch Nichtprogrammierer lesen, verstehen und sogar erstellen können.

7.2.4 Zusätzliche Arbeit

Sams Anmerkung über die viele Arbeit ist verständlich. Es sieht *wirklich* nach einer Menge zusätzlicher Arbeit aus, wenn man solche Akzeptanztests schreibt. Aber mit Blick auf Abbildung 7.1 erkennen wir, dass es gar keine Extraarbeit ist. Das Schreiben solcher Tests gehört einfach zu der Arbeit, das System näher zu spezifizieren. Auf dieser Detailstufe genau zu spezifizieren, ist der einzige Weg, wie wir als Programmierer wissen können, was »done« bedeutet. Nur durch Spezifizieren auf dieser Detailstufe können die Stakeholder davon ausgehen, dass das System, für das sie zahlen, wirklich genau das macht, was sie brauchen. Und nur durch eine detaillierte Spezifizierung kann man die Tests erfolgreich automatisieren. Also betrachten Sie diese Tests nicht als zusätzliche Arbeit, sondern als riesiges Einsparpotenzial für Zeit und Geld. Diese Tests bewahren Sie davor, das falsche System zu implementieren, und erlauben Ihnen zu *wissen*, wann Sie wirklich fertig sind.

7.2.5 Wer schreibt die Akzeptanztests und wann?

In einer idealen Welt würden Stakeholder und Qualitätssicherung gemeinsam am Verfassen dieser Tests arbeiten, und die Entwickler würden sie auf Konsistenz hin prüfen. In der realen Welt haben Stakeholder selten die Zeit oder die Neigung, sich in den erforderlichen Detailgrad einzuarbeiten. Also delegieren sie die Verantwortung oft an Business-Analysten, Qualitätssicherung oder gar die Entwickler. Wenn sich herausstellt, dass Entwickler diese Tests schreiben müssen, dann achten Sie sorgfältig darauf, dass ein anderer Entwickler den Test schreibt als der, der das zu testende Feature implementiert hat.

Normalerweise nehmen Business-Analysten beim Schreiben der Tests den »Standardfall«, weil diese Tests die Features mit echtem Business-Wert beschreiben. Die Qualitätssicherung nimmt für diese Tests eher den Fehlerfall an und schreibt die Grenzwertbedingungen, Ausnahmen und selten auftretenden Fälle. Es gehört schließlich zur Aufgabe der Qualitätssicherung, auch daran zu denken, was schiefgehen kann.

Das Prinzip der »späten Präzisierung« befolgend, sollten Abnahmeprüfungen so spät wie möglich geschrieben werden, üblicherweise nur wenige Tage, bevor das Feature implementiert wird. In agilen Projekten werden die Tests geschrieben, *nachdem* die Features für die nächste Iteration oder den nächsten Sprint ausgewählt wurden.

Die ersten paar Akzeptanztests sollten am ersten Tag der Iteration fertig sein. Bis zur Mitte der Iteration sollten sie nach und nach fertiggestellt werden, und dann sollten alle bereit sein. Wenn in der Mitte der Iteration nicht alle Akzeptanztests fertig sind, sollten ein paar Entwickler einspringen, um sie fertigzustellen. Kommt das häufiger vor, dann sollten mehr Business-Analysten und/oder Personal für die Qualitätssicherung ins Team aufgenommen werden.

7.2.6 Die Rolle des Entwicklers

Die Arbeit an der Implementierung eines Features beginnt, wenn die Akzeptanztests für dieses Feature bereit sind. Die Entwickler führen die Tests für das neue Feature aus und beobachten, wie sie scheitern. Dann arbeiten sie daran, den Akzeptanztests mit dem System zu verbinden, und der Test wird dann bestanden, indem das gewünschte Feature implementiert wird.

Paula: »Peter, können Sie uns mal bei dieser Story helfen?«

Peter: »Klar, Paula, worum geht's denn?«

Paula: »Hier ist der Akzeptanztest. Wie Sie sehen können, klappt's nicht.«
 Gegeben sei der Befehl `LogFileDirectoryStartupCommand`
 Gegeben sei, dass das Verzeichnis `alte_inaktive_logs` nicht existiert, wenn der Befehl ausgeführt wird.
 Dann soll das Verzeichnis `alte_inaktive_logs` existieren und es soll leer sein

Peter: »Ja, alles rot. Keines der Szenarios ist geschrieben. Ich schreibe dann mal das erste.«
`|scenario|given the command _|cmd|`
`|create command|@cmd|`

Paula: »Haben wir bereits eine `createCommand-Operation`?«

Peter: »Ja, die ist im `CommandUtilitiesFixture`, das ich letzte Woche geschrieben habe.«

Paula: »Okay, dann mal los mit dem Test.«

Peter (startet Test): »Ja, die erste Zeile ist grün. Weiter geht's mit der nächsten.«

Halten Sie sich nicht zu sehr mit Szenarien und Zuständen auf. Die gehören einfach zu den Verdrahtungen mit dazu, die Sie schreiben müssen, um die Tests mit dem zu testenden System zu verbinden. Fürs Erste sollte es reichen festzustellen, dass die Tools alle irgendeine Art bieten, um anhand eines Musterabgleichs die Statements des Tests zu erkennen und zu parsen und dann die Funktionen aufzurufen, die die Daten im Test in das zu testende System einspeisen. Der Aufwand ist eher gering, und die Szenarien und Zustände kann man in vielen weiteren Tests wiederverwenden.

Hier soll vor allem angesprochen werden, dass es zur Aufgabe des Entwicklers gehört, die Akzeptanztests mit dem System zu verbinden und dann die Tests erfolgreich zu absolvieren.

7.2.7 Verhandlungen über die Tests und passive Aggression

Testautoren sind nur Menschen und machen Fehler. Manchmal ergeben die Tests, so wie sie geschrieben sind, beim Implementieren nicht sonderlich viel Sinn. Vielleicht sind sie zu kompliziert oder zu umständlich. Möglicherweise setzen sie dumme Annahmen voraus. Oder sie sind einfach falsch. Das kann sehr frustrierend sein, wenn Sie als Entwickler dafür zu sorgen haben, dass der Test erfolgreich absolviert wird.

Als professioneller Entwickler gehört es zu Ihren Aufgaben, mit dem Testautor einen besseren Test auszuhandeln. Was Sie jedenfalls nie machen sollten: sich für die passiv-aggressive Option zu entscheiden und zu sagen: »Tja, das ist halt das Testergebnis, also mache ich das auch so.«

Denken Sie daran, dass es als Profi Ihr Job ist, Ihrem Team dabei zu helfen, die bestmögliche Software zu schaffen. Das bedeutet, dass alle auf Fehler und Pannen zu achten und gemeinsam an deren Korrektur zu arbeiten haben.

- Paula: »Tom, dieser Test haut nicht richtig hin.«
Darauf achten, dass die Postoperation in 2 Sekunden beendet ist.
- Tom: »Für mich sieht das okay aus. Unsere Anforderung ist, dass User nicht länger als 2 Sekunden warten sollen. Worin liegt das Problem?«
- Paula: »Das Problem ist, dass wir das nur in statistischer Hinsicht garantieren können.«
- Tom: »Wie bitte? Das hört sich aber irgendwie ungenau an. Die Anforderung sind zwei Sekunden.«
- Paula: »Richtig, und das gelingt uns 99,5 % der Zeit.«
- Tom: »Das ist nicht die Anforderung, Paula.«
- Paula: »Aber die Realität. Ich kann das ehrlicherweise auf keine andere Weise garantieren.«
- Tom: »Sam wird einen Anfall kriegen.«
- Paula: »Nein, denn ich habe bereits mit ihm darüber gesprochen. Für ihn ist das okay, solange die *normale* User Experience zwei Sekunden oder weniger beträgt.«
- Tom: »Okay, wie soll ich also diesen Test schreiben? Ich kann da nicht einfach sagen, dass die Post-Operation *normalerweise* in zwei Sekunden fertig ist.«
- Paula: »Das wird dann eben statistisch gesagt.«

- Tom: »Sie meinen, dass ich Tausend Post-Operations laufen lassen und dabei darauf achten soll, dass nicht mehr als fünf länger als zwei Sekunden dauern? Das ist absurd.«
- Paula: »Nein, das würde bis zu einer Stunde dauern. Wie wäre es denn hiermit?«
Führe 15 Post-Transaktionen aus und akkumulierte die Dauer.
Gewährleiste, dass der Z-Faktor für 2 Sekunden mindestens 2.57 beträgt.
- Tom: »Huch, was ist denn ein Z-Faktor?«
- Paula: »Nur ein bisschen Statistik. Hier, wie wäre es so?«
Führe 15 Post-Transaktionen aus und akkumulierte die Dauer.
Gewährleiste, dass die Chancen 99,5 % stehen, dass die Dauer nicht länger als 2 Sekunden ist.
- Tom: »Ja, das ist lesbar, jedenfalls irgendwie, aber kann ich den zugrunde liegenden Berechnungen trauen?«
- Paula: »Ich werde darauf achten, dass alle Zwischenrechnungen im Testbericht aufgeführt werden, damit Sie sie nachvollziehen können, falls Sie Bedenken haben.«
- Tom: »Okay, das wäre für mich in Ordnung.«

7.2.8 Akzeptanz- und Unit-Tests

Akzeptanztests sind keine *Unit-Tests*. Unit-Tests werden *von* Programmierern *für* Programmierer geschrieben. Es handelt sich um formale Designdokumente, die die Struktur auf niedrigster Ebene und das Verhalten des Codes beschreiben. Gedacht ist das für Programmierer, nicht fürs Business.

Akzeptanztests werden *vom* Business *fürs* Business geschrieben (auch wenn Sie als Entwickler möglicherweise drauf hängen bleiben, sie zu schreiben). Es sind formale Anforderungsdokumente, die angeben, wie sich das System aus Sicht des Business verhalten soll. Sie wenden sich an Business *und* Programmierer.

Es kann verführerisch sein, sich die »Extraarbeit« zu sparen, indem man davon ausgeht, dass die beiden Testarten redundant sind. Obwohl es zutrifft, dass Unit- und Akzeptanztests oft die gleichen Dinge testen, sind sie absolut nicht redundant.

Erstens erledigen beide Testarten das über unterschiedliche Mechanismen und Vorgehensweisen, auch wenn sie die gleichen Sachen testen. Unit-Tests steigen tief in die Innereien des Systems ein und rufen Methoden in bestimmten Klassen auf. Akzeptanz-

tests aktivieren das System auf einer deutlich höheren Ebene auf z.B. der API- oder manchmal auch erst auf der UI-Ebene. Also unterscheiden sich die Ausführungspfade dieser Tests sehr deutlich voneinander.

Aber diese Tests sind schon deswegen nicht redundant, sondern weil ihre primäre Funktion nicht das *Testen an sich* ist. Dass es sich um Tests handelt, ist nur eine zufällige Tatsache. Unit- und Akzeptanztests sind *in erster Linie Dokumente* und erst in zweiter Linie Tests. Sie dienen vor allem dem Zweck, das Design, die Struktur und das Verhalten eines Systems formell zu dokumentieren. Die Tatsache, dass sie das spezifizierte Design, Struktur und Verhalten automatisch verifizieren, ist höchst praktisch, aber ihre wahre Aufgabe ist die Spezifikation.

7.2.9 GUIs und andere Komplikationen

Es ist schwer, GUIs vorab zu spezifizieren. Das geht schon, aber meist gelingt das nicht sonderlich gut. Das liegt daran, dass Ästhetik subjektiv und somit unberechenbar ist. Die Leute wollen an den GUIs *herumspielen*. Sie wollen sie kneten und manipulieren. Sie wollen verschiedene Schriften und Farben, unterschiedliche Seitenlayouts und auch Workflows ausprobieren. GUIs befinden sich im ständigen Fluss.

Dadurch ist es eine solche Herausforderung, Akzeptanztests für GUIs zu schreiben. Der Trick besteht darin, das System so zu gestalten, dass man die GUIs wie APIs behandelt anstatt als eine Gruppe von Buttons, Schieberegler, Gittern und Menüs. Das hört sich vielleicht eigenartig an, aber es ist wirklich einfach nur gutes Design.

Es gibt ein Designprinzip namens *Single Responsibility Principle* (SRP). Diesem Prinzip zufolge sollte man jene Dinge separieren, die sich aus unterschiedlichen Gründen ändern können, und sie mit solchen Dingen gruppieren, die sich aus den gleichen Gründen ändern. GUIs bilden da keine Ausnahme.

Das Layout, Format und der Workflow der GUI können sich aus Gründen der Ästhetik und Effektivität ändern, aber die zugrunde liegende Kapazität der GUI bleibt trotz dieser Änderungen gleich. Wenn man also Akzeptanztests für eine GUI schreibt, nutzt man die zugrunde liegenden Abstraktionen, die sich nicht häufig ändern.

Es kann beispielsweise mehrere Buttons auf der Seite geben. Anstatt Tests zu kreieren, bei denen man auf die Buttons basierend auf ihrer Position auf der Seite klickt, sollte man anhand ihrer Namen auf sie klicken können. Noch besser wäre, wenn sie jeweils eine eindeutige ID haben. Es ist viel besser, einen Test zu schreiben, der den Button auswählt, dessen ID dann z.B. `ok_button` lautet, als jenen Button in Spalte 3 von Zeile 4 der Gitteransicht zu selektieren.

Anhand der richtigen Schnittstelle testen

Besser ist es, Tests zu schreiben, die die Features des zugrunde liegenden Systems über eine echte API aufrufen anstatt die GUI. Diese API sollte die gleiche sein, mit der auch die GUI arbeitet. Das ist nicht neu, und Designexperten beten uns schon seit Jahrzehnten vor, dass wir unsere GUIs von unseren Business-Regeln trennen sollen.

Tests, die über die GUI laufen, sind immer problematisch, außer Sie testen wirklich *nur* die GUI. Das liegt daran, dass sich die GUI wahrscheinlich noch ändert und deswegen die Tests ziemlich fragil sind. Wenn durch jede GUI-Änderung Tausend Tests zerstört werden, werden Sie entweder die Tests über Bord werfen oder einfach die GUI nicht mehr verändern. Beides keine guten Optionen. Also schreiben Sie die Tests für die Business-Regeln anhand einer API, die sich direkt unterhalb der GUI befindet.

Manche Akzeptanztests legen das Verhalten der GUI selbst fest. Diese Tests *müssen* die GUI durchlaufen. Allerdings werden damit keine Business-Regeln getestet, und somit brauchen die Business-Regeln nicht mit der GUI verbunden zu sein. Also ist es eine gute Idee, GUI und Business-Regeln zu entkoppeln und Letztere durch Stubs zu ersetzen, während man die GUI selbst testet.

Halten Sie die GUI-Tests möglichst minimal. Sie sind fragil, weil die GUI unbeständig ist. Je mehr GUI-Tests Sie haben, desto unwahrscheinlicher wird es, dass Sie sie behalten können.

7.2.10 Andauernde Integration

Achten Sie darauf, dass alle Ihre Unit- und Akzeptanztests mehrere Male täglich in einem *andauernden Integrationssystem* durchlaufen werden. Dieses System sollte durch Ihr Quellcode-Kontrollsystem getriggert werden. Sobald jemand ein Modul committet, sollte dieses CI-System einen Build anstoßen und dann alle Tests im System durchführen. Die Resultate dieses Durchlaufs sollten dann per E-Mail alle im Team erhalten.

Alle Systeme stopp!

Es ist sehr wichtig, dass die CI-Tests jederzeit laufen. Sie sollten nie scheitern. Wenn sie scheitern, sollte das ganze Team die Arbeit unterbrechen und sich darauf konzentrieren, den gescheiterten Test wieder lauffähig zu kriegen. Ein kaputter Build im CI-System sollte als Notfall betrachtet werden, als ein »Alle Systeme stopp«-Event.

Ich war Berater bei Teams, die gescheiterte Tests nicht ernst genug nahmen. Sie waren »zu beschäftigt«, um die kaputten Tests zu beheben, schoben sie beiseite und versprachen, sie später zu fixen. In einem speziellen Fall hat das Team die kaputten Tests sogar aus dem Build entfernt, weil es so unbequem war, sie scheitern zu sehen. Später erkannten sie nach Freigabe an den Kunden, dass sie vergessen hatten, diese Tests wieder in den Build zu integrieren. Das kam heraus, weil ein wütender Kunde wegen seiner Bug-Reports anrief.

7.3 Schlussfolgerung

Kommunikation über Details ist schwierig. Das gilt vor allem für Programmierer und Stakeholder, die über die Details einer Applikation kommunizieren. Es ist für jede Seite viel leichter, einfach abzuwinken und *anzunehmen*, dass die andere verstanden hat. Nur zu oft gehen beide Seiten von der Ansicht aus, man verstehe sich schon, aber trennt sich dann mit völlig unterschiedlichen Vorstellungen voneinander.

Der einzige Weg, um effektiv die Kommunikationsfehler zwischen Programmierern und Stakeholdern zu eliminieren, besteht meines Wissens darin, automatisierte Akzeptanztests zu schreiben. Diese Tests sind derart formal, dass sie ausführbar sind. Sie sind völlig unzweideutig und harmonieren immer mit der Applikation. Sie sind die perfekten Anforderungsdokumente.



Professionelle Entwickler testen ihren Code. Aber fürs Testen reicht es nicht, einfach ein paar Unit-Tests oder Akzeptanztests zu schreiben. Natürlich ist es ganz prima, solche Tests zu schreiben, aber das reicht bei Weitem nicht aus. Was jedes professionelle Entwicklungsteam hingegen braucht, ist eine gute *Teststrategie*.

1989 arbeitete ich bei Rational am ersten Release von Rose. Etwa einmal pro Monat rief unser Manager für die Qualitätssicherung einen »Tag der Bug-Jagd« aus. Jeder im Team, egal ob Programmierer, Manager oder Sekretärin bis hin zu den Datenbankadministratoren, machte sich daran, Rose auf irgendeine Weise zum Scheitern zu bringen. Für die verschiedenen Arten von Bugs gab es jeweils Preise. Wer einen wirklich gravierenden Bug fand, konnte ein Abendessen zu zweit gewinnen. Wer die meisten Bugs gefunden hatte, durfte sich auf ein Wochenende im mexikanischen Monterrey freuen.

8.1 Die Qualitätssicherung sollte keine Fehler finden

Ich habe es bereits gesagt und wiederhole hier: Trotz der Tatsache, dass es in Ihrer Firma vielleicht eine Qualitätssicherungsgruppe gibt, die extra für die Tests von Software zuständig ist, sollte es das Ziel der Entwicklungsgruppe sein, der Qualitätssicherung keine Fehler übrig zu lassen.

Natürlich ist es eher unwahrscheinlich, dass dieses Ziel dauerhaft durchgehalten wird. Denn wenn Sie eine Gruppe intelligenter Leute haben, die es entschlossen darauf anlegt, alle Macken und Defizite in einem Produkt aufzuspüren, werden sie wahrscheinlich auch fündig. Doch jedes Mal, wenn die Qualitätssicherung einen Treffer gelandet hat, sollte das Entwicklerteam mit Furcht und Schrecken reagieren. Man sollte sich dort fragen, wie das nur geschehen konnte, und Schritte einleiten, um so etwas zukünftig zu verhindern.

8.1.1 Die Qualitätssicherung gehört zum Team

Durch den vorigen Abschnitt könnte man annehmen, dass Qualitätssicherung und Entwicklungsabteilung immer miteinander über Kreuz liegen und eine eher feindliche Beziehung haben. Das ist nicht gemeint. Die beiden Gruppen sollten vielmehr gemeinsam daran arbeiten, die Qualität des Systems zu garantieren. Die beste Rolle für den Qualitätssicherungsteil des Teams ist, für Spezifikationen und Charakterisierungen zu sorgen.

Qualitätssicherung und Spezifikationen

Es sollte zur Rolle der Qualitätssicherung gehören, gemeinsam mit dem Business die automatisierten Akzeptanztests zu schaffen, die zu den wahren Spezifikations- und Anforderungsdokumenten für das System führen. Von einer Iteration zur nächsten sammeln sie die Anforderungen vom Business und übersetzen sie in Tests, die den Entwicklern beschreiben, wie sich das System verhalten soll (siehe Kapitel 7, Akzeptanztests). Im Allgemeinen schreibt das Business die Tests für den Standardfall, während die Qualitätssicherung sich um die Grenzbedingungen, Ausnahmen und Fehlerfälle kümmert.

Qualitätssicherung und Charakterisierung

Die andere Rolle der Qualitätssicherung besteht darin, anhand der Disziplin des explorativen Testens¹ das wahre Verhalten des laufenden Systems zu charakterisieren und dieses Verhalten an Entwicklungsabteilung und Business zurückzumelden. In dieser Rolle interpretiert die Qualitätssicherung *nicht* die Anforderungen. Sie identifizieren vielmehr das eigentliche Verhalten des Systems.

¹ http://www.satisfice.com/articles/what_is_et.shtml

8.2 Die Pyramide der Testautomatisierung

Professionelle Entwickler setzen die Disziplin des Test Driven Developments ein, um Unit-Tests zu schaffen. Professionelle Entwicklerteams nutzen Akzeptanztests, um ihr System zu spezifizieren, und die andauernde Integration (siehe Kapitel 7), um Rückschritte zu verhindern. Aber diese Tests sind nur die halbe Miete. So schön es ist, eine Suite mit Unit- und Akzeptanztests an der Hand zu haben, aber wir brauchen auch Tests auf höherer Ebene, damit gewährleistet bleibt, dass die Qualitätssicherung nicht fündig wird. Die Pyramide der Testautomatisierung² in Abbildung 8.1 stellt grafisch dar, welche Arten von Tests eine professionelle Entwicklungsorganisation braucht.

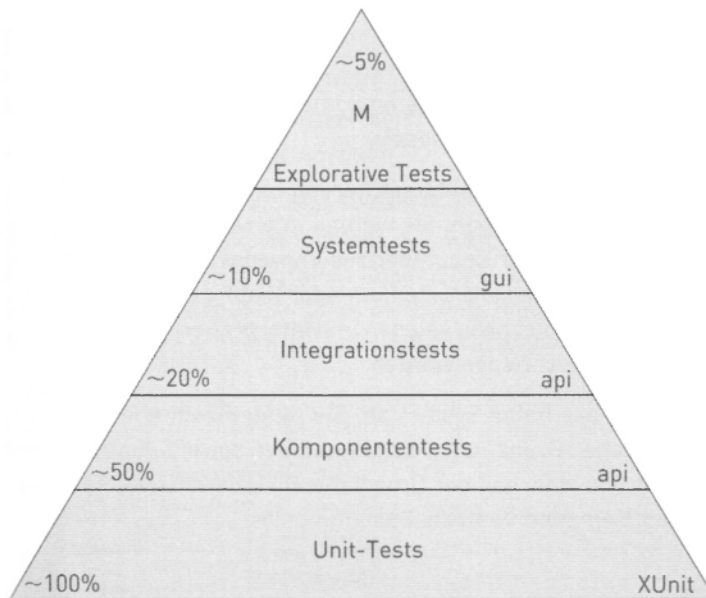


Abbildung 8.1: Die Pyramide der Testautomatisierung

8.2.1 Unit-Tests

Am Fuße der Pyramide stehen die Unit-Tests. Diese Tests werden von Programmierern für Programmierer in der Programmiersprache des Systems geschrieben. Diese Tests sollen das System auf unterster Ebene spezifizieren. Entwickler verfassen diese Tests, bevor sie sich an den Produktionscode machen, und geben damit an, was sie programmieren werden. Diese Tests werden als Teil der andauernden Integration ausgeführt, um sicherzustellen, dass die Intentionen des Programmierers gewahrt bleiben.

² [COHN09] S. 311–312

Unit-Tests bieten eine möglichst hundertprozentige Abdeckung, was noch praktikabel ist. Generell sollte dieser Wert irgendwo im Bereich über 90 % liegen. Und es sollte um *echte* Abdeckung gehen – im Gegensatz zu falschen Tests, die Code ausführen, ohne dessen Verhalten festzustellen.

8.2.2 Komponententests

Dies sind einige der im vorigen Kapitel erwähnten Akzeptanztests. Generell werden diese anhand individueller Komponenten des Systems geschrieben. Die Komponenten des Systems kapseln die Business-Regeln, und somit handelt es sich bei den Tests für die Komponenten um Akzeptanztests dieser Business-Regeln.

Wie in Abbildung 8.2 dargestellt, hüllt ein Komponententest eine Komponente ein. Er übergibt der Komponente Input-Daten und bekommt von ihr Output-Daten. Er testet, ob der Output zum Input passt. Alle anderen Systemkomponenten werden vom Test mittels passender Mocks und Testdoubles entkoppelt.

Die Komponententests werden durch die Qualitätssicherung und Business mit Unterstützung der Entwicklerabteilung geschrieben. Sie werden in einer Testumgebung für Komponenten wie FitNesse, JBehave oder Cucumber zusammengestellt (GUI-Komponenten werden mit GUI-Testumgebungen wie Selenium oder Watir getestet). Der Hintergedanke ist, dass diese Tests vom Business gelesen und interpretiert werden sollen, wenn sie nicht gar vom Business geschrieben werden.

Komponententests decken etwa das halbe System ab. Sie richten sich eher an Standardfällen und sehr offensichtlichen Grenz- und Ausnahmefällen sowie Situationen mit alternativen Pfaden aus. Die große Mehrheit der Grenzfällttests wird von Unit-Tests abgedeckt und ist auf Ebene der Komponententests bedeutungslos.

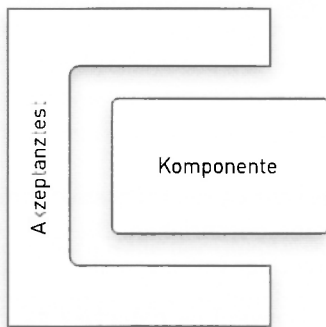


Abbildung 8.2: Akzeptanztest für Komponenten

8.2.3 Integrationstests

Diese Tests sind nur bei größeren Systemen sinnvoll, die aus vielen Komponenten bestehen. Wie in Abbildung 8.3 gezeigt, stellen diese Tests Komponentengruppen zusammen und testen, wie gut diese miteinander kommunizieren. Die anderen Komponenten des Systems sind mit entsprechenden Mocks und Testdoubles gewöhnlich entkoppelt.

Integrationstests sind sozusagen *Choreografie*-Tests. Sie testen keine Business-Regeln, sondern vielmehr, wie gut die Zusammenstellung der Komponenten miteinander »tanzen« kann. Es handelt sich um »Verdrahtungstests«, die gewährleisten sollen, dass die Komponenten richtig miteinander verbunden sind und klar miteinander kommunizieren können.

Integrationstests werden üblicherweise von den Systemarchitekten oder den leitenden Designern des Systems geschrieben. Die Tests sorgen dafür, dass die architektonische Struktur des Systems fehlerfrei bleibt. Genau auf dieser Ebene könnten wir auf Performance- und Durchsatztests treffen.

Integrationstests werden üblicherweise in der gleichen Sprache und Umgebung geschrieben wie Komponententests. Sie werden normalerweise *nicht* als Teil der Suite für die andauernde Integration ausgeführt, weil sie oft längere Laufzeiten aufweisen. Stattdessen werden diese Tests periodisch (nächtlich, wöchentlich etc.) durchgeführt, wie es von ihren Autoren als notwendig erachtet wird.

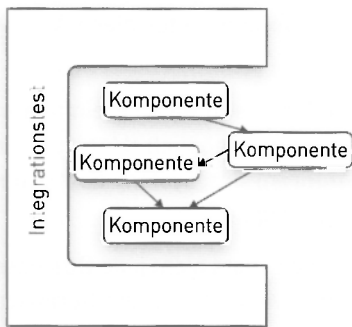


Abbildung 8.3: Integrationstest

8.2.4 Systemtests

Diese automatisierten Tests werden gegen das gesamte integrierte System durchgeführt. Es handelt sich um die ultimativen Integrationstests. Sie testen Business-Regeln nicht direkt, sondern vielmehr, ob das System korrekt verschaltet ist und dessen Bestandteile planmäßig miteinander interagieren. In dieser Suite kann man Durchsatz- und Performancetests erwarten.

Diese Tests werden von den Systemarchitekten und der technischen Leitung geschrieben. Üblicherweise sind sie in der gleichen Sprache und Umgebung verfasst wie die Integrationstests für das UI. Sie werden abhängig von ihrer Dauer relativ unregelmäßig durchgeführt, aber je häufiger, desto besser.

Systemtests decken etwa 10 % des Systems ab. Das liegt daran, dass es nicht ihre Aufgabe ist, für ein korrektes Systemverhalten, sondern für eine korrekte *Systemkonstruktion* zu sorgen. Das korrekte Verhalten des zugrunde liegenden Codes und seiner Komponenten wurde bereits in den unteren Ebenen der Pyramide ermittelt.

8.2.5 Manuelle explorative Tests

Hier legen Menschen ihre Finger auf die Tastatur und schauen auf den Bildschirm. Diese Tests sind weder automatisiert *noch sind sie geskriptet*. Diese Tests untersuchen das System auf unerwartetes Verhalten, während erwartetes Verhalten gleichzeitig bestätigt wird. An dieser Stelle benötigen wir menschliche Gehirne mit menschlicher Kreativität, die das System untersuchen und erforschen. Wenn man für diese Art Test einen Testplan schreibt, untergräbt man seinen Zweck.

Manche Teams haben für diese Arbeit Spezialisten. Andere nehmen sich einfach ein oder zwei Tage für die »Bug-Jagd«, bei der so viele Leute wie möglich (Manager, Sekretärinnen, Programmierer, Tester und technische Autoren) das System derart hart »rannehmen«, dass sie herausfinden, ob etwas daran kaputtgeht.

Das Ziel liegt nicht in der Abdeckung. Wir nehmen mit diesen Tests nicht jede Business-Regel und jeden Ausführungspfad unter die Lupe. Es geht vielmehr darum herauszufinden, ob sich das System bei der Bedienung durch Menschen gut verhält, und wir wollen kreativ so viele Macken wie möglich finden.

8.3 Schlussfolgerung

TDD ist eine sehr leistungsfähige Disziplin, und Akzeptanztests sind ein wertvoller Weg, um Anforderungen zu formulieren und durchzusetzen. Aber sie sind nur Teil einer umfassenderen Teststrategie. Um das Ziel »Die Qualitätssicherung darf nichts finden« wirklich gut umzusetzen, müssen Entwicklerteams gemeinsam mit der Qualitätssicherung an hierarchisch gestaffelten Unit-, Komponenten-, Integrations-, System- und explorativen Tests arbeiten. Diese Tests sollten so häufig wie möglich durchgeführt werden, um für ein maximales Feedback zu sorgen und sicherzustellen, dass das System dauerhaft sauber bleibt.

8.4 Bibliografie

[COHN09]: Mike Cohn, *Succeeding with Agile*, Boston, MA: Addison-Wesley, 2009, vgl. auch: Mike Cohn, *Agile Softwareentwicklung*, ISBN: 978-3-8273-2987-5, Addison-Wesley, 2010



Acht Stunden sind ein bemerkenswert kurzer Zeitraum. Das sind nur 480 Minuten oder 28.800 Sekunden. Als Profi sollten Sie diese wenigen wertvollen Sekunden so effizient und wirksam wie möglich nutzen. Mit welcher Strategie können Sie sicher davon ausgehen, dass die wenige, Ihnen zur Verfügung stehende Zeit nicht verschwendet wird? Wie können Sie Ihre Zeit effektiv managen?

1986 lebte ich in Little Sandhurst in Surrey, England. Ich organisierte für Teradyne in Bracknell eine Software-Entwicklungsabteilung mit 15 Mitarbeitern. Meine Tage waren hektisch und voller Telefonate, improvisierter Meetings, Kundendienstprobleme und Unterbrechungen. Um nun überhaupt meine Arbeit erledigen zu können, hatte ich mir ein paar sehr drastische Vorkehrungen zum Zeitmanagement angeeignet:

- Um fünf Uhr früh ging mein Wecker, und um 6 fuhr ich mit dem Rad zum Büro nach Bracknell. So hatte ich zweieinhalb Stunden Ruhe, bevor das tägliche Chaos ausbrach.
- Beim Eintreffen schrieb ich einen Zeitplan an meine Tafel. Ich teilte die Zeit in 15-minütige Abschnitte und schrieb alle Aktivitäten dort hinein, an denen ich im jeweiligen Zeitfenster arbeiten wollte.
- Ich füllte die ersten drei Stunden dieses Plans vollständig aus. Ab 9 Uhr ließ ich pro Stunde eine Lücke von 15 Minuten. Auf diese Weise konnte ich die meisten Unterbrechungen in einen dieser offenen Zeitslots schieben und mit der Arbeit fortfahren.
- Die Zeit nach dem Mittagessen ließ ich ungeplant, denn ich wusste, dass ab da die Hölle los ist und ich den restlichen Tag nur noch auf alles reagieren würde. Wenn es mal eine jener seltenen Nachmittagsperioden gab, in denen es keine Tumulte gab, arbeitete ich an der wichtigsten Sache, bis das Chaos wieder einkehrte.

Dieser Plan war nicht immer erfolgreich. Ich schaffte es nicht immer, um 5 Uhr aufzustehen, und manchmal dominierte das Chaos alle meine sorgfältig ausbaldowerten Strategien und fraß meinen Tag auf. Aber meist behielt ich den Kopf oben.

9.1 Meetings

Meetings kosten etwa 200 Dollar pro Stunde und Teilnehmer. Dabei sind Gehälter, Zulagen, Gebäudekosten usw. mit eingerechnet. Wenn Sie das nächste Mal in einem Meeting sitzen, kalkulieren Sie mal die Kosten. Da könnten Sie überrascht sein.

Es gibt zwei Wahrheiten über Meetings.

1. Meetings sind notwendig.
2. Meetings sind eine Riesenzeitverschwendung.

Oft beschreiben diese beiden Wahrheiten dasselbe Meeting in gleichem Maße. Manche der Teilnehmer finden es unschätzbar wertvoll, andere halten es für redundant oder nutzlos.

Profis sind sich der hohen Kosten von Meetings bewusst. Ihnen ist klar, dass auch ihre eigene Zeit kostbar ist: Sie müssen Code schreiben und Zeitpläne einhalten. Somit leisten sie aktiv Widerstand gegen Meetings, die keinen direkten oder wesentlichen Vorteil bringen.

9.1.1 Absagen

Sie müssen nicht an jedem Meeting teilnehmen, zu dem Sie eingeladen werden. Tatsächlich ist es unprofessionell, bei zu vielen Meetings anwesend zu sein. Sie müssen Ihre Zeit klug nutzen. Also achten Sie sehr sorgfältig darauf, bei welchen Sie mitmachen und welche Sie höflich absagen.

Die Person, die Sie für ein Meeting einlädt, ist nicht dafür verantwortlich, wie Sie Ihre Zeit managen. Das können nur Sie selbst! Wenn Sie also eine Einladung bekommen, akzeptieren Sie nur dann, falls es sich um ein Meeting handelt, bei dem Ihre Anwesenheit direkt erforderlich und für den Job, den Sie jetzt ausführen, wesentlich ist.

Manchmal geht's bei dem Meeting um etwas, was Sie interessiert, aber nicht direkt notwendig ist. Sie müssen sich entscheiden, ob Sie die Zeit aufbringen wollen. Passen Sie gut auf: Solche Meetings, die Ihre Zeit auffressen, könnte es mehr als genug geben.

Manchmal geht es im Meeting um ein Thema, bei dem Sie etwas beitragen können, aber was nicht direkt für Ihre aktuelle Tätigkeit bedeutsam ist. Sie werden sich entscheiden müssen, ob der Verlust für *Ihr* Projekt den Vorteil fürs andere wert ist. Das mag zynisch klingen, aber in Ihrer Verantwortung kommen *Ihre* Projekte zuerst. Doch ist es oft gut, wenn Teams sich gegenseitig helfen. Also sollten Sie die Teilnahme mit Ihrem Team und Vorgesetzten absprechen.

Manchmal wird Ihre Anwesenheit von einer Autoritätsperson angefordert, z.B. von einem sehr wichtigen erfahrenen Engineer oder dem Manager eines anderen Projekts. Sie werden sich entscheiden müssen, ob dessen Autorität Ihren Arbeitsplan aufwiegt. Wieder können Ihr Team und Ihr Vorgesetzter bei der Entscheidung hilfreich sein.

Eine der wichtigsten Pflichten Ihres Vorgesetzten ist, Ihnen Meetings zu *ersparen*. Ein guter Vorgesetzter ist überaus gewillt, Ihre Entscheidung gegen eine Teilnahme zu verteidigen, weil er genauso um Ihre Zeit besorgt ist wie Sie.

9.1.2 Sich ausklinken

Meetings laufen nicht immer wie geplant ab. Manchmal merken Sie erst mitten in einem Meeting, dass Sie eigentlich abgesagt hätten, wenn Sie besser informiert gewesen wären. Manchmal werden neue TOPs aufgenommen, oder der Lieblingsaufreger eines Anwesenden dominiert die Diskussion. Im Laufe der Jahre habe ich eine einfache Regel entwickelt: Wenn das Meeting langweilig wird, verschwinde ich.

Noch einmal: Sie verantworten selbst, Ihre Zeit gut zu verwalten. Wenn Sie merken, dass Sie in einem Meeting Ihre Zeit verschwenden, müssen Sie einen Weg finden, dieses Meeting auf höfliche Weise zu verlassen.

Natürlich sollten Sie nicht mit dem Ausruf »Was für'n langweiliger Kram!« aus der Sitzung stürmen. Kein Grund, unhöflich zu werden. Sie können sich bei passender Gelegenheit einfach erkundigen, ob Ihre Anwesenheit immer noch notwendig ist. Sie erklären, dass Sie nicht mehr Zeit aufbringen können, und fragen, ob es eine Möglichkeit gibt, die Diskussion voranzutreiben oder die Tagesordnung umzustellen.

Hier besteht die wichtige Erkenntnis darin, dass der Verbleib in einem Meeting, das für Sie zur Zeitverschwendung wird und zu dem Sie nichts Wesentliches beitragen können, unprofessionell ist. Sie haben die Pflicht, Zeit und Geld Ihres Arbeitgebers wohlbedacht zu verwenden. Also ist es nicht unprofessionell, einen passenden Moment zu wählen, um den eigenen Abgang zu verhandeln.

9.1.3 Tagesordnung und Ziel

Wir dulden deswegen die Kosten eines Meetings, weil wir manchmal *wirklich alle* Teilnehmer an einem Tisch brauchen, um ein bestimmtes Ziel zu erreichen. Damit die Zeit aller Anwesenden klug genutzt wird, sollte das Meeting eine klare Agenda und einen Zeitrahmen für jedes Thema sowie ein festgelegtes Ziel haben.

Wenn Sie zu einem Meeting gebeten werden, sollten Sie darauf achten, die anstehenden Punkte zu kennen und zu wissen, wie viel Zeit jeweils für Diskussionen zugewiesen wurde und welche Ziele damit erreicht werden sollen. Wenn Sie keine klare Antwort auf diese Fragen bekommen, sollten Sie eine Teilnahme höflich ablehnen.

Wenn Sie zu einem Meeting gehen und merken, dass jemand die Tagesordnung gekapert oder über Bord geworfen hat, sollten Sie den Antrag stellen, dass das neue Thema in die Folge der TOPs eingebracht wird und der Agenda weiter gefolgt werden sollte. Falls das nicht geschieht, sollten Sie wenn möglich höflich den Raum verlassen.

9.1.4 Stand-up-Meetings

Diese Meetings gehören zum agilen Instrumentarium. Der Name rührt von der Tatsache her, dass die Teilnehmer während des gesamten Meetings stehen sollen. Jeder Teilnehmer beantwortet reihum diese drei Fragen:

1. Was habe ich gestern gemacht?
2. Was werde ich heute machen?
3. Was behindert mich bei meiner Arbeit?

Das ist alles. Jede Antwort sollte *nicht mehr* als 20 Sekunden dauern, und somit spricht jeder Teilnehmer nicht länger als eine Minute. Auch in einer Gruppe mit zehn Leuten sollte das Meeting vorbei sein, bevor zehn Minuten vergangen sind.

9.1.5 Planungstreffen zur Iteration

Diese Treffen gehören zu den Meetings im agilen Kanon, die am schwierigsten gut umzusetzen sind. Werden sie schlecht durchgeführt, brauchen sie viel zu viel Zeit. Es braucht Geschick, diese Meetings gut zu gestalten, und es lohnt, sich das anzueignen.

Bei Meetings zur Iterationsplanung sollen die Backlog-Items ausgewählt werden, die in der nächsten Iteration auszuführen sind. Wenn Kandidaten auf diese Liste kommen, sollten dafür bereits Kalkulationen vorliegen. Einschätzungen über den Business-Wert sollten bereits feststehen. In wirklich guten Organisationen sind die Akzeptanz- und Komponententests schon geschrieben oder zumindest skizziert.

Das Meeting sollte schnell vorangehen, und jeder Kandidat aus dem Backlog sollte kurz diskutiert und dann entweder gewählt oder abgelehnt werden. Für jedes Item sollten nicht mehr als fünf oder zehn Minuten aufgewendet werden. Falls längere Diskussionen erforderlich sind, sollten sie auf eine andere Gelegenheit mit einer Untergruppe des Teams verschoben werden.

Meine Faustregel lautet, dass das Meeting nicht länger als 5 % der Gesamtzeit der Iteration dauert. Also sollte bei einer Iteration von einer Woche (40 Stunden) das Meeting innerhalb von zwei Stunden beendet sein.

9.1.6 Retrospektive und Demo der Iteration

Diese Meetings werden am Ende jeder Iteration durchgeführt. Das Team diskutiert, was gut und was schlecht gelaufen ist. Die Stakeholder sehen eine Demo der neu funktionierenden Features. Diese Meetings können schlimm missbraucht werden und eine Menge Zeit fressen. Also planen Sie sie 45 Minuten vor Feierabend am letzten Tag der Iteration. Reservieren Sie nicht mehr als 20 Minuten für die Retrospektive und 25 Minuten für die Demo. Denken Sie daran, dass die letzte erst eine oder zwei Wochen her ist, also sollte es nicht sonderlich viel zu besprechen geben.

9.1.7 Auseinandersetzungen und Meinungsverschiedenheiten

Kent Beck hat mir mal etwas ganz Wesentliches gesagt: »Jede Debatte, die nicht in fünf Minuten beigelegt werden kann, kann nicht durch Debattieren gelöst werden.« Die Auseinandersetzung hält deswegen lange an, weil es keine klaren Beweise gibt, die eine der Seiten stützen. Da werden dann persönliche Vorlieben mit religiöser Inbrunst begründet und nicht mit Fakten.

Technische Meinungsverschiedenheiten neigen dazu, in Richtung Mond abzuheben. Jede Partei hat alle möglichen Rechtfertigungen für ihre Position, aber selten irgendwelche Daten. Ohne Daten und Belege wird jede Streiterei, die nicht innerhalb kurzer Zeit (etwa fünf bis dreißig Minuten) zu einer Einigung gelangt, niemals zu einer Übereinkunft führen. Hier bleibt einzig und allein, sich Daten und Fakten zu besorgen.

Manche werden versuchen, per Durchsetzungskraft eine Auseinandersetzung zu gewinnen. Sie werden persönlich oder laut oder geben sich herablassend. Aber das ist egal: Eine mit Durchsetzungskraft erzwungene Einigung wird nicht lange vorhalten. Nur wenn sie mit Daten belegt wird.

Einige Leute handeln passiv aggressiv. Sie stimmen zu, einfach um die Streiterei zu beenden, und sabotieren anschließend das Ergebnis, indem sie sich weigern, sich für die Lösung zu engagieren. Sie sagen sich: »Die wollten doch, dass das so endet, und jetzt kriegen sie eben, was sie wollen.« Das ist wahrscheinlich das schlimmstmögliche unprofessionelle Verhalten. Machen Sie so etwas niemals! Wenn Sie zustimmen, *müssen* Sie daran mitarbeiten.

Wie bekommen Sie die Fakten, um eine Auseinandersetzung beizulegen? Manchmal führt man dazu Experimente durch oder nimmt eine Simulation oder Modellierung vor. Aber gelegentlich ist die beste Alternative, einfach eine Münze zu werfen, um einen der beiden möglichen Wege zu beschreiten. Wenn alles klappt, war dieser Weg gangbar. Wenn Sie auf Probleme stoßen, stoppen Sie und nehmen den anderen Pfad. Es ist klug, sich über den Zeitrahmen und auch die Kriterien abzustimmen, um festlegen zu können, wann man den gewählten Pfad wieder verlässt.

Vorsicht vor Meetings, die eigentlich nur der Austragungsort sind, an dem ein Streit ventiliert wird und um Support für die eine oder andere Seite zu sammeln. Und vermeiden Sie jene, wo nur einer der Argumentierenden präsentiert.

Wenn ein Streit wirklich beigelegt werden muss, fordern Sie alle Argumentierenden auf, dem Team ihren Fall in fünf oder weniger Minuten zu präsentieren. Dann lassen Sie das Team abstimmen. Das ganze Meeting wird weniger als eine Viertelstunde dauern.

9.2 Fokus-Manna

Vergeben Sie mir, wenn dieser Abschnitt scheinbar nach metaphysischem New Age duftet oder gar nach Dungeons & Dragons riecht. Ich kann mir nicht helfen – ich denke einfach so über dieses Thema.

Programmieren ist eine intellektuelle Übung, in der über längere Zeitabschnitte Konzentration und Fokus erforderlich sind. Fokus ist eine seltene Ressource – ähnlich wie

Manna¹. Nachdem Sie Ihr Fokus-Manna verbraucht haben, müssen Sie es wieder aufladen, indem Sie etwa eine Stunde oder mehr unfokussierte Aktivitäten durchführen.

Ich weiß nicht genau, worum es sich bei diesem Fokus-Manna handelt, aber ich spüre, dass es dabei um eine materielle Substanz (oder deren Fehlen) geht, die sich auf Wachsamkeit und Aufmerksamkeit auswirkt. Egal wie – Sie können jedenfalls fühlen, wenn sie vorhanden ist, und auch, wenn sie fehlt. Professionelle Entwickler lernen, ihre Zeit so zu managen, dass sie ihr Fokus-Manna zu ihrem Vorteil nutzen. Wir schreiben Code, wenn unser Fokus-Manna voll ist, und wenn es leer ist, machen wir andere, weniger produktive Dinge.

Fokus-Manna ist außerdem eine zerfallende Ressource. Wenn Sie es nicht nutzen, solange es vorhanden ist, werden Sie es wahrscheinlich verlieren. Das ist einer der Gründe, warum Meetings so verheerend sein können. Wenn Sie all Ihr Fokus-Manna in einem Meeting verplempern, bleibt fürs Coding nichts mehr übrig.

Sorgen und Ablenkungen zehren ebenfalls am Fokus-Manna. Der Streit, den Sie gestern Abend mit Ihrem Ehepartner hatten, die Macke, die Sie heute Morgen in die Stoßstange gefahren haben, oder die Rechnung, die Sie letzte Woche zu bezahlen vergaßen, all das saugt schnell das Fokus-Manna ab.

9.2.1 Schlaf

Diesen Punkt kann ich gar nicht genug betonen. Ich habe das meiste Fokus-Manna, wenn ich so richtig gut geschlafen habe. Sieben Stunden Schlaf geben mir oft acht Stunden lang wertvolles Fokus-Manna. Professionelle Entwickler managen ihren Zeitplan fürs Schlafen, um sicher zu sein, dass ihr Fokus-Manna wieder aufgefüllt ist, wenn sie sich morgens auf den Weg zur Arbeit machen.

9.2.2 Koffein

Zweifellos können einige von uns das Fokus-Manna effektiver nutzen, wenn sie moderate Mengen Koffein zu sich nehmen. Aber seien Sie vorsichtig: Koffein lässt den Fokus auch gewissermaßen »flattern«. Zu viel Koffein schickt Ihren Fokus in sehr eigenartige Richtungen. Ein wirklich starker Kaffee-Flash kann dafür sorgen, dass Sie einen ganzen Tag damit verschwenden, sich hyperintensiv auf alle möglichen verkehrten Sachen zu konzentrieren.

¹ Manna ist ein üblicher Handelsartikel in Fantasy- und Rollenspielen wie Dungeons & Dragons. Jeder Spieler bekommt eine bestimmte Menge Manna, und sobald ein Spieler einen Zauberspruch ausführt, verringert sich diese magische Substanz. Je wirkungsvoller der Zauberspruch ist, desto mehr Manna verbraucht der Spieler dafür. Manna wird in einer festgelegten täglichen Dosis langsam wieder aufgeladen. Also kann leicht alles durch ein paar Zaubersprüche aufgebraucht werden.

Koffeinkonsum und dessen Toleranz sind sehr individuell. Meine persönliche Vorliebe sind eine Tasse starker Kaffee morgens und eine Diät-Coke zum Mittagessen. Manchmal verdoppele ich diese Dosis, aber mehr nehme ich selten zu mir.

9.2.3 Die Akkus aufladen

Fokus-Manna kann teilweise durch eine Art »Ent-Fokussierung« wieder aufgeladen werden. Ein schöner, langer Spaziergang, ein Gespräch mit Freunden oder einfach ein längerer Blick aus dem Fenster können helfen, das Fokus-Manna wieder aufzuladen.

Einige meditieren auch. Andere legen sich kurz für einen Power-Nap ab. Wieder andere hören einen Podcast oder blättern eine Zeitschrift durch.

Aus eigener Erfahrung kann ich sagen, wenn das Manna erst einmal verschwunden ist, kann man den Fokus nicht mehr erzwingen. Man kann immer noch Code schreiben, den muss man aber mit ziemlicher Sicherheit am nächsten Tag neu schreiben oder Wochen oder gar Monate mit dieser verrottenden Masse leben. Also ist es besser, sich eine halbe oder besser eine ganze Stunde zum »Ent-Fokussieren« zu nehmen.

9.2.4 Muskelfokus

Dem Training solch körperlicher Disziplinen wie Kampfsport, Tai Chi oder Yoga ist etwas ganz Besonderes eigen: Obwohl diese Aktivitäten eine recht spezielle Konzentration erfordern, ist das eine andere Art Fokus als beim Programmieren: nicht intellektuell, sondern muskulär. Und irgendwie hilft dieser »Muskelfokus« dabei, den mentalen Fokus wieder aufzuladen. Es ist allerdings mehr als ein einfaches Aufladen der Akkus. Ich merke, dass eine regelmäßige Dosis Konzentration auf eine muskuläre Betätigung meine Kapazität für den mentalen Fokus steigert.

Meine Lieblingsform des körperlichen Fokussierens ist Radfahren. Ich radele eine Stunde oder zwei und lege dabei manchmal 30 oder 40 Kilometer zurück. Ich fahre dabei auf einer Strecke, die parallel zum Des Plaines River verläuft, damit ich nicht auf Autos achten muss.

Beim Fahrradfahren höre ich Podcasts über Astronomie oder Politik. Manchmal lege ich meine Lieblingsmusik ein. Und manchmal nehme ich die Kopfhörer ab und lausche nur den Geräuschen der Natur.

Manche Leute nehmen sich die Zeit, praktisch mit den Händen zu werken. Vielleicht machen Ihnen Holzarbeiten Spaß oder Modellbau oder Gartenarbeit. Egal um welche Aktivität es geht – es gibt etwas an diesen Aktivitäten, wo Sie eher körperlich gefordert werden, das die Fähigkeit zur geistigen Arbeit verbessert.

9.2.5 Input vs. Output

Was ich außerdem für den Fokus wesentlich finde, ist das Ausbalancieren meines Outputs mit dem entsprechenden Input. Software zu schreiben, ist eine *kreative* Übung. Ich habe festgestellt, dass ich am kreativsten bin, wenn ich mich mit der Kreativität anderer beschäftige. Also lese ich eine Menge Science-Fiction. Die Schöpfungen dieser Autoren bringen sozusagen auch meine eigenen kreativen Säfte fürs Software-Schreiben ins Fließen.

9.3 Zeitfenster und Tomaten

Eine sehr effektive Übung, um meine Zeit und Konzentration zu managen, besteht in der bekannten *Pomodoro*-Technik². Die Grundidee ist sehr einfach: Sie setzen einen handelsüblichen Küchenwecker (traditionell oft wie eine Tomate geformt) auf 25 Minuten. Während dieser Timer läuft, lassen Sie sich durch *absolut nichts* bei dem stören, was Sie gerade machen. Wenn das Telefon läutet, heben Sie ab und fragen höflich, ob Sie innerhalb der nächsten 25 Minuten zurückrufen können. Wenn jemand vorbeikommt und eine Frage stellt, fragen Sie höflich, ob Sie darauf innerhalb der nächsten 25 Minuten zurückkommen können. Egal wodurch Sie unterbrochen werden, Sie stellen die Störung einfach so lange zurück, bis der Timer klingelt. Immerhin sind nur sehr wenige Störungen so furchtbar dringend, dass sie keine 25 Minuten warten können!

Wenn der Tomaten-Timer klingelt, unterbrechen Sie *sofort* das, was Sie gerade machen. Sie kümmern sich um alle Störungen, die während der »Tomatenzeit« vorgekommen sind. Dann gönnen Sie sich eine Pause von etwa fünf Minuten. Dann stellen Sie den Timer wieder auf 25 Minuten und fangen die nächste »Tomatenzeit« an. Nach jeder vierten »Tomatenzeit« machen Sie eine längere Pause von etwa 30 Minuten.

Über diese Technik ist bereits viel geschrieben worden, und ich rate Ihnen dringend, sich diese Lektüre zu Gemüte zu führen. Doch die obige Beschreibung liefert schon mal das Wesentliche dieser Technik.

Diese Technik teilt Ihre Zeit in eine »Tomatenzeit« und eine »Nicht-Tomatenzeit« ein. Die Erstere ist produktiv. Innerhalb dieser Zeit erledigen Sie die eigentliche Arbeit. Bei der anderen Zeit geht es entweder um Ablenkungen, Meetings, Pausen oder andere Zeitphasen, in denen Sie nicht an Ihren Aufgaben arbeiten.

Wie viele »Tomaten« schaffen Sie an einem Tag? An einem guten Tag sind das vielleicht 12 oder gar 14 Tomaten. An einem schlechten erledigen Sie vielleicht zwei oder drei. Wenn Sie das mal durchzählen und festhalten, bekommen Sie ziemlich schnell einen guten Eindruck, wie viel von Ihrem Tag Sie produktiv verbringen und wie viel mit allem möglichem »Kram«.

² <http://www.pomodoro-technique.com/>

Manchen ist diese Technik so in Fleisch und Blut übergegangen, dass sie ihre Aufgaben in »Tomaten« kalkulieren und dann ihre wöchentliche Entwicklungsgeschwindigkeit in »Tomaten« messen. Aber das ist nur das i-Tüpfelchen. Der wahre Vorteil der Pomodoro-Technik besteht in diesem 25-Minuten-Fenster produktiver Zeit, die Sie vehement gegen alle Störungen verteidigen.

9.4 Vermeidung

Manchmal ist Ihr Herz einfach nicht bei der Sache. Vielleicht sind die anstehenden Aufgaben irgendwie furchteinflößend, unbequem oder langweilig. Vielleicht glauben Sie, dass Sie zu einer Konfrontation gezwungen werden oder in einer Sackgasse landen. Oder vielleicht haben Sie schlicht und einfach keine Lust, es zu machen.

9.4.1 Umkehrung der Prioritäten

Egal mit welcher Begründung: Man findet immer Wege, um der eigentlichen Erledigung der Arbeit aus dem Weg zu gehen. Sie überzeugen sich selbst, etwas anderes sei viel dringender, und erledigen dann lieber das. Das bezeichnet man als *Prioritätsumkehrung*. Sie erhöhen die Priorität einer Aufgabe, damit Sie die andere verschieben können, die die wahre Priorität hat. Solche Prioritätsumkehrungen sind eine Lüge, die wir selbst uns erzählen. Wir stellen uns nicht dem, was erledigt werden muss, und überzeugen uns deswegen selbst davon, dass eine andere Aufgabe wichtiger ist. Wir wissen, dass dem nicht so ist, aber lügen uns damit selbst in die Tasche.

Tatsächlich lügen wir uns eigentlich nicht an. Wir bereiten uns vielmehr auf die Lüge vor, die wir erzählen werden, wenn jemand fragt, was wir machen und warum wir es machen. Wir bauen eine Verteidigung auf, um uns vor dem Urteil anderer zu schützen.

Das ist eindeutig unprofessionelles Verhalten. Profis beurteilen die Priorität jeder Aufgabe ungeachtet ihrer persönlichen Ängste und Wünsche und führen diese Aufgaben in der Reihenfolge der Prioritäten aus.

9.5 Sackgassen

Für alle Software-Entwickler gehören Sackgassen als Tatsache zum Leben. Manchmal trifft man eine Entscheidung und arbeitet sich einen technischen Pfad entlang, der einen aber nirgendwo hinführt. Je mehr Sie sich an Ihrer Entscheidung festbeißen, desto länger irren Sie in der Wildnis umher. Wenn Sie auch noch Ihre professionelle Reputation daran hängen, kommen Sie nie aus dieser Wildnis heraus.

Besonnenheit und Erfahrung helfen Ihnen dabei, bestimmte Sackgassen zu vermeiden, aber um alle werden Sie nicht herumkommen. Stattdessen brauchen Sie die Fähigkeit, schnell zu erkennen, ob Sie in einer Sackgasse stecken, und den Mut, wieder umzukehren. Das nennt man auch das *Gesetz des Loches*: Sitzt man in einem Loch drin, sollte man mit dem Graben aufhören.

Profis vermeiden es, sich so in eine Idee zu verbeißen, dass sie es nicht mehr schaffen, sich davon zu lösen und umzudrehen. Sie bleiben anderen Ideen gegenüber offen, damit sie noch andere Optionen haben, wenn es für sie nicht mehr weitergeht.

9.6 Morast, Moore, Sümpfe und andere Schlamassel

Schlimmer als eine Sackgasse ist ein Morast. Der bremst Sie aus, hält Sie aber nicht auf. Wenn Sie in einer solchen Patsche stecken, behindert das Ihr Vorankommen, aber rein durch rohe Kraft können Sie immer noch Fortschritte machen. Ein Morast ist immer schlimmer als eine Sackgasse, denn Sie können stets den Weg voraussehen, und der wirkt immer kürzer als der Weg zurück (doch das stimmt nicht).

Ich habe erlebt, wie Produkte ruiniert und Firmen zerstört wurden – nur wegen eines solchen Software-Morasts. Ich habe gesehen, wie die Produktivität von Teams in nur wenigen Monaten sank: Erst waren alle Nervenbündel, dann blieb nur noch die Totenklage. Nichts hat einen tiefer greifenden oder länger andauernden negativen Effekt auf die Produktivität eines Software-Teams als ein solcher Morast. Nichts.

Das Problem ist, dass es unvermeidbar ist, in einen Morast zu geraten – so wie man auch Sackgassen nicht generell vermeiden kann. Erfahrung und Besonnenheit helfen Ihnen dabei, so etwas zu vermeiden, aber schließlich werden Sie irgendwann auch mal eine Entscheidung treffen, die ins Chaos führt.

Der Verlauf eines solchen Schlamassels ist heimtückisch. Sie schaffen für ein einfaches Problem die Lösung und achten sorgfältig darauf, den Code einfach und aufgeräumt zu halten. Wenn das Problem dann weiter wächst und komplizierter wird, erweitern Sie diese Codebasis und halten sie so aufgeräumt wie möglich. An irgendeinem Punkt erkennen Sie, dass Sie anfänglich eine falsche Designentscheidung getroffen haben und dass Ihr Code in die Richtung, in die sich die Anforderungen entwickeln, nicht gut skaliert.

Dies ist der Wendepunkt! Sie können immer noch zurückgehen und das Design fixen. Aber Sie können auch fortfahren und weiter vorangehen. Zurückzugehen sieht kostspielig aus, weil Sie den vorhandenen Code überarbeiten müssen, aber es wird *niemals* mehr so einfach sein zurückzugehen wie jetzt. Wenn Sie nach vorne weitergehen, treiben Sie das System in einen Sumpf, aus dem es vielleicht nie wieder entkommen kann.

Profis fürchten solche Sümpfe weitaus mehr als Sackgassen. Sie passen immer auf, ob sie in etwas hineingeraten, was grenzenlos einfach weiterwächst, und nehmen alle nötigen Anstrengungen vor, um so früh und schnell wie möglich daraus zu entkommen.

Wenn man *weiß*, dass man sich durch einen Sumpf kämpft, und trotzdem damit weitermacht, ist das der schlimmste Fall einer Prioritätsumkehr. Indem Sie nach vorne weitergehen, belügen Sie sich selbst, Ihr Team, Ihre Firma und letzten Endes auch Ihre Kunden. Sie erzählen ihnen, dass alles gut laufen wird, während Sie in Wirklichkeit gemeinsam auf den Untergang zulaufen.

9.7 Schlussfolgerung

Software-Profis sind beim Management ihrer Zeit und ihrer Konzentration sehr besonnen. Sie erkennen die Versuchungen einer Prioritätsumkehr, und ihr Kampf dagegen ist für sie Ehrensache. Sie bewahren sich ihre offenen Optionen, indem sie Ausschau nach alternativen Lösungen halten. Sie verstricken sich nie so sehr in eine Lösung, dass sie sie nicht auch wieder über Bord werfen können. Und sie achten sehr sorgfältig darauf, wenn sie in einen wachsenden Sumpf geraten, und räumen damit auf, sobald sie einen erkennen. Es gibt keinen traurigeren Anblick als ein Team Software-Entwickler, das sich fruchtlos in einem immer tiefer werdenden Morast abkämpft.



Schätzungen gehören zu den einfachsten und doch furchteinflößendsten Aktivitäten, denen sich Software-Profis stellen müssen. Von ihnen hängt ein immenser Business-Wert ab. Unsere Reputation steht und fällt in hohem Maße damit. Sie sorgen in unserer Branche für viel Angst und Versagen. Sie sind ursprünglich der Keil, der zwischen Business und Entwickler getrieben wurde, und Quelle beinahe des gesamten Misstrauens, das diese Beziehung dominiert.

1978 war ich Hauptentwickler für ein eingebettetes 32K-Programm, das für einen Z-80 in Assembler geschrieben wurde. Das Programm wurde auf 32 1K x 8 EEPROM-Chips gebrannt. Diese 32 Chips wurden in drei Platinen eingesetzt, die jeweils zwölf Chips aufnehmen konnten.

Von diesen Geräten hatten wir überall in den USA Hunderte im Einsatz, die im ganzen Land in Telefonzentralen installiert waren. Immer wenn wir einen Bug gefixt oder ein Feature ergänzt hatten, mussten wir unsere Außendienstmitarbeiter zu jedem einzelnen Gerät schicken und alle 32 Chips ersetzen lassen!

Es war ein Alptraum. Die Chips und Platinen waren zerbrechlich. Die Pins auf den Chips konnten verbiegen und abknicken. Das dauernde Bewegen der Platinen konnte die Lötverbindungen beschädigen. Das Risiko für Brüche und Fehler war enorm. Die Kosten für die Firma waren viel zu hoch.

Mein Chef Ken Finder kam zu mir und beauftragte mich, für eine Lösung zu sorgen. Er wollte, dass es eine Möglichkeit gab, an einem Chip etwas zu verändern, ohne gleich auch all die anderen Chips tauschen zu müssen. Wenn Sie meine Bücher gelesen oder einen meiner Vorträge gehört haben, wissen Sie, dass unabhängiges Deployment eines meiner Lieblingsthemen ist. Hier habe ich diese Lektion zum ersten Mal gelernt.

Unser Problem bestand darin, dass die Software aus einer einzigen verknüpften und ausführbaren Datei bestand. Wenn das Programm eine neue Codezeile bekam, änderten sich die Adressen aller folgenden Codezeilen. Weil jeder Chip nur 1K des Adressraums enthielt, änderten sich die Inhalte praktisch aller anderen Chips.

Die Lösung war ziemlich einfach: Die Chips mussten alle voneinander entkoppelt werden. Jeder Chip musste in eine unabhängige Kompilierungseinheit verwandelt werden, die man unabhängig von den anderen brennen konnte.

Also maß ich die Größen aller Funktionen in der Applikation und schrieb ein einfaches Programm, das die Funktionen wie Puzzlesteine in jeden Chip einpasste und noch etwa 100 Bytes Platz für Erweiterungen übrig ließ. An den Anfang des Chips stellte ich eine Tabelle mit Zeigern auf alle Funktionen auf diesem Chip. Beim Booten wurden diese Zeiger ins RAM verschoben. Der gesamte Code des Systems wurde umgeschrieben, sodass die Funktionen nicht direkt, sondern nur durch diese RAM-Vektoren aufgerufen wurden.

Genau, Sie haben es schon erkannt: Die Chips waren Objekte mit virtuellen Funktionstabellen (*vtables*). Alle Funktionen wurden polymorph deployed. Und richtig, so erlernte ich einige Prinzipien des OOD, lange bevor ich wusste, was ein Objekt war.

Die Vorteile waren enorm. Wir konnten so nicht nur einzelne Chips ersetzen, sondern bei den eingesetzten Geräten auch Patches vornehmen, indem wir Funktionen ins RAM verschoben und die Adressen in den Vektoren änderten. Das vereinfachte Debugging und Hot Patches im Außendienst deutlich.

Aber ich schweife ab. Als Ken zu mir kam und mir den Auftrag gab, dieses Problem zu beheben, schlug er vor, das mit Zeigern auf Funktionen umzusetzen. Ich verbrachte ein, zwei Tage damit, diese Idee in Form zu bringen, und präsentierte ihm dann einen

detaillierten Plan. Er fragte mich, wie lange das dauern würde, und ich antwortete, dass ich dafür etwa einen Monat bräuchte.

Es dauerte *drei* Monate.

Ich war nur zwei Mal in meinem Leben betrunken und nur einmal *richtig* betrunken. Das war 1978 bei der Weihnachtsfeier bei Teradyne, und ich war 26.

Die Feier fand im Büro von Teradyne statt, einem größtenteils offenen Arbeitsbereich. Alle kamen recht früh, und dann gab es einen heftigen Schneesturm, der verhinderte, dass die Band und der Catering-Service kommen konnten. Zum Glück hatten wir schon eine Menge Alkohol organisiert.

Von dieser Nacht weiß ich nicht mehr viel. Und was ich noch erinnere, würde ich lieber vergessen. Aber einen ergreifenden Moment möchte ich mit Ihnen teilen.

Ich saß im Schneidersitz neben Ken (mein Chef war damals gerade mal 29 Jahre alt und *nicht* betrunken) und heulte ihm vor, wie lange ich für die Vektorisierungsarbeit bräuchte. Der Alkohol hatte meine Ängste und Unsicherheit über meine Kalkulationen unterdrückt. Ich glaube *nicht*, dass ich meinen Kopf auf seinen Schoß gelegt hatte, aber bei dieser Art von Detail streikt meine Erinnerung.

Ich weiß aber noch, dass ich ihn fragte, ob er sauer auf mich sei und ob er glaube, dass ich zu lange für die Arbeit bräuchte. Obwohl der restliche Abend im Nebel des Alkohols versank, blieb mir seine Antwort all die Jahrzehnte noch deutlich in Erinnerung. Er meinte: »Ja, ich finde, du brauchst echt lange dafür, aber ich sehe, dass du dich wirklich sehr reinhängst und gute Fortschritte machst. Wir brauchen das wirklich. Also nein, ich bin nicht sauer.«

10.1 Was eine Schätzung ist

Das Problem ist das unterschiedliche Verständnis von Schätzungen. Vom Business werden Schätzungen gerne als Commitments gesehen. Entwickler hingegen betrachten sie als Vermutung oder Mutmaßung. Der Unterschied ist beachtlich.

10.1.1 Ein Commitment

Ein Commitment ist etwas, was man umsetzen und erreichen *muss*. Wenn Sie sich committen, etwas bis zu einem bestimmten Datum erledigt zu haben, dann müssen Sie einfach zu diesem Zeitpunkt fertig sein. Falls das bedeutet, dass Sie täglich zwölf Stunden arbeiten, auch am Wochenende, und dazu die Familienferien auslassen, dann ist das eben so. Sie haben sich committet und müssen es nun einlösen.

Profis gehen keine Commitments ein, außer sie wissen *genau*, dass sie sie auch erzielen können. Punktum! Wenn Sie gebeten werden, sich für etwas zu committen, bei dem

Sie nicht *sicher* sind, ob Sie das tun können, ist es Ehrensache abzulehnen. Wenn Sie gebeten werden, sich für ein bestimmtes Datum zu committen, und Sie wissen, dass es machbar ist, aber zu Überstunden und Wochenendarbeit führt und Ihre Familie ohne Sie in Urlaub fahren muss, dann müssen Sie sich entscheiden. Doch sollten Sie dann auch die Konsequenzen akzeptieren.

Beim Commitment geht es um *Gewissheit*. Andere werden Ihr Commitment akzeptieren und darauf aufbauend planen. Die Kosten, solche Commitments nicht einzuhalten, sind für die anderen enorm und belasten auch Ihren Ruf. Ein Commitment platzen zu lassen, ist ein Akt der Unehrlichkeit, der kaum weniger belastend ist als eine offene Lüge.

10.1.2 Eine Schätzung

Eine Schätzung ist eine Mutmaßung. Hier wird nichts committet und kein Versprechen abgegeben. Wenn man eine Schätzung nicht einhält, ist das in keiner Weise unehrenhaft. Wir nehmen deswegen Schätzungen vor, weil wir *nicht wissen*, wie lange etwas dauern wird.

Bedauerlicherweise sind die meisten Software-Entwickler miserable Schätzer. Das liegt nicht daran, dass fürs Schätzen irgendeine geheime Fähigkeit vonnöten ist. Dass wir beim Schätzen so schlecht sind, liegt daran, dass wir die wahre Natur einer Schätzung nicht begreifen.

Eine Schätzung ist keine fixe Zahl, sondern eine *statistische Verteilung*. Nehmen wir folgende Situation:

Mike: »Was schätzt du, wie lange du für die Beendigung der »Frickel«-Aufgabe brauchst?«

Peter: »Drei Tage.«

Wird Peter wirklich in drei Tagen fertig sein? Möglich wäre das, aber wie wahrscheinlich? Die Antwort darauf lautet: Wir haben keine Ahnung. Was hat Peter gemeint, und was hat Mike erfahren? Darf Mike, wenn er in drei Tagen zurückkommt, überrascht sein, wenn Peter nicht fertig ist? Warum sollte er? Peter hat sich nicht committet. Peter hat ihm nicht gesagt, wie wahrscheinlich drei Tage im Vergleich zu vier oder gar fünf Tagen sind.

Was würde passieren, wenn Mike Peter danach fragt, wie wahrscheinlich seine Schätzung mit drei Tagen ist?

Mike: »Wie wahrscheinlich ist es, dass du in drei Tagen fertig bist?«

Peter: »Ziemlich wahrscheinlich.«

Mike: »Kannst du das mal in Zahlen sagen?«

Peter: »Fünfundsechzig oder sechzig Prozent.«

- Mike: »Also ist es recht wahrscheinlich, dass du vier Tage brauchst.«
- Peter: »Genau, aber ich könnte vielleicht sogar fünf oder sechs brauchen, obwohl ich das eher bezweifle.«
- Mike: »Wie sehr bezweifelst du das?«
- Peter: »Keine Ahnung ... ich bin zu 95 % sicher, dass ich fertig bin, bevor sechs Tage um sind.«
- Mike: »Du meinst, es könnte auch sieben Tage dauern?«
- Peter: »Tja, nur wenn alles schiefgeht. Ach was, wenn wirklich *alles* danebengeht, brauche ich vielleicht zehn oder gar elf Tage. Aber es ist sehr unwahrscheinlich, dass so viel schiefgeht.«

Nun kommen wir der Wahrheit schon deutlich näher. Peters Schätzung ist eine *Wahrscheinlichkeitsverteilung*. Vor seinem geistigen Auge sieht Peter die Wahrscheinlichkeit der Erfüllung seiner Aufgabe in etwa so wie in Abbildung 10.1.

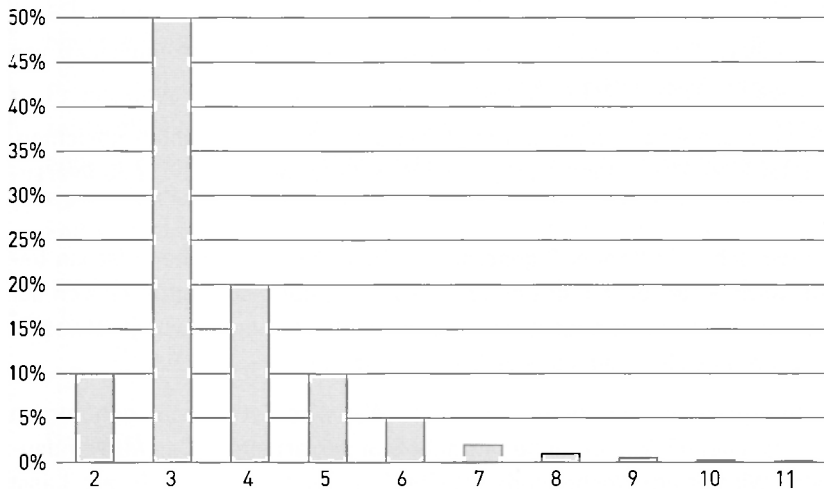


Abbildung 10.1: Die Wahrscheinlichkeitsverteilung

Dieser Abbildung können Sie entnehmen, warum Peter als erste Schätzung drei Tage angegeben hat. Im Diagramm ist das der höchste Balken. Also ist das in Peters Vorstellung die wahrscheinlichste Dauer für die Aufgabe. Aber Mike sieht das anders. Er schaut sich den rechten Bereich des Diagramms an und macht sich Sorgen, dass Peter vielleicht sogar elf Tage bis zur Fertigstellung braucht.

Sollte Mike sich darüber Sorgen machen? Natürlich! Murphy¹ wird sich schon um Peter kümmern, also werden wahrscheinlich auch ein paar Sachen schiefgehen.

¹ Murphys Gesetz besagt: Wenn etwas schiefgehen *kann*, dann wird es auch schiefgehen.

10.1.3 Implizierte Commitments

Nun hat Mike ein Problem. Er ist unsicher angesichts der Zeit, die Peter für die Erledigung seiner Aufgabe braucht. Um diese Ungewissheit zu minimieren, könnte er von Peter einfordern, sich zu committen. Doch Peter ist absolut nicht in der Position, ein Commitment abzugeben.

Mike: »Peter, kannst du mir ein definitives Datum geben, wann du fertig bist?«

Peter: »Nein, Mike. Wie schon gesagt, das braucht drei, vielleicht auch vier Tage.«

Mike: »Können wir dann vier sagen?«

Peter: »Nein, es *können* auch fünf oder sechs sein.«

So, bisher hat sich jeder fair verhalten. Mike hat ein Commitment eingefordert, und Peter hat wohlbedacht abgelehnt, eines abzugeben. Also schlägt Mike einen anderen Pfad ein:

Mike: »Okay, Peter, aber du könntest doch versuchen, das innerhalb von sechs Tagen fertig zu kriegen, oder?«

Mikes Bitte klingt unschuldig genug, und Mike hat sicherlich keine üblen Absichten. Aber um was bittet Mike Peter denn nun genau? Was bedeutet das, etwas zu »versuchen«?

Darüber haben wir schon in Kapitel 2 gesprochen. Das Wort »versuchen« ist ein belasteter Begriff. Wenn Peter einwilligt und es »versucht«, dann committet er sich auf sechs Tage. Das kann man nicht anders interpretieren. Wenn man einwilligt, etwas zu versuchen, erklärt man sich bereit, dass es bis dahin gelingt.

Welche andere Interpretation könnte es hier geben? Was wird Peter genau genommen tun, um es zu »versuchen«? Wird er länger als acht Stunden arbeiten? Das wird eindeutig vorausgesetzt. Wird er am Wochenende arbeiten? Ja, auch das ist impliziert. Lässt er den Urlaub mit der Familie sausen? Ja, auch diesen Effekt wird es haben. All dies gehört zum »Versuchen« mit dazu. Wenn Peter diese Dinge nicht macht, könnte Mike ihm vorwerfen, sich nicht genug Mühe gegeben zu haben.

Profis ziehen eine klare Grenzlinie zwischen Schätzungen und Commitments. Sie committen sich erst dann, wenn sie ganz genau wissen, dass sie es schaffen werden. Sie achten sorgfältig darauf, keine implizierten Commitments abzugeben. Sie kommunizieren die Wahrscheinlichkeitsverteilung ihrer Schätzungen so deutlich wie möglich, damit die Manager entsprechend planen können.

10.2 PERT

1957 wurde die *Program Evaluation and Review Technique* (PERT) geschaffen, um das U-Boot-Projekt Polaris der US-Marine zu unterstützen. Zu PERT gehört eine bestimmte Art und Weise, wie Schätzungen berechnet werden. Dieser Plan bietet eine sehr einfache, aber sehr effektive Weise, um Schätzungen in Wahrscheinlichkeitsverteilungen umzuwandeln, wie sie von Managern benötigt werden.

Für die Einschätzung einer Aufgabe benötigen Sie drei Zahlen: Das nennt man eine *trivariable Analyse*:

- **O:** Optimistische Schätzung. Diese Zahl ist wirklich *höchst* optimistisch. Sie können die Aufgabe letzten Endes nur dann so schnell abschließen, wenn wirklich alles klappt. Damit die Berechnungen funktionieren, darf diese Zahl tatsächlich nur mit einer Wahrscheinlichkeit deutlich unter 1 % auftreten². In Peters Fall wäre das ein Tag – siehe Abbildung 10.1.
- **N:** Standardschätzung (*nominal estimate*). Dies ist die Schätzung mit der größten Erfolgswahrscheinlichkeit. Wenn Sie das in ein Diagramm übertragen, ist es der höchste Balken – siehe Abbildung 10.1. Hier beträgt er drei Tage.
- **P:** Pessimistische Schätzung. Auch dies ist wirklich *höchst* pessimistisch. Darin sollte alles außer Hurrikane, Atomkriege, vagabundierende schwarze Löcher und andere Katastrophen enthalten sein. Auch hier funktioniert die Berechnung nur, wenn die Wahrscheinlichkeit dieser Zahl deutlich geringer als 1 % ist. Bei Peters Fall steht diese Zahl im Diagramm ganz rechts, also 12 Tage.

Mit diesen drei Schätzwerten können wir die Wahrscheinlichkeitsverteilung wie folgt beschreiben:

$$\mu = \frac{O + 4N + P}{6}$$

Dabei ist

$$\sigma = \frac{P - O}{6}$$

σ ist die Standardabweichung³ der Wahrscheinlichkeitsverteilung dieser Aufgabe. Es ist ein Maßstab dafür, wie ungewiss die Aufgabe ist. Wenn diese Zahl groß ist, dann ist auch die Ungewissheit groß. Für Peter beträgt diese Zahl $(12 - 1) / 6$ oder etwa 1,8 Tage.

² Die genaue Zahl einer Normalverteilung ist 1:769 oder 0,13 % oder 3 Sigma. Die Wahrscheinlichkeit von 1:1000 ist wahrscheinlich zutreffend.

³ Wenn Sie nicht wissen, was eine Standardabweichung ist, sollten Sie sich einen guten Überblick über Wahrscheinlichkeit und Statistik verschaffen. Das Konzept ist nicht schwer zu verstehen und wird Ihnen gute Dienste leisten.

Wenn man Peters Schätzung von 4,2/1,8 nimmt, so versteht Mike, dass diese Aufgabe wahrscheinlich innerhalb von fünf Tagen erfüllt sein kann, aber möglicherweise auch sechs oder gar neun Tage dauern könnte.

Aber Mike managt nicht nur diese eine Aufgabe, sondern in seinem Projekt kommen viele Aufgaben vor. Peter hat drei solcher Aufgaben, die er der Reihe nach abarbeiten muss. Er hat diese Aufgaben wie in Tabelle 10.1 eingeschätzt.

Aufgabe	Optimistisch	Standard	Pessimistisch	m	s
Alpha	1	3	12	4,2	1,8
Beta	1	1,5	14	3,5	2,2
Gamma	3	6,25	11	6,5	1,3

Tabelle 10.1: Peters Aufgaben

Was hat es mit dieser »Beta«-Aufgabe auf sich? Es wirkt, als ob Peter sich dessen sehr sicher sei, aber es könnte möglicherweise auch etwas falsch laufen und ihn wesentlich aus der Bahn werfen. Wie soll Mike das interpretieren? Wie viel Zeit soll Mike einplanen, bis Peter alle drei Aufgaben abgeschlossen hat?

Mike kann nun mit einigen einfachen Berechnungen all die drei Aufgaben von Peter kombinieren und gelangt dann zu einer Wahrscheinlichkeitsverteilung für die gesamte Aufgabengruppe. Die Berechnung dafür ist recht unkompliziert:

- $\mu_{\text{sequenz}} = \sum \mu_{\text{Aufgabe}}$

Für jede Aufgabensequenz ist die zu erwartende Zeitdauer die einfache Summe aller Zeiten dieser Aufgaben. Wenn Peter also drei Aufgaben vollenden muss und deren Schätzungen bei 4,2/1,8, 3,5/2,2 und 6,5/1,3 liegen, dann wird Peter mit allen dreien in etwa 14 Tagen fertig sein: $4,2 + 3,5 + 6,5$.

- $\sigma_{\text{sequenz}} = \sqrt{\sum \sigma_{\text{Aufgabe}}^2}$

Die Standardabweichung der Sequenz ist die Quadratwurzel aus der Summe der ins Quadrat erhobenen Standardabweichungen der Aufgaben. Also liegt die Standardabweichung für die drei Aufgaben von Peter insgesamt bei etwa 3.

$$\begin{aligned}
 & [1,8^2 + 2,2^2 + 1,3^2]^{1/2} = \\
 & [3,24 + 2,48 + 1,69]^{1/2} = \\
 & 9,77^{1/2} = \sim 3,13
 \end{aligned}$$

So weiß Mike, dass Peters Aufgaben wahrscheinlich 14 Tage dauern werden, aber es könnten auch 17 Tage (1σ) oder gar bis zu 20 Tage (2σ) werden. Es könnte auch noch länger dauern, aber das ist wiederum sehr unwahrscheinlich.

Schauen Sie sich noch einmal die Tabelle der Schätzungen an. Können Sie den Druck spüren, alle drei Aufgaben in fünf Tagen zu erledigen? Immerhin lauten die Schätzungen bestenfalls 1, 1 und 3. Auch die Standardschätzungen ergeben zusammen nur zehn Tage. Wie sind wir nun auf diese 14 Tage gekommen und möglicherweise noch bis maximal 17 oder 20 Tagen? Die Antwort darauf lautet, dass die Ungewissheit bei diesen Aufgaben auf eine Weise zusammengeführt wird, die die Planung *realistisch* ergänzt.

Wenn Sie als Programmierer schon ein paar Jahre Berufserfahrung haben, sind Ihnen wahrscheinlich schon Projekte begegnet, die man anfangs optimistisch eingeschätzt hat und die dann drei- bis fünfmal so lange dauerten wie erhofft. Das PERT-Schema ist ein einfacher und sinnvoller Weg, um zu verhindern, optimistische Erwartungen zu setzen. Software-Profis achten sorgfältig darauf, keine unvernünftigen Erwartungen zu wecken, obwohl sie unter Druck gesetzt werden, wenigstens zu *versuchen*, schnell zu machen.

10.3 Aufgaben schätzen

Mike und Peter haben einen schrecklichen Fehler begangen. Mike hat Peter gefragt, wie lange er für seine Aufgaben brauchen wird. Peter gab ehrliche trivariable Antworten, aber was ist mit der Meinung seiner Teamkollegen? Könnten die vielleicht andere Vorstellungen haben?

Die wichtigste Ressource für Schätzwerte liefern Ihnen die Leute in Ihrer Umgebung. Sie erkennen Dinge, die Ihnen möglicherweise entgehen. Das hilft Ihnen, Ihre Aufgaben präziser einzuschätzen, als es Ihnen auf eigene Faust gelänge.

10.3.1 Wideband Delphi

In den 1970er-Jahren stellte Barry Boehm eine Schätztechnik vor, die er »Wideband Delphi«⁴ nannte. Im Laufe der Jahre hat es dazu verschiedene Variationen gegeben. Manche sind formell, andere informell, aber alle haben eines gemeinsam: den Konsens.

Die Strategie ist einfach: Ein Team trifft sich, diskutiert eine Aufgabe, schätzt sie ein und wiederholt Diskussion und Schätzung, bis sie sich einig sind.

Zum ursprünglichen Ansatz, so wie Boehm ihn skizziert hat, gehören mehrere Meetings und Dokumente, die für meinen Geschmack zu viel Ritual und Overhead enthalten. Ich ziehe einfache Ansätze mit wenig Overhead wie den folgenden vor.

⁴ [Boehm81]

Fliegende Finger

Alle setzen sich an einen Tisch. Die Aufgaben werden der Reihe nach diskutiert. Für jede Aufgabe wird diskutiert, was dazugehört, was sie durchkreuzen oder verkomplizieren könnte und wie sie implementiert werden sollte. Dann halten die Teilnehmer ihre Hände unter den Tisch und strecken null bis fünf Finger aus – abhängig davon, wie lange ihrer Meinung nach die Aufgabe benötigt. Der Moderator zählt bis drei, und alle decken gleichzeitig die Hand auf.

Wenn alle übereinstimmen, geht's mit der nächsten Aufgabe weiter. Anderenfalls wird die Diskussion fortgesetzt, um festzustellen, woher die Unstimmigkeit kommt. Das wird so lange wiederholt, bis alle übereinstimmen.

Eine Übereinstimmung muss nicht absolut sein. Es reicht aus, dass die Schätzungen dicht beieinander liegen. Also gelten viele 3er und 4er als Übereinkunft. Wenn allerdings alle vier Finger zeigen und jemand nur einen Finger, dann gibt's Gesprächsbedarf.

Man einigt sich zu Beginn des Meetings auf die Skala fürs Schätzen. Dabei könnte es die Zahl der Tage für eine Aufgabe oder eine interessantere Skala wie »Finger mal drei« oder »Finger im Quadrat« sein.

Ganz wichtig ist, dass alle Finger gleichzeitig gezeigt werden. Es sollte nicht passieren können, dass jemand je nachdem, was er bei den anderen sieht, seine Schätzung verändert.

Planungspoker

James Grenning schrieb 2002 eine wunderbare Abhandlung⁵ über »Planungspoker«. Diese Variante von »Wideband Delphi« wurde so populär, dass verschiedene Firmen sich diese Idee für ihr Marketing angeeignet und passende Geschenke dazu herausgebracht haben, und zwar in Form von Spielkarten fürs Planungspoker⁶. Es gibt sogar eine Website namens planningpoker.com, auf der Sie übers Internet mit Teammitgliedern an verschiedenen Standorten Planungspoker spielen können.

Die Idee ist sehr einfach: Jeder Schätzer bekommt ein Blatt Karten mit verschiedenen Zahlen darauf. Die Zahlen 0 bis 5 funktionieren prima, und dadurch entspricht dieses System logisch den »Fliegenden Fingern«.

Das Team wählt eine Aufgabe und diskutiert sie. Dann fordert der Moderator jeden auf, eine Karte zu wählen. Alle entscheiden sich für eine Karte, die ihrer Schätzung entspricht, und halten sie so mit der Rückseite hoch, dass niemand den Wert der Karte sieht. Dann gibt der Moderator die Anweisung, dass alle ihre Karte zeigen.

⁵ [Grenning2002]

⁶ <http://store.mountaingoatsoftware.com/products/planning-poker-cards> oder hier
<http://www.meinspiel.de/ptanning-poker-logo-gestalten-drucken-kaufen>

Der Rest ist wie bei den »Fliegenden Fingern«. Wenn alle übereinstimmen, wird die Schätzung akzeptiert. Anderenfalls werden die Karten wieder ins Blatt gesteckt, und die Spieler erörtern weiter die Aufgabe.

Man hat schon sehr viel »Wissenschaft« aufgebracht, die korrekten Kartenwerte für ein Blatt zu wählen. Manche gehen sogar so weit, mit Karten zu arbeiten, die auf einer Fibonacci-Reihe basieren. Andere fügen Karten mit einem Zeichen für Unendlich oder ein Fragezeichen ein. Ich persönlich halte fünf Karten für ausreichend, auf denen 0, 1, 3, 5 und 10 steht.

Affinitätsschätzung

Eine besonders spezielle Variante des »Wideband Delphi« hat mir Lowell Lindstrom vor einigen Jahren gezeigt. Diesen Ansatz konnte ich bei verschiedenen Kunden und Teams gut nutzen.

Alle Aufgaben werden auf Karten geschrieben, ohne dass Schätzungen sichtbar sind. Das Schätzungsteam steht um einen Tisch oder vor einer Wand, auf der die Karten zufällig verteilt sind. Das Team redet nicht, sondern beginnt einfach damit, die Karten relativ zueinander zu sortieren. Aufgaben, die längere Zeit benötigen, werden auf die rechte Seite verschoben. Kürzere Aufgaben kommen nach links.

Jedes Teammitglied kann jede Karte jederzeit verschieben, auch wenn sie bereits von jemand anderem bewegt wurde. Jede Karte, die öfter als η Mal bewegt wurde, wird zur Diskussion beiseitegelegt.

Nach und nach neigt sich das stille Sortieren dem Ende entgegen, und die Diskussion kann beginnen. Meinungsverschiedenheiten über die Sortierung der Karten werden erörtert. Vielleicht erzielt man einen Konsens, wenn man schnell eine kurze Design-session einberaumt oder einige Wireframes skizziert.

Zum nächsten Schritt gehört es, Linien zwischen den Karten zu ziehen, die für Bucket-Größen stehen. Diese Buckets stehen für Tage, Wochen oder Punkte. Traditionell sind Buckets in einer Fibonacci-Sequenz (1, 2, 3, 5, 8).

Trivariable Schätzungen

Diese Wideband-Delphi-Techniken sind gut geeignet, um eine Standardschätzung einer Aufgabe vorzunehmen. Aber wie bereits erwähnt brauchen wir meist drei Schätzwerte, damit eine Wahrscheinlichkeitsverteilung erstellt werden kann. Die optimistischen und pessimistischen Werte jeder Aufgabe kann man sehr schnell anhand einer der Wideband-Delphi-Varianten generieren. Wenn Sie beispielsweise mit dem Planungspoker arbeiten, soll das Team einfach eine Karte für die pessimistische Einschätzung hochheben, und Sie nehmen dann den größten Wert. Das Gleiche machen Sie für die optimistische Schätzung und nehmen hier den niedrigsten.

10.4 Das Gesetz der großen Zahlen

Schätzungen sind fehlerbehaftet. Darum heißen sie auch Schätzungen. Um dem gegenzusteuern, kann man das *Gesetz der großen Zahlen*⁷ nutzen. Dieses Gesetz bewirkt Folgendes: Wenn Sie eine große Aufgabe auf viele kleinere Aufgaben herunterbrechen und diese dann unabhängig voneinander schätzen, trifft die Summe der einzelnen Schätzwerte viel eher zu als eine Schätzung nur der großen Aufgabe. Der Grund für diese zunehmende Genauigkeit ist, dass sich die Fehler in den kleinen Aufgaben zusammengenommen ausgleichen.

Offen gestanden ist das optimistisch. Bei Schätzungen kommt es eher zu fehlerhaften Unter- als zu Überschätzungen, also ist die Integration bzw. der Ausgleich kaum perfekt zu nennen. Wie dem auch sei: Es ist trotzdem eine gute Technik, große Aufgaben auf kleinere herunterzubrechen und die kleinen unabhängig voneinander zu schätzen. Einige der Fehler gleichen sich wirklich gegenseitig aus, und wenn man auf diese Weise Aufgaben herunterbricht, versteht man sie besser und kann Überraschungen besser aufdecken.

10.5 Schlussfolgerung

Professionelle Software-Entwickler wissen, wie man das Business mit praktischen Schätzungen versorgt, die dann zu Planungszwecken eingesetzt werden können. Sie bieten keine Versprechen an, die sie nicht einhalten können, und committen sich nicht für etwas, bei dem sie nicht sicher sind.

Wenn Profis Commitments abgeben, dann liefern sie *harte Zahlen* und erfüllen sie dann auch. Doch in den meisten Fällen geben Profis keine solchen Commitments ab. Stattdessen liefern sie Wahrscheinlichkeitsschätzungen, die die zu erwartende Vollendungszeit und eine mögliche Varianz beschreiben.

Professionelle Entwickler arbeiten mit dem restlichen Team daran, einen Konsens über die Schätzung zu erzielen, die ans Management weitergegeben wird.

Die in diesem Kapitel beschriebenen Techniken sind *Beispiele* für unterschiedliche Wege, wie professionelle Entwickler solche praktikablen Schätzungen erstellen. Dabei handelt es sich nicht um die einzigen und nicht notwendigerweise die besten Techniken. Es sind einfach jene Techniken, die ich für meine Arbeit am praktikabelsten finde.

⁷ http://de.wikipedia.org/wiki/Gesetz_der_großen_Zahlen

10.6 Bibliografie

- [McConnell2006]: Steve McConnell: *Software Estimation: Demystifying the Black Art*, Redmond, WA: Microsoft Press, 2006.
- [Boehm81]: Barry W. Boehm: *Software Engineering Economics*, Upper Saddle River, NJ: Prentice Hall, 1981.
- [Grenning2002]: James Grenning: *Planning Poker or How to Avoid Analysis Paralysis while Release Planning*, April 2002, <http://renaissancesoftware.net/papers/44-planing-poker.html>



Stellen Sie sich einmal vor, Sie hätten eine außerkörperliche Erfahrung und würden beobachten, wie Sie selbst auf einem OP-Tisch liegen, während ein Chirurg Sie gerade am Herzen operiert. Dieser Chirurg versucht, Ihr Leben zu retten, aber die Zeit ist begrenzt, und so operiert er mit einer Deadline – und das ist durchaus *wörtlich* zu verstehen.

Wie sollte sich dieser Arzt Ihrer Meinung nach verhalten? Wollen Sie, dass er ruhig und gesammelt wirkt? Möchten Sie, dass er dem Team, das ihn unterstützt, klare und präzise Anordnungen gibt? Wollen Sie, dass er seiner Ausbildung gemäß handelt und den Vorgaben seines Fachs folgt?

Oder soll er schwitzen und fluchen? Soll er die Instrumente auf den Tisch knallen oder durch die Gegend werfen? Wollen Sie, dass er das Management beschuldigt, unrealistische Erwartungen zu hegen, und soll er sich andauernd über die knappe Zeit beschweren? Wollen Sie, dass er sich wie ein Profi verhält ... oder wie ein typischer Entwickler?

Der professionelle Entwickler ist unter Druck ruhig und entschlossen. Wenn der Druck steigt, hält er sich an seine Ausbildung und die Regeln seiner Disziplinen, wohl wissend, dass dies der beste Weg ist, die Deadlines und Verpflichtungen zu erfüllen, die ihm auferlegt sind.

1988 arbeitete ich bei Clear Communications. Das war ein Start-up-Unternehmen, das aber nie richtig an den Start kam. Unseren ersten Anschubkredit hatten wir aufgebraucht und mussten uns dann einen zweiten besorgen, später noch einen dritten.

Die anfängliche Produktvision klang sehr gut, aber die Produktarchitektur konnte irgendwie nie auf vernünftige Basis gestellt werden. Zuerst ging es bei dem Produkt sowohl um Software als auch Hardware. Dann wurde es zu reiner Software. Die Software-Plattform wechselte von PCs zu Sparcstations. Die Kunden wechselten von High-End zu Low-End. Schließlich geriet auch der ursprüngliche Zweck des Produkts in den Hintergrund, als die Firma vergebens eine Einnahmequelle zu finden versuchte. In den beinahe vier Jahren, die ich dort zubrachte, glaube ich nicht, dass die Firma je einen Cent verdient hat.

Da braucht gar nicht erwähnt zu werden, dass wir als Software-Entwickler unter beträchtlichem Druck standen. Wir verbrachten viele lange Abende und noch längere Wochenenden im Büro am Terminal. Wir schrieben Funktionen in C, die 3.000 Zeilen lang waren. Es gab laute Streitereien und Beschimpfungen, Intrigen und Ausflüchte. Fäuste wurde durch Wände gestoßen, Stifte zornig ans Whiteboard geworfen. Karikaturen nerviger Kollegen wurden mit der Bleistiftspitze in die Wand geritzt, und der ganze Zorn und Stress hatte nie ein Ende.

Deadlines richteten sich nach Ereignissen. Features mussten wegen einer Messe oder für eine Kundenpräsentation fertiggestellt werden. Alles, was die Kunden wollten, egal wie blöde es war, mussten wir für die nächste Demo fertig haben. Die Zeit war stets zu knapp. Mit der Arbeit waren wir dauernd im Verzug. Die Zeitpläne waren immer erdrückend.

Wenn du 80 Stunden pro Woche arbeitetest, warst du der Held. Wenn du für eine Kundenpräsentation irgendeinen Mist zusammenhacken konntest, warst du der King. Wenn du das oft genug gemacht hast, wurdest du befördert. Wenn nicht, warf man dich raus. Dies war ein Start-up – und es ging immer nur darum, die »Belastung gleichermaßen zu verteilen«. Und damals im Jahre 1988 mit fast zwanzig Jahren Erfahrung im Rücken stieg ich voll darauf ein.

Übrigens war ich jener Entwicklungsmanager, der seine Programmierer anwies, mehr und schneller zu arbeiten. Ich war einer der Typen, die 80 Stunden schufteten und um zwei Uhr früh 3.000 Zeilen lange C-Funktionen schrieb, während meine Kinder zu Hause ohne ihren Vater schlafen gingen. Ich war derjenige, der mit Stiften um sich warf und herumschrie. Ich setzte die Leute vor die Tür, wenn sie sich nicht gut entwickelten. Es war schrecklich. Ich war schrecklich.

Dann kam der Tag, als meine Frau mich zwang, mich mal genau im Spiegel anzusehen. Mir gefiel nicht, was ich sah. Sie sagte mir, dass man einfach nicht mehr gerne in meiner Nähe war. Dem musste ich zustimmen. Aber es gefiel mir nicht, und darum stürmte ich zornig aus dem Haus und lief ziellos herum. Ich lief etwa eine halbe Stunde durch die Gegend und schäumte dabei vor Wut, und dann begann es zu regnen.

Und in meinem Kopf machte etwas »Klick«. Ich begann zu lachen. Ich lachte über meine Torheit. Ich lachte über meinen Stress. Ich lachte den Mann im Spiegel aus, den armen Trottel, der das Leben sich und anderen miserabel machte, und alles nur wegen ... ja, wegen was überhaupt?

Alles veränderte sich an diesem Tag. Ich hörte auf, wie ein Verrückter bis spät in die Nacht zu arbeiten. Ich kippte diesen höchst stressigen Lebensstil. Ich warf nicht mehr mit Stiften und schrieb keine 3.000 Zeilen langen C-Funktionen mehr. Ich war entschlossen, meine berufliche Laufbahn zu genießen, indem ich gut arbeitete und nicht dumm.

Ich schied so professionell, wie ich konnte, aus dieser Stelle aus und wurde Consultant. Seit diesem Tag habe ich niemanden mehr als »Chef« bezeichnet.

11.1 Druck vermeiden

Um unter Druck ruhig zu bleiben, ist es am besten, jene Situationen zu vermeiden, die diesen Druck *verursachen*. Das Vermeiden eliminiert den Druck vielleicht nicht komplett, aber es sorgt in hohem Maße dafür, die Zeiträume mit hohem Stress zu minimieren und abzukürzen.

11.1.1 Commitments

Wie wir in Kapitel 10 festgestellt haben, sollte man es unbedingt vermeiden, sich für Deadlines einverstanden zu erklären, bei denen man nicht sicher sein kann, ob sie einzuhalten sind. Das Business wird diese Commitments immer erwarten, weil es das Risiko eliminieren will. Wir hingegen haben darauf zu achten, dass das Risiko quantifiziert und dem Business präsentiert wird, damit es von dort angemessen gemanagt werden kann. Geht man unrealistische Verpflichtungen ein, hintertreibt man dieses Ziel, und man erweist damit sowohl sich selbst als auch dem Business einen Bärendienst.

Manchmal werden solche Commitments an unserer statt abgegeben. Manchmal merken wir, dass unsere Business-Leute den Kunden etwas versprechen, ohne vorher mit uns Rücksprache zu halten. Wenn dies geschieht, ist es eine Ehrensache, dem Business dabei zu helfen, einen Weg zu finden, um diese Commitments zu erfüllen. Allerdings ist es *keine* Ehrensache, die Commitments zu *akzeptieren*.

Dieser Unterschied ist wichtig. Profis helfen dem Business immer, einen Weg zu finden, um dessen Ziele zu erreichen. Aber Profis werden Commitments, die an ihrer statt vom Business abgegeben wurden, nicht notwendigerweise akzeptieren. Wenn wir letzten Endes keinen Weg finden, um die vom Business abgegebenen Versprechungen einzuhalten, müssen diejenigen, die das Versprechen gaben, die Verantwortung tragen.

Das ist leicht gesagt. Aber wenn Ihr Geschäft misslingt und sich Ihr Gehaltsscheck wegen nicht eingehaltener Commitments verzögert, können Sie sich dem Druck kaum entziehen. Aber wenn Sie sich professionell verhalten haben, können Sie sich wenigstens hoch erhobenen Hauptes auf die Suche nach einem neuen Job machen.

11.1.2 Sauber arbeiten

Wie kann man schnell vorankommen und Deadlines in Schach halten? Indem man sauber bleibt. Profis erliegen nicht der Versuchung, ein Chaos anzurichten, um schneller fertigzuwerden. Profis erkennen, dass *quick and dirty* ein Widerspruch ist. Schmutzig bedeutet immer langsam!

Wir können Druck vermeiden, indem wir unsere Systeme, unseren Code und unser Design so sauber wie möglich halten. Damit ist nicht gemeint, endlose Stunden dafür aufzuwenden, den Code zu polieren, sondern es bedeutet einfach, kein Chaos zu tolerieren. Wir wissen, dass chaotischer Code uns verlangsamt, und das führt dazu, dass Termine nicht eingehalten werden und Commitments platzen. Also machen wir die bestmögliche Arbeit und halten unseren Output so sauber wie möglich.

11.1.3 Verhalten in der Krise

Wenn Sie sich in Krisenzeiten beobachten, wird Ihnen klar werden, woran Sie eigentlich glauben. Halten Sie sich in einer Krise an die Regeln Ihres Faches, dann glauben Sie wirklich an diese Regeln. Wenn Sie entsprechend Ihr Verhalten in einer Krise ändern, dann glauben Sie nicht wirklich an Ihr normales Verhalten.

Wenn Sie in unkritischen Zeiten die Disziplin des Test Driven Development befolgen, aber in Krisenzeiten über den Haufen werfen, dann glauben Sie nicht wirklich daran, dass TDD hilfreich ist. Wenn Sie in normalen Zeiten für einen sauberen Code sorgen, aber in einer Krise das Chaos zulassen, dann glauben Sie nicht wirklich daran, dass ein solches Chaos Sie in Wirklichkeit ausbremst. Wenn Sie in einer Krise paarweise programmieren, aber normalerweise nicht, dann glauben Sie, das Pair Programming effektiver ist als Nicht-Pairing.

Entscheiden Sie sich für Disziplinen, die Sie auch gerne in einer Krise durchhalten würden. *Dann befolgen Sie sie die ganze Zeit.* Befolgt man diese Disziplinen, vermeidet man somit am besten, in eine Krise zu geraten.

Ändern Sie Ihr Verhalten nicht, wenn Sie in die Klemme geraten. Wenn Ihre erwählte Disziplin die beste Art zu arbeiten ist, dann sollten Sie sie auch in schlimmsten Krisenzeiten befolgen.

11.2 Umgang mit Druck

Es ist schön und gut, den Druck zu verhindern, zu lindern und zu eliminieren, aber manchmal ist er trotz aller guten Vorsätze und Vorkehrungen einfach da. Manchmal dauert das Projekt einfach länger, als irgendwer je gedacht hätte. Manchmal ist das anfängliche Design völlig verkehrt und muss überarbeitet werden. Manchmal verliert man ein geschätztes Teammitglied oder einen guten Kunden. Manchmal committen Sie sich für etwas, das Sie schlicht und einfach nicht einhalten können. Was nun?

11.2.1 Keine Panik

Managen Sie Ihren Stress. Durch schlaflose Nächte werden Sie in keiner Weise schneller fertig. Herumhocken und jammern hilft auch nicht. Und am schlimmsten wäre es nun, sich zu beeilen! Widerstehen Sie unter allen Umständen dieser Versuchung! Hektik und Eile treiben Sie nur noch tiefer ins Loch.

Werden Sie stattdessen langsamer. Denken Sie das Problem durch. Entwerfen Sie eine Route zum bestmöglichen Ergebnis und arbeiten Sie dann in einem vernünftigen und kontinuierlichen Tempo auf dieses Ziel hin.

11.2.2 Kommunizieren Sie

Lassen Sie Ihr Team und Ihre Vorgesetzten wissen, dass Sie in Schwierigkeiten stecken. Machen Sie ihnen Vorschläge, wie man am besten aus dem Schlamassel herauskommt. Bitten Sie sie um ihre Einschätzung und Orientierung. Doch vermeiden Sie, für Überraschungen zu sorgen. Nichts macht die Leute wütender und irrationaler als Überraschungen. Überraschungen verzehnfachen den Druck.

11.2.3 Verlassen Sie sich auf Ihr Fachwissen

Wenn es hart auf hart kommt, *vertrauen Sie Ihrem Fachwissen*. Der Grund, warum Sie überhaupt dieses Fachwissen haben, ist, dass es Ihnen in Zeiten mit hohem Druck Orientierung gibt. Genau das sind die Zeiten, in denen man all seinem Fachwissen besondere Aufmerksamkeit schenken sollte. Dies ist *nicht* die Zeit, in denen man es infrage stellen oder über Bord werfen sollte.

Anstatt sich in Panik nach etwas – irgendwas! – umzuschauen, das Ihnen hilft, schneller fertigzuwerden, sollten Sie überlegt und engagiert Ihre erwählten Disziplinen befolgen. Wenn Sie nach TDD arbeiten, dann schreiben Sie eben noch mehr Tests als gewöhnlich. Wenn Sie dem *Merciless Refactoring* anhängen, dann refakturieren Sie noch mehr. Wenn Sie Ihre Funktionen klein halten, dann machen Sie sie sogar noch kleiner. In diesem Hexenkessel können Sie den Kopf nur dann oben behalten, wenn Sie sich auf etwas verlassen, von dem Sie bereits wissen, was funktioniert: Ihr Fachwissen.

11.2.4 Hilfe holen

Machen Sie Pair Programming! Wenn Druck da ist, suchen Sie sich einen Kompagnon als Pairing-Partner. Sie werden schneller fertig und produzieren weniger Defekte. Ihr Pairing-Partner wird Ihnen dabei helfen, sich an die Regeln der Fachdisziplin zu halten, und bewahrt Sie davor, in Panik zu geraten. Ihr Partner wird Dinge entdecken, die Sie übersehen haben, bringt hilfreiche Ideen ein und springt ein, wenn Sie den Fokus verlieren.

Wenn Sie entsprechend merken, dass jemand anderes unter Druck gerät, dann sollten Sie ihm oder ihr das Pairing anbieten. Helfen Sie anderen, wenn sie in einem Loch stecken.

11.3 Schlussfolgerung

Um mit Druck klarzukommen, ist der beste Trick, ihn wenn es irgend geht zu vermeiden und ihn durchzustehen, wenn das nicht möglich ist. Sie vermeiden den Druck, indem Sie Ihre Commitments managen, Ihre Disziplinen befolgen und sauber arbeiten. Sie überstehen den Druck, indem Sie ruhig bleiben, kommunizieren, die Regeln Ihrer Disziplin befolgen und sich Hilfe holen.



Die meiste Software wird in Teams geschaffen. Teams sind am effizientesten, wenn deren Mitglieder professionell zusammenarbeiten. Es ist unprofessionell, der Einzlgänger oder Einsiedler eines Teams zu sein.

Im Jahre 1974 war ich 22. Kaum ein halbes Jahr vorher hatte ich meine wundervolle Frau Ann Marie geheiratet. Die Geburt unseres ersten Kindes Angela sollte noch ein Jahr auf sich warten lassen, und ich arbeitete in einer Abteilung von Teradyne namens Chicago Laser Systems.

Mit mir zusammen arbeitete dort Tim Conrad, mein alter Kumpel aus der Highschool. Tim und ich hatten gemeinsam ein paar echte Wunder vollbracht. In seinem Elternhaus konstruierten wir im Keller Computer. In unserem Keller bauten wir Jakobsleitern. Wir brachten einander bei, wie man PDP-8-Rechner programmiert und integrierte Schaltkreise zu funktionsfähigen Taschenrechnern zusammenfügt.

Wir arbeiteten als Programmierer an einem System, das elektronische Komponenten wie Widerstände und Kondensatoren per Laser mit extremer Genauigkeit zuschneidet. Wir beschnitten beispielsweise den Kristall für die erste Digitaluhr, die Motorola Pulsar

Der von uns programmierte Computer war ein M365, der PDP-8-Klon von Teradyne. Wir schrieben in Assembler, und unsere Quellcodedateien wurden auf Kassetten mit Magnetbändern aufbewahrt. Obwohl wir auf Bildschirmen arbeiten konnten, war der Prozess ziemlich umfassend und verworren, sodass wir den Code für die ersten Arbeiten meist ausdruckten und auf Papier lasen.

Uns stand keine Möglichkeit zur Verfügung, die Codebasis zu durchsuchen. Wir konnten nicht all die Stellen herausfinden, wo eine bestimmte Funktion aufgerufen oder eine Konstante verwendet wurde. Wie Sie sich sicher denken können, bremst einen das echt aus.

Also beschlossen wir eines Tages, einen Kreuzreferenzgenerator zu schreiben. Dieses Programm sollte die Quellcodebänder lesen und eine Auflistung aller Symbole ausdrucken, dazu gleich auch die Datei- und Zeilennummer, in der dieses Symbol erschien.

Das ursprüngliche Programm war recht einfach zu schreiben. Es las einfach die Bänder mit dem Quellcode ein, parste die Assemblersyntax, erstellte eine Symboltabelle und ergänzte die Einträge mit Referenzen. Das funktionierte prima, war aber schrecklich langsam. Erst nach einer ganzen Stunde war unser Master Operating Program (das MOP) verarbeitet.

Das dauerte deswegen so lange, weil wir die stetig wachsende Symboltabelle in einem einzigen Speicherpuffer führten. Wenn wir eine neue Referenz gefunden hatten, fügten wir sie in den Puffer ein und verschoben den restlichen Puffer ein paar Bytes nach unten, um Platz zu schaffen.

Tim und ich waren keine Experten in Sachen Datenstrukturen und Algorithmen. Wir hatten noch nie von Hash-Tabellen oder binären Suchläufen gehört. Wir hatten keine Ahnung, wie man einen Algorithmus schneller macht. Wir wussten bloß, dass das, was wir geschaffen hatten, zu langsam war.

Also probierten wir der Reihe nach Verschiedenes aus. Wir versuchten, die Referenzen in verknüpfte Listen zu packen. Wir probierten, in den Datenfeldern Lücken zu lassen und den Puffer nur zu vergrößern, wenn die Lücken gefüllt waren. Wir versuchten, verlinkte Listen von Lücken zu erstellen. Wir experimentierten einfach mit allen möglichen verrückten Ideen.

Wir standen in unserem Büro am Whiteboard, zeichneten Diagramme unserer Datenstrukturen und führten Berechnungen durch, um die Performance vorauszusagen. Jeden Tag kamen wir mit einer neuen Idee ins Büro. Wir arbeiteten gemeinsam wie besessen.

Ein paar von unseren Ideen verbesserten die Performance. Andere bremsten sie aus. Es war zum Mäusemelken. Damals entdeckte ich zum ersten Mal, wie schwer es ist, Software zu optimieren, und wie wenig intuitiv der Prozess ist.

Am Ende gelang es uns, die Zeit unter 15 Minuten zu drücken, und damit waren wir dem ziemlich nahe gekommen, wie lange es dauerte, einfach das Quellcodeband einzulesen. Also reichte uns das.

12.1 Programmierer kontra Menschen

Wir sind nicht Programmierer geworden, weil wir gerne mit anderen zusammenarbeiten. Generell finden wir zwischenmenschliche Beziehung ziemlich chaotisch und unvorhersagbar. Uns gefällt das saubere und vorhersagbare Verhalten der Maschinen, die wir programmieren. Wir sind am glücklichsten, wenn wir stundenlang allein in einem Raum hocken und uns auf ein wirklich interessantes Problem konzentrieren können.

Okay, das ist nun mächtig verallgemeinert, und es gibt massenhaft Ausnahmen. Viele Programmierer arbeiten gerne und gut mit anderen zusammen und mögen die Herausforderung. Aber der Gruppendurchschnitt neigt weiterhin eher in die von mir aufgezeigte Richtung. Wir Programmierer lieben es, uns von der Umwelt abzuschotten.

12.1.1 Programmierer kontra Arbeitgeber

Als ich in den 1970ern und 1980ern als Programmierer bei Teradyne arbeitete, lernte ich so viel übers Debugging, dass ich *wirklich* gut wurde. Ich liebte die Herausforderung und hingte mich mit Elan und Begeisterung in die Probleme. Vor mir konnte sich kein Bug längere Zeit verstecken!

Wenn ich einen Bug ausgemerzt hatte, war das wie der Sieg in einem Wettkampf oder als hätte ich den Jabberwock erschlagen! Ich ging dann zu meinem Chef Ken Finder und beschrieb ihm (noch mit der Vorpal-Klinge in der Hand) leidenschaftlich, wie *interessant* der Bug war. Eines Tages brach es genervt aus Ken heraus: »Bugs sind nicht interessant. Bugs müssen einfach nur gefixt werden!«

An diesem Tag lernte ich wieder etwas dazu: Bei seiner Arbeit kann man gerne leidenschaftlich sein. Aber es ist auch gut, die Ziele der Leute nicht aus den Augen zu verlieren, die einen bezahlen.

Zur wichtigsten Verantwortung des professionellen Programmierers gehört es, die Ansprüche seines Arbeitgebers zu erfüllen. Das bedeutet, dass Sie mit Ihren Vorgesetzten, Managern, Business-Analysten, Testern und anderen Mitglieder des Teams zusammenarbeiten müssen, um die Business-Ziele *eingehend und umfassend zu verstehen*. Das bedeutet nun nicht, zu einem Business-Streber zu werden. Es bedeutet vielmehr,

dass man begreifen muss, *warum* man nun diesen Code schreibt und wie die Firma davon profitiert, die einem das Gehalt zahlt.

Die schlimmste Aktion eines professionellen Programmierers ist, sich glücklich in einer technologischen Burg zu verschanzen, während um ihn herum das Business tobt und zusammenbricht. Ihr Job ist, das Business am Laufen zu halten!

Also nehmen sich professionelle Programmierer die Zeit, das Business zu verstehen. Sie sprechen mit Nutzern über die Software, die sie verwenden. Sie reden mit der Verkaufsabteilung und den Marketingleuten über die Probleme und Herausforderungen, mit denen sie zu tun haben. Sie unterhalten sich mit ihren Managern, um die kurz- und langfristigen Ziele des Teams zu begreifen.

Kurz gesagt kümmern sie sich um das Schiff, auf dem sie segeln.

Ich bin nur einmal aus einer Programmiererstelle geworfen worden, und zwar im Jahre 1976. Damals arbeitete ich für die Outboard Marine Corp. Ich half dabei, ein automatisches Fabrikationssystem zu schreiben, das anhand von IBM System/7-Rechnern Dutzende von Aluminiumgussmaschinen steuerte.

Technisch gesehen war dies ein herausfordernder und lohnender Job. Die Architektur der System/7-Rechner war faszinierend, und das automatische Fabrikationssystem selbst war wirklich interessant.

Wir hatten außerdem auch ein gutes Team. Der Teamleiter John war kompetent und motiviert. Meine beiden Programmiererkollegen waren sympathisch und hilfsbereit. Wir hatten ein extra für unser Projekt eingerichtetes Computerlabor, in dem wir alle arbeiteten. Der Geschäftspartner war engagiert und saß auch mit bei uns im Labor. Unser Manager Ralph war kompetent, konzentriert und verantwortungsbewusst.

Alles hätte wirklich hervorragend sein können. Das Problem war ich. Ich war bei diesem Projekt und dieser Technologie enthusiastisch genug, aber mit meinen weissen und erfahrenen 24 Jahren brachte ich es nicht über mich, mir Gedanken über das Business oder dessen interne politische Struktur zu machen.

Meinen ersten Fehler beging ich gleich am Tag eins: Ich erschien ohne Krawatte. Ich hatte eine beim Vorstellungsgespräch getragen und zwar gesehen, dass auch alle anderen Krawatten trugen, aber bei mir hatte es nicht Klick gemacht. Also kam Ralph am ersten Tag und sagte mir ganz direkt: »Wir tragen hier Krawatten.«

Ich kann Ihnen gar nicht sagen, wie übel das bei mir ankam. Es ärgerte mich ganz tief in mir drin. Ich trug den Schlips jeden Tag und hasste ihn. Aber warum eigentlich? Ich wusste, worauf ich mich eingelassen hatte. Ich kannte die hier üblichen Konventionen. Warum also war ich so angenervt? Weil ich ein egoistischer narzisstischer kleiner Schwachkopf war.

Ich schaffte es einfach nicht, pünktlich zur Arbeit zu erscheinen. Und ich dachte, das sei auch nicht so wichtig. Immerhin machte ich hier »gute Arbeit«. Und das stimmte auch, meine Programme waren sehr gut geschrieben. Ich war locker der beste technische Programmierer des Teams. Ich konnte schneller und besser Code schreiben als die anderen. Ich konnte Probleme schneller diagnostizieren und lösen. Ich *wusste*, wie wertvoll ich war. Also bedeuteten mir Zeiten und Termine nichts.

Die Entscheidung, mich rauszuwerfen, traf man an jenem Tag, als ich nicht rechtzeitig zu einem Meilenstein erschienen war. Offenbar hatte John uns allen gesagt, dass er bis zum nächsten Montag eine Demo der funktionierenden Features haben wolle. Ich hatte das ziemlich sicher mitbekommen, aber was scherten mich Zeiten und Termine?

Wir waren in der aktiven Entwicklung, und es war kein Produkktivsystem. Es gab keinen Grund, das System laufen zu lassen, wenn niemand im Labor war. Ich muss der Letzte gewesen sein, der am Freitag dort gewesen war, und ich hatte offenbar das System in einem nicht funktionierenden Zustand verlassen. Die Tatsache, dass Montag ein wichtiges Datum war, hatte sich in meinem Gehirn einfach nicht eingeklinkt.

Ich kam am besagten Montag eine Stunde zu spät und sah, wie alle verdrießlich um ein nicht funktionierendes System hockten. John fragte mich: »Warum funktioniert das System heute nicht, Bob?« Ich entgegnete: »Keine Ahnung.« Und machte mich daran, es zu debuggen. Bei mir war der Groschen wegen dieser Montagspräsentation immer noch nicht gefallen, aber ich merkte an der Art, wie alle redeten, dass irgendwas verkehrt war. Dann kam John und flüsterte mir ins Ohr: »Und wenn Stenberg uns nun heute besucht hätte?« Dann wandte er sich verärgert ab.

Stenberg war der für die Automatisierung verantwortliche Vizepräsident. Heutzutage nennen wir so jemanden CIO. Die Frage war für mich bedeutungslos. »Und wenn schon?«, dachte ich. »Das System ist nicht produktiv, warum stellen die sich so an?«

An diesem Tag erhielt ich schriftlich meinen ersten Warnschuss. Darin stand, dass ich meine Haltung auf der Stelle zu ändern hätte oder »es kommt zu einer baldigen Terminierung«. Ich war vor Schreck wie versteinert!

Ich nahm mir etwas Zeit, mein Verhalten zu analysieren, und begann zu erkennen, was ich falsch gemacht hatte. Ich sprach mit John und Ralph darüber. Ich beschloss, mich und meine Arbeit umzukrempeln.

Und das tat ich dann auch! Ich kam nicht mehr zu spät. Ich fing an, auf die Betriebsinterna zu achten. Ich begann zu verstehen, warum John sich wegen Stenberg solche Sorgen gemacht hatte. Ich erkannte die unglückliche Situation, in die ich ihn gebracht hatte, indem das System an jenem Montag in einem nicht lauffähigen Zustand vorgefunden wurde.

Aber es war zu wenig, und das auch noch zu spät. Der Drops war gelutscht. Einen Monat später bekam ich meinen zweiten Warnbrief wegen eines von mir verursachten trivialen Fehlers. Ich hätte an diesem Punkt bereits erkennen sollen, dass die Briefe reine Formsache waren und dass die Entscheidung, mir zu kündigen, längst gefallen war. Aber ich war entschlossen, die Situation zu retten. Also arbeitete ich sogar noch härter.

Das Kündigungsgespräch erfolgte wenige Wochen später.

Ich ging an diesem Tag zu meiner schwangeren, 22-jährigen Frau nach Hause und musste ihr gestehen, dass ich gefeuert worden war. Das ist eine Erfahrung, die ich kein zweites Mal machen möchte.

12.1.2 Programmierer kontra Programmierer

Programmierer haben oft Schwierigkeiten, eng mit anderen Programmierern zusammenzuarbeiten. Das führt zu manch wirklich schrecklichen Problemen.

Eigener Code

Eine der schlimmsten Symptome eines dysfunktionalen Teams ist, wenn jeder Programmierer um *seinen* Code eine Mauer baut und sich weigert, andere Programmierer daran arbeiten zu lassen. Ich habe auch verschiedentlich erlebt, dass Programmierer andere Programmierer ihren Code nicht einmal *anschauen* ließen. Das ist ein Rezept für Katastrophen.

Ich war einmal beratend für eine Firma tätig, die Highend-Drucker fertigte. Diese Maschinen bestanden aus vielen verschiedenen Einzelobjekten wie Einzugsvorrichtungen, Drucker, Stapler, Hefter, Schneidmesser usw. Vom Business wurden diese Geräte und Vorrichtungen jeweils unterschiedlich bewertet. Die Einzugsapparaturen waren wichtiger als die Hefter, und nichts war wichtiger als der Printer.

Jeder Programmierer arbeitete an *seinem* Gerät. Einer schrieb den Code für die Einzugsvorrichtung, ein anderer kümmerte sich um den Code für den Hefter. Jeder behielt seine Technologie für sich selbst und verhinderte, dass irgendwer sonst seinen Code in die Finger bekam. Die politische Macht, die diese Programmierer ausübten, stand in direktem Zusammenhang damit, welchem Wert das Business diesem Gerät zumaß. Der Programmierer, der am Printer arbeitete, war unangreifbar.

Das war ein Desaster für die Technologie. Als Consultant erkannte ich, dass der Code voller Duplikate steckte und die Interfaces zwischen den Modulen schief und krumm waren. Aber ich konnte mit all meinen Argumenten die Programmierer (oder das Business) nicht davon überzeugen, ihre Gebaren zu ändern. Immerhin waren ihre Gehaltsbezüge an die Bedeutung der Geräte gekoppelt, die sie zu pflegen hatten.

Kollektive Eigentümerschaft

Es ist weitaus besser, alle Mauern um Code-Besitztum einzureißen, damit der Code dem gesamten Team gehört. Ich ziehe Teams vor, bei dem jeder jedes Modul auschecken und alle Änderungen vornehmen kann, die er für angemessen hält. Der Code soll dem *Team* gehören und nicht Einzelnen.

Professionelle Entwickler hindern andere nicht daran, am Code zu arbeiten. Sie mauern sich nicht mit ihrem Code ein und beanspruchen ihn nicht allein für sich. Sie arbeiten vielmehr mit den anderen zusammen, und zwar an so vielen Bereichen des Systems wie möglich. Sie lernen voneinander, indem sie zusammen an verschiedenen Teilen des Systems arbeiten.

Pairing

Vielen Programmierern missfällt das Konzept des Pair Programming. Ich finde das eigenartig, denn die meisten Programmierer *werden* in Notfällen paarweise arbeiten. Warum? Weil das eindeutig der effizienteste Weg ist, das Problem zu lösen. Das geht auf die alte Redewendung zurück: Vier Augen sehen mehr als zwei. Aber wenn Pairing der effizienteste Weg ist, im Notfall Probleme zu lösen, warum ist das dann nicht auch der beste Weg, um eine längere Problemperiode zu lösen?

Ich werde Ihnen hier keine Studien dazu vortragen, obwohl man hierzu passend einige zitieren kann. Ich werde keine Anekdoten erzählen, obwohl ich da so einige auf Lager hätte. Ich werde Ihnen nicht mal sagen, wie oft Sie Pair Programming machen sollten. Ich werde Ihnen nur sagen, dass *Profis Pairing machen*. Warum? Weil es zumindest für bestimmte Probleme der effizienteste Weg ist, sie zu beheben. Aber das ist nicht der einzige Grund.

Profis machen auch deswegen Pairing, weil es der beste Weg ist, Wissen miteinander zu teilen. Profis schaffen keine Wissenstresore. Sie lernen vielmehr die unterschiedlichen Bereiche des Systems und die verschiedenen Bestandteile des Business kennen, indem sie paarweise programmieren. Sie erkennen, dass zwar alle Teammitglieder ihren Platz im Spiel abdecken, aber trotzdem auch alle fähig sein sollten, auf einer anderen Position zu spielen, wenn die Situation es erfordert.

Profis machen Pairing, weil das der beste Weg ist, Code zu reviewen. Kein System sollte Code enthalten, der nicht von anderen Programmierern geprüft wurde. Man kann Code-Reviews auf vielerlei Arten durchführen, von denen die meisten entsetzlich ineffektiv sind. Code-Reviews macht man am effektivsten und effizientesten, indem man gemeinsam Code schreibt.

12.2 Kleinhirne

Während des Höhepunkts des Dotcom-Booms fuhr ich im Jahre 2000 einmal mit dem Zug nach Chicago. Als ich auf den Bahnsteig stieg, fiel mir gleich ein riesiges Werbeplakat über dem Ausgang ins Auge. Das Plakat stammte von einer wohlbekannten Software-Firma, die Programmierer rekrutieren wollte. Darauf stand: *Come rub cerebellums with the best* (wörtlich: Reiben Sie Ihr Kleinhirn an den Besten, sinngemäß: Gehen Sie in Wettstreit mit den besten Gehirnen).

Mich traf die ausgesprochene Dummheit dieses Plakats sofort wie ein Schlag. Diese armen, ahnungslosen Werbefritzen versuchten, eine technisch höchst versierte, intelligente und gebildete Zielgruppe von Programmierern anzusprechen. Das ist die Art Menschen, die nicht sonderlich auf Dummheit steht. Die Werbeleute versuchten mit ihrem Slogan, das Bild heraufzubeschwören, wie höchst intelligente Menschen mit anderen ihr Wissen teilen. Unglücklicherweise bezogen sie sich auf jenen Teil des Gehirns, das Kleinhirn, das für die Feinmotorik, aber nicht für die Intelligenz zuständig ist. Also ließen sie gerade jene, deren Aufmerksamkeit sie anzuziehen versuchten, über einen derart dummen Fehler spöttisch lächeln.

Aber an diesem Plakat interessierte mich etwas anderes. Es ließ mich an Personen denken, die ihre Kleinhirne aneinander zu reiben versuchen. Weil sich das Kleinhirn im hinteren Bereich des Gehirns befindet, kann man sich gegenseitig am besten »am Kleinhirn reiben«, wenn man das Gesicht voneinander abwendet. Ich stellte mir ein Team von Programmierern vor, in ihren Arbeitsnischen mit dem Rücken zueinander hockend, die auf ihre Bildschirme starren, während sie Kopfhörer tragen. So rubbelt man das Zerebellum aneinander. Außerdem ist das aber kein Team.

Profis arbeiten *gemeinsam*. Man kann nicht zusammenarbeiten, wenn jeder in seiner eigenen Ecke sitzt und den Kopfhörer übergestülpt hat. Also will ich, dass Sie alle um den Tisch herum sitzen und sich *anschauen*. Ich will, dass Sie die Angst der anderen riechen. Ich will, dass Sie mithören, wenn jemand frustriert vor sich hin murmelt. Ich will gelungene Kommunikation – sowohl verbal als auch körpersprachlich. Ich erwarte von Ihnen, dass Sie alle als Einheit kommunizieren.

Vielleicht glauben Sie, dass Sie alleine besser arbeiten. Das stimmt vielleicht, doch das bedeutet nicht, dass das *Team* besser arbeitet, wenn Sie alleine vor sich hin werkeln. Und tatsächlich ist es eher höchst unwahrscheinlich, dass Sie auf sich allein gestellt bessere Arbeit machen.

Es gibt Zeiten, da ist solitäres Arbeiten genau das Richtige. Manchmal braucht man viel Zeit, um lange und intensiv über ein Problem nachdenken zu können. Es gibt Zeiten, da ist die Aufgabe derart trivial, dass es reine Zeitverschwendung wäre, mit jemandem zusammenzuarbeiten. Aber im Allgemeinen ist es am besten, eng mit anderen zu kooperieren und einen großen Teil der Arbeit paarweise zu schaffen.

12.3 Schlussfolgerung

Vielleicht sind wir nicht deswegen in das Metier der Programmierer gelangt, um mit anderen zu kollaborieren. Tja, Überraschung! Beim Programmieren geht es *vor allem um die Zusammenarbeit mit anderen*. Wir müssen mit unserem Business zusammenarbeiten – genauso wie auch mit unseren Programmiererkollegen.

Ich weiß, ich weiß: Wäre es nicht total super, wenn die uns einfach in einen Raum mit sechs Riesenbildschirmen, einer T3-Leitung, einem parallelen Array superschneller Prozessoren sowie unbegrenztem RAM und Festplattenspeicher sperren und uns endlos mit Diätcola und Pizza versorgen? Leider soll das nicht sein. Wenn wir wirklich unsere Tage mit Programmieren verbringen wollen, dann müssen wir lernen zu reden, und zwar mit ... Menschen¹.

¹ Eine Anspielung auf das letzte Wort im Kinofilm *Soylent Green*.



Was machen Sie, wenn Sie viele kleine Projekte fertigkriegen müssen? Wie sollen Sie diese Projekte den Programmierern zuteilen? Und was ist, wenn Sie ein wirklich riesiges Projekt fertigkriegen müssen?

13.1 Harmoniert es?

Ich habe im Laufe der Jahre eine ganze Reihe Banken und Versicherungsfirmen beraten. Scheinbar ist allen die eigenartige Weise gemeinsam, wie sie Projekte untereinander aufteilen.

Oft läuft ein Projekt bei einer Bank auf einen relativ kleinen Auftrag hinaus, der ein paar Wochen lang ein oder zwei Programmierer beschäftigt. Dieses Projekt wird dann mit einem Projektmanager bestückt, der außerdem noch andere Projekte betreut. Dann kommt noch ein Business-Analyst hinzu, der überdies die Anforderungen für andere Projekte liefert. Hinzu kommen einige Programmierer, die ebenfalls mit anderen Projekten beschäftigt sind. Ein oder zwei Tester werden noch zugeordnet, die ebenfalls andere Projekte am Laufen haben.

Erkennen Sie das Muster? Das Projekt ist so klein, dass man ihm keine Einzelperson zuweist, die Vollzeit daran arbeitet. Alle arbeiten zu 50 oder gar 25 Prozent an diesem Projekt.

Nun muss man sich mal Folgendes klarmachen: *Halbe Personen gibt es nicht.*

Es ist sinnlos, einen Programmierer anzuweisen, die Hälfte seiner Zeit mit Projekt A zu verbringen und die restliche Zeit mit Projekt B, vor allem wenn diese beiden Projekte zwei verschiedene Projektmanager, Business-Analysten, Programmierer und Tester haben. Wie zum Himmel kann man eine solche Monstrosität als *Team* bezeichnen? Das ist kein Team, sondern ein bunt zusammengewürfelter Haufen.

13.1.1 Das zusammengeschweißte Team

Bis sich ein Team geformt hat, braucht es *Zeit*. Die Teammitglieder gehen miteinander Beziehungen ein. Sie lernen, wie sie miteinander kollaborieren können. Sie erfahren, welche Stärken, Macken und Schwächen die jeweils anderen haben. Schließlich beginnt das Team, sich immer mehr miteinander zu verbinden.

An einem derart zusammengeschweißten Team ist wirklich etwas ganz Magisches. Es kann Wunder bewirken. Sie halten sich gegenseitig den Rücken frei und unterstützen einander, sie fordern einander das Beste ab und antizipieren einander. *Sie bringen etwas zustande.*

Ein derart zusammengeschweißtes Team besteht normalerweise aus etwa einem Dutzend Personen. Es können auch bis zu 20 sein oder nur drei, aber die beste Anzahl liegt wahrscheinlich etwa bei zwölf. Das Team sollte sich aus Programmierern, Testern und Analysten zusammensetzen. Und es sollte einen Projektmanager haben.

Das Verhältnis zwischen Testern und Analysten kann sehr unterschiedlich sein, aber 2:1 ist ein gutes Verhältnis. Also hat ein solch eingespieltes Team mit zwölf Leuten sieben Programmierer, zwei Tester, zwei Analysten und einen Projektmanager.

Die Analysten entwickeln die Anforderungen und schreiben dafür automatisierte Akzeptanztests. Die Tester schreiben ebenfalls automatisierte Akzeptanztests. Der Unterschied zwischen beiden besteht in der Perspektive: Beide schreiben Anforderungen, aber während sich die Analysten auf den Business-Wert konzentrieren, achten die Tester auf die Korrektheit. Analysten schreiben die Standardfälle, Tester machen sich Gedanken darum, was schiefgehen könnte, und schreiben die Fälle mit Fehlschlägen und Grenzsituationen.

Der Projektmanager kümmert sich um den Fortschritt des Teams und achtet darauf, dass das Team die Zeitpläne und Prioritäten kennt und versteht.

Eines der Teammitglieder könnte in Teilzeit die Rolle eines Coachs oder Masters spielen und die Verantwortung tragen, den Prozess und die Disziplinen des Teams zu verteidigen. Er agiert als Teamgewissen, wenn das Team in der Versuchung steht, aufgrund von Zeitdruck vom Prozess abzuweichen.

Fermentierung

Ein Team braucht Zeit, um die Differenzen abzubauen, sich zusammenzufinden und wirklich *reibungsarm* miteinander zu arbeiten. Das kann ein halbes Jahr dauern oder auch ein ganzes. Aber wenn es passiert, dann ist das wie Zauberei. Ein so zusammengeschweißtes Team plant gemeinsam, löst Probleme gemeinsam, stellt sich gemeinsam den Herausforderungen *und erledigt einfach seine Arbeit*.

Wenn es erst einmal so weit gekommen ist, ist es grotesk, es wieder auseinanderzunehmen, nur weil das Projekt ausgelaufen ist. Am besten bewahrt man ein solches Team und versorgt es weiter mit Projekten.

Was war zuerst da: das Team oder das Projekt?

Die Banken und Versicherungsunternehmen haben versucht, Teams um Projekte herum zu bilden. Dieses Vorgehen ist töricht. Solche Teams können einfach nicht zusammenfinden und eine verschworene Gemeinschaft bilden. Die Einzelnen haben jeweils nur kurz mit dem Projekt zu tun und dann auch nur für einen kleinen Prozentsatz ihrer Zeit. Also bleibt ihnen gar keine Zeit zu lernen, richtig miteinander umzugehen.

Professionelle Entwicklungsorganisationen bilden keine Teams um Projekte herum, sondern weisen bestehenden, gut eingespielten Teams Projekte zu. Ein reibungsarm arbeitendes Team kann viele Projekte gleichzeitig übernehmen und teilt sich die Arbeit untereinander entsprechend den jeweiligen Fähigkeiten, Ansichten und Skills gemäß auf. Das eingespielte Team wird die Projekte erfüllen.

13.1.2 Aber wie managt man so etwas?

Teams haben eine Geschwindigkeit, mit der sie entwickeln, die sogenannte Velocity¹. Die Velocity eines Teams ist einfach die Menge Arbeit, die es in einer definierten Zeitspanne schafft. Manche Teams messen ihre Velocity in *Punkten* pro Woche, wobei Punkte die Maßeinheit für Komplexität sind. Sie brechen die Features aller bearbeiteten Projekte herunter und schätzen sie in Punkten ein. Dann messen sie, wie viele Punkte sie pro Woche schaffen.

Velocity ist ein statistisches Maß. Ein Team kann in der einen Woche 38 Punkte schaffen, 42 in der nächsten und 25 in der dritten. Im Laufe der Zeit kommt man auf einen Durchschnitt.

Das Management kann Ziele für jedes Projekt ansetzen, das ein Team bekommt. Wenn beispielsweise die durchschnittliche Velocity eines Teams 50 beträgt und es drei Projekte zur Bearbeitung bekommt, dann kann das Management das Team anweisen, seinen Einsatz in 15, 15 und 20 aufzuteilen.

Wenn Sie ein eingespieltes Team an Ihren Projekten arbeiten lassen, hat dieses Schema den Vorteil, dass das Business notfalls sagen kann: »Projekt B ist gerade in einer Krise. Bitte steckt für die nächsten drei Wochen 100 % eurer Arbeit in dieses Projekt.«

Die Prioritäten derart schnell neu zuzuweisen, ist bei zusammengewürfelten Teams praktisch unmöglich, aber eingespielte Teams, die an zwei oder drei Projekten gleichzeitig arbeiten, schwenken im Nu um.

13.1.3 Das Dilemma des Product Owner

Einer der Einwände gegen diesen von mir vertretenen Ansatz lautet, dass die Product Owner einiges von ihrer Sicherheit und Macht verlieren. Projekteigentümer, die ein Team auf ihr Projekt angesetzt haben, können auf den Einsatz dieses Teams zählen. Sie wissen das, denn weil es eine kostspielige Operation ist, ein Team zu formen, wird das Business das Team nicht aus kurzfristigen Gründen auflösen.

Wenn Projekte andererseits an eingespielte Teams vergeben werden und diese Teams mehrere Projekte gleichzeitig angehen, kann das Business für sie auch kurzfristig anders priorisieren. Damit wird der Projekteigentümer mit Blick auf die Zukunft vielleicht unsicher. Die Ressourcen, auf die der Projekteigentümer sich verlassen hat, könnten ihm plötzlich abhandenkommen.

Offen gesagt ziehe ich letztere Situation vor. Dem Business sollten wegen der künstlichen Schwierigkeit, Teams aufzustellen und aufzulösen, nicht die Hände gebunden sein. Wenn das Business beschließt, dass ein Projekt höher priorisiert wird als ein an-

¹ [RCM2003] S. 20–22; [COHN2006]

eres, sollte es Ressourcen schnell umverteilen können. Es obliegt der Verantwortung des Projekteigentümers, für sein Projekt die Argumente zu liefern.

13.2 Schlussfolgerung

Teams sind schwerer zu erstellen als Projekte. Somit ist es besser, dauerhafte Teams zu bilden, die sich ein Projekt nach dem nächsten vornehmen und auch an mehr als einem Projekt auf einmal arbeiten können. Das Ziel beim Formen eines Teams ist, diesem Team genug Zeit zu geben, einander zu verstehen, und es dann als Motor zu bewahren, um viele Projekte umzusetzen.

13.3 Bibliografie

[RCM2003]: Robert C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Upper Saddle River, NJ: Prentice Hall, 2003.

[COHN2006]: Mike Cohn, *Agile Estimating and Planning*, Upper Saddle River, NJ: Prentice Hall, 2006.

14

Mentoring, Lehrzeiten und die Handwerkskunst



Die Qualität der Informatikabsolventen hat mich immer wieder enttäuscht. Es ist nicht so, dass die Absolventen nicht gescheit oder talentiert sind, aber sie haben einfach nicht gelernt, worum es beim Programmieren eigentlich geht.

14.1 Der Grad des Versagens

Ich habe mal ein Bewerbungsgespräch mit einer jungen Frau durchgeführt, die an einer großen Universität an ihrem Master für Informatik arbeitete. Sie hatte sich für ein Praktikum in den Semesterferien beworben. Ich bat sie, mit mir etwas zu programmieren, und sie entgegnete: »Ich schreibe eigentlich keinen Code.«

Bitte lesen Sie den vorigen Absatz noch einmal und überspringen Sie dann diesen hier.

Ich fragte sie, welche Programmierseminare sie im Rahmen ihres Masters belegt habe. Sie meinte, dass sie dazu keine Seminare gemacht hätte.

Vielleicht fangen Sie dieses Kapitel noch einmal an, nur um sicherzugehen, dass Sie nicht in ein Paralleluniversum versetzt wurden oder gerade aus einem bösen Traum erwachen.

An diesem Punkt können Sie sich auch gerne fragen, wie es angehen kann, dass eine Studierende, die einen Master in Informatik machen will, um Programmierkurse herumkommt. Ich habe mich das damals auch gefragt und wundere mich heute immer noch darüber.

Natürlich ist dies das extremste Beispiel aus meiner Reihe enttäuschender Bewerbungsgespräche mit Studienabsolventen. Nicht alle Informatikabsolventen sind enttäuschend – ganz im Gegenteil! Allerdings stellte ich bei denen, die mich enttäuscht haben, eine Gemeinsamkeit fest: Fast alle von ihnen haben sich *das Programmieren selbst beigebracht*, bevor sie zur Uni kamen, und fuhren trotz Uni mit dem Selbstunterricht fort.

Verstehen Sie mich nun aber nicht falsch. Ich halte es für möglich, dass man an einer Universität eine hervorragende Ausbildung bekommt. Ich glaube aber auch, dass man durchs System schlüpfen kann und am Ende ein Diplom in Händen hält, aber auch nicht viel mehr.

Und es gibt noch ein weiteres Problem. Sogar die besten Informatikkurse und -programme bereiten junge Absolventen normalerweise nicht auf das vor, was ihnen in der Branche begegnen wird. Dies soll keine Anklage gegen die Studienordnung generell sein, sondern das gilt vielmehr für fast alle Fachrichtungen. Was man in der Schule lernt und auf was man hinterher bei der Arbeit trifft, sind oft zwei sehr verschiedene Dinge.

14.2 Mentoring

Wie lernen wir zu programmieren? Ich will Ihnen meine Geschichte erzählen, wie ich das Mentoring erlebt habe.

14.2.1 Digi-Comp I – Mein erster Computer

1964 bekam ich von meiner Mutter zu meinem zwölften Geburtstag einen kleinen Plastikcomputer. Er hieß *Digi-Comp I*¹. Er hatte drei Kippschalter und sechs *UND*-Gatter aus Plastik. Man konnte den Output der Kippschalter mit dem Input der *UND*-Gatter verbinden. Man konnte ebenfalls den Output der *UND*-Gatter mit dem Input der Kippschalter verbinden. Kurz gesagt hatte man hier einen 3-Bit-Rechner mit finitem Zustand auf der Hand.

¹ Auf vielen Websites werden Simulatoren für diesen anregenden kleinen Computer angeboten.

Zu diesem Gerät gehörte eine Anleitung mit mehreren Programmen, die man darauf laufen lassen konnte. Die Maschine wurde programmiert, indem man kleine Röhrchen (kurze Segmente von Trinkstrohhalm) auf kleine Zapfen steckte, die aus den Kippschaltern herausragten. Im Handbuch wurde beschrieben, wohin genau man jedes Röhrchen stecken musste, aber nicht, was sie *bewirkten*. Das fand ich sehr frustrierend!

Ich starrte stundenlang diese Maschine an und untersuchte, wie sie auf der niedrigsten Ebene funktionierte, aber ich konnte ums Verrecken nicht herausfinden, wie ich sie dazu brachte, mir zu gehorchen. Auf der letzten Seite stand, man solle per Post einen Dollar an den Hersteller schicken, und bekäme dann ein Handbuch, in dem steht, wie die Maschine zu programmieren sei².

Ich steckte meinen Dollar in den Umschlag und brachte ihn zur Post. Dann wartete ich mit all der Ungeduld eines Zwölfjährigen. Als das Handbuch schließlich eintraf, verschlang ich es regelrecht. In dieser einfachen Abhandlung über boolesche Algebra ging es um die Grundzüge der Zerlegung in boolesche Gleichungen, um das Assoziativ- und das Distributivgesetz und den Lehrsatz von DeMorgan. Das Handbuch zeigte, wie man ein Problem unter Verwendung einer Sequenz boolescher Gleichungen ausdrückt. Es beschrieb außerdem, wie man diese Gleichungen so reduziert, dass sie in sechs *UND*-Gatter passen.

Ich ersann mein erstes Programm. Den Namen weiß ich auch noch: *Mr. Patternsons computerisiertes Gatter*. Ich schrieb die Gleichungen, reduzierte sie und ordnete sie den Röhrchen und Zapfen der Maschine zu. *Und es funktionierte!*

Wenn ich diese drei Wörter schreibe, läuft mir heute noch ein Schauer über den Rücken. Der gleiche Schauer, den dieser Zwölfjährige vor fast einem halben Jahrhundert spürte. Ich sprang sofort darauf an! Mein Leben hatte sich von Grund auf verändert.

Erinnern Sie sich noch an den Moment, als Ihr erstes Programm lief? Hat das Ihr Leben verändert oder Sie auf eine Spur gebracht, von der Sie nie wieder abweichen wollten?

Ich fand das alles nicht ganz alleine heraus. Ich hatte meine Lehrmeister, meine *Mentoren*. Einige sehr freundliche und sehr erfahrene Menschen (denen ich riesengroße Dankbarkeit schulde) nahmen sich die Zeit, um eine Abhandlung über boolesche Algebra zu schreiben, die auch ein Zwölfjähriger nachvollziehen konnte. Sie brachten die mathematische Theorie mit den Gegebenheiten eines kleinen Plastikcomputers zusammen und befähigten mich, diesen Computer dazu zu bringen, das zu machen, was ich wollte.

Ich habe mir gerade eben noch einmal meine Ausgabe dieses schicksalsträchtigen Handbuchs herausgeholt. Ich bewahre es in einer verschlossenen Plastiktüte auf. Aber nichtsdestotrotz forderten die Jahre ihren Tribut: Die Seiten sind vergilbt und spröde. Doch immer noch leuchten die Wörter daraus hervor. Die Eleganz ihrer Beschreibung

2 Dieses Handbuch besitze ich immer noch. In meinem Bücherregal hat es einen Ehrenplatz.

der booleschen Algebra passte auf drei spärliche Seiten. Ihre schrittweise Anleitung durch die Gleichungen der ursprünglichen Programme ist immer noch überzeugend. Es ist ein Meisterwerk. Und ein Werk, das das Leben von zumindest einem jungen Mann änderte. Und doch bezweifle ich, dass ich jemals die Namen der Autoren erfahren werde.

14.2.2 Die ECP-18 in der Highschool

Mit 15 war ich in meinem ersten Highschool-Jahr und verbrachte gerne viel Zeit im Mathematikbereich (wer hätte das gedacht?). Eines Tages wurde dort eine Maschine der Größe einer Tischkreissäge hereingerollt. Es war ein für die Ausbildung gedachter Computer, der speziell für Highschools entwickelt wurde. Unsere Schule bekam eine zweiwöchige Einführung an dieser sogenannten ECP-18.

Ich hielt mich in der Nähe auf, als die Lehrer mit den Technikern sprachen. Diese Maschine hatte einen Wortspeicher mit 15 Bit (*was ist denn mit »Wort« gemeint?*) und einen Trommelspeicher mit 1.024 Wörtern (einen Trommelspeicher kannte ich damals schon, aber nur vom Konzept her).

Als sie das Gerät einschalteten, heulte es auf wie ein Jet beim Start. Ich glaube, das war die Trommel, die auf Touren kam. Nachdem sie ihre Beschleunigung erreicht hatte, lief sie relativ ruhig.

Die Maschine war großartig! Sie bestand im Wesentlichen aus einem Schreibtisch mit einem prächtigen Steuerpult, das oben herausragte wie die Brücke eines Schlachtschiffs. Das Steuerpult war mit Reihen von Lämpchen geschmückt, die gleichzeitig auch Schalter zum Drücken waren. Wenn man an diesem Schreibtisch saß, fühlte man sich wie Captain Kirk in seinem Sessel.

Als ich den Technikern beim Drücken der Schalter zusah, fiel mir auf, dass die Lämpchen beim Drücken angingen. Zum Ausschalten drückte man erneut darauf. Ich stellte auch fest, dass sie auch noch andere Schalter drückten, die mit *Deposit* oder *Run* bezeichnet waren.

Die Schalter in jeder Reihe waren in fünf Cluster mit jeweils drei Schaltern gruppiert. Mein Digi-Comp hatte auch drei Bits, also konnte ich eine oktale Zahl lesen, wenn sie binär ausgedrückt war. Man brauchte nicht viel Gehirnschmalz, um zu erkennen, dass dies einfach fünf oktale Zahlen waren.

Als die Techniker die Schalter drückten, hörte ich sie dabei leise sprechen. Sie drückten 1, 5, 2, 0, 4 in der *Memory Buffer*-Reihe und murmelten »Speichern in 204«. Oder sie drückten 1, 0, 2, 1, 3 und sagten »213 in den *Akkumulator* laden«. Da gab es eine Reihe mit Schaltern namens *Akkumulator*!

Zehn Minuten zuschauen, und für meinen 15-jährigen Grips war ziemlich klar, dass mit 15 *Speichern* gemeint war und mit 10 *Laden* und dass der Akkumulator gespeichert oder geladen wurde und dass die anderen Zahlen die Zahlen eines der 1.024 Wörter in der Trommel waren (also meinten sie *das* mit »Wort«!).

Stück für Stück schnappte mein begieriger Geist immer mehr AnweisungsCodes und Konzepte auf. Als die Techniker dann irgendwann verschwanden, war mir in Grundzügen klar, wie diese Maschine funktionierte.

An diesem Nachmittag schlich ich mich während der Hausaufgabenzeit ins Mathelabor und begann, mit dem Computer herumzuspielen. Ich hatte schon lange vorher gelernt, dass man besser um Verzeihung als um Erlaubnis bittet. Mit den Schaltern gab ich ein kleines Programm ein, das den Akkumulator mit 2 multipliziert und 1 addiert. Ich gab über die Schalter eine 5 in den Akkumulator ein, ließ das Programm laufen und sah im Akkumulator 13(8). Es hatte funktioniert!

Ich gab auf diese Weise mehrere einfache Programme ein, und alle funktionierten wie erwartet. Ich war Herrscher des Universums!

Tage später erkannte ich, wie dumm ich gewesen war – und welch ein Glück ich gehabt hatte. Ich fand ein Blatt mit Instruktionen, das im Mathelabor herumlag. Darauf waren all die verschiedenen Anweisungen und Op-Codes, darunter viele, die ich durch Beobachten der Techniker noch nicht herausgefunden hatte. Ich war erfreut, dass ich jene korrekt interpretiert hatte, die ich kannte. Die anderen fand ich superspannend. Allerdings war HLT eine neue Instruktion. Es war zufällig so, dass die Anweisung *Halt* als Wort nur aus Nullen bestand. Und genauso zufällig hatte ich am Ende aller meiner Programme ein nur aus Nullen bestehendes Wort eingefügt, damit ich es zum Löschen in den Akkumulator laden konnte. Das Konzept eines Halt war mir einfach noch nicht begegnet. Ich hatte einfach angenommen, dass das Programm stoppt, wenn es fertig war!

Ich erinnere mich, dass ich mal im Mathelabor saß und einem Lehrer dabei zusah, wie er sich abmühte, ein Programm ans Laufen zu kriegen. Er versuchte, auf dem angeschlossenen Teletype zwei Dezimalzahlen einzutippen und dann die Summe auszugeben. Jeder, der mal versucht hat, auf einem Minicomputer ein solches Programm in Maschinensprache zu schreiben, weiß, dass das absolut nicht trivial ist. Man musste die Zeichen einlesen, sie in Ziffern konvertieren, dann in Binärzahlen, sie addieren, zurück in Dezimal wandeln und dann wieder in Zeichen konvertieren. Und glauben Sie mir: Es ist *viel* schlimmer, wenn man das Programm über das Steuerpult binär eingibt!

Ich beobachtete, wie er in sein Programm einen Halt einbaute und es dann laufen ließ, bis es stoppte (oh, das ist ja mal eine gute Idee!). Durch diesen primitiven Breakpoint konnte er den Inhalt des Registers untersuchen, um zu sehen, was sein Programm gemacht hatte. Ich weiß noch, wie er »Wow, das war flott!« vor sich hin murmelte. Mensch, da hätte ich ihm aber was erzählen können!

Ich hatte keine Ahnung, was ein Algorithmus war. Diese Art des Programmierens war für mich immer noch Zauberei. Und er redete kein Wort mit mir, während ich ihm über die Schulter schaute. Tatsächlich sprach *niemand* mit mir über diesen Computer. Ich glaube, sie betrachteten mich als Quälgeist, den man ignorieren musste, während er wie eine Motte im Mathelabor herumflatterte. Hier soll nur angemerkt werden, dass weder der Schüler noch die Lehrer ein hohes Maß an sozialen Fertigkeiten entwickelten.

Am Ende bekam er sein Programm ans Laufen. Es war erstaunlich zuzusehen: Er tippte langsam beide Zahlen ein, weil trotz seiner früheren Feststellung dieser Computer eben *nicht* schnell war (stellen Sie sich vor, wie im Jahr 1967 aufeinanderfolgende Wörter aus einer rotierenden Trommel ausgelesen wurden). Als er nach der zweiten Zahl auf die Eingabetaste drückte, blinkte der Computer kurzzeitig ganz heftig und begann mit dem Drucken des Ergebnisses. Das dauerte pro Ziffer etwa eine Sekunde. Er drückte alles bis zur letzten Ziffer, blinkte etwa weitere fünf Sekunden und drückte dann die letzte Ziffer aus, bevor er stoppte.

Warum diese Pause vor dem letzten Zeichen? Das fand ich nie heraus. Aber es ließ mich erkennen, dass die Art, wie ein Problem angegangen wird, sich ganz wesentlich auf den User auswirken kann. Obwohl das Programm die korrekte Antwort ausgab, war immer noch etwas mit ihm nicht in Ordnung.

Das war eine Form des Mentoring, aber ganz gewiss nicht die Art, auf die ich hätte hoffen können. Es wäre schön gewesen, wenn mich einer dieser Lehrer unter seine Fittiche genommen und mit mir gearbeitet hätte. Aber das machte nichts, weil ich sie *beobachtete* und in furiosem Tempo lernte.

14.2.3 Unkonventionelles Mentoring

Ich habe Ihnen diese beiden Geschichten erzählt, weil sie zwei sehr verschiedene Arten des Mentoring beschreiben und beide eher nicht von jener Art sind, die dieses Wort nahelegt. Im ersten Fall lernte ich von den Autoren eines sehr gut geschriebenen Handbuchs. Im zweiten Fall lernte ich durch Beobachten von Leuten, die aktiv versuchten, mich zu ignorieren. In beiden Fällen erlangte ich grundlegendes und profundes Wissen.

Natürlich hatte ich auch andere Lehrmeister. Da war der freundliche Nachbar, der bei Teletype arbeitete und mir zum Spielen eine Schachtel mit 30 Telefonrelais mitbrachte. Gib einem Steppke ein paar Relais und einen elektrischen Eisenbahntransformator, und er erobert die Welt!

Da war der freundliche Nachbar, ein Funkamateurliebhaber, der mir zeigte, wie man ein Multimeter verwendet (das ich prompt kaputt gemacht habe). Da war der Inhaber eines Bürogeschäfts, zu dem ich kommen und mit seinem sehr teuren programmierbaren Taschenrechner »spielen« durfte. Da gab es das Verkaufsbüro von Digital Equipment Corporation, das mir erlaubte, vorbeizukommen und mit deren PDP-8 und PDP-10 zu »spielen«.

Dann gab es Big Jim Carlin: Er war BAL-Programmierer und bewahrte mich davor, bei meinem ersten Programmiererjob rauszufliegen, indem er mir half, ein Cobol-Programm zu debuggen, das ein paar Nummern zu groß für mich war. Er brachte mir bei, wie man Core Dumps liest und Code mit entsprechenden Leerzeilen, Reihen mit Sternchen und Kommentaren formatiert. Er gab mir den ersten Schubs in Richtung gute Handwerkskunst. Es tut mir leid, dass ich mich nicht revanchieren konnte, als er ein Jahr später beim Chef in Ungnade fiel.

Aber offen gesagt war's das schon mit den Mentoren. Es gab Anfang der 1970er-Jahre einfach noch nicht so viele erfahrene Programmierer. Überall, wo ich sonst arbeitete, war *ich* der Erfahrene. Es gab niemanden, der mir dabei half herauszufinden, was echtes professionelles Programmieren ist. Ich hatte kein Vorbild, das mich lehrte, wie man sich verhält oder was man zu wertschätzen hat. Solche Sachen musste ich ganz alleine lernen, und das war absolut nicht leicht.

14.2.4 Schicksalsschläge

Wie ich Ihnen schon berichtet habe, wurde ich ja 1976 bei dieser Arbeit mit dem automatischen Fabrikationssystem gefeuert. Obwohl ich technisch sehr kompetent war, hatte ich nicht gelernt, wie man auf Business oder Business-Ziele achtet. Termine und Deadlines bedeuteten mir nichts. Ich vergaß an einem Montagmorgen eine große Präsentation, verließ das funktionsunfähige System am Freitag vorher und tauchte verspätet am Montag auf, während mich alle böse anstarrten.

Mein Chef schickte mir einen Brief, um mich zu verwarnen: Ich solle mich sofort ändern oder würde entlassen. Das war für mich ein ganz zentraler Schuss vor den Bug. Ich überdachte mein Leben und meine Karriere und machte mich daran, mein Verhalten wesentlich zu ändern – einiges davon habe ich Ihnen in diesem Buch vorgestellt. Aber es war zu wenig, und das auch noch zu spät. Der ganze Schwung lief in die falsche Richtung, und vorher unwesentliche Kleinigkeiten wurden plötzlich relevant. Also hat man mich dann, obwohl ich mich ehrlich bemühte, schließlich aus dem Gebäude eskortiert.

Man kann sich vorstellen, dass es nicht lustig ist, zu seiner schwangeren Frau und der zweijährigen Tochter nach Hause zu kommen und diese Art von Neuigkeit zu beichten. Aber ich riss mich zusammen und nahm diese wirkungsvollen Lebenslektionen mit in den nächsten Job. Den hatte ich dann 15 Jahre inne, und er stellte die echte Grundlage meiner aktuellen Laufbahn.

Am Ende habe ich alles durchgestanden, wuchs und gedieh. Doch es musste einen besseren Weg geben. Es wäre für mich viel besser gewesen, wenn ich einen echten Lehrmeister gehabt hätte, der mir das Tun und Lassen beibringt. Jemand, dem ich hätte zuschauen können, während ich ihm bei kleinen Aufgaben half, und der meine ersten Arbeiten prüft und mich dabei anleitet. Jemand, der mir als Vorbild diente und mich die passenden Werte und Reflexe lehrte. Ein Sensei. Ein Meister. Ein Mentor.

14.3 Die Lehrzeit

Wie machen es die Ärzte? Glauben Sie, dass Medizinabsolventen im Krankenhaus eingestellt und gleich am ersten Tag in den OP gesteckt werden, um eine Herzoperation durchzuführen? Natürlich nicht.

Die Ärzteschaft hat eine Disziplin der intensiven Mentorenschaft entwickelt, die tief in Rituale eingebettet ist und mit Tradition geölt wird. Die Ärzteschaft beaufsichtigt die Universitäten und achtet darauf, dass die Absolventen die beste Ausbildung bekommen. Zu dieser Ausbildung gehören etwa *zu gleichen Teilen* das Studium in den Seminaren sowie die klinische Tätigkeit in Krankenhäusern bei der Arbeit mit Profis.

Für den Studienabschluss und bevor ihnen die Ausübung ihres Berufs gestattet wird, müssen die frischgebackenen Ärzte ein Jahr bei Praktika Berufserfahrung sammeln. Das ist intensives Training *on the job*. Der Praktikant wird von Vorbildern und Lehrern umgeben.

Wenn dieser Arzt mit dem Praktikum fertig ist, muss er je nach Fachrichtung drei bis fünf weitere Jahre im Rahmen seiner Facharztausbildung in der Praxis arbeiten und wird weiter geschult. Diese Assistenzärzte bekommen ihr berufliches Selbstvertrauen, indem sie immer mehr Verantwortung übernehmen, während sie weiterhin von erfahrenen Ärzten angeleitet und beaufsichtigt werden.

Bei vielen Spezialdisziplinen sind möglicherweise weitere ein bis drei Jahre Ausbildung nötig, in denen der Arzt mit seiner Ausbildung und der beaufsichtigten Praxis fortfährt.

Und dann erst haben sie die Berechtigung, ihr Examen abzuschließen und sich zertifizieren zu lassen.

Diese Beschreibung der medizinischen Ausbildungslaufbahn ist sicherlich idealisiert und wahrscheinlich auch nicht sonderlich präzise. Aber die Tatsache bleibt: Wenn viel auf dem Spiel steht, stecken wir unsere Absolventen nicht einfach in ein Zimmer und werfen ihnen gelegentlich etwas zu essen hinein, um dann zu hoffen, dass gute Sachen daraus erwachsen. Warum machen wir das also bei Software-Entwicklern so?

Es stimmt, dass *relativ wenige* Todesfälle durch Software-Bugs verursacht werden. Aber es gibt signifikante *finanzielle* Verluste. Firmen verlieren wegen der inadäquaten Ausbildung ihrer Software-Entwickler eine Menge Geld.

Irgendwie ist die Software-Branche auf den Trichter gekommen, dass Programmierer eben Programmierer sind, und wenn sie fertig studiert haben, dann können sie auch programmieren. Tatsächlich ist es bei Firmen gar nicht so unüblich, Leute direkt aus der Uni abzuwerben, die fast noch Kinder sind, sie in »Teams« zu stecken und ihnen die Aufgabe zu geben, die allerkritischsten Systeme zu erstellen. Welch ein Wahnsinn!

Das machen Malermeister nicht so. Klempner auch nicht, Elektriker ebenfalls nicht. Verflixt, ich glaube, nicht einmal bei Köchen im Schnellrestaurant geht man so vor. Ich denke, dass Firmen, die Informatikabsolventen einstellen, mehr in deren Ausbildung investieren sollten als McDonald's in seine Servicekräfte.

Tun wir nicht so, als wäre das egal. Es steht sehr viel auf dem Spiel. Unsere Zivilisation hängt von Software ab. Mit Software werden die Informationen bewegt und manipuliert, die unser gesamtes Alltagsleben durchdringen. Software steuert bei den Autos die Motoren, das Getriebe und die Bremsen. Sie kümmert sich um unsere Kontostände, schickt uns Rechnungen und akzeptiert unsere Zahlungen. Software wäscht unsere Kleidung und sagt uns die Uhrzeit. Sie bringt Bilder auf den Fernsehschirm, verschickt SMS, erledigt Telefonate und unterhält uns, wenn wir uns langweilen. Überall steckt Software drin.

Da wir Software-Entwickler mit allen Aspekten unseres Lebens betrauen, von den kleinsten Details bis zu den bedeutsamsten Anteilen, bin ich der Ansicht, dass eine vernünftige Periode der Schulung und beaufsichtigter Praxis angemessen sein sollte.

14.3.1 Die Lehrzeit bei der Software

Wie sollte also die Software-Branche junge Absolventen in den Rang der Professionalität einführen? Welche Schritte sollten dabei befolgt werden? Welche Herausforderungen sollten ihnen gestellt werden? Auf welche Ziele sollten sie hinarbeiten? Gehen wir das in umgekehrter Richtung durch.

Meister

Meister nennen wir die Programmierer, die mehr als ein wesentliches Software-Projekt geleitet haben. Üblicherweise haben sie mehr als zehn Jahre Erfahrung: Sie haben auf unterschiedlichen Systemen und Betriebssystemen gearbeitet und kennen verschiedene Sprachen. Sie kennen sich mit Leitung und Menschenführung aus, können mehrere Teams koordinieren, sind tüchtige Designer und Architekten und können wahre Wunder der Informatik vollbringen, ohne hektisch zu werden. Ihnen wurden bereits Positionen im Management angeboten, aber entweder haben sie das abgelehnt oder sie sind – nachdem sie es zuerst akzeptiert haben – wieder ins Glied zurückgetreten oder haben es in ihre primär technische Rolle integriert. Sie pflegen diese technische Rolle, indem sie lesen, studieren, üben, machen und *lehren*. Einem solchen Meister wird die Firma die technische Verantwortung für ein Projekt zuweisen. Denken Sie an »Scotty«.

Gesellen

Dies sind die Programmierer, die geschult, kompetent und voller Elan sind. In dieser Phase ihrer Karriere lernen sie, gut im Team zu arbeiten, und werden zum Teamleiter ernannt. Sie sind in der aktuellen Technologie bewandert, aber ihnen fehlt üblicherweise die Erfahrung mit vielen verschiedenartigen Systemen. Sie kennen sich eher nur mit einer Sprache, einem System und einer Plattform aus, aber bauen ihre Kenntnisse immer weiter aus. Unter ihnen finden wir ein großes Erfahrungsspektrum, aber der Durchschnitt liegt bei fünf Jahren. Auf der einen Seite dieses Spektrums befinden sich zukünftige Meister, auf der anderen stehen die gerade fertig gewordenen Lehrlinge.

Gesellen werden von Meistern oder anderen, erfahreneren Gesellen beaufsichtigt und betreut. Jungen Gesellen erlaubt man Autonomie nur selten. Ihre Arbeit wird engmaschig überwacht. Ihr Code wird *genau* unter die Lupe genommen. Je mehr Erfahrungen sie sammeln, desto eigenständiger werden sie. Die Überwachung tritt in den Hintergrund und wird nuancierter. Schließlich geht sie in die kollegiale Beratung (Peer Review) über.

Lehrlinge/Praktikanten

Die Studienabsolventen beginnen ihre Karriere als Lehrlinge. Lehrlingen wird keine Autonomie zugestanden. Sie werden von Gesellen sehr eng überwacht. Zuerst übernehmen sie gar keine eigenen Aufgaben, sondern assistieren einfach den Gesellen. In dieser Zeit sollten sie intensiv Pair Programming machen. Hier werden die Disziplinen erlernt und verstärkt. Hier schaffen sie sich außerdem die Grundlagen ihrer Werte.

Sie werden durch Gesellen angelernt. Diese achten darauf, dass Lehrlinge die Designprinzipien, Entwurfsmuster, Disziplinen und Rituale kennen. Die Gesellen bringen ihnen TDD, Refactoring, Kalkulationen usw. bei. Sie beauftragen sie mit Lesen, Übungen und Praxis und prüfen ihren Fortschritt.

Die Lehrzeit des Lehrlings sollte etwa ein Jahr dauern. Wenn die Gesellen dann gewillt sind, diesen Lehrling in ihre Ränge aufzunehmen, sollten sie ihn den Meistern empfehlen. Die Meister sollten den Lehrling sowohl durch Interviews als auch Prüfung ihrer erzielten Ergebnisse untersuchen. Wenn die Meister einwilligen, wird der Lehrling zum Gesellen.

14.3.2 Die Realität

Auch dies hier ist alles idealisiert und hypothetisch. Doch wenn Sie die Bezeichnungen ändern und nicht so genau auf die Wörter schauen, werden Sie merken, dass sich das gar nicht so sehr von der Art unterscheidet, wie wir heute *erwarten*, dass die Dinge laufen sollen. Die Absolventen werden von jungen Teamleitern überwacht, die wiederum von Projektleitern beaufsichtigt werden usw. Das Problem ist, dass diese Aufsicht in

den meisten Fällen *nicht technisch ist!* In den meisten Firmen gibt es überhaupt keine technische Beaufsichtigung. Programmierer bekommen Gehaltserhöhungen und werden nach und nach befördert, einfach weil das mit Programmierern eben so läuft.

Der Unterschied zwischen dem, was wir heute machen, und meiner idealisierten Lehrlingszeit ist der Fokus auf das technische Lehren, Trainieren, Beaufsichtigen und Überprüfen. Der Unterschied besteht genau in diesem Konzept, dass professionelle Werte und technischer Scharfsinn gelehrt, genährt, gefördert, gehätschelt und kultiviert werden müssen. Was bei unserem aktuellen sterilen Vorgehen heute fehlt, ist die Verantwortung der Älteren, Jüngere zu unterrichten.

14.4 Die Handwerkskunst

Also können wir nun dieses Wort definieren: *Handwerkskunst*. Was soll das denn sein? Um das zu verstehen, schauen wir uns das Wort *Handwerker* an. Dieses Wort lässt einen an Geschick und Qualität denken. Es bewirkt die Vorstellung von Erfahrung und Kompetenz. Ein Handwerker ist jemand, der schnell arbeitet, ohne hektisch zu werden, der vernünftig kalkuliert und seine Verpflichtungen einhält. Ein Handwerker weiß, wann er *Nein* zu sagen hat, aber legt alles daran, *Ja* sagen zu können. Ein Handwerker ist ein Profi.

Die Handwerkskunst ist jene Mentalität, die Handwerkern eigen ist. Handwerkskunst ist ein Synonym für Werte, Disziplinen, Techniken, innere Haltung und Ergebnisse.

Aber wie übernehmen Handwerker diese Synonyme? Wie erlangen sie diese Mentalität?

Die Handwerkskunst wird von einer Person der nächsten übergeben. Die Älteren unterrichten die Jüngeren darin. Sie wird zwischen Gleichgestellten untereinander ausgetauscht. Man schaut sie sich ab und lernt sie erneut, während die Älteren die Jüngeren beobachten. Handwerkskunst ist ansteckend, also eine Art mentaler Virus. Sie stecken sich damit an, indem Sie andere beobachten und dafür sorgen, dass diese Kunst auch Ihnen in Fleisch und Blut übergeht.

14.4.1 Menschen überzeugen

Man kann niemanden davon *überzeugen*, ein guter Handwerker zu sein. Man kann keinen dazu überreden, die Handwerkskunst und alles, was dazugehört, zu akzeptieren. Argumente sind ineffektiv. Daten sind unmaßgeblich. Fallstudien bedeuten nichts. Die Akzeptanz dieser Deutung von Handwerkskunst ist weniger eine rationale Entscheidung als vielmehr eine emotionale. Denn dies ist etwas sehr *Menschliches*.

Wie schaffen Sie es nun also, dass andere diese Handwerkskunst übernehmen? Denken Sie daran, dass sie ansteckend ist, aber nur, wenn sie auch beobachtet werden kann. Also machen Sie Ihre Kunst *beobachtbar*. Handeln *Sie* als Vorbild. Sie werden zuerst ein solch guter Handwerker und zeigen dann anderen Ihre Handwerkskunst. Die restliche Reifung erledigt dann die Handwerkskunst.

14.5 Schlussfolgerung

In der Schule lernt man die Theorie des Computerprogrammierens. Aber die Schule lehrt keine Disziplin, Praxis und das Geschick, ein guter Handwerker zu sein, und kann das auch nicht. Diese Dinge kann man sich nur durch jahrelange persönliche Betreuung und Mentoring aneignen. Es wird Zeit, dass wir uns in unserer Software-Branche der Tatsache stellen, dass es unsere Aufgabe ist, die nächste Generation der Software-Entwickler zur Reifung zu führen und das nicht den Universitäten überlassen zu können. Es wird Zeit, für Programme zu sorgen, die Lehr- und Praktikumszeiten und langfristige Führung und Orientierung ermöglichen.



1978 arbeitete ich bei Teradyne an dem bereits beschriebenen Telefontestsystem. Das System umfasste etwa 80.000 Zeilen in M365 Assembler geschriebenen Quellcode, und den speicherten wir auf Magnetbändern.

Die Bänder ähnelten jenen 8-Spur-Stereo-Bandkassetten, die damals in den 1970ern so beliebt waren. Dieses Endlosband wurde vom Bandlaufwerk nur in eine Richtung bewegt. Die Kassetten gab es in 10, 25, 50 und 100 Zoll Länge. Je länger das Band war, desto länger dauerte auch das »Zurückspulen«, weil das Laufwerk einfach so lange nach vorne spulte, bis der »Ladepunkt« gefunden war. Ein 100-Zoll-Band brauchte fünf Minuten, um zum Ladepunkt zu kommen. Also passten wir bei der Wahl der Länge unserer Bänder gut auf¹.

¹ Diese Bänder konnten nur in eine Richtung gespult werden. Wenn es also einen Lesefehler gab, konnte man das Bandlaufwerk nicht zurückdrehen und noch einmal lesen lassen. Man musste unterbrechen, was man gerade machte, das Band zurück zum Ladepunkt bringen und dann von vorne anfangen. Das geschah etwa zwei- bis dreimal täglich. Schreibfehler waren ebenfalls sehr häufig, und es gab keine Möglichkeit, wie das Laufwerk dies erkennen konnte. Also beschrieben wir die Bänder immer paarweise und prüften nach Fertigstellung die Paare. War eines der Bänder fehlerhaft, kopierten wir sofort das andere. Wenn beide nicht in Ordnung waren, was sehr selten vorkam, begannen wir mit der ganzen Operation von vorne. So war das Leben damals in den 1970ern.

Logischerweise wurden die Bänder in Dateien unterteilt, und man konnte beliebig viele Dateien auf einem Band haben, wie draufpassten. Um eine Datei zu finden, lud man das Band und sprang dann von einer Datei zur nächsten, bis die gewünschte gefunden war. Wir führten auf einer Liste an der Wand das Verzeichnis für den Quellcode, damit wir immer wussten, wie viele Dateien jeweils zu überspringen waren.

Auf einem Regal im Labor lag eine Kopie des Master-Bands des 100-Zoll-Quellcodebands. Darauf stand MASTER. Wenn wir eine Datei bearbeiten wollten, wurde das MASTER-Quellcodeband in das eine Laufwerk geladen und ein leeres 10-Zoll-Band ins andere. Wir sprangen durch den MASTER bis zur benötigten Datei, die dann auf das leere Band kopiert wurde. Beide Bänder wurden »zurückgespult« und der MASTER wieder ins Regal gestellt.

Im Labor hing an einem schwarzen Brett eine spezielle Liste der Verzeichnisse auf dem MASTER. Nachdem wir die zu bearbeitenden Dateien kopiert hatten, steckten wir eine farbige Nadel neben den Namen dieser Datei ins Brett. So wurden Dateien ausgecheckt!

Wir bearbeiteten die Bänder auf einem Bildschirm. Unser Texteditor ED-402 war tatsächlich sehr gut. Er war ziemlich ähnlich wie vi. Wir lasen eine »Seite« vom Band, bearbeiteten die Inhalte und schrieben diese Seite dann zurück, um die nächste zu lesen. Eine Seite bestand normalerweise aus 50 Codezeilen. Man konnte auf dem Band nicht »vorblättern«, welche Seiten kommen, und auch nicht zurück, um die bisher bearbeiteten Seiten zu sehen. Also arbeiteten wir mit Listings.

In der Tat markierten wir unsere Listings mit allen anstehenden Änderungen, und dann erst bearbeiteten wir die Dateien unseren Auszeichnungen gemäß. *Niemand* schrieb oder modifizierte Code am Terminal! Das galt als Selbstmord.

Wenn wir mit den Änderungen bei allen zu bearbeitenden Dateien fertig waren, fügten wir diese Dateien auf den MASTER ein, um ein funktionierendes Band zu erstellen. Mit diesem Band haben wir dann unsere Kompilierungen und Tests durchgeführt.

Nachdem alle Tests abgeschlossen und wir sicher waren, dass unsere Änderungen funktionierten, schauten wir aufs schwarze Brett. Wenn keine neuen Nadeln im Brett steckten, etikettierten wir das funktionierende Band einfach zu MASTER um und zogen alle Nadeln aus dem Brett. Falls es neue Nadeln gab, zogen wir unsere heraus und übergaben das funktionierende Band an die Person, deren Nadeln immer noch im Brett steckten. Dann musste sie sich um die Zusammenführung kümmern.

Weil wir zu dritt waren und jeder seine eigene Farbe hatte, wussten wir gleich, wer welche Dateien ausgecheckt hatte. Und weil wir alle im gleichen Labor arbeiteten und ständig im Gespräch waren, hatten wir den Status des Bretts im Kopf. Also war das Brett meist überflüssig, und oft brauchten wir es gar nicht.

A.1 Tools

Heutzutage steht den Software-Entwicklern eine große Bandbreite von Tools zur Verfügung. Mit den meisten muss man sich nicht näher beschäftigen, aber es gibt ein paar, mit denen jeder Software-Entwickler vertraut sein sollte. In diesem Kapitel beschreibe ich meinen aktuellen persönlichen Werkzeugkasten. Ich habe keine umfassende Untersuchung aller verfügbaren Tools gemacht, also ist dies nicht als erschöpfende Übersicht zu betrachten. Hier stellte ich einfach die Sachen vor, mit denen ich arbeite.

A.2 Quellcodekontrolle

Wenn es um die Quellcodekontrolle geht, sind Open-Source-Tools meist die beste Option. Warum? Weil sie von Entwicklern für Entwickler geschrieben wurden. Diese Tools schreiben sich die Entwickler selbst, wenn sie etwas brauchen, was funktioniert.

Es gibt auch eine ganze Reihe teurer, kommerzieller »Enterprise«-Versionen solcher Kontrollsysteme. Ich habe gemerkt, dass diese weniger an Entwickler als an Manager, Geschäftsführer und »Tool-Groups« verkauft werden. Ihre Feature-Liste ist beeindruckend und überzeugend. Leider haben sie oft nicht jene Features, die Entwickler eigentlich brauchen. Das wichtigste davon ist *Tempo*.

A.2.1 Ein »Enterprise«-System der Quellcodekontrolle

Vielleicht hat Ihre Firma ein kleines Vermögen in ein solches »Enterprise«-System investiert. Falls ja, haben Sie mein Beileid. Sie sollten aus diplomatischen Gründen wahrscheinlich lieber nicht herumlaufen und allen erzählen: »Uncle Bob sagt, damit solle man nicht arbeiten.« Doch es gibt eine einfache Lösung.

Sie können Ihren Quellcode am Ende jeder Iteration ins »Enterprise«-System einchecken (etwa alle 14 Tage) und in der Mitte jeder Iteration dann mit einem der Open-Source-Systeme arbeiten. So sind alle glücklich und zufrieden, es werden keine Firmenregeln verletzt, und Ihre Produktivität bleibt konstant hoch.

A.2.2 Pessimistisches kontra optimistisches Locking

In den 1980ern schien pessimistisches Locking eine prima Idee zu sein. Schließlich ist der einfachste Weg, um die Aktualisierungsprobleme beim gleichzeitigen Zugriff zu verwalten, sie zu serialisieren. Wenn ich eine Datei bearbeite, lassen Sie also besser die Finger davon. Tatsächlich war das von mir eingesetzte System der farbigen Nadeln Ende der 1970er eine Form des pessimistischen Lockings. Wenn eine Nadel neben der Datei steckte, dann lassen alle anderen sie in Ruhe.

Natürlich bringt auch pessimistisches Locking Probleme. Wenn ich eine Datei sperre und in Urlaub gehe, sitzen alle auf dem Trocknen, die diese Datei bearbeiten wollen. Auch wenn ich eine Datei nur ein oder zwei Tage sperre, Sorge ich unter Umständen bei anderen für Verzögerungen, die an der gleichen Datei etwas ändern müssen.

Unsere Tools sind beim Zusammenführen von gleichzeitig bearbeiteten Quellcodedateien weitaus besser geworden. Eigentlich ist das wirklich erstaunlich, wenn man sich das mal genau überlegt: Die Tools nehmen sich beide Dateien und deren Vorgänger vor und finden anhand mehrerer Strategien heraus, wie man diese gleichzeitigen Änderungen integriert. Und dabei erledigen sie ihre Aufgabe sehr gut.

Also ist die Ära des pessimistischen Lockings vorüber. Wir müssen Dateien nicht mehr sperren, wenn wir sie auschecken. Tatsächlich brauchen wir nicht einmal mehr einzelne Dateien auszuchecken, sondern gleich das ganze System und bearbeiten alle erforderlichen Dateien.

Wenn dann unsere Änderungen eingeecheckt werden sollen, führen wir eine »Update«-Operation durch. So wissen wir, ob jemand vor uns Code eingeecheckt hat. Diese Operation führt automatisch die meisten Änderungen zusammen, findet Konflikte und hilft, den Rest einzufügen. Dann committen wir den zusammengeführten Code.

Ich werde mich weiter hinten im Kapitel noch ausführlich über die Rolle verbreiten, die automatisierte Tests und die fortlaufende Integration bei diesem Prozess spielen. Hier soll nur erwähnt werden, dass *niemals* Code eingeecheckt wird, der nicht alle Tests bestanden hat. *Absolut nie!*

A.2.3 CVS/SVN

Damals war CVS das bewährte einsatzbereite Quellcodekontrollsystem. Es war zu damaligen Zeiten gut, ist aber für heutige Projekte nun etwas in die Jahre gekommen. Obwohl es sehr gut mit einzelnen Dateien und Verzeichnissen umgehen kann, kann es nicht gut Dateien umbenennen und Verzeichnisse löschen. Und der Attic ... Tja, je weniger man darüber sagt, desto besser.

Subversion hingegen funktioniert wirklich sehr schön. Damit kann man das gesamte System in einem Durchgang auschecken. Updaten, Mergen und Committen gehen ganz leicht. Solange Sie keine Verzweigungen brauchen, kann man mit SVN-Systemen ziemlich einfach umgehen.

Verzweigungen

Bis 2008 versuchte ich, Verzweigungen (Branches) generell zu meiden, und ließ nur einfachste Formen zu. Wenn ein Entwickler einen Branch erstellte, musste dieser Zweig vor Ende der Iteration wieder in den Hauptstrang zurückgeführt werden. Tatsächlich war ich beim Thema Verzweigung so streng, dass es in den Projekten, mit denen ich zu tun hatte, nur höchst selten zum Einsatz kam.

Wenn Sie SVN nehmen, dann halte ich dies immer noch für eine gute Richtlinie. Allerdings sind ein paar neue Tools erschienen, durch die die Karten komplett neu gemischt werden. Es handelt sich um *verteilte* Quellcodekontrollsysteme, und git ist mein Favorit.

A.2.4 git

Ich arbeite seit Ende 2008 mit git, und es hat komplett die Art und Weise verändert, wie ich mit Quellcodekontrolle umgehe. Es sprengt den Rahmen dieses Buches zu beschreiben, warum dieses Tool alles aufgemischt hat. Aber wenn man Abbildung A.1 mit Abbildung A.2 vergleicht, könnte man dazu eine Menge ausführlich erklären, was ich aber hier nicht machen werde.

Abbildung A.1: zeigt das Ergebnis von einigen Wochen Arbeit am FitNesse-Projekt unter Verwendung von SVN. Sie sehen die Auswirkungen meiner strengen Regel, die Verzweigungen zu vermeiden. Wir bauten einfach keine Verzweigungen ein. Stattdessen arbeiteten wir mit häufigen Updates, Zusammenführungen und Commits für den Hauptstrang.

Abbildung A.2 zeigt ein paar Wochen Entwicklungsarbeit am gleichen Projekt unter git. Leicht erkennen Sie, wie wir an den verschiedenen Stellen verzweigen und wieder zusammenführen. Nun habe ich aber nicht die Zügel bei meiner Keine-Verzweigungen-Politik fahren lassen, sondern es war einfach ganz naheliegend und völlig praktisch, so zu arbeiten. Jeder Entwickler kann individuell sehr kurzlebige Branches erstellen und sie dann auf Wunsch auch wieder zusammenführen.

Beachten Sie, dass kein Hauptstrang erkennbar ist. Das liegt daran, dass es keinen *gibt*. Bei git braucht man so etwas wie ein zentrales Repository oder einen Hauptstrang nicht. Jeder Entwickler führt seine eigene Kopie der *gesamten* Projekt-History auf seinem lokalen Rechner. Er checkt aus dieser lokalen Kopie aus und wieder ein und führt seinen Code mit anderen bei Bedarf zusammen.

- More bug fixes
- Docs now say that Java 1.5 is required.
- Bug fix
- Many usability and behaviorial improvements.
- Clean up
- Added PAGE_NAME and PAGE_PATH to pre-defined variables.
- Added ** to lpath widget.
- link to the fixture gallery
- fixture gallery release 2.0 (2008-06-09) copied into the trunk wiki at
- Firefox compatability for invisible colliapsible sections; removed .ce
- Updated documentation suite for all changes since last release.
- Enhancement to handle nulls in saved and recalled symbols. Adde
- Added a "Prune" Properties attribute to exclude a page and its chil
- Fixed type-o
- Added check for existing child page on rename.
- Added "Rename" link to Symbolic Links property section; renamed
- Adjusted page properties on recently added pages such that they c
- Enhanced Symbolic Links to allow all relative and absolute path for
- Cleaned up renamPageReponder a bit more.
- Cleaned Up PathParser names a bit. Pop -> RemoveNameFromE
- Cleaned up RenamePageResponder a bit. Fixed TestContentsHel
- updated usage message
- Fixed a bug wherein variables defined in a parent's preformatted bl
- Added explicit responder "getPage" to render a page in case query
- Tweaks to TOC help text.
- New property: Help text; TOCWidget has rollover balloon with new
- Redundant to the JUnit tests and elemental acceptance tests.
- Removed the last of the [acd] tags.
- lcontents -f option enhancement to show suite filters in TOC list; fix
- TOC enhancements for properties (-p and PROPERTY_TOC and F
- 1) Render the tags on non-WikiWord links;
- Added http:// prefix to google.com for firewall transparency.
- Isolate query action from additional query arguments. For example
- Accomodate query strings like "?suite&suiteFilter=X"; prior logic v
- Cleaned up AliasLinkWidget a bit.

Abbildung A.1: FitNesse unter Subversion

Es stimmt, dass ich ein spezielles goldenes Repository führe, in das ich alle Releases und zwischenzeitigen Builds verschiebe. Aber dieses Repository als Hauptstrang zu bezeichnen, wäre etwas daneben. Es ist wirklich nur ein praktischer Snapshot der gesamten History, die jeder Entwickler bei sich lokal führt.

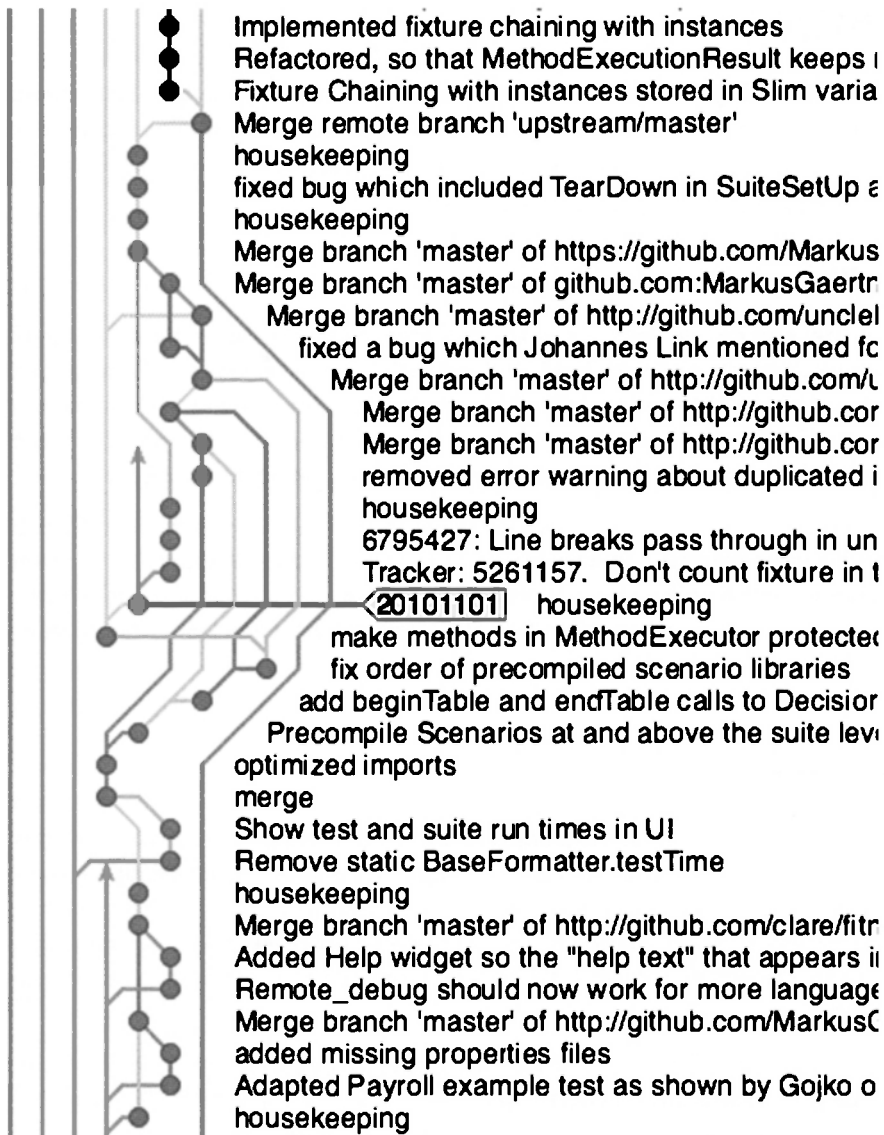


Abbildung A.2: FitNesse unter git

Wenn Sie das nicht begreifen, ist es nicht schlimm. git ist anfangs schon recht knifflig. Man muss sich bei dessen Arbeitsweise eingewöhnen. Aber ich will Ihnen was sagen: In git und anderen ähnlichen Tools sehen wir die Zukunft der Quellcodekontrolle.

A.3 IDE/Editor

Als Entwickler verbringen wir die meiste Zeit damit, Code zu lesen und zu bearbeiten. Im Laufe der Jahrzehnte haben sich die dabei eingesetzten Tools grundlegend gewandelt. Manche sind immens leistungsfähig, und andere haben sich seit den 1970ern nur wenig verändert.

A.3.1 vi

Man könnte annehmen, die Tage von vi als primärem Entwicklungseditor seien lange vorüber. Heutzutage gibt es Tools, die vi (und andere, einfache Texteditoren) um Längen deklassieren. Aber in Wahrheit findet vi jetzt wieder sehr viel Anklang, weil es so einfach zu verwenden, schnell und flexibel ist. vi ist vielleicht nicht so leistungsfähig wie Emacs oder Eclipse, aber immer noch ein schneller und leistungsfähiger Editor.

Aber nun muss ich doch gestehen, dass ich bei vi kein Power-User mehr bin. Es gab mal Zeiten, da war ich als »vi-Gott« bekannt – das ist lange vorbei. Gelegentlich hole ich vi mal hervor, wenn ich schnell eine Textdatei bearbeiten will. Ich habe es sogar kürzlich noch benutzt, um schnell an einer Java-Quelldatei in einer entfernten Umgebung etwas zu ändern. Aber wirklich mit vi programmiert habe ich in den vergangenen zehn Jahren kaum.

A.3.2 Emacs

Emacs ist immer noch einer der leistungsfähigsten Editoren überhaupt und wird das wahrscheinlich auch in den nächsten Dekaden sein – das interne Lisp-Modell garantiert das. Emacs ist ein vielseitiges Allzweck-Tool, dem nichts das Wasser reichen kann. Andererseits finde ich, dass Emacs nicht wirklich mit den spezialisierten IDEs mithalten kann, die jetzt den Markt beherrschen. Und Code zu bearbeiten, ist eben *kein* Allzweck-Job.

In den 1990ern war ich Emacs-Fanatiker. Ich zog anderes nicht einmal in Betracht. Die damaligen Point-and-Click-Editoren waren ein lachhaftes Spielzeug, das kein Entwickler wirklich ernst nehmen konnte. Doch Anfang des Jahres 2000 lernte ich IntelliJ kennen, meine aktuelle IDE der Wahl, und habe nie mehr zurückgeschaut.

A.3.3 Eclipse/IntelliJ

Ich bin absolut überzeugter IntelliJ-User und liebe es. Ich schreibe damit Java, Ruby, Clojure, Scala, JavaScript und vieles andere. Dieses Tool wurde von Programmierern erstellt, die begreifen, was Programmierer brauchen, wenn sie Code schreiben. Im Laufe der Jahre hat es mich selten enttäuscht und fast immer erfreut.

Eclipse gleicht in Leistung und Umfang IntelliJ. Die beiden sind einfach um Längen besser als Emacs, wenn's ums Bearbeiten von Java-Code geht. Es gibt andere IDEs in dieser Kategorie, aber ich stelle sie hier nicht vor, weil ich keine direkte Erfahrung mit ihnen habe.

Diese IDEs hängen Tools wie Emacs ab, weil sie so extrem leistungsfähige Features zur Manipulation von Code enthalten. Bei IntelliJ können Sie beispielsweise mit einem einzigen Befehl aus einer Klasse eine Superklasse extrahieren. Neben vielen weiteren hervorragenden Features können Sie Variablen umbenennen, Methoden extrahieren und Vererbung in Komposition konvertieren.

Mit diesen Tools geht es bei der Bearbeitung von Code nicht mehr länger um Zeilen und Zeichen, sondern um komplexe Manipulationen. Anstatt über die nächsten paar einzutippenden Zeichen und Zeilen nachzudenken, denken Sie an die bevorstehenden Transformationen. Kurz gesagt: Das Programmiermodell ist bemerkenswert anders und höchst produktiv.

Natürlich gibt es diese Leistungsfähigkeit nicht umsonst: Die Lernkurve ist steil, und die Einrichtungszeit für Projekte ist nicht unwesentlich. Diese Tools sind *nicht* leichtgewichtig, sondern brauchen im Betrieb viele Rechnerressourcen.

A.3.4 TextMate

TextMate ist leistungsfähig und leichtgewichtig. Es vollbringt keine solch wundervollen Manipulationen wie IntelliJ und Eclipse und hat auch nicht die leistungsfähige Lisp-Engine und Library wie Emacs. Es ist nicht so flott und flüssig wie vi. Doch die Lernkurve ist flach und die Bedienung intuitiv.

Ich arbeite hier und da mal mit TextMate, vor allem gelegentlich bei C++. Ich würde Emacs für ein großes C++-Projekt nehmen, aber ich bin etwas aus der Übung, um für die kurzen, kleinen C++-Aufgaben Emacs einzusetzen.

A.4 Issue-Tracking-Systeme

Momentan arbeite ich mit Pivotal Tracker. Im Einsatz ist es ein elegantes und einfaches System, das sehr schön zum agilen/iterativen Ansatz passt. Darüber können alle Stakeholder und Entwickler schnell miteinander kommunizieren. Ich bin sehr damit zufrieden.

Bei sehr kleinen Projekten habe ich manchmal Lighthouse genommen, da es sehr schnell und einfach einzurichten und verwendbar ist. Aber es reicht kaum an den Leistungsumfang von Tracker heran.

Ich habe auch oft einfach ein Wiki genommen. Wikis sind für interne Projekte ganz prima. Damit kann man jedes beliebige Schema einrichten. Man wird nicht zu einem bestimmten Prozess oder in eine rigide Struktur gezwungen. Man kann sie schnell und einfach verstehen und nutzen.

Manchmal ist das allerbeste Issue-Tracking-System ein schwarzes Brett und eine Handvoll Karten. Das schwarze Brett wird in Spalten wie »To Do«, »In Bearbeitung« und »Erledigt« aufgeteilt. Die Entwickler verschieben die Karten einfach entsprechend in die jeweilige Spalte. Tatsächlich ist dies heutzutage vielleicht das am häufigsten von agilen Teams eingesetzte Issue-Tracking-System.

Meinen Kunden empfehle ich, zuerst mit einem manuellen System wie dem schwarzen Brett anzufangen, bevor sie sich ein Tracking-Tool zulegen. Wenn man das manuelle System gemeistert hat, weiß man genug, um sich ein passendes Tool auszuwählen. Und tatsächlich könnte die richtige Wahl auch lauten, manuell weiterzumachen.

A.4.1 Bug-Zähler

Entwicklerteams brauchen definitiv eine Auflistung aller anstehenden Aufgaben. Dazu gehören sowohl neue Tasks und Features als auch zu behebende Bugs. Für jedes Team in vernünftiger Größe (fünf bis zwölf Entwickler) sollte diese Liste Dutzende bis Hunderte von Einträgen umfassen. *Keine dreistelligen Einträge!*

Wenn Sie Tausende von Bugs haben oder auch Tausende von Features und/oder Tasks, läuft irgendetwas schief. Generell sollte die Liste der zu klärenden Themen relativ klein und von daher mit einem leichtgewichtigen Tool wie Lighthouse, Tracker oder einem Wiki zu verwalten sein.

Dafür sind auch kommerzielle Tools erhältlich, die recht anständig zu sein scheinen. Ich habe Kunden erlebt, die damit arbeiten, aber selbst hatte ich noch keine Gelegenheit dazu. Ich stehe diesen Tools nicht ablehnend gegenüber, solange die Zahl der Fälle klein und verwaltbar bleibt. Wenn mit solchen Tools Tausende von Fällen nachverfolgt werden müssen, dann verliert das Wort »Nachverfolgung« seinen Sinn. Sie werden zu Müllhalden voller Aufgaben (und stinken oft auch genauso).

A.5 Continuous Build

Kürzlich habe ich für Continuous Build mit Jenkins als Engine gearbeitet. Dieses leichtgewichtige, einfache Tool benötigt praktisch keine Einarbeitungszeit. Man lädt es herunter, startet es, nimmt ein paar schnelle, einfache Konfigurationen vor und kann sich gleich an die Arbeit machen. Sehr schön!

Meine Philosophie über Continuous Build ist einfach: Man sollte sie ins Quellcodekontrollsystem einklinken. Sobald jemand Code eincheckt, sollte es automatisch einen Build vornehmen und den Status dann ans Team melden.

Das Team muss den Build einfach immer funktionsfähig halten. Wenn der Build misslingt, sollte dieser Vorfall bei allen dafür sorgen, dass sie den Griffel sofort fallen lassen und sich gleich treffen, um das Problem zu beheben. Der Fehler sollte keinesfalls einen Tag oder gar länger dauern dürfen.

Beim FitNesse-Projekt lasse ich jeden Entwickler das Skript für den Continuous Build durchlaufen, bevor er committen darf. Der Build dauert keine fünf Minuten und ist somit auch kein Hinderungsgrund. Falls hier Probleme auftauchen, werden sie von den Entwicklern ausgemerzt, bevor sie committen. Also ist der automatische Build selten problematisch. Wenn automatische Builds misslingen, hat sich als Hauptursache herausgestellt, dass irgendetwas mit der Umgebung problematisch ist, da meine Umgebung für automatische Builds sich deutlich von der Entwicklungsumgebung der Programmierer unterscheidet.

A.6 Tools für Unit-Tests

Jede Sprache hat ihr eigenes Tool für Unit-Tests. Meine Favoriten sind JUnit für Java, rspec für Ruby, NUnit für .NET, Midje für Clojure und CppUTest für C und C++.

Egal für welches Tool Sie sich entscheiden, ein paar grundlegende Features sollten alle unterstützen.

1. Die Tests sollte man einfach und schnell ausführen können. Ob das über IDE-Plugins oder einfache Befehlszeilentools erfolgt, ist irrelevant, solange der Entwickler jederzeit nach Belieben diese Tests starten kann. Das Starten der Tests sollte ganz einfach und keine Hürde sein.

Ich starte beispielsweise meine Tests mit CppUTest, indem ich in TextMate einfach `command-M` eintippe. Ich habe diesen Befehl so eingerichtet, dass er mein `makefile` startet. Das führt automatisch die Tests durch und gibt einen einzeiligen Bericht aus, wenn alle Tests bestanden wurden. IntelliJ unterstützt sowohl JUnit als auch rspec, also brauche ich nur auf einen Knopf zu drücken. Für NUnit bekomme ich mit dem Plug-In Resharper den Test-Button.

2. Das Tool sollte klar und deutlich das Bestehen oder Scheitern visualisieren. Dabei ist es gleichgültig, ob das ein grüner Balken oder eine Konsolennachricht mit den Worten »Alle Tests erfolgreich abgeschlossen« ist. Man muss nur schnell und eindeutig erkennen können, ob alle Tests bestanden wurden. Wenn man dazu einen mehrzeiligen Bericht oder schlimmer noch den Output zweier Dateien vergleichen muss, ist das Klassenziel verfehlt.

3. Das Tool sollte klar und deutlich optisch den Fortschritt anzeigen. Dabei ist es nicht wichtig, ob das anhand eines Balkens oder einer Reihe Punkte erfolgt, solange man erkennt, dass der Vorgang noch andauert und die Tests weder abgewürgt noch abgebrochen wurden.
4. Das Tool sollte einzelne Testfälle daran hindern, miteinander zu kommunizieren. JUnit erstellt dazu für jede Testmethode eine neue Instanz der Testklasse und unterbindet so, dass die Tests Instanzvariablen für die Kommunikation untereinander verwenden. Andere Tools durchlaufen die Testmethoden in zufälliger Reihenfolge, damit man sich nicht darauf verlassen kann, dass ein bestimmter Test einem anderen folgt. Das Tool sollte also egal mit welchem Mechanismus dafür sorgen, dass die Tests unabhängig voneinander bleiben. Voneinander abhängige Tests bilden eine sehr tiefe Falle, in die Sie keinesfalls tapen sollten.
5. Mit dem Tool sollte man ganz einfach die Tests schreiben können. Bei JUnit wird das durch eine praktische API gewährleistet, mit der man Behauptungen aufstellen kann. Es arbeitet auch mit Reflexion und Java-Attributen, um die Testfunktionen von den normalen Funktionen zu unterscheiden. Damit kann eine gute IDE automatisch alle Ihre Tests identifizieren, was Ihnen die Mühe erspart, Suites zusammenzuschalten und fehleranfällige Testlisten zu erstellen.

A.7 Tools für Komponententests

Mit diesen Tools werden Komponenten auf API-Ebene getestet. Damit soll gewährleistet sein, dass das Verhalten einer Komponente in einer Sprache beschrieben wird, die für Business und Qualitätssicherung verständlich ist. Tatsächlich wäre es ideal, wenn die Business-Analysten und die Qualitätssicherung diese Spezifikation mit diesem Tool *selbst* schreiben könnten.

A.7.1 Die »Definition of Done«

Mehr als mit jedem anderen Tool sind diese Komponententesttools der Weg, um die »Definition of Done« festzulegen. Wenn Business-Analysten und die Leute von der Qualitätssicherung gemeinsam eine Spezifikation erstellen, die das Verhalten einer Komponente definiert, und wenn diese Spezifikation dann als Test-Suite ausgeführt werden kann, die entweder bestanden wird oder misslingt, dann nimmt »done« eine sehr un-zweideutige Bedeutung an: »Alle Tests wurden erfolgreich bestanden.«

A.7.2 FitNesse

Mein Favorit bei den Tools für Komponententests ist FitNesse. Ich habe einen Großteil davon selbst geschrieben und bin der Hauptentwickler. Also ist es mein Baby.

Das System von FitNesse basiert auf Wikis, mit denen Business-Analysten und die Spezialisten für die Qualitätssicherung Tests in einem sehr einfachen tabellarischen Format schreiben. Diese Tabellen gleichen Parnas-Tabellen sowohl in der Form als auch dem Zweck. Damit kann man Tests schnell zu Suites zusammenstellen, und diese können dann jederzeit durchgeführt werden.

FitNesse ist in Java geschrieben, kann aber auch Systeme in allen möglichen Sprachen testen, weil es mit einem zugrunde liegenden Testsystem kommuniziert, das in einer beliebigen anderen Sprache geschrieben sein kann. Zu den unterstützten Sprachen gehören Java, C#/.NET, C, C++, Python, Ruby, PHP, Delphi und andere.

FitNesse baut auf den beiden Testsystemen Fit und Slim auf. Fit wurde von Ward Cunningham geschrieben und war die ursprüngliche Inspiration für FitNesse und seinesgleichen. Slim ist ein weitaus einfacheres und portableres Testsystem, das heutzutage von den FitNesse-Usern bevorzugt wird.

A.7.3 Andere Tools

Ich kenne mehrere andere Tools, die sich auch für Komponententests eignen.

- RobotFX wurde von Nokia-Ingenieuren entwickelt. Dieses Tool arbeitet mit einem ähnlichen Tabellenformat wie FitNesse, basiert aber nicht auf Wikis, sondern arbeitet mit ganz einfachen Textdateien, die mit Excel o.Ä. vorbereitet wurden. Das Tool ist in Python geschrieben und kann über entsprechende Brücken auch Systeme in allen möglichen anderen Sprachen testen.
- Green Pepper ist ein kommerzielles Tool, das FitNesse in mancherlei Hinsicht gleicht. Es basiert auf dem populären Confluence-Wiki.
- Cucumber ist ein einfaches Text-Tool und wird von einer Ruby-Engine gesteuert, kann aber viele unterschiedliche Plattformen testen. Cucumber ist sprachlich im beliebten Gegeben-wenn-dann-Stil geschrieben.
- JBehave ähnelt Cucumber und ist das logische Elternteil zu Cucumber. Geschrieben wurde es in Java.

A.8 Tools für Integrationstests

Tools für Komponententests sind auch für viele Integrationstests einsetzbar, passen aber weniger gut zu Tests, die über die UI getrieben werden.

Generell sollten Sie nicht viele Tests vom User Interface steuern lassen, weil UIs wegen ihrer Unbeständigkeit berüchtigt sind. Deswegen sind Tests sehr fragil, die das UI durchlaufen.

Doch gibt es ein paar Tests, die das UI durchlaufen *müssen* – am wichtigsten sind hier die Tests des UI selbst zu nennen. Ein paar End-to-end-Tests sollten ebenfalls das gesamte, zusammengefügte System durchlaufen, einschließlich des UI.

Am liebsten teste ich UIs mit Selenium und Watir.

A.9 UML/MDA

Anfang der 1990er-Jahre war ich voller Hoffnung, dass die CASE-Tool-Industrie für eine radikale Änderung in der Art sorgen würde, wie Software-Entwickler arbeiten. Wenn ich aus diesen berausenden Tagen auf die damalige Zukunft blickte, dachte ich, dass dann jeder in Diagrammen auf einem höheren Abstraktionsgrad programmieren und dass textlicher Code der Vergangenheit angehören würde.

Meine Güte, was lag ich da falsch! Mein Traum blieb nicht nur gänzlich unerfüllt, sondern jeder Versuch, sich in diese Richtung zu bewegen, wurde mit elendem Scheitern bestraft. Zwar sind Tools und Systeme im Umlauf, die das Potenzial demonstrieren, aber diese Tools setzen diesen Traum schlicht und einfach nicht um, und kaum jemand scheint Lust zu haben, damit zu arbeiten.

Die Software-Entwickler sollten die Details des textlichen Codes hinter sich lassen und Systeme auf der höheren Ebene einer Diagrammsprache erstellen. Tatsächlich gehört zu diesem Traum, dass sich Programmierer vielleicht gänzlich überflüssig machen. Architekten könnten aus UML-Diagrammen ganze Systeme schaffen. Riesige und coole Engines, die den Sorgen und Nöten einfacher Programmierer keine Beachtung schenken, würden diese Diagramme dann in ausführbaren Code umwandeln. Solcherart war der großartige Traum der Model Driven Architecture (MDA).

Bedauerlicherweise weist dieser prächtige Traum einen winzig kleinen Makel auf: Die MDA geht davon aus, dass Code das Problem ist. Aber Code ist *nicht* das Problem und ist es nie gewesen. Das Problem sind die *Details*.

A.9.1 Die Details

Programmierer sind Verwalter der Details. Das ist unsere Aufgabe. Wir legen das Verhalten vom System bis in die kleinsten Kleinigkeiten fest. Dafür nehmen wir zufällig textliche Sprachen (Code), weil die bemerkenswert komfortabel sind (denken wir beispielsweise an Englisch).

Welche Art von Details managen wir?

Kennen Sie den Unterschied zwischen den Zeichen `\n` und `\r`? Das erste ist ein Zeilenvorschub und das zweite ein Wagenrücklauf. Welcher Wagen ist hier gemeint?

In den 1960er- und Anfang der 1970er-Jahre war die Teletype eines der häufig verwendeten Ausgabegeräte für Computer. Das Modell ASR33² war am weitesten verbreitet.

Dieses Gerät enthielt einen Druckkopf, der zehn Zeichen pro Sekunde drucken konnte. Der Druckkopf bestand aus einem kleinen Zylinder, auf dem sich die erhabenen Zeichen befanden. Der Zylinder rotierte, hob und senkte sich, sodass sich vor dem Papier jeweils das richtige Zeichen befand, und dann schlug ein kleiner Hammer den Zylinder gegen das Papier. Zwischen Zylinder und Papier befand sich ein Farbband, und die Tinte darauf wurde in Form des Zeichens aufs Papier übertragen.

Der Druckkopf fuhr auf einem Wagen hin und her. Nach jedem Zeichen rückte der Wagen eine Stelle nach rechts und nahm den Druckkopf mit. Wenn der Wagen ans Ende der Zeile mit 72 Zeichen angekommen war, musste man den Wagen explizit zurückfahren lassen, indem man die Zeichen (`\r = 0 x 0D`) für den Wagenrücklauf sendete. Anderenfalls hätte der Druckkopf mit dem Drucken der Zeichen in der 72. Spalte weitergemacht und ein hässliches schwarzes Rechteck dort gemalt.

Natürlich reichte das noch nicht: Der Wagenrücklauf schob das Papier nicht bis in die nächste Zeile. Wenn man den Wagen zurückschickte, aber kein Zeichen für den Zeilenvorschub (`\n = 0 x 0A`) sendete, wurde die neue Zeile über die alte gedruckt.

Somit lautete die Zeilenendsequenz für einen ASR33-Fernschreiber »`\r\n`«. Tatsächlich musste man dabei vorsichtig sein, weil der Wagen für den Rücklauf vielleicht länger als 100 ms brauchte. Wenn man »`\n\r`« sendete, wurde das nächste Zeichen vielleicht direkt beim Rücklauf gedruckt und produzierte dann ein verschmiertes Zeichen in der Zeilenmitte. Um auf Nummer sicher zu gehen, polsterten wir die Zeilenendsequenz oft mit ein oder zwei Ausradierzeichen³ (`0 x FF`) aus.

Als in den 1970ern immer weniger Teletypes zum Einsatz kamen, verkürzten Betriebssysteme wie UNIX diese Sequenz einfach auf »`\n`«. Doch andere wie DOS arbeiteten weiterhin entsprechend der Konvention »`\r\n`«.

Wann haben Sie das letzte Mal mit Textdateien zu tun gehabt, die nach der »falschen« Konvention arbeiteten? Auf dieses Problem stoße ich mindestens einmal jährlich. Zwei identische Quellcodedateien sind nicht vergleichbar und generieren unterschiedliche Prüfsummen, weil sie verschiedene Zeilenenden verwenden. Bei Texteditoren funktioniert der Wortumbruch nicht korrekt, oder sie fügen doppelte Leerzeichen in den Text ein, weil die Zeilenenden »verkehrt« sind. Programme stürzen ab, die nicht mit

² <http://de.wikipedia.org/wiki/ASR-33>

³ Ausradierzeichen waren sehr praktisch für die Bearbeitung von Papierbändern. Diese Ausradierzeichen wurden von der Konvention her ignoriert. Deren Code `0 x FF` bedeutete, dass jedes Loch in dieser Zeile des Bands gestanzt war. Das hieß, dass jedes beliebige Zeichen in ein Ausradierzeichen konvertiert werden konnte, indem man es einfach überstanzte. Wenn man also beim Tippen des Programms einen Fehler gemacht hatte, dann konnte man mit der Rückwärtstaste zurückgehen und dann auf Ausradieren drücken. Anschließend fuhr man mit dem Tippen fort.

Leerzeichen umgehen können, weil sie »\r\n« als zwei Zeilen interpretieren. Manche Programme erkennen »\r\n«, aber »\n\r« nicht usw.

Das meine ich mit *Details*. Probieren Sie mal, in UML die grässliche Logik zu programmieren, um mit Zeilenenden umzugehen.

A.9.2 Keine Hoffnung, keine Änderung

Die Hoffnung der MDA-Bewegung bestand darin, dass ein Großteil dieser Details durch Verwendung von Diagrammen statt Code eliminiert werden kann. Diese Hoffnung hat sich bisher als aussichtslos erwiesen. Es stellt sich heraus, dass im Code gar nicht so viele zusätzliche Details eingebettet sind, die über Bilder eliminierbar wären. Bilder besitzen überdies ihre eigenen zufälligen Details. Sie weisen ihre eigene Grammatik, Syntax und Regeln sowie Einschränkungen auf. Also ist der Unterschied im Detail letzten Endes nur Tünche.

Die Hoffnung der MDA war, dass Diagramme sich auf einer höheren Abstraktionsebene bewegen können als Code, so wie Java auf einer höheren Ebene liegt als Assembler. Doch auch diese Hoffnung hat sich bisher als Fehleinschätzung erwiesen. Der Unterschied im Abstraktionsgrad ist bestenfalls winzig.

Und nehmen wir schlussendlich einmal an, dass eines Tages jemand eine wirklich nützliche Diagrammsprache erfinden wird. Dann werden es nicht Architekten sein, die diese Diagramme zeichnen, sondern Programmierer. Diese Diagramme werden dann einfach zum neuen Code, und Programmierer werden diesen Code *zeichnen* müssen, weil es letzten Endes immer um Details geht, und es sind die Programmierer, die die Details managen.

A.10 Schlussfolgerung

Seit ich mit dem Programmieren angefangen habe, sind Software-Tools unglaublich leistungsfähig und vielfältig geworden. Mein aktueller Werkzeugkasten ist einfach eine Auswahl dieser Menagerie. Für Quellcodekontrolle nehme ich git, fürs Issue-Tracking habe ich Tracker. Jenkins nehme ich für Continuous Build, IntelliJ als meine IDE, XUnit für Tests und FitNesse für Komponententests.

Ich habe ein MacBook Pro mit einem Intel Core i7 mit 2,8 GHz und einem matten 17-Zoll-Bildschirm, 8 GB RAM, 512 GB SSD plus zwei Extramonitore.

Index

A

- Abkopplung 109
- Ablenkung 99
- Abstand
 - von der Arbeit 109
- Aggression
 - passive
 - bei Akzeptanztests 145
- Aktivität
 - körperliche 166
- Akzeptanztest 138
 - automatisierter 141
 - Autoren 143
 - Definition 138
 - Verhandlungen 145
 - zusätzliche Arbeit 143
- Anforderung
 - Angst vorm Einschätzen 136
 - verfrühte Präzisierung 135
- Arbeitsethik
 - Mentorenarbeit 57
 - Praxis 56
 - Teamwork 57
- Arbeitsgebiet
 - kennen 58
- Arroganz 58
- Artefakt 55
- Aufgabe
 - schätzen 177
- Autofahren 109
- Automatisierte Qualitätssicherung 51

B

- Bereit sein 98
- Bescheidenheit 58
- Bossavit, Laurent 127
- Bowling Game 127
- Branch 223
- Bug-Zähler 228
- Business-Ziel 193

C

- Code
 - bei Musik geschriebener 102
 - Branches 223
 - Debugging 105
 - im Flow geschriebener 101
 - kreativer Input 105
 - mit Beschlag belegter 196
 - Quellcodekontrolle 221
 - Schreibblockaden 104
 - um drei Uhr früh 99
 - Unterbrechungen 103
 - unter Sorgen geschriebener 100
 - Verzweigungen 223
 - Zeit zum Debuggen einplanen 108
- Coding Dojo 127
- Commitment 71
 - Bedeutung 173
 - fehlendes 71
 - impliziertes 176
 - Schätzungen 171
- Commitment siehe Selbstverpflichtung
- Continuous Build 228
- Courage
 - beim Test Driven Development 119
- Cucumber 231
- CVS 222

D

- Deadline
 - sich beeilen 110
 - Überstunden 111
 - unwahre Ablieferung 111
- Defekteinjektionsrate 119
- Definition of Done 112, 138
- Demo-Meeting 163
- Design
 - Test Driven Development 120
- Design-Pattern 55
- Design-Prinzip 55
- Details 232

Index

Disziplin 55
 in Krisenzeiten 188
 Selbstverpflichtung 92
Dokumentation
 Test Driven Development 120

Druck
 Chaos anrichten 188
 Commitments 187
 Hilfe annehmen 190
 Panik 189
 sauber arbeiten 188
 vermeiden 185

Durchlaufzeit 125

Duschen 109

E

Eclipse 226
Eigentümerschaft
 kollektive 197
Einfachheit 80
Einsatz
 hoher 67
Emacs 226
Entschuldigung 49
Entwickler
 Rolle bei Akzeptanztests 144
Erfahrung
 ausbauen 130
Ergebnis
 bestmögliches 64
Erwartung
 bei Selbstverpflichtung 90
Explorativer manueller Test 156

F

Finger
 fliegende 180
FitNesse 230
Flexibilität 52
Flow-Zustand 101
Funktion
 auf Kosten der Struktur abliefern 51
 nicht beschädigen 49

G

Gaillot, Emmanuel 127
Game of Life 127

Geselle 216
Gesetz
 des Test Driven Development 117
 großer Zahlen 182
Gewissheit
 Test Driven Development 118
git 223
Grad des Versagens 207
Graphical User Interface (GUI)
 Akzeptanztests 147
Green Pepper 231
Grenning, James 180

H

Handwerkskunst 217
Hilfe 112
 annehmen 113
 geben 112
 Mentorenarbeit 114
Hoffnung
 bei Deadlines 110

I

IDE/Editor 226
Identifikation
 mit Kunden 58
Impliziertes Commitment 176
Injektionsrate für Defekte
 beim Test Driven Development 119
Input
 kreativer 167
Integration
 andauernde 148
Integrationstest
 bei Teststrategien 155
 Tools 231
IntelliJ 226
Issue-Tracking-System 227

J

Ja sagen 74
 lernen 91
JBehave 231
Jenkins 228

K

Kalkulation
 Angst bei 136
 Kata 127
 Koffein 165
 Kommunikation
 Anforderungen 133
 bei Akzeptanztests 141
 in Zeiten von Druck 189
 Komponententest
 bei Teststrategien 154
 Tools 230
 Konflikt
 in Meetings 163
 Körperliche Aktivität 166
 Kosten
 eines Ja 74
 Krise
 managen 188

L

Lebenslanges Lernen 56
 Lehrling 216
 Lehrzeit 214
 Leidenschaft 193
 Lernen 53
 Lernmethode 54
 Lesen
 als kreativer Input 105
 Lindstrom, Lowell 181

M

MDA 232
 Meeting
 Absagen 161
 Auseinandersetzungen 163
 Iterationsplanung 163
 Retrospektive 163
 sich ausklinken 161
 Stand-up-Meetings 162
 Tagesordnung 162
 Ziel 162
 Mehrdeutigkeit
 bei Anforderungen 136
 späte 136
 Menschen
 kontra Programmierer 193

Mentor 208
 unkonventioneller 212
 Merciless Refactoring 52
 Methode 55
 Model Driven Architecture (MDA) 232
 Müdigkeit 99
 Muskelfokus 166

O

Open Source 130
 Optimistische Schätzung 177
 Optimistisches Locking 221

P

Pair Programming 104
 in Krisenzeiten 190
 Pairing 197
 Passive Aggression 145
 PERT
 Schätzungen 177
 Pessimistische Schätzung 177
 Pessimistisches Locking 221
 Planungspoker 180
 Praktikant 216
 Praxis
 ethisches Handeln 130
 Prioritätsumkehrung 168
 Problem
 privates 100
 Professionalität 45
 Program Evaluation and Review Technique (PERT) 177
 Programmieren 97
 eigene Energie einteilen 108
 Programmierer
 kontra Arbeitgeber 193
 kontra Programmierer 196
 Meister 215
 Projektvorschlag 75
 Pyramide der Testautomatisierung 153

Q

Qualitätssicherung
 als Bug-Netz 50
 als Teammitglied 152
 automatisierte 50
 Charakterisierung 152

Index

- gefundene Bugs 50
- sollte keine Fehler finden 152
- Spezifikationen 152

R

- Randori 129
- Reputation 48
- Richtlinie
 - Beschädige nicht die Funktion 49
 - Beschädige nicht die Struktur 51
 - Richte keinen Schaden an 48
- RobotFX 231
- Rolle
 - feindliche 64

S

- Santana, Carlos 126
- Schätzung
 - Affinitätsschätzung 181
 - Definition 173
 - Gesetz der großen Zahlen 182
 - trivariable 181
 - von Aufgaben 177, 179
- Schicksalsschlag 213
- Schlaf 165
- Selbstverpflichtung
 - erkennen 89
 - Ja sagen lernen 85
 - mangelnde 88
 - mögliche Lösungen 89
 - verräterische Formulierungen 88
- Standardschätzung 177
- Stand-up-Meeting 162
- Strategie
 - für Tests 151
- Struktur
 - Bedeutung 51
 - Flexibilität 51
 - Tests 53
- SVN 222
- Systemtest
 - in Teststrategien 156

T

- Team
 - Dilemma des Product Owner 204
 - erhalten 202

- managen 204
 - um Projekt gebildetes 203
- Teamplayer 68
- Teamwork 68, 191
 - etwas versuchen 64, 70
 - passive Aggression 72
- Test
 - Akzeptanztests
 - Definition 138
 - automatisierter 50
 - manueller explorativer 156
- Test Driven Development (TDD) 115
 - Design 120
 - drei Gesetze 117
 - Unterbrechungen 103
 - was TDD nicht ist 121
- Testgetriebene Entwicklung
 - Definition 50
- Testlauf
 - Bedeutung 50
- TextMate 227
- Thomas, Dave 127
- Tool 221
 - für Komponententest 230
 - für Unit-Tests 229
 - Integrationstests 231

U

- Üben 123
- UML 232
- Ungewissheit
 - und Anforderungen 135
- Unit-Test
 - Akzeptanztests 146
 - bei Teststrategien 153
 - Tools 229
- Unprofessionell 46

V

- Velocity
 - des Teams 204
- Verantwortung 46
 - Arbeitsethik 53
 - Entschuldigungen 49
 - für Funktion 49
 - für Struktur 51
 - Richte keinen Schaden an 48

Verfrühte Präzisierung 135
 Versuchen 70
 Verzug 110
 vi 226
 Vorteil
 beim Test Driven Development 118

W

Wahrscheinlichkeit 174
 Waza 129
 Wideband Delphi 179
 Wissen
 Arbeitsethik 54

Z

Zeit
 zum Debuggen einplanen 108
 Zeitmanagement
 Akkus aufladen 166
 Beispiele 159
 Fokus 164
 Meetings 160
 Morast 169
 Pomodoro-Technik 167
 Sackgassen 168
 Vermeidung 168
 Zielvorgabe 64
 Zusammenarbeit 57
 Zusammengeschweißtes Team 202
 Zykluszeit
 beim TDD 116

Clean Coder

Der heiß ersehnte Nachfolger von *Uncle Bobs* hochgelobtem Buch *Clean Code*

Programmierern, die unter einem nicht nachlassenden Druck und in beständiger Ungewissheit arbeiten und trotzdem erfolgreich sind, ist eine bestimmte Eigenschaft gemeinsam: Die Praxis der Software-Entwicklung ist ihnen eine Herzensangelegenheit, für die sie sich engagiert einsetzen. Software-Entwicklung ist für sie wie eine Handwerkskunst. Das macht sie zu Profis.



Für Programmierer, die professionell arbeiten wollen

In *Clean Coder: Verhaltensregeln für professionelle Programmierer* stellt der legendäre Software-Experte Robert C. Martin die Disziplinen, Techniken, Tools und Methoden dieser wahren Handwerkskunst vor.

Dieses Buch steckt voller praktischer Ratschläge und deckt u.a. Themen wie Aufwandseinschätzung und Programmierung, Refactoring und Tests ab. Hier geht es um mehr als nur um Technik: Es geht um die innere Haltung. Martin zeigt, wie man der Software-Entwicklung mit Hochachtung begegnet, wie man gut und sauber arbeitet, wie man verlässlich kommuniziert und kalkuliert. Er beschreibt, wie man sich schwierigen Entscheidungen aufrichtig stellt und verstehen lernt, dass ein tiefes Wissen auch zum Handeln verpflichtet.

Sie erfahren:

- was es bedeutet, sich als echter Software-Profi zu verhalten
- wie Sie mit Konflikten, knappen Zeitplänen und unvernünftigen Managern umgehen
- wie Sie beim Programmieren im Fluss bleiben und Schreibblockaden umschiffen
- wie Sie mit unerbittlichem Druck umgehen und Burnout vermeiden
- wie Sie Ihre innere Haltung mit neuen Paradigmen bei der Software-Entwicklung verbinden
- wie Sie das Zeitmanagement optimieren und Sackgassen und Schlamassel vermeiden
- wie Sie für eine Umgebung sorgen, in der Programmierer und Teams wachsen und gedeihen
- wann Sie »Nein« sagen sollten – und wie Sie das anstellen

Großartige Software ist etwas Bewundernswertes:

Sie ist leistungsfähig, elegant, funktional und erfreut bei der Arbeit sowohl den Entwickler als auch den Nutzer. Hervorragende Software wird nicht von Maschinen geschrieben, sondern von Profis, die sich dieser Handwerkskunst bedingungslos verschrieben haben. Der Clean Coder hilft Ihnen, zu diesem Kreis zu gehören – und die Anerkennung zu ernten, die nur hier möglich ist.

Über den Autor

Robert C. »Uncle Bob« Martin ist seit 1970 Programmierer und bei Konferenzen in aller Welt ein begehrter Redner. Als überaus produktiver Autor hat »Uncle Bob« Hunderte von Artikeln, Abhandlungen und Blogs verfasst.

Einsteiger Fortgeschrittene Profis

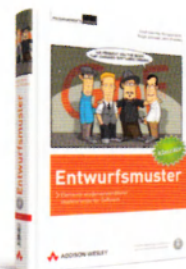
Buchtipp

Programmierung



ADDISON-WESLEY

www.addison-wesley.de



ISBN 978-3-8273-3104-5



9 783827 331045

€ 34,80 [D] € 35,80 [A]