

## Domain-Driven Transformation

Monolithen und Microservices zukunftsfähig machen

» Hier geht's  
direkt  
zum Buch

# DAS VORWORT

## Vorwort

»Alle glücklichen Familien gleichen einander, jede unglückliche Familie ist auf ihre eigene Weise unglücklich.« Dieses Anna-Karenina-Prinzip gilt nicht nur für Familien, sondern auch für Entwicklungsteams von Legacy-Software. Verschiedene Softwaresysteme leiden unter unterschiedlichen »Krankheiten« und wir brauchen eine Vielzahl von Mitteln, um sie alle zu heilen. Die folgenden Krankheiten haben wir in unterschiedlichen Kombinationen in den letzten Jahren bei Legacy-Systemen – egal ob Monolithen oder Microservices – und ihren Organisationen beobachten können:

- Das Legacy-System ist ein **Big Ball of Mud** geworden, weil niemand die Abhängigkeiten kontrolliert hat und nun alles mit allem verbunden ist (s. Kap. 4).
- Die Fachlichkeit ist in einem **großen Domänenmodell** miteinander verwoben, dessen Teile nur begrenzt zusammenpassen oder sich sogar widersprechen (s. Abschnitt 11.4).
- Der **fachliche und technische Sourcecode** ist vermengt, und so wird der Austausch von veralteter Technik oder eine fachliche Erweiterung zu einer Herkulesaufgabe (s. Kap. 7).
- Die Menschen sind in einer **Teamstruktur** organisiert, die für schnelles Vorankommen ungeeignet ist (s. Kap. 8).

In diesem Buch stellen wir die Heilmittel gegen diese Krankheiten vor, die wir in den letzten 20 Jahren bei unserer Arbeit mit **Domain-Driven Design (DDD)** und **Legacy-Software** kennenlernen durften. Dazu gehören: **Refactorings**, **Domain Storytelling**, **Event Storming**, **Team Topologies** und der **Modularity Maturity Index (MMI)**. Wenn wir in unseren Projekten und mit unseren Kunden die Heilung angehen, dann erreichen wir, dass das jeweilige Entwicklungsteam deutlich positiver in die Zukunft schaut und die Motivation steigt. Je weiter die Heilung voranschreitet, desto glücklicher werden auch unsere Fachanwender, Projektleiter und Manager sein, denn die schwergängige und teure Legacy-Software wird stabiler, lässt sich schneller anpassen und erlaubt schließlich sogar (wieder) innovative, zukunftsweisende Erweiterungen.

Unser Ziel mit diesem Buch ist es, Ihnen Mittel an die Hand zu geben, die Ihnen helfen, Ihre Softwaresysteme so aufzuteilen, dass Änderungen möglichst lokale Auswirkungen haben. Dadurch werden die einzelnen Softwaremodule für die Entwicklungsteams wieder beherrschbar und verstehbar und können von entsprechend klei-

nen schlagkräftigen Teams bearbeitet werden. Deshalb finden Sie in diesem Buch eine ganze Reihe Sourcecode-Beispiele. Die Legacy-Systeme, die uns begegnen, sind in ganz unterschiedlichen Programmiersprachen<sup>1</sup> gebaut: Java, C#, PHP, C++ und viele andere. Wir haben uns entschieden, die Beispiele in der Sprache zu schreiben, die im Vergleich am häufigsten auftaucht, nämlich Java. Die Beispiele sind kurz genug, dass sie auch Programmierern anderer Sprachen verständlich sein sollten, und das Vorgehen, das wir beschreiben, ist unabhängig von der Programmiersprache.

Einige Leserinnen und Leser haben möglicherweise unsere anderen Bücher *Langlebige Softwarearchitekturen* [Lilienthal 2019] und *Domain Storytelling* [Hofer & Schwentner 2023] gelesen. Das vor Ihnen liegende Buch ist für uns die Vereinigung und logische Folge aus den beiden anderen Büchern und unserer täglichen Arbeit in unseren Projekten. Die Leser dieser beiden Bücher werden die Beispieldomänen Kino und Autoleasing und unsere Sicht auf Architektur wiedererkennen. Darüber hinaus gibt es viel Neues zu entdecken :-). Wir hoffen sehr, dass Ihnen diese Synthese gefällt, und freuen uns auf Ihr Feedback!

Ganz besonders möchten wir uns an dieser Stelle bei allen bedanken, die uns in den letzten Jahren unterstützt, uns Ideen für dieses Buch gegeben und mit uns diskutiert haben. Die sehr aktive und engagierte **DDD-Community** und unsere Firma, die **WPS** – Workplace Solutions, seien hier als Erstes genannt. Ohne ein gutes Umfeld wäre es uns nicht möglich, die Energie aufzubringen, die es für das Schreiben eines Buches braucht. Viele der hier präsentierten Ideen haben ihren Ursprung bei Christiane Floyd und Heinz Züllighoven. Danke, dass wir von euch lernen durften!

Unsere beiden Kollegen Stefan Hofer und Sönke Magnussen haben unsere Texte gereviewt und uns mit ihren kritischen, konstruktiven Anmerkungen zu einem deutlich besseren Ergebnis verholfen. Unsere Lektorin Christa Preisendanz vom dpunkt.verlag hat uns wieder einmal hervorragend betreut und motiviert. Danke, Christa!

Unsere Familien haben unsere Fokussierung auf ein weiteres Buch voller Verständnis begleitet und uns Freiraum und Unterstützung gegeben. Henning dankt seiner Frau Julia und seinen Kindern Lennart, Joris, Friedo, Tara, Aurelia und Viola für ihre großartige Unterstützung. Carola dankt ihrem Mann, ihrem Sohn und ihren Freunden für einfach alles. Herzlichen Dank an euch alle!

Was die geschlechtergerechte Sprache angeht, haben wir uns für dieses Buch entschieden, abwechselnd pro Kapitel die weibliche und männliche Form zu verwenden. Dann bleibt das Lesen flüssig und jede und jeder kann sich hoffentlich inkludiert fühlen.

*Carola Lilienthal und Henning Schwentner*  
Hamburg, April 2023  
*www.domain-driven-transformation.com*

---

1. Programmierer lieben es ja, darüber fachzusimpeln, welche Programmiersprache die beste ist. Wenn es um das Thema »Bauen von monolithischer Legacy-Software« geht, gibt es einen klaren Gleichstand: Das kann man mit allen einfach schaffen.

---

# 1 Einleitung: Komplexität beherrschen

Wenn man auf der grünen Wiese mit einem Softwareprojekt anfängt, macht alles Spaß und alles ist einfach. Die Entwickler reagieren blitzschnell auf neue Anforderungen. Die Anwender sind begeistert. Die Entwicklung geht in großen Schritten voran.

Über die Lebenszeit des Systems wird sich das ändern. Unweigerlich erhöht sich die Komplexität der Software. Diese ansteigende Komplexität führt zu höherer Fehleranfälligkeit, immer langsamerem Fortschritt und schlechterer Wartbarkeit. Und schließlich braucht es ein halbes Jahr, bis auch nur die kleinste Änderung in Produktion angekommen ist. Aus der blühenden grünen Wiese ist ein schlammiger, stinkender Acker geworden. »Altsystem«, »Legacy-Software«, »Big Ball of Mud«, »Monolith«, »Gummistiefelprojekt« sind die wenig schmeichelhaften Namen, mit denen diese Art von Systemen versehen werden.

»The most fundamental problem in software development is complexity. There is only one basic way of dealing with complexity: divide and conquer.«

*Bjarne Stroustrup*

Aber es gibt Hoffnung! Auch diesen in die Jahre gekommenen Systemen kann Flexibilität, Fehlerrobustheit und Entwicklungsgeschwindigkeit zurückgegeben werden. Kernaufgabe ist dabei die Beherrschung (und Aufteilung) von Komplexität.

---

## 1.1 Komplexität

Um einordnen zu können, wie mit Komplexität bei der Softwareentwicklung umgegangen werden kann, ist der Untertitel von Eric Evans' Buch *Domain-Driven Design* ein guter Wegweiser: *Tackling Complexity in the Heart of Software* [Evans 2004]. Evans gibt uns das Versprechen, dass das von ihm beschriebene Vorgehen Komplexität in der Software(-entwicklung) reduziert oder zumindest handhabbar macht. Domain-Driven Design hilft dabei auf unterschiedlichen Ebenen. In Abbildung 1–1 sind die verschiedenen Aspekte von Komplexität, die uns bei der Softwareentwicklung begegnen, in Anlehnung an [Lilienthal 2008] und [Lilienthal 2019] zusammengefasst.

	essenziell	akzidentell
Problemraum	<p>inhärente Komplexität der Fachdomäne</p> <p>inhärente Komplexität der Domänenmodellierung</p>	<p>Missverständnisse über die Fachdomäne</p> <p>überflüssige Konzepte schlechter Subdomänenschnitt</p>
Lösungsraum	<p>inhärente Komplexität im Domänenmodell und der Architektur</p> <p>inhärente Komplexität der Technologie</p>	<p>überflüssige Lösungsanteile schlechtes Design/Architektur</p> <p>Missverständnisse über die Technologie</p>

**Abb. 1-1** Komplexität und ihre Quellen

Im weiteren Verlauf werden wir sehen, dass die Grundlagen, die wir im ersten Teil des Buches einführen, Methoden und Heuristiken für bestimmte in Abbildung 1-1 aufgeführte Komplexitätsquellen anbieten. Wir werden diese Abbildung in den nächsten Abschnitten daher wiedersehen und sie Schritt für Schritt mit weiteren Informationen füllen.

## 1.2 Herkunft der Komplexität: Problem- und Lösungsraum

Die Herkunftsdimension von Komplexität bei der Softwareentwicklung ist in Abbildung 1-1 auf der vertikalen Achse dargestellt. Diese Komplexität entsteht aus dem Wunsch, für eine Domäne eine Software zu bauen; also von der Domäne, einem Teil der wirklichen Welt (dem sog. *Problemraum*), zu einer Software mit passenden Geschäftsprozessen (in den sog. *Lösungsraum*) zu kommen. Wir haben somit zwei Quellen:

- **Problemraum:** die Komplexität der Fachdomäne, für die das Softwaresystem gebaut wurde – die sogenannte *problemabhängige Komplexität*
- **Lösungsraum:** die Komplexität, die in der Umsetzung bei der Modellierung, in der Architektur und bei der Verwendung der gewählten Technologie entsteht – die sogenannte *lösungsabhängige Komplexität*

In den letzten Jahrzehnten hat unser Berufsstand bei der Softwareentwicklung immer wieder die Erfahrung gemacht, dass es sehr schwer, wenn nicht gar unmöglich ist, die Komplexität im Problemraum am Anfang eines Projekts abzuschätzen und in einem Pflichtenheft niederzuschreiben. Deshalb dauern Projekte häufig viel länger als geplant,

deshalb funktioniert das Wasserfallmodell so schlecht und deshalb sind agile Methoden so populär geworden. Richtig angewandt sind agile Methoden nämlich eine gute Hilfe, denn sie versuchen, die Komplexität im Problemraum in kleineren Häppchen Schritt für Schritt verschränkt mit der Umsetzung zu erfassen.

---

## 1.3 Art der Komplexität: essenziell vs. akzidentell

In Abbildung 1–1 findet sich auf der horizontalen Dimension die Art der Komplexität mit den beiden Begriffen **essenziell** und **akzidentell**. Welche Anteile von Komplexität werden mit diesen beiden Begriffen beschrieben?

Haben wir das weltbeste und erfahrenste Team im Einsatz, dann können wir erwarten, dass wir eine gute Softwarelösung bekommen. Eine Softwarelösung, die eine für das Problem angemessene Komplexität aufweist. Ist die vom Entwicklungsteam gewählte Lösung komplexer als das eigentliche Problem, dann konnte das Entwicklungsteam die essenzielle Komplexität nicht richtig erfassen und die Lösung ist zu komplex, also nicht gut. Dieser Unterschied zwischen besseren und schlechteren Lösungen wird mit essenzieller und akzidenteller Komplexität bezeichnet.

Essenzielle Komplexität nennt man die Art von Komplexität, die im Wesen einer Sache liegt, also Teil seiner Essenz ist. Diese Komplexität lässt sich niemals auflösen oder durch einen besonders guten Entwurf vermeiden. Will man eine Software für eine komplexe Domäne bauen, für eine Domäne, bei der eine hohe Komplexität Teil ihres Wesens – ihrer Essenz – ist, so wird auch die Komplexität in der Software im Lösungsraum hoch ausfallen müssen. So hat eine Softwarelösung für ein Containerterminal eine größere essenzielle Komplexität als eine Software zur Verwaltung von Vereinsmitgliedern in einem Ruderclub.

### 1.3.1 Quellen akzidenteller Komplexität

Akzidentelle Komplexität ist im Gegensatz zur essenziellen nicht notwendig. Sie entsteht aus folgenden Gründen:

- aus *Missverständnissen bei der Analyse der Fachdomäne*, sodass uns die Fachdomäne komplexer erscheint, als sie eigentlich ist;
- weil wir glauben, dass die Anwender für bestimmte Arbeitsschritte Unterstützung brauchen, obwohl sie eigentlich ganz anders arbeiten, und wir somit *Sachen bauen, die keiner braucht*;
- weil wir uns für ein *schlechtes Design und eine schlechte Architektur* entscheiden bzw. Design und Architektur mit der Zeit durch Wartung, Änderung und Unkenntnis erodieren;
- weil wir unpassende oder veraltete *Technologie* einsetzen, die ersetzt werden muss, oder weil wir die Technologie nicht so verwenden, wie es eigentlich vorgesehen war.

Wird bei der Entwicklung aus Unkenntnis oder mangelndem Überblick keine einfache Lösung gefunden, so ist das Softwaresystem unnötig komplex. Beispiele hierfür sind: Mehrfachimplementierungen, Einbau nicht benötigter Funktionalität und das Nichtbeachten softwaretechnischer Entwurfsprinzipien. Akzidentelle Komplexität kann von Entwicklern aber auch billigend in Kauf genommen werden, wenn sie z.B. gern neue, aber für das zu bauende Softwaresystem überflüssige Technologie ausprobieren wollen.

### 1.3.2 Entscheidungsbereiche von Softwarearchitektur

Im Foundation Training des iSAQB – International Software Architecture Qualification Board (s. [iSAQB 2023]) –, der Basisausbildung für Softwarearchitekten, werden vier Bereiche definiert, die für die Entscheidungen über die Softwarearchitektur relevant sind: Anforderungsermittlung, Modellbildung, fachliche Architektur und technische Architektur. Diese vier Bereiche bzw. die in diesen Bereichen vorhandenen Methoden und Techniken lassen sich sehr gut auf Abbildung 1–1 anwenden.

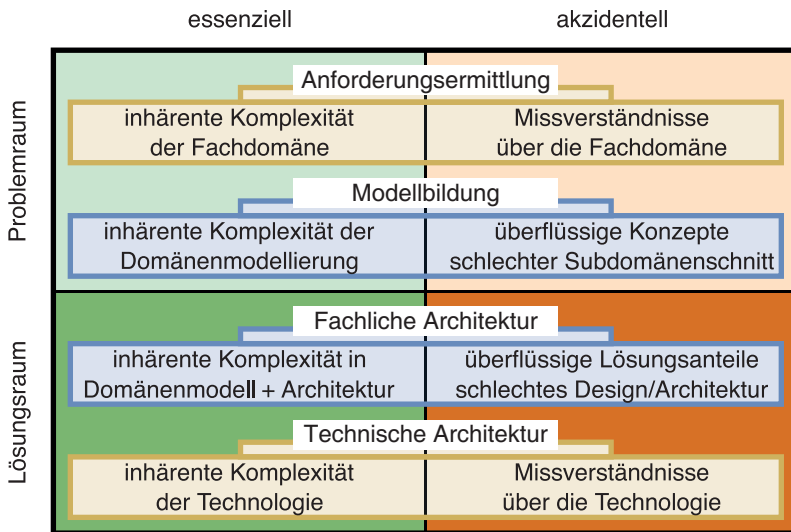
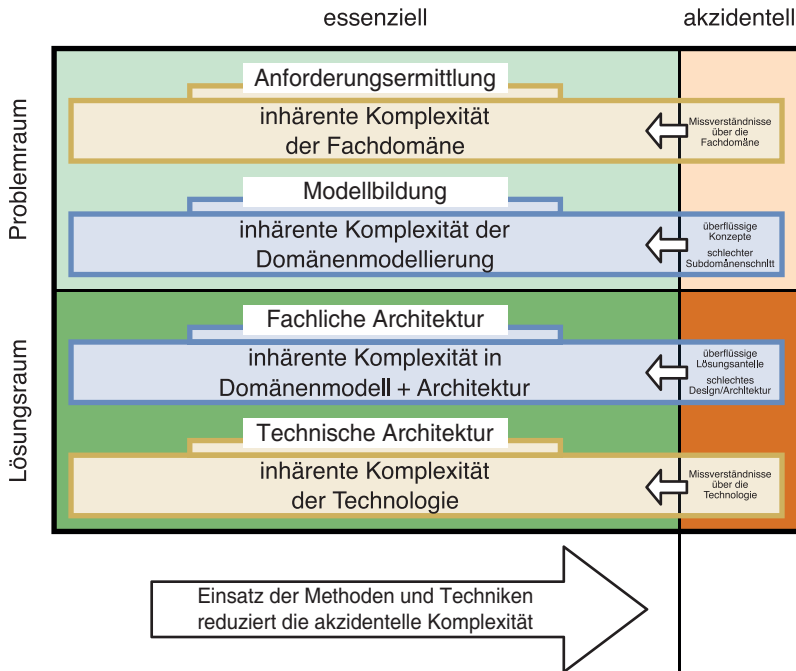


Abb. 1–2 Komplexität und Architektur

In Abbildung 1–2 können Sie sehen, dass Anforderungsermittlung und Modellbildung im Problemraum sowohl auf der essenziellen als auch auf der akzidentellen Seite ansetzen. Fachliche und technische Architektur wirken auf die essenzielle und akzidentelle Komplexität im Lösungsraum. Die Methoden und Techniken der Anforderungsermittlung, der Modellbildung, der fachlichen und technischen Architektur werden sowohl bei der Neuentwicklung als auch bei der Transformation von Legacy-Systemen eingesetzt, um die essenzielle Komplexität herauszudestillieren und möglichst viel akzidentelle Komplexität in unseren Systemen zu eliminieren (s. Abb. 1–3).



**Abb. 1-3** Einsatz von Methoden gegen Komplexität

Vollständig werden wir die akzidentelle Komplexität selbst mit den besten Methoden nicht auflösen können, denn wir befinden uns bei der Softwareentwicklung und Softwarewartung in einem kontinuierlichen Lernprozess. Dieser Lernprozess beinhaltet, dass wir Teile der Fachlichkeit und der Technologie immer wieder missverstehen und überflüssige oder schlechte Lösungen für Teilbereiche konzipieren oder implementieren. Deshalb bleibt selbst bei perfekter Umsetzung auf allen Ebenen in Abbildung 1-3 ein Anteil akzidenteller Komplexität bestehen.

## 1.4 Komplexität in Legacy-Systemen

Bei Legacy-Systemen, die schon eine Anzahl von Jahren auf dem Buckel haben und von verschiedensten Generationen von Entwicklern gewartet und verändert worden sind, ist viel akzidentelle Komplexität zu erwarten. Gründe dafür sind:

- Die Arbeit der Anwender und damit die Fachdomäne hat sich im Laufe der Jahre verschoben. Neue Funktionalität wurde der Software hinzugefügt, um die Anwender auch bei ihren neuen oder etwas veränderten Aufgaben zu unterstützen. Aber die *Domänenmodellierung* und die *fachliche Modularisierung* des Softwaresystems in der Architektur wurden an diese Verschiebung nicht oder nicht ausreichend angepasst.



- Die Entwickler und Architekten des Systems haben über die Jahre aus Zeitdruck, aus Unkenntnis oder aus Desinteresse das *ursprüngliche Design und die Architektur verwässert* und die einzelnen Teile immer stärker miteinander verwoben und so voneinander abhängig gemacht.
- Jede *Technologie veraltet* in unserer Disziplin und muss alle 10 bis 15 Jahre durch etwas Neues ersetzt werden. Bei Legacy-Systemen wird der richtige Zeitpunkt häufig verpasst.

Als Evans sein Buch 2004 schrieb, ging es ihm um die Neuentwicklung von Software, also um die Frage: »Was muss ich tun, damit mein *neues* Softwaresystem domänengetrieben designt ist?« In diesem Buch übertragen wir die Ideen von Domain-Driven Design (DDD) auf Legacy-Systeme. Wir durften in den vergangenen Jahren feststellen, dass wir in DDD Konzepte und Anleitungen finden, die sich für die nötige Neukonzeptionierung und den zum Teil umfassenden Umbau von Legacy-Systemen hervorragend eignen.

Schlüssel dazu ist die schrittweise und behutsame Modernisierung mithilfe einer Kombination aus DDD und weiteren Werkzeugen, die wir im Laufe des Buches einführen.

---

## 1.5 Struktur dieses Buches

Das Buch besteht neben dieser Einleitung (Kap. 1) – in der wir unseren Begriff von Komplexität erläutert haben – aus drei Teilen, dem Abschluss und einer Website:

*Teil I, »Grundlagen für Domain-Driven Transformation«, fasst das benötigte Basiswissen zusammen. Hier zeigen wir, was Domain-Driven Design (Kap. 2) und Collaborative Modeling (Kap. 3) sind und welche Architekturkonzepte (Kap. 4) für das Verständnis dieses Buches wichtig sind. Von diesen drei Kapiteln sollten Sie sich die Abschnitte bzw. Themen herausuchen, in denen Sie sich nicht firm fühlen.*

Im abschließenden Kapitel von Teil I beantworten wir die Frage nach dem richtigen Vorgehen (Kap. 5). Dieses Kapitel ist wichtig für das Gesamtverständnis der nachfolgenden Teile – wir empfehlen es Ihnen daher auf jeden Fall.

*Teil II, »Technische, taktische und teamorganisatorische Domain-Driven Transformation«, zeigt, wie das Altsystem auf diese große Verbesserung vorbereitet werden kann. Grundsätzlich lohnt es sich, das Altsystem technisch zu stabilisieren (Kap. 6) und mit taktischer Domain-Driven Transformation die Fachlichkeit im Sourcecode zu stärken (Kap. 7). Und schließlich sollte das Entwicklungsteam Klarheit haben, wie die Teamstrukturen aussehen sollen, wenn man sich auf den Weg einer strategischen Transformation des Altsystems begibt (Kap. 8).*

*Teil III, »Strategische Domain-Driven Transformation«*, stellt das von uns entwickelte Vorgehen zum Zerlegen vor. Das Vorgehen besteht aus vier Schritten: Es beginnt mit dem Wiederentdecken der Fachdomäne (Kap. 9), führt über das Modellieren der fachlichen Soll-Architektur (Kap. 10) und den Abgleich Ist- mit Soll-Architektur (Kap. 11) und endet beim Priorisieren und Durchführen der Umbaumaßnahmen (Kap. 12).

Im »*Ausblick: Domain Patterns und ihre Umsetzung in Kontexten*« (Kap. 13) stellen wir Ihnen unsere Ideen dazu vor, welche Domänenmuster es gibt, und wie sich das jeweilige Domänenmuster auf die Domain-Driven Transformation auswirkt. Im »*Fazit*« (Kap. 14) fassen wir das Buch abschließend zusammen.

Um die Transformation durchzuführen, braucht man *Domain-Driven Refactorings*, die online in einem Katalog gesammelt werden: <https://hschwentner.io/domain-driven-refactorings>. Wir geben an verschiedenen Stellen im Buch konkrete Referenzen auf das jeweilige zu verwendende Refactoring.