O'REILLY®

Deutsche Ausgabe

Clean Code Kochbuch

Rezepte für gutes Code-Design und bessere Softwarequalität

Maximiliano Contieri
Übersetzung von Jens Olaf Koch

Coypright und Urheberrechte:
Die durch die dpunkt.verlag GmbH vertriebenen digitalen Inhalte sind urheberrechtlich geschützt. Der Nutzer verpflichtet sich, die Urheberrechte anzuerkennen und einzuhalten. Es werden keine Urheber-, Nutzungs- und sonstigen Schutzrechte an den Inhalten auf den Nutzer übertragen. Der Nutzer ist nur berechtigt, den abgerufenen Inhalt zu eigenen Zwecken zu nutzen. Er ist nicht berechtigt, den Inhalt im Internet, in Intranets, in Extranets oder sonst wie Dritten zur Verwertung zur Verfügung zu stellen. Eine öffentliche Wiedergabe oder sonstige Weiterveröffentlichung und eine gewerbliche Vervielfältigung der Inhalte wird ausdrücklich ausgeschlossen. Der Nutzer darf Urheberrechtsvermerke, Markenzeichen und andere Rechtsvorbehalte im abgerufenen Inhalt nicht entfernen.

Clean Code Kochbuch

Rezepte für gutes Code-Design
und bessere Softwarequalität

Maximiliano Contieri

Deutsche Übersetzung von
Jens Olaf Koch

O'REILLY®

Maximiliano Contieri

Lektorat: Ariane Hesse
Übersetzung: Jens Olaf Koch
Korrektorat: Claudia Lötschert, *www.richtiger-text.de*
Satz: III-satz, *www.drei-satz.de*
Herstellung: Stefanie Weidner
Umschlaggestaltung: Karen Montgomery, Michael Oréal, *www.oreal.de*

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

ISBN:
Print 978-3-96009-243-8
PDF 978-3-96010-862-7
ePub 978-3-96010-863-4

1. Auflage 2024
Translation Copyright © 2024 dpunkt.verlag GmbH
Wieblinger Weg 17
69123 Heidelberg

Authorized German translation of the English edition of *Clean Code Cookbook* ISBN 9781098144722
© 2023 Maximiliano Contieri. This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

Dieses Buch erscheint in Kooperation mit O'Reilly Media, Inc. unter dem Imprint »O'REILLY«. O'REILLY ist ein Markenzeichen und eine eingetragene Marke von O'Reilly Media, Inc. und wird mit Einwilligung des Eigentümers verwendet.

Schreiben Sie uns:
Falls Sie Anregungen, Wünsche und Kommentare haben, lassen Sie es uns wissen: *kommentar@oreilly.de*.

Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten. Die Verwendung der Texte und Abbildungen, auch auszugsweise, ist ohne die schriftliche Zustimmung des Verlags urheberrechtswidrig und daher strafbar. Dies gilt insbesondere für die Vervielfältigung, Übersetzung oder die Verwendung in elektronischen Systemen.
Es wird darauf hingewiesen, dass die im Buch verwendeten Soft- und Hardware-Bezeichnungen sowie Markennamen und Produktbezeichnungen der jeweiligen Firmen im Allgemeinen warenzeichen-, marken- oder patentrechtlichem Schutz unterliegen.
Alle Angaben und Programme in diesem Buch wurden mit größter Sorgfalt kontrolliert. Weder Autor noch Übersetzer noch Verlag können jedoch für Schäden haftbar gemacht werden, die in Zusammenhang mit der Verwendung dieses Buches stehen.

Inhalt

Geleitwort . . . 13

Vorwort . . . 15

1 Clean Code . . . 19
Was ist ein Code-Smell? . . . 19
Was ist Refactoring? . . . 20
Was ist ein Rezept? . . . 20
Warum Clean Code? . . . 21
Lesbarkeit, Performance – oder beides? . . . 21
Arten von Software . . . 21
Maschinengenerierter Code . . . 22
Hinweise zu verwendeten Begriffen . . . 22
Entwurfsmuster . . . 22
Paradigmen der Programmiersprachen . . . 23
Objekte versus Klassen . . . 23
Veränderbarkeit . . . 23

2 Festlegung der Axiome . . . 25
Einführung . . . 25
Warum ist es ein Modell? . . . 26
Warum ist es abstrakt? . . . 26
Warum ist es programmierbar? . . . 26
Warum ist es partiell? . . . 27
Warum ist es erklärend? . . . 27
Wieso geht es um Realität? . . . 27
Ableitung der Regeln . . . 28
Das einzig wahre Entwurfsprinzip für Software . . . 28

3 Anämische Modelle **35**
Einführung 35
Anämische Objekte in Rich Objects konvertieren 36
Das Wesentliche Ihrer Objekte erkennen 37
Objekte von Settern befreien 39
Auf Generatoren verzichten, die anämischen Code produzieren 41
Automatische Eigenschaften entfernen 42
DTOs entfernen 44
Leere Konstruktoren vervollständigen 46
Getter entfernen 48
Objektorgie verhindern 50
Dynamische Eigenschaften entfernen 52

4 Primitive Obsession **55**
Einführung 55
Small Objects erstellen 56
Primitive Datentypen in Objekte umwandeln 57
Assoziative Arrays in Objekte umwandeln 58
Keine Strings missbrauchen 60
Zeitstempel in Sequenzierung umwandeln 61
Teilmengen als Objekte modellieren 62
String-Validierungen in Objekte umwandeln 63
Unnötige Eigenschaften entfernen 66
Datumsintervalle berechnen 68

5 Mutabilität **71**
Einführung 71
Variablen von var in const ändern 73
Variablen als veränderlich deklarieren 74
Veränderungen der Objektessenz verbieten 76
Veränderliche const-Arrays vermeiden 77
Lazy Initialization entfernen 78
Veränderliche Konstanten einfrieren 80
Seiteneffekte beseitigen 82
Hoisting verhindern 83

6 Deklarativer Code **85**
Einführung 85
Geltungsbereich wiederverwendeter Variablen begrenzen 86
Code durch Aufteilung in Funktionen strukturieren 87
Versionierte Methoden entfernen 88
Doppelte Verneinungen entfernen 89
Falsch zugeordnete Verantwortlichkeiten verschieben 90

Explizite Iterationen ersetzen 92
Entwurfsentscheidungen dokumentieren 93
Magische Zahlen durch Konstanten ersetzen 94
»Was« und »Wie« trennen 95
Reguläre Ausdrücke dokumentieren 96
Yoda-Conditions neu formulieren 97
Scherzhaft benannte Methoden umbenennen 98
Callback Hell vermeiden 98
Gute Fehlermeldungen formulieren 100
Magische Korrekturen vermeiden 102

7 Namensgebung 105
Einführung 105
Abkürzungen ausschreiben 105
Hilfsfunktionen und -klassen umbenennen und aufteilen 107
myObjects umbenennen 110
Ergebnisvariablen umbenennen 111
Variablen umbenennen, die nach Typen benannt sind 112
Zu lange Namen kürzen 114
Abstrakte Namen ändern 115
Rechtschreibfehler korrigieren 116
Klassennamen aus Attributen entfernen 116
Vorangestellte Buchstaben aus Namen von Klassen und Interfaces entfernen 117
Basic-/Do-Funktionen umbenennen 118
Mit Pluralformen benannte Klassen auf Singularform ändern 120
»Collection« aus Namen entfernen 120
Präfix/Suffix »Impl« aus Klassennamen entfernen 121
Argumente je nach Rolle bzw. Aufgabe benennen 122
Redundante Parameternamen ändern 123
Überflüssigen Kontext aus Namen entfernen 124
Benennung als »data« vermeiden 126

8 Kommentare 127
Einführung 127
Kommentierten Code entfernen 127
Veraltete Kommentare entfernen 129
Temporäre logische Steuerungsanweisungen entfernen 130
Getter-Kommentare entfernen 132
Kommentare in Funktionsnamen umwandeln 133
Kommentare innerhalb von Methoden entfernen 134
Kommentare durch Tests ersetzen 136

9 Standards . . . **139**
Einführung . . . 139
Codestandards befolgen . . . 139
Einrückungen standardisieren . . . 142
Schreibweisen vereinheitlichen . . . 143
Code auf Englisch schreiben . . . 144
Reihenfolge von Parametern vereinheitlichen . . . 145
Kleine Mängel beheben . . . 146

10 Komplexität . . . **149**
Einführung . . . 149
Wiederholten Code entfernen . . . 149
Einstellungen/Konfigurationen und Funktionsumschalter entfernen . . . 151
Zustand über Eigenschaften ändern . . . 153
Übertriebene Raffinesse aus dem Code entfernen . . . 155
Mehrere Promises parallel auflösen . . . 156
Lange Kollaborationsketten auflösen . . . 157
Methode in ein Objekt extrahieren . . . 159
Array-Konstruktoren überprüfen . . . 161
Poltergeist-Objekte entfernen . . . 162

11 Aufgeblähter Code . . . **165**
Einführung . . . 165
Überlange Methoden unterteilen . . . 165
Überflüssige Argumente reduzieren . . . 167
Überflüssige Variablen reduzieren . . . 168
Überflüssige Klammern entfernen . . . 170
Überflüssige Methoden entfernen . . . 171
Überzählige Attribute gruppieren . . . 172
Importlisten kürzen . . . 174
Funktionen mit mehreren Aufgaben aufteilen . . . 175
Überladene Interfaces verschlanken . . . 176

12 YAGNI-Prinzip . . . **179**
Einführung . . . 179
Toten Code entfernen . . . 179
Code anstelle von Diagrammen verwenden . . . 181
Refactoring von Klassen mit nur einer Unterklasse . . . 183
Interfaces entfernen, die nur an einer Stelle genutzt werden . . . 184
Missbräuchlich verwendete Entwurfsmuster entfernen . . . 185
Geschäftsspezifische Collections ersetzen . . . 186

13 Fail-Fast-Prinzip 189
Einführung 189
Neuzuweisung von Variablen refaktorieren 189
Vorbedingungen durchsetzen 191
Striktere Parameter verwenden 193
Standardfall bei Switch-Anweisungen entfernen 194
Beim Iterieren über Collections Änderungen vermeiden 196
Hash und Gleichheit neu definieren 197
Refactoring von funktionalen Änderungen trennen 198

14 If-Anweisungen 201
Einführung 201
Akzidentelle If-Anweisungen durch Polymorphie ersetzen 202
Flag-Variablen für Ereignisse deklarativ umbenennen 208
Boolesche Variablen reifizieren 209
Switch-/Case-/Elseif-Anweisungen ersetzen 211
Hartcodierte Bedingungen durch Collections ersetzen 213
Boolesche Bedingungen in Kurzschluss-Auswertungen umwandeln 214
Implizites Else in explizites If umwandeln 215
Verschachtelte Bedingungen refaktorieren 216
Short-Circuit-Hacks vermeiden 218
Verschachtelten Pfeilcode refaktorieren 219
Rückgabe boolescher Werte für Bedingungsprüfungen vermeiden 220
Vergleiche mit booleschen Werten ändern 222
Ternäre Ausdrücke vereinfachen 223
Nicht-polymorphe Funktionen in polymorphe umwandeln 225
Gleichheitsvergleich ändern 226
Hartcodierte Geschäftsbedingungen reifizieren 227
Überflüssige boolesche Operatoren entfernen 228
Verschachtelte ternäre Ausdrücke refaktorieren 229

15 Nullwerte 231
Einführung 231
Nullobjekte erstellen 231
Optionale Verkettungen entfernen 234
Optionale Attribute in eine Collection umwandeln 236
Reale Objekte für Nullwerte verwenden 238
Darstellung unbekannter Orte ohne Verwendung von null 241

16 Vorzeitige Optimierung 245
Einführung 245
IDs für Objekte vermeiden 246
Vorzeitige Optimierungen entfernen 248

Bitweise vorzeitige Optimierungen entfernen 250
Übergeneralisierung reduzieren 251
Strukturelle Optimierungen ändern 252
»Boat Anchors« beseitigen 253
Caches aus Domänenobjekten extrahieren 255
Auf der Implementierung basierende Callback-Events entfernen 257
Abfragen aus Konstruktoren entfernen 258
Code aus Destruktoren entfernen 259

17 Kopplung 263
Einführung 263
Versteckte Annahmen explizit machen 263
Singletons ersetzen 265
God Objects aufspalten 268
Klassen bei divergenten Änderungen teilen 270
Spezielle als Flags genutzte Werte (wie 9999) in normale Werte umwandeln 271
Shotgun Surgery vermeiden 273
Optionale Argumente entfernen 275
Feature Envy vorbeugen 276
Vermittlerobjekte entfernen 278
Standardargumente ans Ende verschieben 279
Ripple-Effekt vermeiden 281
Zufällige Methoden aus Geschäftsobjekten entfernen 282
Geschäftslogik aus der Benutzeroberfläche entfernen 284
Kopplung an Klassen verringern 287
Datenklumpen beseitigen 289
Unangemessene Intimität auflösen 290
Fungible Objekte konvertieren 292

18 Globals 295
Einführung 295
Globale Funktionen reifizieren 295
Statische Funktionen reifizieren 296
Goto-Anweisungen durch strukturierten Code ersetzen 298
Globale Klassen entfernen 299
Globale Datumserstellung anpassen 301

19 Hierarchien 303
Einführung 303
Tiefe Vererbungshierarchien verflachen 303
Jo-Jo-Hierarchien durchbrechen 306
Subklassifizierung zur Wiederverwendung von Code auflösen 307
»Ist-ein«-Beziehung durch Verhalten ersetzen 309

Verschachtelte Klassen entfernen 311
Isolierte Klassen umbenennen 313
Konkrete Klassen als final deklarieren 314
Klassenvererbung explizit definieren 316
Klassen ohne Verhalten entfernen 317
Keine vorzeitige Klassenbildung vornehmen 318
Geschützte Attribute entfernen 320
Leere Implementierungen vervollständigen 322

20 Testen 325
Einführung 325
Private Methoden testen 326
Beschreibungen zu Assertions hinzufügen 327
assertTrue in spezifischere Assertions konvertieren 329
Mock-Objekte durch echte Objekte ersetzen 330
Generische Assertions verfeinern 332
Unzuverlässige Tests entfernen 333
Vergleiche von Gleitkommazahlen vermeiden 335
Realistische Daten statt Testdaten verwenden 336
Verletzung der Kapselung vermeiden 338
Irrelevante Testinformationen entfernen 340
Keine Pull Requests ohne Testabdeckung zulassen 341
Tests umformulieren, die von Datumswerten abhängen 343
Eine neue Programmiersprache lernen 344

21 Technische Schulden 345
Einführung 345
Produktionsabhängigen Code entfernen 346
Fehlertracker entfernen 347
Warnungen/Strict-Modus nicht ausschalten 349
ToDos und FixMes verhindern und entfernen 350

22 Ausnahmen 353
Einführung 353
Leere Ausnahmeblöcke entfernen 353
Unnötige Ausnahmen entfernen 354
Keine Ausnahmen bei erwarteten Fällen auslösen 356
Verschachtelte Try/Catch-Blöcke vereinfachen 358
Rückgabecodes durch Ausnahmen ersetzen 359
Verschachtelten Pfeilcode refaktorieren 361
Low-Level-Fehler vor Endbenutzern verstecken 362
Try-Blöcke kurz halten 363

23 Metaprogrammierung 365
Einführung 365
Metaprogrammierung entfernen 365
Anonyme Funktionen reifizieren 369
Auf Präprozessoren verzichten 371
Dynamische Methoden entfernen 372

24 Datentypen 375
Einführung 375
Typprüfungen entfernen 375
Mit truthy-Werten umgehen 377
Gleitkommazahlen in Dezimalzahlen konvertieren 380

25 Sicherheit 383
Einführung 383
Benutzereingaben bereinigen 383
Sequenzielle IDs ändern 385
Paketabhängigkeiten entfernen 386
Problematische reguläre Ausdrücke ersetzen 388
Sichere Deserialisierung von Objekten 389

Glossar 391

Index 405

Geleitwort

»Software is eating the world.« Diese Aussage von Marc Andreessen ist eines meiner Lieblingszitate, fast schon ein Meme, das den gegenwärtigen Stand der Dinge beschreibt. Nie zuvor in der Geschichte der Menschheit gab es so viel Software wie heute, und noch nie zuvor war sie für so viele Aspekte des Lebens zuständig. Wer in einem urbanen Umfeld lebt, ist fast nur noch von Dingen umgeben, die durch Software gesteuert werden. Jedes Jahr übertragen wir mehr und mehr Kontrolle an Software. Und durch den explosiven und disruptiven Durchbruch der künstlichen Intelligenz wird dieser Trend nur noch verstärkt: Die KIs, mit denen wir interagieren, sind ebenfalls Software.

Ich gehöre zu einer Generation, in der es normal war, zuerst Programmierer zu werden, bevor man zu einem Anwender von Software wurde. Mit 16 fing ich an, meine ersten kleinen Programme zu schreiben. Mit 18 begann ich, an umfangreicheren Systemen zu arbeiten, und sah mich mit der grundlegenden Tatsache konfrontiert, die auch den Autor zu diesem willkommenen und notwendigen Buch motiviert hat: dass die Software, von der immer mehr abhängt, von Menschen geschrieben wird, als Code. Und die Qualität dieses Codes beeinflusst direkt die Qualität der Software, ihre Wartbarkeit, Lebensdauer, ihre Kosten und Leistung.

Bei der Arbeit an diesen frühen Systemen, die von sehr kleinen Teams von zwei oder drei Personen programmiert wurden, lernte ich auf die harte Tour, warum es so wichtig ist, Clean Code – sauberen Code – zu schreiben. Hätte ich damals ein Buch wie dieses gehabt, hätte ich viel Zeit sparen können.

Das bedeutet nicht, dass dieses Buch nur in den prähistorischen Zeiten der Informatik relevant gewesen wäre oder dass seine Zielgruppe aus Programmieranfängern besteht, denen die Grundlagen beigebracht werden müssen. Ganz im Gegenteil.

In diesem Buch finden Sie – ganz im Stil eines Kochbuchs – zahlreiche einfache Rezepte, um eine Vielzahl von Fallstricken und häufigen Fehlerquellen beim Programmieren zu vermeiden. Oft ist nicht das Rezept der wertvollste Teil, sondern die Tatsache, dass ein bestimmtes Thema angesprochen und diskutiert wird. Das gibt uns eine Grundlage, um darüber nachzudenken, wie wir dieses Problem in unserem Code lösen und die Sauberkeit – die »Cleanliness« – unserer Lösung bewerten können. Maximiliano Contieris Schreibstil ist klar und direkt, genau wie wir uns unse-

ren Code wünschen. Jedes Rezept enthält Codebeispiele, um mögliche Zweifel an dessen korrekter Anwendung zu beseitigen.

Es scheint, als ob Sauberkeit und Klarheit des Codes nur Sache der Programmierer wären. Tatsächlich beginnen die Probleme aber schon lange vorher, in der Entwurfsphase, und erstrecken sich bis zur Bereitstellung von Ressourcen, den Entwicklungsrichtlinien, dem Projekt- und Teammanagement, den Wartungs- und Entwicklungsphasen. Ich glaube, dass fast alle in der Softwarebranche von diesem Buch profitieren können, weil es eine Vielzahl von häufigen Problemen im Code – also in dem »Material«, aus dem Software gemacht wird: die Software, die die Welt »auffrisst« – sehr gut veranschaulicht und erklärt.

Man könnte denken, das Schreiben von Code sei überholt und KI und große Sprachmodelle würden das Programmieren bald ohne jegliches menschliches Zutun übernehmen. Den Beispielen zufolge, die mir täglich begegnen, sieht die Realität noch anders aus. Das Ausmaß der Halluzinationen (grundlegende Fehler, Interpretationsprobleme, Schwachstellen und Wartungsprobleme), die der von KI geschriebene Code aufweist, verhindern das. Wir befinden uns jedoch ganz eindeutig in einer Übergangsphase, in der »technologische Zentauren« von großer Bedeutung sein werden, also erfahrene Programmierer, die den von automatisierten Systemen produzierten Code überwachen, korrigieren und verbessern. Und solange Menschen Code lesen und pflegen müssen, ist Clean Code, so wie er in diesem Buch vermittelt wird, essenziell.

– Carlos E. Ferro
Lic. en Ciencias de la computación
Senior Software Engineer bei Quorum Software
Buenos Aires
20. Juni 2023

Vorwort

Code ist überall zu finden, von der Webentwicklung bis hin zu Smart Contracts, in eingebetteten Systemen, Blockchains, der Steuerungssoftware des James-Webb-Weltraumteleskops, chirurgischen Robotern und in vielen anderen Anwendungen. Software erobert tatsächlich die Welt, und wir erleben derzeit den Aufstieg professioneller KI-Tools zur Codeerzeugung. Das macht Clean Code wichtiger denn je. Da wir mit immer größeren proprietären oder Open-Source-Codebasen arbeiten, ist Clean Code der Schlüssel, um die Codebasis frisch und entwicklungsfähig zu halten.

Für wen dieses Buch gedacht ist

Dieses Buch hilft Ihnen, häufig auftretende Probleme im Code zu identifizieren, erklärt die Konsequenzen dieser Probleme und bietet einfache, nachvollziehbare Rezepte, um sie zu vermeiden. Es unterstützt Programmierer, Code-Reviewer, Softwarearchitekten und Studierende dabei, ihre Programmierfähigkeiten zu verbessern und damit auch die bereits bestehende Software.

Wie dieses Buch aufgebaut ist

Dieses Buch umfasst 25 Kapitel. Jedes Kapitel beginnt mit einigen Prinzipien und Grundlagen, die die Vorteile von Clean Code, seine Konsequenzen und die Nachteile aufzeigen, die bei falscher Anwendung entstehen. Das erste Kapitel behandelt die wesentliche Grundregel für Clean Code: Bilden Sie in Ihrem Design reale Entitäten 1:1 ab. Diese Regel dient als Basis, von der aus alle anderen Prinzipien abgeleitet werden können.

In jedem Kapitel finden Sie mehrere thematisch gruppierte Rezepte, die Werkzeuge und Tipps zur Änderung Ihres Codes enthalten. Alle Rezepten zielen darauf ab, positive Veränderungen und Verbesserungen an Ihrem aktuellen Code vorzunehmen. Neben den Rezepten und Beispielen werden verschiedene Designprinzipien, Heuristiken und Regeln vorgestellt. Die Rezepte enthalten Codebeispiele in verschiedenen Programmiersprachen, denn Clean Code ist keine Eigenschaft, die sich einer bestimmten Programmiersprache zuordnen lässt. Viele Bücher über Refactoring kon-

zentrieren sich auf eine einzige Sprache, und bei aktualisierten Neuausgaben beschäftigen sich die Autoren gerne mit der jeweils neuesten, angesagten Programmiersprache. Dieses Buch ist dagegen sprachagnostisch angelegt, und die meisten Rezepte gelten universell (es sei denn, es ist wird ausdrücklich etwas anderes angegeben).

Lesen Sie den Code als Pseudocode, auch wenn er größtenteils in der dargestellten Form lauffähig ist. Wenn ich mich zwischen Lesbarkeit und Performance entscheiden muss, wähle ich stets die Lesbarkeit. Im Laufe des Buchs definiere ich gängige Begriffe, aber Sie finden sie auch gesammelt im »Glossar«.

Was Sie zur Verwendung dieses Buchs benötigen

Um die Codebeispiele auszuführen, benötigen Sie eine Arbeitsumgebung wie die O'Reilly-Sandboxes (*https://learning.oreilly.com/interactive*) oder Replit (*https://replit.com*). Ich möchte Sie ermutigen, die Codebeispiele in Ihre bevorzugte Programmiersprache zu übersetzen. Das lässt sich mit KI-Codegeneratoren heutzutage kostenlos erledigen. Um die Codebeispiele in diesem Buch zu erstellen, habe ich Tools wie GitHub Copilot, OpenAI Codex, Gemini, ChatGPT und viele andere verwendet. Dadurch konnte ich in diesem Buch mehr als 25 verschiedene Sprachen verwenden, auch wenn ich in vielen davon kein Experte bin.

Zugang zur Digitalversion dieses Buchs

Eine kostenlose englischsprachige Version als stets verfügbare, durchsuchbare Onlineausgabe finden Sie unter *https://www.cleancodecookbook.com*.

Konventionen, die in diesem Buch verwendet werden

In diesem Buch werden die folgenden typografischen Konventionen verwendet:

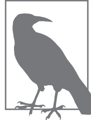
Kursiv
: Zeigt neue Begriffe, URLs, E-Mail-Adressen, Dateinamen und Dateierweiterungen an.

`Nichtproportionalschrift`
: Wird für Programmlistings verwendet, aber auch innerhalb von Absätzen, um dort auf Programmelemente wie Variablen- oder Funktionsnamen, Datenbanken, Datentypen, Umgebungsvariablen, Anweisungen und Schlüsselwörter zu verweisen.

`Fette Nichtproportionalschrift`
: Zeigt Befehle oder anderen Text an, der vom Benutzer wortgetreu eingegeben werden muss.

`Kursive Nichtproportionalschrift`
: Zeigt Text an, der durch Benutzereingaben oder durch kontextabhängige Werte ersetzt werden soll.

Dieses Element weist auf einen Tipp oder Vorschlag hin.



Dieses Element kennzeichnet einen allgemeinen Hinweis.

Dieses Element weist auf eine Warnung hin.

Verwendung von Codebeispielen

Zusätzliches Material wie Codebeispiele und Übungen finden Sie zum Download unter: *https://github.com/mcsee/clean-code-cookbook*.

Bei technischen Fragen oder Problemen mit den Codebeispielen wenden Sie sich bitte per E-Mail an *bookquestions@oreilly.com*.

Dieses Buch soll Ihnen bei Ihren Aufgaben helfen. Sie dürfen den in diesem Buch bereitgestellten Beispielcode in Ihren Programmen und Ihrer Dokumentation verwenden, ohne uns um Erlaubnis bitten zu müssen, außer Sie reproduzieren einen wesentlichen Teil des Codes. Wenn Sie beispielsweis ein Programm schreiben, das mehrere Teile des Codes aus diesem Buch nutzt, benötigen Sie keine Genehmigung. Der Verkauf oder die Verbreitung von Beispielen aus O'Reilly-Büchern erfordert dagegen eine Genehmigung. Die Beantwortung einer Frage durch das Zitieren dieses Buchs und von Beispielcode ist nicht genehmigungspflichtig. Das Einbinden einer erheblichen Menge an Beispielcode aus diesem Buch in die Dokumentation Ihres Produkts erfordert dagegen eine Genehmigung.

Wir freuen uns über eine Quellenangabe, verlangen sie aber im Allgemeinen nicht. Eine Quellenangabe sollte in der Regel den Titel, den Autor, den Verlag und die ISBN umfassen. Zum Beispiel: Clean Code Kochbuch von Maximiliano Contieri (O'Reilly). Copyright 2024 Maximiliano Contieri.

Wenn Sie der Meinung sind, dass Ihre Verwendung von Codebeispielen nicht unter die Fair-Use-Regelung oder die oben genannte Erlaubnis fällt, können Sie uns bitte unter *permissions@oreilly.com* kontaktieren.

Danksagungen

Dieses Buch ist meiner Frau Maria Virginia gewidmet, die mich stets liebevoll unterstützt hat, sowie meinen geliebten Töchtern Malena und Miranda und meinen Eltern Juan Carlos und Alicia.

Ich bin Maximo Prieto und Hernan Wilkinson, die wesentlich zu den Ideen beigetragen haben, die in diesem Buch vorgestellt werden, zu großem Dank für ihre wertvollen Einsichten und ihr Fachwissen verpflichtet. Mein Dank gilt auch meinen Kollegen von Ingenieria de Software dafür, dass sie ihre Ideen mit mir geteilt haben, und meinen Kolleginnen und Kollegen an der Ciencias Exactas, der Fakultät für Exakte und Naturwissenschaften der Universidad de Buenos Aires, für ihr geteiltes Wissens und ihrer Erfahrung über viele Jahre.

Abschließend möchte ich den technischen Gutachtern Luben Alexandrov, Daniel Moka und Carlos E. Ferro sowie meiner Lektorin Sara Hunter danken, deren Betreuung und Ratschläge dieses Buch erheblich verbessert haben.

KAPITEL 1

Clean Code

Als Martin Fowler in seinem Buch *Refactoring: Improving the Design of Existing Code* (dt. *Refactoring: Wie Sie das Design bestehender Software verbessern*) den Begriff »Refactoring« definierte, beschrieb er dessen Stärken und Vorteile und die Gründe für Refaktorierungen. Glücklicherweise kennen die meisten Entwickler mehr als zwei Jahrzehnten später die Bedeutung von Refactoring und Code-Smells. Entwickler kämpfen täglich mit technischen Schulden, und Refactoring ist zu einem zentralen Bestandteil der Softwareentwicklung geworden. In seinem grundlegenden Buch nutzt Fowler Refactorings, um Code-Smells zu eliminieren. Dieses Buch stellt einige dieser Refactorings in Form semantischer Rezepte vor, mit denen Sie Ihre eigenen Lösungen verbessern können.

Was ist ein Code-Smell?

Ein Code-Smell, wörtlich ein »übler Codegeruch«, signalisiert ein Problem im Code. Oft deuten Menschen das Vorhandensein von Code-Smells als Indiz dafür, dass das gesamte System auseinandergenommen und neu aufgebaut werden muss. Das widerspricht aber dem Geist der ursprünglichen Definition. Code-Smells sind lediglich Indikatoren für mögliche Verbesserungen. Ein Code-Smell verrät nicht direkt, was falsch ist, er mahnt nur zu besonderer Vorsicht.

Die in diesem Buch vorgestellten Rezepte bieten Lösungen für diese Symptome und ihre zugrunde liegenden Probleme. Wie in jedem Kochbuch sind auch diese Rezepte Vorschläge und die Hinweise zu Code-Smells Leitlinien und Heuristiken, keine starren Regeln. Bevor Sie ein Rezept blind anwenden, sollten Sie die zugrunde liegenden Probleme verstehen und die Vor- und Nachteile Ihres Entwurfs und Codes abwägen. Zu einem guten Codedesign gehört es, Leitlinien mit praktischen Überlegungen und dem jeweiligen Kontext in Einklang zu bringen.

Was ist Refactoring?

Fowler liefert in seinem Buch zwei sich ergänzende Definitionen:

> *Refactoring (Substantiv)*: eine Änderung der internen Struktur einer Software, um sie verständlicher und kostengünstiger veränderbar zu machen, ohne ihr beobachtbares Verhalten zu ändern.
>
> *refaktorieren (Verb)*: Software durch die Anwendung einer Serie von Refactorings zu restrukturieren, ohne ihr beobachtbares Verhalten zu verändern.

Der Begriff »Refactoring« wurde 1990 von Ralph Johnson und William Opdyke geprägt und im Jahr 1992 zum Thema von Opdykes Dissertation, »Refactoring Object-Oriented Frameworks« (*https://oreil.ly/zBCkI*). Populär wurden Refactorings später durch Fowlers Buch. Seit Fowlers Definition hat sich viel in diesem Bereich getan. Die meisten integrierten Entwicklungsumgebungen (IDEs) unterstützen heutzutage automatische Refactorings und führen strukturelle Änderungen sicher durch, ohne das Systemverhalten zu verändern. Dieses Buch enthält viele Rezepte für solche automatischen, sicher anwendbaren Refactorings. Es bietet darüber hinaus auch semantische Refactorings an, die allerdings nicht automatisch sicher sind, da sie das Systemverhalten ändern können. Seien Sie also bei der Anwendung von Rezepten mit semantischen Refaktorierungen vorsichtig, da sie Ihre Software in Mitleidenschaft ziehen können. Ich werde in entsprechenden Fällen darauf hinweisen. Wenn das Verhalten Ihres Codes durch Tests umfassend abgedeckt ist, können Sie sicher sein, dass wichtige Geschäftsszenarien nicht beeinträchtigt werden. Refactoring-Rezepte sollten nicht gleichzeitig mit Fehlerkorrekturen oder der Entwicklung neuer Funktionen angewendet werden.

Die meisten modernen Unternehmen nutzen in ihren CI/CD-Pipelines leistungsstarke Testsuiten. Sehen Sie sich auch *Software Engineering at Google* von Titus Winters et al. (O'Reilly 2020) an, um herauszufinden, ob diese Testabdeckungssuiten bei Ihnen vorhanden sind.

Was ist ein Rezept?

Ich verwende den Begriff »Rezept« in einem eher lockeren Sinne. Ein Rezept besteht aus einer Reihe von Anweisungen, um etwas zu erstellen oder zu verändern. Die Rezepte in diesem Buch funktionieren am besten, wenn Sie die dahinterstehende Absicht erfassen, es aber auf Ihre eigene Art und Weise umsetzen. Andere Kochbücher in dieser Reihe sind konkreter und enthalten Schritt-für-Schritt-Lösungen. Um die Rezepte dieses Buchs zu verwenden, müssen Sie sie in die von Ihnen genutzte Programmiersprache und für Ihr Softwaredesign übersetzen. Die Rezepte sollen Ihnen helfen, ein Problem zu verstehen, die damit verbundenen Konsequenzen zu erkennen und Ihren Code zu verbessern.

Warum Clean Code?

Clean Code – guter, sauber formulierter Code – lässt sich leicht lesen, verstehen und warten. Er ist gut strukturiert, präzise und verwendet aussagekräftige Bezeichnungen für Variablen, Funktionen und Klassen. Er orientiert sich zudem an etablierten Best Practices und Entwurfsmustern, wobei Lesbarkeit und Verhalten Vorrang vor Performance und Implementierungsdetails haben.

Clean Code ist in allen sich weiterentwickelnden Systemen, in denen tägliche Änderungen stattfinden, sehr wichtig. Das gilt ganz besonders für Umgebungen, in denen es nicht möglich ist, beliebig schnell Updates durchzuführen, wie etwa bei Embedded Systems, Raumsonden, Smart Contracts, mobilen Apps (zum Beispiel wegen Verzögerungen durch interne Reviews der Store-Anbieter).

Während sich klassische Refactoring-Bücher, Websites und IDEs auf Refaktorierungen konzentrieren, die das Systemverhalten nicht ändern, enthält dieses Buch sowohl Rezepte für solche Szenarien (z.B. sichere Umbenennungen), aber auch semantische Refactorings, bei denen sich die Art und Weise ändert, in der bestimmte Probleme gelöst werden. Es ist wichtig, den Code, die zugrunde liegenden Probleme und die Intention des Rezepts zu verstehen, um angemessene Anpassungen vornehmen zu können.

Lesbarkeit, Performance – oder beides?

Dieses Buch dreht sich um Clean Code. Einige der Rezepte mögen nicht zu den herausragend performanten Lösungen gehören. Im Zweifel ziehe ich aber die Lesbarkeit der Leistung vor. Deshalb widme ich ein ganzes Kapitel (Kapitel 16) der vorzeitigen Optimierung, bei der Leistungsprobleme adressiert werden, ohne dass für die Notwendigkeit der Optimierung ausreichende Evidenz vorliegt.

Für leistungsabhängige, unternehmenskritische Aufgaben besteht die beste Strategie darin, zunächst Clean Code zu schreiben und diesen mit Tests abzudecken, um anschließend die Engpässe unter Anwendung des Pareto-Prinzips zu verbessern. Dieses Prinzip, angewendet auf Software, besagt, dass man die Performance um 80% steigern kann, wenn man 20% der kritischen Engpässe bearbeitet.

Diese Herangehensweise wirkt der Tendenz entgegen, vorzeitige Optimierungen vorzunehmen, ohne dass dafür ausreichend Evidenz vorliegt. Vorzeitige Optimierungen führen meist nur zu geringen Verbesserungen und können Clean Code beeinträchtigen.

Arten von Software

Die meisten Rezepte in diesem Buch zielen auf Backend-Systeme mit komplexer Geschäftslogik. Der Simulator, dessen Konstruktion wir in Kapitel 2 beginnen werden, eignet sich perfekt dafür. Da die Rezepte domänenunabhängig sind, können Sie die meisten auch bei der Frontend-Entwicklung, für Datenbanken, Embedded Systems,

Blockchains und in vielen andere Szenarien anwenden. Es gibt zudem spezifische Rezepte mit Codebeispielen für UX, Frontend, Smart Contracts und andere Bereiche (siehe beispielsweise »Low-Level-Fehler vor Endbenutzern verstecken« auf Seite 362).

Maschinengenerierter Code

Muss man denn auch heute, da es eine Vielzahl von Tools für computergenerierten Code gibt, noch über Clean Code nachdenken? Die Antwort ist ein klares Ja. Sogar mehr denn je. Für die Programmierung gibt es viele kommerzielle Codeassistenten. Allerdings haben sie (noch) nicht die vollständige Kontrolle: Sie agieren als Copiloten und Helfer, aber die Designentscheidungen trifft weiterhin der Mensch.

Zum Zeitpunkt der Erstellung dieses Buchs erzeugen die meisten kommerziellen und KI-Tools anämische Lösungen und Standardalgorithmen. Sie sind jedoch erstaunlich nützlich, wenn einem gerade nicht einfällt, wie man eine kleine, spezifische Funktion erstellt, oder um zwischen Programmiersprachen zu übersetzen. Beim Schreiben dieses Buchs habe ich solche Werkzeuge intensiv genutzt. Ich beherrsche die mehr als 25 Programmiersprachen, die ich in den Rezepten verwendet habe, nicht alle selbst. Viele Codebeispiele habe ich mit der Unterstützung einer Vielzahl von Assistenztools in verschiedene Sprachen übersetzt und getestet. Ich möchte auch Sie dazu einladen, alle verfügbaren Werkzeuge zu nutzen, um die Rezepte dieses Buchs in Ihre bevorzugte Sprache zu übersetzen. Diese KI-Werkzeuge sind gekommen, um zu bleiben, und zukünftige Entwickler werden technologische Zentauren sein: halb Mensch, halb Maschine.

Hinweise zu verwendeten Begriffen

Im gesamten Buch verwende ich die folgenden Begriffe austauschbar:

- Methoden/Funktionen/Prozeduren
- Attribute/Instanzvariablen/Eigenschaften
- Protokoll/Verhalten/Interface/Schnittstelle
- Argumente/Kollaborateure/Parameter
- Anonyme Funktionen/Closures/Lambdas

Die Unterschiede zwischen diesen Begriffen sind subtil und teilweise sprachabhängig. Sofern notwendig, werde ich zur Klärung einen Hinweis hinzufügen.

Entwurfsmuster

In diesem Buch setze ich voraus, dass Sie als Leser über ein grundlegendes Verständnis objektorientierter Designkonzepte verfügen. Einige Rezepte basieren auf populären Entwurfsmustern, einschließlich jener, die im Buch »Gang of Four« *Design Patterns* vorgestellt werden. Andere Rezepte nutzen weniger bekannte Muster wie das *Nullobjekt* oder das *Methodenobjekt*. Darüber hinaus bietet dieses Buch Erklärungen und An-

leitungen zum Ersetzen von Mustern, die heute als Antipatterns angesehen werden, wie beispielsweise das *Singleton*-Muster in »Singletons ersetzen« auf Seite 265.

Paradigmen der Programmiersprachen

David Farley schreibt:

> Die Besessenheit unserer Branche von Sprachen und Tools hat unserem Beruf geschadet. Das bedeutet nicht, dass es keine Fortschritte im Sprachdesign gibt. Doch die meisten Veränderungen in diesem Bereich scheinen sich auf die falschen Dinge zu konzentrieren, wie zum Beispiel syntaktische statt strukturelle Verbesserungen.

Die Clean-Code-Konzepte, die in diesem Buch vorgestellt werden, sind auf eine Vielzahl von Programmierparadigmen anwendbar. Viele dieser Ideen haben ihre Wurzeln in der strukturierten und der funktionalen Programmierung, während andere aus der objektorientierten Welt stammen. Diese Konzepte können Ihnen helfen, im Rahmen jedes Paradigmas eleganteren und effizienteren Code zu schreiben.

Die meisten Rezepte werde ich in objektorientierten Sprachen formulieren und einen Simulator namens *MAPPER* bauen, bei dem ich Objekte als Metaphern für reale Entitäten verwende. Ich werde im Verlauf des Buchs häufig auf den MAPPER Bezug nehmen. Viele Rezepte werden Sie ermutigen, an verhaltensorientierten und deklarativen Code (siehe Kapitel 6) statt an Implementierungscode zu denken.

Objekte versus Klassen

In den meisten Rezepten dieses Buchs geht es um Objekte und nicht um Klassen – obwohl es ein ganzes Kapitel über Klassenbildung gibt: Kapitel 19. Zum Beispiel lautet ein Rezept »Das Wesentliche Ihrer Objekte erkennen« auf Seite 37 und nicht »Das Wesentliche Ihrer Klassen erkennen«. In diesem Buch geht es um die Verwendung von Objekten zur Abbildung realer Gegenstände.

Sie können diese Objekte auf beliebige Art und Weise erstellen: durch Klassenbildung, Prototyping, Fabriken, Klonen usw. In Kapitel 2 erörtere ich die Bedeutung des Mappings Ihrer Objekte und die Notwendigkeit, Dinge zu modellieren, die Sie in der realen Welt beobachten können. Um Objekte zu erstellen, verwenden viele Sprachen *Klassen*, die künstliche bzw. abstrakte Konstrukte und in der realen Welt nicht offensichtlich sind. Natürlich benötigen Sie Klassen, wenn Sie eine klassenbasierte Programmiersprache verwenden. Sie stehen aber nicht im Mittelpunkt der Rezepte.

Veränderbarkeit

Bei Clean Code geht es nicht nur darum, dass Ihre Software korrekt arbeitet. Er soll auch gewährleisten, dass sie leicht zu warten und weiterzuentwickeln ist. Dave Farley betont in *Modern Software Engineering* (dt. *Modernes Software Engineering*), dass Sie darin versiert sein müssen, kontinuierlich zu lernen und Software so zu gestalten,

dass sie flexibel anpassbar ist. Das stellt eine bedeutende Herausforderung für die Technologiebranche dar, und ich hoffe, dass Ihnen dieses Buch dabei helfen kann, mit diesen Entwicklungen Schritt zu halten.

KAPITEL 2
Festlegung der Axiome

Einführung

Eine gängige Definition von Software (*https://oreil.ly/MqGxG*) lautet:

> Die Befehle, die ein Computer ausführt, im Gegensatz zur Hardware, dem physischen Gerät, auf dem diese Befehle laufen.

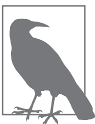
Software wird hier durch ihr Gegenteil definiert: Sie ist demnach alles, was keine Hardware ist. Das ist keine besonders gute Beschreibung ihres Wesens. Eine weitere populäre Definition (*https://oreil.ly/SVbXv*) lautet:

> Software besteht aus Anweisungen, die einem Computer mitteilen, was er tun soll. Sie umfasst die Gesamtheit der Programme, Verfahren und Routinen, die mit dem Betrieb eines Computersystems zusammenhängen. Der Begriff »Software« dient dazu, diese Anweisungen von der Hardware, d.h. den physischen Komponenten eines Computersystems, zu unterscheiden. Ein Satz solcher Anweisungen, der die Hardware eines Computers anweist, eine Aufgabe auszuführen, wird als Programm oder Softwareprogramm bezeichnet.

Bereits vor vielen Jahrzehnten erkannten Entwickler, dass Software weit mehr ist als nur eine Reihe von Anweisungen. Im Laufe dieses Buchs werden wir uns mit Systemverhalten beschäftigen und erkennen, dass der Hauptzweck von Software darin besteht,

> etwas zu simulieren, das in einer möglichen Realität geschieht.

Diese Vorstellung führt uns zurück zu den Ursprüngen moderner Programmiersprachen wie *Simula*.



Simula

Simula war nicht nur die erste objektorientierte Programmiersprache, sondern auch die erste, die Klassen einführte. Der Name der Sprache deutet klar darauf hin, dass der Zweck der Softwareentwicklung die Schaffung einer Simulation war. Das gilt auch heute noch für die meisten Computerprogramme.

In der Wissenschaft erstellt man Simulationen, um die Vergangenheit zu verstehen und die Zukunft vorherzusagen. Seit Platon bemühen sich die Menschen darum, zu-

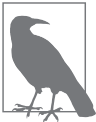
treffende Modelle der Wirklichkeit zu entwickeln. Wir können Software als den Bau eines Simulators definieren, der mit dem Akronym *MAPPER* umschrieben wird:

> **M**odell: **a**bstrakte, **p**artielle und **p**rogrammierbare **E**rklärung der **R**ealität

Dieses Akronym wird uns durch das ganze Buch hindurch begleiten. Schauen wir uns an, was diesen MAPPER ausmacht.

Warum ist es ein Modell?

Ein Modell entsteht, indem man einen bestimmten Aspekt der Realität aus einer speziellen Perspektive, unter Verwendung eines bestimmten Paradigmas, betrachtet. Es verkörpert keine endgültige, unveränderliche Wahrheit, sondern das genaueste Verständnis, das auf der Grundlage des jeweils aktuellen Wissensstands möglich ist. Die Aufgabe eines Softwaremodells, wie die jedes anderen Modells auch, ist die Vorhersage eines Verhaltens in der realen Welt.



Modell

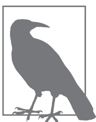
Ein *Modell* erklärt den Gegenstand, den es beschreibt, durch intuitive Konzepte oder Metaphern. Das Endziel eines Modells ist das Verständnis, wie etwas funktioniert. Peter Naur (*https://oreil.ly/6FiD8*) zufolge bedeutet Programmieren, Theorien und Modelle zu entwickeln.

Warum ist es abstrakt?

Das Modell ist mehr als die Summe seiner Teile. Man kann es nicht vollständig verstehen, wenn man nur die einzelnen Komponenten betrachtet. Das Modell basiert auf Vereinbarungen und Verhaltensweisen, die nicht notwendigerweise im Detail beschreiben, wie bestimmte Dinge zu tun sind.

Warum ist es programmierbar?

Ihr Modell sollte innerhalb eines Simulators ausgeführt werden, der die gewünschten Bedingungen nachbildet. Das könnte ein Turing-Modell sein (wie heutige kommerzielle Computer), ein Quantencomputer (Computer der Zukunft) oder jeder andere Simulator, der mit der Entwicklung des Modells Schritt halten kann. Sie können das Modell so programmieren, dass es auf Ihre Eingriffe in bestimmter Weise reagiert, und dann beobachten, wie es sich verändert.



Turing-Modell

Ein Computer, der auf dem *Turing-Modell* basiert, ist eine theoretische Maschine, die jede berechenbare Aufgabe ausführen kann, die durch einen Satz von Anweisungen oder einen Algorithmus beschrieben wird. Die Turing-Maschine gilt als die theoretische Grundlage der modernen Informatik und dient als Modell für den Entwurf und die Analyse von heutigen Computern und Programmiersprachen.

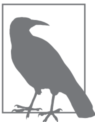
Warum ist es partiell?

Um das Problem zu modellieren, das Sie interessiert, werden Sie nur einen spezifischen Teilaspekt der Realität betrachten. In wissenschaftlichen Modellen ist es üblich, irrelevante Aspekte zu vereinfachen, um ein Problem besser isolieren zu können. Bei der Durchführung wissenschaftlicher Experimente werden nur spezifische, isolierte Variablen kontrolliert verändert, um Hypothesen testen zu können.

In Ihrem Simulator werden Sie nicht die gesamte Realität abbilden können, sondern nur einen wesentlichen Teil davon. Es ist nicht notwendig, den gesamten Beobachtungsgegenstand, also die reale Welt, zu modellieren, sondern nur ein spezifisches Verhalten, das von Interesse ist. Viele Rezepte in diesem Buch adressieren das Problem, dass Modelle unter Overdesign leiden, weil sie zu viele unnötige Details enthalten.

Warum ist es erklärend?

Das Modell sollte hinreichend deklarativ gestaltet sein, um seine Entwicklung beobachten und das Verhalten in der modellierten Realität zu verstehen und dieses vorherzusagen zu können. Es sollte in der Lage sein, zu erklären, was es tut und wie es sich verhält. Viele moderne Algorithmen, die auf maschinellem Lernen beruhen, neigen dazu, als Blackbox zu agieren, und liefern wenig Informationen darüber, wie genau sie ihren Output generieren (und halluzinieren bisweilen). Modelle sollten aber fähig sein, zu erklären, was sie getan haben, auch wenn sie die spezifischen Schritte, die sie dazu unternommen haben, nicht preisgeben.



Erklären

Aristoteles sagte, dass »Erklären bedeutet, die Ursachen zu finden«. Seiner Auffassung zufolge hat jedes Phänomen oder Ereignis eine oder mehrere Ursachen, die es hervorrufen oder bestimmen. Das Ziel der Wissenschaft ist es, die Ursachen von Naturphänomenen zu erkennen und zu verstehen und daraus abzuleiten, wie sich diese in der Zukunft verhalten werden.

Für Aristoteles bestand das »Erklären« darin, all diese Ursachen zu identifizieren und zu verstehen, wie sie miteinander interagieren, um ein bestimmtes Phänomen zu erzeugen. Beim »Vorhersagen« geht es hingegen um die Fähigkeit, dieses Wissen über die Ursachen zu nutzen, um zu prognostizieren, wie sich ein Phänomen in der Zukunft verhalten wird.

Wieso geht es um Realität?

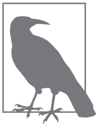
Das Modell muss Bedingungen reproduzieren, die in einer beobachtbaren Umgebung auftreten. Letztlich geht es darum, wie bei jeder Simulation, Verhalten in der realen Welt vorherzusagen. In diesem Buch werden Sie viel über die Realität, die re-

ale Welt und reale Entitäten hören. Die reale Welt wird Ihre ultimative Quelle der Wahrheit sein.

Ableitung der Regeln

Nachdem wir nun verstanden haben, was Software ist, können wir beginnen, gute Modellierungs- und Designpraktiken abzuleiten. Die MAPPER-Prinzipien werden in den Rezepten dieses Buchs regelmäßig erwähnt.

In den folgenden Kapiteln lernen Sie Prinzipien, Heuristiken, Rezepte und Regeln kennen, um hervorragende Softwaremodelle zu erstellen, die auf dem bereits vorgestellten, einfachen Axiom basieren: ***M**odell: **a**bstrakte, **p**artielle und **p**rogrammierbare **E**rklärung der **R**ealität*. Die Arbeitsdefinition von Software für dieses Buch lautet: »Ein Simulator, der den MAPPER-Prinzipien entspricht.«



Axiom

Ein *Axiom* ist eine Aussage oder ein Satz, der ohne Beweis als wahr angenommen wird. Es bildet die Grundlage für ein logisches System, das Schlussfolgerungen und Deduktionen ermöglicht. Durch die Festlegung einer Reihe grundlegender Konzepte und Beziehungen können weitere wahre Aussagen abgeleitet werden.

Das einzig wahre Entwurfsprinzip für Software

Wenn wir das gesamte Paradigma des Softwaredesigns auf einer einzigen Regel aufbauen, können wir es einfach halten und gleichzeitig ausgezeichnete Modelle erstellen. Minimalistisch und axiomatisch zu sein, bedeutet, aus einer einzigen Definition einen Satz von Regeln ableiten zu können:

> Das Verhalten jedes einzelnen Elements ist insofern Teil der Architektur, als es dabei helfen kann, über das System nachzudenken. Das Verhalten von Elementen umfasst ihre Interaktion untereinander und mit der Umwelt. Das ist eindeutig Teil unserer Architekturdefinition und wirkt sich auf die Eigenschaften des Systems aus, z.B. auf seine Laufzeitleistung.
>
> *– Bass et al., Software Architecture in Practice, 4. Auflage*

Eines der am meisten unterschätzten Qualitätsmerkmale von Software ist ihre Vorhersehbarkeit. In der Literatur wird oft betont, dass Software schnell, zuverlässig, robust, beobachtbar, sicher sein sollte. Vorhersehbarkeit zählt selten zu den fünf Top-Prioritäten bei der Entwicklung. Stellen Sie sich als Gedankenexperiment vor, Sie würden objektorientierte Software nach einem einzigen Prinzip entwerfen (wie in Abbildung 2-1 dargestellt): »Jedes Domänenobjekt muss durch ein einzelnes Objekt im berechenbaren Modell dargestellt werden und umgekehrt.« Versuchen Sie dann, alle Entwurfsregeln, Heuristiken und Rezepte aus dieser Prämisse abzuleiten, um Ihre Software gemäß den Rezepten dieses Buchs vorhersehbar zu machen.

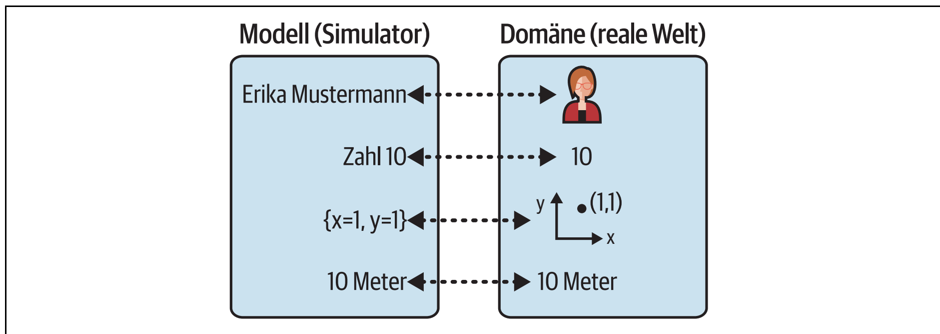


Abbildung 2-1: Es besteht eine 1:1-Beziehung zwischen den Objekten des Modells und den Entitäten der realen Welt

Das Problem

Beim Lesen der Rezepte für Clean Code werden Sie feststellen, dass die meisten in der Industrie verwendeten Sprachimplementierungen darauf verzichten, das gesamte Softwaredesign auf einem einzigen Axiom aufzubauen. Das führt zu erheblichen Problemen. Die meisten modernen Sprachen entstanden, um Implementierungsprobleme zu lösen, die vor drei oder vier Jahrzehnten auftraten, als Ressourcen knapp und Berechnungen vom Programmierer zu optimieren waren. Heute treten solche Probleme nur noch in wenigen Bereichen auf. Die Rezepte in diesem Buch werden Ihnen helfen, diese Probleme zu erkennen, zu verstehen und anzugehen.

Modelle als Retter in der Not

Bei der Entwicklung von Modellen jeder Art ist es entscheidend, die Bedingungen der realen Welt zu simulieren. Sie können jedes Element, für das Sie sich interessieren, in der Simulation verfolgen und auf unterschiedlicher Weise stimulieren, um zu beobachten, ob es sich in gleicher Weise verändert wie in der realen Welt. Meteorologen nutzen mathematische Modelle, um das Wetter vorherzusagen, und viele andere wissenschaftliche Disziplinen stützen sich ebenfalls auf Simulationen. Die Physik sucht nach vereinheitlichenden Modellen, die die Regeln der realen Welt erklären und deren Verhalten vorhersagen. Mit dem Aufkommen des maschinellen Lernens werden auch opake Modelle erstellt, um Verhalten in der realen Welt zu visualisieren.

Die Bedeutung der Bijektion

In der Mathematik ist eine *Bijektion* eine eineindeutige Funktion, die jedes Element der Definitionsmenge auf genau ein Element der Zielmenge abbildet. Umgekehrt ordnet sie jedes Element der Zielmenge genau einem Element der Definitionsmenge zu. Mit anderen Worten: Eine Bijektion ist eine Funktion, die eine *1:1-Beziehung* zwischen den Elementen zweier Mengen herstellt.

Ein *Isomorphismus* ist hingegen eine stärkere Form der Korrespondenz zwischen zwei mathematischen Strukturen, die die Struktur der zugehörigen Objekte bewahrt. Insbesondere ist ein Isomorphismus eine bijektive Funktion, die zusätzlich die Operationen der Strukturen bewahrt.

Eine Bijektion ist also eine eineindeutige Zuordnung zwischen zwei Mengen. Übertragen auf den Bereich der Software bedeutet dies, dass es immer nur ein einziges Objekt geben sollte, das eine reale Entität repräsentiert. Schauen wir uns an, was passiert, wenn man sich nicht an die Grundsätze der Bijektion hält.

Häufige Fälle, bei denen gegen das Bijektionsprinzip verstoßen wird

Es gibt vier häufige Fälle, bei denen das Bijektionsprinzip verletzt wird.

Fall 1

In Ihrem Modell gibt es ein Objekt, das mehr als eine reale Entität repräsentiert. Viele Programmiersprachen modellieren beispielsweise algebraische Maße unter Verwendung nur des skalaren Betrags. In einem solchen Szenario passiert Folgendes, wie in Abbildung 2-2 dargestellt.

- Man kann *10 Meter* und *10 Zoll* (zwei völlig unterschiedliche Entitäten in der realen Welt) durch ein einziges Objekt (*die Zahl 10*) darstellen.
- Man könnte sie im Modell einfach addieren, wobei dann die *Zahl 10* (für *10 Meter*) und die *Zahl 10* (für *10 Zoll*) zusammen die *Zahl 20* ergeben, bei der vollkommen unklar ist, was sie repräsentiert.

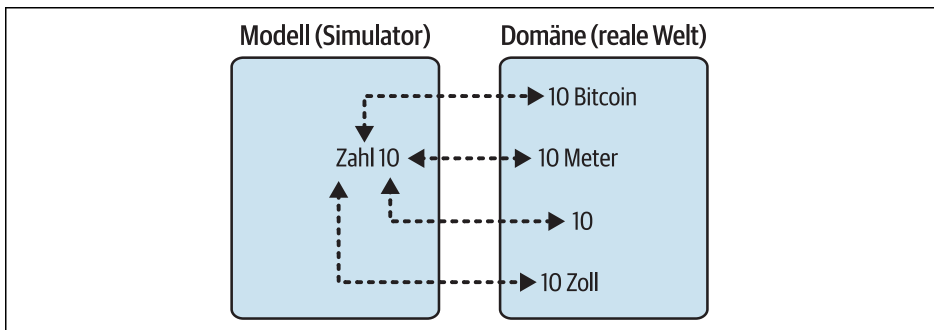


Abbildung 2-2: Die Zahl 10 repräsentiert unterschiedliche Entitäten der realen Welt.

Die Bijektion ist gebrochen: Das führt zu Problemen, die nicht immer rechtzeitig erkannt werden. Da es sich um ein semantisches Problem handelt, wirkt sich der Fehler oft erst nach längerer Zeit aus, wie im berühmten Fall des Mars Climate Orbiters.