mitp

Robert C. Martin

Mit Beiträgen von James Grenning und Simon Brown

Vorwort von Kevlin Henney

Nachwort von Jason Gorman

Deutsche Ausgabe

Clean Architecture

Das Praxis-Handbuch für professionelles Softwaredesign

Regeln und Paradigmen für effiziente Softwarestrukturierung

Inhaltsverzeichnis

Vorwort

Worüber reden wir, wenn wir uns über »Architektur« im Allgemeinen unterhalten?

Wie bei allen metaphorischen Annäherungen kann auch die Betrachtung einer Software unter architektonischen Gesichtspunkten gleichermaßen viel verhüllen wie offenbaren. Ebenso kann sie mehr versprechen, als sie zu liefern imstande ist, oder mehr liefern, als sie ursprünglich zu versprechen schien.

Der offenkundige Reiz der Architektur besteht in ihrer Struktur, deshalb steht Letztere auch im Bereich der Softwareentwicklung im Fokus sämtlicher Paradigmen und Überlegungen – Komponenten, Klassen, Funktionen, Module, Layer und Services, egal, ob Mikro oder Makro. Allerdings erweist sich die Gesamtstruktur vieler Softwaresysteme nicht selten als fragwürdig oder widersinnig – etwa wie die sowjetischen Kollektivbetriebe, die als Vermächtnis für das Volk bestimmt waren, undenkbare Jenga-Türme, die bis zum Himmel hinaufragen, oder wundersame, unter riesigen Schlammlawinen begrabene archäologische Fundschichten. Dass Softwarestrukturen in der gleichen Art und Weise unserer Intuition folgen, wie dies auch bei Gebäudestrukturen der Fall ist, lässt sich häufig nur schwer erkennen.

Gebäude besitzen dagegen eine unübersehbare physische Struktur – ob in Stein oder Beton verwurzelt, ob hoch nach oben ragend oder weit in die Breite verlaufend, ob groß oder klein, ob atemberaubend oder schlicht und banal. Bei Bauwerken gibt es kaum eine Alternative, als sie nach der Physik der Schwerkraft und den verwendeten Materialien auszurichten. Im Gegensatz dazu hat Software – außer im Sinne der Realitätstreue – wenig für die Schwerkraft übrig. Und woraus besteht Software? Anders als Gebäude, die aus Ziegeln, Beton, Holz, Stahl und Glas gefertigt werden, ist Software eben nur aus Software gemacht. Große Softwarekonstrukte setzen sich aus kleineren Softwarekomponenten zusammen, die wiederum aus noch kleineren Softwarekomponenten bestehen und so weiter, und so fort. Oder, wie es Stephen W. Hawking in seinem Buch »Eine kurze Geschichte der Zeit« ausdrückt:

> *Da stehen lauter Schildkröten aufeinander.*[1]

Wenn wir über Softwarearchitektur im Speziellen reden, geht es darum, dass Software in ihrer Beschaffenheit rekursiv und fraktal, im Code prägnant und richtung-

1 Stephen W. Hawking, *Eine kurze Geschichte der Zeit*, Rowohlt Verlag, 2004.

weisend ist. Einfach alles hat mit Details zu tun. Zwar spielen ineinandergreifende Detailebenen auch bei der Architektur von Gebäuden eine Rolle, in Bezug auf Software ist es allerdings kaum sinnvoll, über physische Maßstäbe nachzudenken. Software besitzt eine Struktur – eigentlich jede Menge und zahlreiche Arten von Strukturen –, deren Vielfalt das Spektrum der physischen Gebäudestrukturen problemlos in den Schatten stellt. Man kann durchaus behaupten, dass dem Design in der Softwareentwicklung mehr Einsatz und Aufmerksamkeit zuteilwird, als dies in der Gebäudearchitektur der Fall ist – und insofern ist es auch nicht unbedingt abwegig, die Softwarearchitektur als architektonischer zu betrachten als die Gebäudearchitektur!

Aber physische Maßstäbe sind für den menschlichen Verstand besser zu begreifen. Sie bieten uns Orientierung in der Welt, deshalb halten wir stets Ausschau danach. Und sicherlich sind die einzelnen Kästen in schematischen PowerPoint-Darstellungen ansprechend und vermitteln einen klaren visuellen Eindruck, trotzdem geben sie nicht den vollen und lückenlosen Umfang der Architektur eines Softwaresystems wider. Zweifellos bieten sie eine bestimmte Sicht auf einen architektonischen Aufbau; die Kästen allerdings fälschlicherweise mit dem großen Ganzen – also der Architektur selbst – zu verwechseln, heißt definitiv, das große Ganze – und somit die Architektur als solche – aus den Augen zu verlieren: Softwarearchitektur sieht nicht irgendwie besonders aus. Eine spezifische Visualisierung ist eine Entscheidungsfrage, keine Gegebenheit. Vielmehr handelt es sich um eine Entscheidung, die auf einer weiteren Auswahl von Möglichkeiten basiert, nämlich: was enthalten sein soll, was durch Form oder Farbgebung hervorgehoben werden soll, was durch Gleichförmigkeit oder Auslassung heruntergespielt werden soll. Eine Sichtweise hat gegenüber einer anderen nichts Natürliches oder Intrinsisches an sich.

Nun mag es nicht viel Sinn machen, sich im Kontext der Softwarearchitektur mit physikalischen Gesetzmäßigkeiten und physischen Maßstäben auseinanderzusetzen, dennoch müssen wir auch in diesem Bereich bestimmte physische Einschränkungen bedenken und entsprechend berücksichtigen. Prozessorgeschwindigkeiten und Netzwerkbandbreiten können ein hartes Urteil über die Performance eines Systems fällen. RAM- und Datenspeicherkapazitäten können den Ambitionen jeder Codebasis Grenzen setzen. Software mag einer dieser Stoffe sein, aus denen Träume gemacht sind, sie wird aber trotz allem in einer physischen Welt betrieben.

> *Das ist das Ungheuerliche in der Liebe, dass der Wille unendlich ist und die Ausführung beschränkt, dass das Verlangen grenzenlos ist und die Tat ein Sklave der Beschränkung.*
>
> – William Shakespeare[2]

2 Shakespeare, *Troilus und Cressida*, um 1601, Erstdruck 1610, erste deutsche Übers. von Johann Joachim Eschenburg, 1777. Hier übersetzt von Wolf Graf Baudissin, Georg Andreas Reimer, Berlin, 1832.

Es ist die physische Welt, in der wir ebenso wie unsere Unternehmen und unsere Volkswirtschaften existieren. Und diese Tatsache liefert uns einen weiteren Kalibrierungsgesichtspunkt, anhand dessen wir die Softwarearchitektur verstehen können – andere, weniger physische Kräfte und Größen, auf deren Grundlage wir uns verständigen und argumentieren können.

> *Architektur repräsentiert die signifikanten Designentscheidungen, die ein System formen und gestalten, wobei die Signifikanz an den Kosten von Änderungen bemessen wird.*
>
> – Grady Booch

Zeit, Geld und Aufwand geben uns einen Maßstab vor, um das Große und das Kleine, das Wesentliche und das weniger Wesentliche zu sortieren und die architektonischen Aspekte von dem Rest zu unterscheiden. Dieser Maßstab sagt uns auch, wie wir feststellen können, ob eine Architektur gut ist oder nicht: Eine gute Softwarearchitektur erfüllt die Bedürfnisse aller User, Entwickler und Product Owner (Produkteigentümer), und zwar nicht nur zu einem gegebenen Zeitpunkt, sondern langfristig.

> *Wenn Sie denken, eine gute Architektur sei teuer, dann probieren Sie es mal mit einer schlechten.*
>
> – Brian Foote und Joseph Yoder

Die Modifikationen und Anpassungen, die im Rahmen einer Systementwicklung typischerweise vorzunehmen sind, sollten nicht von der Art sein, dass sie kostspielig und schwer zu realisieren sind und eine jeweils eigene Projektverwaltung erforderlich machen, sondern in die täglichen und wöchentlichen Arbeitsabläufe eingebunden werden können.

Und das bringt uns zu einem nicht unerheblichen Physik-orientierten Problem: der Zeitreise. Wie wissen wir, welcher Art diese typischen Modifikationen bzw. Anpassungen sein werden, sodass wir die damit einhergehenden signifikanten Entscheidungen darauf ausrichten können? Wie reduzieren wir den zukünftigen Entwicklungsaufwand und die Kosten, ohne Kristallkugeln und Zeitmaschinen zu Hilfe zu nehmen?

> *Architektur ist die Menge der Entscheidungen, von denen Sie wünschten, dass Sie sie bereits frühzeitig in einem Projekt richtig treffen, bei denen die Wahrscheinlichkeit, sie auch tatsächlich richtig zu fällen, aber nicht unbedingt höher ist als bei allen anderen Entscheidungen auch.*
>
> – Ralph Johnson

Das Vergangene zu verstehen, ist schon schwierig genug. Unser Verständnis von dem Gegenwärtigen ist bestenfalls vage – und die Vorhersage des Zukünftigen ist alles andere als trivial.

An diesem Punkt verzweigt der Weg in viele Richtungen.

Auf dem dunkelsten Pfad wartet die Vorstellung, dass starke und stabile Architekturen direkte Abkömmlinge von Autorität und Starrheit sind. Ist eine Modifikation kostenintensiv, wird sie verworfen, die ursächlichen Beweggründe werden kleingeredet oder vollständig in bürokratischen Abgründen versenkt. Das Mandat des Architekten ist total und totalitär – und als Folge davon verkümmert die Architektur zu einer Dystopie für ihre Entwickler und eine ständige Quelle der Frustration für alle anderen.

Entlang eines anderen Abzweigs richtet sich das Interesse hingegen auf die sauberste Variante. Hier wird der »Weichheit« der Software Rechnung getragen und darauf hingearbeitet, sie als vorrangigste Eigenschaft des Systems zu bewahren. Ebenso wird nicht nur die Tatsache berücksichtigt, dass wir auf der Grundlage unvollständigen Wissens arbeiten, sondern auch anerkannt, dass dies für uns menschliche Wesen zudem etwas ist, worin wir gut sind. Diese Vorgehensweise kommt unseren Stärken mehr entgegen als unseren Schwächen. Wir erschaffen Dinge und wir entdecken Dinge. Wir stellen Fragen und führen Experimente durch. Eine gute Architektur kommt genau dann zustande, wenn wir sie als Reise und nicht als Ziel verstehen, mehr als einen fortlaufenden Prozess des Untersuchens denn als unumstößliches Artefakt.

> *Architektur ist eine Hypothese, die durch Implementierung und Bewertung bewiesen werden muss.*
>
> – Tom Gilb

Das Beschreiten dieses Pfades erfordert Sorgfalt und Aufmerksamkeit, Überlegungen und Beobachtung, Praxis und Prinzipien. Auf den ersten Blick mag sich dies nach einem schleppenden Prozess anhören, im Endeffekt hängt jedoch alles davon ab, wie Sie den Weg beschreiten.

> *Der einzige Weg, um schnell voranzukommen, ist gut voranzukommen.*
>
> – Robert C. Martin

Genießen Sie die Reise.

Kevlin Henney, Mai 2017

Einleitung

Der Titel dieses Buches lautet *Clean Architecture*, zu Deutsch also »Saubere Architektur«. Zugegeben, das ist eine recht selbstbewusst erscheinende Überschrift. Warum also habe ich mich für diesen Titel entschieden – und wieso habe ich dieses Buch überhaupt geschrieben?

Meine erste Programmzeile tippte ich 1964 im Alter von zwölf Jahren. Mit dem Schreiben dieses Buches begann ich im Jahr 2016 – ich programmiere inzwischen also bereits seit mehr als einem halben Jahrhundert. In all diesen Jahren blieb es natürlich nicht aus, dass ich ein paar Dinge über das Strukturieren von Softwaresystemen gelernt habe – Dinge, von denen ich glaube, dass sie auch für andere nützlich und wertvoll sind.

Dass ich mir dieses Wissen aneignen konnte, ist der Tatsache zu verdanken, dass ich in der Vergangenheit sehr viele Systeme entwickelt habe, sowohl große als auch kleine. Ich errichtete kleine eingebettete Systeme (Embedded Systems) ebenso wie große Stapelverarbeitungssysteme. Außerdem Echtzeit- und Websysteme. Und nicht zu vergessen Konsolen-Apps, GUI-Anwendungen, Prozesssteuerungsapplikationen, Spiele, Kontierungssysteme, Telekommunikationssysteme, Designtools, Zeichenanwendungen sowie viele, viele andere Systeme.

Dasselbe gilt für Singlethread-Anwendungen, Multithread-Anwendungen, Anwendungen mit wenigen schwergewichtigen Prozessen, Anwendungen mit vielen leichtgewichtigen Prozessen, Multiprozessoranwendungen, Datenbankanwendungen, mathematische Anwendungen, algorithmische Geometrieanwendungen sowie viele, viele andere Anwendungen.

Ich habe wirklich jede Menge Systeme entwickelt und Anwendungen programmiert. Und all diese Projekte brachten mich ausnahmslos zu einer erstaunlichen Erkenntnis:

Die Regeln der Architektur sind stets dieselben!

Erstaunlich ist dies vor allem insofern, als sich die Systeme, mit denen ich im Laufe der Jahre befasst war, radikal voneinander unterscheiden. Wie also kommt es, dass sie auf ähnlichen Architekturregeln basieren, wenn sie doch so verschieden sind? Meine Schlussfolgerung bezüglich dieser Frage lautet: *Weil die Regeln der Softwarearchitektur von allen anderen Variablen unabhängig sind.*

Bedenkt man dann noch den Wandel, der sich in den letzten 50 Jahren im Bereich der Hardware vollzogen hat, ist diese Erkenntnis umso bemerkenswerter. Meine ersten Programmierversuche machte ich auf Maschinen von der Größe eines Haushaltskühlschranks, mit einer Taktzeit von einem halben Megahertz, 4 KB Kernspeicher, 32 KB Festplattenspeicher und einer Teletype-Schnittstelle, die gerade mal zehn Zeichen pro Sekunde zuließ. Und nun sitze ich hier im Bus auf einer Rundreise durch Südafrika und schreibe dieses Geleitwort auf einem MacBook mit vier i7-Prozessorkernen, von denen jeder einzelne mit 2,8 Gigahertz läuft. Dieses Gerät verfügt über 16 GB RAM-Speicher, eine SSD-Festplatte mit einer Kapazität von einem Terabyte und ein Retina-Display mit einer Auflösung von 2.880x1.800, das in der Lage ist, extrem hochauflösende Videos abzuspielen. Die in den letzten Jahrzehnten in puncto Rechenpower erzielten Fortschritte sind geradezu atemberaubend. Jede halbwegs vernünftige Analyse wird bestätigen, dass mein MacBook mindestens 1022 Mal leistungsfähiger ist als die ersten Computer, mit denen ich seinerzeit vor einem halben Jahrhundert angefangen habe.

Eine 22-fache Zehnerpotenz ist schon eine geradezu unfassbare Größenordnung. Das entspricht der Anzahl der Angströms von der Erde bis nach Alpha Centauri. Oder auch der Anzahl der in dem Kleingeld in Ihrer Hosentasche oder Geldbörse enthaltenen Elektronen. Und doch beschreibt diese Zahl – *mindestens* – den inzwischen erzielten Anstieg der Rechenleistung, wie ich ihn in meiner bisherigen Lebenszeit erlebt habe.

Legt man nun diese gigantischen Fortschritte in Bezug auf die Rechnerperformance zugrunde, wie hat sich das dann auf die Software ausgewirkt, die ich schreibe? Es steht außer Frage, dass sie umfangreicher geworden ist: Früher hielt ich bereits ein aus 2.000 Zeilen bestehendes Programm für gewaltig – immerhin entsprach das einer etwa fünf Kilo schweren Kiste voller Lochkarten. Im Vergleich dazu gelten heutzutage erst Programme ab 100.000 Zeilen als groß.

Abgesehen davon ist die Software aber auch erheblich performanter geworden. Wir können inzwischen Dinge damit bewerkstelligen, die wir uns in den 1960er-Jahren nicht mal im Traum hätten vorstellen können. Die Science-Fiction-Bücher bzw. -Filme *Colossus, Revolte auf Luna* und *2001: Odyssee im Weltraum* waren allesamt Versuche, Zukunftsszenarien von unserer aktuellen Gegenwart abzubilden, die jedoch reichlich am Ziel vorbeigeschossen sind. In all diesen Fiktionen herrschte die Vorstellung von riesigen Maschinen mit Empfindungsvermögen oder einem Bewusstsein vor – was wir jedoch stattdessen vorweisen können, sind unglaublich winzige Maschinen, die trotzdem immer noch nur Maschinen sind.

Und es gibt noch eine Sache, die auffällt, wenn man die Software, die uns heute zur Verfügung steht, mit der von damals vergleicht: *Sie enthält immer noch dieselben Bestandteile*. Sie besteht wie gehabt aus `if`-Anweisungen, Zuweisungskommandos und `while`-Schleifen.

Nun mögen Sie einwenden, dass wir mittlerweile auf viel bessere Programmiersprachen und überlegenere Paradigmen zurückgreifen können – immerhin programmieren wir inzwischen in Java oder C# oder Ruby. Und wir verwenden objektorientiertes Design. Das ist zwar alles richtig, dennoch ist der Code an sich nach wie vor bloß eine Ansammlung von *Sequenzen* (Abfolgen), *Selektionen* (Verzweigungen) und *Iterationen* (Schleifen) – ganz genau so, wie es schon in den 1950er- und 1960er-Jahren der Fall war.

Wirft man einen genauen Blick auf die Praxis der Computerprogrammierung, stellt man fest, dass sich in den letzten 50 Jahren tatsächlich nur sehr wenig geändert hat. Sicher, die Sprachen sind ein bisschen besser geworden. Und die Tools sind sogar in exorbitantem Maße besser geworden. Die grundlegenden Bausteine eines Computerprogramms selbst haben sich allerdings nicht verändert.

Hätte ich 1966 eine Computerprogrammiererin[1] per Zeitreise in das Jahr 2016 befördert, sie an mein MacBook mit der *IntelliJ-IDE* (**I**ntegrated **D**evelopment **E**nvironment, integrierte Entwicklungsumgebung) gesetzt und ihr Java gezeigt, hätte sie sicherlich 24 Stunden gebraucht, um sich von dem Schock zu erholen. Aber unmittelbar danach wäre sie bereits in der Lage gewesen, Code zu schreiben – bei genauerer Betrachtung unterscheidet sich Java nämlich gar nicht so sehr von C oder auch von Fortran.

Im umgekehrten Fall, wenn ich Sie in das Jahr 1966 zurückbeamen und Ihnen zeigen würde, wie Sie PDP-8-Code schreiben und bearbeiten können, indem Sie mithilfe einer Teletype-Tastatur, die lediglich zehn Zeichen pro Sekunde schafft, einen Lochstreifen erstellen, müssten Sie sich vermutlich erst einmal 24 Stunden von der Enttäuschung erholen – doch dann wären Sie ebenfalls in der Lage, Code zu schreiben, denn: Der Code als solcher hat sich schlicht und ergreifend nicht allzu sehr verändert.

Und genau das ist das Geheimnis: Diese Unveränderlichkeit des Codes ist der Grund dafür, dass die Regeln der Softwarearchitektur über die verschiedenen Systemtypen und -variationen hinweg so konstant sind. Die Regeln, denen die Softwarearchitektur unterliegt, sind Regeln hinsichtlich der Anordnung und Zusammensetzung der einzelnen Bausteine von Programmen. Und da diese Bausteine allgemeingültig sind und sich nicht verändert haben, sind auch die Regeln für deren Anordnung gleichermaßen allgemeingültig und unveränderlich.

Nun mögen Programmierer der jüngeren Generation diese Aussage schlichtweg für Unsinn halten. Möglicherweise beharren sie sogar darauf, dass heutzutage alles neu und anders sei und die Regeln der Vergangenheit passé seien. Diejenigen, die so denken, täuschen sich jedoch. Die Regeln haben sich *nicht* geändert. Trotz all der neuen Programmiersprachen, all der neuen Frameworks und all der

1 Mit hoher Wahrscheinlichkeit hätte es sich hierbei um eine Frau gehandelt, denn damals war die Zunft der Programmierer zum überwiegenden Teil weiblich.

Paradigmen sind sie auch heute noch genau dieselben wie 1946, als Alan Turing den ersten Maschinencode schrieb.

Eines hat sich jedoch in der Tat geändert: Damals wussten wir noch nicht, wie diese Regeln genau lauteten. Und deshalb haben wir auch wieder und wieder dagegen verstoßen. Jetzt allerdings, nachdem wir ein halbes Jahrhundert lang Erfahrungen sammeln konnten, verstehen wir diese Regeln.

Und einzig und allein um genau diese Regeln – diese zeitlosen, unveränderlichen Regeln – geht es in diesem Buch.

Hinweis

Sollte es Updates, Korrekturen oder Ergänzungen zum Buch geben, finden Sie diese auf der Webseite des mitp-Verlags unter `www.mitp.de/724`, sobald sie verfügbar sind.

Über den Autor

Robert C. Martin (auch bekannt als »Uncle Bob«) ist seit 1970 als Softwareprogrammierer tätig. Er ist Mitbegründer der Organisation *cleancoders.com*, die Online-Videotrainings für Softwareentwickler bereitstellt. Des Weiteren gründete er das Unternehmen *Uncle Bob Consulting LLC*, das Dienstleistungen in den Bereichen IT-Beratung, Training und Skill Development für große Firmen weltweit anbietet. Außerdem war er als Master Craftsman in dem in Chicago ansässigen IT-Consulting-Unternehmen *8th Light Inc*. beschäftigt. Martin veröffentlichte Dutzende von Artikeln in diversen Handelsmagazinen und tritt regelmäßig als Redner bei internationalen Konferenzen und Kongressen auf. Er war drei Jahre lang Chefredakteur des Computermagazins *C++ Report* und ist zudem Erster Vorsitzender der *Agile Alliance*, einer gemeinnützigen Organisation zur Förderung der im *Agile Manifesto* festgelegten Konzepte der Agilen Softwareentwicklung.

Martin hat zahlreiche Bücher geschrieben sowie redaktionell bearbeitet, darunter Titel wie *Clean Coder: Verhaltensregeln für professionelle Programmierer* (mitp, 2014), *Clean Code – Refactoring, Patterns, Testen und Techniken für sauberen Code* (mitp, 2009), *UML for Java Programmers* (Pearson, 2003), *Agile Software Development* (Prentice Hall International, 2013), *Extreme Programming in Practice* (Addison-Wesley, 2001), *More C++ Gems* (Cambridge University Press, 2000), *Pattern Languages of Program Design 3* (Addison-Wesley, 1997) sowie *Designing Object Oriented C++ Aplications Using the Booch Method* (Prentice Hall, 1995).

Danksagung

Die folgenden Personen waren maßgeblich an der Entstehung dieses Buches beteiligt (in keiner bestimmten Reihenfolge):

Chris Guzikowski	Kent Beck
Chris Zahn	Martin Fowler
Matt Heuser	Alistair Cockburn
Jeff Overbey	James O. Coplien
Micah Martin	Tim Conrad
Justin Martin	Richard Lloyd
Carl Hickman	Ken Finder
James Grenning	Kris Iyer (CK)
Simon Brown	Mike Carew
Kevlin Henney	Jerry Fitzpatrick
Jason Gorman	Jim Newkirk
Doug Bradbury	Ed Thelen
Colin Jones	Joe Mabel
Grady Booch	Bill Degnan

Darüber hinaus gab es noch zahlreiche weitere Mitwirkende, deren namentliche Erwähnung den Rahmen dieser kurzen Danksagung jedoch sprengen würde.

Die finale Überarbeitung des Kapitels *Schreiende Architektur* rief mir Jim Weirichs strahlendes Lächeln und sein melodisches Lachen in Erinnerung. Mach's gut, Jim!

Kapitel 1

Was bedeuten »Design« und »Architektur«?

Im Laufe der Jahre sorgte die Verwendung der Begriffe »Design« und »Architektur« immer wieder für Verwirrung. Was ist Design? Was ist Architektur? Und wo liegt der Unterschied?

Eine der Zielsetzungen dieses Buches besteht darin, Ihnen einen Weg durch das verworrene Dickicht der Begriffsauslegungen aufzuzeigen und ein für alle Mal zu definieren, was genau unter Design und Architektur zu verstehen ist. Für den Anfang soll an dieser Stelle zunächst einmal festgehalten werden, dass es eigentlich keinen Unterschied gibt. *Überhaupt keinen Unterschied.*

Das Wort »Architektur« wird häufig im Zusammenhang mit Dingen auf einer übergeordneten Ebene verwendet, die sich von den Details auf einer untergeordneten Ebene unterscheiden. Mit »Design« scheinen dagegen oftmals Strukturen und Entscheidungen auf einer untergeordneten Ebene bezeichnet zu werden. Betrachtet man die Arbeit eines echten Architekten, ist diese Begriffsverwendung jedoch geradezu absurd.

Versetzen Sie sich doch einmal in den Architekten, der Ihr neues Haus entwirft. Besitzt dieses Gebäude eine Architektur? Natürlich! Und worin besteht diese Architektur? Nun, einerseits beschreibt sie die Gestalt des Hauses – sein äußeres Erscheinungsbild, den Aufriss und die Anordnung des Wohnraums bzw. der Räume. Wenn Sie sich jedoch die Planungsentwürfe Ihres Architekten anschauen, werden Sie darin zum anderen auch eine Vielzahl von untergeordneten Details entdecken. So ist zum Beispiel ausgewiesen, wo jede einzelne Steckdose, jeder einzelne Lichtschalter und jede einzelne Lampe platziert werden wird. Sie können sehen, welche Schalter welche Lampen steuern. Ebenso lässt sich feststellen, wo der Kamin errichtet werden wird und an welchen Standorten und in welcher Größe der Warmwasserboiler und die Schmutzwasserpumpe installiert werden. Und darüber hinaus werden Sie auch detaillierte Darstellungen der Wand-, Dach- und Fundamentkonstruktionen vorfinden.

Kurz: Die Pläne geben Auskunft über alle winzigen Details, die sämtliche auf der übergeordneten Ebene getroffenen Entscheidungen stützen. Und es wird deutlich, dass diese untergeordneten Details und die übergeordneten Entscheidungen Teil des Gesamtentwurfs des Hauses sind.

Genauso verhält es sich auch mit dem Softwaredesign. Die untergeordneten (low-level) Details und die übergeordnete (high-level) Struktur sind allesamt Teil desselben Ganzen. Zusammen bilden sie ein durchgängiges Konstrukt, das die Gestalt des Systems definiert. Das eine kann nicht ohne das andere sein. Tatsächlich gibt es keine eindeutige Grenzlinie zwischen diesen beiden Komponenten – es gibt einfach nur ein Kontinuum an Entscheidungen, die sich von der höchsten bis zu den niedrigsten Ebenen erstrecken.

1.1 Das Ziel?

Und die Zielsetzung dieser Entscheidungen? Das Erreichen eines guten Softwaredesigns? Dieses Ziel entspricht nichts Geringerem als meiner »utopischen« Beschreibung:

> *Das Ziel der Softwarearchitektur besteht in der Minimierung der menschlichen Ressourcen, die für das Errichten und die Instandhaltung des benötigten Systems erforderlich sind.*

Die Qualität des Designs bemisst sich schlicht und ergreifend an dem Ausmaß des Aufwands, der erforderlich ist, um den Bedürfnissen der Kunden gerecht zu werden. Bedarf es hierfür eines geringen Aufwands und bleibt dies auch während der gesamten Lebenszeit des Systems so, dann handelt es sich um ein gutes Design – steigt der Aufwand mit jedem neuen Release, taugt das Design nichts. So einfach ist das.

1.2 Fallstudie

Die nachstehende Fallstudie soll hier als Beispiel dienen. Sie weist reale Daten eines echten Unternehmens aus, das jedoch anonym bleiben möchte.

Werfen wir zunächst einen Blick auf den Anstieg des technischen Mitarbeiterstabs im Engineering-Bereich. Sicher werden Sie zustimmen, dass dieser Trend äußerst vielversprechend aussieht. Das in Abbildung 1.1 gezeigte Personalwachstum muss doch ein Indikator für einen bedeutenden Erfolg sein!

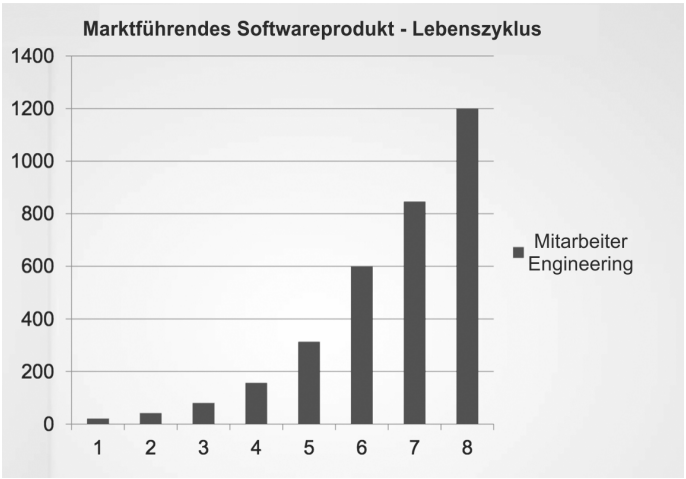

Abb. 1.1: Personalwachstum im Engineering-Bereich
(Abdruck mit freundlicher Genehmigung von Jason Gorman)

Betrachten wir als Nächstes die Produktivität des Unternehmens in demselben Zeitraum, und zwar gemessen an den *Lines of Code*, also der Anzahl der Codezeilen (siehe Abbildung 1.2).

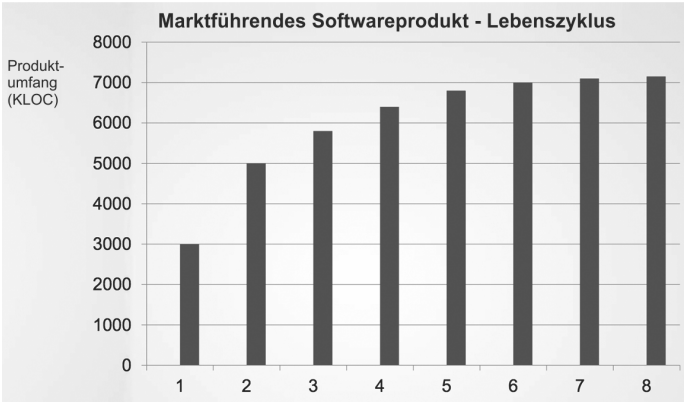

Abb. 1.2: Produktivität über denselben Zeitraum hinweg

Hier läuft ganz offenkundig irgendetwas nicht richtig: Obwohl jedes Release unter Zuhilfenahme einer stetig steigenden Anzahl von Entwicklern bearbeitet wird, scheint der Code lediglich in einem asymptotischen Verhältnis dazu anzuwachsen.

Was nun folgt, ist allerdings eine noch weitaus besorgniserregendere Grafik: Abbildung 1.3 zeigt, wie sich die Kosten pro Codezeile im Zeitverlauf verändert haben.

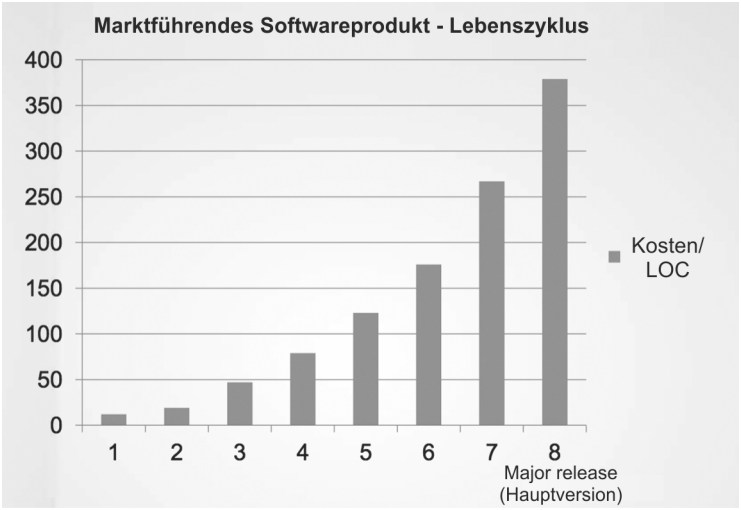

Abb. 1.3: Kosten pro Codezeile im Zeitverlauf

Diese Trendentwicklungen sind definitiv nicht tragbar. Egal, wie profitabel das Unternehmen zum gegenwärtigen Zeitpunkt auch dastehen mag: Verlaufskurven wie diese werden die Gewinne des Geschäftsmodells in katastrophaler Art und Weise belasten und die Firma in die Verlustzone treiben – wenn nicht sogar direkt in die Pleite.

Doch was in aller Welt hat diese bemerkenswerte Veränderung in Bezug auf die Produktivität verursacht? Warum war die Codeerstellung für Release Nummer 8 gleich 40 Mal teurer als für Release Nummer 1?

1.2.1 Die Signatur des Chaos

Was Sie in den vorstehenden Grafiken gesehen haben, könnte man auch als die »Signatur des Chaos« bezeichnen. Wenn Systeme hastig zusammengeschustert werden, wenn die schiere Anzahl der Programmierer der alleinige Antrieb für den zu erzielenden Output ist und wenn nur wenige bis gar keine Überlegungen hinsichtlich der Sauberkeit des Codes oder der Struktur des Designs angestellt werden, dann können Sie gewiss sein, dass Sie den hier demonstrierten eingeschlagenen Weg bis zum bitteren Ende gehen werden.

In Abbildung 1.4 ist zu sehen, wie sich dieser Weg für die Entwickler darstellt: Sie haben mit einer Produktivität von nahezu 100 % angefangen, die dann jedoch mit jedem weiteren Release radikal abfiel. Bereits ab Release Nummer 4 wurde deutlich, dass ihre Produktivität in einer asymptotischen Annäherung gegen null abflachen würde.

Abb. 1.4: Produktivität je Release

Aus Sicht der Entwickler ist das enorm frustrierend, weil sie faktisch alle die ganze Zeit über unvermindert hart an dem Programm weitergearbeitet haben – ohne dass auch nur ein einziger von ihnen seine Anstrengungen verringert hätte.

Und doch, trotz all ihres heroischen Einsatzes, der Überstunden und ihrer Hingabe, bringen sie einfach nicht mehr viel zustande. All ihre Bemühungen entfernen sich zusehends von den Features weg und richten sich nun immer mehr darauf, das entstandene Chaos zu verwalten. Ihre Aufgabe hat sich somit inzwischen dahin gehend verändert, dass sie jetzt das Chaos von einer Stelle zur nächsten verlagern, und wieder zur nächsten und übernächsten, damit sie überhaupt noch in der Lage sind, auch nur ein einziges weiteres mickriges Feature zu ergänzen.

1.2.2 Die Perspektive der Unternehmensleitung

Wenn Sie nun glauben, *das* sei schon übel, dann stellen Sie sich nur einmal vor, wie sich die Sachlage der Führungsebene des Unternehmens darstellt! Abbildung 1.5 zeigt die Entwicklung der monatlichen Gehaltszahlungen für denselben Zeitraum.

Release Nummer 1 wurde bei monatlichen Gehaltszahlungen von wenigen Hunderttausend US-Dollar fertiggestellt. Das zweite Release kostete bereits ein paar Hunderttausend Dollar mehr. Und mit Release Nummer 8 hatten die monatlichen Zahlungen bereits 20 Millionen US-Dollar erreicht, Tendenz steigend!

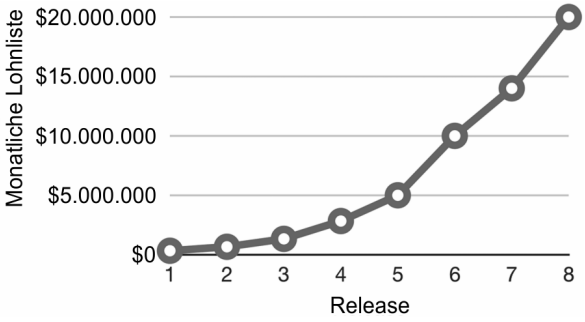

Abb. 1.5: Entwicklung der monatlichen Gehaltszahlungen je Release

Allein dieses Diagramm ist schon erschreckend. Zweifelsohne ist hier irgendetwas Alarmierendes im Busch. Man kann nur hoffen, dass die Erträge die Kosten am Ende übersteigen werden und somit die Ausgaben rechtfertigen – doch egal, wie man diesen Kurvenverlauf auch betrachtet, er gibt in jedem Fall Anlass zur Sorge.

Vergleichen Sie die Kurve in Abbildung 1.5 nun aber mal mit den pro Release geschriebenen Codezeilen in Abbildung 1.2. Mit der Investition von ursprünglich wenigen Hunderttausend Dollar pro Monat wurde eine Menge Funktionalität erzielt – die letzten 20 Millionen Dollar brachten dagegen nahezu nichts! Jeder CFO bzw. Finanzdirektor wird mit einem Blick auf diese beiden grafischen Darstellungen erkennen, dass sofort etwas unternommen werden muss, um die bevorstehende Katastrophe abzuwenden.

Doch was lässt sich dagegen unternehmen? Was ist hier schiefgelaufen? Wodurch wurde dieser unglaubliche Produktivitätsabfall verursacht? Was kann die Managementebene anderes tun, als mit den Füßen aufzustampfen und den Entwicklern ihren Unmut kundzutun?

1.2.3 Was ist schiefgelaufen?

Vor fast 2.600 Jahren erzählte der griechische Dichter Äsop die Fabel von der Schildkröte und dem Hasen. Die Moral dieser Geschichte wurde im Laufe der Zeit auf vielerlei unterschiedliche Weise interpretiert:

- »Eile mit Weile.«
- »Nicht den Schnellen gehört im Wettlauf der Sieg, nicht den Tapferen der Sieg im Kampf.«
- »Blinder Eifer schadet nur.«

Im Prinzip illustriert diese Fabel die Unvernunft der Selbstüberschätzung: Der Hase vertraut dermaßen auf die ihm eigene Schnelligkeit, dass er das Rennen nicht ernst genug nimmt und ein Nickerchen hält, während die Schildkröte unterdessen die Ziellinie überquert.

Moderne Entwickler finden sich in einer ähnlichen Wettbewerbssituation wieder – und stellen eine ganz ähnliche Selbstüberschätzung zur Schau. Allerdings schlafen sie nicht, ganz im Gegenteil: Die Programmierer von heute arbeiten buchstäblich Tag und Nacht. Dennoch: Ein Teil ihres Verstandes befindet sich oftmals tatsächlich im Dämmerschlaf – und zwar der Teil, der im Grunde genommen genau weiß, dass guter, sauberer und ordentlich designter Code den *entscheidenden Unterschied* macht.

Diese Entwickler erliegen einer landläufigen Illusion: »Aufräumen können wir später immer noch, jetzt müssen wir das System aber zuerst mal auf den Markt bringen!« Allerdings wird der Code im Nachhinein natürlich nicht mehr aufgeräumt – ganz einfach weil der Marktdruck niemals nachlässt. Ist die Markteinführung erst einmal vollzogen, hat man sofort eine Horde von Wettbewerbern auf den Fersen, denen man um jeden Preis vorausbleiben muss, indem man im übertragenen Sinne so schnell rennt, wie man nur kann.

Dementsprechend wird den Programmierern zu keinem Zeitpunkt ein Wechsel ihres Arbeitsmodus gelingen. Sie können nicht zurück und den vorhandenen Code säubern, weil sie schon gleich wieder das nächste Feature fertigstellen müssen – und das nächste und das nächste und das nächste. So häuft sich immer mehr Unordnung an, und in der Konsequenz nimmt die Produktivität einen zielstrebigen asymptotischen Verlauf gegen null.

Ebenso wie der Hase der Selbstüberschätzung seiner Schnelligkeit erlag, verfallen auch die Entwickler einer Selbstüberschätzung in Bezug auf ihre Fähigkeit, produktiv zu bleiben. Das Chaos, das sich in den Code einschleicht und ihre Produktivität beeinträchtigt, setzt sich somit unerbittlich fort. Und wenn diesem Prozess nicht Einhalt geboten wird, reduziert er ebendiese Produktivität binnen weniger Monate auf null.

Ein noch gravierenderer Trugschluss, dem die Entwickler erliegen, ist jedoch die Vorstellung, dass ihnen das Schreiben von unordentlichem Code kurzfristig einen Geschwindigkeitsvorteil bringen und alles andere auf lange Sicht nur aufhalten würde. Programmierer, die diesem Irrglauben aufsitzen, demonstrieren nicht nur die Selbstüberschätzung des Hasen in Bezug auf ihre Fähigkeit, den Wechsel vom chaotischen Arbeiten zur zukünftigen Beseitigung der angerichteten Unordnung zu vollziehen, sondern begehen darüber hinaus auch einen simplen faktischen Fehler, denn: Tatsache ist, dass *Unordnung immer für mehr Verzögerungen sorgt als die fortgesetzte Einhaltung sauberer Programmierung*, unabhängig davon, welcher Zeitrahmen zugrunde gelegt wird.

Betrachten Sie in diesem Zusammenhang einmal die Ergebnisse eines bemerkenswerten, von Jason Gorman unternommenen Experiments, das in Abbildung 1.6 veranschaulicht ist. Jason führte seine Untersuchungsreihe über einen Zeitraum von sechs Tagen durch. An jedem dieser Tage stellte er ein einfaches Pro-

gramm fertig, das Integer in römische Zahlen umwandelte. Anschließend ließ er den Code jeweils einen Satz von Funktionstests durchlaufen. Verliefen sie erfolgreich, war seine Arbeit erledigt. Dieser Vorgang nahm pro Tag etwas weniger als 30 Minuten in Anspruch. Am ersten, dritten und fünften Tag wandte Jason eine bekannte Methode für saubere Softwareentwicklung an, die als *Test-Driven Development* (*TDD*, testgetriebene Entwicklung) bezeichnet wird. An den verbleibenden drei Tagen schrieb er den Code hingegen ohne TDD weiter.

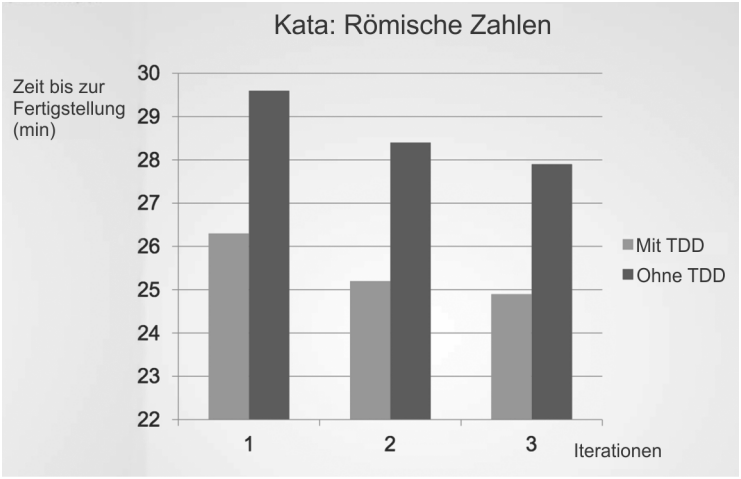

Abb. 1.6: Zeit bis zur Fertigstellung nach Iterationen und Anwendung/Nichtanwendung der TDD-Methode

Auffällig ist in Abbildung 1.6 die Lernkurve: An den späteren Tagen war die Arbeit schneller erledigt als an den vorausgehenden. Beachten Sie auch, dass die Programmierung an den TDD-Tagen um etwa 10 % schneller voranging als an den Tagen ohne Anwendung des TDD-Verfahrens und dass Jason selbst am langsamsten TDD-Tag schneller vorwärtskam als am schnellsten Tag ohne TDD.

Nun mag das Ergebnis dieser Untersuchung so manchen Programmierer in Erstaunen versetzen. Für all jene, die nicht der Selbstüberschätzung des Hasen unterliegen, war es jedoch durchaus zu erwarten – weil sie sich einer einfachen Wahrheit der Softwareentwicklung bewusst sind:

> *Der einzige Weg, schnell voranzukommen, ist, auf die richtige Art und Weise vorzugehen.*

Und das ist auch die Antwort auf das Dilemma der Führungsebene: Der einzige Weg, dem Produktivitätsabfall und dem Kostenanstieg entgegenzuwirken, besteht darin, die Programmierer davon abzubringen, sich die Denkweise des sich selbst überschätzenden Hasen anzueignen, sondern stattdessen anzufangen, Verantwortung für das von ihnen verursachte Chaos zu übernehmen.

Die Entwickler mögen ihrerseits hingegen die Auffassung vertreten, die Antwort bestünde darin, noch mal ganz von vorn anzufangen und das gesamte System neu zu designen – aber damit hätte der Hase dann doch wieder die Oberhand gewonnen, denn: Dieselbe Selbstüberschätzung, die überhaupt erst zu dem angerichteten Chaos geführt hat, würde ihnen an dieser Stelle wiederum vorgaukeln, sie könnten es besser machen, wenn sie nur einen Neustart des Rennens bewirken könnten. Die Realität sieht jedoch weniger rosig aus:

> *Ihre Selbstüberschätzung wird das Redesign in genau dasselbe Chaos stürzen, wie es schon beim ursprünglichen Projektansatz der Fall war.*

1.3 Fazit

Die beste Option für das Entwicklungsteam ist in jedem Fall, seine eigene Selbstüberschätzung zu erkennen, sie abzustellen und stattdessen anzufangen, die Qualität seiner Softwarearchitektur ernster zu nehmen.

Dazu muss man sich natürlich darüber im Klaren sein, was eine gute Softwarearchitektur überhaupt ausmacht. Um ein System zu errichten, dessen Design und Architektur eine Minimierung des zu betreibenden Aufwands und eine Maximierung der Produktivität gewährleisten, müssen Sie wissen, welche Attribute der Systemarchitektur zum Erreichen dieses Ziels führen.

Und genau das ist es, womit sich dieses Buch befasst. Es beschreibt, wie gute, saubere Architekturen[1] und Designs aussehen, damit Softwareentwickler Systeme bauen können, die eine lange und gewinnbringende Lebensdauer erzielen.

1 Um den Lesefluss in deutschen Sätzen zu erleichtern, wird der Begriff »Clean Architecture« in diesem Buch sinngemäß mit »saubere Architektur« übersetzt.

Stichwortverzeichnis

B

C

D

E

F

G

L

M

Q

R

S

T

U

Z