

Robert C. Martin

Clean Code

Refactoring, Patterns, Testen
und Techniken für sauberen Code

Robert C. Martin Series

2. Auflage

Deutsche
Ausgabe



Inhaltsverzeichnis

Vorwort	19
Einleitung	25
Ein Hinweis zu älteren Kapiteln	27
Wie dieses Buch aufgebaut ist	27
Einleitung (aus längst vergangenen Zeiten)	29
Über den Autor	31
1 Sauberer Code	33
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
Teil I Code	51
2 Halten Sie Ihren Code sauber!	53
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
3 Grundlegende Prinzipien	77
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
4	Aussagekräftige Namen	109
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
5	Kommentare	129
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
6	Formatierung	159
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
7	Saubere Funktionen	181
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
8	Heuristiken für Funktionen	205
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

9	Die saubere Methode	237
9.1	Mach es richtig	238
9.2	Beispiel	240
9.2.1	Design und Architektur berücksichtigen	257
9.3	Zusammenfassung	265
10	Eine Aufgabe	267
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
11	Eine Frage des Respekts	299
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
12	Objekte und Datenstrukturen	307
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

13	Saubere Klassen	327
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
14	Testdisziplinen	351
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

15	Saubere Tests	367
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
16	Akzeptanztests	377
16.1	Die Akzeptanztest-Disziplin	378
16.1.1	Die Disziplin	379
16.1.2	Kontinuierlicher Build	380
16.2	Zusammenfassung	381
17	KI, LLMs und wie sie alle heißen	383
17.1	Programmieren per Prompt	385
17.1.1	In den Kinderschuhen	391
17.1.2	Wilde wissenschaftliche Vermutung	391
17.2	Zusammenfassung	396
Teil II	Design	397
18	Einfaches Design	399
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

19	Die SOLID-Prinzipien	409
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
20	Komponentenprinzipien	433
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
21	Kontinuierliches Design	465
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
22	Nebenläufigkeit	505
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
Teil III Architektur		529
23	Die Geschichte zweier Werte	531
23.1	Optionen offenhalten	531
24	Unabhängigkeit	535
24.1	Use Cases	535
24.2	Betrieb	536
24.3	Entwicklung	537
24.4	Deployment	537
24.5	Optionen offenhalten	537
25	Architekturgrenzen	539
25.1	Welche Grenzen sollten Sie ziehen – und wann?	540
25.2	Plugin-Architektur	542
25.3	Fallstudie: FitNesse	543
25.4	Zusammenfassung	545
26	Saubere Grenzen	547
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
27	Saubere Architektur	563
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

Teil IV Craftsmanship	571
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
28 Schaden	585
28.1 Gesellschaftlicher Schaden	586
28.2 Funktionsbeeinträchtigungen	587
28.3 Schädigung der Struktur	590
28.4 Soft	591
28.5 Tests	592
29 Weder im Verhalten noch in der Struktur fehlerhaft	595
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
30 Reproduzierbarer Beweis	603
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
31 Kurze Zyklen	613
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