

Inhaltsverzeichnis

	Vorwort	15
	Einführung	21
I	Sauberer Code	25
I.1	Code, Code und nochmals Code	26
I.2	Schlechter Code	27
I.3	Die Lebenszykluskosten eines Chaos	28
	Das große Redesign in den Wolken	29
	Einstellung	30
	Das grundlegende Problem	31
	Sauberen Code schreiben – eine Kunst?	31
	Was ist sauberer Code?	32
I.4	Denkschulen	40
I.5	Wir sind Autoren	41
I.6	Die Pfadfinder-Regel	43
I.7	Vorläufer und Prinzipien	43
I.8	Zusammenfassung	43
2	Aussagekräftige Namen	45
2.1	Einführung	45
2.2	Zweckbeschreibende Namen wählen	45
2.3	Fehlinformationen vermeiden	47
2.4	Unterschiede deutlich machen	49
2.5	Aussprechbare Namen verwenden	50
2.6	Suchbare Namen verwenden	51
2.7	Codierungen vermeiden	52
	Ungarische Notation	52
	Member-Präfixe	53
	Interfaces und Implementierungen	54
2.8	Mentale Mappings vermeiden	54
2.9	Klassennamen	55
2.10	Methodennamen	55

2.11	Vermeiden Sie humorige Namen.	55
2.12	Wählen Sie ein Wort pro Konzept	56
2.13	Keine Wortspiele	56
2.14	Namen der Lösungsdomäne verwenden	57
2.15	Namen der Problemdomäne verwenden	57
2.16	Bedeutungsvollen Kontext hinzufügen	57
2.17	Keinen überflüssigen Kontext hinzufügen	60
2.18	Abschließende Worte	60
3	Funktionen	61
3.1	Klein!	64
	Blöcke und Einrückungen.	65
3.2	Eine Aufgabe erfüllen	65
	Abschnitte innerhalb von Funktionen	66
3.3	Eine Abstraktionsebene pro Funktion	67
	Code Top-down lesen: die Stepdown-Regel	67
3.4	Switch-Anweisungen.	68
3.5	Beschreibende Namen verwenden.	70
3.6	Funktionsargumente.	71
	Gebräuchliche monadische Formen	72
	Flag-Argumente	72
	Dyadische Funktionen.	73
	Triaden	73
	Argument-Objekte.	74
	Argument-Listen	74
	Verben und Schlüsselwörter.	75
3.7	Nebeneffekte vermeiden	75
	Output-Argumente	76
3.8	Anweisung und Abfrage trennen	77
3.9	Ausnahmen sind besser als Fehler-Codes	77
	Try/Catch-Blöcke extrahieren	78
	Fehler-Verarbeitung ist eine Aufgabe	79
	Der Abhängigkeitsmagnet Error.java	79
3.10	Don't Repeat Yourself	80
3.11	Strukturierte Programmierung	80
3.12	Wie schreibt man solche Funktionen?	81
3.13	Zusammenfassung	81
3.14	SetupTeardownIncluder	82

4	Kommentare	85
4.1	Kommentare sind kein Ersatz für schlechten Code	86
4.2	Erklären Sie im und durch den Code	87
4.3	Gute Kommentare	87
	Juristische Kommentare	87
	Informierende Kommentare	88
	Erklärung der Absicht	88
	Klarstellungen	89
	Warnungen vor Konsequenzen	90
	TODO-Kommentare	91
	Verstärkung	91
	Javadocs in öffentlichen APIs	92
4.4	Schlechte Kommentare	92
	Geraune	92
	Redundante Kommentare	93
	Irreführende Kommentare	95
	Vorgeschriebene Kommentare	96
	Tagebuch-Kommentare	96
	Geschwätz	97
	Beängstigendes Geschwätz	99
	Verwenden Sie keinen Kommentar, wenn Sie eine Funktion oder eine Variable verwenden können	100
	Positionsbezeichner	100
	Kommentare hinter schließenden Klammern	101
	Zuschreibungen und Nebenbemerkungen	101
	Auskommentierter Code	102
	HTML-Kommentare	102
	Nicht-lokale Informationen	103
	Zu viele Informationen	104
	Unklarer Zusammenhang	104
	Funktions-Header	104
	Javadocs in nicht-öffentlichem Code	105
	Beispiel	105
5	Formatierung	109
5.1	Der Zweck der Formatierung	109
5.2	Vertikale Formatierung	110
	Die Zeitungs-Metapher	111
	Vertikale Offenheit zwischen Konzepten	112

	Vertikale Dichte	113
	Vertikaler Abstand	114
	Vertikale Anordnung	119
5.3	Horizontale Formatierung	119
	Horizontale Offenheit und Dichte	120
	Horizontale Ausrichtung	121
	Einrückung	123
	Dummy-Bereiche	124
5.4	Team-Regeln	125
5.5	Uncle Bobs Formatierungsregeln	125
6	Objekte und Datenstrukturen	129
6.1	Datenabstraktion	129
6.2	Daten/Objekt-Anti-Symmetrie	131
6.3	Das Law of Demeter	133
	Zugkatastrophe	134
	Hybride	135
	Struktur verbergen	135
6.4	Datentransfer-Objekte	136
	Active Record	137
6.5	Zusammenfassung	138
7	Fehler-Handling	139
7.1	Ausnahmen statt Rückgabe-Codes	139
7.2	Try-Catch-Finally-Anweisungen zuerst schreiben	141
7.3	Unchecked Exceptions	143
7.4	Ausnahmen mit Kontext auslösen	144
7.5	Definieren Sie Exception-Klassen mit Blick auf die Anforderungen des Aufrufers	144
7.6	Den normalen Ablauf definieren	146
7.7	Keine Null zurückgeben	147
7.8	Keine Null übergeben	148
7.9	Zusammenfassung	150
8	Grenzen	151
8.1	Mit Drittanbieter-Code arbeiten	151
8.2	Grenzen erforschen und kennen lernen	154
8.3	log4j kennen lernen	154
8.4	Lern-Tests sind besser als kostenlos	156

8.5	Code verwenden, der noch nicht existiert	157
8.6	Saubere Grenzen	158
9	Unit-Tests	159
9.1	Die drei Gesetze der TDD	160
9.2	Tests sauber halten	161
	Tests ermöglichen die -heiten und -keiten	162
9.3	Saubere Tests	163
	Domänenspezifische Testsprache	166
	Ein Doppelstandard	166
9.4	Ein assert pro Test	168
	Ein Konzept pro Test	170
9.5	F.I.R.S.T.	171
9.6	Zusammenfassung	172
10	Klassen	173
10.1	Klassenaufbau	173
	Einkapselung	174
10.2	Klassen sollten klein sein!	174
	Fünf Methoden sind nicht zu viel, oder?	176
	Das Single-Responsibility-Prinzip	176
	Kohäsion	178
	Kohäsion zu erhalten, führt zu vielen kleinen Klassen	179
10.3	Änderungen einplanen	185
	Änderungen isolieren	188
11	Systeme	191
11.1	Wie baut man eine Stadt?	191
11.2	Konstruktion und Anwendung eines Systems trennen	192
	Trennung in main	193
	Factories	194
	Dependency Injection	195
11.3	Aufwärtsskalierung	196
	Cross-Cutting Concerns	199
11.4	Java-Proxies	200
11.5	Reine Java-AOP-Frameworks	202
11.6	AspectJ-Aspekte	205
11.7	Die Systemarchitektur testen	205
11.8	Die Entscheidungsfindung optimieren	207

11.9	Standards weise anwenden, wenn sie nachweisbar einen Mehrwert bieten	207
11.10	Systeme brauchen domänenspezifische Sprachen	208
11.11	Zusammenfassung	208
12	Emergenz	209
12.1	Saubere Software durch emergentes Design.	209
12.2	Einfache Design-Regel 1: Alle Tests bestehen	210
12.3	Einfache Design-Regeln 2–4: Refactoring	211
12.4	Keine Duplizierung	211
12.5	Ausdrucksstärke	214
12.6	Minimale Klassen und Methoden	215
12.7	Zusammenfassung	215
13	Nebenläufigkeit	217
13.1	Warum Nebenläufigkeit?	218
	Mythen und falsche Vorstellungen	219
13.2	Herausforderungen	220
13.3	Prinzipien einer defensiven Nebenläufigkeitsprogrammierung ...	221
	Single-Responsibility-Prinzip	221
	Korollar: Beschränken Sie den Gültigkeitsbereich von Daten	221
	Korollar: Arbeiten Sie mit Kopien der Daten.	222
	Korollar: Threads sollten voneinander so unabhängig wie möglich sein	222
13.4	Lernen Sie Ihre Library kennen	223
	Thread-sichere Collections	223
13.5	Lernen Sie Ihre Ausführungsmodelle kennen	224
	Erzeuger-Verbraucher	224
	Leser-Schreiber	225
	Philosophenproblem	225
13.6	Achten Sie auf Abhängigkeiten zwischen synchronisierten Methoden	226
13.7	Halten Sie synchronisierte Abschnitte klein	226
13.8	Korrekten Shutdown-Code zu schreiben, ist schwer	227
13.9	Threaded-Code testen	227
	Behandeln Sie gelegentlich auftretende Fehler als potenzielle Threading-Probleme	228
	Bringen Sie erst den Nonthreaded-Code zum Laufen	228

	Machen Sie Ihren Threaded-Code pluggable	229
	Schreiben Sie anpassbaren Threaded-Code	229
	Den Code mit mehr Threads als Prozessoren ausführen	229
	Den Code auf verschiedenen Plattformen ausführen	229
	Code-Scheitern durch Instrumentierung provozieren	230
	Manuelle Codierung	230
	Automatisiert	231
13.10	Zusammenfassung	232
14	Schrittweise Verfeinerung	235
14.1	Args-Implementierung	236
	Wie habe ich dies gemacht?	243
14.2	Args: der Rohentwurf	243
	Deshalb hörte ich auf	256
	Über inkrementelle Entwicklung	256
14.3	String-Argumente	258
14.4	Zusammenfassung	300
15	JUnit im Detail	301
15.1	Das JUnit-Framework	301
15.2	Zusammenfassung	316
16	Refactoring von SerialDate	317
16.1	Zunächst bring es zum Laufen!	318
16.2	Dann mach es richtig!	320
16.3	Zusammenfassung	336
17	Smells und Heuristiken	337
17.1	Kommentare	337
	C1: Ungeeignete Informationen	337
	C2: Überholte Kommentare	338
	C3: Redundante Kommentare	338
	C4: Schlecht geschriebene Kommentare	338
	C5: Auskommentierter Code	339
17.2	Umgebung	339
	E1: Ein Build erfordert mehr als einen Schritt	339
	E2: Tests erfordern mehr als einen Schritt	339
17.3	Funktionen	340
	F1: Zu viele Argumente	340
	F2: Output-Argumente	340

	F3: Flag-Argumente	340
	F4: Tote Funktionen	340
17.4	Allgemein	340
	G1: Mehrere Sprachen in einer Quelldatei	340
	G2: Offensichtliches Verhalten ist nicht implementiert.	341
	G3: Falsches Verhalten an den Grenzen	341
	G4: Übergangene Sicherungen	341
	G5: Duplizierung	342
	G6: Auf der falschen Abstraktionsebene codieren	342
	G7: Basisklasse hängt von abgeleiteten Klassen ab.	344
	G8: Zu viele Informationen	344
	G9: Toter Code.	345
	G10: Vertikale Trennung.	345
	G11: Inkonsistenz.	345
	G12: Müll	346
	G13: Künstliche Kopplung.	346
	G14: Funktionsneid	346
	G15: Selektor-Argumente	347
	G16: Verdeckte Absicht.	348
	G17: Falsche Zuständigkeit.	349
	G18: Fälschlich als statisch deklarierte Methoden	350
	G19: Aussagekräftige Variablen verwenden	350
	G20: Funktionsname sollte die Aktion ausdrücken	351
	G21: Den Algorithmus verstehen	351
	G22: Logische Abhängigkeiten in physische umwandeln	352
	G23: Polymorphismus statt If/Else oder Switch/Case verwenden	353
	G24: Konventionen beachten	354
	G25: Magische Zahlen durch benannte Konstanten ersetzen	354
	G26: Präzise sein.	355
	G27: Struktur ist wichtiger als Konvention	356
	G28: Bedingungen einkapseln	356
	G29: Negative Bedingungen vermeiden	356
	G30: Eine Aufgabe pro Funktion!.	356
	G31: Verborgene zeitliche Kopplungen	357
	G32: Keine Willkür	358
	G33: Grenzbedingungen einkapseln	359
	G34: In Funktionen nur eine Abstraktionsebene tiefer gehen	359

	G35: Konfigurierbare Daten hoch ansiedeln	361
	G36: Transitive Navigation vermeiden	362
17.5	Java	362
	J1: Lange Importlisten durch Platzhalter vermeiden	362
	J2: Keine Konstanten vererben	363
	J3: Konstanten im Gegensatz zu Enums	364
17.6	Namen	366
	N1: Deskriptive Namen wählen	366
	N2: Namen sollten der Abstraktionsebene entsprechen	367
	N3: Möglichst die Standardnomenklatur verwenden	368
	N4: Eindeutige Namen	368
	N5: Lange Namen für große Geltungsbereiche	369
	N6: Codierungen vermeiden	369
	N7: Namen sollten Nebeneffekte beschreiben	370
17.7	Tests	370
	T1: Unzureichende Tests	370
	T2: Ein Coverage-Tool verwenden	370
	T3: Triviale Tests nicht überspringen	370
	T4: Ein ignorierte Test zeigt eine Mehrdeutigkeit auf	371
	T5: Grenzbedingungen testen	371
	T6: Bei Bugs die Nachbarschaft gründlich testen	371
	T7: Das Muster des Scheiterns zur Diagnose nutzen	371
	T8: Hinweise durch Coverage-Patterns	371
	T9: Tests sollten schnell sein	371
17.8	Zusammenfassung	372
A	Nebenläufigkeit II	373
A.1	Client/Server-Beispiel	373
	Der Server	373
	Threading hinzufügen	375
	Server-Beobachtungen	375
	Zusammenfassung	377
A.2	Mögliche Ausführungspfade	377
	Anzahl der Pfade	378
	Tiefer graben	380
	Zusammenfassung	383
A.3	Lernen Sie Ihre Library kennen	383
	Executor Framework	383

	Nicht blockierende Lösungen	384
	Nicht thread-sichere Klassen	385
A.4	Abhängigkeiten zwischen Methoden können nebenläufigen Code beschädigen.	387
	Das Scheitern tolerieren	388
	Clientbasiertes Locking	388
	Serverbasiertes Locking	390
A.5	Den Durchsatz verbessern	391
	Single-Thread-Berechnung des Durchsatzes	392
	Multithread-Berechnung des Durchsatzes	393
A.6	Deadlock	393
	Gegenseitiger Ausschluss	395
	Sperren & warten	395
	Keine präemptive Aktion	395
	Zirkuläres Warten	395
	Den gegenseitigen Ausschluss aufheben	396
	Das Sperren & Warten aufheben	396
	Die Präemption umgehen	397
	Das zirkuläre Warten umgehen	397
A.7	Multithreaded-Code testen	398
A.8	Threadbasierten Code mit Tools testen	401
A.9	Zusammenfassung	402
A.10	Tutorial: kompletter Beispielcode	402
	Client/Server ohne Threads	402
	Client/Server mit Threads	406
B	org.jfree.date.SerialDate	407
C	Literaturverweise	463
	Epilog	465
	Stichwortverzeichnis	467