O'REILLY®

Der US-Bestseller von Kent Beck

Tidy First?

Mini-Refactorings für besseres Software-Design

Kent Beck

Mit einem Vorwort von Larry Constantine

Übersetzung von Thomas Demmig

Einleitung

Was ist Tidy First?

»Ich muss diesen Code ändern, aber er ist so unordentlich. Was sollte ich als Erstes tun?«

»Vielleicht sollte ich den Code aufräumen, bevor ich die Änderung vornehme. Vielleicht. Ein bisschen. Oder vielleicht nicht?«

Das sind Fragen, die Sie sich eventuell selbst stellen, und wenn es einfache Antworten darauf gäbe, hätte ich nicht das Gefühl, ein Buch darüber schreiben zu müssen.

Tidy First? beschreibt:

- wann Sie unordentlichen Code aufräumen sollten, bevor Sie ändern, was dieser Code tut,
- wie Sie unordentlichen Code sicher und effizient aufräumen,
- wann Sie damit aufhören, unordentlichen Code aufzuräumen, und
- warum das Aufräumen funktioniert.

Softwaredesign ist eine Übung in zwischenmenschlichen Beziehungen. In *Tidy First?* beginnen wir mit der sprichwörtlichen Person im Spiegel – mit der Beziehung des Programmierers oder der Programmiererin zu sich selbst. Warum nehmen wir uns keine Zeit, uns um uns selbst zu kümmern? Uns die Arbeit zu erleichtern? Warum stürzen wir uns auf das Aufräumen von Code und vergessen dabei Aufgaben, die unseren Anwenderinnen und Anwendern helfen würden?

Tidy First? ist der nächste Schritt bei meiner Mission, Geeks ein sichereres Gefühl zu geben. Es ist auch der erste Schritt bei der Arbeit mit unordentlichem Code. Softwaredesign ist ein mächtiges Werkzeug, mit dem sich viele Schmerzen verringern lassen – wenn man es richtig einsetzt. Falsch verwendet, wird es nur ein weiteres Mittel der Unterdrückung, das die Effektivität von Softwareentwicklung beeinträchtigt.

Tidy First? ist das erste Buch einer Reihe, die sich auf Softwaredesign fokussiert. Ich möchte, dass Softwaredesign zugänglich und wertgeschätzt wird, daher beginne ich mit der Art von Design, die Sie selbst umsetzen können. In den nachfolgenden Bänden wird Softwaredesign dafür eingesetzt, die Beziehungen zwischen den Program-

mierern und Programmiererinnen in einem Team wiederherzustellen, um dann das ganz große Ding anzugehen: die Beziehung zwischen Business und Technologie. Aber zuerst wollen wir Softwaredesign auf eine Art und Weise verstehen und umsetzen, die uns in der tagtäglichen Arbeit hilft.

Nehmen wir an, Sie hätten eine große Funktion mit vielen Zeilen Code. Bevor Sie sie ändern, schauen Sie sich den Code an, um zu verstehen, was darin passiert. Dabei sehen Sie, wie Sie den Code in kleinere, logisch zusammengehörige Stücke unterteilen können. Extrahieren Sie diese Stücke, räumen Sie auf. Andere Arten von Aufräumarbeiten beinhalten den Einsatz von Guard Clauses sowie das Erklären in Kommentaren und Hilfsfunktionen.

Als Buch setzt *Tidy First?* das um, was es vorschlägt – die »Aufräumereien« werden in kleinen Häppchen vorgestellt, und es wird vorgeschlagen, wann und wo sie sinnvoll sein können. Statt also zu versuchen, gleich alles komplett auf einmal aufzuräumen, können Sie ein paar Techniken ausprobieren, die für Ihr Problem sinnvoll erscheinen. *Tidy First?* beginnt zudem damit, die Theorie hinter dem Softwaredesign zu beschreiben: Kopplung, Kohäsion, abgezinste Zahlungsströme und Optionalität.

Wer dieses Buch lesen sollte

Dieses Buch ist für Programmiererinnen und Programmierer, Lead Developer, programmierende Softwarearchitekten und technische Manager gedacht. Es ist nicht an eine Programmiersprache gebunden; Entwicklerinnen und Entwickler können die Konzepte in diesem Buch lesen und auf ihre eigenen Projekte anwenden. Dieses Buch setzt voraus, dass die Leserinnen und Leser über einen gewisse Programmiererfahrung verfügen.

Was Sie lernen werden

Nachdem Sie das Buch gelesen haben, werden Sie Folgendes verstanden haben:

- Den fundamentalen Unterschied zwischen Änderungen am Verhalten eines Systems und Änderungen an seiner Struktur.
- Die Magie der abwechselnden Investition in Struktur und Verhalten, wenn man als Einzelner oder Einzelne Code verändert.
- Die Grundlagen der Theorie, wie Softwaredesign funktioniert und welche Kräfte dabei wirken.

Und Sie werden in der Lage sein:

- Ihr eigenes Programmiererlebnis zu verbessern, indem Sie manchmal zuerst aufräumen (und manchmal danach).
- Damit zu beginnen, große Änderungen in kleinen, sicheren Schritten vorzunehmen.
- Design als eine menschliche Tätigkeit, mit der unterschiedliche Anreize verbunden sind, zu betrachten.

Struktur des Buchs

Tidy First? ist in eine Einleitung und drei Hauptteile gegliedert:

Einleitung
: Ich beginne mit einer kurzen Beschreibung meiner Motivation, dieses Buch zu schreiben, wie ich dazu kam, für wen es gedacht ist und was Sie erwarten können. Dann steigen wir richtig ein.

Teil I: »Aufräumereien«
: Eine Aufräumerei ist wie ein winzig kleines Mini-Refactoring. Jedes kurze Kapitel ist eine Aufräumarbeit. Sehen Sie Code wie diesen, ändern Sie ihn in Code wie jenen. Dann lassen Sie ihn auf die Produktivumgebung los.

Teil II: Managen
: Als Nächstes kümmern wir uns um das Managen des Aufräumprozesses. Teil der Aufräumphilosophie ist, dass sie nie ein großes Ding sein sollte. Es ist nichts, das berichtet, verfolgt, vorbereitet oder geplant werden sollte. Sie müssen diesen Code ändern, aber das ist schwierig, weil er so unordentlich ist – also räumen Sie zuerst auf. Selbst als Teil des Tagesgeschäfts ist das trotzdem ein Prozess, der durch Nachdenken besser wird.

Teil III: Theorie
: Hier kann ich endlich meine Flügel ausbreiten und mich mit den Themen beschäftigen, die mich begeistern. Was meine ich mit: »Softwaredesign ist eine Übung in menschlichen Beziehungen?« Wer sind diese Menschen? Wie werden ihre Bedürfnisse besser durch besseres Softwaredesign erfüllt? Warum kostet Software so verdammt viel? Was können wir dagegen tun? (Spoiler: Softwaredesign) Kopplung? Kohäsion? Potenzgesetze?

Mein Ziel ist, dass Sie beim Lesen schon nach einem Tag besser designen. Und an jedem folgenden Tag noch etwas besser. Ziemlich schnell wird Softwaredesign beim Schaffen von Wert nicht mehr länger das schwächste Glied in der Kette sein.

Warum »empirisches« Softwaredesign?

Die eindrücklichsten Diskussionen beim Softwaredesign scheinen sich darum zu drehen, *was* zu designen ist:

- Wie groß sollten Services sein?
- Wie groß sollten Repositories sein?
- Events versus explizites Aufrufen von Services.
- Objekte versus Funktionen versus imperativer Code.

Diese Debatten um das *Was* sorgen dafür, dass ein grundlegender Dissens beim Softwaredesign untergeht: *Wann*? Die beiden Extreme sind hier überspitzt dargestellt:

Spekulatives Design
: Wir wissen, was wir als Nächstes tun wollen, also designen wir heute dafür. Es ist günstiger jetzt zu designen. Und wenn die Software erst produktiv ist, ist es zu spät zum Designen, also lass es uns lieber gleich machen.

Reaktives Design
: Es geht nur um Features, also kümmern wir uns jetzt um so wenig Design wie möglich, damit wir wieder an Features arbeiten können. Erst dann, wenn es gar nicht mehr möglich ist, neue Features hinzuzufügen, verbessern wir notgedrungen das Design – aber gerade so viel, dass wir danach mit Features fortfahren können.

Ich bin geneigt, das *Wann* mit »irgendwann dazwischen« zu beantworten. Stellen wir fest, dass sich eine bestimmte Art von Feature nur noch schwierig hinzufügen lässt, designen wir, bis der Druck wieder nachlässt. Dabei beginnen wir mit gerade so viel Design, dass unser Feedback-Zyklus laufen kann:

Features
: Was wollen die Anwender und Anwenderinnen?

Design
: Wie kann man Programmiererinnen und Programmierer am besten darin unterstützen, diese Features auszuliefern?

Die Antwort des empirischen Softwaredesigns auf die *Wann*-Frage ist unterschiedlich: Designen Sie eine Zeit lang, wenn Sie dadurch Vorteile haben. Es erfordert Erfahrung, Verhandlungsgeschick und eigenes Urteilsvermögen. Ist es eine Schwäche, dass Erfahrung und Urteilsvermögen notwendig sind? Sicher, aber das ist unvermeidbar. Spekulatives und reaktives Design erfordern ebenfalls eigene Bewertungen, aber beim Softwaredesign hat man dann weniger Werkzeuge, die man nutzen kann.

Ich finde, das Wort »empirisch« beschreibt diesen Stil gut, weil es den Unterschied deutlich macht, den ich beim Timing bei spekulativem und reaktivem Design mache. »Basiert auf, befasst sich mit oder ist durch Beobachtung und Erfahrung überprüfbar und baut nicht auf Theorie oder reine Logik.« Klingt nicht verkehrt.

Warum habe ich »Tidy First?« geschrieben?

Als Student habe ich einen Kurs zu Softwaredesign belegt, bei dem das Buch *Structured Design* von Ed Yourdon (RIP) und Larry Constantine zum Einsatz kam. Viel habe ich nicht von dem Buch verstanden, vor allem weil mir die darin angesprochenen Probleme noch nicht begegnet waren.

Springen wir 25 Jahre weiter ins Jahr 2005. Ich hatte mittlerweile eine ganze Menge Software designt und dachte, ich hätte das mit dem Designen schon ganz gut verstanden. Stephen Fraser organisierte ein Panel auf der OOPSLA (der großen Konferenz zu objektorientierter Programmierung), um die Veröffentlichung des oben genannten Buchs vor 30 Jahren zu feiern. Ed und Larry waren dabei, aber auch Rebecca Wirfs-Brock, Grady Booch, Steve McConnell und Brian Henderson-Sellers.

Wenn ich nicht mit Schimpf und Schande von der Bühne gejagt werden wollte, musste ich meine Hausaufgaben erledigen. Also schlug ich meine vergilbte Ausgabe von *Structured Design* auf und begann zu lesen. Stunden später schaute ich wieder auf – restlos begeistert. Das waren Newtons Gesetze, nur für das Softwaredesign. Als ich es erst einmal verstanden hatte, war alles völlig klar. Wie konnten wir als Branche diese Klarheit vergessen haben?

Ich erinnere mich daran, dass das Panel gut ablief. Ein Höhepunkt der Konferenz war ein Frühstück mit Ed und Larry – zwei außerordentlich klugen Köpfen, mit sich und der Welt zufrieden. In Abbildung E-1 sehen Sie die Unterschriften, die sie mir damals in meinem Lehrbuch hinterlassen haben.

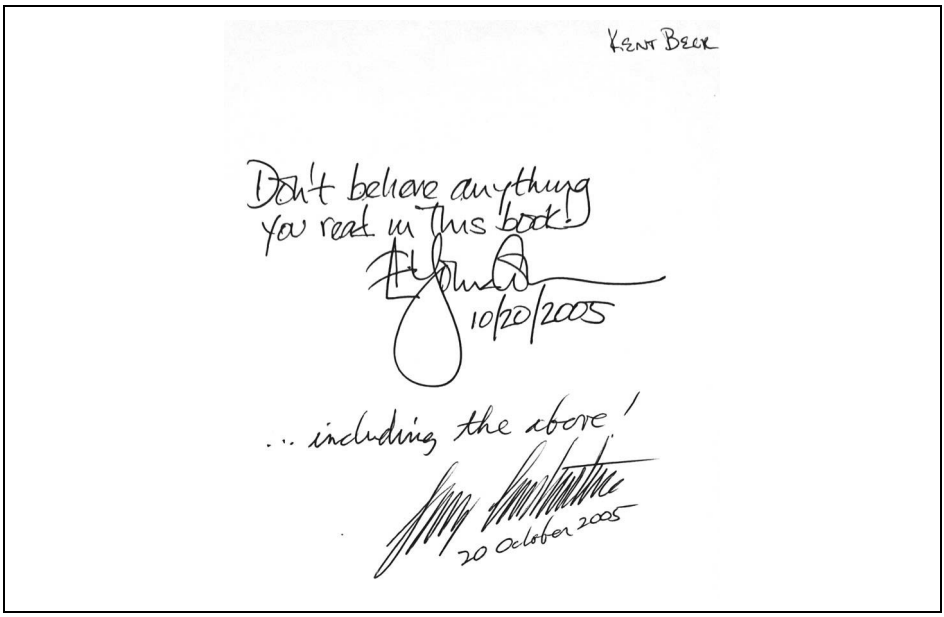

Abbildung E-1: Autogramme von Ed Yourdon (»Glaube nichts, was du in diesem Buch liest!«) und Larry Constantine (»… einschließlich des obigen Satzes!«)

Das Buch war damals nicht mehr ganz aktuell. Beispiele mit Papier- und Magnetband waren nicht länger relevant. Und auch nicht die Diskussion über Assembler und die neuen Hochsprachen. Die Grundlagen trafen aber immer noch den Punkt. Ich versprach, das Material für ein aktuelles Publikum anzupassen.

Es gab in den folgenden Jahren diverse erfolglose Versuche, ein Buch über Softwaredesign zu schreiben (suchen Sie mal nach »Kent Beck Responsive Design«, wenn Sie sehen wollen, an was ich gearbeitet habe). Erst im Jahr 2019 hatte ich plötzlich zwei Wochen nicht verplante Freizeit. Ich entschied mich dazu, mal zu schauen, wie viel ich von diesem Buch in den zwei Wochen schreiben könnte.

Zehntausend Wörter später hatte ich eine wichtige Lektion gelernt – ich würde die gesamte Bandbreite des Softwaredesigns nicht in einem Buch abdecken können. Ein Szenario, das ich in meinen Notizen immer wieder skizziert hatte, war dieser Mo-

ment des Designs im kleinen Maßstab: Ich hatte unordentlichen Code – ändere ich ihn zuerst, oder räume ich ihn erst auf?

Meine Erfahrungen beim Schreiben von Büchern waren schon immer so. Nimm ein Thema, das zu klein für ein Buch zu sein scheint. Schreibe. Stelle fest, dass das Thema viel zu groß für ein Buch ist. Nimm einen winzigen, viel zu kleinen Ausschnitt davon. Schreibe. Erkenne, dass der Ausschnitt zu groß ist. Wiederhole.

Und jetzt halten Sie (virtuell oder real) die ersten Früchte dieses nahezu 20 Jahre alten Versprechens in Ihren Händen. Ich fand, dass ich durch das Behandeln der regelmäßig wiederkehrenden Frage »Soll ich zuerst aufräumen?« viele der Themen ansprechen konnte, die mir an meinem Designer-Herzen liegen. Ich freue mich auf Ihr Feedback und darauf, mein Verständnis all dessen weiter zu vertiefen, was das Softwaredesign so schön und wertvoll macht.

Danksagungen

Der »Autor« eines Buchs ist eine vereinfachende Fiktion. Ich habe die Wörter getippt, aber Sie würden diese Wörter nicht lesen können, wenn nicht ganz viele Menschen ihre Hände im Spiel gehabt hätten. Hier möchte ich einige davon aufführen.

Vielen Dank für das frühe technische Feedback an Anna Goodman, Matan Zruya, Jeff Carbonella, David Haley, Kelly Sutton und den Rest meiner Studentinnen und Studenten bei Gusto. Ebenso für das technische Feedback zum Manuskript an Maude Lemaire, Rebecca Wirfs-Brock, Vlad Khononov und Oleksii Toruniv. Vielen Dank an meine zahlenden Abonnenten von *https://tidyfirst.substack.com* dafür, dass sie mir das Schreiben ermöglicht haben, und für ihr Feedback zu den Kapiteln, deren Entwürfe ich dort veröffentlichte.

Dank geht an das ausgezeichnete Produktionsteam bei O'Reilly, die das fortschreitende Projekt so angenehm wie möglich gestaltet haben: Melissa Duffield, Michele Cronin, Louise Corrigan. Vielen Dank auch an Tim O'Reilly für die Chance, so ein schlankes Buch umzusetzen.

Vielen Dank an Keith Adams und Pamela Vagata für die technischen Gespräche, die Unterstützung und die gelegentlichen Cocktails. Dank an Susan für die richtige Mischung aus Unterstützen und Anschubsen. Vielen Dank an meine Kinder Beth, Lincoln, Lindsey, Forrest und Joëlle.

Dank an meine Softwaredesignmentoren und -kollegen: Ward Cunningham, Martin Fowler, Ron Jeffries, Erich Gamma, David Saff und Massimo Arnoldi.

Ein Dank geht schließlich auch an Ed Yourdon (in seligem Andenken) und Larry Constantine, die das Ganze schon vor so langer Zeit erkannt haben.

KAPITEL 1

Guard Clauses

Sie sehen Code wie den folgenden:

```
if (Bedingung)
    ... Code ...
```

Oder noch besser:

```
if (Bedingung)
    if (not andere Bedingung)
        ... Code ...
```

Beim Lesen verliert man sich leicht in verschachtelten Bedingungen. Räumen Sie das Ganze wie folgt auf:

```
if (not Bedingung) return
if (andere Bedingung) return
... Code ...
```

Das lässt sich leichter lesen. Der Code sagt so: »Bevor wir in die Details des Codes einsteigen, gibt es ein paar Vorbedingungen, die wir berücksichtigen müssen.«

(Aber was ist mit MuLTipLen ReTuRns? Die »Regel« bezüglich eines einzelnen Returns pro Routine stammt aus den Tagen von FORTRAN, als eine einzelne Routine mehrere Einstiegs- *und* Ausstiegspunkte haben konnte. Solcher Code ließ sich so gut wie gar nicht debuggen. Niemand konnte sagen, welche Anweisungen ausgeführt wurden. Code mit Guard Clauses lässt sich leichter analysieren, weil die Vorbedingungen explizit sind.)

Treiben Sie es mit den Guard Clauses aber nicht zu weit. Eine Routine mit sieben oder acht Guard Clauses (die mir durchaus in der freien Wildbahn schon begegnet ist) lässt sich *nicht* leichter lesen. Stattdessen muss man sich dann dringender darum kümmern, die Komplexität aufzuteilen.

Räumen Sie mit einer Guard Clause nur auf, wenn die Anforderung genau erfüllt wird:

```
if (Bedingung)
    ... der gesamte Rest des Codes in der Routine ...
```

Ich sehe Code, den ich aufräumen will, aber nicht kann:

```
if (Bedingung)
    ... Code ...
... anderer Code ...
```

Vielleicht lassen sich die ersten beiden Zeilen in eine Hilfsmethode auslagern, die dann durch eine Guard Clause aufgeräumt wird, aber gehen Sie wirklich *immer* nur in kleinen Schritten vor.

Hier ein Beispiel: *https://github.com/Bogdanp/dramatiq/pull/470*

KAPITEL 2

Toter Code

Löschen Sie ihn. Das ist alles. Wenn der Code nicht ausgeführt wird, löschen Sie ihn einfach.

Das Löschen von totem Code kann sich erstaunlich seltsam anfühlen. Schließlich hat jemand Zeit und Aufwand in sein Schreiben investiert. Die Organisation hat dafür bezahlt. Er ist jetzt da. Man muss ihn nur aufrufen, damit er Werte schafft. Wenn wir ihn wieder benötigen, wäre es doch töricht, ihn zu löschen.

Ich überlasse es Ihnen, all die kognitiven Vorurteile herauszufinden, die ich gerade demonstriert habe.

Manchmal ist es einfach, toten Code zu identifizieren. Manchmal – insbesondere bei intensiver Nutzung von Reflection – ist es nicht so einfach. Haben Sie den Verdacht, dass Code nicht genutzt wird, treffen Sie vorbereitende Aufräummaßnahmen, indem Sie seine Verwendung loggen. Bringen Sie die vorbereitenden Aufräumereien in die Produktivumgebung und warten Sie so lange, bis Sie sicher genug sind.

Sie fragen sich vielleicht: »Und wenn wir den Code später brauchen?« Nun, dafür ist die Versionsverwaltung da. Wir löschen ja nicht wirklich etwas. Wir müssen es uns jetzt nur nicht mehr anschauen. Wenn wir (und das sind wirklich viele Wenns) 1. viel Code haben, der 2. jetzt nicht genutzt wird, den wir aber 3. in Zukunft verwenden wollen – und zwar 4. genau so, wie er ursprünglich geschrieben wurde –, und wenn er dann 5. immer noch funktioniert, können wir ihn ja zurückholen. Oder ihn neu schreiben, dieses Mal besser. Aber wenn es hart auf hart kommt, bekommen wir ihn immer wieder zurück.

Löschen Sie wie immer mit jeder Aufräumerei nur ein bisschen Code. Stellt sich dabei heraus, dass Sie falsch lagen, lässt sich die Änderung recht einfach wieder rückgängig machen (siehe Kapitel 28). »Ein bisschen« ist ein subjektiver Wert, keine harte Anzahl von Codezeilen. Es kann eine Klausel in einer Bedingung sein (weil Sie zum Beispiel sehen, dass sie sich auf true eindampfen lässt), eine Routine, eine Datei, ein Verzeichnis.

KAPITEL 5

Lesereihenfolge

Nehmen wir an, Sie lesen eine Datei (wir können ein andermal darüber diskutieren, ob Quellcode in Dateien gehört). Sie lesen die gesamte Datei, gelangen an deren Ende, und da ist es – das Detail, das Ihnen dabei geholfen hätte, den ganzen Rest der Datei zu verstehen.

Ordnen Sie den Code in der Datei in der Reihenfolge an, in der eine Leserin oder ein Leser es beim Lesen bevorzugen würde (und denken Sie daran – auf jeden Schreibenden kommen viele Leser).

Sie sind ein Leser oder eine Leserin. Sie haben es gerade gelesen. Also wissen Sie es.

Widerstehen Sie der Versuchung, gleichzeitig irgendwelche anderen Aufräumereien anzuwenden. Vermutlich werden Sie beim Lesen über andere Details gestolpert sein, die ein Verstehen und Ändern schwerer machen, als sie sein sollten. Gehen Sie solche Details später an. Alternativ kümmern Sie sich jetzt um diese Details und verschieben das Anpassen der Lesereihenfolge auf später. Aber vermischen Sie nicht beides.

Manche Sprachen reagieren beim Deklarieren von Elementen empfindlich auf deren Reihenfolge. Ein Vertauschen der Reihenfolge beim Deklarieren der Funktionen A und B kann dann andere Ergebnisse bei der Ausführung hervorbringen. Seien Sie bei solchen Sprachen vorsichtig. Ordnen Sie vielleicht nicht die ganze Datei neu an, sondern nur die für Lesende relevanten Abschnitte.

Die eine perfekte Reihenfolge gibt es nicht. Manchmal wollen Sie erst die Primitive verstehen und dann sehen, wie sie sich zusammensetzen. Manchmal wollen Sie zuerst die API verstehen und dann die Details der Implementierung. Sie sind der Leser oder die Leserin – nutzen Sie also Ihre Einschätzung und Ihre (aktuelle) Erfahrung. Welche Reihenfolge hätten Sie gern gesehen? Geben Sie diese als Geschenk an die nächste Person, die den Code lesen wird.

Inhalt

O'REILLY®

Tidy First?

Unordentlicher Code ist ein Ärgernis und schwer zu lesen. In diesem praktischen Leitfaden demonstriert Kent Beck, Schöpfer von Extreme Programming, wann und wie Sie kleinere Aufräumarbeiten durchführen können, um Ihren Code zu optimieren und dabei die Gesamtstruktur Ihres Systems immer besser zu verstehen.

Anstatt sich mit zu viel Aufräumen auf einmal zu überfordern, zeigt Kent Beck, wie Sie pragmatisch vorgehen. Sie lernen etwa, wie Sie eine Funktion mit vielen Codezeilen logisch in kleinere Stücke aufteilen. Und nebenbei verstehen Sie wichtige Aspekte der Theorie wie Kopplung, Kohäsion, abgezinste Zahlungsströme und Optionalität.

Dieses Buch unterstützt Sie dabei:

- **die grundlegende Theorie zu verstehen, wie Software-Design funktioniert und welche Kräfte darauf einwirken**
- **unterschiedliche Auswirkungen bei Änderungen am Verhalten eines Systems und bei Änderungen an seiner Struktur einzuschätzen**
- **Ihr Programmiererlebnis zu verbessern, indem Sie manchmal zuerst aufräumen und manchmal später**
- **zu lernen, wie Sie große Veränderungen in kleinen, sicheren Schritten vornehmen**
- **Softwareentwicklung als ein Pflegen menschlicher Beziehungen zu verstehen**

Kent Beck ist Begründer von Extreme Programming, Pionier der Software Patterns, Mitautor von JUnit, Wiederentdecker der testgetriebenen Entwicklung und Beobachter von 3X: Explore/Expand/Extract. Kent ist – in alphabetischer Reihenfolge – der erste Unterzeichner des Agilen Manifests.

»Code aufzuräumen ist essenziell! Aber wann und wie viel? *Tidy First?* gibt Antworten … aus unterschiedlichsten Perspektiven, nicht nur für Entwickler!«

— Marco Emrich
Software Architect/Primary Consultant bei codecentric AG

»Wann lohnt es sich, Code aufzuräumen? Kent Beck bietet wertvolle Tipps: von ›Tidyings‹ zur Code-Verbesserung bis hin zur Kosten-Nutzen-Analyse.«

— Patrick Baumgartner
Technical Agile Coach/ Software Crafter bei 42talents

»Einfache Ideen, und doch werden Sie sich fragen, warum Sie nicht früher daran gedacht haben. Für alle, denen sauberer und lesbarer Code am Herzen liegt.«

— Gergely Orosz
The Pragmatic Engineer

9 783960 092445

www.dpunkt.de

Euro 26,90 (D)
ISBN 978-3-96009-244-5