

Robert C. Martin

# Clean Code

Refactoring, Patterns, Testen  
und Techniken für sauberen Code

Robert C. Martin Series

2. Auflage

Deutsche  
Ausgabe



# Inhaltsverzeichnis

<b>Vorwort</b> .....	<b>19</b>
<b>Einleitung</b> .....	<b>25</b>
Ein Hinweis zu älteren Kapiteln .....	27
Wie dieses Buch aufgebaut ist .....	27
<b>Einleitung (aus längst vergangenen Zeiten)</b> .....	<b>29</b>
<b>Über den Autor</b> .....	<b>31</b>
<b>1 Sauberer Code</b> .....	<b>33</b>
1.1 Code, Code und nochmals Code .....	34
1.2 Schlechter Code .....	35
1.2.1 Einstellung .....	36
1.2.2 Das grundlegende Problem .....	38
1.3 Sauberen Code schreiben – eine Kunst? .....	38
1.3.1 Was ist sauberer Code? .....	39
1.4 Das große Ganze .....	42
1.4.1 Was bringt uns sauberer Code? .....	42
1.4.2 Produktivität .....	44
1.4.3 Lebensqualität .....	48
1.5 Wir lesen mehr, als wir schreiben .....	48
1.6 Die Pfadfinder-Regel .....	50
<b>Teil I Code</b> .....	<b>51</b>
<b>2 Halten Sie Ihren Code sauber!</b> .....	<b>53</b>
2.1 Der Bereinigungsverfahren .....	65
2.2 Zusammenfassung .....	72
2.3 Postskriptum: Bob aus der Zukunft spielt mit Grok 3 .....	73
2.4 Zusammenfassung des Postskriptums .....	76
<b>3 Grundlegende Prinzipien</b> .....	<b>77</b>
3.1 Klein, aussagekräftig benannt, strukturiert und sortiert .....	78
3.1.1 Funktionen .....	78

3.2	Ein aussagekräftigeres Beispiel .....	80
3.2.1	Unabhängige Auslieferbarkeit .....	106
3.3	Abschließende Bemerkungen .....	107
<b>4</b>	<b>Aussagekräftige Namen .....</b>	<b>109</b>
4.1	Zweckbeschreibende Namen wählen .....	110
4.1.1	Ein Namenssystem aufbauen .....	111
4.1.2	Fehlinformationen vermeiden .....	113
4.1.3	Unterschiede deutlich machen .....	114
4.1.4	Aussprechbare Namen verwenden .....	115
4.1.5	Suchbare Namen verwenden .....	116
4.1.6	Namen von sinnvoller Länge verwenden .....	118
4.1.7	Codierungen vermeiden .....	119
4.1.8	Passende Wortarten verwenden .....	121
4.1.9	Schlüsselwort-Parameter beachten .....	123
4.1.10	Keine lustig gemeinten Namen verwenden .....	123
4.1.11	Ein Wort pro Konzept wählen .....	123
4.1.12	Namen der Lösungsdomäne verwenden .....	124
4.1.13	Namen der Problemdomäne verwenden .....	124
4.1.14	Sinnstiftenden Kontext hinzufügen .....	125
4.1.15	Keinen überflüssigen Kontext hinzufügen .....	127
4.2	Abschließende Worte .....	128
<b>5</b>	<b>Kommentare .....</b>	<b>129</b>
5.1	Unser Scheitern kaschieren .....	129
5.1.1	Ausgeblendete oder verborgene Kommentare .....	130
5.1.2	Lügende Kommentare .....	131
5.1.3	Insider-Kommentare .....	132
5.1.4	Kommentare sind kein Ersatz für guten Code .....	132
5.1.5	Erklären Sie Ihre Intention durch den Code .....	132
5.2	Gute Kommentare .....	133
5.2.1	Juristische Kommentare .....	133
5.2.2	Informierende Kommentare .....	133
5.2.3	Erklärung der Intention .....	134
5.2.4	Klarstellungen .....	136
5.2.5	Warnungen vor Konsequenzen .....	137
5.2.6	Verstärkung .....	138
5.2.7	Javadocs (und Verwandte) in öffentlichen APIs .....	138
5.3	Schlechte Kommentare .....	138
5.3.1	Selbstgespräche .....	138
5.3.2	Redundante Kommentare .....	140
5.3.3	Irreführende Kommentare .....	140

5.3.4	Redundanz und Ungenauigkeit .....	141
5.3.5	Vorgeschriebene Kommentare .....	144
5.3.6	Tagebuch-Kommentare .....	144
5.3.7	Geschwätz .....	145
5.3.8	Grauenvolles Geschwätz .....	148
5.3.9	TODO-Kommentare .....	148
5.3.10	Funktionen oder Variablen anstelle von Kommentaren verwenden .....	149
5.3.11	Positionsbezeichner .....	149
5.3.12	Zuschreibungen und Nebenbemerkungen .....	150
5.3.13	Auskommentierter Code .....	150
5.3.14	HTML-Kommentare .....	151
5.3.15	Nicht lokale Informationen .....	152
5.3.16	Zu viele Informationen .....	152
5.3.17	Unklarer Zusammenhang .....	153
5.3.18	Funktionskopf-Kommentare .....	153
5.3.19	Javadocs in nicht öffentlichem Code .....	154
5.3.20	Beispiel .....	154
5.4	Zusammenfassung .....	158
<b>6</b>	<b>Formatierung .....</b>	<b>159</b>
6.1	Der Zweck der Formatierung .....	160
6.2	Vertikale Formatierung .....	160
6.2.1	Vertikaler Weißraum zwischen Konzepten .....	162
6.2.2	Vertikale Dichte .....	163
6.2.3	Vertikaler Abstand .....	164
6.2.4	Variablendeklarationen .....	165
6.2.5	Abhängige Funktionen .....	167
6.2.6	Konzeptionelle Affinität .....	169
6.3	Horizontale Formatierung .....	170
6.3.1	Horizontaler Weißraum und Dichte .....	171
6.3.2	Horizontale Ausrichtung .....	173
6.3.3	Einrückung .....	174
6.3.4	Einrückungsregeln brechen .....	176
6.4	Team-Regeln .....	177
6.5	Uncle Bobs Formatierungsregeln .....	177
<b>7</b>	<b>Saubere Funktionen .....</b>	<b>181</b>
7.1	Klein! .....	182
7.1.1	Wohlgeschriebene Prosa .....	183
7.1.2	Eine Abstraktionsebene pro Funktion .....	183

7.2	Code von oben nach unten lesen: Stepdown-Regel .....	185
7.2.1	Kuddelmuddel .....	187
7.3	Switch-Anweisungen .....	187
7.4	Saubere Funktionen: Eine genauere Betrachtung .....	189
7.4.1	Kontextbezogen .....	189
7.4.2	Benennbar .....	190
7.4.3	Isoliert .....	194
7.4.4	Homogen .....	197
7.4.5	Rein .....	199
7.5	Zusammenfassung .....	203
<b>8</b>	<b>Heuristiken für Funktionen .....</b>	<b>205</b>
8.1	Funktionsargumente .....	205
8.1.1	Variadische Argumente .....	207
8.1.2	Mehr als drei Argumente? .....	207
8.1.3	Schlüsselwort-Argumente .....	207
8.1.4	Flag-Argumente .....	208
8.1.5	Output-Argumente .....	209
8.1.6	Fehlercodes .....	210
8.2	Anweisung und Abfrage trennen .....	211
8.3	Exceptions sind besser als Fehlercodes .....	212
8.3.1	Auf eigene Gefahr! .....	213
8.3.2	try/catch-Blöcke extrahieren .....	214
8.3.3	Fehler-Handling ist eine Aufgabe .....	215
8.3.4	Der Abhängigkeitsmagnet von Fehlercodes .....	215
8.4	DRY: Don't Repeat Yourself .....	216
8.4.1	Einfacher wiederholter Code .....	217
8.4.2	Ähnlicher Code .....	218
8.4.3	Schleifen-Duplizierung .....	222
8.4.4	Zufällige und essenzielle Duplizierung .....	225
8.5	Nebeneffekte .....	226
8.5.1	Wir sind nicht gut darin .....	227
8.5.2	Funktionale Programmiersprachen .....	228
8.5.3	Objektorientierte Sprachen .....	229
8.6	Strukturierte Programmierung .....	231
8.6.1	Sequenzen .....	232
8.6.2	Selektionen .....	232
8.6.3	Iterationen .....	232
8.7	Wer soll sich das alles merken? .....	234
8.8	Zusammenfassung .....	234

<b>9</b>	<b>Die saubere Methode</b> .....	<b>237</b>
9.1	Mach es richtig .....	238
9.2	Beispiel .....	240
9.2.1	Design und Architektur berücksichtigen .....	257
9.3	Zusammenfassung .....	265
<b>10</b>	<b>Eine Aufgabe</b> .....	<b>267</b>
10.1	Refactoring per Extract-Methode .....	268
10.1.1	Widerstand ist zwecklos! .....	270
10.2	Was sind große Funktionen überhaupt? .....	274
10.3	Extraktion und Klassen .....	293
10.4	Zusammenfassung .....	298
<b>11</b>	<b>Eine Frage des Respekts</b> .....	<b>299</b>
11.1	Die Zeitungsmetapher .....	301
11.1.1	Respektvoll schreiben .....	302
11.2	Die Stepdown-Regel: »Hello again« .....	304
11.3	Die Abstraktions-Achterbahn .....	304
11.4	Schreiben und Lesen funktionieren unterschiedlich .....	305
<b>12</b>	<b>Objekte und Datenstrukturen</b> .....	<b>307</b>
12.1	Was ist ein Objekt? .....	308
12.2	Datenabstraktion .....	309
12.3	Daten/Objekt-Antisymmetrie .....	311
12.4	Das Gesetz von Demeter .....	314
12.4.1	Train Wreck .....	314
12.4.2	Hybride .....	315
12.4.3	Struktur verbergen .....	316
12.5	Datentransfer-Objekte .....	317
12.5.1	Das objekt-relationale »Impedance Mismatch« .....	318
12.5.2	Objekte und Datenstrukturen verwenden .....	319
12.5.3	switch-Anweisungen .....	320
12.5.4	Die objektorientierte Lösung .....	323
12.5.5	Ruhig, Brauner .....	324
12.6	Kompromiss zwischen objektorientierten und prozeduralen Konzepten .....	325
12.7	Aber wie sieht es mit der Geschwindigkeit aus? .....	326
12.8	Zusammenfassung .....	326

<b>13</b>	<b>Saubere Klassen</b> .....	<b>327</b>
13.1	Klassen und Module im Vergleich zu Dateien .....	327
13.2	Was sollte eine Klasse enthalten? .....	328
13.2.1	Klassendesign auf den Punkt gebracht .....	329
13.2.2	Heuristiken und Charakteristiken .....	330
13.2.3	Wann ist eine Klasse zu groß? .....	332
13.2.4	Policys im Code .....	334
13.2.5	Wo sich Gründe für Veränderungen verbergen .....	335
13.2.6	Die Lösung .....	336
13.2.7	Eine übermäßig offene Implementierung .....	338
13.2.8	Jetzt handeln oder abwarten? .....	340
13.2.9	Und was jetzt? .....	340
13.3	Geschlossene, kohäsive Single-Responsibility-Klassen .....	344
13.3.1	Wenn sich Policys ändern .....	346
13.3.2	Ist das Overengineering? .....	347
13.3.3	Einfacheres Testen .....	348
13.4	Bühne frei für die KI .....	349
13.4.1	Fehler sind unvermeidlich .....	349
<b>14</b>	<b>Testdisziplinen</b> .....	<b>351</b>
14.1	Disziplin 1: Test-Driven Development (TDD) .....	353
14.1.1	Die drei Gesetze des TDD .....	353
14.2	Disziplin 2: Test && Commit    Revert (TCR) .....	354
14.3	Disziplin 3: Small Bundles .....	355
14.4	Design .....	355
14.5	Disziplin .....	356
14.5.1	Lästig, langweilig und langsam .....	356
14.5.2	Debuggen .....	357
14.5.3	Dokumentation .....	357
14.5.4	Zuverlässigkeit .....	358
14.5.5	Design .....	359
14.5.6	Reprise .....	359
14.5.7	Engelchen und Teufelchen .....	360
14.5.8	Das Teufelchen mundtot machen .....	361
14.5.9	Komplikationen und Schlupflöcher .....	361
14.5.10	Kosten und Konsequenzen .....	363
14.6	Tests sauber halten .....	363
14.7	Tests ermöglichen die »-keiten« .....	364

<b>15</b>	<b>Saubere Tests</b> .....	<b>367</b>
15.1	Domänenspezifische Testsprache .....	371
15.1.1	Zusammengesetzte Assertions .....	371
15.1.2	Zusammengesetzte Testergebnisse .....	371
15.1.3	Ein Doppelstandard .....	373
15.1.4	Single-Assert-Regel .....	374
15.1.5	Single-Act-Regel .....	374
15.2	F.I.R.S.T. ....	375
15.2.1	Fast (Schnell) .....	375
15.2.2	Isolated (Isoliert) .....	375
15.2.3	Repeatable (Wiederholbar) .....	375
15.2.4	Self-Validating (Selbstvalidierend) .....	375
15.2.5	Timely (Zeitnah) .....	376
15.3	Testdesign .....	376
15.4	Zusammenfassung .....	376
<b>16</b>	<b>Akzeptanztests</b> .....	<b>377</b>
16.1	Die Akzeptanztest-Disziplin .....	378
16.1.1	Die Disziplin .....	379
16.1.2	Kontinuierlicher Build .....	380
16.2	Zusammenfassung .....	381
<b>17</b>	<b>KI, LLMs und wie sie alle heißen</b> .....	<b>383</b>
17.1	Programmieren per Prompt .....	385
17.1.1	In den Kinderschuhen .....	391
17.1.2	Wilde wissenschaftliche Vermutung .....	391
17.2	Zusammenfassung .....	396
<b>Teil II</b>	<b>Design</b> .....	<b>397</b>
<b>18</b>	<b>Einfaches Design</b> .....	<b>399</b>
18.1	YAGNI .....	401
18.2	Abgedeckt durch Tests .....	401
18.2.1	Ein asymptotisches Ziel .....	402
18.2.2	Design? .....	402
18.3	Aussagekraft maximieren .....	403
18.3.1	Die zugrunde liegende Abstraktion .....	404
18.3.2	Tests: Die zweite Hälfte des Problems .....	406
18.4	Duplizierung minimieren .....	406
18.4.1	Zufällige Duplizierung .....	407
18.4.2	Größe minimieren .....	408
18.4.3	Einfaches Design .....	408

<b>19</b>	<b>Die SOLID-Prinzipien</b> .....	<b>409</b>
19.1	SRP: Das Single-Responsibility-Prinzip .....	411
19.1.1	Zufällige Duplizierung .....	412
19.1.2	Lösungen .....	414
19.1.3	Höhere Ebenen .....	415
19.2	OCP: Das Open-Closed-Prinzip .....	415
19.2.1	Ein Gedankenexperiment .....	416
19.2.2	Richtungssteuerung .....	419
19.2.3	Information Hiding .....	420
19.2.4	Zusammenfassung .....	420
19.3	LSP: Das Liskov'sche Substitutionsprinzip .....	420
19.3.1	LSP und Softwaredesign .....	422
19.3.2	Taxi-Aggregator .....	422
19.4	ISP: Das Interface-Segregation-Prinzip .....	424
19.4.1	ISP und Programmiersprachen .....	425
19.4.2	ISP und Softwaredesign .....	425
19.5	DIP: Das Dependency-Inversion-Prinzip .....	426
19.5.1	Stabile Abstraktionen .....	429
19.5.2	Factorys .....	431
19.5.3	Konkrete Komponenten .....	432
<b>20</b>	<b>Komponentenprinzipien</b> .....	<b>433</b>
20.1	Komponenten .....	433
20.2	Eine kurze Historie der Komponenten .....	434
20.2.1	Relokatierbarkeit .....	437
20.2.2	Linker .....	437
20.3	Komponentenkohäsion .....	439
20.3.1	Das Reuse/Release-Equivalence-Prinzip (REP) .....	440
20.3.2	Das Common-Closure-Prinzip (CCP) .....	441
20.3.3	Das Common-Reuse-Prinzip (CRP) .....	442
20.3.4	Das Spannungsdiagramm für die Komponentenkohäsion .	444
20.3.5	Zusammenfassung .....	445
20.4	Komponentenkopplung .....	446
20.4.1	Das Acyclic-Dependencies-Prinzip (ADP) .....	446
20.4.2	Das Stable-Dependencies-Prinzip (SDP) .....	452
20.4.3	Das Stable-Abstractions-Prinzip (SAP) .....	458
20.5	Zusammenfassung .....	463
<b>21</b>	<b>Kontinuierliches Design</b> .....	<b>465</b>
21.1	Kontinuierliche Veränderung .....	466
21.2	Kontinuierliches Design .....	468

21.3	Die vier K des kontinuierlichen Designs .....	468
21.3.1	Klarheit .....	469
21.3.2	Kürze .....	476
21.3.3	Konfirmierung .....	486
21.3.4	Kohäsion .....	497
21.4	Wann sind wir außerdem noch Designer? .....	501
21.4.1	Vorabdesign .....	501
21.4.2	Auf die Plätze! .....	502
21.4.3	Fertig! .....	503
21.4.4	Los! .....	503
<b>22</b>	<b>Nebenläufigkeit .....</b>	<b>505</b>
22.1	Warum Nebenläufigkeit? .....	506
22.1.1	Mythen und falsche Vorstellungen .....	507
22.1.2	Herausforderungen .....	508
22.2	Prinzipien zur Absicherung von Nebenläufigkeit .....	509
22.2.1	Single-Responsibility-Prinzip .....	509
22.2.2	Korollar: Beschränken Sie den Gültigkeitsbereich von Daten .....	510
22.2.3	Korollar: Arbeiten Sie mit Kopien der Daten .....	510
22.2.4	Korollar: Threads sollten voneinander so unabhängig wie möglich sein .....	511
22.2.5	Lernen Sie Ihre Programmiersprache und Bibliothek kennen .....	511
22.2.6	Threadsichere Collections .....	511
22.3	Lernen Sie Ihre Ausführungsmodelle kennen .....	512
22.3.1	Erzeuger-Verbraucher .....	513
22.3.2	Leser-Schreiber .....	513
22.3.3	Philosophenproblem .....	514
22.4	Achten Sie auf Abhängigkeiten zwischen synchronisierten Methoden .....	514
22.5	Halten Sie synchronisierte Abschnitte klein .....	515
22.6	Korrekten Startup- und Shutdown-Code zu schreiben, ist schwer ..	515
22.7	Threaded-Code testen .....	516
22.7.1	Behandeln Sie gelegentlich auftretende Fehler als potenzielle Threading-Probleme .....	517
22.7.2	Bringen Sie erst den Nonthreaded-Code zum Laufen .....	517
22.7.3	Machen Sie Ihren Threaded-Code austauschbar .....	518
22.7.4	Schreiben Sie anpassbaren Threaded-Code .....	518
22.7.5	Den Code mit mehr Threads als Prozessoren ausführen ..	518
22.7.6	Den Code auf verschiedenen Plattformen ausführen .....	518
22.7.7	Code instrumentieren, um Fehler zu provozieren .....	519

22.8	Nachtrag im Jahr 2025: Neue Erkenntnisse aus der Praxis .....	522
22.8.1	Datenintegrität .....	522
22.9	Zusammenfassung .....	527
<b>Teil III Architektur .....</b>		<b>529</b>
<b>23</b>	<b>Die Geschichte zweier Werte .....</b>	<b>531</b>
23.1	Optionen offenhalten .....	531
<b>24</b>	<b>Unabhängigkeit .....</b>	<b>535</b>
24.1	Use Cases .....	535
24.2	Betrieb .....	536
24.3	Entwicklung .....	537
24.4	Deployment .....	537
24.5	Optionen offenhalten .....	537
<b>25</b>	<b>Architekturgrenzen .....</b>	<b>539</b>
25.1	Welche Grenzen sollten Sie ziehen – und wann? .....	540
25.2	Plugin-Architektur .....	542
25.3	Fallstudie: FitNesse .....	543
25.4	Zusammenfassung .....	545
<b>26</b>	<b>Saubere Grenzen .....</b>	<b>547</b>
26.1	IoT-Framework vom Drittanbieter: Jede Menge Grenzen .....	548
26.2	Grenze zwischen UI und Anwendung .....	553
26.2.1	SOLID und hexagonale Architektur .....	556
26.2.2	Grenzen erforschen und kennenlernen .....	557
26.2.3	Code verwenden, der noch nicht existiert .....	560
26.3	Saubere Grenzen .....	562
<b>27</b>	<b>Saubere Architektur .....</b>	<b>563</b>
27.1	Die Abhängigkeitsregel .....	564
27.1.1	Entitäten .....	565
27.1.2	Use Cases .....	566
27.1.3	Schnittstellenadapter .....	566
27.1.4	Frameworks und Treiber .....	567
27.1.5	Nur vier Kreise? .....	567
27.1.6	Grenzen überschreiten .....	567
27.1.7	Welche Daten überschreiten die Grenzen? .....	568
27.2	Ein typisches Beispiel .....	568
27.3	Zusammenfassung .....	570

<b>Teil IV Craftsmanship</b> .....	<b>571</b>
IV.1 »Eine große Anzahl« .....	572
IV.2 Acht Jahrzehnte .....	573
IV.2.1 Nerds und Helden .....	577
IV.2.2 Berühmt und berüchtigt .....	577
IV.2.3 Vorbilder und Schurken .....	579
IV.2.4 Wir regieren die Welt .....	580
IV.2.5 Katastrophen .....	581
IV.3 Der Eid .....	583
<b>28 Schaden</b> .....	<b>585</b>
28.1 Gesellschaftlicher Schaden .....	586
28.2 Funktionsbeeinträchtigungen .....	587
28.3 Schädigung der Struktur .....	590
28.4 Soft .....	591
28.5 Tests .....	592
<b>29 Weder im Verhalten noch in der Struktur fehlerhaft</b> .....	<b>595</b>
29.1 Mach es richtig .....	596
29.1.1 Was ist gute Struktur? .....	597
29.1.2 Eisenhower-Matrix .....	598
29.2 Programmierer sind Stakeholder .....	599
29.3 Ihr Bestes geben .....	601
<b>30 Reproduzierbarer Beweis</b> .....	<b>603</b>
30.1 Dijkstra .....	603
30.1.1 Beweis der Korrektheit .....	604
30.2 Strukturierte Programmierung .....	606
30.3 Funktionale Dekomposition .....	608
30.4 Test-Driven Development und mehr .....	609
<b>31 Kurze Zyklen</b> .....	<b>613</b>
31.1 Die Geschichte der Versionsverwaltung .....	613
31.1.1 Lochkarten .....	613
31.1.2 Kontinuierliche Integration .....	618
31.1.3 Kurze Zyklen .....	619
31.2 Kontinuierliche Integration .....	620
31.3 Branches versus Toggles .....	620
31.4 Kontinuierliches Deployment .....	622
31.5 Kontinuierlicher Build .....	623

<b>32</b>	<b>Unablässliche Verbesserung</b> .....	<b>625</b>
32.1	Testabdeckung .....	625
32.2	Mutationstests .....	626
32.3	Semantische Stabilität .....	627
32.4	Aufräumen .....	627
32.5	Kreationen .....	628
<b>33</b>	<b>Hohe Produktivität beibehalten</b> .....	<b>629</b>
33.1	Viskosität .....	629
33.1.1	Build erstellen .....	630
33.1.2	Testen .....	630
33.1.3	Debugging .....	631
33.1.4	Deployment .....	631
33.2	Ablenkungen .....	632
33.2.1	Meetings .....	632
33.2.2	Musik .....	633
33.2.3	Stimmung .....	633
33.2.4	Der Flow .....	634
33.3	Zeitmanagement .....	635
<b>34</b>	<b>Teamarbeit</b> .....	<b>637</b>
34.1	Kollaborative Programmierung .....	637
34.2	Offenes/virtuelles Büro .....	638
<b>35</b>	<b>Ehrliche und faire Schätzungen</b> .....	<b>641</b>
35.1	Lügen .....	641
35.2	Ehrlichkeit, Genauigkeit, Präzision .....	642
35.3	Lehren aus meiner Praxis .....	643
35.3.1	Geschichte 1: Vektoren .....	643
35.3.2	Geschichte 2: pCCU .....	645
35.3.3	Die Lehre .....	646
35.4	Genauigkeit und Präzision .....	646
35.5	Aggregation .....	648
35.6	Ehrlichkeit .....	648
35.7	Druck .....	650
<b>36</b>	<b>Respekt gegenüber Mitprogrammierern</b> .....	<b>651</b>
<b>37</b>	<b>Niemals aufhören zu lernen</b> .....	<b>653</b>

<b>Nachwort</b> .....	<b>655</b>
<b>Bibliografie</b> .....	<b>661</b>
<b>A Die Clean-Code-Debatte</b> .....	<b>665</b>
A.1 Einführung .....	665
A.2 Methodenlänge .....	667
A.2.1 Zusammenfassung zur Methodenlänge .....	681
A.3 Kommentare .....	682
A.3.1 Zusammenfassung zu Kommentaren .....	696
A.3.2 Johns Neufassung von PrimeGenerator .....	697
A.3.3 Eine Geschichte von zwei Programmierern .....	703
A.3.4 Bobs Neufassung von PrimeGenerator2 .....	706
A.4 Test-Driven Development .....	710
A.4.1 TDD-Zusammenfassung .....	722
A.5 Abschließende Bemerkungen .....	722
<b>Stichwortverzeichnis</b> .....	<b>725</b>



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

## 1.1 Code, Code und nochmals Code

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

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

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

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

---

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

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

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

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

## 1.2 Schlechter Code

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



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

---

3 [IMP]

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

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

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

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

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

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

## 1.2.1 Einstellung

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

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

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

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

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

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

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

---

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

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

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

## 1.2.2 Das grundlegende Problem

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

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

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

*Der beste Weg zu schneller Arbeit ist sorgfältige Arbeit.*

## 1.3 Sauberen Code schreiben – eine Kunst?

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

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

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

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

*Zunächst bring es zum Laufen. Dann mach es richtig.*

### 1.3.1 Was ist sauberer Code?

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



»Sauberer Code erledigt eine Aufgabe gut.«

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

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

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



»Sauberer Code liest sich wie wohlgeschriebene Prosa.«

– Grady Booch, bekannter Autor vieler Bücher  
zum Thema Softwareentwicklung

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

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



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

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

---

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

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

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



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

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

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

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



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

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

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

## 1.4 Das große Ganze

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

### 1.4.1 Was bringt uns sauberer Code?

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

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

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

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

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

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

*Sie und ich, wir regieren die Welt.*

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

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

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

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

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

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

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

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

Auch das ist bereits geschehen. Mehrfach. Zu oft.

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

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

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

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

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

## **1.4.2 Produktivität**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

# Stichwortverzeichnis

## Symbole

49FNGO 56, 61

.NET

    Komponenten 433

## A

Abdeckung

    Tests 625

Abfragen 308

abgeleitete Klassen 192

    instanciieren 103

abhängige Funktionen 167

Abhängigkeiten 87, 95, 99, 597, 598

    Abhängigkeitszyklen abschaffen 446

    Architektur 546

    entkoppeln 446

    invertieren 427

    isolierte Tests 375

    Komponenten 442

    konkret und flüchtig 428, 452

    Quellcode 417, 418, 424, 427, 431

    synchronisierte Methoden 514

    transitiv 420

    von Anbietern 557

Abhängigkeitsmagneten 215

Abhängigkeitsregel 262, 432, 546, 564, 568

Ablauf

    Code schreiben und prüfen 238

Ablenkung

    ausmerzen 632

    durch Musik 633

    Kommentare 130

Abschaltung

    kontrolliert 515

Absicht 61, 134

    Aussagekraft maximieren 403

Abstand 164

Abstract Factory 121, 188, 430, 431

Abstraktionen 400

    algorithmische Beweise 604

    Bedeutung 688

    Datenstrukturen 309

    Definition 688, 689

    Duplizierungen reduzieren mittels 482

    Funktionsnamen 300

    hexagonale Architektur 556

    stabil 428

    Stable-Abstractions-Prinzip (SAP) 458

    Stepdown-Regel 301, 304

    und Aussagekraft 404

    und Kommentare 688, 693, 696

Abstraktions-Achterbahn 304

Abstraktionsebenen 183, 186, 304

Active X 439

Acyclic-Dependencies-Prinzip (ADP) 446

Affinität

    konzeptionelle 169

Affordable Care Act 586

Agile Software Development, Principles, Pat-

    terns, and Practices (Buch) 154

agile Softwareentwicklung 352, 353, 380,

    638

ähnlicher Code 218

Akzeptanztests 377, 543

ALGOL 271, 573

Algorithmen

    Beweis der Korrektheit 604

    klärende Kommentare 693

    verstehen 676, 703, 704

Algorithms + Data Structures = Programs

    (Buch) 307

Analog/Digital-Wandler (ADC) 551

Änderungen

    an Datenstrukturen 99, 261

    an Grenzen 562

    im System während des Lebenszyklus

    538

- kontinuierlich 466
  - Minimierung durch gute Architektur 416
  - Softwareentwicklung 596
  - Stabilität und Widerstand gegen 452
  - TDD und kleine Änderungen 549
  - tolerieren (SOLID) 410
  - und Lern-Tests 560
  - Anforderungen
    - spezifizieren 34
  - Anforderungsspezifikation 35
  - Ant 161
  - Anti-Idiome 484
  - Anweisungen
    - Kommentare sind unnötig 479
    - und Abfrage trennen 211
  - Anwendungen
    - Abhängigkeitsmanagement 451
  - Anwendungsfälle (Softwarearchitektur) 535, 566
  - A Philosophy of Software Design (Buch) 300, 665, 711, 712
  - APIs
    - Drittanbieter 558
    - öffentlich 138, 154
    - Swing 362
    - Tests 371
    - undefiniert 561
  - Architektur 261
    - Achse der Veränderung 415
    - hexagonal 556
    - Plugins 542
    - sauber 563
    - SOLID-Designprinzipien 330, 331, 409
    - SOLID-Prinzipien 407
    - Trennung der Systemaspekte 680
    - Trennung der Zuständigkeiten 563
    - und das OCP 416
    - Zweck 530, 545
  - Architekturgrenzen 97, 261, 293, 415, 431, 539
    - Abhängigkeitsregel 546
    - Änderungen an 562
    - Bekanntes von Unbekanntem trennen 560
    - Hardwaregrenzen 361
    - IoT-Software 548
    - und das DIP 427
    - zwischen UI und Anwendung 553
  - A Relational Model of Data for Large Shared Databanks (Artikel) 308
  - Argumente 245
    - Deklarationen 206
    - einer Funktion 194
    - explizit und implizit 206
    - Flags 208
    - Funktionen 205
    - Hash-Maps 208
    - in Objekten 206
    - in verschiedenen Programmiersprachen 206
    - Liste 205
    - mehr als drei 207
    - Output 209
    - Schlüsselwörter 207
    - sind Kopplungen 206
    - überladene Konstruktoren 122
    - variadisch 207
  - Arrange/Act/Assert-Pattern (AAA) 370, 374, 378
  - Assembler (Programmiersprache) 173
  - Assertions 255, 280, 371
  - Aufgaben
    - aus Funktionsnamen ersichtlich 300
  - Aufgabenliste 149
  - Aufrufe 194
  - Aufzählungen
    - algorithmische Beweise 604
    - goto-Anweisungen 607
  - Ausführung
    - linear 232
  - auskommentierter Code 150
  - Auslieferung 377, 537, 622
    - unabhängig 106, 433, 644
  - Ausrichtung
    - horizontal 173
  - Aussagekraft
    - von Code 403
  - aussagekräftige Namen 78
  - Ausschlusszonen 460
  - Automated Computing Engine (ACE) 572
  - Auto-Vervollständigung 113, 127
  - Axis of Change 261
- B**
- Backus, John 384, 573
  - Banner 149
  - BASIC 120

- Beans 317
- Beck, Kent 39, 154, 182, 238, 330, 331, 353, 354, 399, 400, 406, 408
  - Aussagekraft von Code 406
  - Designprinzipien 35
  - YAGNI 401
- Behavior-Driven Development (BDD) 378
- Beispielcode 714
- Benutzeroberfläche
  - Testen in der 280
- Bereinigen
  - Angst 360, 486
  - Anleitung 239
  - Code zum Umwandeln römischer Zahlen 54
  - Schritt für Schritt 67
- beschreibende Namen *siehe* sprechende Namen
- Betrieb (Softwarearchitektur) 536
- Beziehungen
  - Eltern–Kind 526
  - Nähe 679
  - Vererbungen und Implementierungen 541
  - wechselseitig 258
- Bibliotheken 435, 527
- Bildschirmseite 182
- Bindelader 437
- BitKeeper 618
- Bobolia 240
- Bonus-Code 95
- Booch, Grady 40, 78
- boole'sche Ausdrücke 483
- Boundary–Control–Entity (BCE)
  - Architektur 563
- bound Resources 512
- Boyce, Raymond 308
- Branches und Toggles 620
- Browser
  - Komponenten-Cache 107
- Build
  - kontinuierlich 380, 623
- Büros
  - offen und virtuell 638
- C**
- C (Programmiersprache) 190, 271, 575
  - sauberer Code in 51
- C# 166, 207, 307
  - Konstruktoren 122
  - Namen für Eigenschaften 119
- C++ 166, 307
  - Erfinder 39
  - Konstruktoren 122
  - Namen für Variablen 119
- CatalogItem (Typ) 89
- catch-Blöcke 139, 146
- checkForIllegalPrefixCombinations (Funktion) 67
- checkForImproperRepetitions (Funktion) 67
- Chirurgie mit der Schrotflinte 331
- Classitis 261
- Clean Agile (Buch) 641
- Clean Architecture (Buch) 397, 400, 409, 433, 530, 598
- Clean Code *siehe* sauberer Code
  - Definition 33, 39, 329
  - gute Gründe 44
- Clean Code (Buch, erste Ausgabe) 522, 655, 657, 665, 681, 699
- Clean Craftsmanship (Buch) 354, 362, 376, 397, 399, 571
- Clojure 123, 167, 208, 228, 273, 297, 302
  - sauberer Code in 51
- COBOL 271, 392, 573
- Cockburn, Alistair 556, 563
- Codd, Edgar F. 217, 308
- Code
  - Ähnlichkeit 218
  - als Ausdruckssprache 683
  - anstelle von Kommentaren 132
  - ausführbar 84
  - auskommentiert 150
  - aussagekräftige Namen 78
  - bearbeiten 61
  - Becks Gesetz 39, 54, 234, 238, 305, 595, 599
  - bereinigen 65, 238, 239, 293
  - Duplizierungen 217
  - eine Aufgabe 39, 78, 267, 331, 411
  - Einheitlichkeit 160
  - Formatierung 160
  - Genauigkeit 143
  - idiomatisch 484
  - immer in Mode 34
  - kleine Elemente 78
  - Kontext 111

- kontinuierliche Änderungen 131, 152
  - Kosten für die Programmierung 238
  - Lesbarkeit 61, 78, 120, 128, 160, 302, 305, 470, 676
  - lesen 61
  - lesen, um ihn zu verstehen 704
  - lesen und schreiben 305
  - nachvollziehbar 78
  - Nebenläufigkeit 506
  - Nervfaktor 61
  - pluggable 517
  - reproduzierbarer Beweis 603, 610
  - Robustheit 85
  - schreiben und bearbeiten 53, 58, 128, 234
  - Struktur 160
  - Testabdeckung 401
  - Tests für flexiblen und wiederverwendbaren Code 364
  - Verständlichkeit 61
  - visuelles Layout 162
  - Wartung 41
  - Wiederholungen 217
  - Code bearbeiten
    - Angst 486
    - für mehr Klarheit 472
    - Kommentare 129
    - Zeitaufwand 58
  - Code bereinigen
    - Beispiel 240
    - vorgreifend 340
  - Code Smells 331
    - Feature-Neid 337, 338
    - Primitive Obsession 125
  - Code That Fits in Your Head (Buch) 42
  - Code verstehen
    - wesentliche Prinzipien 78
  - Codierungen 119
  - Coding-Stil 177
  - Collections
    - threadsicher 511
  - COLT (Central Office Line Tester) 643
  - Command Query Separation (CQS) 211, 213, 317
  - Commit
    - und Revert 355
  - Common-Closure-Prinzip (CCP) 415, 441, 444
  - Common-Reuse-Prinzip (CRP) 426, 442
  - Communications of the ACM (Zeitschrift) 231
  - Compiler 573
    - Geschichte 435
    - Kontext 125
    - Linking 437
  - Computer
    - Geschichte 573
  - Concurrent Programming in Java (Buch) 511
  - convert (Funktion) 69
  - convertLettersToNumbers (Funktion) 62
  - Copilot 54, 58, 265
  - Coplien, James O. 505, 563
  - Copy & Paste 406
  - Copyright 133
  - CountDownLatch 512
  - CQS 211
  - Cunningham, Ward 41, 192, 299, 543, 656, 657
  - CVS (Concurrent Versions System) 617, 619, 620
- ## D
- DAG, Directed Acyclic Graph 448
  - Dahl, Ole-Johan 189, 307, 384, 575
  - Data Context Interaction (DCI)
    - Architektur 563
  - Date, Chris 308
  - Dateien
    - und Klassen und Module 327
    - vertikale Größe 160
  - Daten
    - gemeinsam genutzt 510
    - gemeinsam nutzen 526
  - Datenbanken 532
    - Architekturgrenzen 540, 545, 568
    - Grenzen überschreiten 568
    - objektorientiert 318
    - relational 308
    - statisch 319
  - Datenintegrität 522, 525
  - Datenstrukturen 99, 308
    - Abstraktionen 309
    - Änderungen 99, 261
    - Datentransfer-Objekte 317
    - eine Aufgabe 411
    - Gesetz von Demeter 314
    - im Gegensatz zu Objekten 311, 316
    - Open-Closed-Prinzip 85

- relationale Datenbanken 308
- switch-Anweisungen 85
- und die SOLID-Prinzipien 409
- und Objekte 315
- verbergen 261
- Deadlock 513
- Debets, Maria 603
- Debuggen
  - Geschwindigkeit 631
  - KI-Prompts 395
  - Kommentare 151, 705
  - Threaded-Code 517
  - weniger Aufwand dank Tests 357
- Debugging
  - beim Säubern 58
  - TDD 251, 713, 714
- Deklarationen 327
- Demeter, Gesetz von *siehe* Gesetz von Demeter
- Dependency-Injection-Framework 532
- Dependency-Inversion-Prinzip (DIP) 87, 103, 261, 322, 324, 326, 400, 426, 546
  - Abhängigkeitszyklen abschaffen 449
  - Grenzen überschreiten 567
  - Übersicht 411
- Deployment
  - Geschwindigkeit 631
- Design
  - aufschieben 718
  - Build-Geschwindigkeit 630
  - die vier K 468, 501
  - einfach 399
  - Ideen überprüfen 666
  - Klassen 258
  - kontinuierlich 338, 380, 465
  - modular 667
  - nicht testbaren Code eliminieren 361
  - objektorientiert oder relational 308
  - Objektorientierung vs. Datenstrukturen 311
  - OCP als treibende Kraft 420
  - Perspektivwechsel 713
  - Prinzipien für einfaches Design 468
  - Prioritäten ermitteln 709
  - Projektentwicklung 501
  - starres 321
  - TDD 711, 713, 715, 722
  - und Abstraktionen 688
  - und das DIP 427
  - und das ISP 425
  - und das LSP 422
  - und Testdisziplinen 355, 359
  - verständlich 677
- Design Pattern 41
  - Singleton 134
- Design Patterns (Buch) 225
- Designperspektiven 331
- Designprinzipien
  - SOLID 330
- Desk-Check 238
- Details
  - Software 532
- Dichte 163
- digitale Vermittlung 645
- Digrafen 67
- Dijkstra, Edsger 231, 574, 575, 603
- DIP
  - für Komponenten 459
- Direktionalität 418, 419
- domänenspezifische Testsprache 371
- Don't Repeat Yourself (DRY) 216, 317
- Dringlichkeit und Wichtigkeit 598
- Drittanbieter-Code
  - testen 557
- Drittanbieter-Frameworks
  - Grenzen 548
  - IoT 548
  - unpraktische Tests 362
- DRY *siehe* Don't Repeat Yourself (DRY)
- DTO
  - Daten-Transferobjekte 317
- Duck-Typing 216
- Duplikat 407
  - zufällig und erforderlich 407
- Duplizierungen 216, 330
  - bereinigen 217
  - Definition 217
  - minimieren 406
  - reduzieren 72, 481
  - Schleifen 222
  - wann beseitigen 217
  - zufällig und erforderlich 412
  - zufällig und essenziell 225
- Durchlaufcode 407
- Durchsatz 513
- E**
- Eid
  - fehlerfreies Verhalten und Struktur 595

- Eid für Programmierer 583
  - eine Aufgabe 39, 78, 83, 187, 208, 215, 267, 269
    - Definition 268
    - nicht übertreiben 331
    - Zerlegung 669, 671, 681
  - Einfaches Design 408
  - Einfachheit 399
  - Einkapselung 197, 231
  - Einrückungen 174, 271
  - Eisenhower, General Dwight D. 598, 719
  - Eisenhowers Entscheidungsmatrix 598
  - Elemente
    - Größe im Code 78
  - else-Anweisungen
    - extrahieren 270
    - unnötig 484
  - Eltern-Kind-Beziehung 526
  - Emacs 48
  - emergentes Design
    - Regeln 330
  - Emotionen 633
  - Entitäten 565
  - Entkopplung
    - Abhängigkeiten 446
    - Code 402, 713
    - Nebenläufigkeit 506
    - Sperren und kritische Abschnitte 669
  - Entscheidungen
    - aufschieben 533, 539, 544
  - Entscheidungsproblem 572
  - Entwicklung (Softwarearchitektur) 537
    - taktisch 715, 719
  - Entwicklungstempo 595
  - Enumerationen
    - in Klassen umwandeln 89
  - Enumeratoren
    - und Instanzen 86
  - enums
    - in Interfaces umwandeln 89
  - Erzeuger-Verbraucher 513
  - Ethik als Softwareentwickler 583
  - Exceptions 212
    - besser als Fehlercodes 212
  - Extract-Methode (Refactoring) 83, 268
  - Extrahieren
    - Beispiel 220
    - Einwände 270
    - Implementierungsdetails 338
  - Extrahieren bis zum Abwinken 270, 283, 299, 336
  - extrahierte Funktionen 79, 83, 268, 285, 295, 299, 304, 471, 473, 479, 669
  - Extraktionsmethode 268
  - Extreme Programming 41, 352, 353, 401
- ## F
- Facade Pattern 414
  - Fan-in/Fan-out-Metriken 454
  - Feathers, Michael 40, 48
  - Fehler
    - gelegentlich auftretend 517
    - katastrophale Softwarefehler 581
    - Schaden für die Gesellschaft 586
    - Schädigung der Struktur 590
    - vermeiden durch formale Sprache für KI-Prompts 391, 395
    - wiederholtes Handling 481
  - Fehlercodes 210
    - Exceptions 212
  - Fehler-Handling
    - eine Aufgabe 215
  - Finite State Machine 194
  - Fit 41
  - FitNesse 133, 136, 139, 161, 168, 177, 218, 367, 378, 543, 621
  - FIT-Tool 543
  - Flag-Argumente 208
  - Flow (Geisteszustand) 634
  - Flüchtigkeit 428, 450, 460
  - Fluktuationen 450
  - Formatierung 160, 293
    - horizontal 170
    - vertikal 160
    - von Kommentaren 151
  - Formatierungsregeln 177
  - for-Schleifen 135, 705
    - extrahieren 270
  - FORTRAN 120, 182, 271, 384, 573, 606, 683
  - Fowler, Martin 72, 239, 276, 330, 331
  - Fragilität 85, 597
  - Frameworks 567
  - Freeman, Steve 563
  - fromRoman (Funktion) 61
  - Füllwörter 114, 115
  - Functional Design (Buch) 229
  - funktionale Dekomposition *siehe* funktionale Zerlegung 608

funktionale Programmiersprachen 228, 297, 327, 510  
 funktionale Programmierung 226  
 funktionale Zerlegung 186, 608, 609  
 Funktionalität  
   im Griff behalten 667  
   und Zuverlässigkeit 358  
 Funktionen 78, 299  
   Abhängigkeit 167  
   Argumente 205, 221  
   aufteilen 209, 210, 245  
   benennbar 190  
   eine Aufgabe 215  
   extrahieren 79, 83, 217, 268, 285, 295, 299, 304, 471, 473, 479, 669  
   Extraktion rückgängig machen 273  
   Funktionsbibliotheken 435  
   Geschichte der 189  
   Größe 182, 270, 471, 667, 671  
   Heuristiken 205  
   Hilfsfunktionen 472  
   homogen 197  
   in Klassen 245  
   isoliert 194  
   Kapselung 231  
   Klassen in großen 276, 293  
   Kontext 190  
   kontextbezogen 189  
   Länge 78  
   Namen 61, 78  
   Nebeneffekte 226  
   öffentlich 299, 308  
   ohne Argumente 205  
   privat 299, 308  
   Reihenfolge 61  
   rein 62, 199  
   schreiben 234  
   switch-Anweisungen 325  
   und die SOLID-Prinzipien 409  
   unrein 226  
   verheddert 187  
   zerhackt 61, 89  
 Funktionsaufrufe 168, 273, 300  
 Funktionsnamen 183  
   abstrakt 300  
   beschreibend 191  
   Länge 118  
 Funktion vs. Sauberkeit 53

**G**

Garbage Collection 227, 511  
 gemeinsam genutzte Daten 522, 526  
 Geschäftsregeln 98, 293, 428, 431, 540, 564, 565  
 Geschwätz  
   in Kommentaren 145  
 Geschwindigkeit 272  
   Build erstellen 630  
   Debugging 631  
   Nebenläufigkeit 507  
   Objektorientiertes Design 326  
   Programmierung 629  
   Tests 375, 630  
   vs. Klarheit 62  
 Gesetz von Demeter 314  
 Gewissensentscheidungen 586  
 gi (graphic interpreter) 39, 267, 272, 274  
 GitHub 54, 293, 618, 620  
 Given-When-Then-Konvention 378, 392  
 Go 117, 210, 286, 307  
   sauberer Code in 51  
 Golang 285  
 Goldstine, Herman 329  
 goto-Anweisungen 271, 607  
 GOTO-Anweisungen 231  
 Go To Statement Considered Harmful (Paper) 231, 607  
 Grenning, James 426, 547  
 Grenzen  
   zwischen UI und Anwendung 553  
 Grok 3 84, 386  
   als Assistent 73  
   Grenzen 107, 388, 395  
   in der Programmierung 658  
 Guards 250  
 Gültigkeitsbereiche 174, 206  
   Namenslänge 686  
   Nebenläufigkeit 510  
   und Länge von Funktionsnamen 118, 193  
   und Länge von Variablennamen 118

**H**

Hardware-Abstraktions-Schichten (HAL) 561  
 Hardwaregrenzen 361  
 Harvard Mark 1 189, 383  
 Hash-Maps 207, 208  
 Hauptreihe 462  
 Hauptwörter *siehe* Nomen

- Header-Dateien 215
  - Healthcare.gov 43, 582, 586, 642
  - Hello World (Beispielcode) 549
  - Heuristiken 330
    - Funktionen 205
  - hexagonale Architektur 556, 563
  - Hierarchien 174, 293
    - Klassen 134
    - Klassen und Namespaces 271
    - und Namen 193
  - High-Level-Policy
    - Stepdown-Regel 301, 304
    - Trennung von Low-Level-Details 400, 427, 431
    - und das Stable-Abstractions-Prinzip (SAP) 458
  - Hilfsfunktionen 69, 472
  - HN *siehe* ungarische Notation
  - Hollerith-Grenze 171
  - Homeoffice 638
  - Hopper, Grace 189, 383, 384, 573
  - HTML
    - in Kommentaren 151
  - Humble Object Pattern 362
  - Hunt, Andy 317
- I**
- IBM 573, 574
  - Ideen
    - gute und schlechte 66
  - IDE (Integrated Development Environment) 237, 268
    - Abhängigkeiten 94
    - Argumente 206
    - Codierungen 120
    - Kommentare 130, 688, 693
    - Namen 192
  - if-Anweisungen 90
    - Code-Änderungen 321
    - extrahieren 270
    - kleine Funktionen 667, 668
  - if/else-Verschachtelung 405, 484, 608
  - if/else-Verzweigung 320
  - Impedance Mismatch 318
  - Implementation Patterns (Buch) 35
  - Implementierungen 191
    - Module 299
  - Implementierungsdetails 335, 336, 338, 346, 348, 400, 431, 472, 473, 481, 482, 509, 561, 689
  - Induktion 604
  - Informationen
    - in Kommentaren 133
  - Initialisierung 670, 671
  - Inlining 273
  - Inputs
    - Akzeptanztests 378
  - Instabilität 454
  - instanziierten
    - abgeleitete Klassen 103
  - Instanzvariablen 166, 245, 275, 296
    - Namen 61
    - Reinheit von Funktionen 62
  - Integration
    - kontinuierlich 618
  - IntelliJ 265
  - Interfaces 216, 667
    - Abstraktionen 428
    - anwendungsspezifisch 561
    - hexagonale Architektur 557
    - Kommentare 690, 692, 696
    - Module 299
    - polymorph 427
    - Schnittstellenadapter 566
    - Strategy-Pattern 262
    - unbekannt 561
    - und das LSP 422
  - Interface-Segregation-Prinzip (ISP) 424
    - Übersicht 411
  - IoT (Internet of Things) 43
  - IoT-Software 548, 557
  - Isolation
    - IoT 555
    - UI 555
  - ItemList.java 90
  - Iterationen 678
- J**
- Jacobson, Ivar 563
  - jar (Datei) 106, 324, 439
  - Java 122, 138, 166, 190, 274, 293, 307
    - Aussagekraft 404
    - Komponenten 433
    - sauberer Code in 51
  - JavaBeans 122

- Javadocs 138, 154, 208
  - Dokumentation 138
  - redundant 141
- JavaScript 87, 106, 499
  - sauberer Code in 51
- JCommon (Bibliothek) 78
- JDepend 161
- Jennings, Jean 574
- Jiggling 521
- Jitters 450
- JUnit 161, 165, 169
- JUnit 4 137
- JVM-Startzeiten 265
  
- K**
- Kapselung 194, 275, 510
  - von Funktionen 283
- Kay, Alan 189, 307, 308, 317
- Kerievsky, Josh 128
- Kernighan, Brian W. 129
- KI (künstliche Intelligenz) 34, 385
  - als Assistent 73, 265, 349
  - Bedarf an Programmierern 577
  - ersetzt Programmierer 385
  - Fehler 388
  - Grenzen 107, 349, 395
  - in der Programmierung 658
  - trainieren 349
- KI-Prompts
  - debuggen 395
- Klarheit 123
  - die vier K des Designs 469
  - durch Kohäsion 500
- Klassen 271
  - abgeleitet 102, 192
  - aus Enumeratoren 89
  - Beziehungen 258
  - Design 328, 329
  - extrahieren 293
  - Größe 331, 347
  - große Funktionen 276, 293
  - Namen 121
  - Nebenläufigkeit 512
  - neu erstellen 334
  - öffentlich 299
  - privat 299
  - SOLID-Designprinzipien 330
  - und Dateien 327
  - und die SOLID-Prinzipien 409
  - zu viele 261
- Klassenhierarchie 134
- Klassennamen
  - Länge 119
- Knight Capital 43, 582, 587, 590
- Knuth, Don 674, 693, 699
- kognitive Belastungen 667, 680
- Kohäsion 412, 445, 500
  - die vier K des Designs 497
- kollaborative Programmierung 637
- Kommentare
  - Ablenkung 130
  - aktuell halten 131
  - als Tagebuch 144
  - als Warnung 137
  - ausgeblendet oder verdeckt 130, 688, 693
  - bearbeiten 154
  - Code-Zweck erklären 132
  - durch Funktionen oder Variablen
    - ersetzen 149
  - erklärend 156
  - Erklärung der Absicht 134
  - Farbe 130
  - fehlerhaft 131, 134, 140, 684
  - Formatierung 151
  - Funktionsbezeichner 149
  - Funktionskopf 153
  - Geschwätz 145
  - Grenzen 129, 336
  - gut 133
  - gute und schlechte 131
  - HTML 151
  - in den Code verschieben 686
  - informierend 133
  - Insider 132
  - Interfaces 690, 692, 696
  - irreführend 140
  - juristisch 133
  - klärend 136
  - lange Namen 686
  - Lügen 131
  - minimieren 132
  - nicht-lokal 152
  - notwendiges Übel 129
  - Nutzen 129, 158, 682, 703, 723
  - prüfen 692
  - Redundanz 140
  - schlecht 138
  - schlechter Code 132
  - Selbstgespräche 138
  - TODO 148

- und Abstraktionen 688, 693, 696
    - unübersichtlich 144
    - vorgeschrieben 144
    - zur Verstärkung 138
    - zu viele Informationen 152
  - Kommunikation
    - durch Formatierung 160
  - Komplexität 194
    - durch Struktur 107
    - erklärende Kommentare 697, 699, 703, 709
    - Methoden aufteilen 677
    - Software-design 666
  - Komponenten 299
    - abstrakt 457
    - Architekturgrenzen 540, 545
    - Common-Closure-Prinzip (CCP) 441
    - Definition 433
    - Geschichte 434
    - Hauptreihe 459
    - Kohäsion 439
    - Kopplung 443
    - Stabilitätsmetriken 454
  - Komponenten-Cache
    - Browser 107
  - Komponentenkohäsion
    - Spannungsdiagramm 444
  - Komponentenkopplung 446
  - Kontrollstrukturen
    - Verzweigungen 232
  - Konfirmierung
    - die vier K des Designs 486
  - konkrete Komponenten 428
  - Konstanten 252, 288
  - Kontext 124, 190
    - als Grenze 190
    - überflüssig 127
  - kontinuierliche Integration 618, 620
  - kontinuierlicher Build 380, 623
  - kontinuierliches Deployment 622
  - kontinuierliches Design 338, 380, 465
  - Kontrollfluss 233, 338, 339, 567
  - Kontrollstrukturen 232
    - Iterationen 232
    - Schleifen 232
    - Selektionen 232
    - Sequenzen 232
  - Kontrollvariablen 165
  - Konzepte
    - in Konstrukten gruppieren 327
  - Kopplungen 229, 288, 539
    - Argumente 206
    - Komponenten 443
    - verfrühte Entscheidungen 539
    - zeitlich 226
  - Kosten 45
    - ehrlische und faire Schätzungen 641
  - Kostenschätzung 502
  - kritische Abschnitte 510, 515, 669
  - Kuddelmuddel 187, 273, 496, 556, 667, 676, 681
  - Kürze
    - die vier K des Designs 476
  - Kurzweil, Ray 349
- ## L
- Lambdas 222, 475
  - Langr, Jeff 33, 129, 327, 465
  - Laszczak, Robert 293
  - Law of Demeter *siehe* Gesetz von Demeter
  - Law of Least Astonishment 209
  - Layout
    - visuell 162
  - Lea, Doug 511
  - lebenslanges Lernen 653
  - Lebenszyklus von Softwaresystemen 530
  - Leerzeilen
    - im Code 162
  - Lernen
    - lebenslang 653
  - Lern-Tests 558
  - Lesbarkeit 40, 78, 162, 175, 294, 367, 372
  - Leser-Schreiber 513
  - Linker 438
  - Liskov, Barbara 411, 420
  - Liskov'sches Substitutionsprinzip (LSP) 420
    - Übersicht 411
  - LISP 573
  - Literate Programming (Paper) 674
  - Livelock 513
  - LLM 34
  - LLMs (Large Language Models) 389, 391, 395
  - Lochkarten 613
  - Lochstreifen 189, 383
  - Locking
    - clientbasiert, serverbasiert oder adapted Server 515
  - Locks 512

- lokale Variablen 274
- Lösungsdomäne
  - Namen 124
- Low-Level-Details
  - Trennung von High-Level-Policys 400, 427, 431
- Luftfahrtkatastrophen 581
- M**
- makeStatement (Funktion) 285, 300, 302
- Management
  - Konfrontation 600
- manuelles Coding 519
- Marick, Brian 592
- Martin, Justin Michael 655
- Martin, Robert C. (Uncle Bob) 109, 159, 665
- max (Funktion) 251
- McCarthy, John 573
- Meetings 632
- Member-Präfixe 120
- Merges 619
  - automatisch 617
- Methoden
  - aufteilen 699
  - extrahieren 336
  - flach 667, 671, 687
  - Gesetz von Demeter 314
  - Kommunikation 245
  - Länge 667
  - Namen 122
  - tief 667, 671
  - verstrickt 667
- Meyer, Bertrand 217, 415
- Minecraft 439, 579
- Minesweeper 112
- Miniaturisierung 575
- mittlere Schicht 410
- Mocken 715
- moderne Umgebungen
  - Nebenläufigkeit 522
- Modul 412
- modulare Programmierung 189, 271
- Modularität 189
- Module 271, 299
  - abstrakte 458
  - eine Aufgabe 411
  - Gesetz von Demeter 314
  - Implementierungen 299
  - Interfaces 299
  - neu kompilieren 87
  - SOLID-Prinzipien 410
  - und Dateien 327
- Mojang 579
- Monolithen 329
- Monotone 618
- Multithread-Umgebung 296
- Multithread-Umgebungen 231
- Musik
  - beim Programmieren 633
- Mutationstests 626
- Mutatoren 122
- mutual Exclusions 512
- MySQL 544
- N**
- Namen 95
  - Ähnlichkeit vermeiden 113
  - Algorithmen und Verständlichkeit 676
  - ändern 128, 191, 294
  - bedeutungsvoll 125
  - Buchstaben und Ziffern in 114
  - Einheitlichkeit 113
  - Funktionen 61
  - für Elemente im Code 78
  - humorvoll 123
  - Kommentare oder lange Namen 691
  - Konsistenz 123
  - Länge 686
  - praktisch 192
  - Redundanz 115
  - sinnvolle Länge 118
  - Subjektivität 110
  - Unterscheidung 113
  - Verständlichkeit 686
  - Wichtigkeit 95
  - zweckbeschreibend 110
- Namensschema 170
- Namespace-Namen
  - Länge 119
- Namespaces 231, 250, 271, 327
- Nassi-Shneiderman-Diagramm 233
- Nebenbemerkungen 150
- Nebeneffekte 226, 229
  - Definition 226
  - vermeiden 228
- Nebenläufigkeit 212, 298, 506
  - Gründe 506
  - Herausforderungen 506

- IoT-Systeme 555
  - Konflikte 509
  - Mythen 507
  - Prinzipien für Effizienz 509
  - Terminologie 512
  - Nervfaktor 61
  - Neuauslieferung 87
  - Neukompilieren
    - Aufwand 89
  - Newkirk, Jim 558
  - Nicht lokale Kommentare 152
  - Nomen 121, 303
  - Normalformen 217
  - null (Subskript) 112
  - Nutzlosigkeitszone 461
  - Nygaard, Kristen 189, 307, 384, 575
- O**
- Obamacare 586
  - Objective-C 307
  - Object Mentor (Firma) 109
  - Objekte
    - Definition 308
    - erstellen und binden 555
    - erzeugen 62
    - Gesetz von Demeter 314
    - im Gegensatz zu Datenstrukturen 311, 316
    - und Abstract Factorys 430, 431
    - und Datenstrukturen 315
  - objektorientierte Programmierung 189, 229, 307, 325, 327
    - DTO 323
    - Geschwindigkeit 326
    - und das LSP 422
  - objektrelationale Abbildungen 318
  - Offenheit 162
  - öffentliche Elemente 299
  - Once and only once *siehe* Don't Repeat Yourself (DRY)
  - Open-Closed-Prinzip (OCP) 188, 293, 321, 324, 415
    - Änderungen in alten Modulen 86
    - Klassendesign 333, 340, 346, 347
    - switch-Anweisungen 85, 326
    - Systemdesign 350, 408
    - Übersicht 410
  - optimistisches Locking 617, 619
  - Optionen
    - offenhalten 531, 536
  - optische Illusionen 136
  - ORM *siehe* objektrelationale Abbildungen
  - Ottinger, Tim 109
  - Ousterhout, Dr. John Kenneth 61, 187, 273, 300, 331, 355, 665
  - Output-Argumente 209
  - Outputs
    - Akzeptanztests 378
- P**
- Parnas, David 189
  - Pascal (Programmiersprache) 231
  - Pauling, Linus 66
  - PDP11s 616
  - PDP-Systeme 575
  - Peer-to-Peer-Systeme 618
  - PEP 8 (Standard) 242
  - Per Anhalter durch die Galaxis (Buch) 655
  - Perspektivwechsel 58
  - PERT 648
  - pessimistisches Locking 617
  - Pfadfinder-Regel 50, 625
  - Pflichtenheft 63
  - Philosophenproblem 514
  - Pipelines, funktional 474, 476
  - PL/1 182
  - Plaugher, P. J. 129
  - Plug-ins 95
  - Plugins 542
  - Policies
    - Software 532
  - polymorphe Methoden 90, 262, 427
    - Objektorientierung vs. Datenstrukturen 312
  - Polymorphismus 187, 216, 289, 326, 400, 568
  - Pomodoro-Technik 635
  - Ports and Adapter 556, 563
  - Positionsbezeichner
    - in Kommentaren 149
  - Powell, Robert Baden 625
  - Prädikate 122, 337
  - Präfixe 120
  - Prefactoring 340
  - PrimeGenerator (Funktion) 672, 676, 677, 680, 681, 685, 690, 693, 697, 706, 723
  - Primitive Obsession
    - Code Smell 125
  - Principle of Least Surprise 209

- Prinzip der geringsten Überraschung 209
- Prinzipien
- Abhängigkeitsregel 262, 432, 546, 564
  - architektonische Trennung von Zuständigkeiten 563
  - eine Aufgabe 39, 78, 267, 331, 669
  - Extrahieren bis zum Abwinken 270, 283, 336
  - Ideen überprüfen 666
  - Komponentenkohäsion 439
  - Komponentenkopplung 443
  - Nebenläufigkeit 509
  - Respekt 286, 299, 301
  - SOLID 330, 331, 409
  - Stepdown-Regel 301, 304
  - YAGNI 401
- private Elemente 299
- Problemdomäne 124
- Produktionscode
- generisch 250
  - nach den Tests schreiben 353
  - saubere Tests 364
  - tabellengesteuert 257
  - von Tests entkoppelt 106, 244, 257
- Produktivität 44, 629, 637
- durch sauberen Code 45
  - Homeoffice 638
  - Prinzipien 658
- Programmablauf *siehe* Kontrollfluss
- Programmieren
- mit KI und Prompts 385
- Programmierer 33, 76
- Abhängigkeit von 43, 580
  - Ablenkungen ausmerzen 632
  - Berühmtheiten der Popkultur 577
  - Demografie 575
  - Ehre 580
  - Eid 583
  - Fehler 43
  - Gehalt 576
  - Geschichte 573
  - katastrophale Fehler 581
  - Respekt untereinander 651
  - Unerfahrenheit 576
  - unethisch 579, 585
  - Verantwortung 43, 76
  - Verdrängung durch KI 577
- Programmiersprachen 51, 271
- Ausdruckssprache 684
  - Aussagekraft 404
  - Dokumentationssystem 138
  - formale Sprache für KI-Prompts 391, 395
  - Geschichte 307, 384, 573
  - Klassen, Module und Dateien 327
  - lernen 653
  - statische und dynamische Typisierung 425
  - und das ISP 425
- Programmierung
- Akzeptanztests 377
  - als Erzählkunst 234
  - Definition 34
  - Design 501
  - Geschichte 238, 383, 572, 613
  - Katastrophen 581
  - kollaborativ 637
  - Nebenläufigkeit 507
  - Objektorientierung vs. Datenstrukturen 311
- Projektentwicklung
- Design 501
- Prompts
- als Basis für die Programmierung 34
  - formale Sprache für KI-Prompts 391, 395
  - Programmieren per KI 385
- Prosa 40, 183
- prozeduraler Code 325
- Pryce, Nat 563
- Python 117, 123, 216, 240, 262, 307
- abstrakte Komponenten 458
  - IoT 549
  - nebenläufige Programmierung 558
  - Plug-in 265
  - sauberer Code in 51
- ## Q
- Quellcode
- Abhängigkeiten 417, 418, 424, 427, 431
  - Editor 406
  - kontinuierlicher Build 380
  - Kontrollsysteme 145, 150
  - Versionsverwaltung 613
- Quelldateien 299, 328
- Queue (Funktion) 558
- ## R
- Race-Bedingungen 582

- Race Condition 135
  - RCS (Revision Control System) 617, 619
  - React Native 522, 525
  - Rechenschaftspflicht 585
  - Redeployment *siehe* Neuauslieferung
  - Redundanz 141
    - in Kommentaren 140
    - in Namen 115
  - Reenskaug, Trygve 563
  - ReentrantLock 512
  - Reeves, Jack W. 465
  - Refactoring 187, 238, 402
    - Ablauf 239
    - aufschieben 716
    - Beispiel 154
    - Definition 239
    - Extract-Methode 83, 268
    - und Tests 106
    - Unit-Tests 716
  - Refactoring (Buch) 72, 239, 276, 315, 330
  - Referenzzählung 227
  - Regeln
    - im Team 160
    - vereinbaren 177
  - reguläre Ausdrücke 134
  - Reinheit 202
    - teilweise 202
  - relationale Datenbanken 308, 318, 524
  - Remote Procedure Call (RPC) 549
  - Rendering
    - parallel 525
  - Respekt 299
    - durch Struktur 301
  - Respekt unter Programmierern 651
  - REST-Schnittstelle 422, 532
  - Reuse/Release-Equivalence-Prinzip (REP) 440, 444
  - Revert
    - und Commit 354
  - Richtungssteuerung 419
  - robuster Code 85
  - Rochkind, Marc 617
  - Römische Zahlen
    - umwandeln 54
  - Rot-Grün-Refaktor-Zyklus 599, 609, 657, 711, 716
  - Routinen 181
  - Ruby 123, 208, 307
    - abstrakte Komponenten 458
    - Komponenten 433
    - sauberer Code in 51
- ## S
- saubere Funktionen 181, 233
    - Eigenschaften 189
  - sauberer Code 267, 299
    - Definition 33, 39, 329
    - gute Gründe 44
    - ist kein perfekter Code 48
    - kontinuierliche Verbesserungen 627
    - Lesbarkeit 676
    - plattformunabhängig 51
    - Produktivitätsturbo 45
    - saubere Grenzen 561
    - saubere Klassen 330
    - sauberer Architektur 563
    - saubere Tests 363, 367
    - sprachunabhängig 51
    - Standard 62
    - Techniken 240
    - unerbittliche Verbesserung 625
    - Voraussetzungen 239
    - Zielgruppe 110
  - Sauberkeit
    - Standard 62
  - Sauberkeit vs. Funktion 53
  - SCCS 617, 619
  - schädlicher Code 585
  - Schätzung
    - Ehrlichkeit 648
  - Schätzungen
    - Projektplan 502
  - Scheren-Regel 166
  - schlechte Kommentare
    - Beispiel 154
  - schlechter Code 36, 716
    - Beispiel 154
    - Kommentare 132
    - transformieren 38
  - Schleifen
    - Duplizierungen 222
    - Kommentare zu Schleifengrenzen 158
    - verschachtelt 223
    - Zeitungslesen 301
  - Schlüsselwort-Argumente 207
  - Schlüsselwörter 183
  - Schlüsselwort/Wert-Paare 208
  - Schmerzzone 460
  - Schuchert, Brett L. 505

- schwache Kopplung 194
- Scissors-Regel 166
- Seeman, Mark 42
- Selbstgespräche 138
- selbstvalidierende Tests 375
- semantische Stabilität 627
- Semaphore 135, 512
- Semmelweis, Ignaz 37
- Servlet-Modell 506
- Shutdown-Code 515
- Sieb des Eratosthenes 701, 702
- Sigmas 646, 648
- Simonyi, Charles 120
- SimpleDateFormat 137
- Simula (Programmiersprache) 307, 384, 575
- Single-Act-Regel 374
- Single-Assert-Regel 374
- Single-Choice-Prinzip 217
- Single Responsibility Principle (SRP) 321, 333, 411
- Single-Responsibility-Prinzip 83, 84, 87, 407, 500
- Single-Responsibility-Prinzip (SRP) 188, 226, 261, 302, 676
  - Klassendesign 331, 333, 344, 347
  - Nebenläufigkeit 509, 527
  - Softwarearchitektur 537
  - Übersicht 410
  - und das Common-Closure-Prinzip (CCP) 441
- Single-Thread-Anwendungen 231
- Single-Thread-Code 506
- Singleton-Pattern 134, 343
- Singularität 349
- Skia 525
- Smalltalk (Programmiersprache) 41, 307, 328, 347
- SNOBOL 617
- Software
  - Akzeptanztests 377
  - allgegenwärtig 43
  - Architektur 530
  - Auswirkungen fehlerhafter 43
  - Bausteine 468
  - Flexibilität 531, 591, 627
  - Katastrophen 581
  - mathematische Grundlagen 573, 603
  - Policies und Details 532
  - Veränderbarkeit 591, 627
  - Verhalten und Struktur 531, 591, 596
  - von Drittanbietern 548
- Software Development (Zeitschrift) 154
- Softwareentwicklung 501
- SOLID-Designprinzipien 330, 407, 598
  - Dependency-Inversion-Prinzip (DIP) 411, 426
  - hexagonale Architektur 556
  - Interface-Segregation-Prinzip (ISP) 411, 424
  - Liskov'sches Substitutionsprinzip (LSP) 411, 420
  - Open-Closed-Prinzip (OCP) 410, 415
  - Single-Responsibility-Prinzip (SRP) 410
  - Zweck 409
- Sorgfalt 41
- Source Forge 617, 621
- Sparkle 182
- Speicher
  - Geschwindigkeit vs. Klarheit 62
- Speicherbereinigung 227
- Sperren 524, 527
  - in SCCS 617
- sprechende Namen 78
- Sprunganweisungen 232
- SQL 543, 564
- Stabilität 452
- Stable-Abstractions-Prinzip (SAP) 458, 546
- Stable-Dependencies-Prinzip (SDP) 452
- Stack 212
- Stakeholder
  - Programmierer 599
- Standard 63
- Standardabweichungen 646, 648
- Starrheit 597
- Startup-Code 515
- Startups 592
- Starvation 512
- State Machine 66
- Statement.java 90
- Statement (Klasse) 82
- Statement (Modul) 86
- State Pattern 347
- statische Datenbanken 319
- statische Initialisierer 137
- statische Variablen 296
- statistische Analyse 159
- Stepdown-Regel 185, 197, 269, 301, 304
- Stimmung und Stress 633
- Strategy-Pattern 262, 340, 343, 407

- Stress 633
  - Stroustrup, Bjarne 39, 267, 307
  - Struktur 187
    - durch Austausch von Botschaften 308
    - Kuddelmuddel 676, 679
    - Nebenläufigkeit 506
    - Software 531
    - zusätzliche Komplexität 107
  - Strukturen 176, 272, 301, 327
    - Elemente guter 597
    - sauber und offen für Änderungen 596
  - strukturierte Programmierung 186, 231, 575, 606
  - Stubs 544
  - Subroutine 406
  - Subroutinen 181, 189, 329, 644
    - aufrufen 181
  - Subtypen 420
  - Subversion 617, 618, 620
  - Swing 182, 362
  - Swing-Programme 668
  - switch-Anweisungen 90, 187, 320
    - OCP-Verstoß 85
  - synchronisierte Abschnitte 515
  - synchronized (Schlüsselwort) 510, 514
  - Syntaxdarstellung 114
  - Systemanforderungen 377
  - Systemaspekte 723
- T**
- Tagebuch-Kommentare 144
  - taktische Programmierung 715, 719, 720
  - TCR 351, 354, 592
  - TDD 182, 241, 351, 592
    - Alternativen 712
    - Debugging 251
    - Design 715, 722
    - drei Gesetze 353, 711
    - kleine, verifizierte Änderungen 549
    - reproduzierbarer Beweis 609
    - Vorteile 713
    - Zyklus 354
  - Team
    - Regeln 177
  - Teamarbeit 637
  - Teams
    - Standard 63
  - Techniken
    - für sauberen Code 240
  - Template-Method-Pattern 224, 407
  - Teradyne 643
  - Termine
    - einhalten 38, 45
  - Testabdeckung 625
  - Testdisziplinen 351, 354, 592, 657
    - Small Bundles 355, 712, 713, 715, 719, 720
    - und Design 355, 359
    - Vorteile 356, 359, 592
  - Test-Driven Development (TDD) 710
  - Testen
    - KI-Prompts 395
    - SRP-Klassen 333
    - Threaded-Code 516
  - Testfälle 250
    - abschalten 137
  - Test-First-Programmierung *siehe* TDD
  - TestNG 161, 165
  - Tests 35, 72, 624
    - Abdeckung 401, 486, 495, 622
    - Akzeptanztests 377, 543
    - als Dokumentation 358, 473, 713, 714
    - Aufgabe beim Design 503
    - Aussagekraft von Code 406
    - Bestehen oder Scheitern 375
    - bündeln 355, 712, 713, 715, 719
    - domänenspezifische Sprache 371
    - Fehler im Design 243
    - F.I.R.S.T. 375
    - Fragilität 280
    - für Drittanbieter-Code 557
    - Geschwindigkeit 630
    - Jiggler 520
    - kleine Unit-Tests 348, 363, 486, 710, 713
    - Lern-Tests 558
    - Lesbarkeit 367, 372
    - Mutationstests 626
    - offenlegen der Absicht 406
    - parametrisch 257
    - Robustheit 287
    - saubere Tests 363
    - schnell, sicher und reproduzierbar 610
    - semantische Stabilität 627
    - Single-Act-Regel 374
    - spezifisch 250
    - Störfaktoren 254
    - Testdesign 376
    - Testen und nochmals Testen 63
    - Testsuite 361, 363

- und Bereinigung 244
- und Refactoring 106, 239
- unpraktisch 361
- vom Produktionscode entkoppelt 106
- vor dem Produktionscode schreiben 353
- zur Konfirmierung 486
- Therac-25-Strahlentherapiegerät 582
- Thomas, Dave 317
- Threads
  - Sperren 669
  - testen 516
  - unabhängig 511
  - und Race Condition 135
- Tichy, Walter 617
- Time and Money 161
- TODO-Kommentare 148
- Toggles und Branches 620
- Tomcat 141, 161
- Top-down-Design 451
- Toyota
  - Softwarekatastrophe 582, 588, 590
- Train Wrecks 314
- Transaktionen
  - isoliert 523, 524
- transitive Abhängigkeiten 420
- Treiber 567
- trim-Funktion 138
- try/catch-Blöcke 146, 214
  - Fehlernamen 117
- Tunnel (Geisteszustand) 634
- Turing, Alan 572
- Tuwörter *siehe* Verben
- Typen
  - Designänderungen 325
  - mehrere Typen in einer Datei 328
  - und Subtypen 420
- Typisierung 425

**U**

- überlebende Mutationen 626
- unabhängige Auslieferung 106, 644
- Unbeweglichkeit 597
- ungarische Notation 120
- Ungenauigkeit 141
- Unified Modeling Language (UML) 332, 334, 718
- Units 466

- Unit-Tests 348, 363, 486, 619, 710, 713, 715, 722
- UNIX 617
- unsauberer Code 38
  - Definition 329
  - führt zu langsamer Arbeit 38
  - Herausforderungen 33, 468
- Unterklassen 420
- Urheberhinweis 133
- Use Cases *siehe* Anwendungsfälle

**V**

- Variablen 299
  - in funktionalen Programmiersprachen 228
  - statisch 296
- Variablendeklarationen 165
- Variablennamen
  - Länge 118
- variadische Argumente 207
- VAXes 616
- Veränderungen 319
  - Achsen 261
  - Fragilität und Starrheit 322, 597
  - Kompromiss zwischen objektorientierten und prozeduralem Konzept 325
  - Kompromiss zwischen OO- und prozeduralem Konzept 319
  - und Klassendesign 329, 346
- Verantwortung
  - Programmierer 43, 76, 581, 585, 588
- Verben 121, 191, 211
- Verbesserungen
  - kontinuierlich 625, 653
- Verhalten
  - Software 531, 591, 595, 596
- Verschachtelung
  - ausbrechen 709
- Verschachtelungen 183, 223
- Versionsnummern 440
- Versionsverwaltung 613
- Verständlichkeit 61
  - Algorithmen 676, 703, 704
- Vertikale Formatierung 160
- Verwalten
  - Tests 363
- Verzeichnisstrukturen 89, 96
  - Abhängigkeiten 96

Verzögerungen  
 zufällig 135  
 virtuelle Büros 638  
 Viskosität 629  
 Volkswagen (Abgasskandal) 579  
 von Neumann, John 329, 574  
 vorangestellte Kommentare  
 Interface-Dokumentation 690

**W**

Wachstum 319  
 Nebenläufigkeit 507  
 Wading 36  
 Wahrscheinlichkeitsverteilungen 646, 647  
 Wartbarkeit  
 von Code 160  
 Wartung  
 Abhängigkeitsstrukturen 451  
 Architektur 530  
 von Code 248  
 Waten *siehe* Wading  
 Webserver 532  
 Weißraum 162, 163, 171  
 We, Programmers (Buch) 307, 383  
 Wheeler, David 329  
 while-Anweisungen 668  
 Wichtigkeit und Dringlichkeit 598  
 wiederholbare Tests 375  
 Wiederholungen  
 von Codezeilen 217  
 Wiki  
 Erfinder 41, 656, 657  
 Wilkes, Maurice 329  
 Wirth, Niklaus 231, 307, 607  
 wohlgeschriebene Prosa 78  
 Wortarten 121

**X**

XP Immersion 154

**Y**

YAGNI (You Aren't Going to Need It) 94, 401

**Z**

Zeilenlänge 170  
 Zeilenstruktur 568  
 Zeitdruck 38, 45, 595, 649  
 zeitliche Kopplungen 226, 227, 229  
 Zeitmanagement 635  
 Zeitschätzung 502  
 Zeitstempel 134  
 Zeitungsmetapher 301  
 Zenons Paradoxon 46  
 zerhackte Funktionen 61, 89  
 Zerlegung 187, 667, 671, 680, 681  
 funktional 186  
 Zielgruppe 128  
 Erwartungen 110  
 sauberer Code 110  
 Zone of Pain 460  
 Zone of Uselessness 461  
 zufällige Verzögerungen 135  
 Zugriffsmethoden 122  
 Zugunglück 314  
 Zunächst bring es zum Laufen. Dann mach  
 es richtig. 54, 187, 234, 238, 305, 595, 599  
 zusammengesetzte Testergebnisse 371  
 Zuschreibungen 150  
 Zusicherungen *siehe* Assertions 255  
 Zuständigkeiten  
 trennen 85  
 Zustandsänderungen 226, 229  
 in funktionalen Programmiersprachen  
 228  
 Zuweisungsanweisungen  
 in funktionalen Programmiersprachen  
 228  
 Zweck 134  
 deklarieren 473  
 im Code ausdrücken 133  
 Namen, die den Zweck verraten 110  
 Tests 280  
 verschleiern 272  
 Zyklen 619