

Inhalt

Materialien zum Buch	16
Vorwort	17
Einführung	19
Danksagungen	23
Der Autor	25
Der Übersetzer	25

1 Refactorings für Refactoring 27

1.1 Was ist Refactoring?	28
1.2 Fähigkeiten: Was sollte ich refactor?	29
1.2.1 Ein Beispiel für Code Smell	30
1.2.2 Ein Beispiel für eine Regel	30
1.3 Kultur: Wann sollte ich refactor?	31
1.3.1 Refactoring in einem Legacy-System	33
1.3.2 Wann sollte ich nicht refactor?	33
1.4 Werkzeuge: Wie sollte ich (sicher) refactor?	33
1.5 Werkzeuge für den Anfang	34
1.5.1 Programmiersprache: TypeScript	34
1.5.2 Editor: Visual Studio Code	35
1.5.3 Versionskontrolle: Git	35
1.6 Das durchgehende Beispiel: ein 2D-Rätselspiel	36
1.6.1 Übung macht den Meister: eine zweite Codebasis	38
1.7 Ein Wort zu Software aus der echten Welt	38
1.8 Zusammenfassung	39

2 Ein Blick unter die Haube 41

2.1 Lesbarkeit und Wartbarkeit verbessern	41
2.1.1 Code verbessern	41
2.1.2 Code warten ... ohne zu ändern, was er tut	44
2.2 Geschwindigkeit, Flexibilität und Stabilität gewinnen	45

2.2.1	Komposition statt Vererbung	45
2.2.2	Code ändern durch Hinzufügen statt Umschreiben	46
2.3	Refactoring und die tägliche Arbeit	47
2.3.1	Refactoring als Lernmethode	48
2.4	Die Domäne einer Software definieren	48
2.5	Zusammenfassung	49

TEIL I Das Refactoring eines Computerspiels als Lernbeispiel

3 Lange Funktionen zerschlagen 53

3.1	Unsere erste Regel: Warum fünf Zeilen?	54
3.1.1	Regel: »Fünf Zeilen«	54
3.2	Ein Refactoring, um Funktionen aufzubrechen	57
3.2.1	Refactoring: »Methode extrahieren«	62
3.3	Funktionen teilen, um Abstraktionsebenen zu trennen	66
3.3.1	Regel: »Aufrufen oder Übergeben«	67
3.3.2	Die Regel anwenden	68
3.4	Eigenschaften eines guten Funktionsnamens	69
3.5	Funktionen aufbrechen, die zu viel tun	72
3.5.1	Regel: »if« nur am Anfang	72
3.5.2	Die Regel anwenden	74
3.6	Zusammenfassung	76

4 Typen richtig nutzen 77

4.1	Refactoring einer einfachen »if«-Anweisung	77
4.1.1	Regel: »Benutze niemals »if« mit »else«	78
4.1.2	Die Regel anwenden	80
4.1.3	Refactoring: »Typcodes durch Klassen ersetzen«	82
4.1.4	Code in Klassen schieben	86
4.1.5	Refactoring: »Code in Klassen schieben«	89
4.1.6	Überflüssige Methoden integrieren	95
4.1.7	Refactoring: »Methode integrieren«	96

4.2	Refactoring einer großen »if«-Anweisung	98
4.2.1	Generalität entfernen	103
4.2.2	Refactoring: »Methode spezialisieren«	105
4.2.3	Das einzige erlaubte »switch«	108
4.2.4	Regel: »Benutze niemals ›switch‹«	109
4.2.5	Das »if« löschen	111
4.3	Mit doppeltem Code umgehen	114
4.3.1	Hätten wir nicht statt des Interfaces eine abstrakte Klasse benutzen können?	116
4.3.2	Regel: »Erbe nur von Interfaces«	116
4.3.3	Was soll dieser ganze doppelte Code?	118
4.4	Refactoring von zwei komplexen »if«-Anweisungen	118
4.5	Toten Code entfernen	122
4.5.1	Refactoring: »Versuchsweise löschen und kompilieren«	123
4.6	Zusammenfassung	124
5	Ähnlichen Code zusammenführen	127
5.1	Ähnliche Klassen zusammenführen	128
5.1.1	Refactoring: »Ähnliche Klassen zusammenführen«	138
5.2	Einfache Bedingungen zusammenführen	145
5.2.1	Refactoring: »›if‹s zusammenführen«	148
5.3	Komplexe Bedingungen zusammenführen	150
5.3.1	Rechenregeln für Bedingungen	151
5.3.2	Regel: »Benutze reine Bedingungen«	152
5.3.3	Grundrechenarten für Bedingungen anwenden	155
5.4	Code in verschiedenen Klassen zusammenführen	156
5.4.1	UML-Klassendiagramme zeigen Beziehungen zwischen Klassen	163
5.4.2	Refactoring: »Strategie einführen«	165
5.4.3	Regel: »Keine Interfaces mit nur einer Implementierung«	174
5.4.4	Refactoring: »Interface aus Implementierung extrahieren«	175
5.5	Ähnliche Funktionen zusammenführen	178
5.6	Ähnlichen Code zusammenführen	182
5.7	Zusammenfassung	187

6	Die Daten verteidigen	189
6.1	Kapselung ohne Getter	189
6.1.1	Regel: »Benutze keine Getter und Setter«	189
6.1.2	Die Regel anwenden	192
6.1.3	Refactoring: »Getter und Setter löschen«	194
6.1.4	Den letzten Getter löschen	197
6.2	Einfache Daten kapseln	201
6.2.1	Regel: »Vermeide gemeinsame Affixe«	201
6.2.2	Die Regel anwenden	203
6.2.3	Refactoring: »Daten kapseln«	210
6.3	Komplexe Daten kapseln	214
6.4	Invariante Reihenfolgen entfernen	222
6.4.1	Refactoring: »Reihenfolge erzwingen«	223
6.5	Ein anderes Vorgehen, um Enums zu löschen	226
6.5.1	Enumerationen durch private Konstruktoren	226
6.5.2	Zahlen auf Klassen abbilden	229
6.6	Zusammenfassung	232

TEIL II Das Gelernte in die Praxis übertragen

7	Mit dem Compiler zusammenarbeiten	235
7.1	Den Compiler kennenlernen	236
7.1.1	Schwäche: Das Halteproblem begrenzt unser Wissen zur Kompilierzeit	236
7.1.2	Stärke: Erreichbarkeit garantiert, dass eine Methode »return« erreicht	237
7.1.3	Stärke: Definitive Zuweisung verhindert Zugriff auf uninitialisierte Variablen	238
7.1.4	Stärke: Zugriffskontrolle hilft beim Kapseln von Daten	239
7.1.5	Stärke: Typprüfung beweist Eigenschaften	239
7.1.6	Schwäche: »null« zu dereferenzieren verursacht Abstürze	241
7.1.7	Schwäche: Arithmetische Fehler verursachen Überläufe und Abstürze	241
7.1.8	Schwäche: Zugriff außerhalb der Array-Grenzen verursacht Abstürze	242
7.1.9	Schwäche: Endlosschleifen lassen unsere Anwendung stillstehen	242
7.1.10	Schwäche: Deadlocks und Wettlaufsituationen verursachen unbeabsichtigtes Verhalten	243

7.2	Den Compiler benutzen	245
7.2.1	Den Compiler zum Arbeiten bringen	246
7.2.2	Kämpfe nicht gegen den Compiler	249
7.3	Vertraue dem Compiler	255
7.3.1	Lehre den Compiler Invarianten	255
7.3.2	Beachte seine Warnungen	257
7.4	Vertraue nur dem Compiler	258
7.5	Zusammenfassung	259

8 Finger weg von Kommentaren

8.1	Veraltete Kommentare löschen	263
8.2	Auskommentierten Code löschen	263
8.3	Überflüssige Kommentare löschen	264
8.4	Kommentare in Methodennamen umsetzen	265
8.4.1	Kommentare zur Planung nutzen	265
8.5	Kommentare behalten, die Invarianten dokumentieren	266
8.5.1	Prozessinvarianten	266
8.6	Zusammenfassung	267

9 Lerne, das Löschen zu lieben

9.1	Code löschen: das nächste Abenteuer	270
9.2	Code löschen, um anfallende Komplexität zu reduzieren	271
9.2.1	Technisches Unwissen aus mangelnder Erfahrung	271
9.2.2	Technischer Abraum aus Zeitdruck	272
9.2.3	Technische Schuld aus Sachzwängen	273
9.2.4	Technische Reibung aus Wachstum	273
9.3	Code nach Vertrautheit kategorisieren	274
9.4	Code in einem Legacy-System löschen	275
9.4.1	Code verstehen nach Art der Würgefeige	275
9.4.2	Den Code mit dem Würgefeigenverfahren verbessern	278

9.5	Code aus einem eingefrorenen Projekt löschen	278
9.5.1	Das gewünschte Ergebnis zum Default machen	279
9.5.2	Abraum reduzieren durch Prototypen	279
9.6	Branches aus der Versionskontrolle löschen	280
9.6.1	Abraum reduzieren durch Branch Limits	281
9.7	Codedokumentation löschen	282
9.7.1	Algorithmus, um herauszufinden, wie wir Wissen festhalten	283
9.8	Testcode löschen	284
9.8.1	Optimistische Tests löschen	284
9.8.2	Pessimistische Tests löschen	284
9.8.3	Instabile Tests reparieren oder löschen	284
9.8.4	Refactorings durchführen, um komplexe Testfälle loszuwerden	285
9.8.5	Tests spezialisieren, um sie zu beschleunigen	285
9.9	Konfigurationscode löschen	286
9.9.1	Konfiguration zeitlich kategorisieren	286
9.10	Code löschen, um Bibliotheken loszuwerden	288
9.10.1	Den Einsatz von Bibliotheken einschränken	290
9.11	Code aus funktionierenden Features entfernen	291
9.12	Zusammenfassung	292

10 Keine Angst vor neuem Code 293

10.1	Unsicherheit akzeptieren: In die Gefahr eintreten	294
10.2	Prototypen: gegen die Angst, das Falsche zu entwickeln	294
10.3	Verhältnismäßige Arbeit: gegen die Angst vor Verschwendung und Risiko	296
10.4	Schrittweise Verbesserung: gegen die Angst vor Imperfektion	297
10.5	Wie Copy & Paste unsere Geschwindigkeit beeinflusst	298
10.6	Verändern durch Hinzufügen: geplante Erweiterbarkeit	299
10.7	Verändern durch Hinzufügen erlaubt Abwärtskompatibilität	300
10.8	Verändern durch Hinzufügen mit Featureschaltern	302
10.9	Verändern durch Hinzufügen mit Verzweigung durch Abstraktion	306
10.10	Zusammenfassung	309

11 Folge der Struktur im Code	311
11.1 Strukturen einteilen nach Wirkungsbereich und Herkunft	311
11.2 Drei Arten, wie Code Verhalten spiegelt	313
11.2.1 Verhalten im Kontrollfluss ausdrücken	313
11.2.2 Verhalten in der Struktur der Daten ausdrücken	315
11.2.3 Verhalten in den Daten ausdrücken	319
11.3 Code hinzufügen, um Struktur zu betonen	321
11.4 Beobachten statt vorhersagen – empirische Techniken einsetzen	322
11.5 Sicherheit gewinnen, ohne den Code zu verstehen	323
11.5.1 Sicherheit durch Tests	323
11.5.2 Sicherheit durch Handwerkskunst	323
11.5.3 Sicherheit durch Werkzeuge	323
11.5.4 Sicherheit durch formale Verifikation	324
11.5.5 Sicherheit durch Fehlertoleranz	324
11.6 Ungenutzte Strukturen finden	324
11.6.1 Leerzeilen nutzen: extrahieren und kapseln	325
11.6.2 Doppelten Code zusammenführen	326
11.6.3 Gemeinsame Affixe nutzen – durch Kapselung	330
11.6.4 Den Laufzeittyp bearbeiten – mit dynamischem Dispatch	332
11.7 Zusammenfassung	333
12 Vermeide Optimierung und Generalität	335
12.1 Nach Einfachheit streben	336
12.2 Verallgemeinern – wann und wie	338
12.2.1 Minimale Lösungen bauen, um Generalität zu vermeiden	338
12.2.2 Dinge ähnlicher Stabilität zusammenführen	339
12.2.3 Unnötige Generalität ausmerzen	339
12.3 Optimieren – wann und wie	340
12.3.1 Refactoring vor Optimierung	341
12.3.2 Optimiere nach der Engpasstheorie	343
12.3.3 Optimiere von Metriken geführt	346
12.3.4 Die richtigen Algorithmen und Datenstrukturen wählen	346
12.3.5 Caches benutzen	348
12.3.6 Optimierten Code isolieren	350
12.4 Zusammenfassung	352

13 Schlechter Code soll schlecht aussehen	353
13.1 Auf Prozessprobleme mit schlechtem Code aufmerksam machen	353
13.2 Trennung in sauberen und problematischen Code	355
13.2.1 Die Theorie der zerbrochenen Fenster	356
13.3 Ansätze, schlechten Code zu definieren	356
13.3.1 Die Regeln aus diesem Buch: einfach und konkret	356
13.3.2 Code Smells: komplett und abstrakt	357
13.3.3 Zyklomatische Komplexität: algorithmisch (objektiv)	357
13.3.4 Kognitive Komplexität: algorithmisch (subjektiv)	358
13.4 Regeln für sicheren Vandalismus	359
13.5 Methoden für sicheren Vandalismus	359
13.5.1 Enums benutzen	360
13.5.2 »Int« und »String« als Typcodes benutzen	361
13.5.3 Magische Zahlen in den Code schreiben	362
13.5.4 Kommentare hinzufügen	362
13.5.5 Leerzeilen und -zeichen einfügen	363
13.5.6 Dinge nach Namen gruppieren	364
13.5.7 Namen um Kontext erweitern	365
13.5.8 Lange Methoden schaffen	366
13.5.9 Methoden viele Parameter geben	367
13.5.10 Getter und Setter benutzen	368
13.6 Zusammenfassung	369
14 Zum Abschluss	371
14.1 Ein Rückblick auf die Reise in diesem Buch	371
14.1.1 Einführung: was und warum?	371
14.1.2 Teil I: konkrete Anwendungen	372
14.1.3 Teil II: den Horizont erweitern	372
14.2 Die Philosophie dahinter	372
14.2.1 Die Suche nach immer kleineren Schritten	372
14.2.2 Die Suche nach der zugrunde liegenden Struktur	373
14.2.3 Die Regeln zur Zusammenarbeit einsetzen	374
14.2.4 Das Team über die Person stellen	374
14.2.5 Einfachheit über Vollständigkeit stellen	375
14.2.6 Objekte oder Funktionen höherer Ordnung benutzen	376

14.3	Wie mache ich weiter?	377
14.3.1	Die Mikroarchitektur-Straße	377
14.3.2	Der Makroarchitektur-Weg	377
14.3.3	Die Softwarequalitäts-Route	378
14.4	Zusammenfassung	378

Anhang 381

A	Die Werkzeuge für Teil I installieren	381
A.1	Node.js	381
A.2	TypeScript	381
A.3	Visual Studio Code	381
A.4	Git	382
A.5	Das TypeScript-Projekt einrichten	382
A.6	Das TypeScript-Projekt bauen	382
A.7	Wie du den Level änderst	383
Index		385

Materialien zum Buch

Auf der Webseite zu diesem Buch stehen folgende Materialien für Sie zum Download bereit:

- ▶ **alle Codebeispiele aus dem Buch**

Gehen Sie auf www.rheinwerk-verlag.de/5604. Klicken Sie auf den Reiter MATERIALIEN. Sie sehen die herunterladbaren Dateien samt einer Kurzbeschreibung des Dateiinhalts.

- ▶ Um den Download zu starten, klicken Sie auf den Button HERUNTERLADEN.
- ▶ Alternativ: Um über das GitHub-Repository des Autors auf den Code zuzugreifen, folgen Sie dem Link zum GITHUB-REPOSITORY.

Diese Leseprobe haben Sie beim
 edv-buchversand.de heruntergeladen.
Das Buch können Sie online in unserem
Shop bestellen.

[Hier zum Shop](#)