Legacy-Software

Sanieren statt Neumachen!

Dieser Artikel zeigt Optionen zum Weiterbetrieb von Altsystemen als Alternative zur Neuentwicklung auf. Die Systeme werden dabei unter dem Gesichtspunkt der Rentabilität betrachtet. Unternehmen geht es nicht darum, neueste Technologien einzusetzen, sondern schnell Funktionalität mit möglichst geringem Einsatz nutzen zu können. Der Artikel erklärt, wie Altsoftware mit fokussierten Sanierungsmaßnahmen kosten- und risikobewusst fit für die Zukunft gemacht werden kann.

Die meisten großen Unternehmen in Deutschland betreiben für das Unternehmen existenziell wichtige Software, die in Teilen 10 Jahre und älter ist [Fri00]. Dass Legacy-Software nicht zwingend alt sein muss, erklären die Definitionen namhafter Softwareentwickler. Für Michael Feathers ist Legacy-Software schlicht Software ohne automatisierte Tests [Fea04]. Er begründet seine Definition damit, dass Software ohne automatisierte Tests nur schwer änderbar ist, da Regressionen nicht automatisch erkannt werden. Die Pragmatischen Programmierer Andrew Hunt und Dave Thomas gehen noch einen Schritt weiter und bezeichnen Code als Legacy-Code ab dem Zeitpunkt, an dem er geschrieben wurde [Hun99].

Legacy überall

Legacy-Systeme stehen auf der Attraktivitätsskala weit unten. Für Unternehmen sind Altsysteme die Systeme, in denen die Fehler auftreten. Sie sind starr und gleichzeitig fragil. Auch für Entwickler sind Legacy-Projekte wenig attraktiv und setzen häufig veraltete Technologien ein, die sich im Lebenslauf schlecht machen. Hinzukommt, dass die Arbeit mit Legacy-Software sehr viel komplexer und (über-) fordernder ist, als ein Projekt auf der grünen Wiese.

Altsysteme sind aber auch die Systeme, die das Unternehmen am Laufen halten, das Geld verdienen und nicht einfach abgeschaltet oder schnell ausgetauscht werden können.

Geerbte Software

Frei übersetzt steht Legacy für „Erbe" oder „Nachlass". Mit Erben wird in der Regel etwas Positives assoziiert: Menschen erben Häuser oder Geldvermögen. Auf der anderen Seite kann Erben auch etwas Negatives bedeuten, wenn man zum Beispiel ein insolventes Unternehmen oder Schulden erbt. Beiden Arten des Erbens ist gemeinsam, dass man etwas bekommt, in dessen Entstehung man nicht involviert war, wofür man aber ab jetzt verantwortlich ist.

Ähnlich dem Erben analoger Güter verhält es sich mit dem Erben von Softwaresystemen. Es gibt wohl kaum einen Softwareentwickler, der im Laufe seiner Karriere nicht ein existierendes System übernommen hat, für das er ab dem Zeitpunkt verantwortlich war.

Softwareschulden wurden bereits 1992 von Ward Cunningham mit dem Begriff *Technical Debt* belegt [Cun92]. Technical Debt steht als Metapher für den zu erbringenden Mehraufwand, der für die Implementierung neuer Features in existierender Software notwendig ist. So erfordert die Erweiterung einer schlecht entworfenen und nicht kontinuierlich refaktorisierten Software einen Aufwand von $n+x$, wobei n für den Aufwand des Features und x für den Aufwand der erforderlichen Aufräumarbeiten und das Beheben von Regressionen steht. Je schlechter der Code, desto größer wird x (**siehe Abbildung 1**). Im schlechtesten Fall wird x so groß, dass die insgesamt zur Verfügung stehende Entwicklungszeit ins Aufräumen fließt.

Technical Debt lässt sich nicht vermeiden, aber minimieren, indem kontinuierlich refaktorisiert wird. Kontinuierliches Refactoring setzt das Vorhandensein automatisierter Tests voraus, die sicherstellen, dass der Code nach dem Aufräumen genauso funktioniert wie davor.

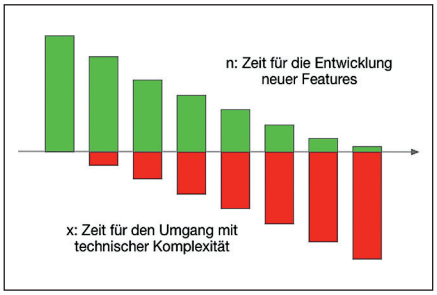

Abb. 1: Technical Debt

Dann doch lieber Neubauen

Automatisierte Tests sind im Legacy-Umfeld leider eher Ausnahme denn die Regel. Folglich ist die Software weder änderbar noch erweiterbar. Diese scheinbar aussichtslose Situation führt in Unternehmen bei der Frage nach der Zukunft ihrer Altsysteme zur Standardantwort „Neuentwicklung". Hinter dieser Tendenz verbirgt sich die grundsätzliche Annahme, dass „neu" gleich „besser" ist. Es gibt psychologische Effekte, die Einfluss darauf haben, ob wir uns für etwas Neues entscheiden.

Einen solchen Effekt beschreibt die *Hauptsache-Neu-Neigung* (Mere Newness Bias) [Jie15]. Es handelt sich dabei um die Tendenz, sich für etwas Neues zu entscheiden, ausschließlich aufgrund des Erscheinungsdatums. Wir schreiben neuen Dingen tendenziell eine höhere Leistungsfähigkeit zu und sind bereit, dafür einen höheren Preis zu zahlen. Das hat ja auch seinen guten Grund. Im technischen Bereich sind wir es gewohnt, dass neue Dinge mehr Funktionalität haben, immer leichter und kleiner werden und einfacher zu bedienen sind. Aber kann ich mit der neuen Programmiersprache meine Probleme wirklich einfacher und schneller lösen?

Einen weiteren Einfluss hat der *Mitläufer-Effekt* (Bandwagon Effect). Dieser Effekt beschreibt eine steigende Rate bei der Übernahme von Überzeugungen, Ideen und Trends, je mehr sie bereits von anderen übernommen wurden. Gerade in der IT kann man den Effekt gut beobachten. Je häufiger ein Thema in Fachzeitschriften und Konferenzen besprochen wird, umso stärker scheint der Druck zu wachsen, es auch im eigenen Unternehmen einsetzen zu müssen.

Eng mit dem Einfluss des Mitläufer-Effekts ist ein dritter Effekt verbunden. Das *Sonnenschein-Szenario* (Feature Positive Effect) hat das Merkmal, dass ein Produkt nur mit den positiven Eigenschaften

beschrieben wird. Die genauso wichtigen negativen Eigenschaften hingegen fehlen schlichtweg. Gerade bei der Vorstellung von neuen Produkten spielt dieser Effekt eine Rolle. Eine NoSQL-Datenbank wird mit all ihren überzeugenden Vorteilen angepriesen, aber hat jemand daran gedacht, dass alle bisher genutzten SQL-Werkzeuge nicht mehr verwendet werden können?
Auch Entwickler sind von der Tendenz zum Neumachen nicht ausgenommen. Eine einleuchtende Erklärung dafür liefert uns Joel Spolsky: "It's harder to read code than to write it" [Spo00]. Entwickler gehen häufig davon aus, dass alter Code schlecht ist und sie diesen sehr viel besser schreiben können.

Über den Wert von Legacy-Software

Neuentwicklung scheint auf den ersten Blick attraktiv, übersieht jedoch den Wert existierender Software. Altsoftware verfügt über jahrzehntelange Produktionserfahrung, ist ausgiebig getestet und hat unzählige Bugfixes erfahren. Der mit viel Arbeit und Geld entstandene Quelltext ist codiertes Wissen, das im Falle einer Neuentwicklung einfach weggeworfen wird.
Vor einer Entscheidung zum Neubau sollte immer die Quantifizierung des Werts des existierenden Systems stehen. Es ist zu beachten, dass für den Zeitraum der Neuentwicklung ein zweites Team bereitstehen muss, da das Altsystem mindestens solange betrieben werden muss, bis das neue System produktiv geht. Während dieser Zeit kommen neue oder sich ändernde Anforderungen dazu, die in beiden Systemen implementiert werden müssen. Das Alt-Team ist unmotiviert, weil es mit veralteten Technologien an Systemen auf dem Abstellgleis arbeiten muss. Das neue Team ist weniger erfahren mit der Materie und steht unter hohem Erwartungsdruck. Das neue Team ist auf das fachliche Wissen angewiesen, das in dem Alt-System steckt. Teammitglieder des Alt-Teams erfahren so eine Doppelbelastung, da sie sowohl das bestehende System betreuen und zugleich Wissen für das neue System bereitstellen müssen.

Warum beim nächsten Mal nichts besser wird

Trotz der genannten Gründe entscheidet sich ein Großteil der Unternehmen, ihre existierende Legacy-Software neu zu entwickeln. Nicht selten, um nach vier Jahren Entwicklungszeit festzustellen, dass die neu entwickelte Software inzwischen veraltet, aber immer noch nicht produktiv ist.
Der Grund für das Scheitern von Neuentwicklungen ist, dass Softwareentwicklung der Klasse *komplexer* Probleme angehört. Während *komplizierte* Probleme mit ausreichendem Wissen beherrschbar sind, zeichnen sich komplexe Probleme durch ein hohes Maß an Überraschungen aus, die weder ein exaktes Planen noch die Vorhersage eines konkreten Ergebnisses zulassen.
Ein klassisches Beispiel für ein komplexes Problem ist Fußball, ein Spiel mit einem hohen Maß an Überraschungen: Wie ist die Strategie des gegnerischen Teams? Was macht der Schiedsrichter? Fußball ist selbst mit ausreichendem Wissen nicht deterministisch beherrschbar und vorhersagbar.
Ganz ähnlich verhält es sich mit der Softwareentwicklung. Diese besteht im Kern aus den drei Dimensionen *Anforderungen*, *Technologie* und *Menschen*. Anforderungen sind vage, unzureichend verstanden und veränderlich. Technologie ist oft neu, fehleranfällig und nur schwer beherrschbar. Dass Menschen und deren Zusammenwirken enorm komplex sind, weiß wohl jeder. Alle drei Dimensionen sind für sich genommen komplex. Kombiniert man komplexe Elemente zu einem Ganzen, explodiert die Komplexität.
Der Wert existierender Software kombiniert mit der Komplexität und Unvorhersagbarkeit von Softwareentwicklung lässt die Entscheidung für eine Neuentwicklung absurd erscheinen. Im besten Fall liefert eine Neuentwicklung ein dem Altsystem ähnliches System, mit deutlich weniger Produktionserfahrung und mit neuen Problemen, die wir heute noch nicht vorhersehen können.

Softwaresanierung als Alternative zum Neubau

Softwaresanierung ist eine Alternative zur Neuentwicklung und besteht im Kern aus dem iterativen Abarbeiten der wichtigsten Baustellen des Altsystems. Am Anfang jeder Sanierung steht die *Analyse*, in der die zentralen Pain Points des Altsystems gemeinsam mit allen relevanten Stakeholdern erarbeitet werden. Im Großteil der von uns analysierten Systeme sind dies häufig:

- Regressionsanfälligkeit aufgrund fehlender Tests,
- fehlendes automatisiertes Deployment und
- fehlende Feature-Fähigkeit.

Parallel zur Pain Point-Analyse wird der Wert des Altsystems fürs Unternehmen bestimmt. Systeme von großem Wert stehen im Zentrum des Unternehmens. Ohne sie würde das Unternehmen nicht funktionieren. Beispiele für diese Primärsysteme wären der Webshop von Amazon, die Lotto-Webanwendung einer Onlinelotterie oder die Smartphone-App von Moia.
Andere Systeme laufen eher nebenher und haben nur geringe Bedeutung für das Kerngeschäft des Unternehmens. Ein Beispiel für ein solches Sekundärsystem ist das von uns im Auftrag eines Automobilherstellers betreute REACH-System. Hierbei handelt es sich um eine Software zur Erfüllung einer EU-Richtlinie, welche von Kunden selten genutzt wird, aber laufen muss.
Basierend auf den Ergebnissen der Pain Point-Analyse und der Wertbestimmung wird das System in eine der vier Kategorien eingeordnet:

- *Wartung:* Das System ist wichtig fürs Unternehmen und muss laufen. Das System generiert wenig Einnahmen und wird mittelfristig nicht weiterentwickelt.
- *Sanierung:* Das System ist wichtig fürs Unternehmen und hat unmittelbaren Einfluss auf dessen Kerngeschäft. Das System wir kontinuierlich geändert und erweitert. Feature-Neuentwicklung ist zwingend erforderlich.
- *Recycling:* Das System ist nicht mehr wichtig, muss aber betrieben werden. Das Unternehmen verdient längst mit einem anderen System Geld, muss das Altsystem aber ohne essenzielle Einnahmen weiter betreiben, weil es zum Beispiel bestehende Verträge mit Kunden gibt.
- *Neuentwicklung:* Das System ist wichtig, steht aber vor dem Aus, weil zum Beispiel die Hardware nicht mehr existiert und sich auch nicht virtualisieren lässt. Andere Gründe können sein, dass für die verwendete Programmiersprache keine Entwickler gefunden werden können.

Von der Analyse zum Sanierungs-Backlog

Ausgehend von der Analyse und Bewertung des Altsystems wird ein Sanierungs-Backlog erstellt. Das Backlog ist priorisiert und enthält Aufgaben, die messbar und risikobewusst innerhalb eines begrenzten Zeitraums (< drei Monate) umgesetzt werden können.
Sanierungs-Backlogs sind Return-on-Investment-zentriert. Es geht darum, die Maßnahmen herauszufinden, die den größten Kosten-Nutzen-Effekt für das Unter-

nehmen haben. Technologische Aspekte, wie die Wahl der Programmiersprache, spielen dabei eine untergeordnete Rolle. Sind zum Beispiel neben dem fehlenden Deployment eine hohe Regressionsrate das Kernproblem der Software, dann helfen auch einzelne Refactorings nicht weiter. Vielmehr geht es darum, einen Weg zu finden, die Altsoftware testbar zu machen.

Je nach Systembewertung fällt das Sanierungs-Backlog unterschiedlich aus. Für Systeme in der Kategorie Wartung geht es primär darum, ein Störungsmanagement in angemessener Reaktionszeit sicherzustellen. In der Regel wird dies durch das Herstellen eines schnellen Deployments mit geringer Downtime erreicht. Selbst Regressionssicherheit wird weniger wichtig, da ein schnelles und nachvollziehbares Deployment genauso schnell wieder zurückgerollt werden kann.

Sanierungs-Backlog zur Umsetzung

Analyse und Bewertung liefern einen neuen Blick auf das Altsystem und zeigen Optionen auf. Viele Unternehmen sehen nach Abschluss der Analysephase von ihrer ursprünglichen Idee einer Neuentwicklung ab und ziehen eine Sanierung in Betracht. Startet die Sanierung, ist das Herstellen automatisiertet Testbarkeit fast immer die wichtigste und erste Aufgabe.

Altsysteme lassen sich so gut wie nie von innen heraus testen. Erstens gibt der vorhandene Quellcode dies nicht her und zweitens gewinnt man wenig mit dem Testen einzelner Klassen oder Funktionen. Stattdessen gilt, Tests soweit außen wie möglich zu schreiben. Je nach Technologie gibt es dafür verschiedene Ansätze. In API-Projekten ist das einfach, da diese sich mithilfe von Standard-HTTP-Clients automatisiert testen lassen. Auch für Web-Anwendungen gibt es inzwischen eine Vielzahl von Tools zur Testautomatisierung. Aber wie sieht es zum Beispiel mit alten Delphi-Anwendungen aus, deren einziger Zugang eine Windows-GUI ist? Bei dieser Art von Systemen kann Robotic Process Automation [RPA] ein vielversprechender Ansatz sein.

Langlaufende Änderungen

Eine Softwaresanierung umfasst Maßnahmen, die sich in der Regel nicht innerhalb eines 2-wöchigen Sprints umsetzen lassen. Statt des kontinuierlichen Anwendens überschaubarer Refactorings besteht eine Sanierung oftmals aus dem Ersetzen kompletter Systemteile. Zwei bekannte Muster zur Durchführung größerer Sanierungsarbeiten sind Strangler Application und Branch by Abstraction.

Strangler Application steht bildlich für das Erwürgen der Altanwendung (**siehe Abbildung 2**), indem neue beziehungsweise zu ersetzende Funktionalität um das Altsystem herum gebaut wird [Fow04]. Neu entwickelte Systemteile erhalten Zugriff auf Daten und Ereignisse des Altsystems, indem zum Beispiel Ereignisse abgehört werden oder die Altanwendung um Datenzugriffs-APIs erweitert wird. Eine von uns erfolgreich angewendete Ablösestrategie ist der Einsatz von Datenbank-Triggern. Bei diesem Ansatz werden alle Datenbank-Updates des Altsystems zeitgleich auf einer neuen Datenbank ausgeführt, die als Basis für die parallele Neuentwicklung einzelner Komponenten genutzt wird.

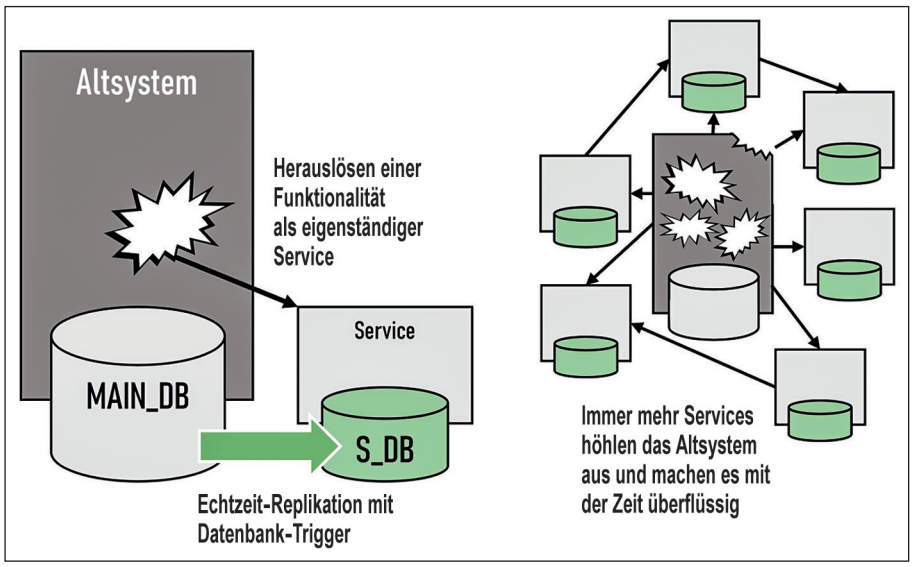

Abb. 2: Ablösen eines Altsystems durch „Erwürgen“

Branch by Abstraction ist ein Large Scale Refactoring, das größere Änderungen ermöglicht, die kontinuierlich released werden können. Im Kern steht die Idee, eine zu ersetzende Komponente durch ein Interface zu abstrahieren. Über mehrere Sprints hinweg kann das neue Interface parallel von einer neuen Komponente implementiert werden. Clients werden sukzessive auf die Neuimplementierung umgestellt, sobald Teile davon fertig sind. Eine anschauliche Erklärung des Musters findet sich unter [Fow14].

Und was ist mit den Entwicklern?

Sanierungsprojekte sind nicht nur technisch, sondern auch menschlich herausfordernd. Softwareentwickler lieben das Programmieren, die Anwendung neuer Technologien und das Schreiben von Grüne-Wiese-Code. Kaum eine dieser drei genannten Tätigkeiten spielt eine tragende Rolle in Sanierungsprojekten. Stattdessen ist es heute die Regel, dass Entwickler das Unternehmen eher verlassen, wenn sie viel mit Legacy-Software zu tun haben.

Der entscheidende, sich beim Programmieren fast automatisch einstellende Zustand ist Flow. Der Begriff *Flow* bezeichnet das vollständige Aufgehen in der aktuellen Tätigkeit, einen Zustand, bei dem man Zeit und Raum, vergisst. Codezeile für Codezeile erwächst, Puzzleteil fügt sich an Puzzleteil und am Ende des Tages steht ein präsentierbares Ergebnis.

Arbeiten mit Legacy-Code stellt sich oft anders da. Es finden sich haufenweise Code-Stellen, von denen man nicht weiß, was sie tun. Niemand ist da, den man fragen könnte, von Dokumentation ganz zu schweigen. Eine Internet-Suche nach einem Problem bleibt ohne Ergebnis, denn die gefundene Lösung ist nur für eine Version 5.0.8 gültig, während die zurzeit eingesetzte Version noch bei 2.5.4 steht. Kein Test ist vorhanden, der die notwendige Änderung im Modul gegen Regression absichert. Ein richtiger Flow kommt hier nicht auf.

Fehlender Flow, Frust und Überforderungen erfordern eine starke innere Haltung beim Umsetzen von Sanierungsprojekten. Diese Haltung bringen in der Regel erfahrene Softwareentwickler mit, die bereits mit unterschiedlichsten Problemen konfrontiert wurden und gelernt haben, dass man sich durch manche Situationen einfach durchbeißen muss. Diese Haltung muss die Grundhaltung des Teams werden. Sie wird von den erfahrenen Entwicklern vorgelebt und täglich neu motiviert.

Neben innerer Haltung sind es kleine, innerhalb einer Woche oder besser innerhalb eines Tages erreichbar Ziele. Montags kommt das Team zusammen und entscheidet gemeinsam, was am Ende der Woche erreicht sein soll. Die Wochenziele werden für alle sichtbar visualisiert. Die Anwendung der Mikado-Methode [Ell14] kann hier helfen. Zusätzlich gibt es ein tägliches Stand-up, in dem die Ziele des Tages besprochen werden. Dies schafft Fokus und Erfolge.

Abschluss

Legacy-Systeme sind wichtig und häufig das Herz der Unternehmen. Dieses Herz schlägt von Jahr zu Jahr langsamer. Bevor es zum Stillstand kommt, entscheiden sich viele Unternehmen für die Neuentwicklung ihres Altsystems. Was häufig nicht gesehen wird: Ab einer bestimmten Systemgröße wird jede Neuentwicklung zwangsläufig in ein neues Legacy-System münden.

Deshalb: Erkenne den Wert deiner Software. Führe eine detaillierte Pain Point-Analyse durch. Bestimme Maßnahmen, die wirklich etwas bringen. Priorisiere und quantifiziere dein Sanierungs-Backlog. Stelle den Wert der existierenden Software den Kosten einer Sanierung und einer Neuentwicklung gegenüber. Motiviere dein Team immer wieder aufs Neue zu einer positiven Grundhaltung gegenüber ergrautem Quellcode. Etabliere und verbessere Arbeitsweisen, die dein Team in einen bestmöglichen Flow beim Umgang mit dem Code versetzen. ||

Literatur & Links

[Cun92] W. Cunningham, The WyCash Portfolio Management System, in: OOPSLA '92 Experience Report, 26.3.1992, siehe: **http://c2.com/doc/oopsla92.html**

[Ell14] O. Ellnestam, D. Brolund, The Mikado Method, Manning, 2014

[Fea04] M. Feathers, Working Effectively with Legacy Code, Prentice Hall, 2004

[Fow04] M. Fowler, StranglerFigApplication, 29.6.2004, siehe: **https://martinfowler.com/bliki/StranglerFigApplication.html**

[Fow14] M. Fowler, BranchByAbstraction, 7.1. 2014, siehe: **https://martinfowler.com/bliki/BranchByAbstraction.html**

[Fri00] M. Friedewald u. a, Analyse und Evaluation der Softwareentwicklung in Deutschland, Dezember 2000, siehe: **http://publica.fraunhofer.de/dokumente/N-3238.html**

[Hun99] A. Hunt, D. Thomas, The Pragmatic Programmer. From Journeyman to Master, Addison-Wesley, 1999

[Jie15] Yun Jie, Ye Li, Consumer Mere Newness Bias, 2015, siehe: **https://mindmodeling.org/cogsci2015/papers/0177/paper0177.pdf**

[RPA] **https://weissenberg-solutions.de/was-ist-robotic-process-automation/**

[Spo00] **https://www.joelonsoftware.com/2000/04/06/things-you-should-never-do-part-i/**

Die Autoren

Ralf Wirdemann
(ralf.wirdemann@codekeepers.de)
ist Geschäftsführer der Softwaresanierungsfirma CodeKeepers GmbH. Gemeinsam mit seinen Kunden erarbeitet er Sanierungsstrategien und begleitet Projekte in der Umsetzungsphase. Er ist Autor der Bücher „Scrum mit User Stories" und „RESTFul Go APIs".

Torsten Lueckow
(torsten.lueckow@codekeepers.de)
ist Geschäftsführer der Softwaresanierungsfirma CodeKeepers GmbH. Er verfügt als Entwickler, Architekt und Projektleiter über eine 20-jährige Berufserfahrung im Java-Umfeld. Er ist Cloud-Experte sowie Autor des Fachbuchs „Spring & Hibernate".