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

Hinweis des Verlages zum Urheberrecht und Digitalen Rechtemanagement (DRM)

Inhaltsverzeichnis

Dieses Buch ist meiner wunderbaren Frau,
meinen vier großartigen Kindern und deren Familien
sowie der Schar meiner fünf Enkelkinder gewidmet
– sie sind die Freude meines Lebens.

Robert C. Martin

Clean Architecture

Das Praxis-Handbuch für professionelles Softwaredesign

Regeln und Paradigmen für effiziente Softwarestrukturen

Übersetzung aus dem Amerikanischen
von Maren Feilen und Knut Lorenzen

Impressum

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

ISBN 978-3-95845-726-3
1. Auflage 2018

www.mitp.de
E-Mail: mitp-verlag@sigloch.de
Telefon: +49 7953 / 7189 - 079
Telefax: +49 7953 / 7189 - 082

Lektorat: Sabine Schulz
Sprachkorrektorat: Petra Heubach-Erdmann
Coverbild: © Vadim Sadovski/ShutterStock
electronic **pub**lication: III-satz, Husby, www.drei-satz.de

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 richtungweisend 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]

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

---

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

[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.

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 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.

---

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

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

Chris Zahn

Matt Heuser

Jeff Overbey

Micah Martin

Justin Martin

Carl Hickman

James Grenning

Simon Brown

Kevlin Henney

Jason Gorman

Doug Bradbury

Colin Jones

Grady Booch

Kent Beck

Martin Fowler

Alistair Cockburn

James O. Coplien

Tim Conrad

Richard Lloyd

Ken Finder

Kris Iyer (CK)

Mike Carew

Jerry Fitzpatrick

Jim Newkirk

Ed Thelen

Joe Mabel

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!

Teil I
Einführung

In diesem Teil:

Um ein Programm zum Laufen zu bringen, bedarf es nicht notwendigerweise Unmengen an Wissen und Fertigkeiten – schon Schüler sind dazu in der Lage. Nicht selten gründen junge Männer und Frauen noch während ihrer College- bzw. Hochschulzeit Milliarden-Dollar-Unternehmen, die lediglich auf ein paar Zeilen PHP oder Ruby basieren. Horden von Nachwuchsprogrammierern in Großraumbüros rund um den Globus rackern sich durch seitenschwere Anforderungsdokumentationen, die in riesigen Issue-Tracking-Systemen (auch »Fallbearbeitungssysteme« genannt) vorgehalten werden, um ihre eigenen Softwareentwicklungen durch schiere unerschütterliche *Willenskraft* zum Funktionieren zu bringen. Der Code, den sie dabei schreiben, mag nicht unbedingt hübsch anzusehen sein, aber er funktioniert, weil es in der Tat nicht allzu schwierig ist, ein Programm – zumindest einmalig – zum Laufen zu bringen.

Eine völlig andere Sache ist es allerdings, das Ganze *richtig* anzufangen. Eine wirklich gute Software auf die Beine zu stellen, das ist ein *schwieriges* Unterfangen. Es erfordert in der Tat ein gewisses Maß an Wissen und Fertigkeiten, das die meisten jungen Programmierer noch nicht erworben haben. Ebenso sind Überlegungen und Einblicke notwendig, die zu erlangen die meisten Softwareentwickler sich nicht die Zeit nehmen. Darüber hinaus erfordert gutes Programmieren auch einiges an Disziplin und Hingabe, wie es sich die meisten von uns nicht mal im Traum vorgestellt hätten. Vor allem aber erfordert es echte Leidenschaft für das Handwerk selbst und den unbedingten Wunsch, dieser Arbeit professionell nachzugehen.

Doch wenn Sie die Software richtig hinbekommen, dann passiert etwas Magisches: Sie brauchen keine Armada von Programmierern, um sie funktionsfähig zu halten. Sie brauchen keine seitenschweren Anforderungsdokumentationen und keine riesigen Issue-Tracking-Systeme. Ebenso wenig brauchen Sie Großraumbüros, und Sie müssen auch nicht an sieben Tagen in der Woche rund um die Uhr programmieren.

Wenn Software in der richtigen Art und Weise entwickelt wird, erfordert ihre Erstellung und Instandhaltung lediglich einen Bruchteil der vorgenannten menschlichen Ressourcen. Vielmehr lassen sich Modifikationen und Anpassungen schnell und einfach umsetzen. Mängel und Fehler treten nur hin und wieder mal in Erscheinung. Der Aufwand ist minimal, und das bei maximaler Funktionalität und Flexibilität.

Zugegeben, diese Vision klingt ein wenig nach Utopie, aber ich weiß, dass es funktioniert – denn ich habe es selbst erlebt. Ich habe an Projekten mitgewirkt, deren Design und Architektur sowohl das Schreiben als auch die Instandhaltung des jeweiligen Systemcodes leichtfallen ließen. Ich habe auch Projekte erlebt, die lediglich einen Bruchteil der ursprünglich angenommenen menschlichen Ressourcen erforderten. Und ich habe an Systemen gearbeitet, die extrem niedrige Fehlerraten aufwiesen. Ich selbst war Zeuge des außergewöhnlichen Effekts, den eine gute Softwarearchitektur auf ein System, ein Projekt und ein Team haben kann. Kurz: Ich war in diesem Gelobten Land.

Aber natürlich sollten Sie sich nicht nur auf mein Wort verlassen. Blicken Sie doch einmal auf Ihre eigenen Erfahrungen zurück: Haben Sie das Gegenteil erlebt? Haben Sie an Systemen gearbeitet, die so ineinander verwoben und derart kompliziert verknüpft waren, dass sich jede noch so simple Modifikation bzw. Anpassung wochenlang hinzog und mit enormen Risiken einherging? Haben Sie die durch schlecht geschriebenen Code und miserables Design verursachten Hemmnisse am eigenen Leib erfahren? Haben Sie erlebt, dass sich das Design der Systeme, an denen Sie gearbeitet haben, in massiver Weise negativ auf die Moral des Teams, das Vertrauen der Kunden und die Geduld der Managementebene ausgewirkt hat? Haben Sie miterlebt, dass Teams, Abteilungen und vielleicht sogar ganze Unternehmen von der armseligen Struktur Ihrer Software zugrunde gerichtet wurden? Mit anderen Worten: Waren Sie schon mal in der Programmierhölle?

Ich für meinen Teil war es – so wie die meisten unserer Zunft. Denn leider kommt es bedeutend häufiger vor, dass man sich durch grauenvolle Softwaredesigns hindurchkämpfen muss, als dass man das große Vergnügen hat, mit einem wirklich guten Design zu arbeiten.

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!

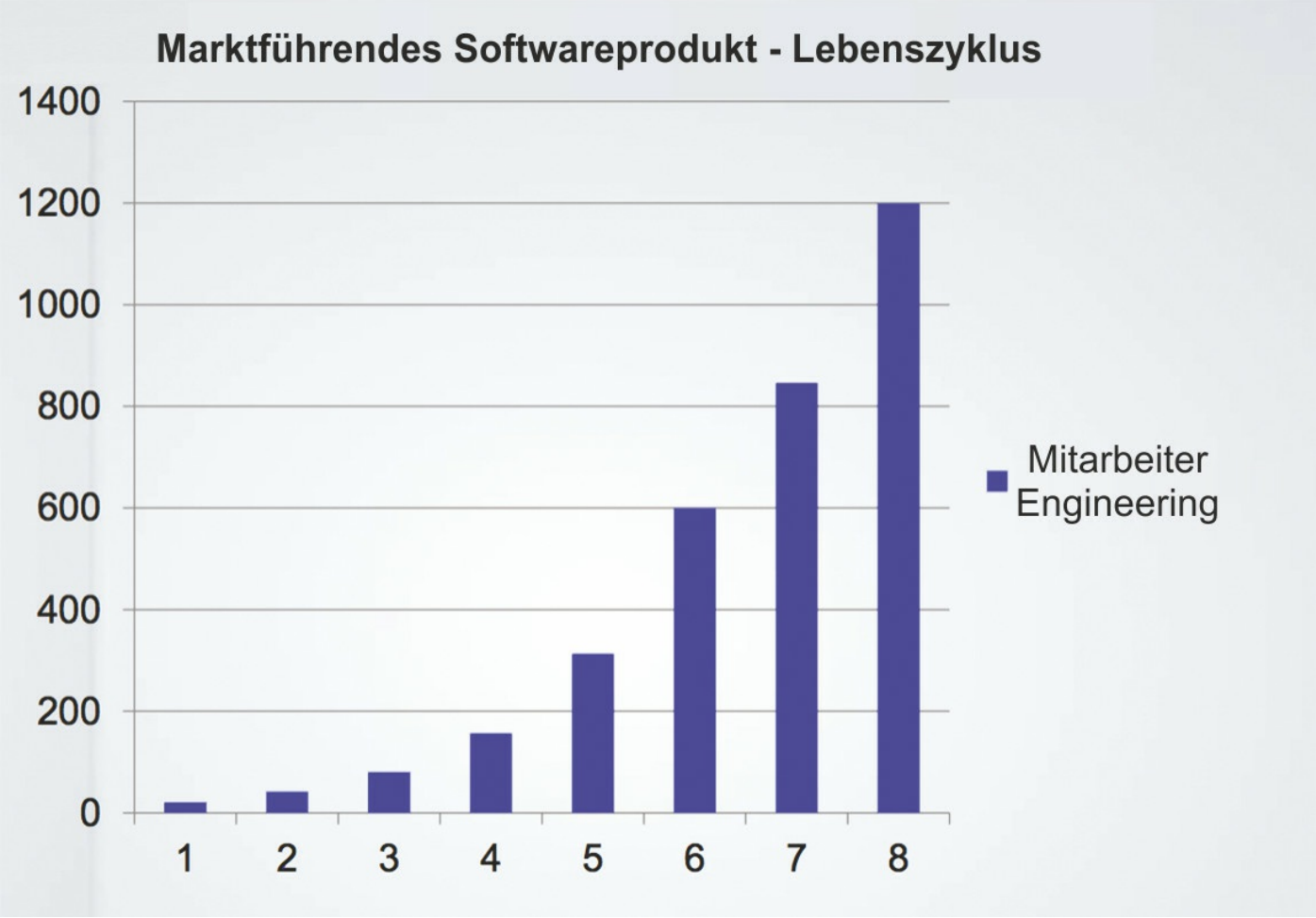


Abb. 1.1: Personalwachstum im Engineering-Bereich (Abdruck mit freundlicher Genehmigung von Jason Gorman)

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).

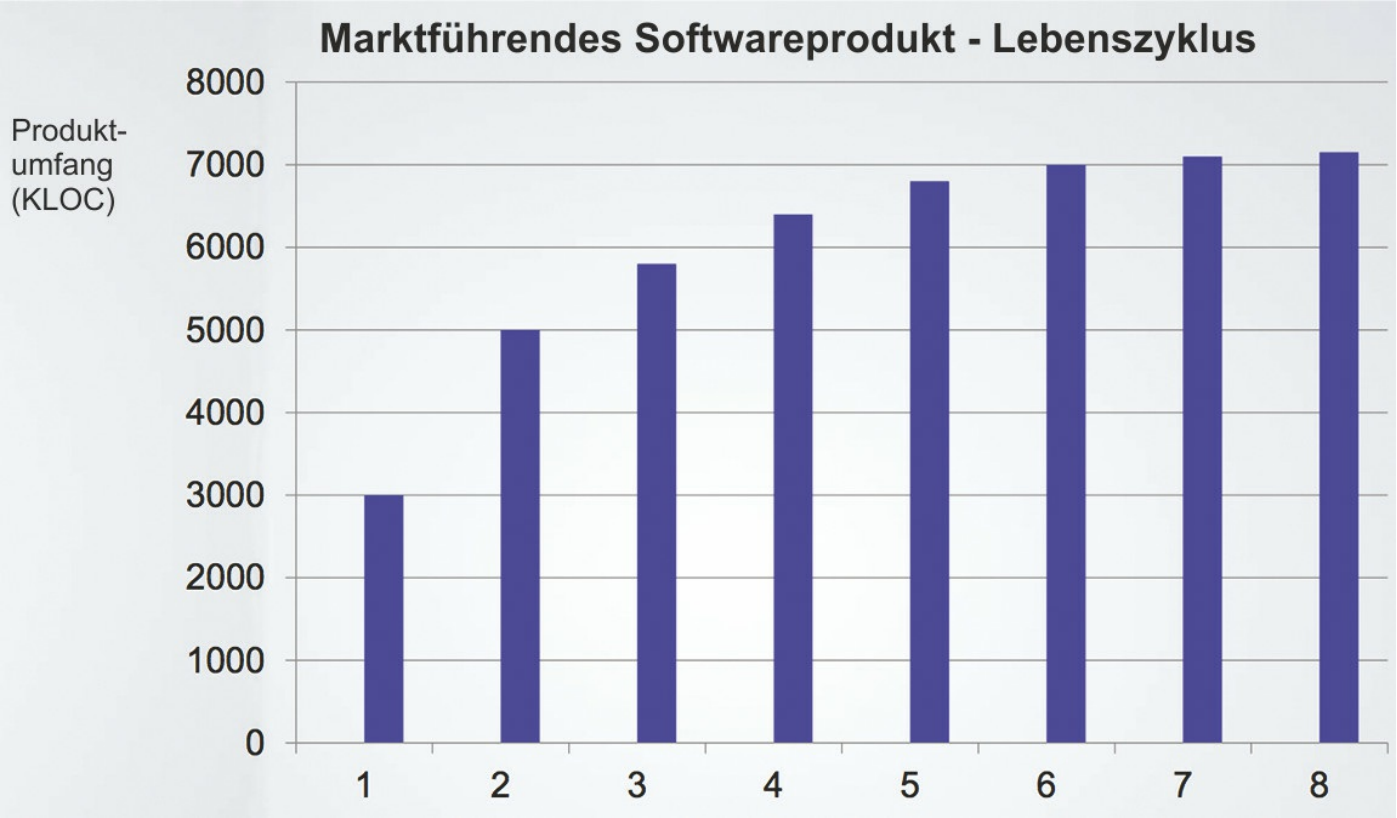


Abb. 1.2: Produktivität über denselben Zeitraum hinweg

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.

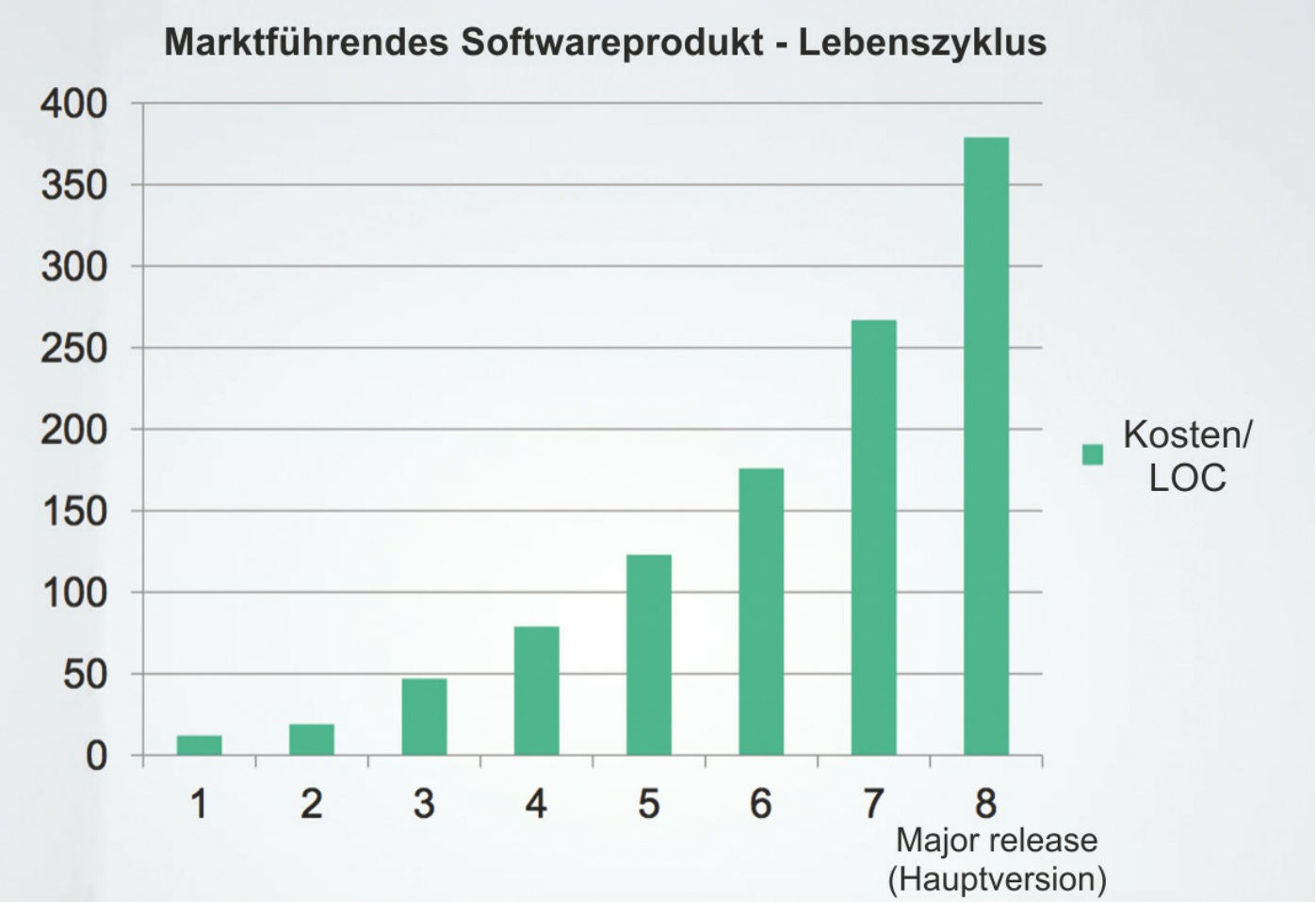


Abb. 1.3: Kosten pro Codezeile im Zeitverlauf

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.

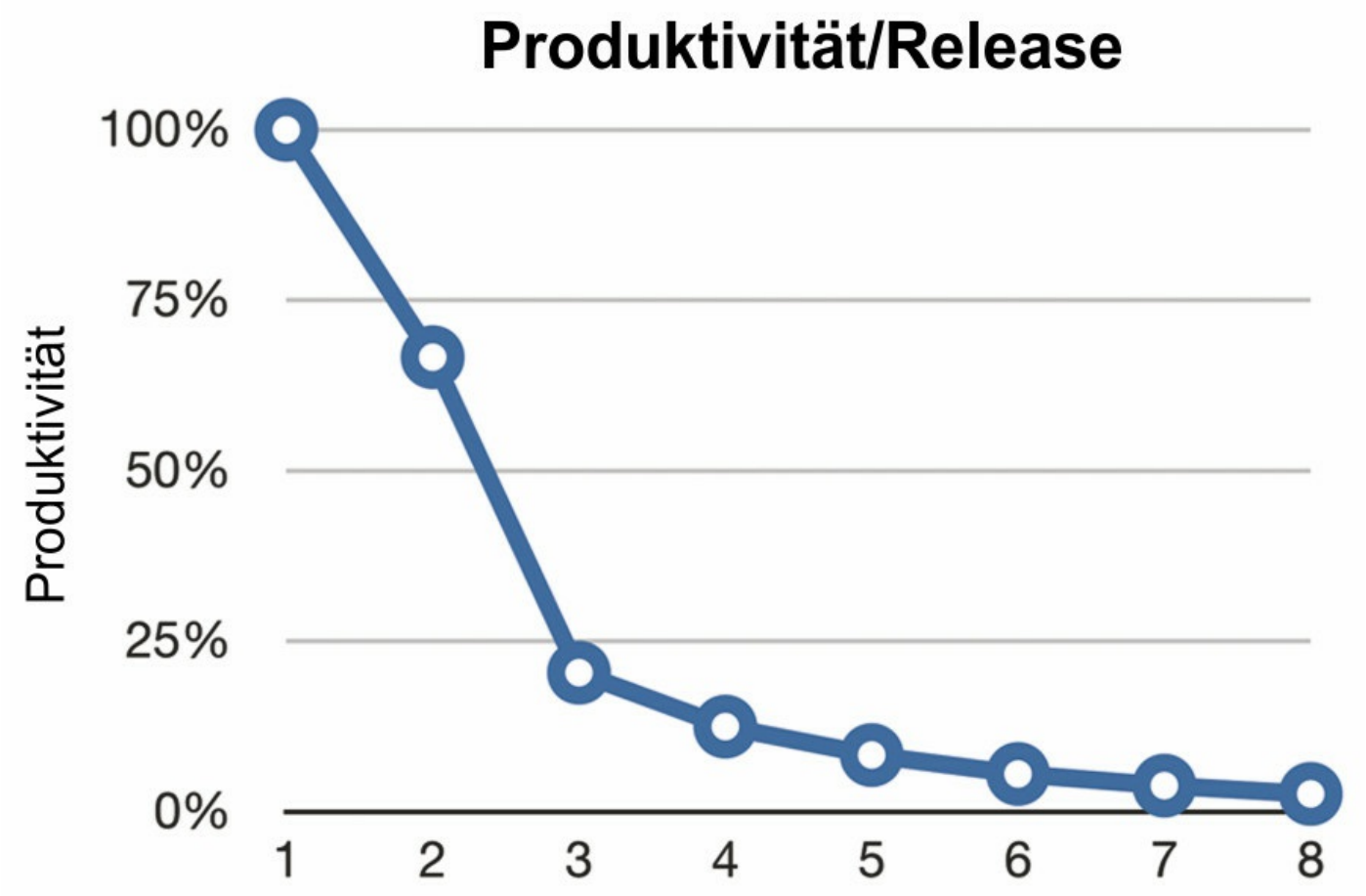


Abb. 1.4: Produktivität je Release

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!

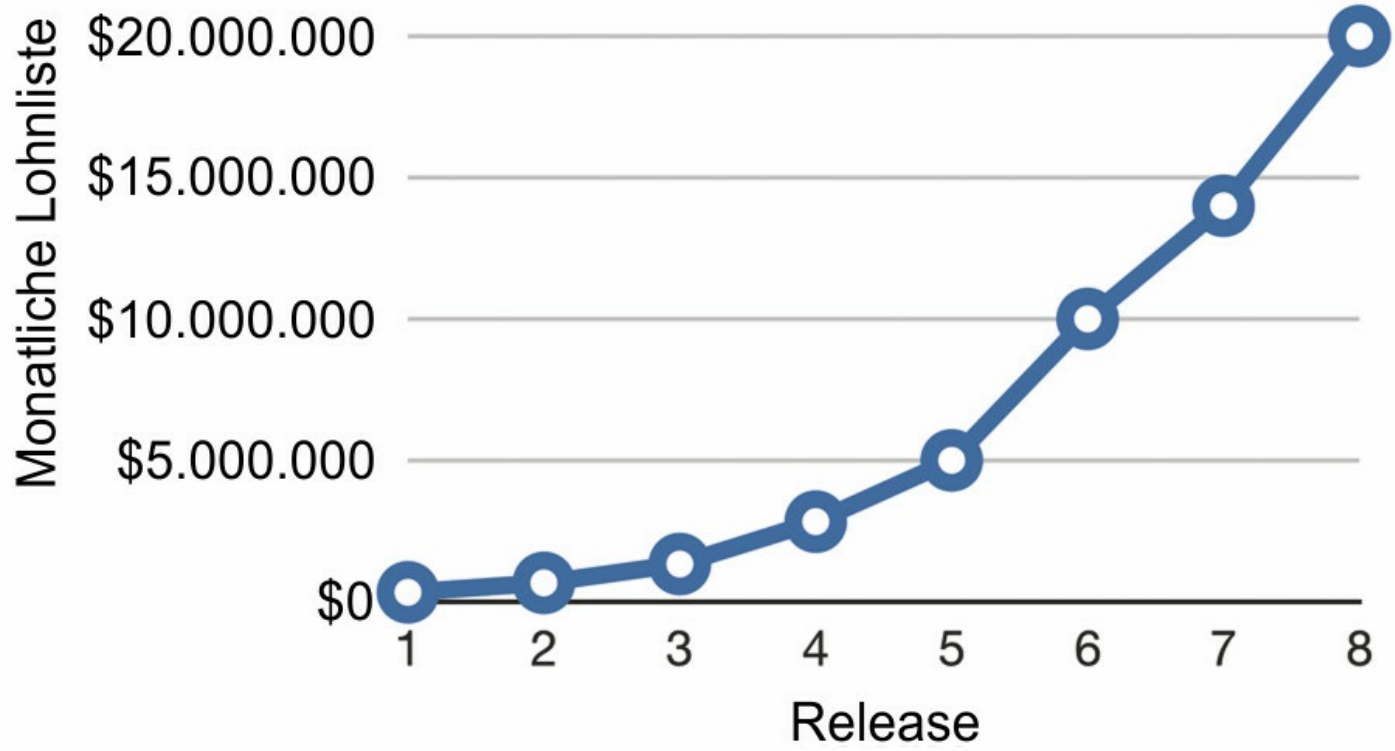


Abb. 1.5: Entwicklung der monatlichen Gehaltszahlungen je Release

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 Programm 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.

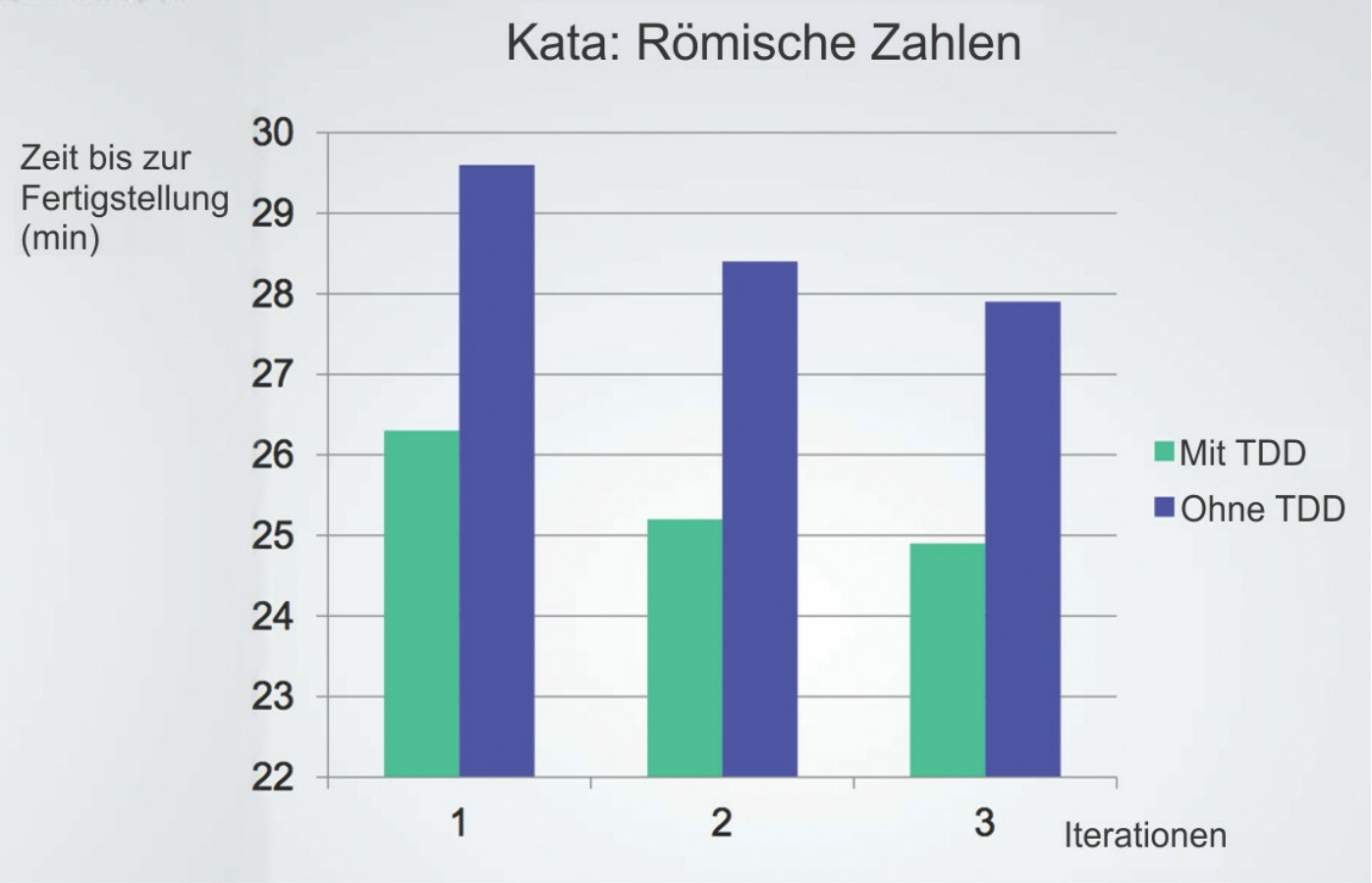


Abb. 1.6: Zeit bis zur Fertigstellung nach Iterationen und Anwendung/Nichtanwendung der TDD-Methode

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.

Kapitel 2
Die Geschichte zweier Werte

Jedes Softwaresystem stellt den Stakeholdern zwei unterschiedliche Werte zur Verfügung: das *Verhalten* und die *Struktur* – und es ist Aufgabe der Softwareentwickler, zu gewährleisten, dass beides auf einem hohen Niveau beibehalten wird. Leider konzentrieren sich Programmierer jedoch häufig nur auf einen dieser Werte, was dann zulasten des jeweils anderen geht. Und schlimmer noch: Oftmals versteifen sie sich zudem auf den weniger relevanten Aspekt, wodurch das Softwaresystem letzten Endes insgesamt wertlos wird.

2.1 Verhalten

Der erste charakteristische Wert einer Software ist ihr *Verhalten*. Programmierer werden von Unternehmen eingestellt, um dafür Sorge zu tragen, dass sich deren Maschinen in einer Art und Weise verhalten, die den Stakeholdern Profite oder Einsparungen sichert. Dazu müssen die Programmierer die Stakeholder zunächst einmal dabei unterstützen, eine funktionale Spezifikation oder entsprechende Anforderungsdokumentationen zu entwickeln. Dann schreiben sie den Code, der es den Maschinen ermöglicht, diese Anforderungen zu erfüllen. Und sollten die Maschinen an irgendeinem Punkt den Anforderungen zuwiderlaufen, nehmen die Softwareentwickler ihre Debugger zur Hand und beseitigen das Problem.

Nun sind allerdings viele Programmierer der Auffassung, dass dies schon alles sei, was ihren Job ausmacht. Sie glauben, sie seien nur dafür zuständig, die Implementierung der Anforderungen aufseiten der Maschine sicherzustellen und auftretende Bugs zu beheben. Doch damit liegen sie leider falsch.

2.2 Architektur

Der zweite charakteristische Wert einer Software steht in unmittelbarem Zusammenhang mit der eigentlichen Bedeutung dieses Begriffs – der sich aus den englischen Wortbestandteilen »soft« und »ware« zusammensetzt, was ins Deutsche übersetzt so viel wie »weiches Produkt« heißt.

Die Software als solches war von Anfang an als »weiches«, sprich »formbares« Produkt gedacht. Sie sollte eine einfache Möglichkeit bieten, das Verhalten von Maschinen zu modifizieren. Hätte die Absicht dahintergestanden, Verhaltensänderungen von Maschinen schwieriger zu gestalten, dann wäre sie von vornherein als »*Hard*ware« bezeichnet worden, also als »hartes, starres« Produkt.

Um ihren Zweck erfüllen zu können, muss Software »weich« bzw. leicht anzupassen sein. Wenn die Stakeholder eine Änderung eines Features beabsichtigen, dann sollte sich die entsprechende Modifikation einfach und unkompliziert bewerkstelligen lassen. Der Aufwand für die Durchführung einer solchen Anpassung sollte proportional zu ihrem Ausmaß sein, nicht jedoch zu ihrer *Form*.

Häufig ist ebendiese Diskrepanz zwischen Ausmaß und Form der Modifikation der ausschlaggebende Faktor für den Anstieg der Kosten in der Softwareentwicklung. Sie ist auch die Ursache für einen in Relation zum Umfang der erforderlichen Änderungen unverhältnismäßigen Kostenanstieg. Und sie ist der Grund dafür, warum die Entwicklungsarbeit im ersten Jahr deutlich weniger kostenintensiv ist als im zweiten und warum das zweite Jahr wiederum deutlich günstiger als das dritte Jahr ausfällt.

Aus Sicht der Stakeholder liefern die Programmierer lediglich eine Reihe von

Modifikationen in einem in etwa vergleichbaren Umfang. Aus Sicht der Softwareentwickler übergeben ihnen die Stakeholder dagegen ein ganzes Bündel voller Puzzleteile, die sie in ein Puzzlebild einpassen müssen, das unaufhörlich komplexer wird. Jedem neuen Anpassungswunsch ist schwieriger nachzukommen als dem vorherigen – weil die Form des Systems nicht der Form der Anforderung entspricht.

Der Begriff »Form« wird hier in unkonventioneller, aber metaphorisch durchaus passender Weise verwendet, denn: Nicht selten stellt sich die Situation für die Softwareentwickler so dar, als würde von ihnen erwartet, quadratische Bauklötzchen in runde Löcher hineinzuzwängen.

Das eigentliche Problem ist jedoch die Architektur des Systems: Je mehr diese Architektur eine Form gegenüber einer anderen begünstigt, desto wahrscheinlicher ist es, dass sich neue Features immer schwieriger in diese Struktur einpassen lassen. Und deshalb sollten Architekturen ebenso formagnostisch (die Formgebung sollte für sie keine Rolle spielen) wie praktisch sein.

2.3 Der größere Wert

Funktionalität oder Architektur? Welches von beidem liefert den größeren Wert? Ist es eher von Belang, dass das Softwaresystem arbeitet, oder muss es in erster Linie einfach anzupassen sein?

Wenn man die Managementebene eines Unternehmens dazu befragt, erhält man oftmals die Antwort, dass der Lauffähigkeit des Systems eine größere Bedeutung beigemessen wird. Und so übernehmen die Softwareentwickler dieselbe Haltung. *Doch das ist die falsche Einstellung zu der Sache*. Dies lässt sich mithilfe einer einfachen logischen Gegenüberstellung der Extreme belegen:

- *Wenn Sie mir ein Programm geben, das perfekt läuft, aber keine Anpassungen zulässt, dann wird es bei veränderten Anforderungen nicht mehr funktionieren, und ich werde auch nicht in der Lage sein, es funktionsfähig zu machen. Letztendlich wird das Programm daher nutzlos werden.*

- *Wenn Sie mir ein Programm geben, das nicht läuft, sich aber leicht anpassen lässt, dann kann ich es zum Laufen bringen und auch dann noch funktionsfähig halten, wenn sich die Anforderungen ändern. Letztendlich wird das Programm also kontinuierlich nützlich bleiben.*

Nun finden Sie diese Argumentation vielleicht nicht überzeugend – immerhin gibt es so etwas wie ein unveränderliches Softwareprogramm nicht. Es gibt allerdings Systeme, die es *praktisch* unmöglich machen, Anpassungen vorzunehmen, weil der

Aufwand für die durchzuführenden Modifikationen die dadurch zu erzielenden Vorteile zunichtemacht. Und faktisch stoßen viele Systeme mit einigen ihrer Features und Konfigurationen an diese Grenze.

Fragt man die Manager eines Unternehmens, ob sie in der Lage sein wollen, Anpassungen vorzunehmen, werden sie dies natürlich bejahen – und gleich anschließend einschränkend hinzufügen, dass der aktuellen Funktionalität jedoch größere Bedeutung zukommt als jedweder zukünftigen Flexibilität. Und wenn ebendiese Manager Sie zu einem späteren Zeitpunkt bitten, eine Modifikation an der Software vorzunehmen und Ihre dazugehörige Kalkulation exorbitant hoch ausfällt, dann werden genau dieselben Leute Sie wutentbrannt dafür verantwortlich machen, dass das System an einem Punkt angelangt ist, an dem Anpassungen nicht mehr praktikabel sind.

2.4 Das Eisenhower-Prinzip

Betrachten Sie dazu die von dem US-amerikanischen Präsidenten Dwight D. Eisenhower ins Leben gerufene Priorisierungsmethode, die die Aspekte der *Wichtigkeit* und der *Dringlichkeit* gegenüberstellt (siehe Abbildung 2.1). Eisenhower selbst sagte dazu:

> *Es gibt zwei Arten von Problemen, die dringenden und die wichtigen. Die dringenden sind selten wichtig und die wichtigen sind selten dringend.*[1]

Abb. 2.1: Das Eisenhower-Prinzip

In diesem alten Ausspruch steckt viel Wahres. Tatsächlich sind die vordringlichen Dinge nur selten auch wirklich wichtig, und andererseits sind die wichtigen Dinge nur

selten von großer Dringlichkeit.

Der erste Wert der Software – das *Verhalten* – ist dringend, aber nicht immer auch besonders wichtig.

Der zweite Wert der Software – die *Architektur* – ist wichtig, aber zu keiner Zeit besonders dringend.

Selbstverständlich sind manche Dinge dennoch gleichermaßen dringend wie wichtig – und andere ebenso nicht dringend wie nicht wichtig. Unterm Strich können wir auf der Grundlage dieser Erkenntnisse somit vier Priorisierungskategorien festlegen:

1. Dringend und wichtig
2. Nicht dringend und wichtig
3. Dringend und nicht wichtig
4. Nicht dringend und nicht wichtig

Beachten Sie, dass die *Architektur* von Code – die wichtigen Dinge – den ersten beiden Kategorien dieser Liste zuzuordnen ist, wohingegen das *Verhalten* von Code in die erste und *dritte* Kategorie gehört.

Der Fehler, den Manager und Programmierer oft begehen, besteht darin, bestimmte Dinge, die eigentlich in die dritte Kategorie fallen, in die erste Kategorie zu erheben. Mit anderen Worten: Sie nehmen eine falsche Einteilung der Features vor, die dringend, aber nicht wichtig sind, und derjenigen, die wirklich dringend und wichtig sind. Und diese Fehleinschätzung führt in der Folge dazu, dass die wichtige Architektur des Systems in der Priorisierung hinter die unwichtigen Features des Systems zurückfällt.

Das Dilemma für Softwareentwickler ist, dass die Geschäftsleitung nicht die Möglichkeit hat, die wahre Bedeutung der Architektur zu bemessen. *Das ist die Aufgabe der Softwareentwickler, dafür wurden sie eingestellt.* Dementsprechend trägt auch das Entwicklungsteam die Verantwortung dafür, die Wichtigkeit der Architektur gegenüber der Dringlichkeit der Features festzustellen.

2.5 Der Kampf für die Architektur

Dieser Verantwortung nachzukommen, heißt auch, in gewisser Weise in den Kampf zu ziehen – nun ja, im Sinne einer nicht körperlichen Auseinandersetzung natürlich. Ehrlich gestanden, läuft es eigentlich immer darauf hinaus, denn: Das

Entwicklungsteam muss grundsätzlich um die Durchsetzung dessen ringen, was es für das Unternehmen für das Beste hält ... und das Gleiche gilt auch für die Teams der Geschäftsbereiche Management, Marketing, Sales und Operations. *Es ist ein ständiger Kampf.*

Effiziente Softwareentwicklungsteams stellen sich dieser Konfrontation ohne Umschweife. Auf Augenhöhe gehen sie unablässig in die Auseinandersetzung mit allen anderen Stakeholdern. Vergessen Sie nicht: *Als Softwareentwickler sind Sie selbst ebenfalls ein Stakeholder.* Sie haben einen eigenen Anteil an der Software, den Sie sichern müssen. Das gehört zu Ihrer Rolle und zu Ihren Kernaufgaben. Und nicht zuletzt ist diese Form von Einsatz für die Sache auch einer der wesentlichen Gründe, warum Sie überhaupt erst ins Team geholt wurden.

Eine noch größere Bedeutung kommt dieser Herausforderung zu, wenn Sie Softwarearchitekt sind. Softwarearchitekten sind per Definition weitaus mehr mit der Struktur des Systems befasst als mit seinen Features und Funktionen. Sie erschaffen eine Architektur, die es ermöglicht, dass die vorgesehenen Features und Funktionen leicht entwickelt, geändert und erweitert werden können.

Denken Sie immer daran: Wenn die Architektur hintangestellt wird, hat dies unweigerlich einen stetigen Kostenanstieg in der Entwicklungsarbeit zur Folge – und am Ende werden Modifikationen in Teilbereichen des Systems oder sogar dem gesamten System praktisch unmöglich durchzuführen sein. Wenn es so weit kommt, bedeutet dies im Umkehrschluss, dass das Entwicklungsteam nicht hart genug für das gekämpft hat, von dem es genau wusste, dass es unentbehrlich war.

---

[1] Aus einer Rede an der Northwestern University im Jahr 1954.

Teil II
Die ersten Bausteine setzen: Programmierparadigmen

In diesem Teil:

Der zentrale Ausgangspunkt einer jeden Softwarearchitektur ist der Code, also der Quelltext eines Systems – und so beginnt unser Diskurs zum Thema Architektur mit einem Rückblick auf die Erkenntnisse, die wir seit den ersten jemals geschriebenen Programmzeilen über Code im Allgemeinen gewonnen haben.

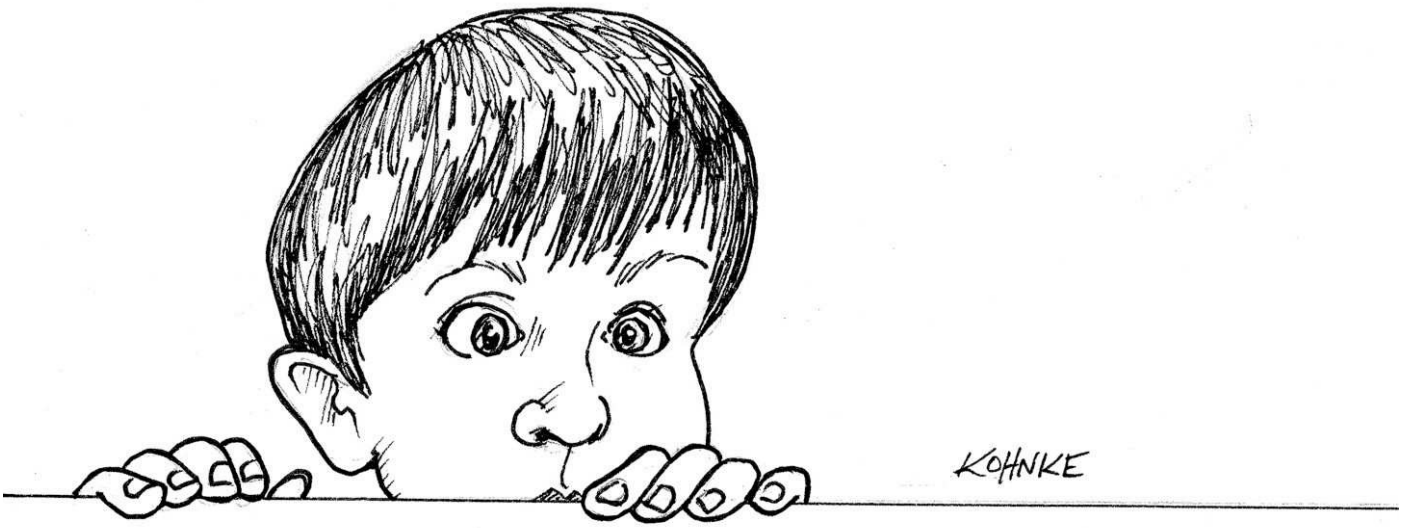
Im Jahr 1938 legte Alan Turing das Fundament für das, was die Computerprogrammierung der Zukunft werden sollte. Er war zwar nicht der erste Mensch, der eine programmierbare Maschine auch als solche begreifen konnte, aber er war der Erste, der verstand, dass Programme einfach nur Daten waren. Bis 1945 schrieb Turing echte Programme auf echten Computern in realem Code, der auch als solcher erkennbar war (wenn man genau genug hinsah). Diese Programme nutzten Schleifen, Verzweigungen, Zuweisungen, Subroutinen, Stacks (Stapelspeicher) und andere vertraute Strukturen. Turings Sprache war binär – eine *Maschinensprache*.

Seit dieser Zeit haben im Bereich der Programmierung eine Reihe von Revolutionen stattgefunden. Eine davon, die uns allen sehr gegenwärtig ist, betraf die Programmiersprachen. Als Erstes kamen in den späten 1940er-Jahren die *Assembler* auf. Diese »Sprachen« befreiten die Softwareentwickler von der Bürde, ihre Programme in Binärcode übersetzen zu müssen. Im Jahr 1951 erfand Grace Hopper

dann den ersten Compiler namens *A0*. Sie war es auch, die den Begriff *Compiler* prägte. *Fortran* wurde im Jahr 1953 (in dem Jahr nach meiner Geburt) entwickelt. Und was danach folgte, war eine nicht enden wollende Flut von neuen Programmiersprachen – *COBOL, PL/1, SNOBOL, C, Pascal, C++, Java* und endlos so weiter.

Eine andere, vielleicht noch bedeutsamere Revolution war die Programmierung gemäß *Paradigmen*. Hierbei handelt es sich um Programmierstile, die relativ unabhängig von den Sprachen sind. Ein Paradigma gibt vor, welche Programmierstrukturen zu verwenden sind und wann. Bis heute wurden drei solcher Paradigmen etabliert – und aus Gründen, die später noch näher erläutert werden, ist es auch eher unwahrscheinlich, dass noch weitere folgen werden.

Kapitel 3
Die Paradigmen



Die drei in diesem Kapitel beschriebenen Paradigmen werden als *strukturierte Programmierung*, *objektorientierte Programmierung* und *funktionale Programmierung* bezeichnet.

3.1 Strukturierte Programmierung

Das erste allgemein anerkannte (wenn auch nicht das zuerst entdeckte) Paradigma, die *strukturierte Programmierung*, wurde 1968 von Edsger Wybe Dijkstra aufgestellt. Er führte den Beweis, dass die Verwendung von uneingeschränkten absoluten Sprunganweisungen (`goto`-Anweisungen) der Programmstruktur schadet. Wie in den folgenden Kapiteln noch genauer ausgeführt wird, ersetzte er besagte Sprunganweisungen durch geläufigere Kontrollstrukturen wie `if/then/else` und `do/while/until`.

Das Paradigma der strukturierten Programmierung lässt sich wie folgt zusammenfassen:

> *Die strukturierte Programmierung unterwirft die direkte Kontrollübertragung einer konsequenten Disziplin.*

3.2 Objektorientierte Programmierung

Das zweite adaptierte Paradigma war bereits zwei Jahre zuvor, im Jahr 1966, von Ole Johan Dahl und Kristen Nygaard entdeckt worden. Die beiden Programmierer erkannten, dass der *Stack Frame*, der Stapelrahmen für Funktionsaufrufe, in ALGOL in den *Heap* (abstrakte Datenstruktur) verschoben werden konnte, wodurch eine fortgesetzte Existenz der von einer Funktion deklarierten lokalen Variablen auch lange nach der Rückgabe der Funktion möglich wurde. Die Funktion wurde zum *Konstruktor* einer Klasse, die lokalen Variablen wurden zu *Instanzvariablen*, und die verzweigten Funktionen wurden zu *Methoden*. Und das wiederum führte unweigerlich zur Entdeckung der *Polymorphie* durch die disziplinierte Verwendung von *Funktionszeigern*.

Zusammenfassend lässt sich die objektorientierte Programmierung folgendermaßen beschreiben:

> *Die objektorientierte Programmierung unterwirft die indirekte Kontrollübertragung einer konsequenten Disziplin.*

3.3 Funktionale Programmierung

Das dritte Paradigma, das erst seit einigen Jahren in der Programmierwelt Anwendung findet, wurde interessanterweise als Allererstes entdeckt – noch vor der Computerprogrammierung selbst. Die *funktionale Programmierung* ist das unmittelbare Ergebnis der Arbeit von Alonzo Church, der 1936 demselben mathematischen Problem auf der Spur war, das zur gleichen Zeit auch Alan Turing umtrieb, und das λ-*(Lambda-)Kalkül* einführte. Dieses Funktionsmodell bildet die Grundlage der Programmiersprache LISP, die 1958 von John McCarthy entwickelt wurde. Ein grundlegendes Konzept des Lambda-Kalküls ist die Unveränderlichkeit – d.h. die Annahme, dass sich die Werte von Symbolen nicht verändern. Effektiv bedeutet das, dass eine funktionale Sprache kein Zuweisungskommando besitzt. Die meisten funktionalen Sprachen verfügen zwar über einige Methoden, mit denen sich der Wert einer Variablen ändern lässt, allerdings funktioniert das nur unter Anwendung einer sehr strengen Disziplin.

Kurz gefasst lässt sich das Paradigma der funktionalen Programmierung wie folgt beschreiben:

> *Die funktionale Programmierung unterwirft die Zuweisung konsequenter Disziplin.*

3.4 Denkanstöße

Beachten Sie das Muster, das ich hier ganz bewusst für die Vorstellung dieser drei Programmierparadigmen gewählt habe: Sie alle *berauben* die Softwareentwickler einiger Fähigkeiten, ohne ihnen jedoch neue Fähigkeiten zuzugestehen. Jedes einzelne dieser Paradigmen unterwirft die Programmierarbeit einer zusätzlichen Disziplinierung, die von ihrer Intention her *negativ* behaftet ist. Sie geben uns also vor, *was nicht gemacht* werden darf, statt uns aufzuzeigen, *was gemacht* werden soll.

Aus einer anderen Perspektive betrachtet, kann man somit festhalten, dass uns jedes einzelne Paradigma etwas untersagt. Zusammengenommen berauben uns die drei Paradigmen der `goto`-Anweisungen, der Funktionszeiger und der Zuweisung. Bleibt dann überhaupt noch etwas übrig, was uns vorenthalten werden könnte?

Das ist eher unwahrscheinlich, und dementsprechend werden diese drei Paradigmen vermutlich auch in Zukunft die einzigen bleiben, mit denen wir es zu tun bekommen. Ein weiterer Indikator dafür, dass keine neuen Paradigmen mehr folgen werden, ist, dass sie alle innerhalb der ersten zehn Jahre der Programmier-Ära zwischen 1958 und 1968 entdeckt wurden und in den vielen nachfolgenden Dekaden kein weiteres mehr ergänzt wurde.

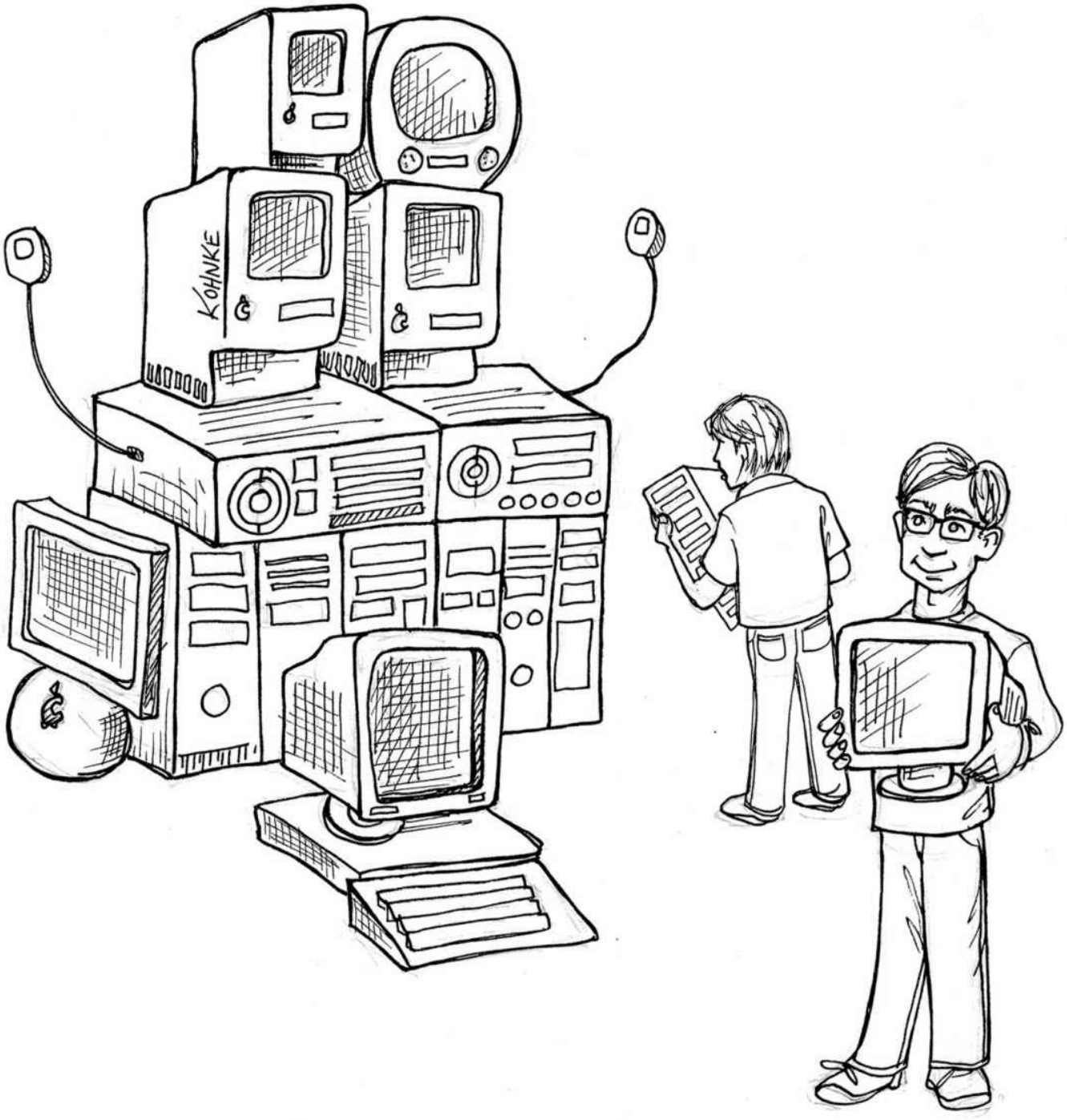
3.5 Fazit

Was aber hat diese Geschichtsstunde nun eigentlich mit der Softwarearchitektur zu tun? Die Antwort auf diese Frage lautet: einfach alles. Wir machen uns die Polymorphie als Mechanismus zur Überschreitung architektonischer Grenzen zunutze. Wir wenden die funktionale Programmierung an, um den Standort von und den Zugriff auf Daten einer konsequenten Disziplin zu unterwerfen. Und wir nutzen die strukturierte Programmierung als algorithmisches Fundament unserer Module.

Es ist kaum zu übersehen, wie gut diese drei Aspekte mit den drei großen Anliegen der Softwarearchitektur übereinstimmen: Funktionalität, Komponententeilung und Datenmanagement.

Kapitel 4

Strukturierte Programmierung



Edsger Wybe Dijkstra wurde 1930 in Rotterdam, Niederlande, geboren. Er überlebte sowohl die Bombardierung seiner Heimatstadt als auch die darauf folgende deutsche Besatzung der Niederlande im Zweiten Weltkrieg und machte 1948 seinen Hochschulabschluss in den Fächern Mathematik, Physik, Chemie und Biologie mit Bestnoten. Im März 1952 nahm er im Alter von 21 Jahren (etwa neun Monate bevor ich geboren wurde) eine Stellung am »Mathematical Center of Amsterdam« (Zentrum für Mathematik und Informatik) an – und wurde damit zum ersten Programmierer der Niederlande überhaupt.

Nachdem er bereits drei Jahre als Programmierer praktiziert und gleichzeitig auch noch sein Studium fortgesetzt hatte, merkte Dijkstra, dass er den aus seiner Sicht

größeren intellektuellen Herausforderungen der Programmierung doch mehr zugetan war als denen der theoretischen Physik – und so entschied er sich 1955 für eine langfristige Karriere im Bereich der Programmierung.

Im Jahr 1957 heiratete Dijkstra Maria Debets. Damals musste man für Eheschließungen in den Niederlanden seinen Beruf angeben, allerdings erkannten die niederländischen Behörden die Berufsbezeichnung »Programmierer« nicht an – denn dieses Betätigungsfeld war zu jener Zeit noch völlig unbekannt. Um sie zufriedenzustellen, gab Dijkstra daher an, er sei »theoretischer Physiker«.

Im Rahmen seiner Überlegungen bezüglich der Frage, ob er tatsächlich eine Karriere als Programmierer einschlagen sollte, wandte sich Dijkstra an seinen damaligen Chef, Adriaan van Wijngaarden. Die Programmierung war zu dieser Zeit noch weit von einer Anerkennung als Fachbereich oder Wissenschaft entfernt, und deshalb fürchtete Dijkstra, er könne möglicherweise nicht ernst genommen werden. Van Wijngaarden gelang es jedoch, seine Bedenken zu zerstreuen, denn er war überzeugt, dass Dijkstra das Zeug hatte, einer der Pioniere sein zu können, die die Softwareentwicklung zu einer Wissenschaft erheben würden.

Den Anfang nahm Dijkstras Karriere im Zeitalter der Vakuumröhren – als Computer noch riesig, anfällig, langsam und unzuverlässig waren sowie (nach heutigen Standards) lediglich über ein extrem eingeschränktes Funktionsspektrum verfügten. In jenen Jahren wurden Programme noch in Binärcode geschrieben oder in sehr rudimentärem Assembler. Die Dateneingabe erfolgte in physischer Form mittels Papierstreifen oder Lochkarten. Und die Arbeitsprozesse für die Bearbeitung, Kompilierung und das Testen dauerten viele Stunden – wenn nicht sogar Tage.

In dieser primitiven Umgebung standen ihm alle Möglichkeiten für seine großen Entdeckungen offen.

4.1 Die Beweisführung

Das Problem, das Dijkstra schon sehr früh erkannte war, dass das Programmieren an sich *schwierig* ist – und dass Softwareentwickler diese Kunst nicht allzu gut meisterten. Ein Programm jedweder Komplexität enthält einfach zu viele Details, als dass sie ein menschliches Gehirn ohne Hilfestellung verarbeiten könnte. Wird auch nur ein kleines Detail übersehen, resultiert das bereits in Programmen, die zwar zu funktionieren *scheinen*, aber dann doch auf überraschende Weise fehlschlagen.

Dijkstras Lösung bestand darin, das mathematische Konzept der *Beweisführung* anzuwenden. Er hatte die Vision, eine euklidische Hierarchie von Postulaten, Theoremen, Korollaren und Lemmata zu errichten. Nach seiner Vorstellung könnten

Programmierer diese Hierarchie ebenso nutzen, wie es Mathematiker tun, soll heißen: Die Softwareentwickler würden bewährte Strukturen verwenden und sie dann mit dem Code verbinden, deren Korrektheit sie selbst beweisen würden.

Natürlich war sich Dijkstra bewusst, dass er die Technik zum Schreiben grundlegender Beweisführungen für simple Algorithmen zunächst darstellen müsste, um sein Vorhaben voranzubringen – und das stellte ihn vor eine ziemliche Herausforderung.

Während seiner Recherche entdeckte er, dass bestimmte Verwendungen von `goto`-Anweisungen verhindern, dass Module rekursiv in immer kleinere Einheiten (*Units*) zerlegt werden und somit auch die Nutzung des für vernünftige Beweise erforderlichen Teile-und-herrsche-Ansatzes unterbunden wird.

Bei anderen Verwendungen von `goto` trat dieses Problem hingegen nicht in Erscheinung. Dijkstra erkannte, dass diese »guten« Verwendungsarten von `goto` im Grunde genommen einfachen Kontrollstrukturen für Selektionen und Iterationen wie `if/then/else` und `do/while` entsprachen. Module, in denen ausschließlich diese Arten von Kontrollstrukturen zum Einsatz kamen, *konnten* also rekursiv in beweisbare Einheiten unterteilt werden.

Dijkstra wusste, dass solche Strukturen in Kombination mit einer sequenziellen Ausführung etwas Besonderes waren. Sie waren bereits zwei Jahre zuvor von Böhm und Jacopini entdeckt worden, die den Nachweis geführt hatten, dass alle Programme unter Zugrundelegung von lediglich drei Strukturformen aufgebaut werden können: *Sequenz*, *Selektion* und *Iteration*.

Und diese Erkenntnis war in der Tat bemerkenswert: Genau diejenigen Kontrollstrukturen, die ein Modul beweisbar machten, entsprachen dem Minimum an Kontrollstrukturen, aus denen alle Programme erstellt werden können. Damit war das Paradigma der strukturierten Programmierung geboren.

In der Folge demonstrierte Dijkstra, dass sequenzielle Anweisungen durch einfache *Enumeration* (Aufzählung) als korrekt bewiesen werden konnten. Dabei wurden die Eingaben der *Sequenz* unter Anwendung eines mathematischen Verfahrens zu den Ausgaben der Sequenz verfolgt. Dieser Ansatz unterschied sich somit nicht von jedem herkömmlichen mathematischen Beweis.

Die *Selektion* realisierte Dijkstra durch erneute Anwendung der Enumeration. Jeder Pfad durch die Selektion wurde aufgezählt, und wenn schließlich beide Pfade entsprechende mathematische Ergebnisse erzeugten, war der Beweis sicher erbracht.

Bei der *Iteration* verhielt es sich etwas anders. Um eine Iteration als korrekt zu beweisen, musste er das *Induktionsverfahren* anwenden. Dijkstra erbrachte zunächst den Fallbeweis für 1 durch Enumeration. Als Nächstes bewies er – wieder durch Enumeration – den Fall, dass, wenn *N* als korrekt angenommen wurde, *N+1* ebenfalls

korrekt war. Und auch den Beweis für die Anfangs- und Endkriterien der Iteration erreichte er durch Enumeration.

Solche Beweisführungen waren aufwendig und komplex – aber es waren gültige Beweise. Und damit schien die Vorstellung, dass sich eine euklidische Hierarchie von Theoremen herstellen ließ, in greifbare Nähe gerückt zu sein.

4.2 Eine »schädliche« Proklamation

1968 schickte Dijkstra einen Brief an den Redakteur der *CACM* (**C**ommunications of the **A**ssociation for **C**omputing **M**achinery), der in der März-Ausgabe des Magazins veröffentlicht wurde. Die Überschrift des Artikels lautete *»Go To Statement Considered Harmful«* (zu Deutsch etwa »Goto-Anweisung als abträglich betrachtet«). Diese Abhandlung skizzierte Dijkstras Position zu den drei Kontrollstrukturen.

Und die Programmierwelt fing Feuer. Damals gab es noch kein Internet, dementsprechend konnten die Leute keine verächtlichen Memes oder sonstige Kommentare über Dijkstra posten und sie konnten ihn auch keinem Online-Shitstorm aussetzen. Aber sie konnten Briefe an die Redakteure vieler Zeitschriften jener Zeit schreiben – und das taten sie auch.

Diese Zuschriften waren nicht unbedingt alle freundlich. Einige hatten einen äußerst negativen Tenor, in anderen wurde dagegen eine nachdrückliche Unterstützung für Dijkstras Position deutlich. Und so begann eine Auseinandersetzung, die etwa zehn Jahre andauern sollte.

Irgendwann ebbte der Streit dann aber schließlich doch ab. Aus einem einfachen Grund: Dijkstra hatte gewonnen. Mit der Fortentwicklung der Computersprachen rückte die `goto`-Anweisung immer weiter in den Hintergrund, bis sie am Ende ganz verschwand. In den meisten modernen Programmiersprachen ist sie überhaupt nicht vorgesehen – und natürlich war sie in LISP von vornherein nicht enthalten.

Heutzutage sind wir alle »strukturierte Programmierer«, wenn auch nicht unbedingt freiwillig: Die von uns verwendeten Sprachen stellen uns die Option der undisziplinierten direkten Kontrollübertragung schlicht gar nicht erst zur Verfügung.

Der ein oder andere mag nun auf die benannten `break`-Anweisungen in Java oder Ausnahmen als `goto`-Analogien verweisen. Faktisch entsprechen diese Strukturen aber nicht den vollkommen uneingeschränkten Kontrollübertragungen, die früher in älteren Sprachen wie Fortran oder COBOL möglich waren. Tatsache ist vielmehr, dass selbst Sprachen, die das Schlüsselwort `goto` nach wie vor unterstützen, den Zielbereich oftmals auf die aktuelle Funktion einschränken.

4.3 Funktionale Dekomposition

Die strukturierte Programmierung ermöglicht die rekursive Dekomposition von Modulen in beweisbare Einheiten, was wiederum bedeutet, dass die Module funktional zerlegt werden können. Somit ist es möglich, eine umfassende Problemstellung in übergeordnete Funktionen zu gliedern und anschließend jede dieser Funktionen immer weiter in untergeordnete Funktionen aufzulösen. Zudem kann jede auf diese Art zerlegte Funktion mithilfe der eingeschränkten Kontrollstrukturen der strukturierten Programmierung dargestellt werden.

Darauf aufbauend erfreuten sich Fachgebiete wie die strukturierte Analyse und das strukturierte Design in den späten 1970er-Jahren und den 1980er-Jahren immer größerer Beliebtheit und wurden von Männern wie Ed Yourdon, Larry Constantine, Tom DeMarco und Meilir Page-Jones propagiert und weithin bekannt gemacht. Indem die Programmierer diesen Lehren folgten, waren sie nunmehr in der Lage, große geplante Systeme in Module und Komponenten zu gliedern, die sich dann weiter in winzige beweisbare Funktionen zerlegen ließen.

4.4 Keine formalen Beweise

Doch die Beweise blieben aus. Die euklidische Hierarchie der Theoreme wurde nie realisiert. Und die Programmierer erkannten im Großen und Ganzen auch niemals die Vorzüge, die sich durch den mühsamen Prozess ergeben konnten, jede einzelne kleine Funktion formell zu beweisen. Schließlich verblasste Dijkstras Traum und wurde letztendlich zu Grabe getragen. Nur wenige der heutigen Softwareentwickler teilen seine Auffassung, dass formale Beweise tatsächlich einen geeigneten Weg für die Erzeugung qualitativ hochwertiger Software darstellen.

Aber natürlich sind formale mathematische Beweise im euklidischen Stil nicht die einzige Strategie, um zu belegen, dass etwas korrekt ist. Eine weitere höchst erfolgreiche Vorgehensweise ist die *wissenschaftliche Methode*.

4.5 Wissenschaft als Rettung

Die Wissenschaft unterscheidet sich insofern fundamental von der Mathematik, als dass wissenschaftliche Theorien und Gesetze nicht als korrekt bewiesen werden können. Ich kann Ihnen beispielsweise nicht beweisen, dass das Zweite Newton’sche Gesetz der Bewegung, *F=ma*, oder das Newton’sche Gravitationsgesetz, *F=Gm1m2/r2*, korrekt sind. Zwar lassen sich diese Gesetze demonstrieren und

Messungen durchführen, die zeigen, dass sie auf viele Dezimalstellen korrekt sind, aber sie können nicht im mathematischen Sinne bewiesen werden. Egal, wie viele Experimente man durchführt oder wie viele empirische Beweise auch gesammelt werden, es besteht immer noch die Möglichkeit, dass irgendein Experiment die Gesetze der Bewegung und Gravitation widerlegt.

Das ist das Wesen der wissenschaftlichen Theorien und Gesetze: Sie sind *widerlegbar*, aber nicht beweisbar.

Und doch verwetten wir tagtäglich unser Leben darauf. Jedes Mal, wenn Sie in Ihr Auto steigen, überantworten Sie Ihr Leben der Annahme, dass *F=ma* eine verlässliche Definition der Art und Weise darstellt, wie die Welt funktioniert. Und mit jedem Schritt, den Sie gehen, vertrauen Sie Ihr körperliches Wohlergehen und Ihre Sicherheit der Annahme an, dass *F=Gm1m2/r2* korrekt und zutreffend ist.

Die Wissenschaft basiert nicht auf der Grundlage einer dahin gehenden Beweisführung, dass Annahmen wahr sind, sondern dass *Annahmen als falsch belegt* werden. Und diejenigen Annahmen, die wir trotz aller Bemühungen nicht als falsch belegen können, betrachten wir als korrekt und für unsere Zwecke wahr genug.

Aber natürlich sind nicht alle Annahmen beweisbar. Die Aussage »Dies ist eine Lüge« ist weder wahr noch falsch – vielmehr repräsentiert sie eins der simpelsten Beispiele für eine Annahme, die nicht zu beweisen ist.

Schlussendlich können wir festhalten, dass die Mathematik die Disziplin ist, in der beweisbare Aussagen als korrekt und wahr bestätigt werden. Im Gegensatz dazu ist die Wissenschaft die Disziplin, bei der beweisbare Annahmen als falsch belegt werden.

4.6 Tests

Dijkstra hat einmal gesagt: »Durch Testen kann man stets nur die Anwesenheit, nicht aber die Abwesenheit von Bugs beweisen.« Mit anderen Worten: Ein Programm kann im Rahmen eines Tests als unkorrekt bestätigt werden, aber nicht als korrekt. Alles, was in hinreichendem Maße durchgeführte Tests zu beweisen vermögen, ist, dass sie uns gestatten, ein Programm als für unsere Zwecke korrekt genug zu erachten.

Die Tragweite dieser Tatsache ist erstaunlich: Die Softwareentwicklung ist kein mathematisches Bestreben, obwohl sie mathematische Konstrukte zu manipulieren scheint. Stattdessen ähnelt sie eher einer Wissenschaft: Wir belegen ihre Korrektheit, indem wir es trotz aller Bemühungen nicht schaffen, ihre Unkorrektheit zu beweisen.

Solche Beweisführungen für Unkorrektheit können nur auf beweisbare Programme angewendet werden. Ein Programm, das nicht beweisbar ist – zum Beispiel aufgrund

der uneingeschränkten Verwendung von `goto` –, kann unabhängig davon, wie viele Tests damit durchgeführt wurden, nicht als korrekt angesehen werden.

Die strukturierte Programmierung zwingt uns, ein Programm rekursiv in einen Satz kleiner beweisbarer Funktionen zu zerlegen. Anschließend können Tests durchgeführt werden, um zu versuchen, den Beweis dafür zu erbringen, dass diese kleinen beweisbaren Funktionen nicht korrekt sind. Kann die Unkorrektheit im Rahmen der Tests nicht erwiesen werden, dann betrachten wir die Funktionen als für unsere Zwecke korrekt genug.

4.7 Fazit

Es ist diese Fähigkeit, widerlegbare Programmiereinheiten zu erzeugen, die die strukturierte Programmierung heute so wertvoll macht. Sie ist der Grund dafür, dass moderne Programmiersprachen typischerweise keine uneingeschränkten `goto`-Anweisungen unterstützen. Und auf der architektonischen Ebene ist sie darüber hinaus auch der Grund, warum die *funktionale Dekomposition* nach wie vor zu unseren Best Practices gehört.

Auf jeder Ebene, von der kleinsten Funktion bis hin zur größten Komponente, ähnelt die Softwareentwicklung einer Wissenschaft – und wird daher durch Widerlegbarkeit angetrieben. Softwarearchitekten sind bestrebt, Module, Komponenten und Services zu definieren, die testfähig und leicht zu widerlegen sind. Und um dies zu erreichen, setzen sie restriktive Disziplinen ähnlich der strukturierten Programmierung ein, wenn auch auf einem sehr viel höheren Niveau.

In den folgenden Kapiteln werden wir diese restriktiven Disziplinen im Einzelnen genauer untersuchen.

Kapitel 5
Objektorientierte Programmierung

Wie Sie im Folgenden noch sehen werden, bildet eine gute Architektur die Grundlage für das Verständnis und die Anwendung der Prinzipien des *objektorientierten Designs* (kurz auch *OO* für **O**bjekt**o**rientierung). Aber was genau ist eine Objektorientierung eigentlich?

Eine viel zitierte Antwort auf diese Frage ist »Die Kombination von Daten und Funktionalität« – allerdings handelt es sich hierbei lediglich um eine

unzufriedenstellende Erklärung, weil sie unterstellt, dass sich `o.f()` irgendwie von `f(o)` unterscheidet. Das ist jedoch absurd, denn: Schon lange bevor Dahl und Nygaard 1966 den *Stack Frame* in den *Heap* verschoben und damit die Objektorientierung konzipierten, haben Programmierer Datenstrukturen in Funktionen überführt.

Eine weitere geläufige Antwort auf diese Frage lautet: »Eine Möglichkeit, die reale Welt zu modellieren.« Das ist aber bestenfalls eine ausweichende Erklärung. Was soll denn mit der »Modellierung der realen Welt« eigentlich gemeint sein? Und warum sollten wir etwas Derartiges tun wollen? Vielleicht soll damit einfach nur ausgedrückt werden, dass die Objektorientierung Software verständlicher macht, weil sie einen engeren Bezug zur realen Welt herstellt – doch auch das ist eine eher ausweichende und zu unspezifische Definition. Sie erklärt uns immer noch nicht, was Objektorientierung ist.

Manche greifen zur Beschreibung der Objektorientierung auf die folgenden drei magischen Worte zurück: *Datenkapselung*, *Vererbung* und *Polymorphie*. Gemeint ist damit, dass die Objektorientierung die richtige Beimischung dieser drei Komponenten darstellt oder zumindest dass eine objektorientierte Sprache diese drei Konzepte unterstützen muss.

Untersuchen wir das im Folgenden einmal etwas genauer.

5.1 Datenkapselung?

Der Grund, warum die Datenkapselung, kurz einfach *Kapselung* genannt, als Teilkonzept der Definition der Objektorientierung betrachtet wird, ist die Tatsache, dass objektorientierte Sprachen eine einfache und effiziente Kapselung von Daten und Funktionalität ermöglichen. Und das führt wiederum dazu, dass eine Art Grenzlinie um einen zusammenhängenden Satz von Daten und Funktionen gezogen werden kann. Außerhalb dieser Linie bleiben die Daten verborgen, sodass nur einige der Funktionen bekannt sind. Wie dieses Konzept funktioniert, lässt sich am Beispiel von privaten Datenmembern und öffentlichen Memberfunktionen einer Klasse betrachten.

Allerdings stellt dieses Verfahren kein einzigartiges Merkmal für die Objektorientierung dar, denn bereits in C war eine perfekte Kapselung möglich. Schauen Sie sich dazu einmal dieses einfache C-Programm an:

```
point.h
struct Point;
struct Point* makePoint(double x, double y);
double distance (struct Point *p1, struct Point *p2);

point.c
```

```
#include "point.h"
#include <stdlib.h>
#include <math.h>
struct Point {
   double x,y;
};

struct Point* makepoint(double x, double y) {
   struct Point* p = malloc(sizeof(struct Point));
   p->x = x;
   p->y = y;
   return p;
}

double distance(struct Point* p1, struct Point* p2) {
   double dx = p1->x - p2->x;
   double dy = p1->y - p2->y;
   return sqrt(dx*dx+dy*dy);
}
```

Die User von `point.h` haben keinerlei Zugriff auf die Members von `struct Point`. Sie können die Funktion `makePoint()` ebenso aufrufen wie die Funktion `distance()`, aber sie haben absolut keine Kenntnis von der Implementierung der `Point`-Datenstruktur oder von den Funktionen.

Somit handelt sich hier um eine perfekte Datenkapselung – und zwar in einer nicht objektorientierten Sprache. C-Programmierer greifen von jeher auf diese Vorgehensweise zurück: Sie deklarieren Datenstrukturen und Funktionen in Headerdateien und binden sie dann in Implementierungsdateien ein. Ihre User haben also zu keinem Zeitpunkt Zugriff auf die Elemente in diesen Implementierungsdateien.

Mit der Einführung der Objektorientierung in Form der Programmiersprache C++ wurde die perfekte Datenkapselung von C jedoch durchbrochen.

Aus technischen Gründen[1] macht es der C++-Compiler erforderlich, dass die Membervariablen einer Klasse in der Headerdatei eben dieser Klasse deklariert werden. Dementsprechend sieht das vorgenannte `Point`-Programm nun so aus:

```
point.h
class Point {
public:
   Point(double x, double y);
   double distance(const Point& p) const;

private:
   double x;
   double y;
};

point.cc
```

```
#include "point.h"
#include <math.h>

Point::Point(double x, double y)
: x(x), y(y)
{}

double Point::distance(const Point& p) const {
   double dx = x-p.x;
   double dy = y-p.y;
   return sqrt(dx*dx + dy*dy);
}
```

Die Clients der Headerdatei `point.h` haben Kenntnis von den Membervariablen `x` und `y`! Der Compiler verweigert ihnen zwar den Zugriff darauf, dennoch weiß der Client von ihrer Existenz. Wenn also beispielsweise die Namen dieser Members geändert werden, muss die `point.cc`-Datei neu kompiliert werden – die Kapselung wird durchbrochen.

Die Methode, um die Datenkapselung teilweise wieder instand zu setzen, besteht darin, die Schlüsselwörter `public`, `private` und `protected` in die Sprache aufzunehmen. Dazu ist allerdings ein *Hack* erforderlich, der sich aus der technischen Notwendigkeit ergibt, dass der Compiler diese Variablen in der Headerdatei sehen können muss.

In Java und C# wurde die Separierung von Header und Implementierung komplett abgeschafft, was eine weitere Beeinträchtigung der Kapselung zur Folge hatte. In diesen Sprachen ist es unmöglich, die Deklaration und die Definition einer Klasse voneinander zu trennen.

Aus diesen Gründen ist es auch nur schwer zu akzeptieren, dass die Objektorientierung von einer soliden Kapselung abhängig ist. Tatsächlich unterstützen viele objektorientierte Sprachen[2] lediglich eine minimale bis gar keine erzwungene Datenkapselung.

Die Objektorientierung basiert sicherlich nicht auf der Vorstellung, Programmierer seien diszipliniert genug, gekapselte Daten *nicht* zu umgehen – trotzdem haben die Programmiersprachen, die sich selbst als objektorientiert einstufen, die einst perfekte Kapselung, der wir uns in C erfreuen durften, letztlich nur geschwächt.

5.2 Vererbung?

Wenn die objektorientierten Sprachen nun schon keine verbesserte Datenkapselung bieten, dann aber doch bestimmt Vererbungsmethoden, oder?

Nun ja, so ungefähr. Die Vererbung wird im Grunde genommen einfach durch die Neudeklaration einer Gruppe von Variablen und Funktionen innerhalb eines umschließenden *Scopes* (Anwendungsbereich) realisiert – das konnten C-Programmierer[3] auch schon lange vor dem Aufkommen der objektorientierten Sprachen manuell umsetzen.

Werfen Sie nun einmal einen Blick auf diese Ergänzung unseres ursprünglichen `point.h`-C-Programms:

```
namedPoint.h
struct NamedPoint;

struct NamedPoint* makeNamedPoint(double x, double y, char* name);
void setName(struct NamedPoint* np, char* name);
char* getName(struct NamedPoint* np);

namedPoint.c
#include "namedPoint.h"
#include <stdlib.h>

struct NamedPoint {
   double x,y;
   char* name;
};

struct NamedPoint* makeNamedPoint(double x, double y, char* name) {
   struct NamedPoint* p = malloc(sizeof(struct   NamedPoint));
   p->x = x;
   p->y = y;
   p->name = name;
   return p;
}

void setName(struct NamedPoint* np, char* name) {
   np->name = name;
}

char* getName(struct NamedPoint* np) {
   return np->name;
}

main.c
#include "point.h"
#include "namedPoint.h"
#include <stdio.h>

int main(int ac, char** av) {
   struct NamedPoint* origin = makeNamedPoint(0.0, 0.0, "origin");
   struct NamedPoint* upperRight = makeNamedPoint
      (1.0, 1.0, "upperRight");
   printf("distance=%f\n",
      distance(
               (struct Point*) origin,
```

```
                (struct Point*) upperRight));
}
```

Wenn Sie sich das `main`-Programm genauer anschauen, werden Sie feststellen, dass sich die `NamedPoint`-Datenstruktur verhält, als sei sie von der `Point`-Datenstruktur abgeleitet. Dies liegt darin begründet, dass die Reihenfolge der ersten beiden Felder in `NamedPoint` dieselbe wie bei `Point` ist. Oder anders ausgedrückt: `NamedPoint` kann als `Point` maskiert werden, weil `NamedPoint` eine reine Untermenge von `Point` ist und die Anordnung der Members der von `Point` entspricht.

Dieser Trick war vor dem Aufkommen der Objektorientierung gängige Programmierpraxis[4] – tatsächlich ist die Einfachvererbung in C++ auf genau diese Weise implementiert.

Nun könnte man behaupten, dass bereits lange vor der Erfindung der objektorientierten Sprachen eine Form von Vererbung zur Verfügung gestanden hätte – das entspricht aber nicht ganz der Wahrheit: Es gab zwar einen passenden Trick, der war jedoch nicht annähernd so praktisch wie eine echte Vererbungsmethode. Außerdem ist die Mehrfachvererbung auf diese Art erheblich schwieriger zu bewerkstelligen.

Beachten Sie auch, dass man in `main.c` gezwungen ist, die `NamedPoint`-Argumente auf `Point` zu casten. In einer echten objektorientierten Sprache würde ein solches Upcasting implizit erfolgen.

Die objektorientierten Programmiersprachen haben also zwar keine komplett neue Funktionalität mit sich gebracht, das Maskieren von Datenstrukturen gestaltete sich mit ihrer Hilfe aber sehr viel komfortabler.

Zusammenfassend lässt sich somit an dieser Stelle festhalten: Im Bereich der Datenkapselung kann die Objektorientierung nicht punkten und für die Vererbung kann man ihr allenfalls einen halben Punkt zugestehen – insgesamt also bislang keine allzu tolle Wertung.

Aber es ist ja noch ein weiteres Attribut zu berücksichtigen.

5.3 Polymorphie

Gab es bereits vor dem Aufkommen der objektorientierten Sprachen polymorphisches Verhalten in der Programmierung? Aber natürlich. Betrachten Sie dazu folgendes einfache, in C geschriebene `copy`-Programm.

```
#include <stdio.h>
```

```
void copy() {
   int c;
   while ((c=getchar()) != EOF)
      putchar(c);
}
```

Die Funktion `getchar()` liest von `STDIN`. Doch um was für ein Gerät handelt es sich hierbei? Die `putchar()`-Funktion schreibt hingegen an `STDOUT`. Und welches Gerät ist das nun wieder? Beide hier erwähnten Funktionen sind *polymorph* – ihr Verhalten ist vom jeweiligen Typ von `STDIN` und `STDOUT` abhängig.

Es verhält sich in etwa so, als seien `STDIN` und `STDOUT` Java-artige Schnittstellen, die Implementierungen für jedes Gerät besitzen. Aber natürlich gibt es in dem Beispielprogramm in C keine Schnittstellen – wie also wird der Aufruf von `getchar()` schließlich an den Gerätetreiber übermittelt, der das Zeichen ausliest?

Die Antwort auf diese Frage ist ziemlich einfach. Das Betriebssystem UNIX bedingt, dass jeder I/O-Gerätetreiber (*Input/Output*, Ein-/Ausgabe) fünf Standardfunktionen bereitstellt:[5] `open`, `close`, `read`, `write` und `seek`. Die Signaturen dieser Funktionen müssen für jeden I/O-Treiber identisch sein.

Die `FILE`-Datenstruktur enthält fünf Zeiger auf Funktionen. In unserem Beispiel könnte das so aussehen:

```
struct FILE {
   void (*open)(char* name, int mode);
   void (*close)();
   int (*read)();
   void (*write)(char);
   void (*seek)(long index, int mode);
};
```

Der I/O-Treiber für die Konsole definiert diese Funktionen und lädt eine `FILE`-Datenstruktur mit ihren Adressen hoch, etwa wie folgt:

```
#include "file.h"
void open(char* name, int mode) {/*...*/}
void close() {/*...*/};
int read() {int c;/*...*/ return c;}
void write(char c) {/*...*/}
void seek(long index, int mode) {/*...*/}
struct FILE console = {open, close, read, write, seek};
```

Wenn `STDIN` nun als ein `FILE*` definiert ist und auf die Datenstruktur der Konsole

zeigt, dann könnte `getchar()` auf folgende Art implementiert werden:

```
extern struct FILE* STDIN;

int getchar() {
   return STDIN->read();
}
```

Anders ausgedrückt ruft `getchar()` einfach die Funktion auf, auf die der Lesezeiger der `FILE`-Datenstruktur zeigt, auf den wiederum `STDIN` zeigt.

Dieser einfache Trick bildet die Grundlage für jegliche Polymorphie in der Objektorientierung. Zum Beispiel besitzt in C++ jede virtuelle Funktion innerhalb einer Klasse einen Zeiger in eine Tabelle namens `vtable`. Alle Aufrufe von virtuellen Funktionen durchlaufen diese Tabelle. Konstruktoren von Ableitungen (Derivaten) laden einfach ihre Versionen dieser Funktionen in die `vtable` des erzeugten Objekts.

Unterm Strich bedeutet Polymorphie die Anwendung von Zeigern auf Funktionen. Schon seit der erstmaligen Implementierung der Von-Neumann-Architekturen in den späten 1940er-Jahren nutzen Programmierer Zeiger auf Funktionen, um polymorphes Verhalten zu erzielen. Mit anderen Worten: Die Objektorientierung hat hier im Prinzip nichts Neues gebracht.

Na ja, so ganz stimmt das allerdings nun auch wieder nicht: Die objektorientierten Sprachen mögen uns die Polymorphie zwar nicht beschert haben, aber sie haben sie um einiges sicherer und bequemer gemacht.

Das Problem bei der expliziten Verwendung von Zeigern auf Funktionen zur Erzeugung polymorphen Verhaltens ist, dass solche Zeiger *riskant* sind. Eine derartige Zeigernutzung unterliegt einem Satz manueller Konventionen. So müssen Sie etwa die Konvention zur Initialisierung der Zeiger beachten. Das Gleiche gilt für die Konvention, um alle Ihre Funktionen mithilfe dieser Zeiger aufzurufen. Sollte irgendein an dem Code beteiligter Programmierer die entsprechenden Konventionen versehentlich nicht berücksichtigen, kann es teuflisch schwierig werden, den daraus resultierenden Bug aufzuspüren und zu beseitigen.

Objektorientierte Sprachen schaffen diese Konventionen ebenso wie die damit einhergehenden Risiken ab. Der Einsatz einer objektorientierten Sprache macht die Polymorphie belanglos. Und dieser Umstand bietet ein enormes Potenzial, von dem gestandene C-Programmierer nur träumen konnten. Auf dieser Grundlage können wir festhalten, dass die Objektorientierung die indirekte Kontrollübertragung einer gewissen Disziplin unterwirft.

5.3.1 Die Macht der Polymorphie

Was aber ist denn nun eigentlich so toll an der Polymorphie? Werfen Sie noch mal einen Blick auf das `copy`-Beispielprogramm, um ihre Vorzüge besser bewerten zu können. Was passiert mit diesem Programm, wenn ein neues I/O-Gerät erstellt wird? Angenommen, Sie wollten das `copy`-Programm benutzen, um Daten von einem Gerät zur Handschrifterkennung auf einen Sprachsynthesizer zu kopieren: Wie müssten Sie das `copy`-Programm modifizieren, damit es mit diesen neuen Geräten funktioniert?

Sie müssten überhaupt keine Modifizierungen vornehmen! Tatsächlich brauchen Sie das `copy`-Programm nicht einmal neu zu kompilieren. Warum? Weil der Quellcode des `copy`-Programms nicht von dem Quellcode der I/O-Treiber abhängig ist. Solange diese I/O-Treiber die durch `FILE` definierten fünf Standardfunktionen implementieren, wird das `copy`-Programm sie ohne Probleme nutzen.

Vereinfacht ausgedrückt: Die I/O-Geräte sind für das `copy`-Programm zu Plug-ins geworden.

Und warum wurden I/O-Geräte unter dem UNIX-Betriebssystem zu Plug-ins gemacht? Weil wir in den späten 1950er-Jahren gelernt haben, dass unsere Programme *geräteunabhängig* sein sollten. Wieso? Weil wir jede Menge Programme geschrieben haben, die *geräteabhängig* waren, nur um dann festzustellen, dass wir eigentlich wollten, dass diese Programme genau dieselben Aufgaben auch auf anderen Geräten ausführen können sollten.

So wurden beispielsweise häufig Programme geschrieben, die Daten von Lochkartenstapeln[6] einlasen und davon dann wiederum neue Lochkartenstapel ausgaben. Später gingen unsere Auftraggeber dazu über, uns anstelle der Lochkarten Magnetbandrollen zu übergeben. Das war sehr unpraktisch, weil es bedeutete, dass große Teile des Originalprogramms umgeschrieben werden mussten. Hier wäre es äußerst vorteilhaft gewesen, wenn ein und dasselbe Programm austauschbar mit Karten oder Magnetband hätte arbeiten können.

Die Plug-in-Architektur wurde entwickelt, um genau diese Art von I/O-Geräteunabhängigkeit zu unterstützen, und ist seit ihrer Einführung in fast jedem Betriebssystem implementiert. Trotzdem haben die meisten Programmierer dieses Konzept nicht für ihre eigenen Programme übernommen, weil die Verwendung von Zeigern auf Funktionen riskant war.

Die Objektorientierung gestattet den Einsatz der Plug-in-Architektur überall und für alles Mögliche.

5.3.2 Abhängigkeitsumkehr

Versuchen Sie sich einmal vorzustellen, wie Software aussah, bevor ein sicherer und bequemer Mechanismus für die Polymorphie zur Verfügung stand. In dem typischen Aufrufschema riefen die Hauptfunktionen (`main`) die hochschichtigen übergeordneten Funktionen (**H**igh-**L**evel, *HL*) auf, die ihrerseits für den Aufruf der mittelschichtigen Funktionen (**M**id-**L**evel, *ML*) sorgten, die wiederum die tiefschichtigen untergeordneten Funktionen (**L**ow-**L**evel, *LL*) aufriefen. In diesem Aufrufschema folgten die Quellcode-Abhängigkeiten jedoch unerbittlich dem Kontrollfluss (siehe Abbildung 5.1).

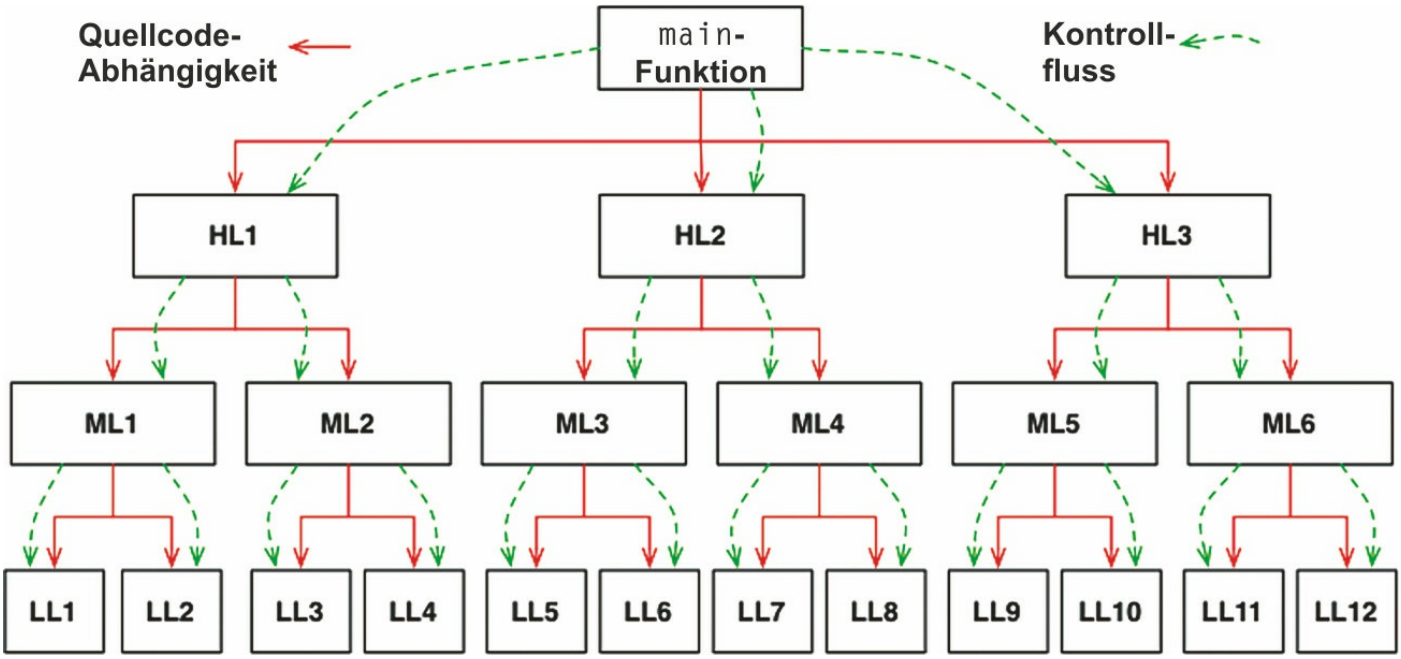


Abb. 5.1: Quellcode-Abhängigkeiten vs. Kontrollfluss

Damit `main` eine der übergeordneten Funktionen aufrufen konnte, musste sie den Namen des Moduls angeben, das die betreffende Funktion enthielt. In C war das `#include`, in Java eine `import`- und in C# eine `using`-Anweisung. Somit war jeder Aufrufer gezwungen, den Namen des Moduls zu benennen, in dem sich die aufzurufende Funktion befand.

Diese Anforderung ließ Softwarearchitekten nur wenige, wenn überhaupt irgendwelche Optionen offen. Der Kontrollfluss wurde von dem Verhalten des Systems diktiert, und die Quellcode-Abhängigkeit wurde von diesem Kontrollfluss diktiert.

Wenn man nun die Polymorphie in dieses Verfahren einbringt, kann allerdings etwas ganz anderes passieren (siehe Abbildung 5.2).

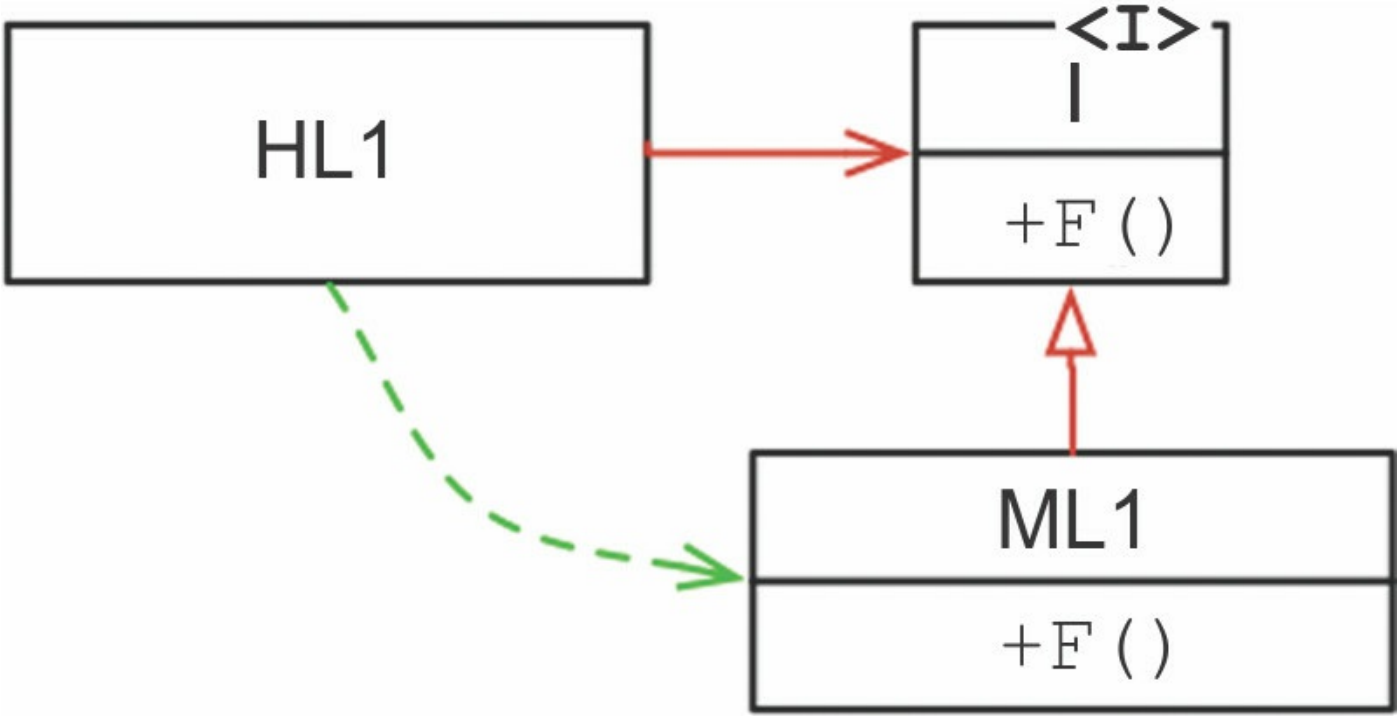


Abb. 5.2: Die Abhängigkeitsumkehr

In Abbildung 5.2 ruft das Modul `HL1` die Funktion `F()` im Modul `ML1` auf. Die Tatsache, dass diese Funktion über eine Schnittstelle aufgerufen wird, ist eine Quellcode-Vorgabe. Zur Laufzeit existiert die Schnittstelle nicht. `HL1` ruft einfach `F()` innerhalb von `ML1` auf.[7]

Beachten Sie jedoch, dass die Quellcode-Abhängigkeit (die Vererbungsbeziehung) zwischen `ML1` und der Schnittstelle `I` im Vergleich zum Kontrollfluss in die entgegengesetzte Richtung weist. Dieses Prinzip wird als *Abhängigkeitsumkehr* bezeichnet und hat tiefgreifende Auswirkungen auf die Softwarearchitektur.

Die Tatsache, dass objektorientierte Sprachen eine sichere und komfortable Polymorphie ermöglichen, bedeutet, dass *jede Quellcode-Abhängigkeit, unabhängig davon, wo sie besteht, umgekehrt werden kann*.

Betrachten Sie nun noch einmal das Aufrufschema in Abbildung 5.1 und dessen viele Quellcode-Abhängigkeiten. Jede davon kann durch das Einsetzen einer Schnittstelle zwischen ihnen umgekehrt werden.

Mit diesem Ansatz haben Softwarearchitekten in Systemen, die in objektorientierten Sprachen geschrieben sind, die *absolute Kontrolle* über die Ausrichtung aller systemweiten Quellcode-Abhängigkeiten. Sie sind nicht darauf beschränkt, diese Abhängigkeiten nach dem Kontrollfluss auszurichten. Egal, welches Modul den Aufruf initiiert und welches Modul aufgerufen wird, der Softwarearchitekt kann die Quellcode-Abhängigkeit in jede Richtung anzeigen.

Das ist Potenzial! Das ist das Potenzial, das die Objektorientierung gewährt. Und

genau darum geht es im Grunde genommen bei der Objektorientierung – zumindest aus der Perspektive des Softwarearchitekten.

Aber was können Sie mit diesem Potenzial anfangen? Zum Beispiel können Sie die Quellcode-Abhängigkeiten Ihres Systems so neu ordnen, dass die Datenbank und die Benutzerschnittstelle (**U**ser **I**nterface, *UI*) von den Geschäftsregeln abhängig sind statt anders herum.

Abb. 5.3: Die Datenbank und die Benutzerschnittstelle sind von den Geschäftsregeln abhängig.

Somit können die Benutzerschnittstelle und die Datenbank Plug-ins für die Geschäftsregeln sein – und das bedeutet, dass beide zu keinem Zeitpunkt im Quellcode der Geschäftsregeln erwähnt werden.

Demzufolge können die Geschäftsregeln, die Benutzerschnittstelle und die Datenbank in drei separate Komponenten oder Verteilungseinheiten (z.B. `.jar`-Dateien, DLLs oder `.gem`-Dateien) kompiliert werden, die dieselben Abhängigkeiten wie der Quellcode aufweisen. Die Komponente, die die Geschäftsregeln enthält, wird nicht von den Komponenten abhängig sein, die die Benutzerschnittstelle und die Datenbank enthalten.

Im Gegenzug ist eine von der Benutzerschnittstelle und der Datenbank *unabhängige Verteilung* der Geschäftsregeln möglich. Änderungen an der Benutzerschnittstelle oder der Datenbank brauchen keine Auswirkungen auf die Geschäftsregeln zu haben. Diese Komponenten können separat und unabhängig verteilt werden.

Zusammengefasst heißt das: Wenn sich der Quellcode in einer Komponente ändert, muss nur diese Komponente neu verteilt werden. Dieses Konzept wird als *unabhängige Verteilbarkeit* bezeichnet.

Wenn die Module in Ihrem System unabhängig voneinander verteilt werden können, dann können sie auch unabhängig voneinander von verschiedenen Teams entwickelt werden. Hier spricht man von *unabhängiger Entwickelbarkeit*.

5.4 Fazit

Was ist Objektorientierung? Zu dieser Frage gibt es viele Meinungen und Ansichten. Für den Softwarearchitekten ist die Antwort jedoch eindeutig: Die Objektorientierung ist die Fähigkeit, durch die Nutzung der Polymorphie die absolute Kontrolle über jede Quellcode-Abhängigkeit im System zu erlangen. Sie ermöglicht es dem Softwarearchitekten, eine Plug-in-Architektur zu errichten, in der Module, die übergeordnete Richtlinien (engl. *Policy*) umfassen, von Modulen, die untergeordnete Details enthalten, unabhängig sind. Die untergeordneten Details werden auf die Plug-in-Module verteilt, die unabhängig von denjenigen Modulen, in denen sich übergeordnete Richtlinien befinden, verteilt und entwickelt werden können.

---

[1] Der C++-Compiler muss die Größe der Instanzen jeder Klasse kennen.

[2] Zum Beispiel Smalltalk, Python, JavaScript, Lua und Ruby.

[3] Und nicht nur C-Programmierer: Die meisten Programmiersprachen dieser Ära ermöglichten es, eine Datenstruktur als eine andere zu maskieren.

[4] Im Grunde ist das auch heute noch so.

[5] UNIX-Systeme variieren, hierbei handelt es sich lediglich um ein Beispiel.

[6] IBM-Hollerith-Lochkarten, 80 Spalten breit. Ich bin ziemlich sicher, dass nicht allzu viele von Ihnen solche Karten je zu Gesicht bekommen haben, dennoch waren sie in den 1950er-, 1960er- und auch 1970er-Jahren gang und gäbe.

[7] Wenn auch indirekt.

Kapitel 6

Funktionale Programmierung

Die der *funktionalen Programmierung* innewohnenden Konzepte sind in vielerlei Hinsicht älter als die Programmierung selbst. Dieses Paradigma basiert in weiten Teilen auf dem von Alonzo Church in den 1930er-Jahren erfundenen λ-*(Lambda-*

)Kalkül.

6.1 Quadrierung von Integern

Am besten lässt sich die funktionale Programmierung anhand einiger Beispiele erklären. Untersuchen wir einmal ein einfaches Problem: das Ausdrucken der ersten 25 quadrierten Integerwerte.

In einer Sprache wie Java würden wir dazu etwa Folgendes schreiben:

```
public class Squint {
   public static void main(String args[]) {
      for (int i=0; i<25; i++)
         System.out.println(i*i);
   }
}
```

In einer Sprache wie Clojure, einem funktionalen Ableger von LISP, ließe sich das gleiche Programm wie folgt implementieren:

```
(println (take 25 (map (fn [x] (* x x)) (range))))
```

Wenn Sie sich nicht mit LISP auskennen, mag Ihnen das ein wenig seltsam erscheinen. Deshalb formatieren wir das Ganze nun etwas um und ergänzen einige Anmerkungen:

```
(println ;_____________________ Ausgabe
   (take 25 ;__________________ der ersten 25
      (map (fn [x] (* x x)) ;___ quadrierten
         (range))) ;___________ Integer
```

Hier wird deutlich, dass `println`, `take`, `map` und `range` allesamt Funktionen sind. Zum Aufruf einer Funktion in LISP wird diese in Klammern gesetzt. So wird mit `(range)` beispielsweise die `range`-Funktion aufgerufen.

Bei dem Ausdruck `(fn [x] (* x x))` handelt es sich um eine anonyme Funktion, die die Multiplizierfunktion aufruft und dabei ihr Eingabeargument zweimal übergibt. Mit anderen Worten: Sie quadriert ihren Eingabewert.

Zur genaueren Betrachtung des Codes fängt man am besten mit dem innersten Funktionsaufruf an:

- Die `range`-Funktion gibt eine endlose Liste von Integern aus, die mit 0 beginnt.
- Diese Liste wird an die `map`-Funktion übergeben, die die anonyme Quadrierungsfunktion für jedes Element aufruft und so eine neue Endlosliste aller quadrierten Werte erzeugt.
- Die Liste der quadrierten Werte wird an die `take`-Funktion übergeben, die daraufhin eine neue Liste mit lediglich den ersten 25 Elementen zurückliefert.
- Die `println`-Funktion gibt ihre Eingabewerte aus, also eine Liste der ersten 25 quadrierten Integerwerte.

Wenn Ihnen nun angesichts des Konzepts der endlosen Listen mulmig wird – keine Sorge! Tatsächlich werden nur die ersten 25 Elemente dieser Endloslisten erzeugt, und zwar weil kein Element einer solchen Liste ausgewertet wird, bis der Zugriff darauf erfolgt.

Sollte Ihnen dies alles verwirrend vorkommen, dann können Sie sich auf eine großartige Zeit freuen, in der Sie sich alle nötigen Kenntnisse über Clojure und die funktionale Programmierung aneignen. Was dieses Buch betrifft, ist es allerdings nicht mein Ziel, Ihnen diese Themen näherzubringen.

Vielmehr soll Ihnen hier ein drastischer Unterschied zwischen den Clojure- und Java-Programmen vor Augen geführt werden. Das Java-Programm nutzt eine *veränderbare Variable* – eine Variable, die während der Ausführung des Programms Zustandsänderungen vornimmt. Hierbei handelt es sich um die Variable `i`, die Variable zur Schleifensteuerung. In dem Clojure-Programm existiert hingegen keine solche veränderbare Variable. In diesem Fall werden Variablen wie `x` initialisiert, aber zu keinem Zeitpunkt modifiziert.

Dies führt uns zu einer überraschenden Erkenntnis: Variablen in funktionalen Sprachen *variieren nicht.*

6.2 Unveränderbarkeit und Architektur

Warum muss dieser Punkt als wichtiger Aspekt bei den Überlegungen zur Architektur berücksichtigt werden? Weshalb sollte sich ein Softwarearchitekt mit der Veränderbarkeit von Variablen auseinandersetzen? Aus einem absurd einfachen Grund: Alle *Race Conditions* (Wettlaufsituationen), *Deadlocks* (auch »Verklemmungen« genannt) und das Problem der *nebenläufigen Aktualisierungen* sind auf veränderbare Variablen zurückzuführen. Wenn keine Variable jemals aktualisiert wird, kann es auch nicht zu Race Conditions oder einem Problem mit parallel ausgeführten Aktualisierungen kommen. Ebenso sind auch Deadlocks ohne

veränderbare Locks (Sperren) nicht möglich.

Das bedeutet, dass sämtliche Probleme, die in nebenläufigen Anwendungen auftreten – also alle Schwierigkeiten in Anwendungen, die Multithreading und Multiprocessing erfordern –, nicht entstehen können, wenn es keine veränderbaren Variablen gibt.

Als Softwarearchitekt sollten Sie den Problemen mit der Nebenläufigkeit besondere Aufmerksamkeit zukommen lassen – immerhin wollen Sie ja, dass die Systeme, die Sie designen, auch angesichts von Multithreading und Multiprocessing stabil laufen. Die Frage, die Sie sich selbst stellen müssen, ist, ob die Unveränderbarkeit praktikabel ist.

Wenn Sie über unendliche Speicherkapazitäten und Prozessorleistung verfügen, lautet die Antwort auf diese Frage: Ja, sie ist praktikabel. Fehlen diese Ressourcen jedoch, dann fällt die Antwort etwas differenzierter aus: Ja, die Unveränderbarkeit kann praktikabel sein, wenn bestimmte Kompromisse eingegangen werden.

Welche genau das sind, erfahren Sie im Folgenden.

6.3 Unterteilung der Veränderbarkeit

Einer der gängigsten Kompromisse in Bezug auf die Unveränderbarkeit besteht darin, die Anwendung bzw. die darin befindlichen Services in veränderbare und unveränderbare Komponenten zu unterteilen. Die unveränderbaren Komponenten führen ihre Aufgaben in einer rein funktionalen Art und Weise aus, ohne irgendwelche veränderbaren Variablen zu benutzen. Sie kommunizieren mit einer oder mehreren Komponenten, die nicht rein funktional sind, und gestatten Zustandsänderungen an den Variablen (siehe Abbildung 6.1).

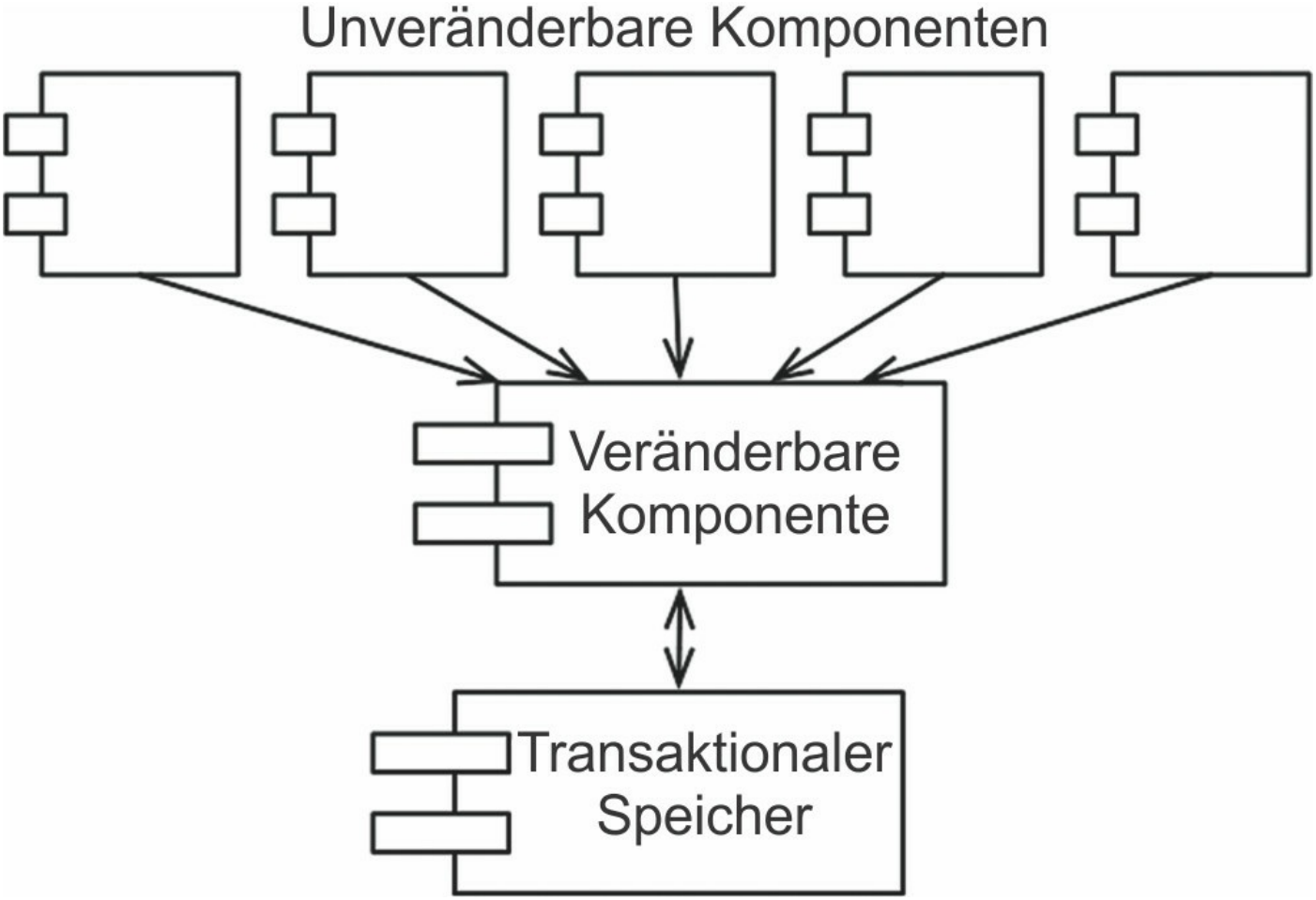


Abb. 6.1: Veränderungszustand und transaktionaler Speicher

Da diese Komponenten durch Zustandsänderungen allen Schwierigkeiten der Nebenläufigkeit ausgesetzt sind, hat es sich als gängige Praxis eingebürgert, irgendeine Form von *transaktionalem Speicher* zu verwenden, um veränderbare Variablen vor nebenläufigen Aktualisierungen und Race Conditions zu schützen.

Der transaktionale Speicher behandelt Variablen im Speicher einfach auf dieselbe Weise wie eine Datenbank die auf einer Diskette befindlichen Daten.[1] Er schützt diese Variablen mithilfe eines transaktions- oder wiederholungsbasierten Schemas.

Ein einfaches Beispiel für diese Herangehensweise ist das in Clojure zur Verfügung stehende Hilfsmittel `atom`:

```
(def counter (atom 0)) ; initialisiert counter auf 0
(swap! counter inc)    ; sichere Inkrementierung von counter
```

In diesem Code ist die `counter`-Variable als ein Atom (`atom`) definiert. Hierbei handelt es sich um eine spezielle, in Clojure verwendete Variablenart, deren Wert unter sehr disziplinierten Bedingungen, die von der `swap!`-Funktion erzwungen werden, mutieren darf.

Die `swap!`-Funktion, die in dem vorstehenden Code enthalten ist, erhält zwei Argumente: das zu ändernde `atom` und eine Funktion, die den neuen, in dem Atom zu speichernden Wert berechnet. In diesem Beispielcode wird das `counter atom` in den von der `inc`-Funktion berechneten Wert geändert, die einfach dessen Argument erhöht.

Die mittels `swap!` angewendete Strategie entspricht einem traditionellen *Compare-and-Swap-Algorithmus* (*CAS*, Vergleichen und Tauschen). Der Wert von `counter` wird ausgelesen und an `inc` übergeben. Bei der Rückgabe durch `inc` wird der Wert von `counter` gesperrt und mit demjenigen verglichen, der an `inc` übergeben wurde. Sind beide Werte identisch, wird der von `inc` zurückgegebene Wert in `counter` gespeichert und die Sperre wird aufgehoben. Ansonsten wird die Sperre ebenfalls aufgehoben und die Strategie wird von Neuem versucht.

Das Clojure-Atom ist für einfache Anwendungen hinreichend geeignet. Leider bietet es keinen vollständigen Schutz gegen nebenläufige Aktualisierungen und Deadlocks, sobald mehrere abhängige Variablen ins Spiel kommen. In diesen Fällen können aufwendigere Hilfsmittel eingesetzt werden.

Der Punkt ist, dass gut strukturierte Anwendungen in diejenigen Komponenten unterteilt werden, die keine Variablen verändern, und diejenigen, die es tun. Eine derartige Unterteilung wird durch den Einsatz geeigneter Maßnahmen unterstützt, um diese veränderten Variablen zu schützen.

Softwarearchitekten sind gut beraten, so viel Verarbeitungslast wie möglich auf die unveränderbaren Komponenten zu verlagern und so viel Code wie möglich aus denjenigen Komponenten herauszuholen, die Veränderungen zulassen müssen.

6.4 Event Sourcing

Die in der Vergangenheit geltenden Einschränkungen hinsichtlich Speicherkapazitäten und Verarbeitungsleistung haben rapide an Bedeutung verloren. Heutzutage ist es üblich, dass Prozessoren Milliarden von Anweisungen pro Sekunde ausführen und Milliarden von Bytes im RAM-Speicher halten können. Je mehr Speicherkapazität zur Verfügung steht und je schneller unsere Maschinen sind, desto weniger Bedarf besteht auch für veränderbare Zustände.

Stellen Sie sich als einfaches Beispiel einmal eine Bankanwendung vor, die die Kontostände und -bewegungen der Bankkunden vorhält und diese Daten entsprechend der ausgeführten Ein- und Auszahlungstransaktionen modifiziert.

Stellen Sie sich nun außerdem vor, dass statt der Kontodaten nur die Transaktionen gespeichert würden. Wann immer jemand seinen Kontostand abfragt, werden einfach alle zu dem betreffenden Konto gehörigen Daten zusammenaddiert, und zwar vom Tag

der Kontoeröffnung an. Dieses Verfahren erfordert keine veränderbaren Variablen.

Natürlich erscheint dieser Ansatz völlig absurd: Die Zahl der Transaktionen würde im Laufe der Zeit ins Unermessliche steigen und die für die Berechnung der Kontostände erforderliche Verarbeitungsleistung würde jeden Rahmen sprengen. Damit ein solches System dauerhaft funktionieren könnte, müsste man sowohl über unendliche Speicherkapazität als auch über unendliche Verarbeitungsleistung verfügen.

Aber vielleicht muss es ja gar nicht bis in alle Ewigkeit funktionieren. Und vielleicht sind ja auch genügend Speicherkapazitäten und Verarbeitungsleistung vorhanden, damit die Anwendung hinreichend langfristig genutzt werden kann.

Das Konzept, das dahintersteckt, wird als *Event Sourcing* (zu Deutsch etwa »Ereignisverfolgung«) bezeichnet.[2] Hierbei handelt es sich um eine Strategie, in deren Kontext zwar die Transaktionen, nicht jedoch die Zustände gespeichert werden. Sobald eine Zustandsabfrage erforderlich ist, werden einfach alle Transaktionen rückwirkend von Anbeginn an berücksichtigt.

Natürlich sind dabei auch Abkürzungen möglich. Beispielsweise kann man die Berechnungen jeden Tag um Mitternacht durchführen und den daraus jeweils resultierenden Zustand speichern. Wenn die Zustandsinformationen dann benötigt werden, müssen nur noch die seit Mitternacht ausgeführten Transaktionen berechnet werden.

Berücksichtigt man nun den für dieses Verfahren benötigten Datenspeicher, so muss er schon von beachtlicher Größe sein. Dementsprechend sind unsere Anwendungen auch nicht mit den *CRUD-Datenbankoperationen* (**C**reate, **R**ead, **U**pdate, **D**elete) ausgestattet, sondern bieten nur die *CR-Operationen* (**C**reate, **R**ead). Und weil weder Aktualisierungen noch Löschungen im Datenspeicher vorgenommen werden, können auch keine Probleme mit nebenläufigen Aktualisierungen auftreten.

Sofern jedoch genügend Speicher und Verarbeitungsleistung bereitsteht, können wir unsere Anwendungen vollständig unveränderbar gestalten – und damit *vollständig funktional*.

Sollte Ihnen das immer noch absurd vorkommen, dann hilft es vielleicht, wenn Sie sich daran erinnern, dass dies exakt der Art und Weise entspricht, wie Ihre Versionsverwaltung (**S**ource **C**ode **C**ontrol **S**ystem, *SCCS*) arbeitet.

6.5 Fazit

Fassen wir zusammen:

- Die strukturierte Programmierung unterwirft die direkte Kontrollübertragung einer strengen Disziplin.
- Die objektorientierte Programmierung unterwirft die indirekte Kontrollübertragung einer strengen Disziplin.
- Die funktionale Programmierung unterwirft die Variablenzuweisung einer strengen Disziplin.

Jedes dieser drei Paradigmen entzieht uns einen Teil unserer Programmierfreiheit. Sie alle schränken irgendeinen Aspekt in Bezug auf die Art und Weise, wie wir Code schreiben, ein. Nicht eines dieser Paradigmen erweitert dagegen unsere Möglichkeiten oder Fähigkeiten.

Was wir im letzten halben Jahrhundert gelernt haben, ist, *was wir nicht tun dürfen*.

Angesichts dieser Erkenntnis müssen wir uns einer unliebsamen Tatsache stellen: Die Softwareentwicklung ist keine rasant voranschreitende Technologie. Auch heute noch gelten dieselben Regeln für die Programmierung von Software, wie dies schon 1946 der Fall war, als Alan Turing seinen allerersten Code schrieb, der auf einem elektronischen Computer ausgeführt wurde. Die Tools haben sich geändert, ebenso die Hardware, aber die Software bleibt im Wesentlichen immer gleich.

Software – also alles, was mit Computerprogrammen zu tun hat – setzt sich aus Sequenz, Selektion, Iteration und Dereferenzierung zusammen. Nicht mehr und nicht weniger.

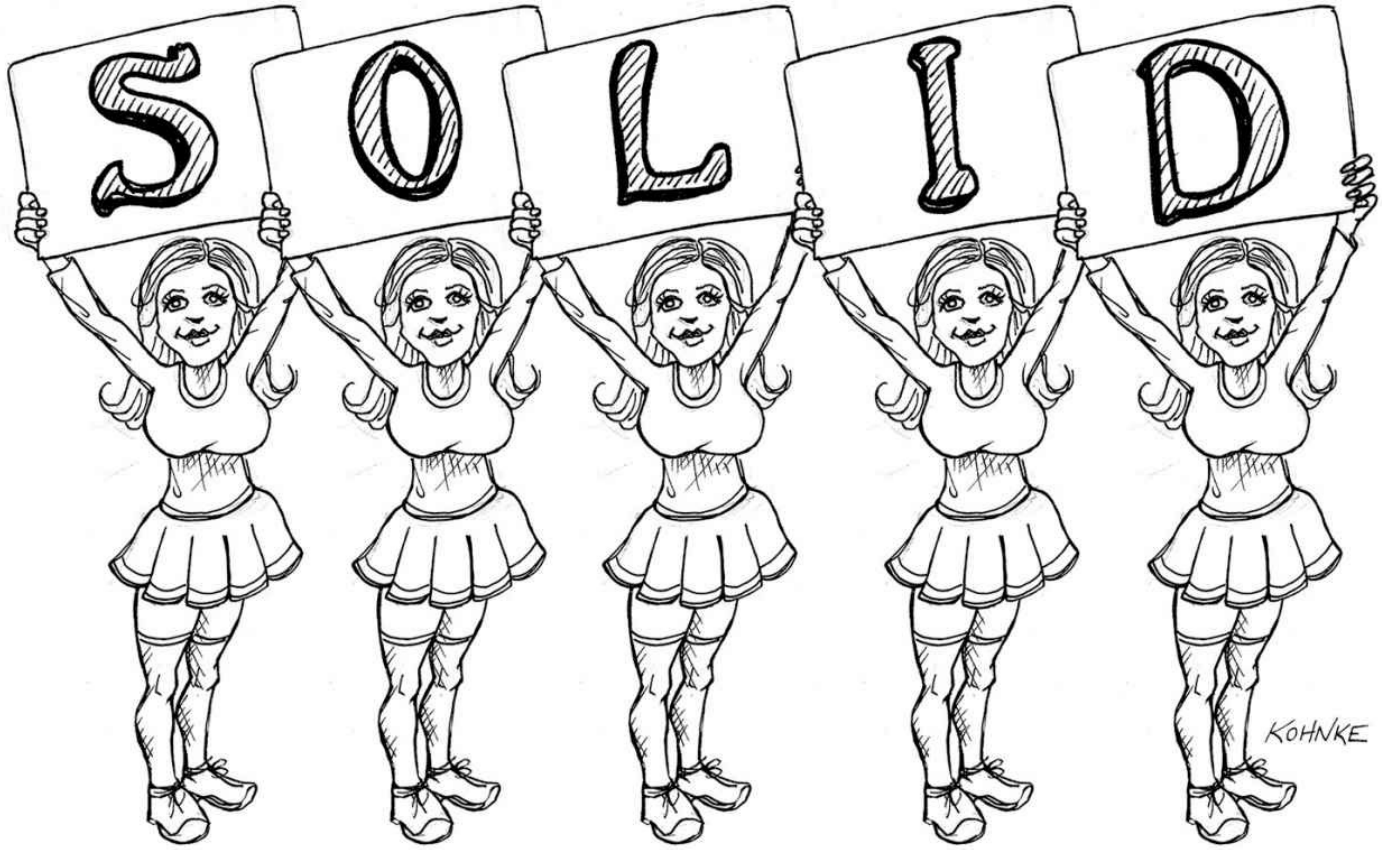
---

[1] Ja, ich weiß schon ... Was in aller Welt ist eine Diskette?

[2] Ich bedanke mich bei Greg Young, der mich über dieses Konzept aufgeklärt hat.

Teil III
Designprinzipien

In diesem Teil:



Gute Softwaresysteme bauen auf *Clean Code*, also sauberem Code auf. Einerseits spielt die Architektur eines Softwarekonstrukts keine große Rolle, wenn die einzelnen Bausteine nicht von guter Qualität sind – andererseits kann man aber durchaus auch mit qualitativ hochwertigen Bausteinen erhebliches Chaos anrichten. Und genau hier kommen die sogenannten *SOLID-Prinzipien* (**S**ingle-Responsibility-Prinzip, **O**pen-Closed-Prinzip, **L**iskov'sches Substitutionsprinzip, **I**nterface-Segregation-Prinzip, **D**ependency-Inversion-Prinzip) zum Tragen.

Die SOLID-Prinzipien geben vor, wie wir unsere Funktionen und Datenstrukturen in Klassen anordnen und wie diese Klassen miteinander verbunden werden. Die Verwendung des Begriffs »Klasse« soll an dieser Stelle allerdings keineswegs bedeuten, dass die Prinzipien nur auf objektorientierte Software angewendet werden sollten – eine Klasse ist einfach eine gekoppelte Gruppierung von Funktionen und Daten. Jedes Softwaresystem besitzt solche Gruppierungen, ob sie nun als Klassen bezeichnet werden oder nicht. Und genau dafür gelten die SOLID-Prinzipien.

Das Ziel dabei ist die Erzeugung von mittelschichtigen Softwarestrukturen, die

- Modifikationen tolerieren,
- leicht nachzuvollziehen sind und
- die Basis der Komponenten bilden, die in vielen Softwaresystemen eingesetzt werden können.

Der Begriff »mittelschichtig« bezieht sich auf den Umstand, dass die Prinzipien von Programmierern angewendet werden, die auf der Modulebene arbeiten. Sie werden unmittelbar oberhalb des Codes angewendet und helfen bei der Definition der Arten von Softwarestrukturen, die in Modulen und Komponenten eingesetzt werden.

Ebenso wie sich mit qualitativ hochwertigen Bausteinen ein beachtliches Chaos anrichten lässt, ist es aber ebenso möglich, ein systemweites Chaos mit gut designten mittelschichtigen Komponenten zu verursachen. Deshalb werden wir uns nach der Betrachtung der SOLID-Prinzipien im weiteren Verlauf auch mit ihren Gegenstücken in der Komponentenwelt sowie anschließend mit den Prinzipien der übergeordneten hochschichtigen Architektur befassen.

Die Geschichte der SOLID-Prinzipien reicht lange zurück. Ich persönlich begann in den späten 1980er-Jahren, sie zu ergründen – während einer Diskussion mit anderen Leuten auf USENET (einer frühen Form von Facebook). Im Laufe der Jahre haben sich die Prinzipien verschoben und verändert. Einige wurden verworfen, andere verschmolzen, und auch neue wurden hinzugefügt. Die endgültige Zusammenstellung verfestigte sich erst in den frühen 2000er-Jahren, wenngleich ich sie hier in einer anderen Reihenfolge aufgeführt habe.

Etwa um das Jahr 2004 schickte mir Michael Feathers eine E-Mail, in der er mich darauf aufmerksam machte, dass die Anfangsbuchstaben der Prinzipien, das englische Wort *SOLID* (zu Deutsch »solide, stabil«) ergeben würden, wenn ich sie umordnete – und so wurde der Begriff »SOLID-Prinzipien« geboren.

In den nachfolgenden Kapiteln wird jedes einzelne Prinzip genauer beschrieben. Hier zunächst eine kurze Zusammenfassung:

SRP: Das *Single-Responsibility-Prinzip*

Eine logische Folge aus dem Gesetz von Conway: Die beste Struktur für ein Softwaresystem ist stark von der sozialen Struktur der Organisation beeinflusst, die es benutzt, sodass es für jedes Softwaremodul einen – und nur einen – Änderungsgrund gibt.

OCP: Das *Open-Closed-Prinzip*

Bertrand Meyer machte dieses Prinzip in den 1980er-Jahren berühmt. Das Wesentliche ist hierbei, dass Softwaresysteme zwecks einfacher Anpassung so entworfen sein müssen, dass eine Veränderung ihres Verhaltens durch die Ergänzung neuen Codes statt durch das Modifizieren vorhandenen Codes möglich ist.

LSP: Das *Liskov'sche Substitutionsprinzip*

Barbara Liskovs berühmte Definition der Subtypen aus dem Jahr 1988. Vereinfacht ausgedrückt beschreibt dieses Prinzip, dass es für die Errichtung von Softwaresystemen mit austauschbaren Objekten erforderlich ist, dass diese Elemente wechselseitig ersetzbar sein müssen.

ISP: Das *Interface-Segregation-Prinzip*

Dieses Prinzip hält Softwaredesigner dazu an, Abhängigkeiten von nicht genutzten Modulen zu vermeiden.

DIP: Das *Dependency-Inversion-Prinzip*

Der Code, der die übergeordneten Richtlinien (engl. *Policy*) implementiert, sollte nicht von dem Code abhängig sein, der untergeordnete Details implementiert. Vielmehr sollten Details von den Richtlinien abhängig sein.

Diese Prinzipien wurden im Laufe der Zeit in zahlreichen verschiedenen Publikationen detailliert beschrieben.[1] Die nachfolgenden Kapitel konzentrieren sich nicht so sehr auf die Wiederholung dieser detaillierten Diskussionen, sondern vielmehr auf die architektonischen Auswirkungen dieser Prinzipien. Sollten Sie noch nicht mit den SOLID-Prinzipien vertraut sein, werden die folgenden Ausführungen nicht für ein tieferes Verständnis ausreichen – insofern empfiehlt es sich, gegebenenfalls die in der Fußnote erwähnten Informationsquellen zurate zu ziehen.

---

[1] Beispielsweise in *Agile Software Development, Principles, Patterns, and Practices*, Robert C. Martin, Prentice Hall, 2002, http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod und https://de.wikipedia.org/wiki/Prinzipien_objektorientierten_Designs#SOLID-Prinzipien (oder googeln Sie einfach den Begriff »SOLID«).

Kapitel 7

SRP: Das Single-Responsibility-Prinzip

Das *Single-Responsibility-Prinzip (SRP)* dürfte das von allen SOLID-Prinzipien am meisten Missverstandene sein. Einer der Gründe dafür ist vermutlich seine besonders unpassende Bezeichnung (»Prinzip der eindeutigen Verantwortlichkeit«), die Programmierer allzu leicht zu der Annahme verleitet, dass jedes Modul ausschließlich eine einzige Aufgabe erfüllen sollte.

Und tatsächlich gibt es auch eine solche Richtlinie, die da lautet: Eine *Funktion* sollte immer nur eine bestimmte, und nur diese eine Aufgabe haben. Dieser Grundsatz wird beim Refactoring umfangreicher Funktionen in kleinere Funktionen angewendet, und zwar auf den niedrigsten Ebenen. Er gehört jedoch *nicht* zu den SOLID-Prinzipien –

mit anderen Worten: Das ist *nicht*, worum es beim SRP geht.

Traditionell lautet die Beschreibung des SRPs wie folgt:

> *Es sollte nie mehr als einen Grund geben, eine Klasse zu modifizieren.*[1]

Softwaresysteme werden in erster Linie modifiziert, um User und Stakeholder zufriedenzustellen. Sie sind der eigentliche Grund für eine Anpassung der Art, um die es bei diesem Prinzip geht. Man könnte die Beschreibung also auch folgendermaßen umformulieren:

> *Ein Modul sollte für einen, und nur einen, User oder Stakeholder verantwortlich sein.*

Die Aussage »für einen User oder Stakeholder« ist hier allerdings nicht wirklich zutreffend, denn es ist sehr wahrscheinlich, dass ein und dieselbe Systemänderung gleich von mehreren Usern oder Stakeholdern gewünscht wird. Es handelt sich daher eher um Gruppen, die wir nachstehend mit dem Sammelbegriff »Akteur« bezeichnen.

Die finale Version unserer SRP-Beschreibung muss somit lauten:

> *Ein Modul sollte für einen, und nur einen, Akteur verantwortlich sein.*

Doch was ist denn eigentlich mit dem Begriff »Modul« gemeint? Die einfachste Antwort auf diese Frage ist: eine Quelldatei. In den meisten Fällen funktioniert diese Definition wunderbar. Einige Sprachen und Entwicklungsumgebungen nutzen jedoch keine Quelldateien in ihrem Code. In diesen Fällen versteht man unter einem Modul einfach einen zusammenhängenden Satz von Funktionen und Datenstrukturen. Ausschlaggebend für das SRP ist hierbei das Wort »zusammenhängend«: Der Zusammenhalt (die *Kohäsion*) des Moduls ist die treibende Kraft, die den für einen einzelnen Akteur verantwortlichen Code bündelt.

Am besten lässt sich das Single-Responsibility-Prinzip veranschaulichen, indem man die Symptome der möglichen Verstöße gegen Selbiges betrachtet.

7.1 Symptom 1: Versehentliche Duplizierung

Mein Lieblingsbeispiel ist die `Employee`-Klasse einer Anwendung für Lohn- und Gehaltsabrechnungen. Sie verfügt über drei Methoden: `calculatePay()`, `reportHours()` und `save()` (siehe Abbildung 7.1).

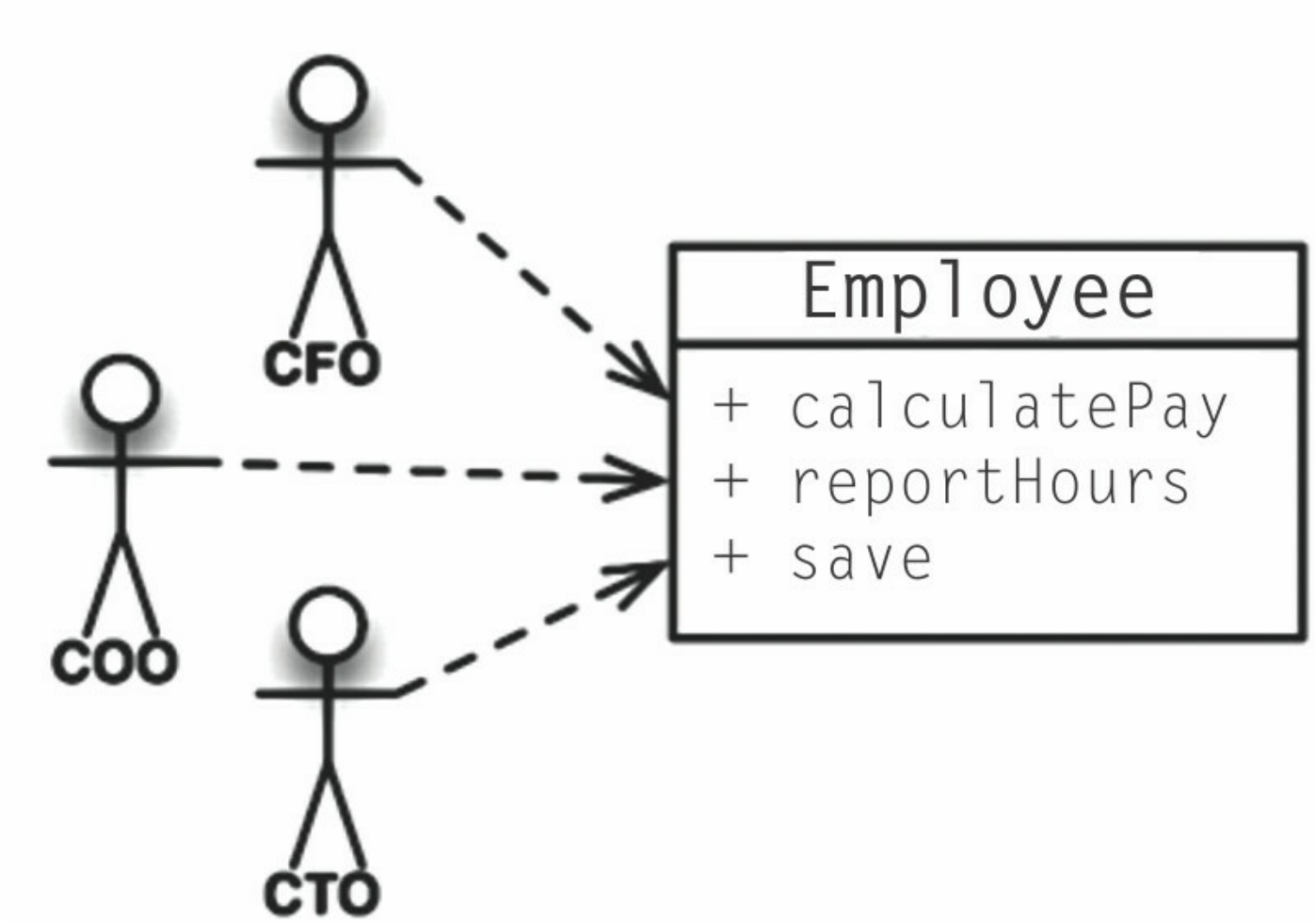


Abb. 7.1: Die Klasse `Employee`

Diese Klasse verstößt gegen das SRP, weil die zugehörigen drei Methoden drei sehr verschiedenen Akteuren gegenüber verantwortlich sind:

- Die Methode `calculatePay()` ist durch die Buchhaltung spezifiziert, die dem CFO (Finanzvorstand) Bericht erstattet.
- Die Methode `reportHours()` ist durch die Personalabteilung spezifiziert und wird von dieser genutzt, um dem COO (Geschäftsleitung) Bericht zu erstatten.
- Und die Methode `save()` ist durch die Datenbankadministratoren (DBAs) spezifiziert, die dem CTO (Technischen Leiter) berichten.

Dadurch, dass der Quellcode für diese drei Methoden in einer einzigen `Employee`-Klasse abgelegt wurde, haben die Softwareentwickler jeden der genannten Akteure mit den anderen verbunden. Eine derartige Koppelung kann dann beispielsweise zur Folge haben, dass sich vom CFO-Team durchgeführte Aktionen auf etwas auswirken, von dem das COO-Team abhängig ist.

Nehmen wir einmal an, dass sich die Funktion `calculatePay()` und die Funktion `reportHours()` einen gemeinsamen Algorithmus für die Berechnung von unbezahlten

Überstunden teilen. Gehen wir außerdem davon aus, dass die Programmierer, die sorgfältig darauf bedacht sind, Codeduplizierungen zu vermeiden, diesen Algorithmus in einer Funktion namens `regularHours()` verwenden (siehe Abbildung 7.2).

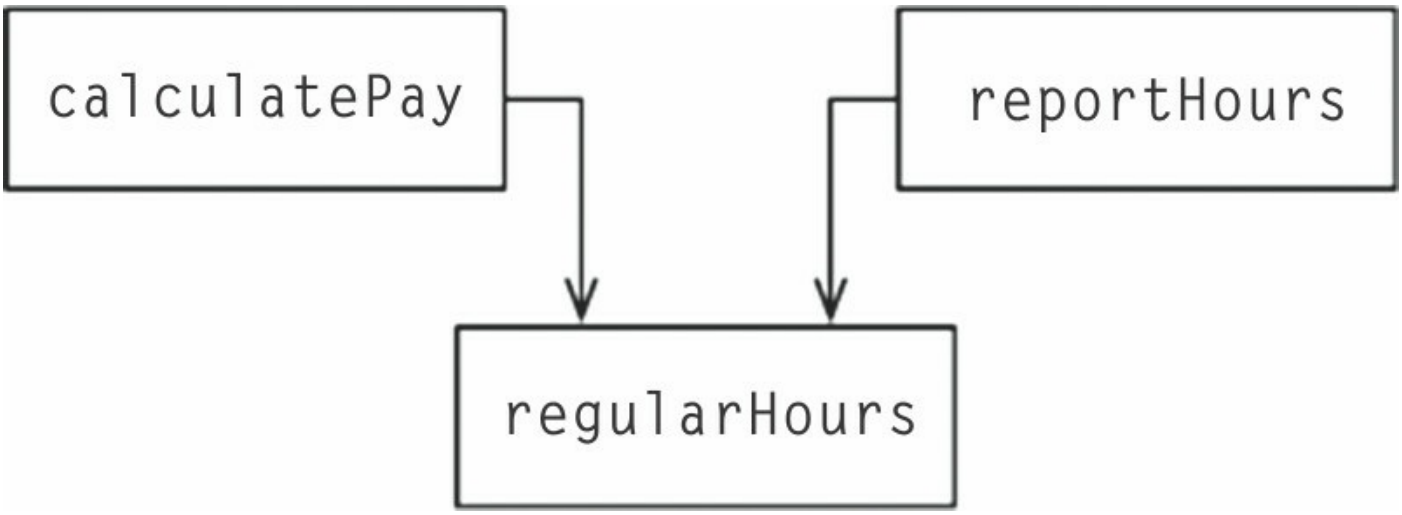


Abb. 7.2: Geteilter Code

Angenommen, das CFO-Team entscheidet nun, dass die Art und Weise der Berechnung der unbezahlten Überstunden optimiert werden muss. Im Gegensatz dazu möchte das COO-Team in der Personalabteilung diese Optimierung jedoch nicht, weil sie die unbezahlten Überstunden für einen anderen Zweck nutzen.

Ein Softwareentwickler wird beauftragt, die Anpassung vorzunehmen und stellt fest, dass die Funktion `regularHours()` praktischerweise von der `calculatePay()`-Methode aufgerufen wird. Allerdings fällt diesem Entwickler nicht auf, dass sie darüber hinaus auch von der Funktion `reportHours()` aufgerufen wird.

Er nimmt also die angeforderte Anpassung vor und testet sie sorgfältig. Das CFO-Team aus der Buchhaltung validiert, dass die neue Funktion wie gewünscht funktioniert, und das System wird deployt.

Allerdings hat das COO-Team keine Kenntnis davon. Die Personalabteilung verwendet weiterhin die Berichte, die über die Funktion `reportHours()` generiert werden – doch diese enthalten nun falsche Zahlen. Schließlich wird das Problem aufgedeckt und der COO ist stinksauer, weil ihn die verfälschten Daten Millionen von Euro seines Budgets kosten.

Wir alle haben solche Situationen schon selbst erlebt. Probleme wie dieses treten auf, weil Code, von dem verschiedene Akteure abhängig sind, in zu große Nähe zueinander rückt. Deshalb besagt das SRP, dass *Code, von dem verschiedene Akteure abhängig sind, separiert werden muss*.

7.2 Symptom 2: Merges

Es ist nicht allzu schwer nachzuvollziehen, dass das *Mergen*, also das »Abgleichen und Zusammenführen« unterschiedlicher Versionen von Quelldateien, die viele verschiedene Methoden enthalten, gängige Praxis ist. Das gilt insbesondere dann, wenn die betreffenden Methoden für diverse Akteure verantwortlich sind.

Nehmen wir beispielsweise an, die Datenbankadministratoren des CTO-Teams beschließen, dass eine simple schematische Änderung an der `Employee`-Tabelle der Datenbank vorgenommen werden soll. Außerdem entscheiden die Personalmitarbeiter des COO-Teams, dass die Formatierung der Stundenberichte modifiziert werden muss.

Zwei verschiedene Programmierer, womöglich von zwei unterschiedlichen Teams, prüfen nun die `Employee`-Klasse und beginnen, Änderungen daran vorzunehmen. Aber leider kollidieren ihre jeweiligen Anpassungen – die Folge ist dann ein Merge.

Wie Sie sicher schon wissen, sind Merges eine riskante Angelegenheit. Die Tools, die uns heutzutage zur Verfügung stehen, sind ziemlich gut, dennoch ist keins von ihnen jedem Merge-Fall gewachsen – im Endeffekt bleibt immer ein Restrisiko.

In unserem Beispiel bedeutet der Merge sowohl für den CTO als auch für den COO ein Risiko. Und es ist auch nicht ausgeschlossen, dass der CFO ebenfalls davon betroffen wäre.

Es gibt noch viele weitere Symptome, die wir untersuchen könnten, sie alle haben jedoch gemeinsam, dass sie mit mehreren Leuten zu tun haben, die dieselbe Quelldatei aus unterschiedlichen Gründen ändern.

Deshalb noch einmal: Der richtige Weg, um dieses Problem zu vermeiden, ist *die Separierung von Code, der von verschiedenen Akteuren genutzt wird*.

7.3 Lösungen

Zur Lösung dieser Problematik stehen zahlreiche Möglichkeiten zur Verfügung, wobei in allen Fällen die Funktionen in unterschiedliche Klassen verschoben werden.

Der vielleicht offensichtlichste Lösungsweg besteht darin, die Daten von den Funktionen zu trennen. In unserem Beispiel haben alle drei Klassen Zugriff auf die Klasse `EmployeeData`, die eine einfache Datenstruktur ohne Methoden enthält (siehe Abbildung 7.3). Jede dieser Klassen hält lediglich den für ihre spezifische Funktion notwendigen Quellcode vor. Es ist den drei Klassen nicht erlaubt, Kenntnis voneinander zu haben – so werden versehentliche Duplizierungen vermieden.

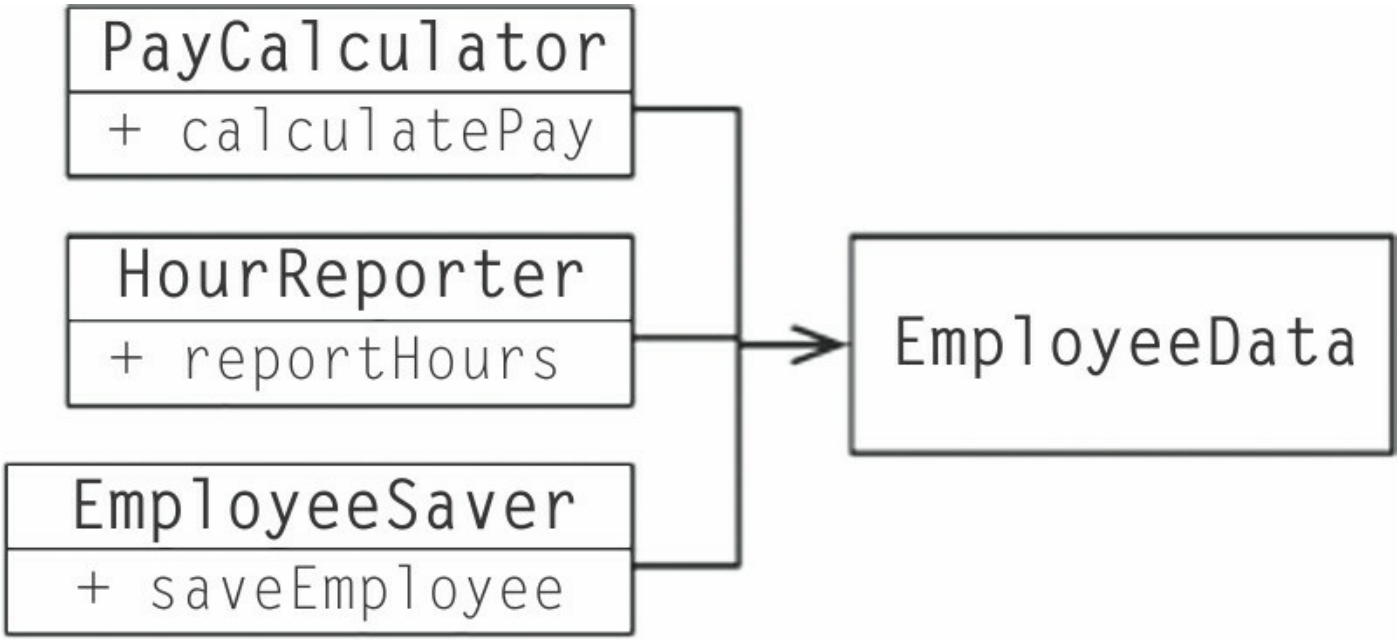


Abb. 7.3: Die drei Klassen haben keine Kenntnis voneinander.

Der Nachteil dieser Lösung ist, dass die Entwickler nun drei Klassen haben, die sie instanziieren und nachverfolgen müssen. Ein gebräuchlicher Ausweg aus diesem Dilemma ist die Verwendung des Patterns *Facade (Fassade)* (siehe Abbildung 7.4).

Die Klasse `EmployeeFacade` enthält nur sehr wenig Code. Sie ist für die Instanziierung und Delegierung der Klassen mitsamt den darin enthaltenen Funktionen verantwortlich.

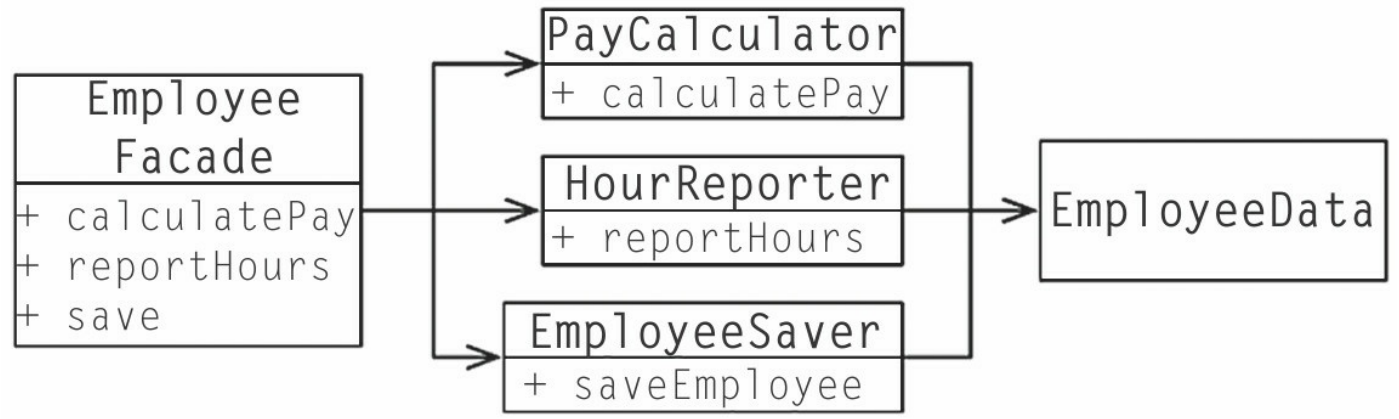


Abb. 7.4: Das Pattern *Facade (Fassade)*

Manche Softwareentwickler bevorzugen es, die wichtigsten Geschäftsregeln näher bei den Daten zu halten. Das kann dadurch realisiert werden, dass die wichtigste Methode in der ursprünglichen `Employee`-Klasse beibehalten und diese dann als *Facade (Fassade)* für untergeordnete Funktionen genutzt wird (siehe Abbildung 7.5).

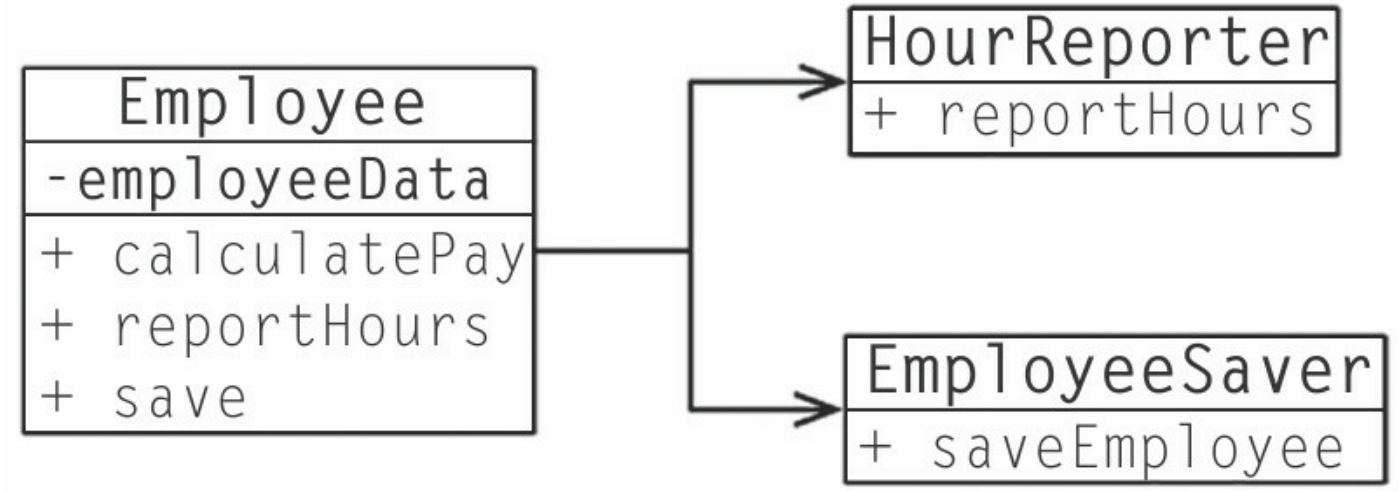


Abb. 7.5: Die wichtigste Methode wird in der ursprünglichen `Employee`-Klasse belassen und als *Facade (Fassade)* für untergeordnete Funktionen genutzt.

Nun könnten Sie gegen diese Lösungswege einwenden, dass jede Klasse nur eine Funktion enthalten würde – doch das dürfte wohl eher nicht zutreffen. Die Anzahl der zur Berechnung der Gehälter, für die Erstellung eines Berichts oder zum Speichern der Daten benötigten Funktionen ist in allen diesen Fällen sehr wahrscheinlich hoch. Jede dieser Klassen würde zahlreiche *private* Methoden beinhalten.

Alle Klassen, die solch eine Familie von Methoden enthalten, bilden einen *Scope* bzw. Anwendungsbereich. Außerhalb dieses Bereichs ist die Existenz der privaten Mitglieder der Familie nicht bekannt.

7.4 Fazit

Beim *Single-Responsibility-Prinzip* geht es um Funktionen und Klassen – darüber hinaus tritt es in einer abgewandelten Form aber auch auf zwei weiteren Ebenen in Erscheinung: Auf der Ebene der Komponenten wird es zum *Common-Closure-Prinzip*, auf der Ebene der Architektur wird es zum *Axis-of-Change-Modell* zur Errichtung architektonischer Grenzen. Diese Konzepte werden in den nachfolgenden Kapiteln betrachtet.

---

[1] *Agile Software Development, Principles, Patterns, and Practices*, Robert C. Martin, Prentice Hall, 2002.

Kapitel 8

OCP: Das Open-Closed-Prinzip

Toasted OATS
Toasted OATS
KOHNKE

Das *Open-Closed-Prinzip (OCP)* wurde im Jahr 1988 von Bertrand Meyer formuliert. [1] Es besagt:

> *Eine Softwareentität sollte offen für Erweiterungen, aber zugleich auch geschlossen gegenüber Modifikationen sein.*

Mit anderen Worten: Das Verhalten einer Softwareentität sollte erweiterbar sein, ohne dass sie modifiziert werden muss.

Dies ist ohne Frage die grundlegendste Motivation für das Studium der Softwarearchitektur. Es besteht kein Zweifel, dass die Architekten eines Softwaresystems dann, wenn einfache Erweiterungsanforderungen massive Änderungseingriffe an der Software erforderlich machen, in spektakulärer Weise versagt haben.

Die meisten Studenten des Fachbereichs Softwaredesign erkennen das OCP als ein Prinzip an, das sie beim Entwerfen von Klassen und Modulen anleitet. Bezieht man dieses Prinzip jedoch auch auf der Ebene der architektonischen Komponenten mit ein, gewinnt es sogar noch größere Bedeutung.

Das folgende Gedankenexperiment wird dies deutlich machen.

8.1 Ein Gedankenexperiment

Stellen Sie sich für einen Moment vor, Sie hätten ein System vor sich, dass eine Finanzübersicht auf einer Website ausgibt. Die Daten auf der Seite sind scrollbar und negative Zahlen werden in Rot ausgegeben.

Nun wollen die Stakeholder, dass dieselben Informationen in Berichtsform auf einem Schwarz-Weiß-Drucker ausgegeben werden. Alle Druckseiten sollen mit entsprechenden Titel- und Fußzeilen sowie Spaltenüberschriften ausgestattet sein, negative Zahlen sollen in Klammern gesetzt werden.

Es steht außer Frage, dass zu diesem Zweck neuer Code geschrieben werden muss. Aber wie viel von dem alten Code muss geändert werden?

Eine gute Softwarearchitektur würde den Umfang des zu ändernden Codes auf das absolute Minimum beschränken – idealerweise auf null.

Nur wie? Durch die saubere Trennung der Bestandteile, die sich aus unterschiedlichen Gründen ändern (*Single-Responsibility-Prinzip*), und die ebenso saubere Organisation

der Abhängigkeiten dieser Bestandteile (*Dependency-Inversion-Prinzip*).

Durch die Anwendung des SRPs könnte man zu dem in Abbildung 8.1 dargestellten Datenflussdiagramm gelangen. Im Rahmen eines Analysevorgangs werden die Finanzdaten untersucht und druckbare Daten erstellt, die dann von den beiden *Reporter*-Prozessen entsprechend formatiert werden.

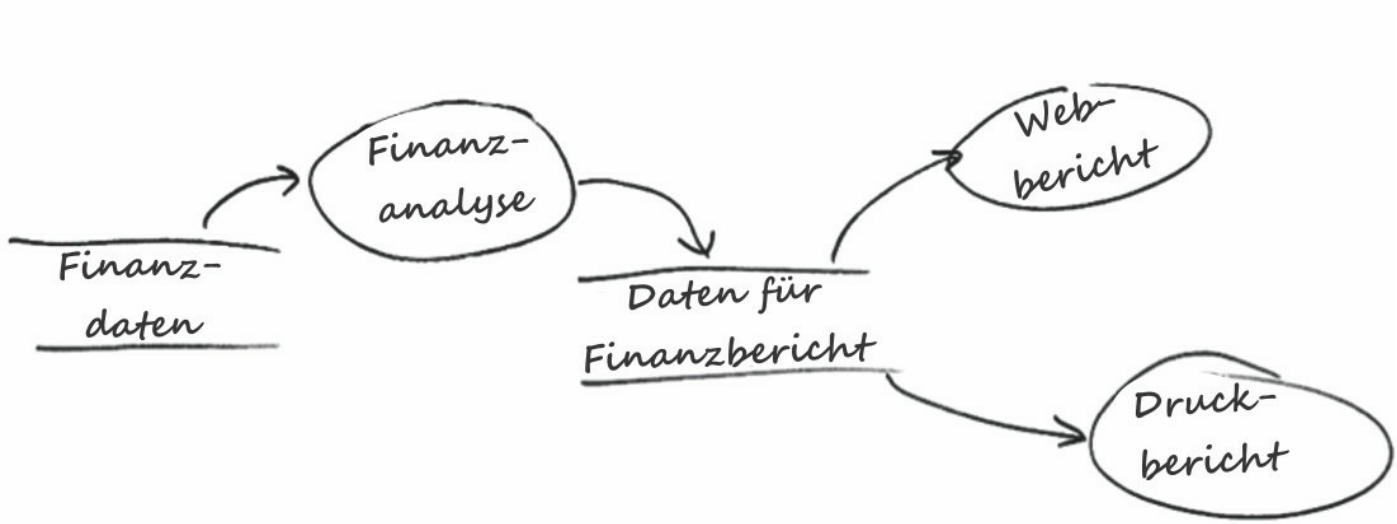


Abb. 8.1: Anwendung des SRPs

Die wesentliche Erkenntnis ist hier, dass die Generierung des Berichts zwei separate Verantwortlichkeiten umfasst: die Berechnung der berichteten Daten und die Präsentation dieser Daten in einer web- und druckergeeigneten Form.

Nachdem diese Trennung erfolgt ist, müssen die Quellcode-Abhängigkeiten organisiert werden, damit gewährleistet ist, dass Modifikationen an einer dieser Verantwortlichkeiten keine Veränderungen an der anderen nach sich ziehen. Die Neuorganisation soll außerdem eine Erweiterbarkeit des Verhaltens sicherstellen, ohne dass Modifikationen rückgängig gemacht werden müssen.

Dies wird erreicht, indem der Prozess in Klassen aufgeteilt wird und diese Klassen wiederum in Komponenten gegliedert werden, wie durch die doppelten Linien in Abbildung 8.2 dargestellt. Die Komponente oben links ist in diesem Fall der *Controller*. Oben rechts befindet sich der *Interaktor*. Im unteren rechten Bereich ist die *Datenbank* angesiedelt. Und schließlich sind unten links vier Komponenten angeordnet, die die *Ausgaben* (`Presenters`) und die *Ansichten* (`Views`) repräsentieren.

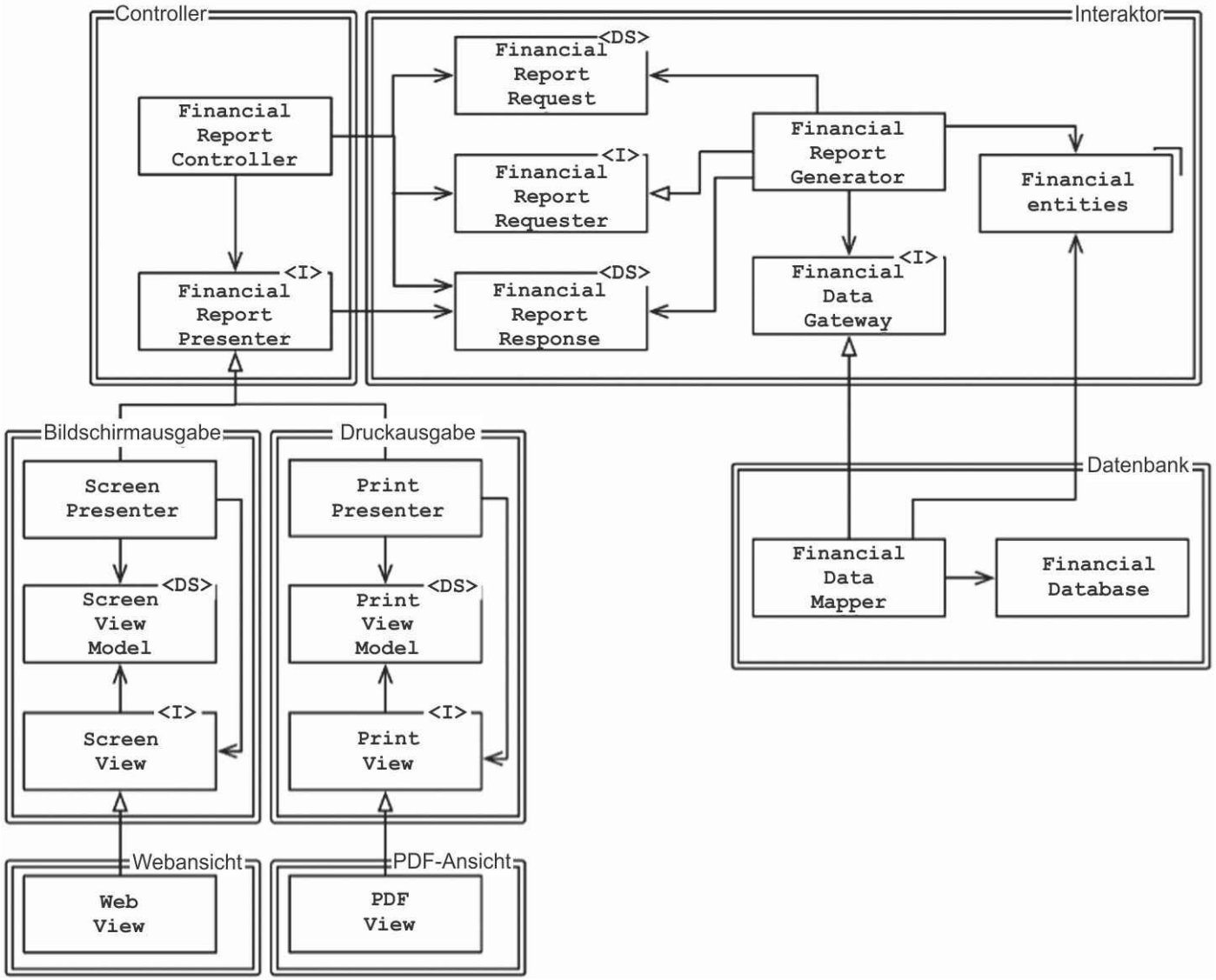


Abb. 8.2: Aufteilung des Prozesses in Klassen und Gliederung der Klassen in Komponenten

Bei den mit `<I>` markierten Klassen handelt es sich um Schnittstellen (*Interfaces*), Datenstrukturen sind mit `<DS>` gekennzeichnet. Offene Pfeilspitzen bedeuten eine *Nutzung* der Beziehungen, geschlossene Pfeilspitzen stehen für *Implementierungs-* oder *Vererbungsbeziehungen*.

Als Erstes ist hier zu erkennen, dass sich alle Abhängigkeiten auf den *Quellcode* beziehen. Ein Pfeil, der von Klasse A auf Klasse B zeigt, bedeutet, dass im Quellcode der Klasse A die Bezeichnung der Klasse B erwähnt wird, während die Klasse B keinerlei Hinweis auf Klasse A enthält. Dementsprechend lässt sich in Abbildung 8.2 ablesen, dass `FinancialDataMapper` durch eine Implementierungsbeziehung Kenntnis von `FinancialDataGateway` hat, `FinancialGateway` aber andererseits keine Kenntnis von `FinancialDataMapper` besitzt.

Die nächste Auffälligkeit ist, dass alle doppelten Linien *immer nur in eine Richtung* gekreuzt werden – soll heißen: Alle Komponentenbeziehungen sind unidirektional, wie in dem Komponentendiagramm in Abbildung 8.3 zu sehen. Diese Pfeile zeigen

auf die Komponenten, die es vor Modifikationen zu schützen gilt.

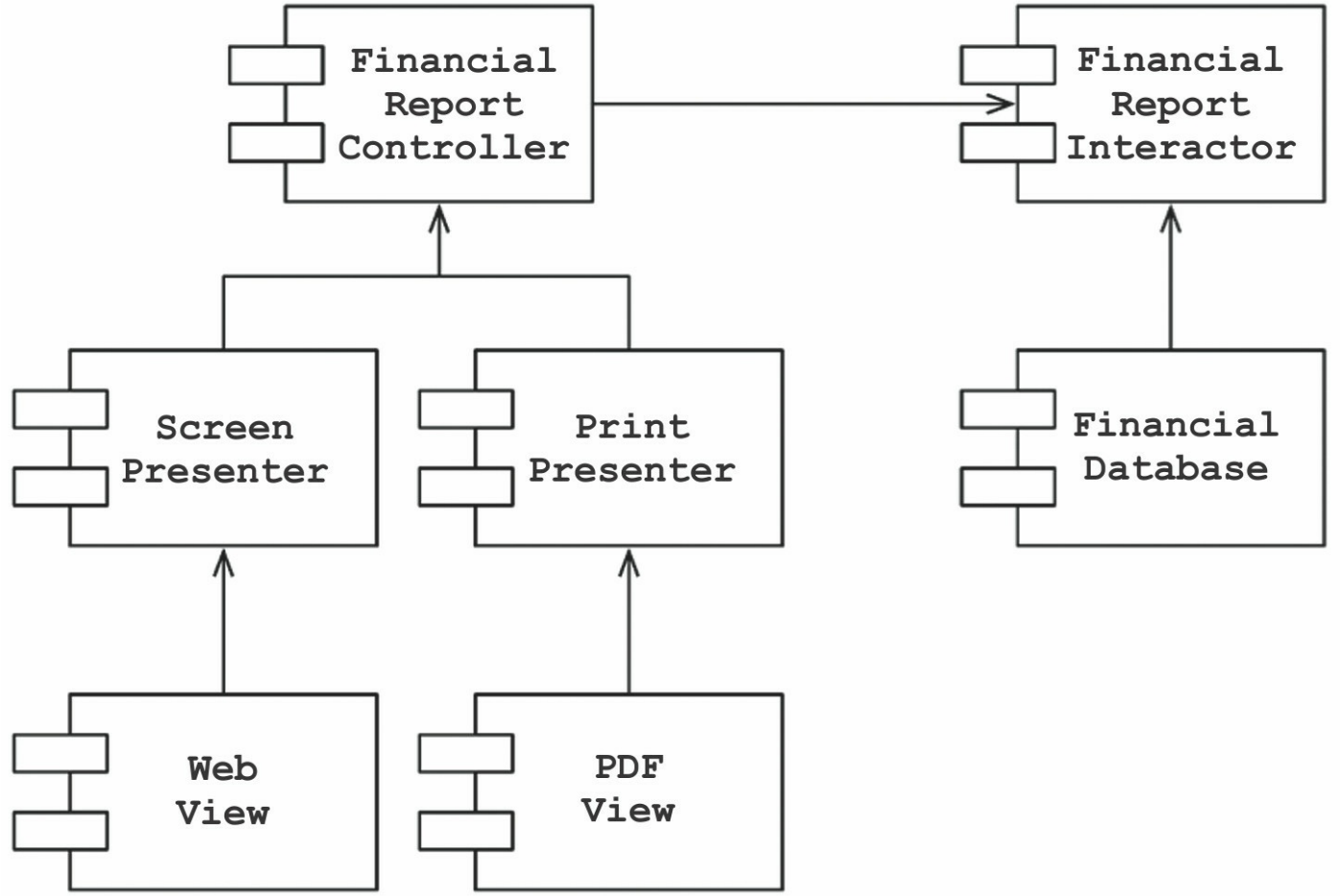


Abb. 8.3: Die Komponentenbeziehungen sind unidirektional.

Noch mal: Wenn Komponente A vor Modifikationen in Komponente B geschützt werden soll, dann sollte Komponente B von Komponente A abhängig sein.

Der `Controller` soll gegen Änderungen an den `Presenters` abgesichert werden. Diese wiederum sollen gegen Änderungen an den `Views` abgesichert werden. Und schließlich soll der `Interactor` gegen Änderungen an ... nun ja, an *allem* gesichert werden.

Der `Interactor` befindet sich in der Position, die dem OCP am ehesten entspricht. Modifikationen an der Datenbank oder dem `Controller` oder den `Presenters` oder den `Views` haben keinen Einfluss auf den `Interactor`.

Doch warum sollte sich der `Interactor` in einer derart privilegierten Position befinden? Weil er die Geschäftsregeln enthält. Er beherbergt die übergeordneten Richtlinien der Anwendung. Alle anderen Komponenten kümmern sich um periphere Belange – der `Interactor` hingegen ist für das zentrale Anliegen zuständig.

Wenngleich der `Controller` in einer peripheren Beziehung zum `Interactor` steht, ist er für die `Presenters` und `Views` dennoch ein zentrales Anliegen. Und während die

`Presenters` für den `Controller` eine periphere Rolle spielen können, sind sie für die `View` eine zentrale Angelegenheit.

Beachten Sie, inwiefern dies eine Art von Schutzhierarchie auf »Ebenengrundlage« erzeugt. Dabei stellen die `Interactors` das Konzept auf der höchsten Ebene dar, das heißt, sie sind am besten geschützt. Die `Views` gehören zu den Konzepten der untersten Ebene und sind damit am wenigsten geschützt. Währenddessen befinden sich die `Presenters` auf einer höheren Ebene als die `Views`, aber unterhalb der Ebene der `Controllers` oder des `Interactors`.

Dies entspricht der Funktionsweise des OCPs auf der architektonischen Ebene. Architekten trennen die Funktionalität auf der Grundlage dessen, wie, warum und wann sie sich ändert, und organisieren die so unterteilte Funktionalität dann in eine Komponentenhierarchie. Die auf den oberen Ebenen angeordneten Komponenten in dieser Hierarchie sind gegen Modifikationen, die an den untergeordneten Komponenten vorgenommen werden, geschützt.

8.2 Richtungssteuerung

Wenn Sie angesichts des zuvor aufgezeigten Klassendesigns etwas verunsichert sind, dann schauen Sie es sich noch einmal in Ruhe an. Die Komplexität dieser schematischen Darstellung soll größtenteils sicherstellen, dass die Abhängigkeiten zwischen den Komponenten in die richtige Richtung weisen.

Die Schnittstelle `FinancialDataGateway` zwischen den Klassen `FinancialReportGenerator` und `FinancialDataMapper` ist z.B. dafür gedacht, die Abhängigkeit, die andernfalls von der `Interaktor`-Komponente auf die `Database`-Komponente gezeigt hätte, zu invertieren. Dasselbe gilt auch für die Schnittstelle `FinancialReportPresenter` und die beiden `View`-Schnittstellen.

8.3 Information Hiding

Die Schnittstelle `FinancialReportRequester` dient einem anderen Zweck. Sie soll den `FinancialReportController` davor bewahren, zu viel Kenntnis von den Interna des `Interactors` zu erlangen. Wäre diese Schnittstelle dort nicht vorhanden, würde der Controller in transitiver Abhängigkeit zu `FinancialEntities` stehen.

Transitive Abhängigkeiten stellen einen Verstoß gegen das allgemeine Prinzip dar, dass Softwareentitäten nicht von Faktoren abhängig sein sollten, die sie nicht unmittelbar nutzen. Dieses Prinzip wird bei der Erörterung des *Interface-Segregation-*

und des *Common-Reuse-Prinzips* noch einmal zur Sprache kommen.

Obwohl also die oberste Priorität dem Schutz des `Interactors` gegen Modifikationen am `Controller` gilt, darf der Schutz des `Controllers` gegenüber Modifikationen am `Interactor` durch das Verbergen von dessen Interna ebenfalls nicht vernachlässigt werden.

8.4 Fazit

Das OCP ist eine der treibenden Kräfte hinter der Architektur von Systemen. Die Zielsetzung lautet dabei, das System so zu gestalten, dass es leicht zu erweitern ist, ohne einen zu hohen Einfluss durch Modifikationen zuzulassen. Das wird erreicht, indem man das System in Komponenten unterteilt und diese dann in einer Abhängigkeitshierarchie anordnet, die Komponenten höherer Ebenen vor Änderungen an untergeordneten Komponenten schützt.

---

[1] *Object Oriented Software Construction*, Bertrand Meyer, Prentice Hall, 1988, S. 23. (1990 in deutscher Fassung unter dem Titel »Objektorientierte Softwareentwicklung« beim Hanser Verlag veröffentlicht.)

Kapitel 9

LSP: Das Liskov’sche Substitutionsprinzip

Im Jahr 1988 formulierte Barbara Liskov folgende Definitionsmethode für Subtypen:

> *Was hier erreicht werden sollte, ist etwas wie die folgende Substitutionseigenschaft: Wenn für jedes Objekt o1 vom Typ S ein Objekt o2 vom Typ T existiert, sodass für alle Programme P, die in T definiert sind, das Verhalten von P unverändert bleibt, wenn o1 für o2 substituiert wird, dann ist S ein Subtyp von T.*[1]

Um dieses als *Liskov’sches Substitutionsprinzip* (**L**iskov **S**ubstitution **P**rinciple, *LSP*) bezeichnete Konzept besser verstehen zu können, werden im Folgenden einige passende Beispiele aufgezeigt.

9.1 Gesteuerte Nutzung der Vererbung

Nehmen wir eine Klasse namens `License` an, wie in Abbildung 9.1 dargestellt. Diese Klasse hat eine Methode mit Namen `calcFee()`, die von der Anwendung `Billing` aufgerufen wird. `License` besitzt darüber hinaus zwei »Subtypen« bzw. Unterklassen: `PersonalLicense` und `BusinessLicense`. Sie verwenden unterschiedliche Algorithmen, um die Lizenzgebühr zu ermitteln.

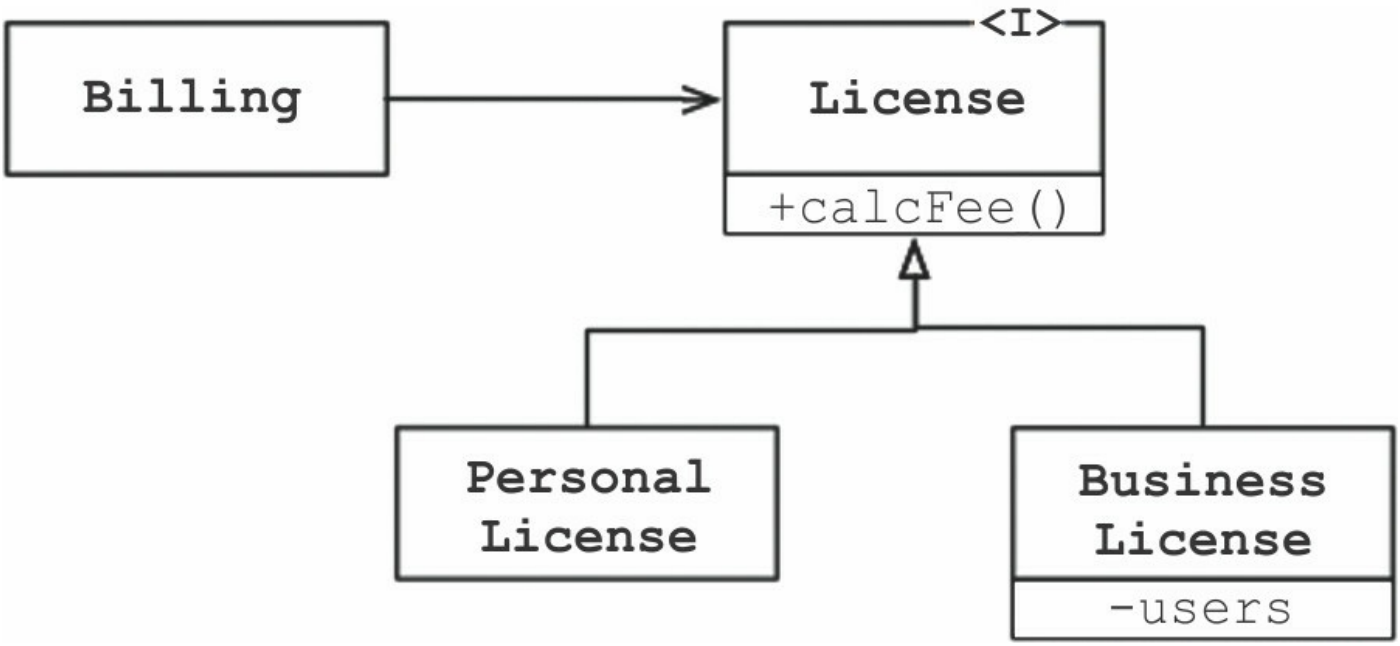


Abb. 9.1: Die Klasse `License` und ihre Ableitungen sind LSP-konform.

Dieses Design entspricht dem LSP, weil das Verhalten der `Billing`-Anwendung in keiner Weise davon abhängig ist, welche der beiden Subtypen sie nutzt. Aus Sicht der Klasse `License` sind beide Unterklassen substituier- bzw. ersetzbar.

9.2 Das Quadrat-Rechteck-Problem

Ein häufig diskutiertes Beispiel für einen Verstoß gegen das LSP ist das berühmte (oder, je nach Sichtweise, auch berüchtigte) *Quadrat-Rechteck-Problem* (siehe Abbildung 9.2).

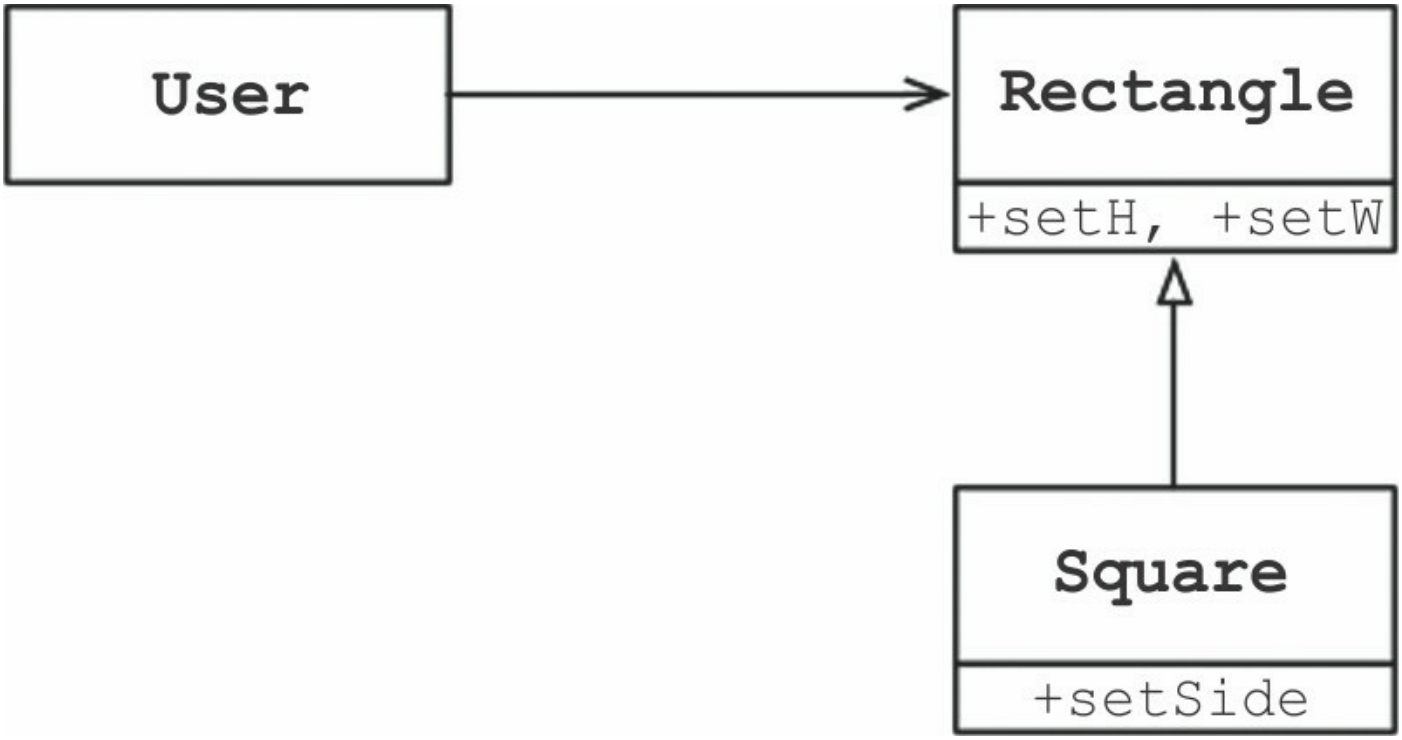


Abb. 9.2: Das berühmte Quadrat-Rechteck-Problem

In diesem Beispiel ist `Square` (Quadrat) keine echte Unterklasse von `Rectangle` (Rechteck), weil Höhe und Breite der Klasse `Rectangle` unabhängig voneinander veränderbar sind. Im Gegensatz dazu müssen Höhe und Breite der Klasse `Square` zusammen verändert werden. Da `User` davon ausgeht, mit `Rectangle` zu kommunizieren, kann dies leicht zu Verwirrung führen. Der folgende Code zeigt, warum:

```
Rectangle r = ...
r.setW(5);
r.setH(2);
assert(r.area() == 10);
```

Wenn der `...`-Code ein Quadrat – `Square` – erzeugen würde, würde die Assertion fehlschlagen.

Die einzige Möglichkeit, diese Art von Verstoß gegen das LSP zu verhindern, besteht darin, den Typ `User` um Mechanismen (wie beispielsweise eine `if`-Anweisung) zu ergänzen, die erkennen, ob ein `Rectangle` eigentlich ein `Square` ist. Da das Verhalten der Klasse `User` von den Klassen abhängig ist, die sie benutzt, sind diese nicht substituierbar.

9.3 Das LSP und die Softwarearchitektur

In der Anfangszeit der objektorientierten Revolution wurde das LSP, wie in den vorangehenden Abschnitten beschrieben, als eine Methode betrachtet, um die Nutzung der Vererbung zu steuern. Im Laufe der Jahre hat sich das LSP jedoch zu einem breiter gefassten Prinzip des Softwaredesigns entwickelt, das auch Schnittstellen (*Interfaces*) und Implementierungen betrifft.

Diese Interfaces können verschiedener Art sein. So könnte z.B. eine Schnittstelle im Java-Stil vorliegen, die von mehreren Klassen implementiert wird. Ebenso könnten diverse Ruby-Klassen vorhanden sein, die dieselben Methodensignaturen gemeinsam haben. Oder es existiert ein Satz von Services, die alle dieselbe *REST-Schnittstelle* (**Re**presentational **S**tate **T**ransfer, zu Deutsch etwa »repräsentative Zustandsübermittlung«) ansprechen.

In all diesen und weiteren Situationen ist das LSP anzuwenden, weil es User gibt, die von klar definierten Schnittstellen und der Substituierbarkeit der Implementierungen dieser Interfaces abhängig sind.

Die beste Methode, um sich das LSP aus architektonischer Sicht zu vergegenwärtigen, ist, sich vorzustellen, was mit der Architektur eines Systems passiert, wenn gegen das Prinzip verstoßen wird.

9.4 Beispiel für einen Verstoß gegen das LSP

Angenommen, Sie würden einen Aggregator für mehrere Taxizentraldienste errichten. Die Kunden nutzen Ihre Website, um das für sie am besten geeignete Taxi zu finden, unabhängig von dem jeweiligen Taxiunternehmen. Sobald der Kunde sich entschieden hat, schickt das System das gewählte Taxi unter Zuhilfenahme eines »RESTful Service« los.

Gehen wir weiter davon aus, dass der URI (**U**niform **R**esource **I**dentifier) für den »RESTful Service« ein Bestandteil der in der Fahrerdatenbank enthaltenen Informationen ist. Nachdem das System einen für den Kunden geeigneten Fahrer ermittelt hat, ruft es diesen URI aus dem Fahrerdatensatz ab und verwendet ihn dann, um den Fahrer zuzuteilen.

Der Fahrer Bob hat in diesem Beispiel einen URI, der folgendermaßen aussieht:

```
purplecab.com/driver/Bob
```

Das System hängt die Zuteilungsinformation an diesen URI an und versendet sie mittels einer PUT-Methode, etwa so:

```
purplecab.com/driver/Bob
      /pickupAddress/24 Maple St.
      /pickupTime/153
      /destination/ORD
```

Dies würde definitiv bedeuten, dass alle Zuteilungsdienste für alle Unternehmen mit derselben REST-Schnittstelle konform sein müssen. Sie müssten die Felder `pickupAddress`, `pickupTime` und `destination` identisch behandeln.

Unterstellen wir als Nächstes, das Taxiunternehmen Acme hätte ein paar Programmierer engagiert, die die Anforderungsspezifikationen nicht sorgfältig genug durchgelesen haben – und die Bezeichnung des Feldes `destination` einfach in `dest` abkürzen. Nun ist Acme aber mittlerweile das größte Taxiunternehmen in der Umgebung und die Ex-Frau des Acme-CEOs ist inzwischen mit dem CEO Ihres Unternehmens verheiratet und ... nun ja, Sie verstehen schon. Was würde mit der Architektur Ihres Systems geschehen?

Zweifellos müssten Sie einen Sonderfall ergänzen: Die Zuteilungsanfrage für die Acme-Fahrer müsste so konstruiert werden, dass sie einen anderen Regelsatz verwendet als für alle übrigen Fahrer.

Die einfachste Möglichkeit, dies zu erreichen, bestünde darin, das Modul, das den Zuteilungsbefehl erstellt, um eine `if`-Anweisung zu ergänzen:

```
if (driver.getDispatchUri().startsWith("acme.com"))...
```

Aber natürlich würde kein Softwarearchitekt, der sein Geld wert ist, eine solche Konstruktion in seinem System zulassen, denn: Durch das Einfügen des Begriffs `acme` in den Code stünden Tür und Tor für alle möglichen fatalen und mysteriösen Fehler offen, ganz zu schweigen von Sicherheitsverstößen.

Was würde nun zum Beispiel passieren, wenn Acme noch erfolgreicher arbeiten und das Taxiunternehmen Purple aufkaufen würde? Und wenn das fusionierte Unternehmen sowohl die Markennamen als auch die zugehörigen Websites getrennt beibehalten, aber die Systeme der beiden ursprünglichen Unternehmen zusammenführen würde? Müsste dann eine weitere `if`-Anweisung für »purple« hinzugefügt werden?

Der Softwarearchitekt wäre gezwungen, das System von derartigen Bugs zu isolieren, indem er eine Art Modul zur Erstellung von Zuteilungsbefehlen anlegt, das von einer durch den Zuteilungs-URI verschlüsselten Konfigurationsdatenbank gesteuert würde. Die Konfigurationsdaten könnten dann ungefähr so aussehen:

URI	Zuteilungsformat
Acme.com	/pickupAddress/%s/pickupTime/%s/dest/%s
.	/pickupAddress/%s/pickupTime/%s/destination/%s

Damit müsste der Softwarearchitekt einen bedeutenden und komplexen Mechanismus ergänzen, um die Tatsache zu berücksichtigen, dass nicht alle Schnittstellen der »RESTful Services« substituierbar sind.

9.5 Fazit

Das LSP kann und sollte auf die Architekturebene erweitert werden, denn: Schon ein einfacher Verstoß gegen die Substituierbarkeit kann die Verunreinigung der Systemarchitektur mit einer erheblichen Anzahl an zusätzlichen Mechanismen zur Folge haben.

---

[1] Barbara Liskov, *Data Abstraction and Hierarchy*, SIGPLAN Notices 23, 5 (Mai 1988).

Kapitel 10
ISP: Das Interface-Segregation-Prinzip

Das *Interface-Segregation-Prinzip (ISP)* verdankt seinen Namen dem in Abbildung 10.1 dargestellten schematischen Aufbau:

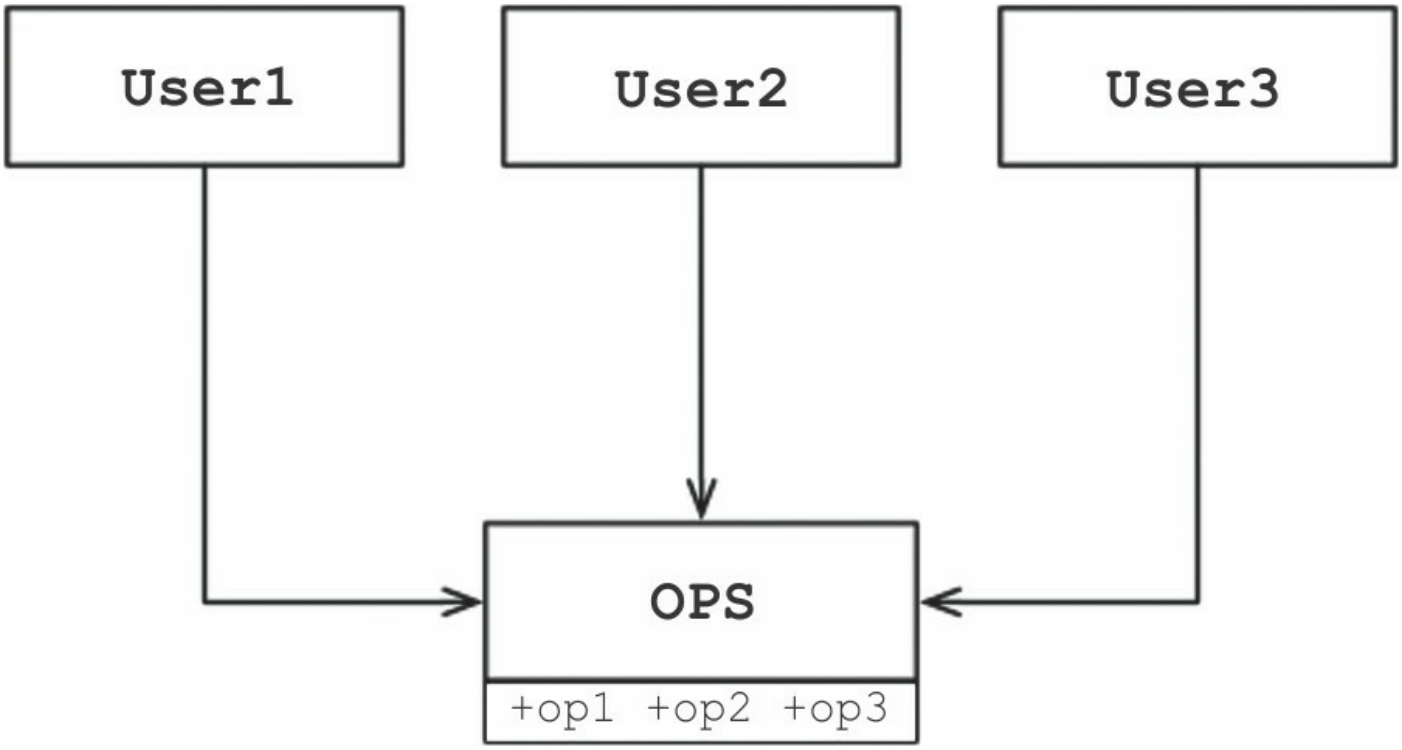


Abb. 10.1: Das Interface-Segregation-Prinzip

In der hier skizzierten Konstellation gibt es mehrere User, die die Operationen der `OPS`-Klasse verwenden. Nehmen wir an, dass `User1` nur `op1`, `User2` nur `op2` und `User3` nur `op3` nutzt.

Außerdem gehen wir davon aus, dass `OPS` in einer Sprache wie Java geschrieben ist. In diesem Fall wäre der Quellcode von `User1` unbeabsichtigterweise von `op2` und `op3` abhängig, auch wenn Letztere nicht von der Klasse aufgerufen werden. Diese Abhängigkeit bedeutet allerdings, dass eine Änderung des Quellcodes von `op2` in `OPS` eine Neukompilierung und ein erneutes Deployment von `User1` erzwingt – obwohl faktisch überhaupt keine Modifikationen an irgendwelchen Modulen, mit denen diese Klasse zu tun hat, vorgenommen wurden.

Das hier vorliegende Problem lässt sich durch die Aufspaltung der Operationen in Schnittstellen lösen, wie in Abbildung 10.2 gezeigt.

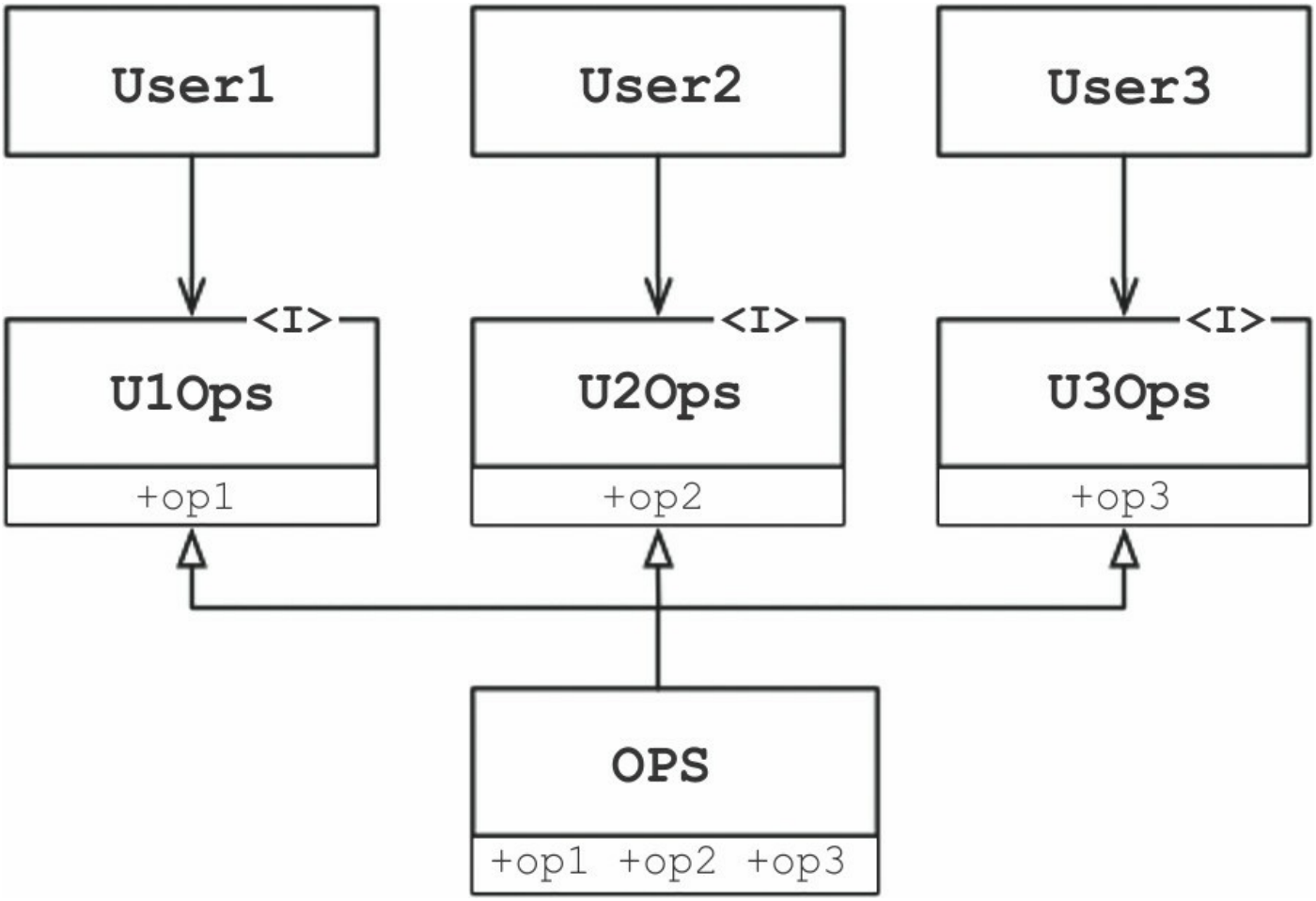


Abb. 10.2: Aufgespaltene Operationen

Würde diese Lösung in einer statisch typisierten Sprache wie Java implementiert, dann wäre der Quellcode von `User1` von den Klassen `U1Ops` und `op1` abhängig, aber nicht von `OPS`. Dementsprechend hätten Modifikationen an der Klasse `OPS`, mit der `User1` nichts zu tun hat, auch keine Neukompilierung und kein erneutes Deployment von `User1` zur Folge.

10.1 Das ISP und die Programmiersprachen

Die vorerwähnte Beschreibung steht zweifellos in hohem Maße mit der jeweils verwendeten Programmiersprache in Zusammenhang. Sprachen mit einer statischen Typisierung wie Java zwingen die Programmierer zur Erstellung von Deklarationen, die die User mithilfe der Anweisungen `import`, `use` oder auch `include` referenzieren müssen – und ebendiese im Quellcode fest verankerten Deklarationen erzeugen Quellcode-Abhängigkeiten, die eine Neukompilierung und ein erneutes Deployment erforderlich machen.

In dynamisch typisierten Sprachen wie Ruby und Python existieren solche Deklarationen im Quellcode nicht. Hier werden sie zur Laufzeit abgeleitet, sodass

keine Quellcode-Abhängigkeiten auftreten, die eine Neukompilierung und ein erneutes Deployment erzwingen würden. Das ist der Hauptgrund dafür, dass Sprachen mit einer dynamischen Typisierung Systeme ermöglichen, die flexibler und weniger eng gekoppelt sind als statisch typisierte Sprachen.

Und diese Tatsache könnte nun den Schluss zulassen, dass es sich beim ISP eher um ein sprachbezogenes als ein architekturbezogenes Problem handelt.

10.2 Das ISP und die Softwarearchitektur

Wenn Sie jedoch noch mal einen Schritt zurückgehen und die Kernmotivationen des ISPs in Ruhe betrachten, werden Sie ein tiefer liegendes Problem erkennen. Generell ist es nicht zu empfehlen, sich auf Module zu verlassen, die mehr enthalten, als benötigt wird. Das gilt natürlich auf jeden Fall für Quellcode-Abhängigkeiten, die überflüssige Neukompilierungen und erneute Deployments erzwingen – darüber hinaus trifft es jedoch auch auf eine viel höhere Architekturebene zu.

Betrachten wir hierzu einmal den Beispielfall eines Softwarearchitekten, der an einem System *S* arbeitet und nun ein bestimmtes Framework namens *F* darin einbinden möchte. Die Autoren von *F* haben das Framework allerdings an eine bestimmte Datenbank gebunden, nennen wir sie *D*. Damit ist *S* von *F* abhängig, das wiederum in Abhängigkeit zu *D* steht (siehe Abbildung 10.3).

Abb. 10.3: Eine problematische Architektur

Einmal angenommen, die Datenbank *D* enthielte außerdem Features, die in *F* nicht genutzt werden und für *S* daher überhaupt nicht relevant sind. Modifikationen an den Features in der Datenbank *D* könnten dann durchaus ein erneutes Deployment von *F* erzwingen – und somit auch von *S*. Schlimmer noch: Ein Versagen eines der Features innerhalb der Datenbank *D* könnte in dieser Situation sogar Fehlfunktionen im Framework *F* und dem System *S* auslösen.

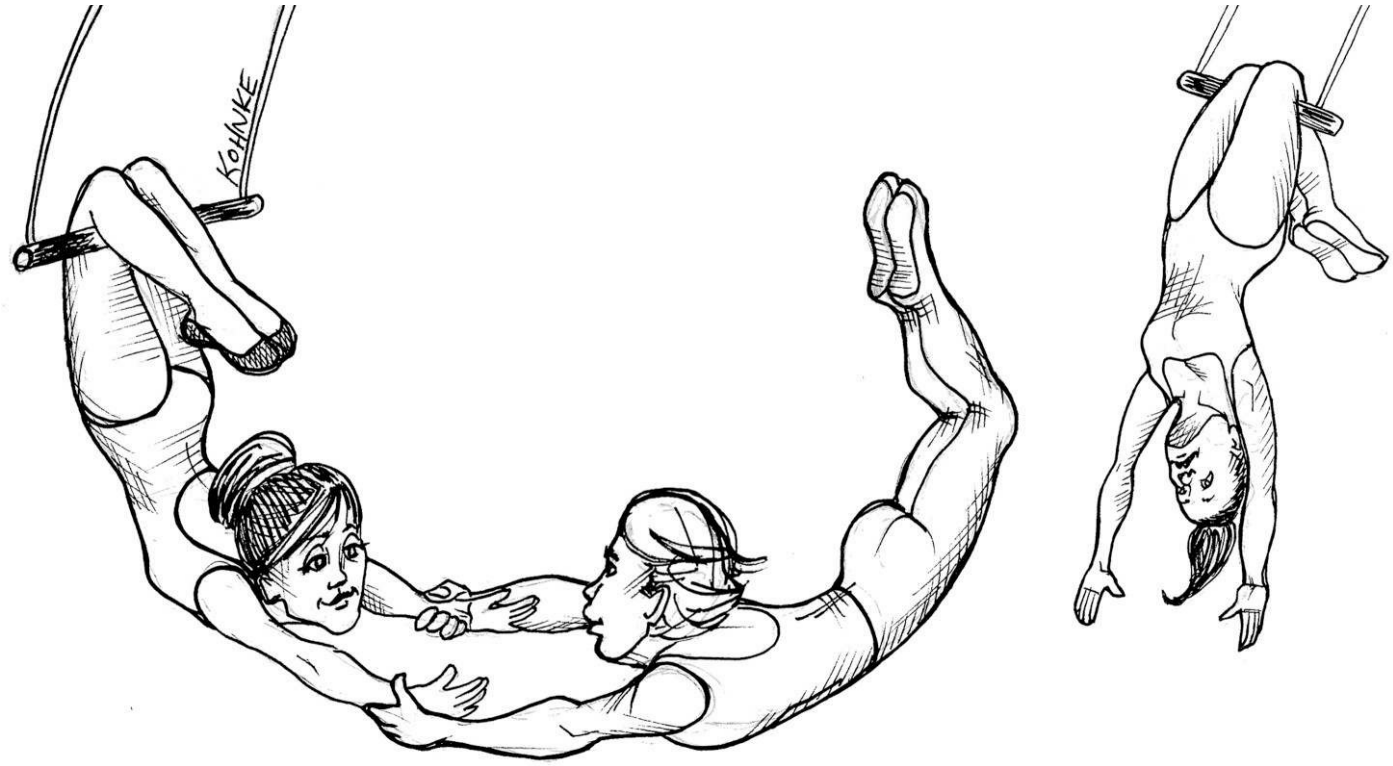
10.3 Fazit

Die vorgenannten Beispiele zeigen eindrucksvoll auf, dass die Abhängigkeit von

Modulen, die unnötige Lasten mit sich schleppen, völlig unerwartete Probleme verursachen kann.

Dieses Konzept wird im Zusammenhang mit dem *Common-Reuse-Prinzip* im Kapitel 13 »Komponentenkohäsion« noch eingehender vorgestellt.

Kapitel 11
DIP: Das Dependency-Inversion-Prinzip



Das *Dependency-Inversion-Prinzip (DIP)* macht deutlich, dass Systeme, in denen sich Quellcode-Abhängigkeiten ausschließlich auf Abstraktionen beziehen statt auf Konkretionen, am flexibelsten sind.

In einer statisch typisierten Sprache wie Java bedeutet dies, dass sich die Anweisungen `use`, `import` und `include` nur auf Quellmodule beziehen sollten, die Schnittstellen, abstrakte Klassen oder andere Formen von abstrakten Deklarationen enthalten. Dagegen sollten keinerlei Abhängigkeiten zu konkreten Modulen bestehen.

Dasselbe gilt auch für dynamisch typisierte Sprachen wie Ruby und Python: Quellcode-Abhängigkeiten sollten auch hier keine konkreten Module referenzieren. Allerdings ist es in diesen Programmiersprachen ein bisschen schwieriger zu definieren, was ein konkretes Modul ist – grundsätzlich ist darunter jedes Modul zu verstehen, dessen aufgerufene Funktionen implementiert werden.

Dieses Konzept als Regel aufzufassen, ist jedoch sicherlich illusorisch, weil

Softwaresysteme natürlich auch von vielen konkreten Entitäten abhängig sein müssen. Beispielsweise ist die `String`-Klasse in Java konkret – und zu versuchen, sie zwingend abstrakt zu gestalten, wäre unrealistisch. Die Quellcode-Abhängigkeit von dem konkreten `java.lang.string`-Objekt kann und sollte nicht vermieden werden.

Hinzu kommt, dass die Klasse `String` sehr stabil ist: Sie wird nur selten modifiziert und zudem engmaschig kontrolliert. Im Allgemeinen brauchen sich Programmierer und Softwarearchitekten keine Gedanken über häufige, willkürliche Änderungen an dieser Klasse zu machen.

Deshalb neigen wir dazu, den stabilen Hintergrund des Betriebssystems und der Plattformentitäten im Zusammenhang mit dem DIP zu ignorieren. Wir tolerieren konkrete Abhängigkeiten dieser Art, weil wir uns darauf verlassen können, dass sie sich nicht ändern.

Anders verhält sich das hingegen mit konkreten, flüchtigen `volatile`-Elementen unseres Systems – hier gilt es, Abhängigkeiten zu vermeiden, denn diese Module werden aktiv entwickelt und unterliegen daher häufigen Anpassungen und Modifikationen.

11.1 Stabile Abstraktionen

Jede Änderung an einer abstrakten Schnittstelle hat zugleich auch eine Modifikation ihrer konkreten Implementierungen zur Folge. Umgekehrt gehen Anpassungen konkreter Implementierungen dagegen nicht immer mit Änderungen an den Schnittstellen einher, von denen sie implementiert wurden – in der Praxis ist das sogar eher unüblich. Insofern sind Schnittstellen weniger flüchtig als Implementierungen.

Tatsächlich achten gute Softwaredesigner und -architekten bei ihrer Arbeit sehr darauf, die Flüchtigkeit von Schnittstellen zu reduzieren. Zu diesem Zweck suchen sie nach Möglichkeiten, um die Funktionalität der Implementierungen ohne die Notwendigkeit entsprechender Anpassungen der Schnittstellen erweitern zu können. Diese Vorgehensweise wird als »Software Design 101« bezeichnet.

Im Umkehrschluss bedeutet dies, dass Softwarearchitekturen genau dann stabil sind, wenn sie Abhängigkeiten von flüchtigen Konkretionen vermeiden und stattdessen dem Einsatz stabiler abstrakter Schnittstellen den Vorzug geben. Und das lässt sich durch die Anwendung einer Reihe von sehr spezifischen Programmierpraktiken erreichen:

- **Referenzieren Sie keine flüchtigen konkreten Klassen.** Verwenden Sie stattdessen Referenzierungen auf abstrakte Schnittstellen. Diese Richtlinie gilt für alle Programmiersprachen, ob statisch oder dynamisch typisiert. Zudem unterwirft sie die Erzeugung von Objekten strikten Einschränkungen und forciert

allgemein den Einsatz des Design Patterns *Abstract Factory (Abstrakte Fabrik)*.

- **Nutzen Sie keine Ableitungen von flüchtigen konkreten Klassen.** Das ist im Grunde genommen eine logische Folge der vorherigen Richtlinie, soll aber an dieser Stelle dennoch besondere Erwähnung finden. In statisch typisierten Sprachen stellt die Vererbung die stärkste und strikteste aller Quellcode-Beziehungen dar – und dementsprechend sollte sie auch mit besonderer Sorgfalt angewendet werden. In dynamisch typisierten Sprachen ist die Vererbung ein geringfügigeres Problem, es existiert aber immer noch eine Abhängigkeit – und eine vorsichtige Vorgehensweise ist stets die klügste Wahl.

- **Überschreiben Sie keine konkreten Funktionen.** Konkrete Funktionen bedingen häufig Quellcode-Abhängigkeiten. Wenn Sie diese Funktionen überschreiben, heben Sie die zugehörigen Abhängigkeiten damit nicht auf – faktisch *vererben* Sie sie. Um solche Abhängigkeiten zu verwalten, sollten Sie die betreffende Funktion abstrakt gestalten und mehrere Implementierungen erzeugen.

- **Erwähnen Sie konkrete und flüchtige Elemente zu keinem Zeitpunkt namentlich.** Hierbei handelt es sich im Kern um eine nochmalige Darlegung des Prinzips selbst.

11.2 Factories

Um den vorerwähnten Richtlinien zu entsprechen, muss der Erzeugung flüchtiger konkreter Objekte eine besondere Behandlung zukommen. Diese Vorsichtsmaßnahme ist deshalb erforderlich, weil zur Erstellung eines Objekts in buchstäblich allen Programmiersprachen eine Quellcode-Abhängigkeit von der konkreten Definition des betreffenden Objekts notwendig ist. In den meisten objektorientierten Sprachen wie Java würde man zur Verwaltung dieser unerwünschten Abhängigkeit das Design Pattern *Abstract Factory (Abstrakte Fabrik)* anwenden.

Die schematische Darstellung in Abbildung 11.1 veranschaulicht den strukturellen Aufbau dieses Konzepts. Die Klasse `Application` nutzt `ConcreteImp` über die `Service`-Schnittstelle. Allerdings muss `Application` irgendwie Instanzen von `ConcreteImpl` erzeugen. Um dies zu erreichen, ohne eine Quellcode-Abhängigkeit von `ConcreteImpl` herzustellen, ruft `Application` die Methode `makeSvc` der Schnittstelle `ServiceFactory` auf. Diese Methode wird wiederum von der Klasse `ServiceFactoryImpl` implementiert, die von `ServiceFactory` abgeleitet ist. Und diese Implementierung instanziiert schließlich die Klasse `ConcreteImpl` und gibt sie als `Service` zurück.

Die geschwungene Linie in Abbildung 11.1 kennzeichnet eine architektonische Grenze: Sie trennt das Abstrakte von dem Konkreten. Alle Quellcode-Abhängigkeiten kreuzen diese Linie und zeigen in dieselbe Richtung – zur abstrakten Seite.

Sie unterteilt das System in zwei Komponenten: Die eine ist abstrakt, die andere konkret. Die abstrakte Komponente enthält alle übergeordneten Geschäftsregeln der Anwendung. Die konkrete Komponente enthält alle Implementierungsdetails, die diese Geschäftsregeln manipulieren.

Beachten Sie hierbei, dass der Kontrollfluss die geschwungene Linie in die entgegengesetzte Richtung der Quellcode-Abhängigkeiten kreuzt. Letztere sind gegen den Kontrollfluss invertiert – weshalb dieses Prinzip als *Dependency Inversion* (zu Deutsch »Abhängigkeitsumkehr«) bezeichnet wird.

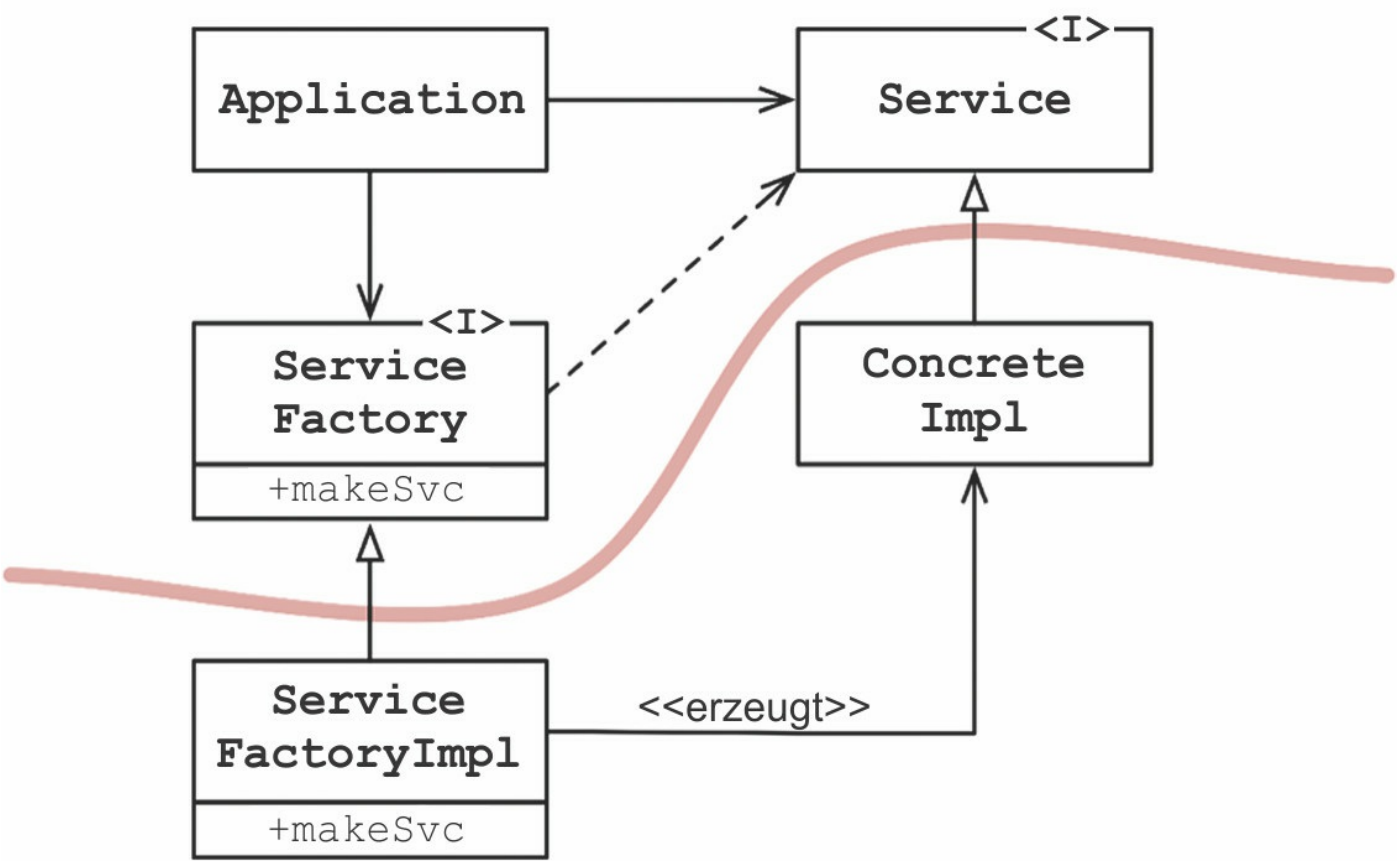


Abb. 11.1: Anwendung des Design Patterns *Abstract Factory (Abstrakte Fabrik)* zur Verwaltung der Abhängigkeit

11.3 Konkrete Komponenten

Die konkrete Komponente in Abbildung 11.1 enthält eine einzelne Abhängigkeit und verstößt damit gegen das DIP. Das ist ein typischer Fall. DIP-Verstöße lassen sich nicht vollständig beseitigen, aber sie können in einer kleineren Anzahl von konkreten

Komponenten gesammelt und vom Rest des Systems getrennt gehalten werden.

Die meisten Systeme enthalten mindestens eine solche konkrete Komponente – die oftmals mit `main`[1] bezeichnet ist, weil sie die `main`-Funktion umfasst. In dem in Abbildung 11.1 gezeigten Beispielfall würde die `main`-Funktion die Klasse `ServiceFactoryImpl` instanziieren und diese Instanz dann in einer globalen Variablen vom Typ `ServiceFactory` platzieren. Der Zugriff von `Application` auf die Factory würde somit über diese globale Variable erfolgen.

11.4 Fazit

Wenn wir uns im weiteren Verlauf dieses Buches den höheren Architekturprinzipien zuwenden, werden wir dem DIP immer wieder begegnen. Es ist das offensichtlichste Organisationsprinzip der noch folgenden Architekturschemata. Die geschwungene Linie in Abbildung 11.1 beschreibt die architektonischen Grenzen in den späteren Kapiteln. Die Art und Weise, in der die Abhängigkeiten diese Linie in eine Richtung kreuzen – hin zu den abstrakteren Entitäten –, wird später noch in einer neuen Regel erfasst, die als *Abhängigkeitsregel* (engl. *Dependency Rule*) bezeichnet wird.

---

[1] Mit anderen Worten die Funktion, die beim ersten Start der Anwendung vom Betriebssystem aufgerufen wird.

Teil IV
Komponentenprinzipien

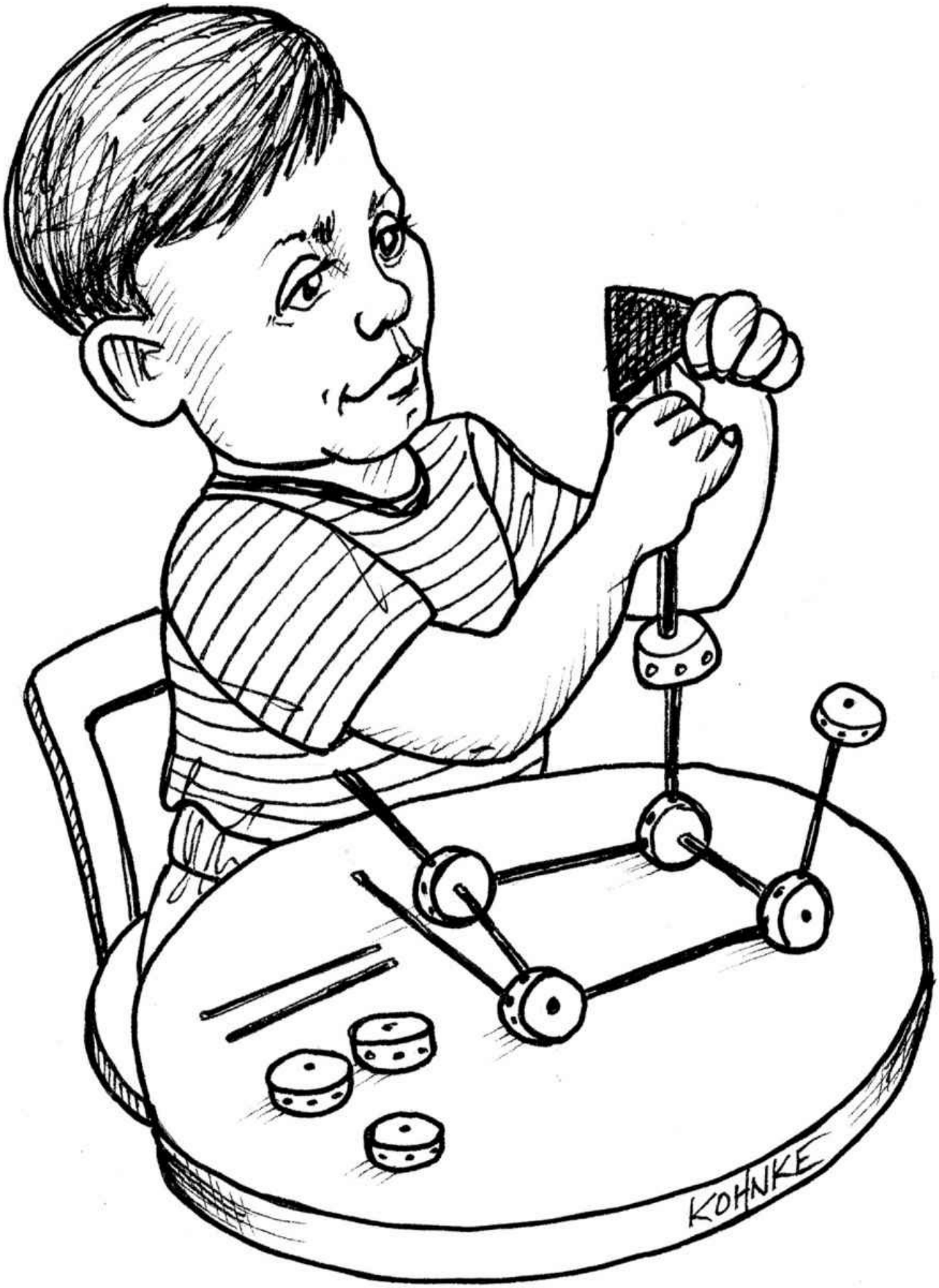
In diesem Teil:

Während die SOLID-Prinzipien darlegen, wie wir die Bausteine zu Wänden und Räumen arrangieren müssen, zeigen uns die Komponentenprinzipien auf, wie die Räume in Gebäuden anzuordnen sind. Große Softwaresysteme bauen wie große Gebäude ebenfalls auf kleineren Komponenten auf.

In diesem vierten Teil des Buches werden Sie erfahren, was genau Softwarekomponenten sind, aus welchen Elementen sie erstellt werden und wie sie dann in Systemen zusammengesetzt werden sollten.

Kapitel 12

Komponenten


KOHNKE

Komponenten sind Deployment-Einheiten. Sie repräsentieren die kleinsten Entitäten, die als Teil eines Systems deployt werden können. In Java sind dies `.jar`-Dateien, in Ruby `.gem`-Dateien, und in .NET DLLs. In kompilierten Programmiersprachen handelt es sich um Aggregationen von Binärdateien, in interpretierten Sprachen um Aggregationen von Quelldateien. In allen Programmiersprachen stellen sie sozusagen das Granulat dar, aus dem das Deployment entsteht.

Komponenten können zu einer einzelnen ausführbaren Datei verknüpft werden. Oder sie können zu einem einzelnen Archiv zusammengefügt werden, etwa zu einer `.war`-Datei. Oder sie werden unabhängig voneinander deployt, als separate, dynamisch zu ladende Plug-ins, etwa als `.jar`-, `.dll`- oder `.exe`-Dateien. Egal, wie sie letztlich in das Deployment eingebracht werden, behalten gut designte Komponenten immer die Fähigkeit, unabhängig voneinander deployt und somit auch unabhängig voneinander entwickelt zu werden.

12.1 Eine kurze Historie der Komponenten

In den Anfangsjahren der Softwareentwicklung legten die Programmierer den Speicherort und das Layout ihrer Programme selbst fest. Damals enthielten die ersten Codezeilen in einem Programm die `origin`-Anweisung, die die Adresse deklarierte, an der das Programm zu laden war.

Betrachten Sie dazu das folgende einfache PDP-8-Programm. Es besteht aus einer Subroutine namens `GETSTR`, die eine Tastatureingabe in Empfang nimmt und sie in einen Zwischenspeicher (*Buffer*) ablegt. Darüber hinaus enthält es zum besseren Verständnis des `GETSTR`-Konzepts auch ein kleines Programm zum Testen der Einheit.

```
            *200
            TLS
START,      CLA
            TAD BUFR
            JMS GETSTR
            CLA
            TAD BUFR
            JMS PUTSTR
            JMP START
BUFR,       3000

GETSTR,     0
            DCA PTR
NXTCH,      KSF
            JMP -1
            KRB
            DCA I PTR
            TAD I PTR
```

```
            AND K177
            ISZ PTR
            TAD MCR
            SZA
            JMP NXTCH

K177,       177
MCR,        -15
```

Beachten Sie den Befehl `*200` am Anfang dieses Programms. Hiermit wird der Compiler angewiesen, Code zu generieren, der an der Adresse 2008 geladen wird.

Diese Form der Programmierung ist den meisten Softwareentwicklern heute fremd. Inzwischen müssen sie sich nur selten darüber Gedanken machen, wo ein Programm in den Speicher des Computers geladen wird. Früher gehörte dies allerdings zu den ersten Entscheidungen, die ein Softwareentwickler zu treffen hatte, denn damals waren Programme nicht relativierbar.

Und wie wurde zu dieser Zeit eine Bibliotheksfunktion realisiert? Der vorstehende Code illustriert die damalige Herangehensweise. Die Programmierer integrierten den Quellcode der Bibliotheksfunktionen in den Code ihrer Anwendung und kompilierten beides als ein einziges Programm.[1] Die Bibliotheken wurden also im Quellcode beibehalten und nicht ausgelagert.

Das Problem bei diesem Ansatz war jedoch, dass die damaligen Geräte nur langsam arbeiteten und Speicher teuer war und deshalb nur in eingeschränktem Maße zur Verfügung stand. Die Compiler mussten den Quellcode mehrfach durchlaufen, die Speicherkapazität war aber zu gering, um den gesamten Quellcode vorzuhalten. Dementsprechend musste bei der Kompilierung mit den langsamen Geräten mehrere Male lesend auf den Quellcode zugegriffen werden.

Dieser Vorgang war ziemlich langwierig – und je umfangreicher die Funktionsbibliothek war, desto länger brauchte der Compiler. Das Kompilieren eines größeren Programms konnte durchaus mehrere Stunden dauern.

Um die Wartezeit zu verkürzen, trennten die Programmierer den Quellcode der Funktionsbibliothek daher von den Anwendungen. Sie kompilierten die Funktionsbibliothek separat und luden den Binärcode an einer bekannten Adresse – beispielsweise 20008. Außerdem erstellten sie eine Symboltabelle für die Funktionsbibliothek und kompilierten sie mit ihrem Anwendungscode. Zum Starten einer Anwendung luden sie dann zuerst die binäre Funktionsbibliothek[2] und anschließend die Anwendung. Die Speicherzuordnung sah damit wie in dem in Abbildung 12.1 gezeigten Layout aus.

Solange die Anwendung in den Adressbereich zwischen 00008 und 17778 passte, funktionierte das auch prima. Doch sobald eine Anwendung an einen Punkt gelangte,

an dem sie über den ihr zugewiesenen Bereich hinauswuchs, mussten die Programmierer sie in zwei Adressbereiche unterteilen und in der Funktionsbibliothek umherspringen (siehe Abbildung 12.2).

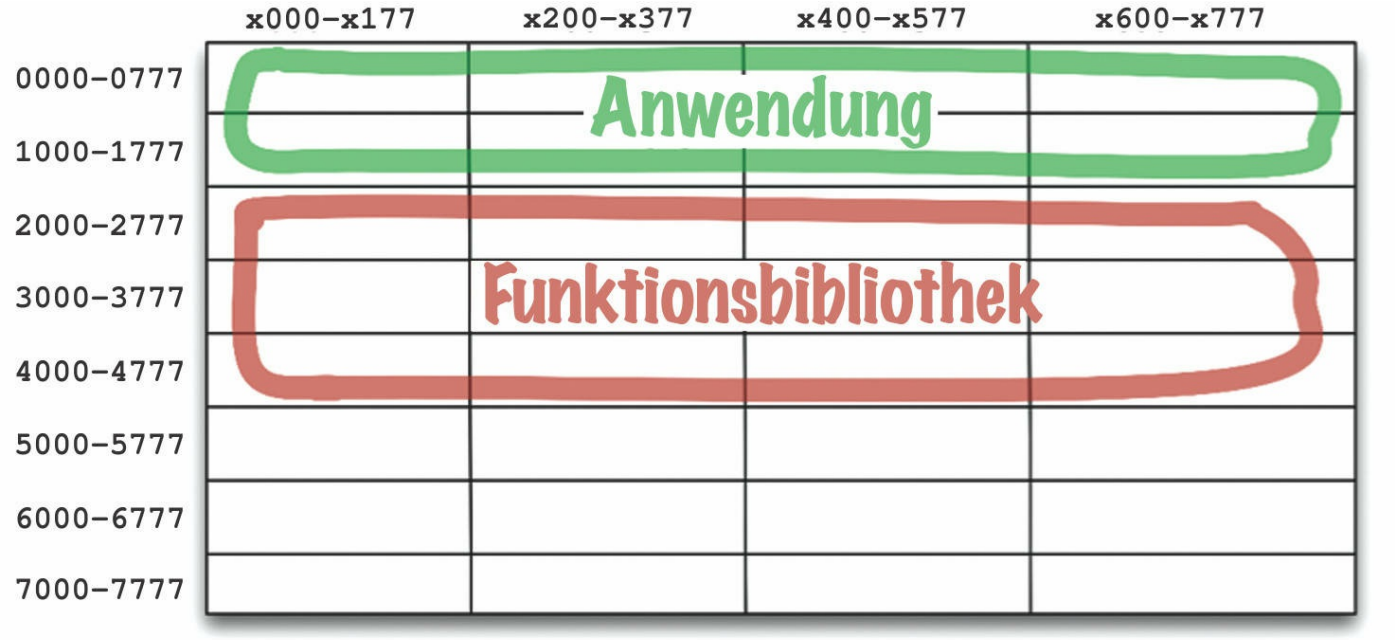


Abb. 12.1: Frühe Speicherzuordnungstabelle

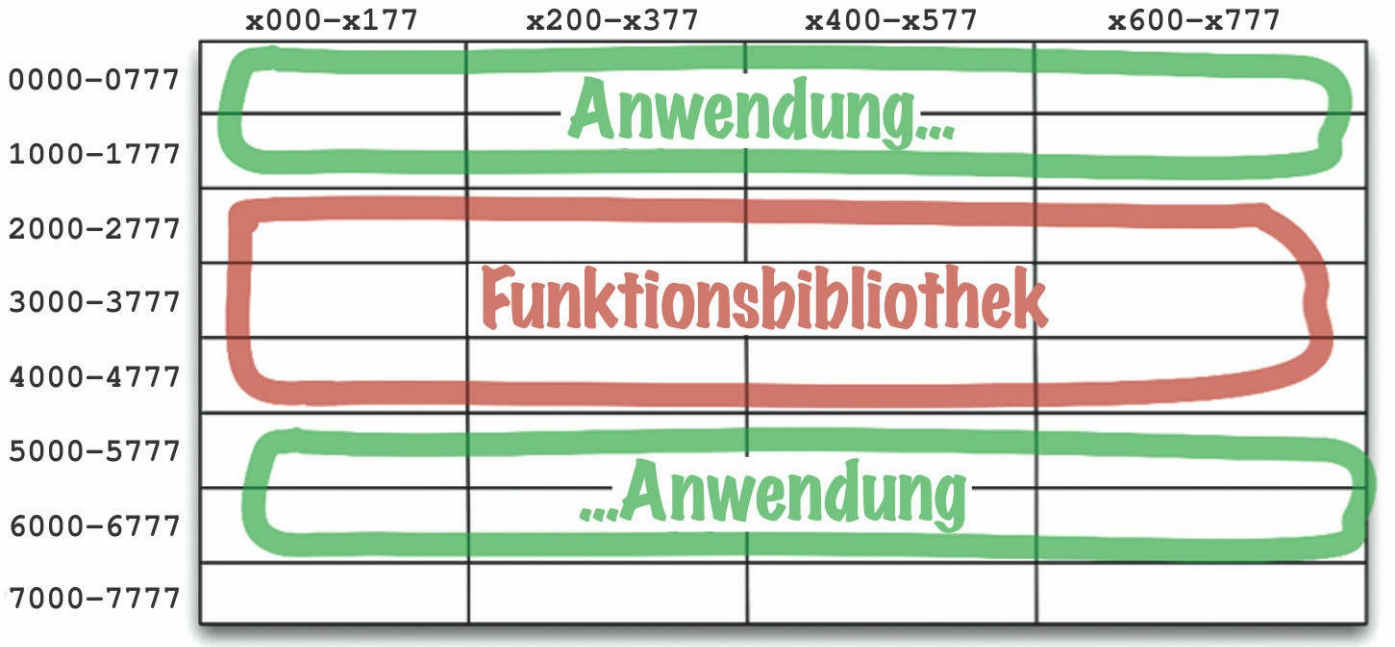


Abb. 12.2: Aufteilung einer Anwendung in zwei Adressbereiche

Natürlich war das auf Dauer kein haltbarer Zustand: Je mehr Funktionen die Programmierer in die Funktionsbibliothek ergänzten, desto eher wurde der Grenzbereich überschritten und sie mussten der Anwendung weiteren Speicherplatz zuweisen (in diesem Beispiel bis etwa 70008). Und so setzte sich diese Fragmentierung der Programme und Bibliotheken mit dem wachsenden

Speicherplatzbedarf zwangsläufig immer so fort.

Es musste also unbedingt etwas unternommen werden.

12.2 Relokatierbarkeit

Die Lösung waren *relokatierbare* (verlagerbare) Binärdateien. Das dahinterstehende Konzept war sehr simpel: Der Compiler wurde so modifiziert, dass er eine Binärcodeausgabe erzeugte, die von einem sogenannten »Smart Loader«, einem intelligenten Lader, im Arbeitsspeicher verlagert werden konnte. Dazu wurde ihm einfach mitgeteilt, wo er den relokatierbaren Code laden sollte. Der Code selbst wurde mit Flags gekennzeichnet, die dem Loader angaben, welche Teile der geladenen Daten angepasst werden mussten, um an der gewählten Adresse geladen zu werden. Im Allgemeinen bedeutete das einfach, dass jede Speicherreferenzadresse in dem Binärcode mit der entsprechenden Startadresse versehen werden musste.

Anschließend konnte der Programmierer den Loader anweisen, wo die Funktionsbibliothek und wo die Anwendung geladen werden sollte. Im Grunde genommen nahm er gleich mehrere Binäreingaben entgegen, lud sie dann hintereinander in den Arbeitsspeicher und verschob sie noch während des Ladevorgangs an ihre jeweiligen Standorte. Dadurch waren Softwareentwickler in der Lage, nur die Funktionen aufzurufen, die sie auch wirklich brauchten.

Zusätzlich wurde der Compiler auch dahin gehend modifiziert, dass er die Namen der Funktionen als Metadaten in dem relokatierbaren Binärcode hinterlegte. Sobald ein Programm eine Bibliotheksfunktion aufrief, übergab der Compiler deren Bezeichnung als *externe Referenz*. Und wenn ein Programm eine Bibliotheksfunktion definiert hatte, übermittelte der Compiler ihren Namen als *externe Definition*. Auf diese Weise konnte der Loader nach der Festlegung der Speicheradressen die externen Referenzen mit den externen Definitionen *verlinken*.

Damit war der *Binder* (auch »Bindelader« genannt) geboren.

12.3 Linker

Der Binder ermöglichte es den Programmierern, ihre Programme in separat kompilierbare und ladbare Segmente aufzuteilen. Solange relativ kleine Programme mit relativ kleinen Bibliotheken verlinkt werden mussten, funktionierte das sehr gut – doch als die Softwareentwickler in den späten 1960er- und frühen 1970er-Jahren ambitioniertere Ziele verfolgten, wurden auch ihre Programme deutlich

umfangreicher.

Und so erwiesen sich die Binder letztendlich als zu langsam. Funktionsbibliotheken wurden üblicherweise auf behäbigen Datenträgern wie etwa Magnetbändern gespeichert. Auch die damals verfügbaren Disketten arbeiteten nur recht schleppend. Doch die Binder waren auf diese unzureichenden Speichermedien angewiesen, um Dutzende, wenn nicht gar Hunderte von Binärbibliotheken einzulesen und die externen Referenzen aufzulösen. Je umfangreicher die Programme wurden und je mehr Funktionen sich in den Bibliotheken anhäuften, desto länger brauchten sie, um ein Programm zu laden – oftmals mehr als eine Stunde.

In der Folge ging man dazu über, den Lade- und den Verlinkungsvorgang in zwei getrennten Prozessen ablaufen zu lassen. Den langsameren Prozess – die Verlinkung – verlagerten die Programmierer in eine separate Anwendung, die als *Linker* bezeichnet wurde. Dieser gab eine verknüpfte relokatierbare Datei aus, die der relokatierende Programm-Loader sehr schnell einlesen konnte. Dadurch war es den Softwareentwicklern möglich, mithilfe des eher langsam arbeitenden Linkers eine ausführbare Datei vorzubereiten, die sie anschließend jederzeit schnell laden konnten.

Dann kamen die 1980er-Jahre. Programmierer schrieben nun in C oder einer anderen höheren Programmiersprache. Je ambitionierter sie waren, umso ambitionierter wurden auch ihre Programme. Inzwischen war es keineswegs mehr unüblich, dass Programme Hunderttausende von Codezeilen umfassten.

Die Quellmodule wurden aus `.c`-Dateien in `.o`-Dateien kompiliert und dann zwecks Erstellung von ausführbaren Dateien, die sich schnell laden ließen, an den Linker weitergeleitet. Das Kompilieren jedes einzelnen Moduls ging relativ schnell vonstatten, die Kompilierung *aller* Module dauerte allerdings eine Weile. Und der Linker brauchte im Anschluss noch länger. Dementsprechend stiegen die Verarbeitungszeiten in vielen Fällen erneut auf eine Stunde oder gar mehr an.

Es schien, als wären die Programmierer dazu verdammt, sich unentwegt im Kreis zu drehen. Sämtliche Modifikationen, die man in den 1960er-, 1970er- und 1980er-Jahren zur Beschleunigung des Workflows vorgenommen hatte, wurden immer wieder schon bald darauf durch die ehrgeizigen Ziele der Softwareentwickler und die Größe der von ihnen geschriebenen Programme ausgebremst. Sie konnten den stundenlangen Verarbeitungszeiten offenbar einfach nicht entgehen. Die Ladezeiten blieben zwar im Rahmen, bei der Dauer der Kompilier- und Verlinkungsprozesse tat sich dann aber der Flaschenhals auf.

Was die Programmierer da erlebten, war ein »Murphys Gesetz« der Computerprogrammierung:

Ein Programm wird so lange expandieren, bis es den verfügbaren Speicher füllt.

Murphy war jedoch nicht der einzige Unruhestifter in der Welt der Programmierung – denn auch ein gewisser Mr. Moore[3] trat nun auf den Plan. Und so kam es in den späten 1980er-Jahren zum Showdown zwischen diesen beiden, den Moore am Ende für sich entschied. Datenträger wurden kleiner und bedeutend schneller. Die Preise für Arbeitsspeicher fielen dermaßen ab, dass ein Großteil der bislang auf Disketten gespeicherten Daten nunmehr im RAM gecacht werden konnte. Und die Taktraten stiegen von 1 MHz auf 100 MHz.

Etwa Mitte der 1990er-Jahre begann der Zeitaufwand für die Verlinkung schneller zu sinken, als die Zielsetzungen der Softwareentwickler die Programme anwachsen ließen. In vielen Fällen reduzierte sich die Verlinkungszeit auf lediglich *Sekunden*. Und damit wurde das Konzept des Binders wieder interessant und lohnenswert.

Die Ära von Active X, *Shared Libraries* (dynamische Bibliotheken) sowie der ersten `.jar`-Dateien war angebrochen. Computer und Peripheriegeräte waren so schnell geworden, dass Verlinkungen wieder während der Ladezeit durchgeführt werden konnten. Die Programmierer waren nun in der Lage, mehrere `.jar`-Dateien oder auch mehrere Shared Libraries in Sekundenschnelle miteinander zu verknüpfen und das daraus resultierende Programm auszuführen. Und so entstand die Plug-in-basierte Komponentenarchitektur.

Inzwischen werden Anwendungen bereits routinemäßig mit `.jar`-Dateien oder DLLs oder Shared Libraries als Plug-ins ausgestattet. Wollen Sie beispielsweise eine Mod für *Minecraft* erstellen, dann brauchen Sie dazu einfach nur ihre eigenen `.jar`-Dateien in ein bestimmtes Verzeichnis einzufügen. Und wenn Sie die *ReSharper*-Erweiterung zu *Visual Studio* ergänzen möchten, fügen Sie einfach die entsprechenden DLLs zu dem Programm hinzu.

12.4 Fazit

Die in diesem Kapitel beschriebenen dynamisch verlinkten Dateien, die zur Laufzeit zusammengefügt werden können, sind Softwarekomponenten unserer Architekturen. Es mag zwar 50 Jahre gedauert haben, inzwischen sind wir aber an einem Punkt angekommen, an dem die Plug-in-basierte Komponentenarchitektur der gängige Standard sein kann – statt die Herkulesarbeit verrichten zu müssen, die dazu früher einmal erforderlich war.

---

[1] Mein erster Arbeitgeber bewahrte mehrere Dutzend Lochkartenstapel mit dem Quellcode von Subroutinen-Bibliotheken in einem Regal auf. Wenn man ein neues Programm schrieb, griff man sich einfach einen dieser Stapel und fügte ihn am Ende des eigenen Codes an.

[2] Tatsächlich nutzten die meisten alten Maschinen den Kernspeicher, der nach dem Herunterfahren des Computers nicht geleert wurde. Oftmals blieb die Funktionsbibliothek mehrere Tage lang geladen.

[3] Moore’sches Gesetz: Die Prozessorleistung, der Arbeitsspeicher und die Datendichte von Computern verdoppeln sich alle 18 Monate. Dieses Gesetz behielt von den 1950er-Jahren bis 2000 seine Gültigkeit, wurde dann, zumindest was die Taktraten anbelangt, jedoch ungültig.

Kapitel 13

Komponentenkohäsion

Welche Klassen gehören zu welchen Komponenten? Diese wichtige Entscheidung sollte grundsätzlich auf guten Grundsätzen der Softwareentwicklung basieren. Leider hat sich im Laufe der Jahre jedoch immer wieder gezeigt, dass sie oftmals lediglich rein kontextbezogen in Ad-hoc-Manier getroffen wird.

In diesem Kapitel werden die drei Prinzipien der *Komponentenkohäsion* betrachtet:

- **REP:** Das *Reuse-Release-Equivalence-Prinzip*
- **CCP:** Das *Common-Closure-Prinzip*
- **CRP:** Das *Common-Reuse-Prinzip*

13.1 REP: Das Reuse-Release-Equivalence-Prinzip

Die Granularität der Wiederverwendung ist die Granularität des Release.

Die letzten zehn Jahre brachten eine Vielfalt an Tools für das Modulmanagement hervor. *Maven*, *Leiningen* und *RVM* sind nur einige Beispiele, die zudem schnell an Bedeutung gewannen, weil zeitgleich auch eine riesige Anzahl von wiederverwendbaren Komponenten und Komponentenbibliotheken aufkam. Wir leben in einem Zeitalter der Softwarewiederverwendung – die Erfüllung eines der ältesten Versprechen des objektorientierten Modells.

Das *Reuse-Release-Equivalence-Prinzip* beschreibt einen Grundsatz, der im Grunde genommen – zumindest rückblickend – vollkommen selbstverständlich erscheint: Wer Softwarekomponenten wiederverwenden will, kann und wird erst dann dazu in der Lage sein, wenn diese Komponenten im Rahmen eines Release-Tracking-Prozesses erfasst und mit Versionsnummern versehen wurden.

Dieses Verfahren liegt nicht einfach nur darin begründet, dass sich ohne die Versionsnummern nicht feststellen lässt, ob alle wiederverwendeten Komponenten miteinander kompatibel sind. Vielmehr müssen die Softwareentwickler auch wissen, wann neue Releases erscheinen und welche Änderungen bzw. Neuerungen sie mit sich bringen werden.

Es ist nicht unüblich, dass Softwareentwickler, nachdem sie über ein neues Release in Kenntnis gesetzt wurden, wegen der in dieser nächsten Version vorzunehmenden Anpassungen beschließen, das alte Release weiterzuverwenden. Deshalb muss der Release-Tracking-Prozess entsprechende Meldungen und Release-Dokumentationen erzeugen, damit die User fundierte Entscheidungen dahin gehend treffen können, wann und ob sie das neue Release in ihre Systeme integrieren.

Aus Sicht des Softwaredesigns und der Softwarearchitektur bedeutet dies, dass die Klassen und Module, aus denen eine Komponente gebildet wird, einer kohärenten Gruppe angehören müssen. Die Komponente kann sich nicht einfach aus einem willkürlichen Sammelsurium von Klassen und Modulen zusammensetzen, sondern sie müssen allesamt einem jeweils gemeinsamen übergeordneten Thema oder Zweck dienen.

Im Grunde genommen sollte sich das von selbst verstehen, darüber hinaus ist in diesem Kontext aber auch noch ein anderer Aspekt zu berücksichtigen, der vielleicht nicht ganz so augenfällig ist: Klassen und Module, die in einer Komponente zusammengeführt werden, sollten auch *gemeinsam releast* werden können. Da sie dieselbe Versionsnummer haben, dasselbe Release Tracking durchlaufen und außerdem in derselben Release-Dokumentation geführt werden, macht das sowohl für die Autoren als auch für die User am meisten Sinn.

Zugegeben, als Empfehlungsgrundlage ist die Aussage, dass etwas »am meisten Sinn ergibt«, eher dürftig – das ist bestenfalls so, als würde man den Zeigefinger heben und versuchen autoritär zu klingen. Andererseits ist es schwierig, den »Kitt«, der die Klassen und Module in einer Komponente zusammenhält, präzise zu erklären. Doch so

dürftig diese Empfehlung auch sein mag, so bedeutsam ist dennoch das Prinzip selbst, weil Verstöße schnell zutage treten: Sie ergeben schlicht und ergreifend »keinen Sinn«. Wenn Sie gegen das REP verstoßen, werden Ihre User das zweifellos merken – und in der Folge nicht gerade begeistert von Ihren architektonischen Fähigkeiten sein.

Diese REP-Schwachstelle wird jedoch durch die Stärken der nachfolgend vorgestellten beiden Prinzipien mehr als nur wieder wettgemacht. Tatsächlich wird das REP sogar in hohem Maße durch das *Common-Closure-Prinzip* und das *Common-Reuse-Prinzip* definiert, wenn auch in eher widrigem Sinne.

13.2 CCP: Das Common-Closure-Prinzip

> *Fassen Sie Klassen, die aus denselben Gründen und zur selben Zeit modifiziert werden, in denselben Komponenten zusammen. Separieren Sie Klassen, die aus unterschiedlichen Gründen und zu unterschiedlichen Zeiten modifiziert werden, dagegen in verschiedene Komponenten.*

Im Wesentlichen handelt es sich hierbei um eine Variante des *Single-Responsibility-Prinzips* für die Komponentenebene: So, wie das SRP besagt, dass es nie mehr als einen Grund geben sollte, eine *Klasse* zu modifizieren, besagt das *Common-Closure-Prinzip*, dass es nie mehr als einen Grund geben sollte, eine *Komponente* zu modifizieren.

Für die meisten Anwendungen gilt, dass die Wartbarkeit einen höheren Stellenwert hat als die Wiederverwendbarkeit. Wenn der Code einer Anwendung geändert werden muss, dann sollten alle nötigen Anpassungen besser nur an einer Komponente vorgenommen werden müssen, statt an mehreren verteilten Komponenten.[1] Denn wenn sich die Modifikationen nur auf eine einzige Komponente beschränken, dann braucht auch nur diese eine geänderte Komponente erneut deployt zu werden. Andere Komponenten, die nicht von der modifizierten Komponente abhängig sind, müssten damit weder revalidiert noch erneut deployt werden.

Tatsächlich bezieht sich die im Namen des OCPs betonte *Closure* (zu Deutsch etwa »Geschlossenheit«) im wörtlichen Sinne auf die CCP-Adressen. Das OCP besagt, dass Klassen für Erweiterungen offen, aber zugleich auch Modifikationen gegenüber geschlossen sein sollten. Weil eine 100%ige Geschlossenheit jedoch nicht erreicht werden kann, muss sie strategischer Art sein. Und somit designen wir unsere Klassen in der Form, dass sie den geläufigsten Modifikationsarten, die wir erwarten oder bereits erlebt haben, gegenüber geschlossen sind.

Das CCP verstärkt diesen Grundsatz außerdem dahin gehend, dass diejenigen Klassen, die denselben Änderungsarten gegenüber geschlossen sind, in ein und derselben

Komponente zusammengefasst werden. Wenn sich dann die bisherigen Anforderungen ändern, stehen die Chancen, dass sich die betreffende Modifikation nur auf eine minimale Anzahl von Komponenten beschränkt, sehr gut.

13.2.1 Ähnlichkeiten mit dem SRP

Wie zuvor bereits erwähnt, ist das CCP das Äquivalent zum SRP auf Komponentenebene: Das SRP besagt, dass Methoden in verschiedene Klassen unterteilt werden, wenn sie sich aus unterschiedlichen Gründen ändern. Und das CCP schreibt vor, dass Klassen in verschiedene Komponenten unterteilt werden, wenn sie sich aus unterschiedlichen Gründen ändern. Beide Prinzipien lassen sich also wie folgt prägnant zusammenfassen:

> *Fassen Sie alle Elemente zusammen, die aus denselben Gründen und zur selben Zeit modifiziert werden. Separieren Sie dagegen alle Elemente, die aus unterschiedlichen Gründen und zu unterschiedlichen Zeiten modifiziert werden.*

13.3 CRP: Das Common-Reuse-Prinzip

> *Zwingen Sie die User einer Komponente nicht in eine Abhängigkeit von Elementen, die sie nicht benötigen.*

Das *Common-Reuse-Prinzip* ist ein weiterer Grundsatz, der als Entscheidungshilfe zur Bestimmung der Klassen und Module dient, die in einer Komponente zusammengeführt werden sollten. Es besagt, dass Klassen und Module, die im Allgemeinen gemeinsam wiederverwendet werden, in dieselbe Komponente gehören.

Klassen werden nur selten isoliert wiederverwendet. Üblicher ist es, dass wiederverwendbare Klassen mit anderen Klassen zusammenarbeiten, die Teil der wiederverwendbaren Abstraktion sind. Das CRP gibt vor, dass diese Klassen gemeinsam in dieselbe Komponente gehören. In solch einer Komponente würde man Klassen erwarten, die viele wechselseitige Abhängigkeiten aufweisen.

Ein einfaches Beispiel hierfür wäre eine Containerklasse mit den dazugehörigen Iteratoren. Diese Klassen werden gemeinsam wiederverwendet, weil sie eng miteinander verknüpft sind – und dementsprechend sollten sie sich auch in derselben Komponente befinden.

Darüber hinaus definiert das CRP aber noch mehr, als nur welche Klassen in derselben Komponente zusammengefasst werden sollten: Es beschreibt außerdem, welche Klassen *nicht* gemeinsam in einer Komponente enthalten sein sollten. Wenn eine

Komponente eine andere nutzt, entsteht eine Abhängigkeit zwischen diesen Komponenten. Möglicherweise greift die *nutzende* Komponente nur auf eine einzige Klasse innerhalb der *genutzten* Komponente zu – das ändert jedoch nichts an dem bestehenden Abhängigkeitsverhältnis an sich, denn: Die *nutzende* Komponente ist nach wie vor von der *genutzten* Komponente abhängig.

Aufgrund dieser Abhängigkeit ist es sehr wahrscheinlich, dass jede Änderung an der *genutzten* Komponente auch entsprechende Anpassungen an der *nutzenden* Komponente nach sich zieht. Und selbst wenn keine Modifikationen an der *nutzenden* Komponente vorgenommen werden müssen, muss diese trotzdem mit großer Wahrscheinlichkeit neu kompiliert, revalidiert und erneut deployt werden. Das gilt auch dann, wenn die *nutzende* Komponente überhaupt nicht von der Änderung an der *genutzten* Komponente betroffen ist.

Dementsprechend sollte bei einem Abhängigkeitsverhältnis zu einer Komponente stets sichergestellt sein, dass eine Abhängigkeit von jeder Klasse innerhalb dieser Komponente besteht. Oder anders ausgedrückt: Es muss sichergestellt werden, dass die Klassen, die in einer Komponente zusammengeführt werden, untrennbar sind – und somit ausgeschlossen ist, dass eine Abhängigkeit von manchen dieser Klassen besteht, von anderen aber nicht. Andernfalls würden mehr Komponenten als nötig ein erneutes Deployment durchlaufen und viel unnötiger Aufwand betrieben werden müssen.

Insofern gibt das CRP den Softwareentwicklern also eher vor, welche Klassen *nicht* zusammengeführt werden sollten, als welche Klassen *gemeinsam* zu gruppieren sind. Es besagt vielmehr, dass Klassen, zwischen denen keine enge wechselseitige Bindung besteht, *nicht* in dieselbe Komponente gehören.

13.3.1 Relation zum ISP

Das CRP ist die generische Version des ISP: Das ISP hält Softwareentwickler dazu an, keine Abhängigkeiten von Klassen zu erzeugen, in denen Methoden enthalten sind, die sie gar nicht nutzen. Und in ähnlicher Weise fordert das CRP dazu auf, keine Abhängigkeiten von Komponenten zu erzeugen, in denen Klassen enthalten sind, die gar nicht genutzt werden.

All diese Empfehlungen lassen sich auf einen einzigen prägnanten Nenner bringen:

> *Erzeugen Sie keine Abhängigkeiten von Elementen, die Sie nicht benötigen.*

13.4 Das Spannungsdiagramm für die

Komponentenkohäsion

Wie Ihnen sicher schon aufgefallen ist, tendieren die drei Kohäsionsprinzipien dazu, gegenläufig zu sein. Beim REP und dem CCP handelt es sich um *inkludierende* Prinzipien: Beide sind darauf ausgerichtet, den Umfang der Komponenten zu erweitern. Das CRP ist hingegen ein *exkludierendes* Prinzip, das die Reduzierung des Komponentenumfangs zum Ziel hat. Und für ebendiese Spannungssituation zwischen den drei Prinzipien müssen gute Softwarearchitekten eine Lösung finden.

Abbildung 13.1 zeigt ein Spannungsdiagramm,[2] das veranschaulicht, wie die drei Prinzipien der Kohäsion miteinander interagieren. Die Eckkreise dieses Diagramms beschreiben die *Kosten* für die Aufgabe des Prinzips auf der jeweils gegenüberliegenden Seite.

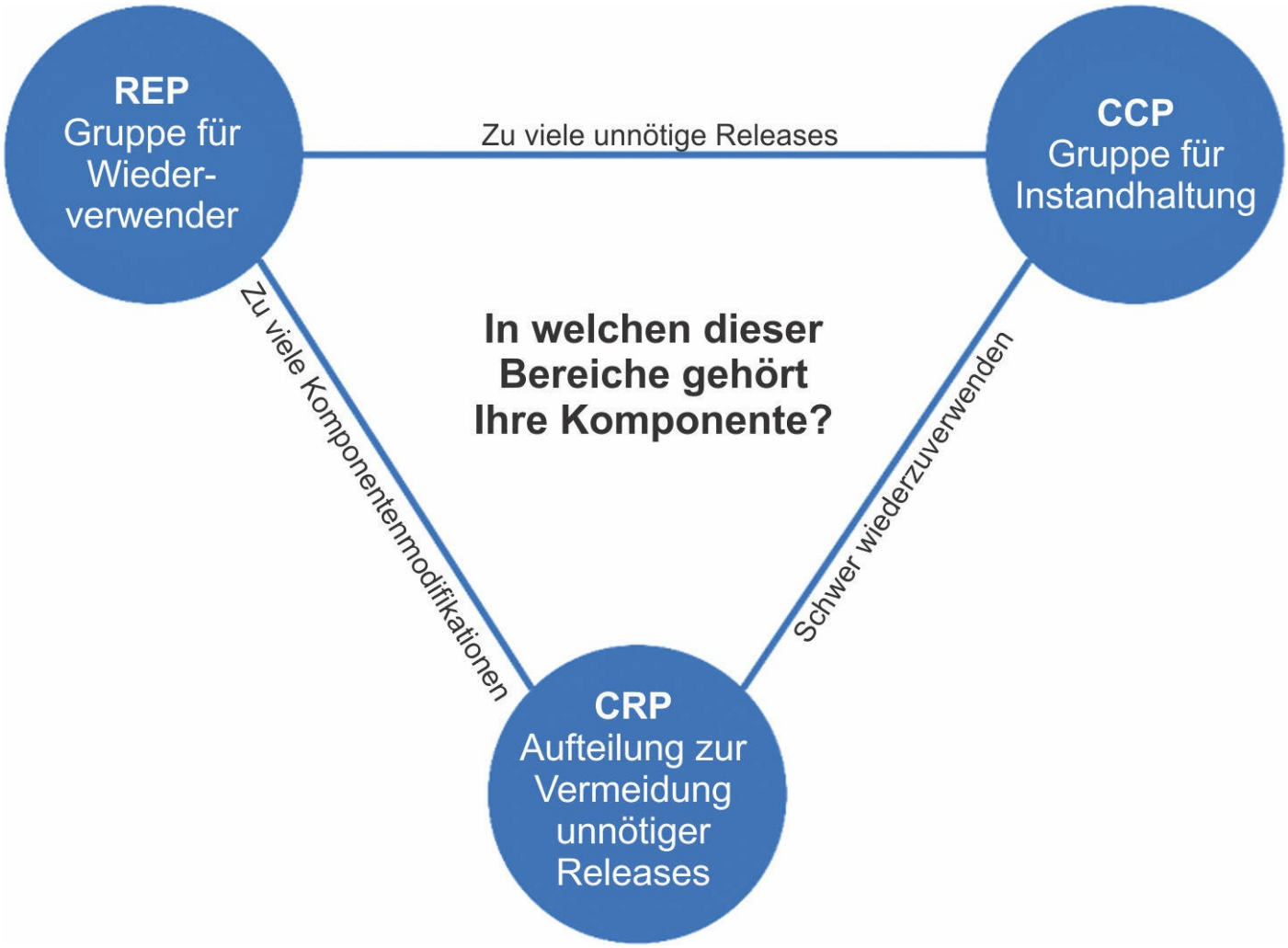


Abb. 13.1: Spannungsdiagramm der Kohäsionsprinzipien

Ein Softwarearchitekt, der sich nur auf das REP und das CRP konzentriert, wird feststellen, dass in der Folge zu viele Komponenten von simplen Modifikationen betroffen sind. Demgegenüber wird ein Softwarearchitekt, der sich zu sehr auf das CCP und das REP verlässt, zu viele unnötige Releases generieren.

Ein guter Softwarearchitekt findet eine Position innerhalb dieses Spannungsdreiecks, die den gegenwärtigen Belangen des Entwicklungsteams entgegenkommt, verliert darüber hinaus aber auch nicht aus dem Blick, dass sich diese Belange mit der Zeit ändern werden. Beispielsweise ist das CCP im Frühstadium eines Entwicklungsprojekts von deutlich größerer Bedeutung als das REP, weil hier statt der Wiederverwendbarkeit die Entwickelbarkeit im Vordergrund steht.

Grundsätzlich beginnen Projekte zunächst auf der rechten Seite des Dreiecks, wo lediglich die Wiederverwendbarkeit eingebüßt wird. Im weiteren Projektverlauf und mit dem Beginn neuer, davon abgeleiteter Projekte verlagert sich der Projektschwerpunkt dann auf die linke Seite. Soll heißen: Die Komponentenstruktur eines Projekts kann mit der Zeit sowie mit dem weiteren Voranschreiten variieren. Und das hat mehr mit der Art zu tun, wie das Projekt entwickelt und genutzt wird, als mit dem, was es eigentlich bezweckt.

13.5 Fazit

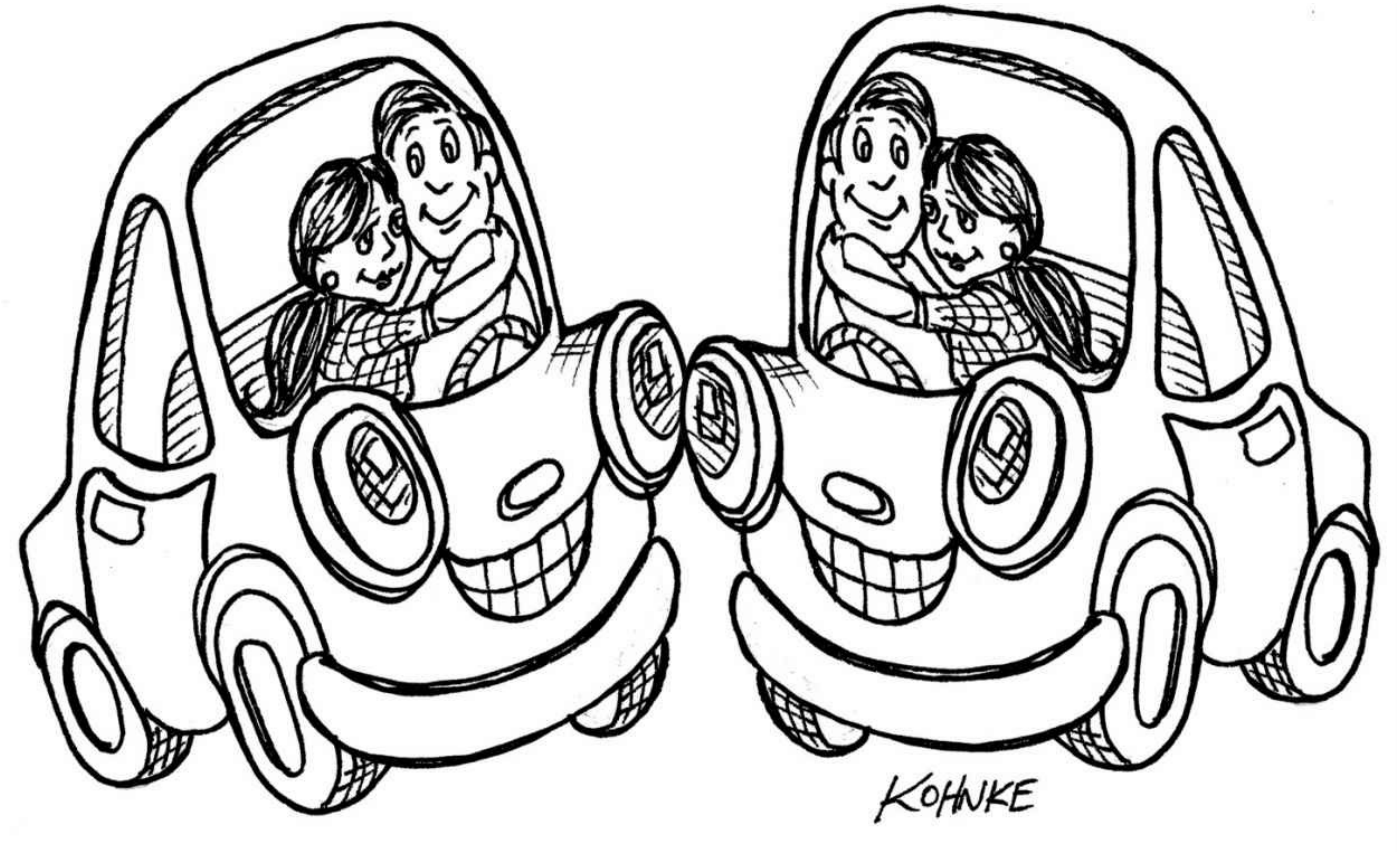
Unser Verständnis von der Kohäsion war in der Vergangenheit um einiges unvollständiger, als vom REP, CCP und CRP unterstellt. Damals galt die Kohäsion einfach als die Eigenschaft, die vorsah, dass ein Modul eine, und nur eine, Funktion ausführt. Die drei Prinzipien der Komponentenkohäsion beschreiben jedoch eine sehr viel komplexere Vielfalt des Konzepts der Kohäsion. Durch die Auswahl der Klassen, die in Komponenten zusammengefasst werden, müssen wir die gegensätzlichen Kräfte berücksichtigen, die mit der Wiederverwendbarkeit und der Entwickelbarkeit einhergehen – und das Ausbalancieren dieser Kräfte im Einklang mit den Anforderungen der Anwendung ist keine triviale Angelegenheit. Darüber hinaus weist diese Balance auch fast immer eine gewisse Dynamik auf, soll heißen: Eine Klassenaufteilung, die heute noch angebracht ist, mag schon nächstes Jahr nicht mehr geeignet sein. Und in der Folge wird die Komponentenkomposition aller Wahrscheinlichkeit nach ins Wanken geraten und sich mit der Zeit weiterentwickeln, während sich der Fokus des Projekts von der Entwickelbarkeit hin zur Wiederverwendbarkeit verlagert.

---

[1] Siehe den Abschnitt »Das Kätzchen-Problem« in Kapitel 27 »Services – große und kleine«.

[2] Ich bedanke mich bei Tim Ottinger für diese Idee.

Kapitel 14
Komponentenkopplung



Die drei im Folgenden vorgestellten Prinzipien betreffen die Beziehungen zwischen den einzelnen Komponenten. Auch hier kommt es wieder zu einem Spannungsverhältnis zwischen der Entwickelbarkeit und dem logischen Design. Die Kräfte, die auf die Architektur einer Komponentenstruktur wirken, sind in diesem Kontext sowohl technischer und strategischer als auch flüchtiger Natur.

14.1 ADP: Das Acyclic-Dependencies-Prinzip

Lassen Sie im Schema der Komponentenabhängigkeiten keine Zyklen zu.

Haben Sie vielleicht auch schon mal einen ganzen Tag damit zugebracht, ein paar Dinge zum Funktionieren zu bringen, nur um dann, nachdem Sie zufrieden nach Hause gegangen waren, am nächsten Morgen feststellen zu müssen, dass nichts davon mehr läuft? Was ist da passiert? Nun, höchstwahrscheinlich war jemand anders noch länger im Büro als Sie und hat irgendetwas verändert, von dem Ihre Arbeit abhängig war! Ich

bezeichne das als das *Morning-After-Syndrom* (zu Deutsch etwa das »Der-Morgen-danach-Syndrom«).

Dieses Syndrom tritt in Entwicklungsumgebungen zutage, in denen mehrere Softwareentwickler dieselben Quelldateien bearbeiten. Bei relativ kleinen Projekten mit nur wenigen Programmierern ist das noch kein allzu großes Problem, aber je umfangreicher das Projekt und damit auch das zugehörige Entwicklungsteam wird, desto albtraumhafter kann »der Morgen danach« ausarten. So sind manche Teams oftmals wochenlang nicht in der Lage, eine stabile Version ihres Projekts auf die Beine zu stellen, weil jeder ununterbrochen damit beschäftigt ist, seinen Teil des Codes an die zuletzt von jemand anderem vorgenommenen Änderungen anzupassen.

Im Laufe der letzten paar Jahrzehnte haben sich zwei Lösungen für diese Problematik etabliert, die aus der Telekommunikationsindustrie übernommen wurden. Bei der ersten handelt es sich um den sogenannten »wöchentlichen Build« und bei der zweiten um das *Acyclic-Dependencies-Prinzip* (zu Deutsch »Prinzip der azyklischen Abhängigkeiten«).

14.1.1 Der wöchentliche Build

Der wöchentliche Build war ursprünglich ein für mittelgroße Projekte gängiges Verfahren, das folgendermaßen funktioniert:

An den ersten vier Wochentagen ignorieren sich die Programmierer gegenseitig. Jeder arbeitet ausschließlich an seinen eigenen, »privaten« Codekopien und kümmert sich zunächst einmal nicht darum, seine individuellen Fortschritte auf einer kollektiven Basis einzubringen. Die Integration sämtlicher vorgenommenen Änderungen und Anpassungen findet dann immer jeweils erst am Freitag der Woche statt – dem Tag, an dem der Systembuild erstellt wird.

Diese Herangehensweise hat den wunderbaren Vorteil, dass die Softwareentwickler die Möglichkeit haben, an vier von fünf Wochentagen quasi in einer isolierten Welt zu agieren. Der Nachteil ist aber natürlich der massive Integrationsaufwand, den es am Freitag einer jeden Woche zu bewältigen gilt.

Je weiter das Projekt allerdings voranschreitet, desto weniger praktikabel wird es, die komplette Integration des Projekts an besagtem Freitag abzuschließen – stattdessen steigt die Integrationslast immer weiter an, bis sie irgendwann auch den Samstag vereinnahmt. In aller Regel reichen dann schon wenige solcher arbeitsreichen Wochenendtage aus, um die Softwareentwickler davon zu überzeugen, dass die Integration am besten doch bereits donnerstags beginnen sollte. Und so wird der Integrationsstart ganz allmählich in die Mitte der Woche vorverlegt.

Je mehr der Arbeitszyklus des Entwicklungsmodus gegenüber der Integration sinkt,

umso mehr sinkt auch die Effizienz des Teams. Schließlich wird diese Situation irgendwann so frustrierend, dass die Softwareentwickler oder Projektmanager beschließen, zu einem zweiwöchentlichen Build-Rhythmus zu wechseln. Das geht dann auch eine Weile gut, der Zeitaufwand für die Integration steigt aber trotz allem weiterhin unaufhaltsam an.

Letzten Endes führt dieses Szenario somit unweigerlich in eine Krise: Um die Effizienz aufrechtzuerhalten, muss der Zeitplan für den Build ständig verlängert werden – diese Verschiebungen erhöhen andererseits jedoch auch die Risiken für das Projekt. Außerdem sind sowohl die Integration als auch das Testen zunehmend schwieriger zu bewerkstelligen und das Team büßt den Vorteil eines zügigen Feedbacks ein.

14.1.2 Abhängigkeitszyklen abschaffen

Die Lösung für dieses Problem ist die Unterteilung der Entwicklungsumgebung in releasefähige Komponenten. Diese Komponenten werden zu Arbeitseinheiten, die der Verantwortung eines einzelnen Softwareentwicklers oder auch eines Entwicklungsteams unterstellt werden können. Sobald die Softwareentwickler eine Komponente zum Funktionieren gebracht haben, geben sie sie für die Nutzung durch andere Programmierer frei. Sie weisen ihr eine Versionsnummer zu und verschieben sie in ein Verzeichnis, das für den Zugriff durch die anderen Softwareentwickler freigegeben ist. Und anschließend fahren sie damit fort, ihre Komponente wieder in ihrer eigenen, privaten Umgebung weiter zu modifizieren – während alle anderen die freigegebene Version benutzen.

Sobald eine neue Version einer Komponente zur Verfügung gestellt wird, können die anderen Teams selbst entscheiden, ob sie diese sofort übernehmen wollen. Ist das nicht der Fall, dann arbeiten sie einfach mit der alten Version weiter. Und wenn sie dann zu einem späteren Zeitpunkt dazu bereit sind, nutzen sie die neue Version.

Auf diese Weise unterliegt kein Team der »Willkür« der anderen Programmierer: Modifikationen, die an einer Komponente vorgenommen werden, müssen sich nicht zwangsläufig unmittelbar auf die Arbeit der anderen Teams auswirken. Vielmehr kann jedes Team für sich selbst entscheiden, wann es die eigenen Komponenten an die übrigen neuen Komponentenversionen adaptiert. Darüber hinaus erfolgt auch die Integration in kleineren Inkrementabschnitten. Es gibt also keinen bestimmten Zeitpunkt, an dem sämtliche Softwareentwickler zusammenkommen und alles, woran sie gerade arbeiten, integrieren müssen.

Hierbei handelt es sich um einen sehr einfachen und rationalen Prozess, der weit verbreitet Anwendung findet. Voraussetzung dafür, dass er reibungslos ablaufen kann, ist allerdings ein entsprechender Aufbau der Abhängigkeitsstruktur aller

Komponenten: *Es dürfen keine Zyklen auftreten* – andernfalls lässt sich das »Morning-After-Syndrom« nicht vermeiden.

Betrachten Sie dazu einmal das Komponentendiagramm in Abbildung 14.1. Es zeigt eine recht typische Struktur der in einer Anwendung verwendeten Komponenten. Die Funktion dieser Anwendung ist für den Zweck dieses Beispiels nicht von Bedeutung, hier geht es nur um die Abhängigkeitsstruktur der Komponenten. Wie Sie sehen, handelt es sich um einen *gerichteten Graphen*: Die Komponenten sind *Knoten*, und die Abhängigkeitsbeziehungen sind *gerichtete Kanten*.

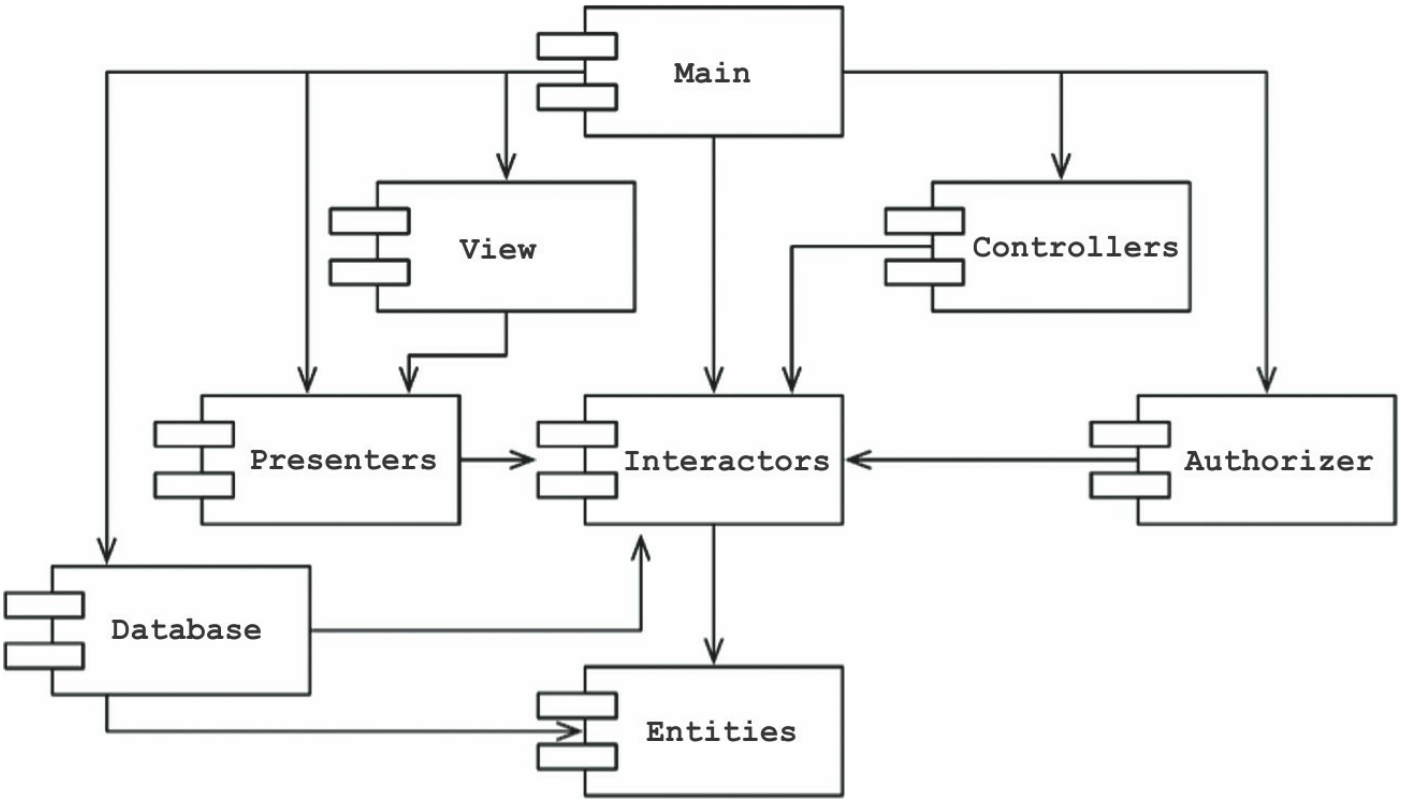


Abb. 14.1: Typisches Komponentendiagramm

Beachten Sie außerdem Folgendes: Egal, von welcher Komponente Sie ausgehen, es ist nicht möglich, den Abhängigkeitsbeziehungen zu folgen und am Ende wieder bei derselben Komponente anzukommen. Die hier gezeigte Struktur enthält keine Zyklen. Sie ist ein *gerichteter azyklischer Graph* (*DAG*, **D**irected **A**cyclic **G**raph).

Überlegen Sie nun einmal, was passiert, wenn das für `Presenters` zuständige Team eine neue Version seiner Komponente erstellt. Sie können problemlos nachvollziehen, welche anderen Komponenten von dieser neuen Version beeinflusst werden, indem Sie den Abhängigkeitspfeilen in rückwärtiger Richtung folgen. In diesem Beispiel sind sowohl `View` als auch `Main` betroffen. Die Softwareentwickler, die gegenwärtig an diesen Komponenten arbeiten, werden also irgendwann entscheiden müssen, wann sie ihre Arbeit an die neue Version von `Presenters` anpassen.

Ein weiterer wichtiger Punkt in diesem Schema: Das Release von `Main` hat faktisch

keinerlei Auswirkung auf eine der anderen Systemkomponenten. Letztere haben überhaupt keine Kenntnis von `Main` und werden insofern auch nicht davon beeinflusst, ob sie sich ändert oder nicht. Und das ist insofern günstig, als es bedeutet, dass die Folgen der Freigabe von `Main` relativ geringfügig sind.

Wenn die Entwickler der Komponente `Presenters` diese testen wollen, brauchen sie für ihren Versionsbuild lediglich die Kompatibilität mit den zu diesem Zeitpunkt gültigen Versionen der `Interactors`- und `Entities`-Komponenten zu gewährleisten. Ansonsten müssen keine anderen Systemkomponenten berücksichtigt werden. Und dies ist wiederum von Vorteil, weil es bedeutet, dass die Entwickler der Komponente `Presenters` nur relativ wenig Aufwand für das Testen betreiben und auch nur verhältnismäßig wenig andere Variablen mit einbeziehen müssen.

Sobald es dann Zeit für das Release des gesamten Systems ist, wird der Prozess von unten nach oben fortgesetzt: Als Erstes wird die Komponente `Entities` kompiliert, getestet und freigegeben. Das Gleiche passiert im Anschluss mit `Database` und `Interactors`. Daraufhin folgen die Komponenten `Presenters`, `View`, `Controller` und `Authorizer`. Und zuletzt ist `Main` an der Reihe. Diese Vorgehensweise ist sehr transparent und einfach zu handhaben, weil jeder Beteiligte weiß, wie das System aufgebaut werden muss und die Abhängigkeiten zwischen den einzelnen Bestandteilen versteht.

14.1.3 Auswirkung eines Zyklus in einem Komponentenabhängigkeitsgraphen

Angenommen, eine neue Anforderung macht die Modifikation einer der in `Entities` enthaltenen Klassen in der Form erforderlich, dass sie auf eine Klasse in `Authorizer` zugreifen muss. Sagen wir zum Beispiel, dass die Klasse `User` in der Komponente `Entities` die Klasse `Permissions` der Komponente `Authorizers` verwenden soll. Dadurch wird ein wie in Abbildung 14.2 dargestellter Abhängigkeitszyklus erzeugt.

Dieser Zyklus verursacht einige unmittelbare Probleme. Zum Beispiel sind sich die Entwickler der `Database`-Komponente darüber im Klaren, dass Letztere zwecks Release mit der Komponente `Entities` kompatibel sein muss. Allerdings muss die Komponente nun angesichts dieses Zyklus *auch* mit `Authorizer` kompatibel sein – wobei `Authorizer` von `Interactors` abhängig ist. Und das macht die Freigabe von `Database` deutlich schwieriger. `Entities`, `Authorizer` und `Interactors` sind im Endeffekt zu einer großen Komponente verschmolzen, was wiederum bedeutet, dass alle Softwareentwickler, die an diesen Bestandteilen arbeiten, das gefürchtete »Morning-After-Syndrom« erleben werden. Sie werden regelrecht übereinander stolpern, weil sie alle exakt dieselbe Releaseversion ihrer jeweiligen Komponenten verwenden müssen.

Doch das ist nur ein Teil des Problems. Stellen Sie sich beispielsweise einmal vor, was passiert, wenn die `Entities`-Komponente getestet werden muss: Wie sich zeigt, muss der Build nun ärgerlicherweise mit `Authorizer` und `Interactors` kompatibel sein. Tatsächlich ist das Kopplungsniveau zwischen diesen Komponenten beunruhigend und störend, wenn nicht gar unakzeptabel.

Vielleicht fragen Sie sich, warum so viele verschiedene Libraries und die Arbeit der anderen Softwareentwickler mit berücksichtigt werden müssen, nur um einen einfachen Unit-Test für eine Ihrer Klassen durchzuführen. Bei genauer Betrachtung werden Sie allerdings feststellen, dass der Abhängigkeitsgraph Zyklen ausweist – und solche Zyklen machen die Isolierung der Komponenten sehr schwierig: Unit-Tests und Release werden dadurch sehr kompliziert und fehleranfällig. Außerdem steigt die Zahl der Build-Probleme in geometrischer Progression zu der Anzahl der Module.

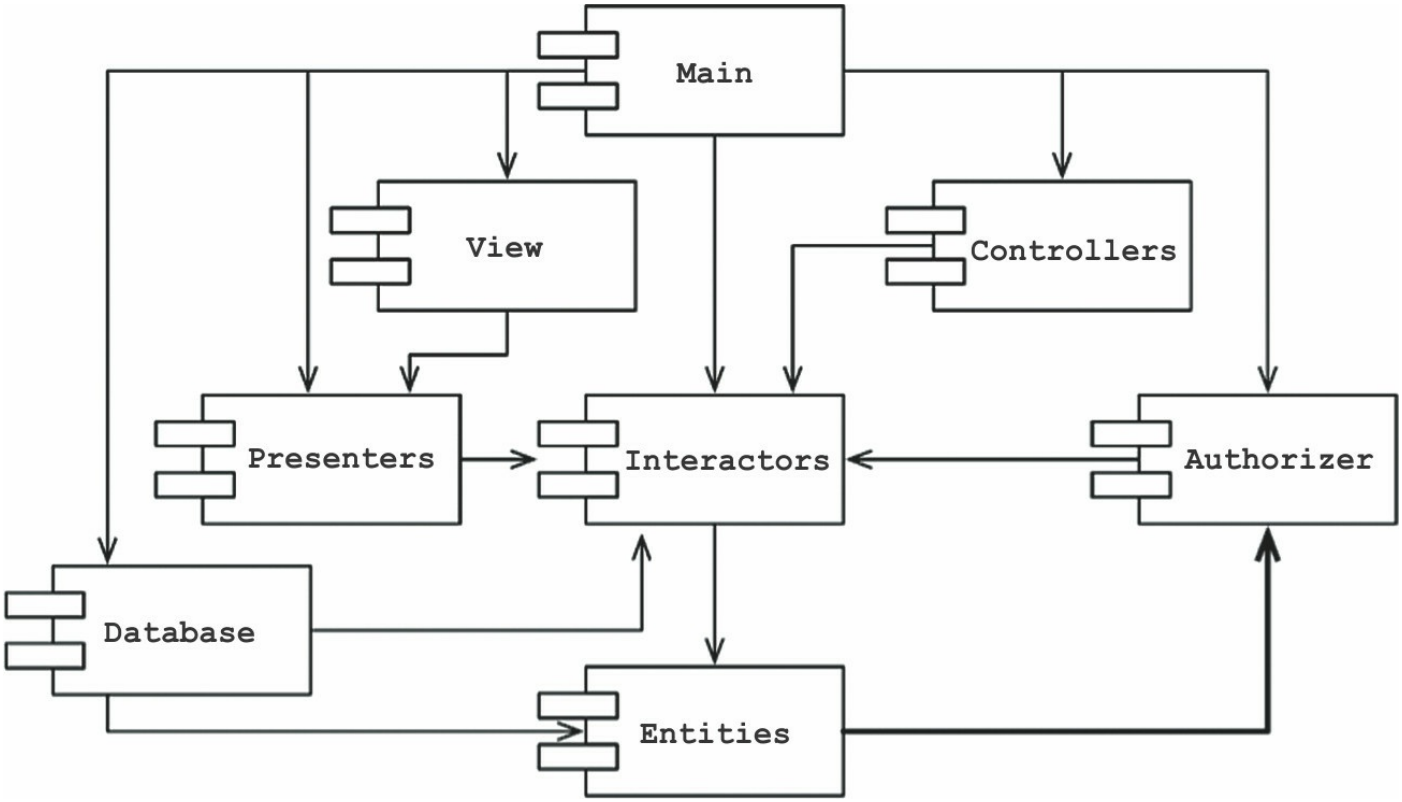


Abb. 14.2: Ein Abhängigkeitszyklus

Sobald sich Zyklen im Abhängigkeitsgraphen finden, kann es darüber hinaus auch sehr schwierig sein, die Reihenfolge festzulegen, in der die Komponenten erstellt werden müssen. Tatsächlich gibt es aber auch gar keine richtige Reihenfolge, doch das kann dann in Sprachen wie Java, die ihre Deklarationen aus kompilierten Binärdateien auslesen, einige sehr knifflige Probleme nach sich ziehen.

14.1.4 Den Zyklus durchbrechen

Es ist immer möglich, einen Komponentenzyklus aufzubrechen und den Abhängigkeitsgraphen wieder als DAG einzusetzen. Zu diesem Zweck stehen vor allem zwei Hauptmechanismen zur Verfügung:

1. Anwendung des *Dependency-Inversion-Prinzips (DIP)*. In dem in Abbildung 14.3 dargestellten Fall könnten die Softwareentwickler eine Schnittstelle mit den von `User` benötigten Methoden erzeugen, die sich anschließend in `Entities` einbinden und in `Authorizer` vererben lässt. Dadurch wird die Abhängigkeit zwischen `Entities` und `Authorizer` invertiert und der Zyklus damit durchbrochen.

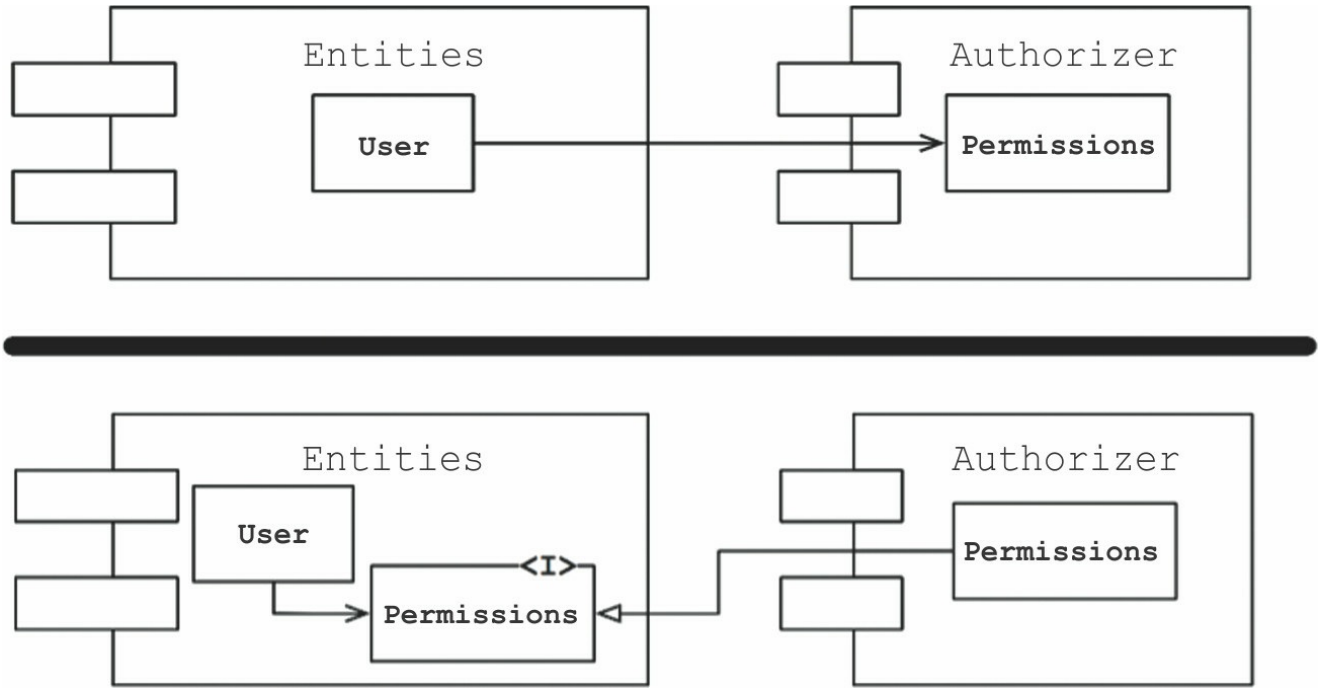


Abb. 14.3: Invertierung der Abhängigkeiten zwischen `Entities` und `Authorizer`

2. Erstellung einer neuen Komponente, von der sowohl `Entities` als auch `Authorizer` abhängig sind. Verschieben Sie die Klasse(n), von denen beide gemeinsam abhängig sind, in die neue Komponente (Abbildung 14.4).

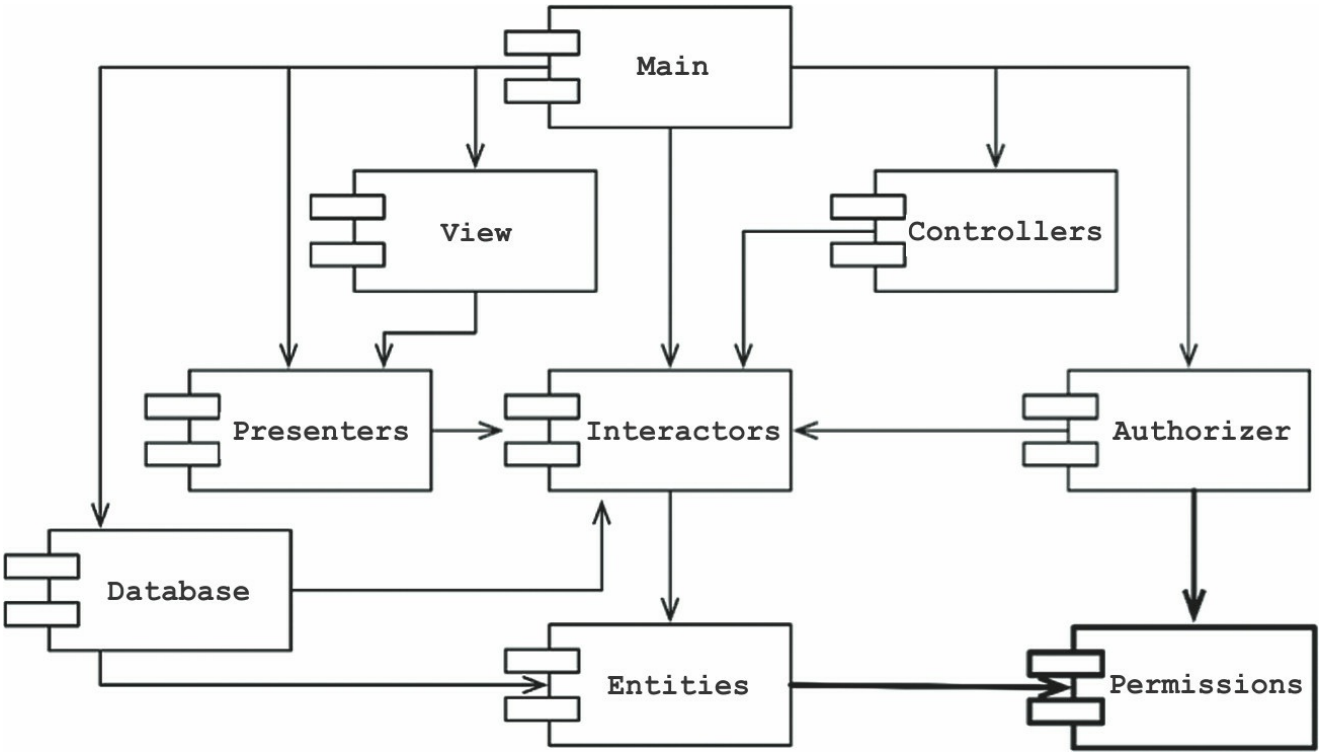


Abb. 14.4: Die neue Komponente, von der sowohl `Entities` als auch `Authorizer` abhängig sind.

14.1.5 Jitters (Fluktuationen)

Bei der zweiten Lösung wird unterstellt, dass die Komponentenstruktur angesichts der Modifikationsanforderungen flüchtig ist. Tatsächlich fluktuiert und wächst die Struktur der Komponentenabhängigkeiten mit steigendem Umfang der Anwendung. Dementsprechend muss die Abhängigkeitsstruktur kontinuierlich auf Zyklen überwacht werden. Und sollten Zyklen auftreten, müssen sie irgendwie durchbrochen werden. Mitunter bedingt dies auch die Erzeugung neuer Komponenten, was dann in der Konsequenz wiederum zu einer Ausweitung der Abhängigkeitsstruktur führt.

14.2 Top-down-Design

Die bisher in diesem Kapitel ausgeführten Probleme lassen letztendlich nur einen Schluss zu: Die Komponentenstruktur kann nicht von oben nach unten designt werden. Sie gehört zwar zu den ersten Systembestandteilen, die entworfen werden, aber sie entwickelt sich parallel zum Wachstum des Systems und den daran vorgenommenen Modifikationen.

Dieser Punkt mag dem ein oder anderen Leser kontraintuitiv erscheinen: Sie müssen davon ausgehen, dass grobgranulare Dekompositionen wie Komponenten gleichzeitig auch übergeordnete *funktionale* Dekompositionen sein werden.

Die schematische Betrachtung einer grobgranularen Gruppierung, wie etwa einer Komponentenabhängigkeitsstruktur, suggeriert, dass die Komponenten in irgendeiner Weise die Funktionen des Systems repräsentieren. Tatsächlich scheint das jedoch keine Eigenschaft der Komponentendiagramme zu sein.

Im Wesentlichen haben sie sogar nur sehr wenig mit der Beschreibung der Funktion einer Anwendung zu tun, sondern stellen eine Art Karte in Bezug auf die *digitale Baubarkeit* und *Wartbarkeit* der Anwendung dar. Aus diesem Grund werden solche Diagramme auch nicht gleich zu Beginn des Projekts entworfen: Solange es noch keine Software zu bauen oder instand zu halten gibt, besteht auch kein Bedarf für eine Bau- und Wartbarkeitskarte. Je mehr Module dann jedoch in den frühen Phasen der Implementierung und des Softwaredesigns zusammengeführt werden, desto mehr müssen auch die dadurch entstehenden Abhängigkeiten verwaltet werden, damit das Projekt ohne das lästige »Morning-After-Syndrom« durchgeführt werden kann. Darüber hinaus ist es ohnehin sinnvoll, die Modifikationen so lokal wie möglich zu halten, damit sich die Aufmerksamkeit der Softwareentwickler vermehrt auf das SRP und das CCP sowie die Zusammenlegung der Klassen richten kann, die aller Voraussicht nach gemeinsam modifiziert werden müssen.

Eine der vordringlichsten Überlegungen zur Abhängigkeitsstruktur gilt dem Isolieren der Flüchtigkeit: Komponenten, die häufig und aus unvorhersehbaren Gründen modifiziert werden, dürfen sich nicht auf andere Komponenten auswirken, die ansonsten stabil bleiben sollten. Beispielsweise sollen kosmetische Änderungen an der grafischen Benutzerschnittstelle (*GUI*, **G**raphical **U**ser **I**nterface) keine Auswirkungen auf die Geschäftsregeln haben. Die Ergänzung oder Modifikation von Berichten soll die übergeordneten Richtlinien in keiner Weise beeinflussen. Also müssen die Softwarearchitekten den Komponentenabhängigkeitsgraphen entsprechend aufbauen und formen, um die stabilen übergeordneten Komponenten gegen Änderungen aufseiten der flüchtigen Komponenten zu schützen.

Je umfangreicher die Anwendung wird, desto mehr gilt es auch, sich mit der Erzeugung wiederverwendbarer Elemente zu befassen. An diesem Punkt beginnt der Einfluss des CRPs auf die Komponentenzusammenstellung. Und wenn dann Zyklen auftreten, wird das ADP angewendet – und die Fluktuationen und der Umfang des Komponentenabhängigkeitsgraphen steigen an.

Der Versuch, die Komponentenabhängigkeitsstruktur vor dem Entwurf irgendwelcher Klassen zu planen, wäre definitiv zum Scheitern verurteilt, denn: Wir wüssten kaum etwas über die Common-Closure-Verhältnisse, wir könnten noch keine wiederverwendbaren Elemente vorhersehen, und wir würden mit großer Sicherheit Komponenten erstellen, die Abhängigkeitszyklen erzeugen. Deshalb wächst und entwickelt sich die Komponentenabhängigkeitsstruktur parallel zum logischen Design des Systems.

14.3 SDP: Das Stable-Dependencies-Prinzip

Abhängigkeiten sollten in dieselbe Richtung verlaufen wie die Stabilität.

Softwaredesigns können nicht vollständig statisch sein. Vielmehr ist ein gewisses Maß an Flüchtigkeit notwendig, wenn das Design wartbar sein soll. Durch die Einhaltung des *Common-Closure-Prinzips* erzeugen wir Komponenten, die auf bestimmte Arten von Modifikationen empfindlich reagieren, während sie anderen Anpassungen gegenüber immun sind. Einige dieser Komponenten sind bewusst flüchtig designt – in diesen Fällen wird *erwartet*, dass sie sich ändern werden.

Keine Komponente, die potenziell als flüchtig erachtet werden, sollte eingehende Abhängigkeiten von einer Komponente aufweisen, die sich nur schwer modifizieren lässt – andernfalls wird auch die Anpassung der flüchtigen Komponente schwierig sein.

Es ist die Perversion der Software, dass ein Modul, das vom Design her einfach zu modifizieren sein sollte, durch die Ergänzung einer Abhängigkeit vonseiten eines anderen Entwicklers oder Teams schwer anzupassen ist. Dazu muss sich nicht einmal eine einzige Zeile im Quellcode Ihres Moduls ändern – dennoch erweist sich die Modifizierung des Moduls plötzlich als größere Herausforderung als erwartet. Durch die Einhaltung des *Stable-Dependencies-Prinzips* wird sichergestellt, dass Module, die leicht anzupassen sein sollen, nicht von Modulen abhängig sind, die sich nur mit größerem Aufwand ändern lassen.

14.3.1 Stabilität

Was bedeutet »Stabilität«? Stellen Sie mal eine 1-Euro-Münze hochkant auf eine Tischplatte. Ist sie in dieser Position stabil? Die Antwort auf diese Frage würde wohl eher »Nein« lauten. Solange jedoch nicht an der Tischplatte gewackelt wird, kann das Geldstück für sehr lange Zeit in dieser Position verbleiben. Stabilität hat also nicht unmittelbar mit der Häufigkeit einer Änderung zu tun: Die Position der Euro-Münze wird sich nicht verändern, trotzdem fällt es schwer, sie als stabil zu betrachten.

Die Bedeutung des Wortes »stabil« wird im Duden als »in sich konstant bleibend, gleichbleibend, relativ unveränderlich« beschrieben. Stabilität steht im Verhältnis zu dem Arbeitsaufwand, der erforderlich ist, um eine Veränderung dieses Zustands herbeizuführen. Auf der einen Seite ist die hochkant stehende 1-Euro-Münze nicht stabil, weil es nur wenig Aufwands bedarf (z.B. ein Stoß gegen die Tischplatte), um sie ins Wanken und zum Kippen zu bringen. Der Tisch jedoch ist auf der anderen Seite sehr stabil, weil es eines deutlich größeren Aufwands bedarf, ihn umzustoßen.

Doch wie hängt das nun mit Software zusammen? Es gibt viele Faktoren, die die

Modifikation einer Softwarekomponente erschweren – hier spielen unter anderem beispielsweise ihre Größe, Komplexität und Übersichtlichkeit eine Rolle. An dieser Stelle werden wir all diese Faktoren jedoch außer Acht lassen und uns stattdessen auf etwas anderes konzentrieren. Eine sichere Methode, um Änderungen an einer Softwarekomponente schwierig zu gestalten, besteht darin, viele andere Softwarekomponenten in Abhängigkeit zu ihr zu setzen. Eine Komponente mit vielen eingehenden Abhängigkeiten ist sehr stabil, weil es eines hohen Aufwands bedarf, jegliche Modifikationen daran mit allen von ihr abhängigen Komponenten in Einklang zu bringen.

Die in Abbildung 14.5 gezeigte schematische Darstellung weist X als stabile Komponente aus. Es gibt drei Komponenten, die von X abhängig sind – und damit drei gute Gründe, keine Änderungen an X vorzunehmen. Man spricht in einer solchen Konstellation davon, dass X für diese drei Komponenten *verantwortlich* ist. Umgekehrt ist X aber nicht von einem anderen Element abhängig und unterliegt damit auch keinem externen Einfluss, der eine Modifikation an dieser Komponente erforderlich machen würde. Damit ist sie zugleich *unabhängig*.

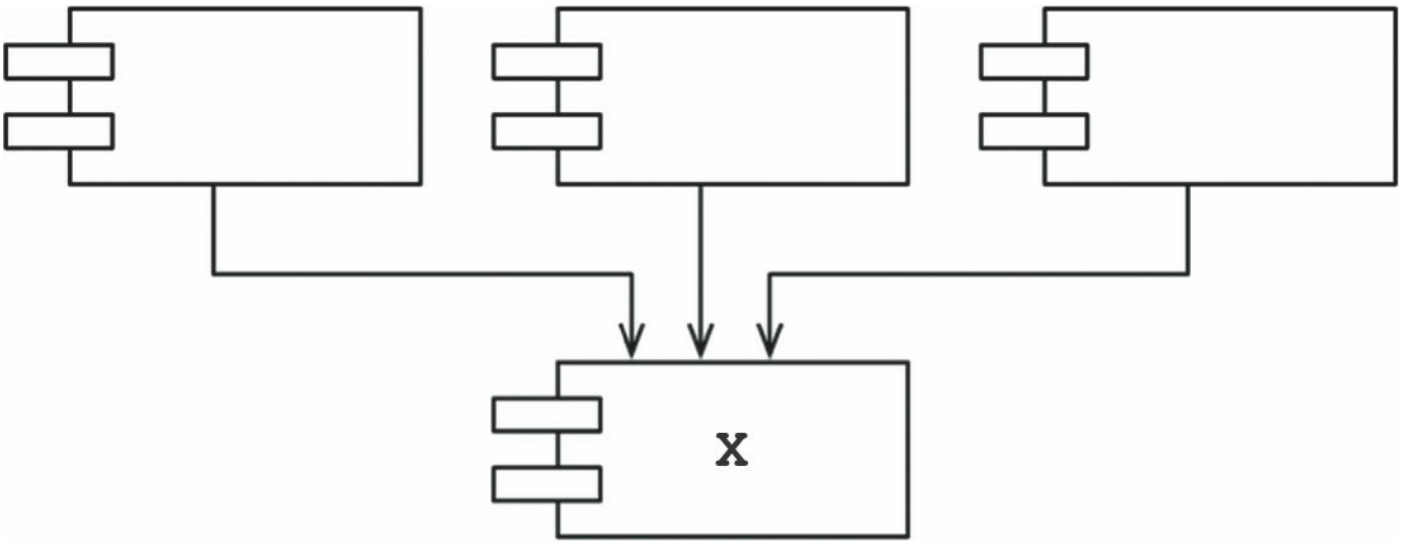


Abb. 14.5: x: Eine stabile Komponente

In Abbildung 14.6 ist dagegen die sehr instabile Komponente Y zu sehen. Es sind keine anderen Komponenten von ihr abhängig, das heißt, sie hat *keine Verantwortlichkeit*. Allerdings ist Y selbst von drei anderen Komponenten abhängig, sodass sie möglichen Modifikationen vonseiten drei externer Quellen ausgesetzt ist. Damit ist Y *abhängig*.

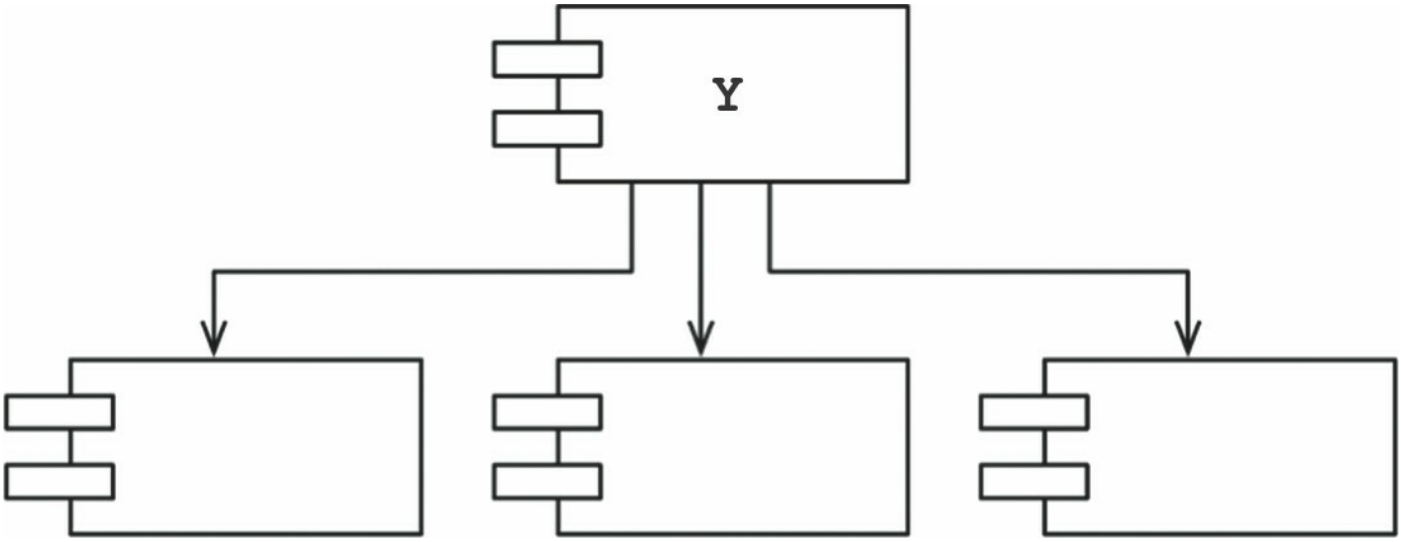


Abb. 14.6: Y: Eine sehr instabile Komponente

14.3.2 Stabilitätsmetriken

Und wie lässt sich die Stabilität einer Komponente bemessen? Eine Möglichkeit ist, ihre eingehenden und ausgehenden Abhängigkeiten zu zählen. Eine solche Zählung ermöglicht die Berechnung der *positionalen* Stabilität der Komponente.

- **Fan-in:** Eingehende Abhängigkeiten. Diese Messgröße gibt die Anzahl der Klassen außerhalb der betreffenden Komponente an, die von Klassen innerhalb der Komponente abhängig sind.
- **Fan-out:** Ausgehende Abhängigkeiten. Diese Messgröße beschreibt die Anzahl der Klassen innerhalb der betreffenden Komponente, die von Klassen außerhalb der Komponente abhängig sind.
- **I:** Instabilität. *I=Fan-out/(Fan-in+Fan-out)*. Der Wertebereich dieser Messgröße beträgt `[0,1]`. Dabei entspricht `I=0` einer maximal stabilen Komponente und `I=1` einer maximal instabilen Komponente.

Die Berechnung der *Fan-in*- und *Fan-out*-Metriken[1] erfolgt durch die Zählung der Klassen außerhalb der fraglichen Komponente, die Abhängigkeiten zu den Klassen innerhalb dieser Komponente aufweisen. Betrachten Sie dazu das Beispiel in Abbildung 14.7.

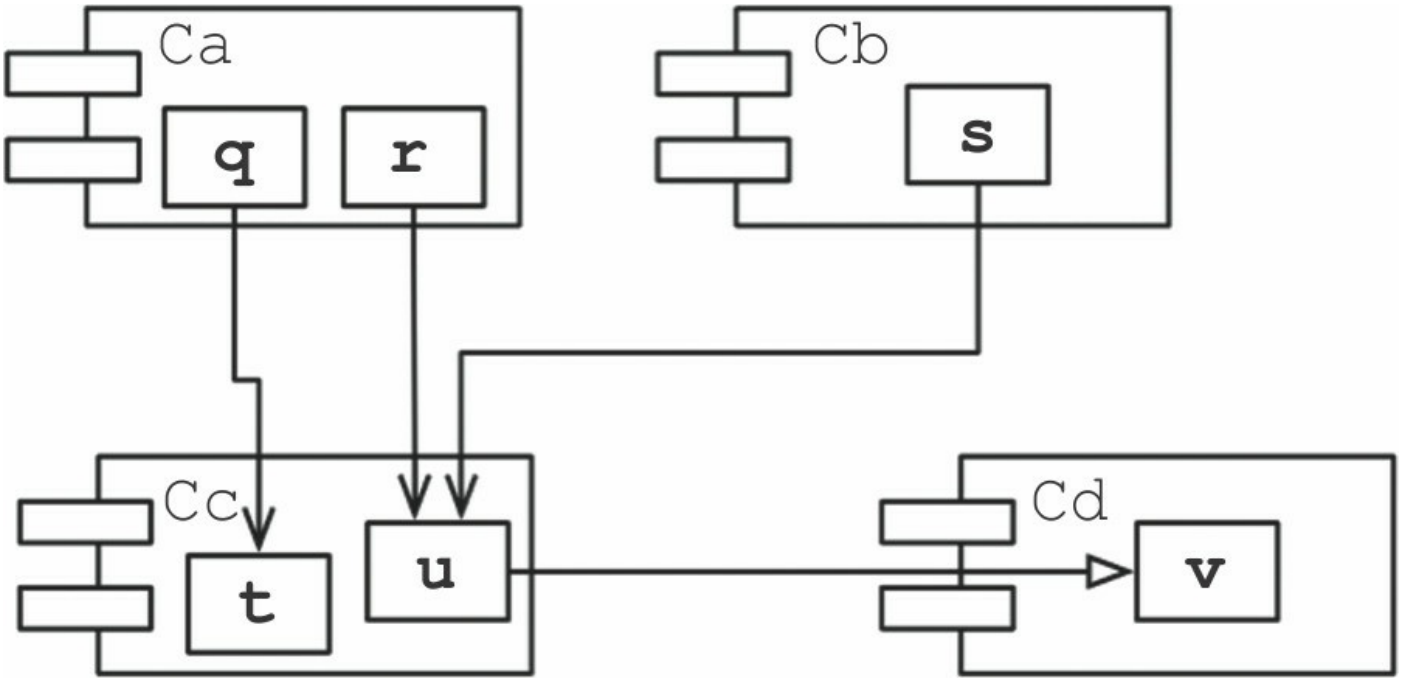


Abb. 14.7: Beispiel für die Berechnung der Komponentenstabilität

Angenommen, Sie wollten die Stabilität der Komponente `Cc` berechnen und stellen fest, dass es drei Klassen außerhalb dieser Komponente gibt, die von Klassen innerhalb von `Cc` abhängig sind. Dementsprechend ist *Fan-in=3*. Darüber hinaus gibt es eine Klasse außerhalb von `Cc`, von denen der Komponente innewohnende Klassen abhängig sind. Somit ergibt sich hier *Fan-out=1* und *I=1/4*.

In C++ werden diese Abhängigkeiten typischerweise durch `#include`-Anweisungen repräsentiert. Tatsächlich ist die Messgröße *I* am einfachsten zu berechnen, wenn Sie Ihren Quellcode so strukturiert haben, dass es immer nur eine Klasse in jeder Quelldatei gibt. In Java lässt sich die Messgröße *I* durch die Zählung der `import`-Anweisungen und der gültigen Bezeichner ermitteln.

Wenn *I* gleich 1 ist, bedeutet dies, dass keine andere Komponente von der aktuellen Komponente abhängig ist (*Fan-in=0*) und sie selbst wiederum von anderen Komponenten abhängig ist (*Fan-out>0*). Diese Situation stellt sich so instabil dar, wie eine Komponente nur sein kann – sie ist nicht *verantwortlich* und *abhängig* zugleich. Der Mangel an von ihr abhängigen Komponenten liefert keinen Grund, warum sie nicht selbst verändert werden könnte, während die Komponenten, von denen sie abhängig ist, jede Menge Gründe für Modifikationen darstellen könnten.

Wenn die Messgröße *I* im Gegensatz dazu gleich 0 ist, bedeutet dies, dass eingehende Abhängigkeiten vonseiten anderer Komponenten bestehen (*Fan-in>0*), sie selbst jedoch von keinen anderen Komponenten abhängig ist (*Fan-out=0*). In dieser Konstellation ist die Komponente *verantwortlich* und *nicht abhängig* – und damit so stabil wie es nur geht. Die von ihr abhängigen Komponenten erschweren Modifikationen und sie selbst hat keine ausgehenden Abhängigkeiten, die Anpassungen an ihr erzwingen könnten.

Das SDP besagt, dass die Messgröße *I* einer Komponente größer als die Messgröße *I* der von ihr abhängigen Komponenten sein sollte. Oder, anders ausgedrückt: Die Messgröße *I* sollte in Richtung der Abhängigkeit abnehmen.

14.3.3 Nicht alle Komponenten sollten stabil sein

Wenn alle Komponenten in einem System maximal stabil wären, wäre das System damit gleichzeitig unveränderbar. Doch das ist in aller Regel kein wünschenswerter Zustand. Im Grunde genommen wollen die Softwareentwickler ihre Komponentenstruktur doch so designen, dass einige Komponenten instabil und andere stabil sind. Die schematische Darstellung in Abbildung 14.8 zeigt eine ideale Konfiguration für ein System mit drei Komponenten.

Die änderbaren Komponenten sind oben im Diagramm angeordnet und von der stabilen Komponente darunter abhängig. Die instabilen Komponenten im oberen Bereich des Schemas zu platzieren, ist insofern nützlich und üblich, weil bei dieser Darstellungsweise jeder Pfeil, der *nach oben* weist, schnell erkennbar einen Verstoß gegen das SDP darstellt (und, wie später noch erläutert wird, auch gegen das ADP).

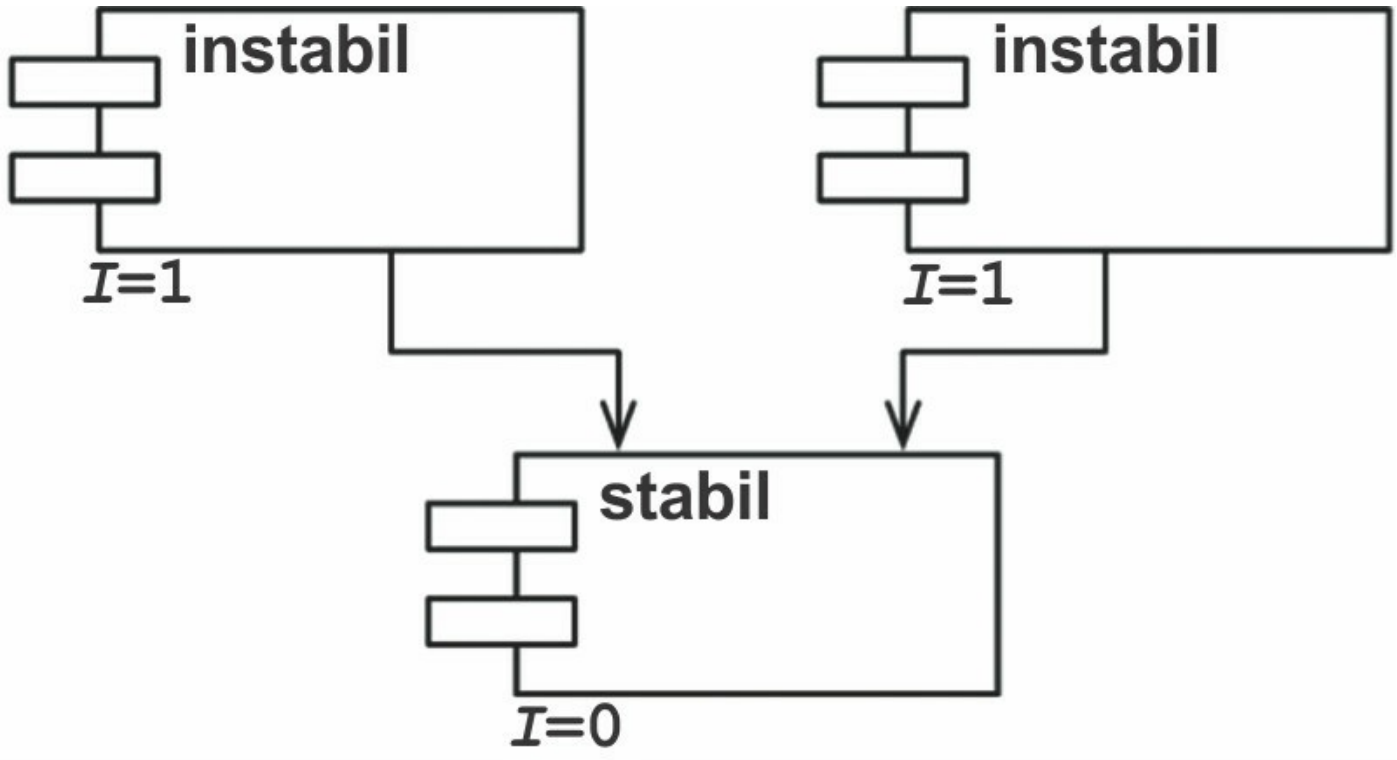


Abb. 14.8: Eine ideale Konfiguration für ein System mit drei Komponenten

Das Diagramm in Abbildung 14.9 zeigt, wie gegen das SDP verstoßen werden kann.

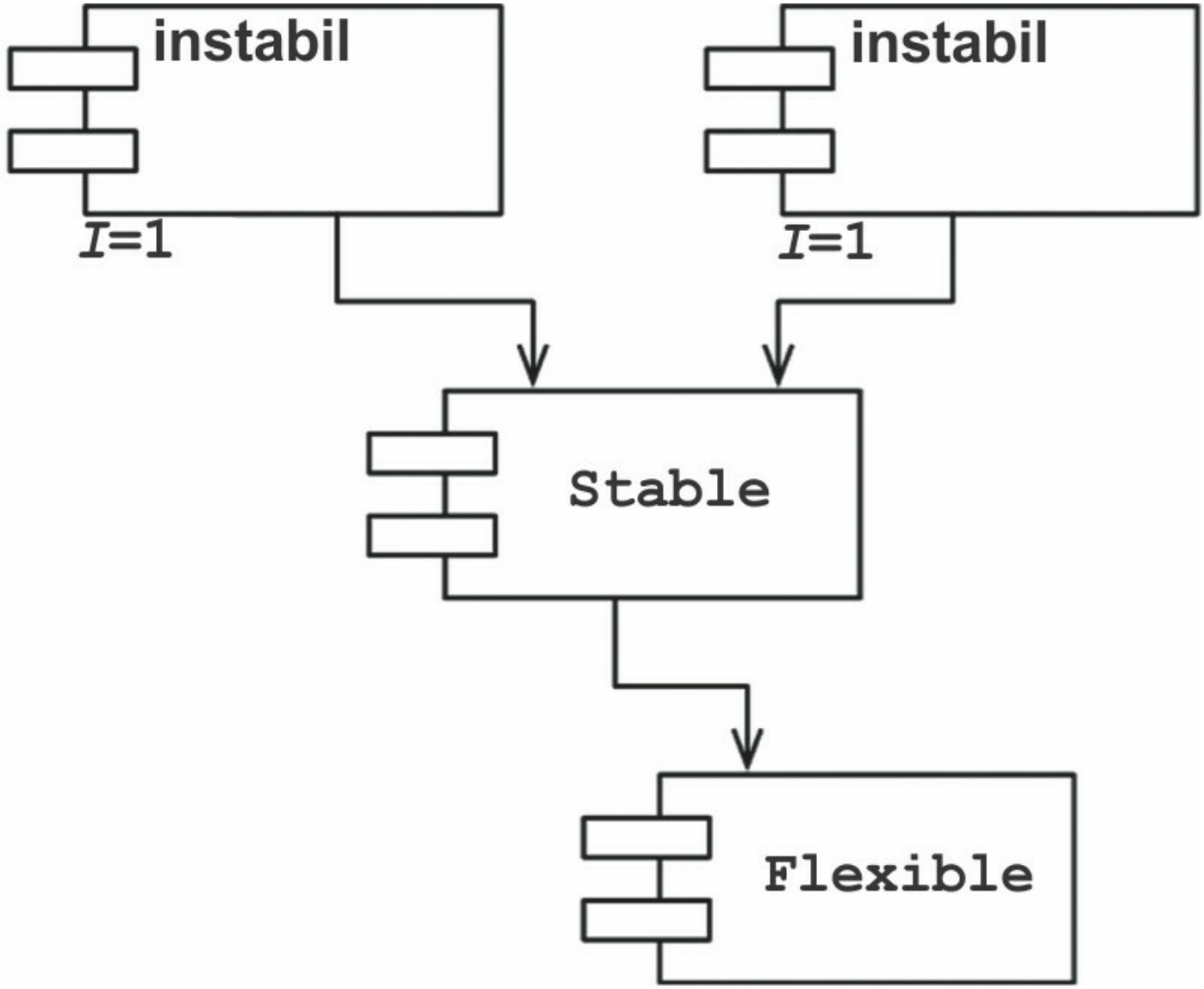


Abb. 14.9: Verstoß gegen das SDP

Die Komponente `Flexible` wurde ganz im Sinne des einfachen Modifizierens designt. Sie soll also instabil sein. Allerdings hat irgendein anderer Softwareentwickler, der an der Komponente mit dem Namen `Stable` arbeitet, eine Abhängigkeit zu `Flexible` vorgesehen. Und das wiederum verstößt gegen das SDP, weil die Messgröße *I* für `Stable` deutlich kleiner ist als für `Flexible`. Daher ist `Flexible` nicht länger einfach zu modifizieren: Eine Änderung an `Flexible` erzwingt nun auch die Berücksichtigung der Komponente `Stable` und sämtlicher Klassen, die von ihr abhängig sind.

Um dieses Problem zu bewältigen, muss nun irgendwie die Abhängigkeit von `Stable` gegenüber `Flexible` durchbrochen werden. Warum also existiert diese Abhängigkeit? Angenommen, innerhalb von `Flexible` gäbe es eine Klasse `C`, die von einer weiteren Klasse `U` innerhalb von `Stable` verwendet werden muss (siehe Abbildung 14.10).

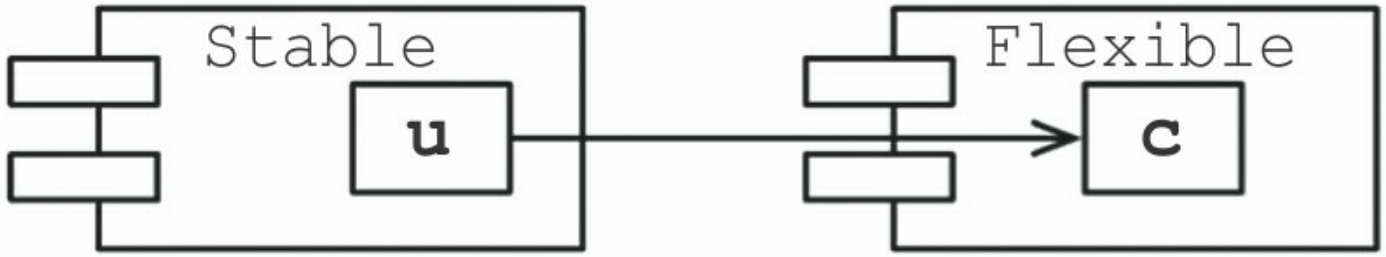


Abb. 14.10: Die Klasse `u` in der Komponente `Stable` verwendet die Klasse `c` der Komponente `Flexible`.

Die Lösung ist hier die Anwendung des DIPs: Wir erzeugen eine Schnittstellenklasse namens `US` und setzen sie in eine Komponente, die wir mit `UServer` benennen. In dieser Schnittstelle müssen alle Methoden deklariert sein, die von der Klasse `u` benötigt werden. Anschließend wird die Schnittstelle von `c` implementiert, wie in Abbildung 14.11 zu sehen. Dadurch wird die Abhängigkeit von `Stable` gegenüber `Flexible` durchbrochen und beide Komponenten werden zwangsweise von `UServer` abhängig. `UServer` ist sehr stabil (*I=0*) und die Komponente `Flexible` behält ihre notwendige Instabilität (*I=1*). Alle Abhängigkeiten fließen nun in Richtung *Verringerung von I*.

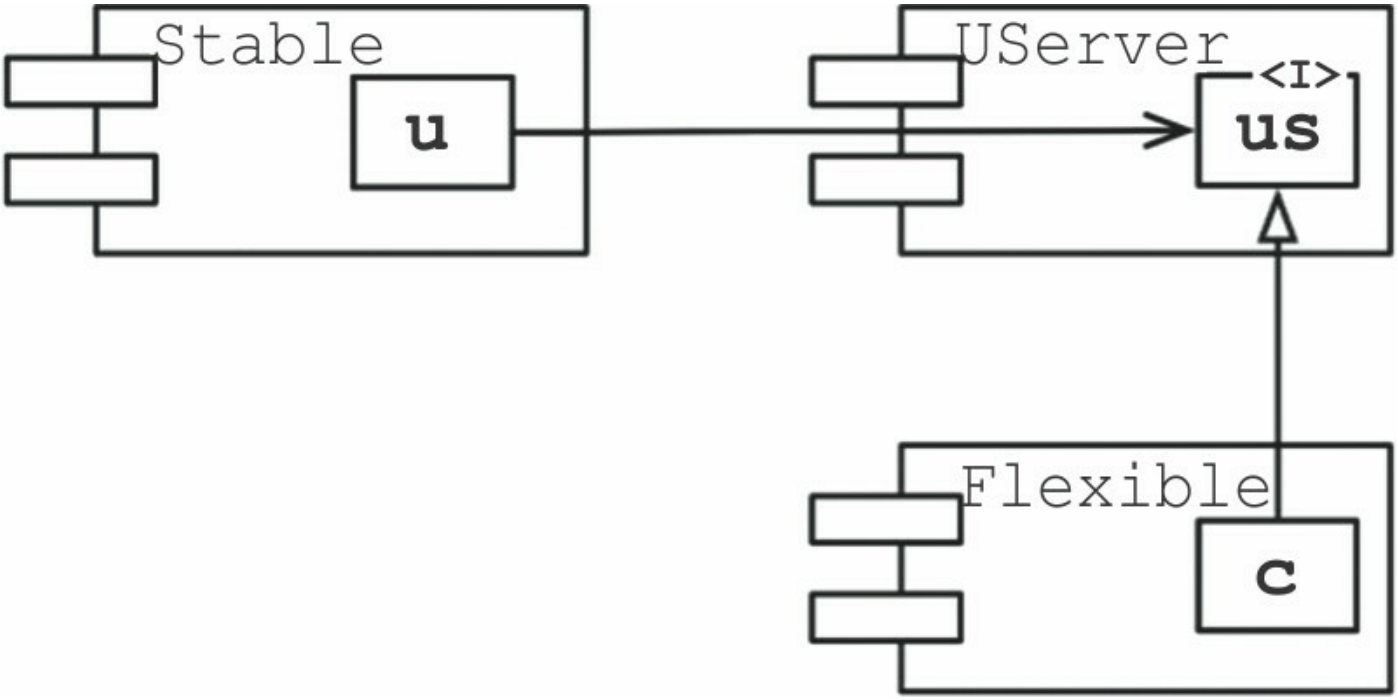


Abb. 14.11: `c` implementiert die Schnittstellenklasse `us`.

14.3.4 Abstrakte Komponenten

Vermutlich wundern Sie sich darüber, dass hier eine Komponente erzeugt wird – `UService` –, in der sich ausschließlich eine Schnittstelle befindet. Eine solche Komponente enthält keinen ausführbaren Code! Das ist jedoch beim Einsatz statisch

typisierter Programmiersprachen wie Java und C# eine sehr gängige und notwendige Taktik. Diese *abstrakten Komponenten* sind sehr stabil und daher ideale Ziele für eingehende Abhängigkeiten seitens weniger stabiler Komponenten.

In dynamisch typisierten Sprachen wie Ruby und Python existieren weder solche abstrakten Komponenten noch Abhängigkeiten, die auf sie abzielen würden. Die Abhängigkeitsstrukturen in diesen Programmiersprachen sind um vieles simpler, weil die Abhängigkeitsumkehr weder die Deklaration noch die Vererbung von Schnittstellen erfordert.

14.4 SAP: Das Stable-Abstractions-Prinzip

Eine Komponente sollte ebenso abstrakt sein, wie sie stabil ist.

14.4.1 Wo werden die übergeordneten Richtlinien hinterlegt?

Der Teil der Software in einem System, der die übergeordneten Entscheidungen zur Architektur und den Richtlinien (engl. *Policy*) repräsentiert, sollte aus einleuchtenden Gründen natürlich nicht allzu häufig modifiziert werden. Diese geschäftlichen und strukturellen Maßgaben sollten keinesfalls flüchtig sein. Dementsprechend macht es am meisten Sinn, die Software, die die übergeordneten Richtlinien des Systems kapselt, in stabile Komponenten (*I=0*) zu platzieren. Instabile Komponenten (*I=1*) sollten hingegen nur flüchtige Software enthalten – also solche, die schnell und einfach zu modifizieren sein soll.

Wenn die übergeordneten Richtlinien allerdings in stabilen Komponenten untergebracht werden, dann wird es schwierig, Modifikationen an dem Quellcode vorzunehmen, der diese Richtlinien umsetzt – und dadurch könnte die gesamte Softwarearchitektur unflexibel werden. Wie also kann eine Komponente, die maximal stabil ist (*I=0*), auch flexibel genug sein, um Anpassungen standzuhalten? Die Antwort auf diese Frage findet sich im OCP, das besagt, dass es möglich und durchaus wünschenswert ist, Klassen zu erzeugen, die flexibel genug sind, um Erweiterungen zuzulassen, ohne dass Modifikationen erforderlich sind. Und welche Klassentypen sind mit diesem Prinzip konform? *Abstrakte* Klassen.

14.4.2 Einführung in das SAP (Stable-Abstractions-Prinzip)

Das *Stable-Abstractions-Prinzip* stellt eine Beziehung zwischen Stabilität und Abstraktheit her. Einerseits besagt es, dass eine stabile Komponente zugleich auch

abstrakt sein sollte, damit ihre Stabilität nicht verhindert, dass sie erweitert werden kann. Andererseits gibt es aber auch vor, dass eine instabile Komponente konkret sein sollte, weil ihre Instabilität es ermöglicht, den in der Komponente enthaltenen Code problemlos zu modifizieren.

Demzufolge sollte eine Komponente, wenn sie stabil sein soll, aus Schnittstellen und abstrakten Klassen bestehen, damit sie erweitert werden kann. Stabile Komponenten, die gleichzeitig auch erweiterbar sind, erweisen sich als flexibel und beschränken zudem die Softwarearchitektur nicht übermäßig.

Das SAP und das SDP ergeben zusammengenommen das »DIP für Komponenten« – und zwar weil das SDP vorgibt, dass Abhängigkeiten proportional in Richtung Stabilität verlaufen sollten, und das SAP besagt, dass Stabilität proportional zur Abstraktion verläuft. Dementsprechend *verlaufen Abhängigkeiten in Richtung Abstraktion*.

Das eigentliche DIP befasst sich hingegen mit Klassen – und wenn es um Klassen geht, gibt es keine Grauzonen: Eine Klasse ist entweder abstrakt oder sie ist es nicht. Die Kombination von SDP und SAP zielt im Gegensatz dazu jedoch auf den Umgang mit Komponenten ab. Sie ermöglicht, dass eine Komponente zum Teil abstrakt und zum Teil stabil sein kann.

14.4.3 Bemessung der Abstraktion

Die Messgröße *A* beschreibt den Grad der Abstraktheit einer Komponente. Ihr Wert gibt einfach das Verhältnis der Schnittstellen und abstrakten Klassen in einer Komponente zur Gesamtanzahl der Klassen in derselben Komponente wieder.

- **Nc:** Die Anzahl der Klassen in der Komponente.
- **Na:** Die Anzahl der abstrakten Klassen und Schnittstellen in der Komponente.
- **A:** Grad der Abstraktheit. *A=Na/Nc*.

Der Messwertbereich von *A* liegt bei 0 bis 1. Ein Wert von `0` bedeutet hier, dass die Komponente keinerlei abstrakte Klassen enthält, und der Wert `1` gibt an, dass die Komponente ausschließlich aus abstrakten Klassen besteht.

14.4.4 Die Hauptreihe

Damit sind wir an dem Punkt angekommen, an dem es Zeit ist, die Beziehung zwischen der Stabilität (*I*) und der Abstraktheit (*A*) zu definieren. Dazu wird im

Folgenden ein Graph mit `A` auf der vertikalen Achse und `I` auf der horizontalen Achse erstellt (siehe Abbildung 14.12). Bildet man nun die beiden »guten« Komponentenarten darauf ab, finden sich die Komponenten mit maximaler Stabilität und Abstraktheit in der oberen linken Ecke bei `(0,1)` wieder. Die Komponenten, die maximal instabil und konkret sind, werden in der unteren rechten Ecke bei `(1,0)` aufgeführt.

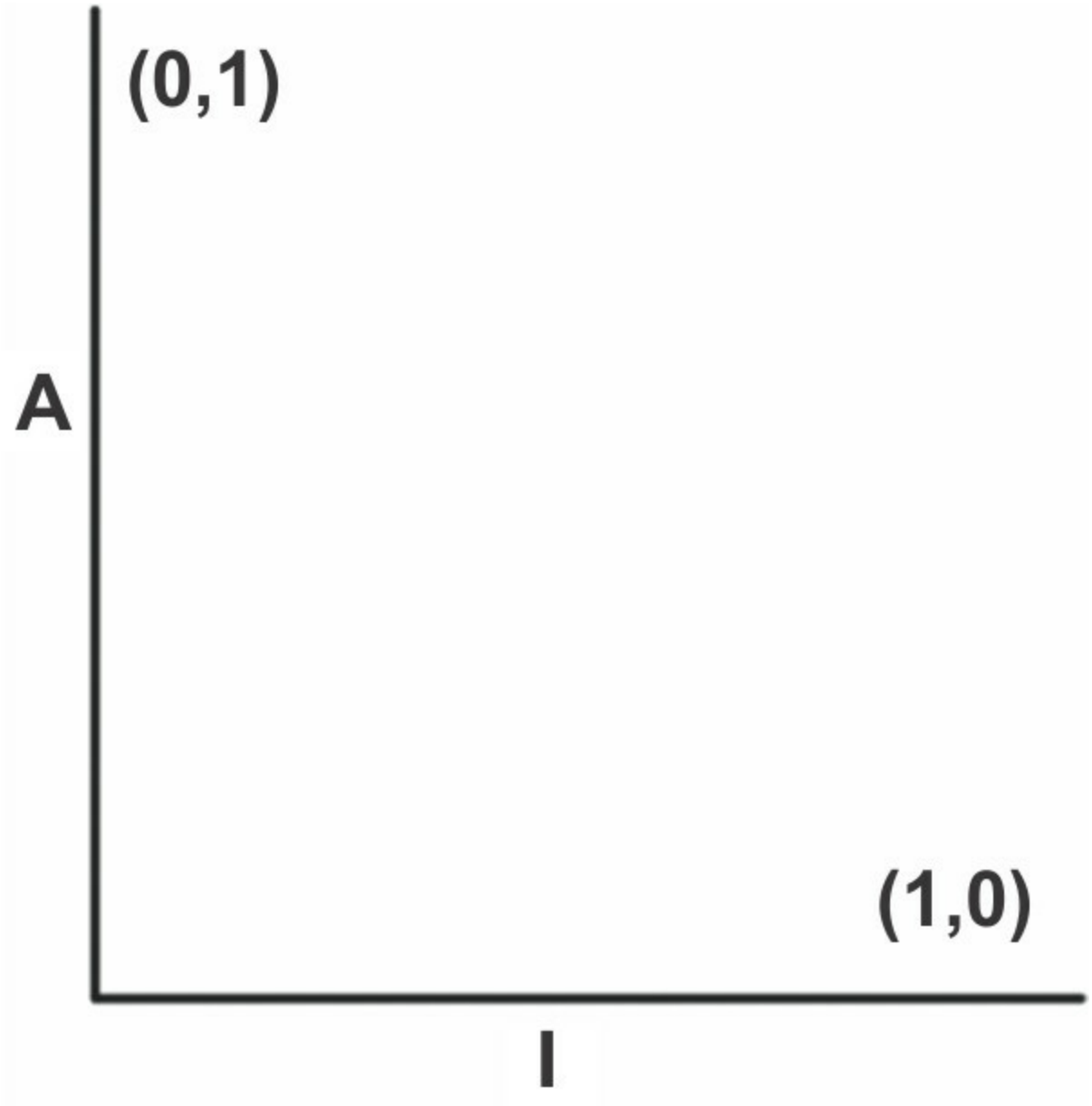


Abb. 14.12: Der I/A-Graph

Allerdings landen nicht alle Komponenten an einer dieser beiden Positionen, weil sich häufig *Gradabstufungen* hinsichtlich ihrer Abstraktion und Stabilität ergeben. Beispielsweise ist es durchaus üblich, dass eine abstrakte Klasse von einer anderen abstrakten Klasse abgeleitet ist. Die Ableitung ist damit eine Abstraktion, die in einem Abhängigkeitsverhältnis steht. Und daher wird sie, obwohl sie maximal abstrakt ist, nicht auch maximal stabil sein, denn ihre Abhängigkeit verringert ihre Stabilität.

Weil sich keine Regel erzwingen lässt, mit der alle Komponenten entweder einen Wert von `(0,1)` oder von `(1,0)` erreichen, muss davon ausgegangen werden, dass sich eine Punktesammlung auf dem *A/I*-Graphen ergibt, der angemessene Positionen für die Komponenten definiert. Diese geometrischen Standorte lassen sich bestimmen, indem Sie die Bereiche ausfindig machen, in denen sich keine Komponenten befinden sollten – mit anderen Worten durch die Bestimmung der *Ausschlusszonen* (engl. *Zones of Exclusion*) (siehe Abbildung 14.13).

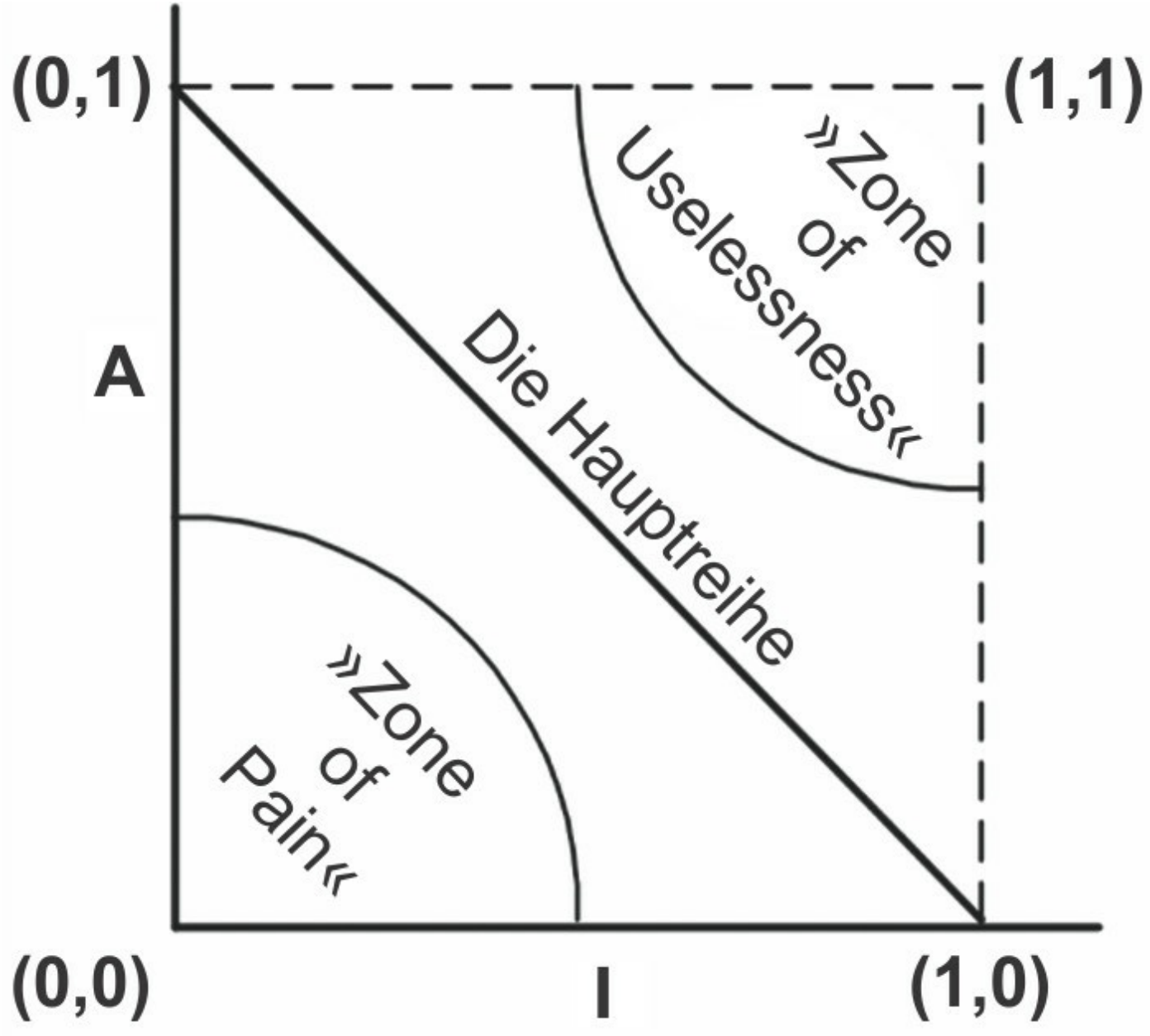


Abb. 14.13: Ausschlusszonen

14.4.5 Die »Zone of Pain«

Stellen Sie sich einmal eine Komponente im Wertebereich von `(0,0)` vor. Hierbei würde es sich um eine äußerst stabile und konkrete Komponente handeln, die aufgrund

ihrer nicht vorhandenen Flexibilität nicht wünschenswert wäre, denn: Sie ließe sich nicht erweitern, weil sie nicht abstrakt ist, und sie wäre wegen ihrer Stabilität nur sehr schwer zu modifizieren. Deshalb befinden sich gut designte Komponenten normalerweise nicht in der Nähe der Koordinate `(0,0)`, denn dieser Bereich ist eine *Ausschlusszone*, die sogenannte *Zone of Pain* (zu Deutsch etwa »Schmerzzone«).

Einige Softwareentitäten fallen jedoch tatsächlich in die »Zone of Pain«. Ein Beispiel hierfür wäre ein Datenbankschema. Solche Schemata sind notorisch flüchtig, extrem konkret und mit einer hohen Zahl von eingehenden Abhängigkeiten verbunden. Dies ist einer der Gründe, warum die Schnittstelle zwischen objektorientierten Anwendungen und Datenbanken so schwierig zu verwalten ist und warum Aktualisierungen des Schemas generell »schmerzlich« sind.

Ein weiteres Beispiel für Software, die im Umfeld der Koordinate `(0,0)` angesiedelt ist, wäre eine konkrete *Utility Library* (zu Deutsch »Dienstbibliothek«). Auch wenn eine solche Library einen *I*-Messwert von `1` hat, kann sie dennoch nichtflüchtig sein. Denken Sie dabei zum Beispiel an die Komponente `String`: Obwohl alle in dieser Komponente enthaltenen Klassen konkret sind, wird sie doch so häufig verwendet, dass eine daran vorgenommene Modifikation Chaos verursachen würde. Deshalb ist `String` nichtflüchtig.

Nichtflüchtige Komponenten sind im Wertebereich `(0,0)` harmlos, weil es eher unwahrscheinlich ist, dass sie modifiziert werden müssen. Deshalb sind nur flüchtige Softwarekomponenten in der Zone of Pain problematisch – daher auch der Name dieser Ausschlusszone: Je flüchtiger eine Komponente ist, desto »schmerzvoller« (painful) ist sie. Im Grunde genommen bildet die Flüchtigkeit eine Art dritte Achse in dem *I/A*-Graphen. So gesehen ist in Abbildung 14.13 nur die schmerzvollste Ebene dargestellt, in der die Flüchtigkeit `=1` beträgt.

14.4.6 Die »Zone of Uselessness«

Stellen Sie sich als Nächstes eine Komponente in der Nähe der Koordinate `(1,1)` vor. Eine Positionierung in diesem Wertebereich wäre ebenfalls nicht wünschenswert, weil die Koordinate in diesem Fall zwar maximal abstrakt wäre, aber keine eingehenden Abhängigkeiten aufweisen würde. Komponenten dieser Art sind nutzlos, weswegen dieser Bereich ebenfalls eine Ausschlusszone darstellt, die als *Zone of Uselessness* (zu Deutsch etwa »Zone der Nutzlosigkeit«) bezeichnet wird.

Softwareentitäten, die sich in dieser Zone befinden, verkörpern eine Art Zerfallsprodukte. Nicht selten handelt es sich hierbei um übrig gebliebene abstrakte Klassen, die nie implementiert wurden und sich hin und wieder in der Codebasis eines Systems finden, wo sie ungenutzt vor sich hinvegetieren.

Eine Komponente, die sich an irgendeiner Position tief im Innern der »Zone of Uselessness« befindet, muss zwangsläufig einen maßgeblichen Anteil solcher Entitäten enthalten. Dabei steht allerdings auch außer Frage, dass das Vorhandensein derartig nutzloser Entitäten in jedem Fall unerwünscht ist.

14.4.7 Die Ausschlusszonen vermeiden

Es erscheint offensichtlich, dass die flüchtigsten Komponenten unserer Systeme so weit wie nur möglich von beiden Ausschlusszonen ferngehalten werden sollten. Die Punktesammlung, die am weitesten von jeder dieser Zonen entfernt ist, bildet die Verbindungslinie zwischen `(1,0)` und `(0,1)`. Ich bezeichne diese Linie als die *Hauptreihe*.[2]

Eine Komponente, die sich unmittelbar auf der Hauptreihe befindet, ist weder »zu abstrakt« für ihre Stabilität, noch »zu instabil« für ihre Abstraktheit. Sie ist weder nutzlos (*Uselessness*) noch besonders schmerzlich (*Pain*). Ihre eingehenden Abhängigkeiten beruhen darauf, inwieweit sie abstrakt ist, und ihre ausgehenden Abhängigkeiten basieren darauf, inwieweit sie konkret ist.

Die erstrebenswerteste Position für eine Komponente ist an einem der beiden Endpunkte der Hauptreihe. Gute Softwarearchitekten versuchen, den Großteil ihrer Komponenten nach Möglichkeit genau dort zu platzieren. Erfahrungsgemäß ist ein kleiner Teil der Komponenten eines umfangreichen Systems in aller Regel jedoch weder vollkommen abstrakt noch vollkommen stabil. Hier kommen dann die besten Eigenschaften zum Tragen, wenn sie sich *auf oder in unmittelbarer Nähe* der Hauptreihe befinden.

14.4.8 Abstand von der Hauptreihe

Und damit wären wir auch schon bei der letzten Messgröße angelangt. Wenn sich Komponenten auf oder in der Nähe der Hauptreihe befinden sollen, dann lässt sich dazu eine Messgröße erzeugen, die angibt, wie weit eine Komponente von ihrer idealen Position entfernt ist.

- **D**[3]**:** Distanz. $D=|A+I-1|$. Der Wertebereich dieser Messgröße beträgt `[0,1]`. Dabei gibt der Wert `0` an, dass sich die Komponente unmittelbar auf der Hauptreihe befindet. Der Wert `1` weist hingegen aus, dass die Komponente in der größtmöglichen Distanz zur Hauptreihe positioniert ist.

Mithilfe dieser Messgröße lässt sich ein Softwaredesign auf seine allgemeine Konformität mit der Hauptreihe hin analysieren. Der Wert von *D* kann für jede einzelne Komponente berechnet werden. Und jede Komponente, die einen *D*-Wert hat,

der nicht in der Nähe von `0` ist, kann dann erneut untersucht und neu strukturiert werden.

Auch eine statistische Analyse eines Softwaredesigns ist möglich. So lassen sich das Mittel und die Varianz aller *D*-Werte für Komponenten in einem Design berechnen. In einem konformen Softwaredesign müsste sich beides nahe `0` bewegen. Die Varianz kann genutzt werden, um »Kontrollgrenzen« zur Identifizierung von Komponenten zu errichten, die im Verhältnis zu allen anderen Komponenten »aus dem Rahmen« fallen.

Das Streudiagramm in Abbildung 14.14 zeigt, dass sich der überwiegende Teil der Komponenten zwar entlang der Hauptreihe befindet, gleichzeitig aber auch einige über die erste Standardabweichungszone (`Z=1`) hinaus von ihr entfernt sind. Diese abweichenden Komponenten lohnen eine genauere Betrachtung. Aus irgendeinem Grund sind sie entweder sehr abstrakt und verfügen über sehr wenige eingehende Abhängigkeiten, oder sie sind sehr konkret und es sind sehr viele andere Komponenten von ihnen abhängig.

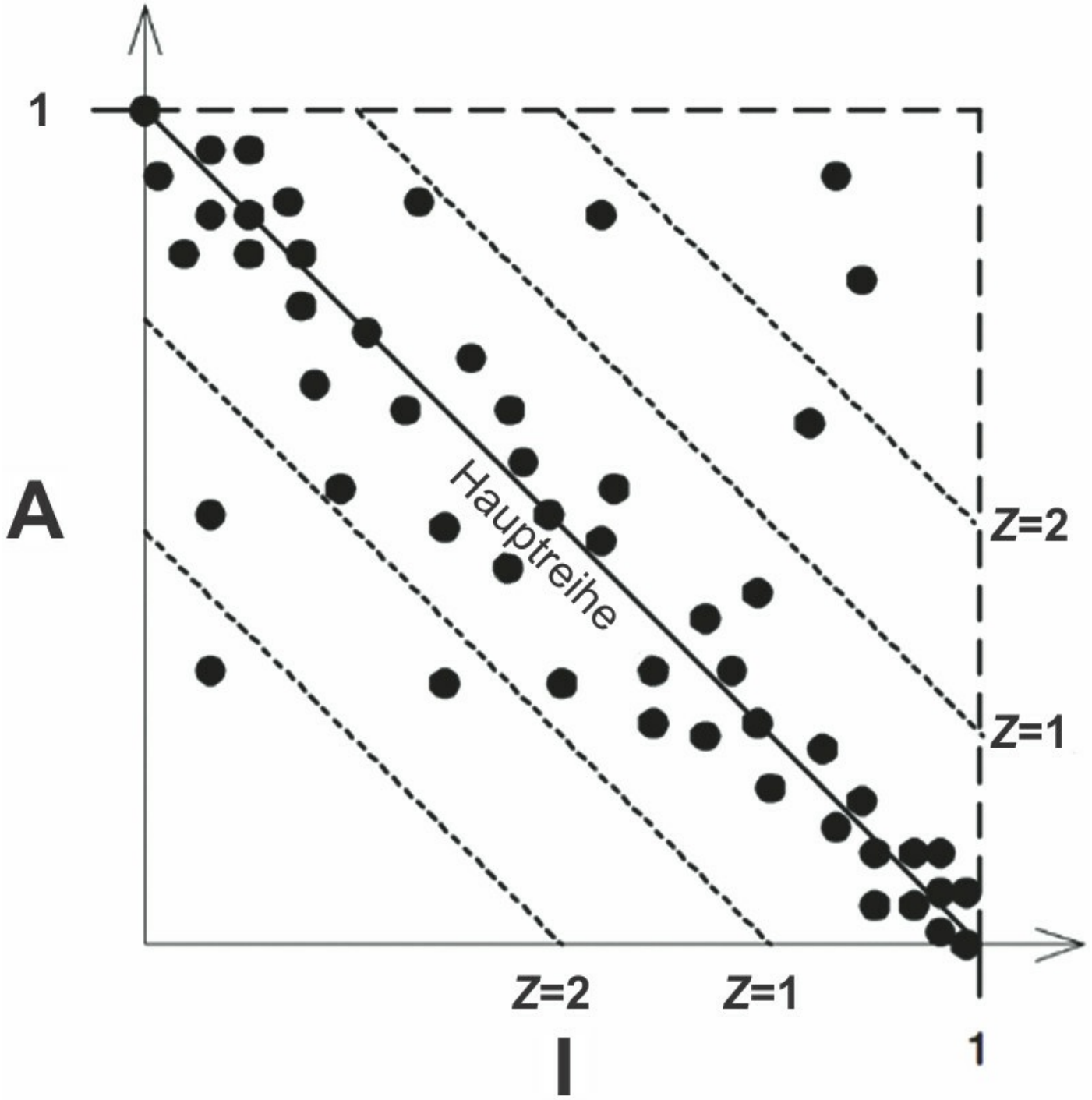


Abb. 14.14: Streudiagramm der Komponenten

Eine andere Methode zur Nutzung der Messgrößen besteht darin, den Wert von *D* für jede Komponente im Zeitverlauf darzustellen. Dieses Verfahren wird in der in Abbildung 14.15 dargestellten Grafik simuliert. Hier ist zu sehen, dass sich im Laufe der letzten paar Releases einige merkwürdige Abhängigkeiten in die Komponente `Payroll` eingeschlichen haben. Die Grafik zeigt eine Kontrollgrenze an `D=0,1`. Der Punkt bei `R2.1` hat diese Grenze überschritten, deshalb sollte in diesem Fall untersucht werden, warum sich die betreffende Komponente überhaupt so weit von der Hauptreihe entfernt hat.

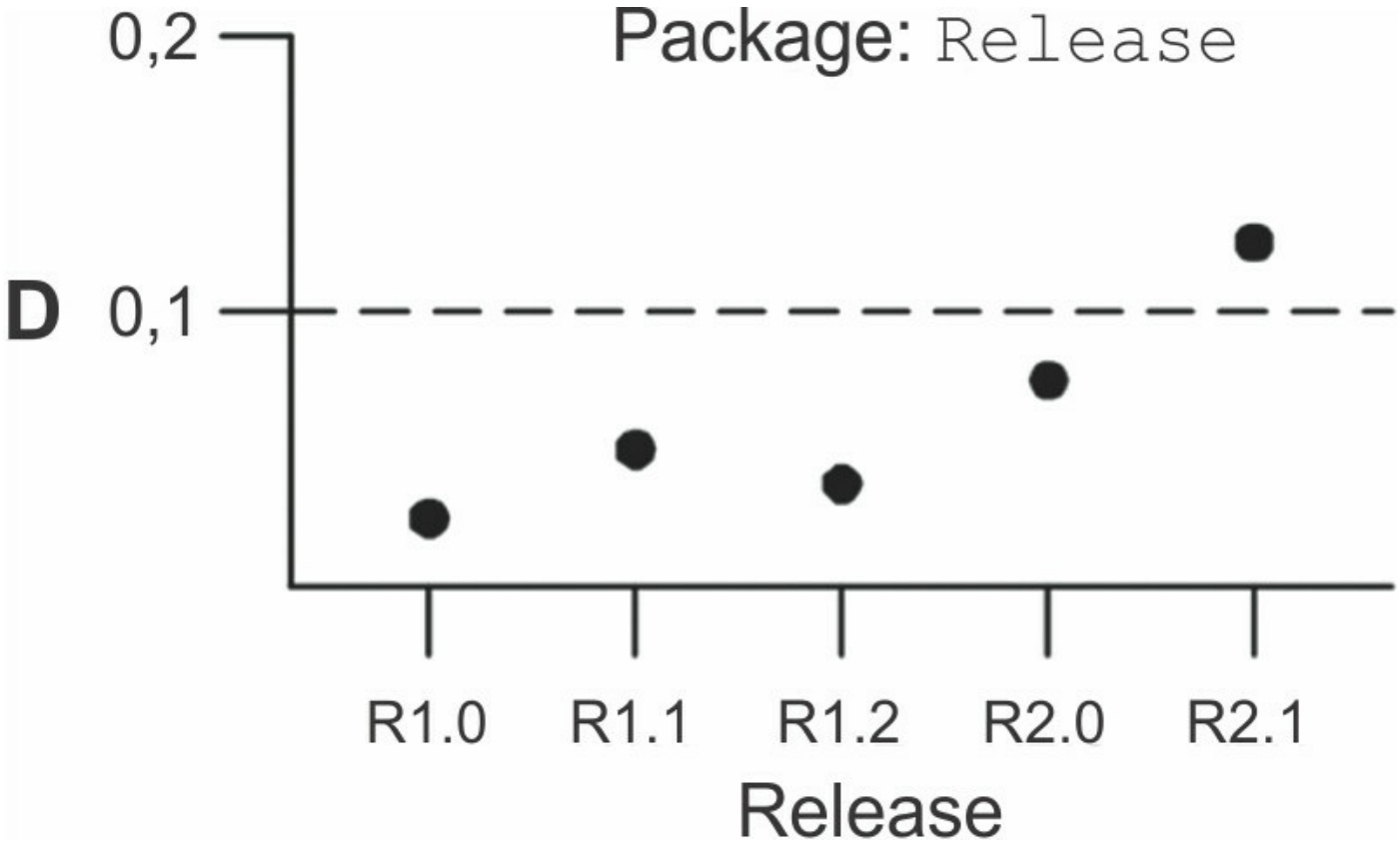


Abb. 14.15: Grafische Darstellung des D-Wertes für eine einzelne Komponente im Zeitverlauf

14.5 Fazit

Die in diesem Kapitel beschriebenen *Metriken für das Abhängigkeitsmanagement* bemessen die Konformität eines Softwaredesigns im Sinne eines Abhängigkeits- und Abstraktionsmusters, das ich persönlich gut finde. Die praktische Erfahrung zeigt, dass bestimmte Abhängigkeiten besser sind als andere – und das spiegelt sich in diesem Muster wider. Andererseits ist aber auch ein Messwert nicht unfehlbar, sondern bloß ein Maßstab gegenüber einem beliebigen Standard. Die erwähnten Metriken sind bestenfalls unvollkommen, dennoch hoffe ich, dass sie Ihnen von Nutzen sein werden.

---

[1] In früheren Veröffentlichungen habe ich die *Fan-out*-Methode als *efferente* (ableitende) Kopplung (Ce) und die *Fan-in*-Methode als *afferente* (zuleitende) Kopplung (Ca) bezeichnet. Diese metaphorische Begriffsverwendung in Anlehnung an das zentrale Nervensystem war allerdings etwas anmaßend und im Kontext der Softwareentwicklung nicht wirklich zutreffend.

[2] Für meine Arroganz, einen so wichtigen Begriff aus der Astronomie übernommen zu haben, bitte ich um Nachsicht.

[3] In früheren Veröffentlichungen habe ich diese Messgröße mit D' bezeichnet, inzwischen sehe ich aber keine Veranlassung mehr, dies in der Praxis fortzuführen.

Teil V
Softwarearchitektur

In diesem Teil:

Kapitel 15

Was ist Softwarearchitektur?

Der Begriff »Architektur« ruft allgemein Visionen von Macht und Mysterien hervor und wird mit gewichtigen Entscheidungen und technischer Überlegenheit assoziiert. Die Softwarearchitektur im Speziellen verkörpert den Zenit der technischen Errungenschaften. Unter einem Softwarearchitekten stellen wir uns jemanden vor, der Autorität, Einfluss und Respekt genießt. Welcher junge angehende Softwareentwickler hat wohl noch nicht davon geträumt, eines Tages Softwarearchitekt zu sein?

Aber was ist unter einer »Softwarearchitektur« eigentlich zu verstehen? Was genau macht ein Softwarearchitekt und wann wird er aktiv?

Zuallererst einmal ist ein Softwarearchitekt natürlich ein Programmierer – und das bleibt er auch. Lassen Sie sich nicht vorgaukeln, Softwarearchitekten würden den Code als solches plötzlich links liegen lassen und sich nur noch auf die höherrangigeren Problemstellungen konzentrieren. Dem ist ganz und gar nicht so! Im Gegenteil: Softwarearchitekten sind die besten Programmierer überhaupt – und sie übernehmen auch weiterhin Programmieraufgaben, während sie *zusätzlich* noch das übrige Entwicklerteam an ein Softwaredesign heranführen, das die Produktivität maximiert. Sie mögen zwar vielleicht nicht ganz so viel Code schreiben wie andere Programmierer, aber sie sind durchaus weiterhin mit Programmiertätigkeiten befasst – weil sie ihren Job schlicht und ergreifend nicht beherrschen würden, wenn sie sich nicht auch selbst mit den Schwierigkeiten auseinandersetzen würden, mit denen sie die restlichen Programmierer konfrontieren.

Die Architektur eines Softwaresystems entspricht der Form, die diesem System von seinen Schöpfern gegeben wird. Und die Ausgestaltung dieser Form ergibt sich aus der Komponentenunterteilung des Systems, der Anordnung dieser Komponenten sowie der Art und Weise, wie sie miteinander kommunizieren.

Sinn und Zweck der Systemform ist es, die Entwicklung, das Deployment, den Betrieb und die Instandhaltung des darin enthaltenen Softwaresystems zu erleichtern.

> *Die dahinterstehende Strategie verfolgt das Ziel, so lange wie möglich so viele Optionen wie möglich offenzuhalten.*

Nun mag Sie diese Aussage überraschen – vielleicht haben Sie ja gedacht, die Zielsetzung der Softwarearchitektur bestünde lediglich darin, dafür Sorge zu tragen, dass das System vernünftig funktioniert. Zweifellos ist dem auch so und sicherlich gehört dies zu den obersten Prioritäten einer Systemarchitektur.

Allerdings hat die Architektur eines Softwaresystems nur wenig Einfluss darauf, ob das betreffende System letztendlich funktioniert – immerhin gibt es jede Menge Systeme mit fürchterlichen Architekturen, die trotz allem recht gut arbeiten. Die hier auftretenden Probleme liegen meist gar nicht so sehr in deren Betriebsfähigkeit begründet, sondern vielmehr in ihrem Deployment, ihrer Instandhaltung und ihrer fortgesetzten Entwicklung.

Das soll nun aber keinesfalls heißen, dass die Softwarearchitektur bei der Unterstützung des richtigen Systemverhaltens keine Rolle spielen würde – denn das tut sie definitiv, sogar eine bedeutende. Allerdings ist diese Rolle eher passiver und kosmetischer als aktiver oder maßgeblicher Natur. Es gibt nur wenige, wenn überhaupt irgendwelche Optionen, die die Architektur hinsichtlich des *Verhaltens* offenlassen kann.

Der vorrangigste Zweck der Softwarearchitektur besteht jedoch darin, den Lebenszyklus des Systems zu sichern. Eine gute Architektur sorgt dafür, dass das

System leicht zu verstehen, einfach zu entwickeln, problemlos instand zu halten und reibungslos zu deployen ist. Die ultimativen Ziele sind dabei die Minimierung der *Lifetime Costs* (zu Deutsch »Gesamtlebenszeitkosten«) und die Maximierung der Produktivität der Programmierer.

15.1 Entwicklung

Ein Softwaresystem, das schwierig zu entwickeln ist, hat wenig Aussicht auf eine lange und »gesunde« Lebensdauer. Deshalb sollte seine Architektur so ausgelegt sein, dass sie dem Team die Entwicklungsarbeit möglichst leicht macht.

Dabei bedingen unterschiedliche Teamstrukturen selbstverständlich auch unterschiedliche architektonische Entscheidungen. So kann ein kleines Team aus fünf Softwareentwicklern bei der Entwicklung eines monolithischen Systems ohne klar definierte Komponenten und Schnittstellen recht effizient zusammenarbeiten. Tatsächlich würde ein solches Team die Strukturen einer Softwarearchitektur in der frühen Entwicklungsphase wahrscheinlich sogar eher als hinderlich betrachten. Das ist vermutlich auch der Grund, warum so viele Systeme keine gute Architektur haben: Sie wurde zu Beginn der Entwicklungsarbeit erst einmal außer Acht gelassen, weil das kleine Team sich zunächst nicht mit einer übergeordneten Struktur belasten wollte.

Andererseits kann ein System, das von fünf verschiedenen Teams entwickelt wird, von denen jedes aus sieben Programmierern besteht, keine Fortschritte erzielen, solange das System nicht in klar definierte Komponenten mit zuverlässigen, stabilen Schnittstellen strukturiert ist. Lässt man alle anderen Faktoren außer Acht, würde dies sehr wahrscheinlich dazu führen, dass das System schlussendlich zu genau fünf Komponenten tendiert – je einer pro Team.

Eine solche »Eine Komponente pro Team«-Struktur ist jedoch höchstwahrscheinlich nicht die beste Architekturform für das Deployment, den Betrieb und die Instandhaltung des Systems – dennoch ist es genau die Art von Architektur, auf die eine Gruppe von Teams zusteuern wird, wenn sie sich ausschließlich von dem Zeitplan des Entwicklungsprojekts leiten lässt.

15.2 Deployment

Softwaresysteme müssen deploybar sein, um effizient sein zu können. Je höher der Aufwand für das Deployment, desto weniger nützlich ist das System. Ein Ziel der Softwarearchitektur sollte deshalb sein, das System so zu errichten, dass es einfach *mit einer einzigen Aktion* deployt werden kann.

Leider wird der Deployment-Strategie zu Beginn der Entwicklung meist kaum Beachtung geschenkt. Und das führt dann zu Architekturen, durch die die Systeme zwar einfach zu entwickeln, aber nur schwer zu deployen sind.

Beispielsweise könnten sich die Programmierer in der Frühphase der Systementwicklung für eine *Microservice-Architektur* entscheiden – weil sie der Auffassung sind, dass dieser Ansatz die Entwicklungsarbeit sehr erleichtert, da es den Systemkomponenten enge Grenzen setzt und die Schnittstellen relativ stabil sind. Wenn es dann jedoch an das Deployment geht, werden sie allerdings möglicherweise feststellen, dass sich inzwischen eine geradezu beängstigende Anzahl von Microservices angehäuft hat, sodass sich die Konfiguration der zugehörigen Verbindungen ebenso wie das Timing ihrer Initiierung als enorm große Fehlerquelle erweisen könnte.

Hätten die Softwarearchitekten hingegen die mit dem Deployment einhergehenden Problemstellungen bereits früher mit berücksichtigt, hätten sie sich vielleicht für weniger Services, eine Kombination aus Services und Prozesskomponenten und eine integralere Möglichkeit der Verbindungsverwaltung entschieden.

15.3 Betrieb

Die Auswirkungen der Softwarearchitektur auf den Systembetrieb sind in aller Regel weniger dramatisch als ihr Einfluss auf Entwicklung, Deployment und Instandhaltung. Fast jedes betriebsbezogene Problem im System lässt sich durch die Ergänzung weiterer Hardware beheben, ohne dass dies drastische Konsequenzen für die Softwarearchitektur hätte.

Derartige Dinge kommen sogar relativ häufig vor. Softwaresysteme mit wenig effizienten Architekturen lassen sich oftmals problemlos durch die Ergänzung weiterer Speicherkapazitäten oder Server effizienter gestalten. Die Tatsache, dass Hardware preiswert und Personal teuer ist, bedeutet, dass Architekturen, die den Systembetrieb beeinträchtigen, nicht so kostenintensiv sind wie Architekturen, die die Entwicklung, das Deployment und die Instandhaltung behindern.

Das soll aber nicht heißen, dass eine gut auf den Systembetrieb abgestimmte Softwarearchitektur nicht wünschenswert wäre. Im Gegenteil! Es bedeutet lediglich, dass die Kostengleichung mehr in Richtung Entwicklung, Deployment und Instandhaltung tendiert.

Abgesehen davon kommt der Architektur in Bezug auf den Betrieb des Systems aber noch eine andere Rolle zu: Eine gute Softwarearchitektur kommuniziert auch die Erfordernisse des Systems.

Man könnte in diesem Zusammenhang auch sagen, dass die Architektur eines Systems dessen Betriebsanforderungen unmittelbar erkennbar macht. Die Systemarchitektur sollte die Betriebsweise offenlegen. Sie sollte die Anwendungsfälle, die Features und das erforderliche Verhalten des Systems auf First-Class-Entitäten ausweiten, die für die Softwareentwickler sichtbare Orientierungspunkte darstellen. Das erleichtert das Verständnis des Systems und unterstützt somit auch die Entwicklung und die Instandhaltung in erheblichem Maße.

15.4 Instandhaltung

Von allen Aspekten eines Softwaresystems ist die Instandhaltung die kostenintensivste. Die unablässige Parade neuer Features und die unvermeidliche Spur von Fehlern und Korrekturen verschlingen enorme Mengen an personellen Ressourcen.

Die teuersten Faktoren der Instandhaltung sind das *Spelunking* (zu Deutsch etwa »die Höhlenerkundung«) und das Risiko. Der Begriff »Spelunking« umschreibt hier den Aufwand bzw. die Kosten für das Durchforsten der vorhandenen Software auf der Suche nach der besten Platzierung und der besten Strategie, um ein neues Feature zu ergänzen oder einen Fehler zu beheben. Bei der Durchführung solcher Modifikationen besteht immer auch die Gefahr, dass sich unbeabsichtigte Fehler einschleichen – was dann wiederum zu den Risikokosten beiträgt.

Eine sorgfältig durchdachte Softwarearchitektur dämpft diese Kosten ungemein, denn: Durch die Unterteilung des Systems in Komponenten und das Isolieren dieser Komponenten mittels stabiler Schnittstellen lassen sich die Wege für zukünftige Features bereiten und auch das Risiko eines unbeabsichtigten Systemschadens maßgeblich reduzieren.

15.5 Optionen offenhalten

Wie in Kapitel 2 beschrieben, zeichnet sich Software durch zwei Werte aus: ihr Verhalten und ihre Struktur. Letztere ist dabei von größerer Bedeutung, weil dieser Wert die Software »soft«, sprich »formbar« macht.

Die Software wurde erfunden, weil wir eine Möglichkeit brauchten, das Verhalten von Maschinen schnell und einfach zu modifizieren. Eine solche Flexibilität ist jedoch in hohem Maße von der Form des Systems abhängig, der Anordnung seiner Komponenten und der Art und Weise, wie diese Komponenten miteinander verbunden sind.

Um Software formbar zu halten, ist es notwendig, so viele Optionen wie möglich offenzulassen, und zwar so lange wie möglich. Doch um was für Optionen handelt es sich dabei eigentlich? *Um die Details, die eigentlich nicht von Bedeutung sind.*

Alle Softwaresysteme lassen sich in zwei große Bereiche unterteilen: die übergeordneten Richtlinien (engl. *Policy*) und die Details. Der Bereich der Richtlinien umfasst alle Geschäftsregeln und Prozeduren. Und sie sind es auch, die den wahren Wert des Systems ausmachen.

Bei den Details handelt es sich dagegen um die Dinge, die notwendig sind, damit menschliche User, andere Systeme und Programmierer mit den Richtlinien kommunizieren können, die aber trotzdem absolut keinen Einfluss auf deren Verhalten haben. Dazu gehören Ein-/Ausgabegeräte, Datenbanken, Websysteme, Server, Frameworks, Kommunikationsprotokolle und vieles mehr.

Das Ziel des Softwarearchitekten lautet, eine Form für das System zu erarbeiten, die die Richtlinien als das wichtigste Systemelement anerkennt und die zugehörigen Details *irrelevant* macht. Auf diese Weise lassen sich Entscheidungen über ebendiese Details *hinauszögern* und *aufschieben*.

Zum Beispiel:

- Es ist nicht nötig, im Frühstadium der Entwicklungsarbeit ein Datenbanksystem festzulegen, weil die übergeordneten Richtlinien nichts damit zu tun haben sollten, welche Art von Datenbank verwendet wird. Ob die Datenbank am Ende nun relational, verteilt oder hierarchisch strukturiert ist oder einfach nur aus Dateien besteht: Sofern der Softwarearchitekt Umsicht walten lässt, wird dies keinen Einfluss auf die übergeordneten Richtlinien haben.

- Es ist nicht nötig, im Frühstadium der Entwicklungsarbeit einen Webserver festzulegen, weil die übergeordneten Richtlinien keine Kenntnis davon haben sollten, dass sie über das Web bereitgestellt werden. Wenn sie nichts von HTML, AJAX, JSP, JSF oder irgendeinem anderen Sammelsurium aus der Buchstabensuppe der Webentwicklung wissen, dann müssen Sie bis zu einem späteren Zeitpunkt in der Projektentwicklung nicht entscheiden, welches Websystem eingesetzt wird – *im Grunde genommen müssen Sie nicht einmal entscheiden, ob eine Bereitstellung des Systems über das Web erfolgen wird.*

- Es ist nicht nötig, im Frühstadium der Entwicklungsarbeit die REST-Schnittstelle zu adaptieren, weil es für die übergeordneten Richtlinien unerheblich sein sollte, welche Schnittstelle zur Außenwelt verwendet werden wird. Ebenso wenig ist es notwendig, ein Microservices- oder ein SOA-Framework (**S**ervice-**O**riented **A**rchitecture, serviceorientierte Architektur) zu adaptieren – denn auch von diesen Dingen sollten die übergeordneten Richtlinien keine Kenntnis haben.

- Es ist nicht nötig, im Frühstadium der Entwicklungsarbeit ein *Dependency-Injection-Framework* zu adaptieren, weil die übergeordneten Richtlinien nicht davon tangiert sein sollten, wie Abhängigkeiten aufgelöst werden.

Ich denke, Sie verstehen, was damit gemeint ist. Wenn Sie die übergeordneten Richtlinien entwickeln, ohne sich auf die umgebenden Details festzulegen, können Sie diese Details betreffende Entscheidungen für eine lange Zeit verzögern und aufschieben. Und je länger Sie damit warten, diese Entscheidungen zu treffen, *desto mehr Informationen stehen Ihnen zur Verfügung, um sie in der richtigen Art und Weise fällen zu können.*

Dadurch haben Sie Gelegenheit, in verschiedenen Richtungen zu experimentieren. Wenn Sie bereits einen Teil der übergeordneten Richtlinien umgesetzt haben und sie der Datenbank gegenüber agnostisch sind, könnten Sie versuchen, sie mit verschiedenen Datenbanken zu verknüpfen, um ihre Anwendbarkeit und Performance zu testen. Das Gleiche gilt für Websysteme, Web-Frameworks oder die allgemeine Webtauglichkeit selbst.

Je länger Sie diese Optionen offenhalten, desto mehr können Sie experimentieren, umso mehr Dinge können Sie ausprobieren und umso mehr Informationen werden Ihnen zur Verfügung stehen, wenn Sie an dem Punkt ankommen, an dem die betreffenden Entscheidungen nicht mehr länger aufgeschoben werden können.

Doch was, wenn besagte Entscheidungen bereits von jemand anderem getroffen wurden? Was, wenn Ihr Unternehmen sich schon auf eine bestimmte Datenbank, einen bestimmten Webserver oder ein bestimmtes Framework festgelegt hat? *Ein guter Softwarearchitekt unterstellt grundsätzlich erst einmal, dass noch nichts endgültig entschieden ist*, und gestaltet das System so, als könnten ebendiese Entscheidungen noch so lange wie möglich aufgeschoben bzw. geändert werden.

> *Ein guter Softwarearchitekt maximiert die Anzahl der noch nicht gefällten Entscheidungen.*

15.6 Geräteunabhängigkeit

Um sich ein Beispiel für diese Denkweise vor Augen zu führen, lassen Sie uns in die 1960er-Jahre zurückreisen, als Computer noch in ihren Kinderschuhen steckten und die meisten Programmierer Mathematiker oder Ingenieure anderer Fachrichtungen waren (sowie ein Drittel mehr Frauen diesen Beruf ausübten).

Damals wurden eine Menge Fehler gemacht – aber natürlich war das den Beteiligten zu diesem Zeitpunkt nicht unmittelbar bewusst. Nur wie war das möglich?

Einer dieser Fehler war die direkte Bindung des Codes an die I/O-Geräte. Wenn etwas an einen Drucker ausgegeben werden musste, wurde Code geschrieben, der die I/O-Anweisungen für die Steuerung des betreffenden Geräts verwendete. Mit anderen Worten: Der Code war *geräteabhängig*.

Ich schrieb seinerzeit beispielsweise PDP-8-Programme für die Datenausgabe auf dem Fernschreiber (engl. *Teleprinter*) und verwendete dafür einen Satz Maschinenanweisungen, der etwa folgendermaßen aussah:

```
PRTCHR, 0
        TSF
        JMP .-1
        TLS
        JMP I PRTCHR
```

`PRTCHR` ist eine Subroutine, die ein Zeichen auf dem Fernschreiber ausgibt. Die Null am Anfang wurde zum Speichern der Rückgabeadresse verwendet. (Fragen Sie nicht ...) Die `TSF`-Anweisung sorgte dafür, dass die nachfolgende Anweisung übersprungen wurde, sofern der Fernschreiber ausgabebereit war. War er das hingegen nicht, dann wurde nach `TSF` einfach die Anweisung `JMP .-1` ausgeführt, die wiederum zur Anweisung `TSF` zurücksprang. War der Fernschreiber dann schließlich bereit, ging es stattdessen mit der `TLS`-Anweisung weiter, die das Zeichen im Register `A` an das Gerät weiterleitete. Und anschließend erfolgte mithilfe der Anweisung `JMP I PRTCHR` die Rückgabe an den Aufrufer.

Diese Strategie funktionierte zunächst wunderbar. Wenn wir Karten vom Kartenlesegerät einlesen mussten, nutzten wir Code, der den Kartenleser direkt ansprach. Wenn wir Karten lochen mussten, schrieben wir Code, der den Lochprozess unmittelbar steuerte. Die Programme funktionierten perfekt. Wie also hätten wir wissen können, dass diese Art von Code ein Fehler war?

Umfangreichere Lochkartenstapel sind allerdings schwer zu verwalten: Karten können verloren gehen, beschädigt, durcheinandergebracht oder vermischt werden, einzelne Karten können ausgelassen werden und es können sich unbeabsichtigt zusätzliche Karten einschleichen. Damit wurde die Datenintegrität zu einem erheblichen Problem.

Die Lösung waren Magnetbänder. Hiermit war es uns möglich, die Lochmuster bzw. die dadurch repräsentierten Daten auf Band zu übertragen. Wenn man ein Magnetband fallen ließ, wurden die Datensätze nicht durcheinandergebracht. Auch konnte man einen Datensatz bei der Bandübergabe nicht versehentlich verlieren oder einen leeren Datensatz einfügen. Dieses Verfahren war somit sehr viel sicherer. Zudem ließen sich die Bänder schneller auslesen und beschreiben und es war sehr einfach, Backupkopien davon anzufertigen.

Allerdings war unsere gesamte Software speziell zu dem Zweck geschrieben worden, die Kartenlesegeräte sowie die Lochung der Karten zu steuern. Also mussten die Programme für die Magnetbänder umgeschrieben werden – und das bedeutete einen Haufen Arbeit.

In den späten 1960er-Jahren hatten wir schließlich unsere Lektion gelernt – und erfanden das Konzept der *Geräteunabhängigkeit*. Die Betriebssysteme jener Zeit abstrahierten die I/O-Geräte in Softwarefunktionen, die einzelne Datensätze verarbeiteten, die wie Karten aussahen. Und die Programme riefen Betriebssystemdienste auf, die abstrakte Datensatzaufzeichnungsgeräte ansteuerten. Auf diese Weise waren die Operatoren nunmehr in der Lage, dem Betriebssystem mitzuteilen, ob die abstrakten Dienste mit Kartenlesern, Magnetbändern oder anderen Aufzeichnungsgeräten verbunden werden sollten.

So konnte ein und dasselbe Programm *ohne weitere Anpassungen* Lochkarten lesen und schreiben oder auch Bänder auslesen und beschreiben – das *Open-Closed-Prinzip* war geboren (wenn auch noch nicht als solches benannt).

15.7 Junk Mail

In den späten 1960er-Jahren war ich für ein Unternehmen tätig, das Werbeanschreiben für Firmenkunden druckte. Die Unternehmen übersandten uns Magnetbänder mit Einzeldatensätzen, die die Namen und Anschriften ihrer Kunden enthielten, und wir schrieben eine Software, die ansprechend personalisierte Serienbriefe mit diesen Daten ausgab.

Etwa in dieser Art:

Hallo, Herr Martin,

herzlichen Glückwunsch!

SIE wurden als einziger Bewohner der Witchwood Lane ausgewählt, von unserem fantastischen neuen und einmaligen Angebot für ... zu profitieren.

Zu diesem Zweck statteten uns unsere Auftraggeber mit riesigen Formbriefrollen aus, die mit Ausnahme der Namen und Anschriften der Adressaten bereits den gesamten Text sowie alle anderen gewünschten Briefbestandteile enthielten, die sie in gedruckter Form geliefert bekommen wollten. Und wir schrieben jeweils passende Programme dazu, die die Namen, Adressen und ggf. weitere Briefelemente von den bereitgestellten Magnetbändern extrahierten, an exakt den Positionen in den Formbriefen einfügten, an denen sie erscheinen sollten, und die Briefe schließlich druckten.

Jede Formbriefrolle, die zu Hunderten angeliefert wurden, wog etwa 500 Pfund und ergab Tausende solcher Werbebriefe. Unsere Aufgabe bestand also darin, jeden Brief zu individualisieren und einzeln auszudrucken.

Zunächst benutzten wir ein IBM-360-System, mit dem die Ausdrucke auf dem einzigen verfügbaren Zeilendrucker ausgegeben wurden. Damit konnten wir pro Arbeitsschicht ein paar Tausend Briefe herstellen. Leider war damit eine sehr teure Maschine über die Maßen langfristig ausgelastet: Zur damaligen Zeit betrug der Mietpreis für die IBM 360 Zehntausende Dollar im Monat.

Also wiesen wir das Betriebssystem an, die Briefe zunächst auf Magnetband statt auf den Zeilendrucker auszugeben – für unsere Programme war das einerlei, weil sie ja so geschrieben waren, dass sie die I/O-Abstraktionen des Betriebssystems verwendeten.

Die IBM 360 war in der Lage, binnen etwa 10 Minuten ein komplettes Magnetband vollzuschreiben – genug, um mehrere Formbriefrollen zu bedrucken. Anschließend wurden die Bänder aus dem Computerraum geholt und in Bandlaufwerke eingesetzt, die an Offlinedrucker angeschlossen waren. Wir hatten fünf davon, und allesamt waren an sieben Tagen die Woche 24 Stunden am Tag im Einsatz und druckten jede Woche Hunderttausende von Werbebriefen.

Die Geräteunabhängigkeit war für uns von unschätzbarem Wert! Wir konnten unsere Programme schreiben, ohne zu wissen oder uns damit zu befassen, welches Gerät letztendlich benutzt werden würde. Außerdem konnten wir diese Programme mit einem lokal an den Computer angeschlossenen Zeilendrucker testen. Und schließlich konnten wir das Betriebssystem einfach anweisen, alles auf Magnetband auszugeben, und Hunderttausende von Formularen anfertigen.

Wir hatten unseren Programmen eine Form gegeben – und zwar eine, die die Richtlinien und die Details voneinander loslöste: Die Richtlinien formatierten die Namens- und Anschriftendatensätze, und die Details entsprachen dem Gerät. Mit anderen Worten: Wir schoben die Entscheidung darüber, welches Gerät wir schlussendlich nutzen würden, schlicht und ergreifend auf.

15.8 Physische Adressierung

In den frühen 1970er-Jahren arbeitete ich an einem großen Buchhaltungssystem für eine regionale Fernfahrergewerkschaft. Wir hatten ein 25-MB-Laufwerk, auf dem Datensätze für `Agents` (Spediteure), `Employers` (Arbeitgeber) und `Members` (Mitglieder) gespeichert waren. Die verschiedenen Datensätze waren unterschiedlich groß, also formatierten wir die ersten paar Zylinder des Laufwerks so, dass jeder Sektor nur der Größe eines `Agent`-Datensatzes entsprach. Die nächsten paar Zylinder

waren passend für die `Employer`-Datensätze formatiert. Und die Formatierung der letzten paar Zylinder war schließlich auf die `Member`-Datensätze angepasst.

Unsere Software schrieben wir so, dass sie sich an der detaillierten Struktur des Datenträgers orientierte. Sie »wusste« also, dass das Laufwerk 200 Zylinder und 10 Köpfe besaß und dass jeder Zylinder einige Dutzend Sektoren pro Kopf hatte. Und es wusste, welche Zylinder jeweils für die `Agents`, die `Employers` und die `Members` zuständig waren. All diese Informationen waren *hartcodiert*, also fest programmiert.

Auf der Festplatte hatten wir einen Index hinterlegt, der es uns ermöglichte, jeden einzelnen `Agent`, `Employer` und `Member` aufzurufen. Dieser Index befand sich wiederum auf einem eigenen, speziell formatierten Zylindersatz auf dem Datenträger. Der `Agent`-Index setzte sich aus den Datensätzen zusammen, die die ID eines Spediteurs sowie die Zylindernummer, die Kopfnummer und die Sektornummer des betreffenden `Agent`-Datensatzes enthielten. Die Indizes für die `Employers` und `Members` waren in vergleichbarer Weise aufgebaut. Darüber hinaus wurden die `Members` außerdem in einer doppelt verknüpften Liste auf der Festplatte geführt: Jeder `Member`-Datensatz enthielt auch die Zylinder-, Kopf- und Sektornummer des jeweils nächsten und des jeweils vorausgehenden `Member`-Datensatzes.

Doch was wäre passiert, wenn wir das Ganze irgendwann auf eine neue Festplatte hätten übertragen müssen – eine mit mehr Köpfen oder mit mehr Zylindern oder Sektoren pro Zylinder? Dann hätten wir ein spezielles Programm schreiben müssen, das die alten Daten von dem alten Datenträger auslesen und sie anschließend auf das neue Laufwerk schreiben und dabei alle Zylinder-, Kopf- und Sektornummern übersetzen würde. Und zusätzlich hätten wir auch sämtliche hartcodierten Passagen in unserem Code anpassen müssen – wobei die sich so ziemlich *überall* befanden, denn sämtliche Geschäftsregeln hatten genaue Kenntnis von der Zylinder/Kopf/Sektor-Einteilung!

Eines Tages stieß dann ein erfahrenerer Programmierer zu unserer Truppe. Als er sah, wie wir vorgegangen waren, wich ihm das Blut aus dem Gesicht und er starrte uns entsetzt an, als wären wir Außerirdische. Behutsam empfahl er uns, unser Adressierungsschema auf die Verwendung von relativen Adressen umzustellen.

Unser kluger Kollege schlug vor, dass wir die Festplatte als ein riesiges lineares Array von Sektoren betrachten sollten, von denen jeder durch einen sequenziellen Integer adressierbar wäre. So konnten wir eine kleine Konvertierungsroutine schreiben, die Kenntnis von der physischen Struktur des Datenträgers hatte und die relative Adresse »on-the-fly« in eine Zylinder-/Kopf-/Sektornummer übersetzen konnte.

Zu unserem Glück setzten wir diesen Vorschlag daraufhin auch in die Tat um. Wir änderten die übergeordneten Richtlinien des Systems dahin gehend, dass sie gegenüber der physischen Struktur des Datenträgers agnostisch waren – dadurch war es uns möglich, auch die Entscheidung über die Datenträgerstruktur von der Anwendung

abzukoppeln.

15.9 Fazit

Die beiden Beispiele in diesem Kapitel demonstrieren im kleinen Rahmen ein Prinzip, das Softwarearchitekten im großen Rahmen umsetzen. Gute Softwarearchitekten trennen die Details sorgfältig von den Richtlinien und entkoppeln sie dabei so gründlich, dass Letztere überhaupt keine Kenntnis von den Details haben und auch in keiner Weise von ihnen abhängig sind. Sie gestalten die Richtlinien in der Art, dass Entscheidungen über die Details so lange wie möglich hinausgezögert und aufgeschoben werden können.

Kapitel 16
Unabhängigkeit

Wie wir inzwischen bereits festgestellt haben, muss eine gute Softwarearchitektur folgende Aspekte stützen:

- die Use Cases und den Systembetrieb
- die Instandhaltung des Systems
- die Entwicklung des Systems
- das Deployment des Systems

16.1 Use Cases

Der erste Punkt dieser Liste – die *Use Cases* – bedeutet, dass die Architektur dem übergeordneten Zweck des Systems dienlich sein muss. Handelt es sich beispielsweise um eine Warenkorbanwendung, dann muss die Softwarearchitektur auch die entsprechenden Use Cases unterstützen. Unterm Strich ist dies für den Softwarearchitekten die oberste Direktive – und die oberste Priorität für die Architektur selbst: Sie muss die anfallenden Use Cases unterstützen.

Wie jedoch schon an anderer Stelle erwähnt, wirkt sich die Softwarearchitektur nicht allzu sehr auf das Verhalten des Systems aus. Es gibt nur wenige Verhaltensoptionen, die die Architektur offenlassen kann. Andererseits ist Einfluss aber auch nicht alles. Das Wichtigste, was eine gute Softwarearchitektur in Bezug auf das Verhalten bewirken kann, ist, ebendieses Verhalten deutlich zu machen und klar hervorzuheben, sodass der Zweck des Systems auf der Architekturebene erkennbar wird.

Eine Warenkorbanwendung mit einer guten Softwarearchitektur wird auch *aussehen* wie eine Warenkorbanwendung. Die Use Cases dieses Systems werden innerhalb seiner Struktur klar und deutlich zutage treten. Die Softwareentwickler werden nicht erst auf die Suche nach Verhaltensweisen gehen müssen, weil sie als Elemente erster Klasse auf der obersten Ebene des Systems sichtbar sein werden. Bei diesen Elementen wird es sich um Klassen oder Funktionen oder Module handeln, die an prominenter Stelle innerhalb der Architektur positioniert sind und aus deren Bezeichnungen ihre jeweilige Funktionalität eindeutig hervorgeht.

In Kapitel 21, »Die schreiende Softwarearchitektur«, wird dieser Aspekt noch sehr viel deutlicher werden.

16.2 Betrieb

Der Softwarearchitektur kommt in Bezug auf die Unterstützung des Systembetriebs eine eher substanzielle als eine kosmetische Rolle zu. Wenn das System 100.000 Kunden pro Sekunde verarbeiten muss, muss auch seine Architektur diese Art von Durchsatz und Antwortzeiten für jeden Use Case unterstützen, der dies erfordert. Wenn das System große Datenwürfel (engl. *Data Cubes*) binnen Millisekunden abfragen muss, dann muss die Softwarearchitektur so strukturiert sein, dass diese Art von Operation möglich ist.

Für manche Systeme bedeutet das, dass ihre Verarbeitungselemente des Systems in ein Array von kleineren Services angeordnet werden müssen, die parallel auf mehreren verschiedenen Servern ausgeführt werden können. Für andere Systeme bedeutet es dagegen eine Fülle von kleinen leichtgewichtigen Threads, die sich den Adressbereich

eines einzigen Prozesses innerhalb ein und desselben Prozessors teilen. Wieder andere Systeme werden lediglich ein paar Prozesse benötigen, die in isolierten Adressbereichen laufen. Und einige Systeme können sogar als einfache monolithische Programme überdauern, die in einem einzigen Prozess ausgeführt werden.

So merkwürdig es auch erscheinen mag: Diese Entscheidung ist eine der Optionen, die sich ein guter Softwarearchitekt offenhält. Ein System, das als Monolith geschrieben wird und von dieser monolithischen Struktur abhängig ist, kann nicht so einfach bei Bedarf auf multiple Prozesse, multiple Threads oder Microservices upgegradet werden. Im Vergleich dazu wird sich eine Softwarearchitektur, die eine saubere Isolation ihrer Komponenten einhält und keine Annahmen hinsichtlich der Kommunikationsmethoden zwischen diesen Komponenten unterstellt, sehr viel einfacher durch das Spektrum von Threads, Prozessen und Services führen lassen, wenn sich die operativen Anforderungen des Systems im Laufe der Zeit verändern.

16.3 Entwicklung

Die Softwarearchitektur spielt bei der Unterstützung der Entwicklungsumgebung eine maßgebliche Rolle. Hierbei tritt das Gesetz von Conway auf den Plan, das besagt:

> *Organisationen, die Systeme entwerfen, sind auf Entwürfe festgelegt, die die Kommunikationsstrukturen dieser Organisationen abbilden.*

Ein System, das von einer Organisation mit vielen Entwicklungsteams und für viele verschiedene Anforderungen entwickelt wird, muss eine Architektur aufweisen, die unabhängige Aktionen dieser Teams erleichtert, sodass sie sich während der Entwicklungsarbeit nicht gegenseitig behindern. Dies wird durch die saubere Partitionierung des Systems in sorgsam isolierte, unabhängig voneinander zu entwickelnde Komponenten erreicht. So ist eine individuelle Zuweisung dieser Komponenten an die Teams möglich, die dann jeweils autark daran arbeiten können.

16.4 Deployment

Die Softwarearchitektur spielt ebenso eine bedeutende Rolle, wenn es darum geht zu bestimmen, wie einfach sich das System deployen lässt. Das Ziel lautet hier »sofortiges Deployment«. Eine gute Softwarearchitektur basiert nicht auf Dutzenden von kleinen Konfigurationsskripten und verzweigten Eigenschaftsdateien. Sie erfordert keine manuelle Erstellung von Verzeichnissen oder Dateien, die akkurat angelegt und angeordnet werden müssen. Eine gute Softwarearchitektur trägt vielmehr dazu bei, dass das System unmittelbar nach der Fertigstellung des Builds verzögerungsfrei

deployt werden kann.

Auch dies lässt sich wieder durch das saubere Partitionieren und Isolieren der Systemkomponenten erreichen, einschließlich derjenigen Master-Komponenten, die das gesamte System zusammenhalten und sicherstellen, dass jede einzelne Komponente sauber gestartet, integriert und überwacht wird.

16.5 Optionen offenhalten

Eine gute Softwarearchitektur balanciert all diese Anforderungen in einer Komponentenstruktur so aus, dass sie allen gemeinsam gerecht wird. Klingt doch einfach, oder? Nun ja, zumindest habe ich hier leicht reden ...

In der Realität ist diese Balance allerdings nur recht schwer zu erzielen. Das Problem ist, dass die Softwareentwickler meist weder alle möglichen Use Cases kennen, noch genaue Kenntnisse von den operativen Einschränkungen, der Teamstruktur oder den Deployment-Anforderungen besitzen. Und was noch schlimmer ist: Selbst wenn sie all diese Informationen hätten, werden sich diese Faktoren im Verlauf des Lebenszyklus eines Systems unweigerlich verändern. Kurz: Die Zielsetzungen, die die Entwicklungsteams erreichen müssen, sind unkalkulierbar und unbeständig. Willkommen in der realen Welt!

Doch noch ist nicht alles verloren: Einige Prinzipien der Softwarearchitektur lassen sich zu relativ geringen Kosten implementieren und können dabei helfen, die diversen Anforderungen auszubalancieren, selbst wenn Sie keinen klaren Überblick über die zu erreichenden Ziele haben. Diese Prinzipien unterstützen die Softwareentwickler dabei, ihre Systeme in gut isolierte Komponenten zu unterteilen, die es ihnen ermöglichen, sich so lange wie möglich so viele Optionen wie möglich offenzuhalten.

Eine gute Softwarearchitektur gewährleistet, dass das System leicht zu modifizieren ist, und zwar in jeder Form, die nötig ist, indem Optionen offengelassen werden.

16.6 Layer entkoppeln

Denken Sie noch einmal an die Use Cases. Der Softwarearchitekt will die Systemstruktur auf alle erforderlichen Use Cases ausrichten, ohne jedoch genau zu wissen, welche das genau sind. Was er jedoch weiß, ist, welchen grundsätzlichen Zweck das System erfüllen soll – sei es, dass es sich um ein Warenkorbsystem, ein Stücklistensystem oder ein Auftragsabwicklungssystem handelt. Und so kann der Softwarearchitekt das *Single-Responsibility-Prinzip* und das *Common-Closure-Prinzip*

anwenden, um diejenigen Elemente, die sich aus verschiedenen Gründen ändern, zu separieren und jene Elemente, die sich aus denselben Gründen ändern, zusammenzufassen – entsprechend dem Kontext des Systemzwecks.

Was aber bedeuten nun Änderungen aus verschiedenen Gründen? In manchen Fällen ist das offensichtlich. Benutzerschnittstellen werden beispielsweise aus Gründen modifiziert, die nichts mit den Geschäftsregeln zu tun haben. Bei Use Cases spielt beides eine Rolle. Hier würde ein guter Softwarearchitekt die UI-Bestandteile eines Use Case von den auf die Geschäftsregeln bezogenen Bestandteilen in einer Art und Weise separieren, dass sie unabhängig voneinander angepasst werden können, wobei die Use Cases klar und deutlich erkennbar bleiben.

Die Geschäftsregeln selbst können sehr eng mit der Anwendung verknüpft oder auch eher allgemeinerer Art sein. Zum Beispiel ist die Validierung der Eingabefelder sehr eng an die Anwendung selbst gekoppelt. Im Gegensatz dazu stellen die Zinsberechnung für ein Konto oder die Zählung des Inventars Geschäftsregeln dar, die in einem weitaus engeren Bezug zu der Domäne stehen. Diese beiden verschiedenartigen Regeln werden sich im Laufe der Zeit unterschiedlich oft und aus unterschiedlichen Gründen ändern – und deshalb sollten sie getrennt werden, damit sie unabhängig voneinander angepasst werden können.

Die Datenbank, die Datenbankabfragesprache (engl. *Query Language*) und auch das Schema sind hingegen technische Details, die nichts mit den Geschäftsregeln oder der Benutzerschnittstelle zu tun haben. Sie werden sich zu mehreren Zeitpunkten und aus Gründen ändern, die nicht von anderen Aspekten des Systems abhängig sind. Und folgerichtig sollte die Softwarearchitektur sie daher ebenfalls vom übrigen System trennen, damit sie unabhängig davon modifiziert werden können.

Daraus ergibt sich dann eine Aufteilung des Systems in entkoppelte horizontale Layer: die Benutzerschnittstelle, anwendungsspezifische Geschäftsregeln, anwendungsunabhängige Geschäftsregeln und die Datenbank, um nur einige zu nennen.

16.7 Use Cases entkoppeln

Und was ändert sich sonst noch aus unterschiedlichen Gründen? Die Use Cases selbst! Der Anwendungsfall des Einpflegens einer Bestellung in ein Bestellsystem wird sich mit ziemlicher Sicherheit unterschiedlich oft und aus anderen Gründen verändern als ein Use Case zum Löschen einer Bestellung in einem Bestellsystem. Use Cases stellen eine sehr natürliche Grundlage für die Aufteilung eines Systems dar.

Gleichzeitig sind sie aber auch schmale vertikale Teilstücke, die sich sozusagen durch

die horizontalen Layer des Systems hindurchfräsen. Bei jedem Use Case kommen einige Benutzer, einige anwendungsspezifische Geschäftsregeln, einige anwendungsunabhängige Geschäftsregeln und einige Datenbankfunktionen zum Einsatz. Insofern wird das System ebenso, wie es in horizontale Layer unterteilt wird, auch in dünne vertikale Use Cases aufgeteilt, die durch diese Layer vordringen.

Um eine derartige Entkopplung zu erreichen, wird beispielsweise die Benutzerschnittstelle des Use Case »Bestellung hinzufügen« von der Benutzerschnittstelle des Use Case »Bestellung löschen« separiert. Das Gleiche geschieht auch mit den Geschäftsregeln und der Datenbank. Diese Use Cases werden abwärts verlaufend durch den vertikalen Aufbau des Systems getrennt gehalten.

Darin ist ein Muster zu erkennen: Wenn Sie die Systemelemente, die sich aus unterschiedlichen Gründen ändern, entkoppeln, dann können Sie damit fortfahren, neue Use Cases zu ergänzen, ohne die bereits bestehenden zu beeinträchtigen. Und wenn Sie darüber hinaus auch die für diese Use Cases relevanten Benutzerschnittstellen und die Datenbank gruppieren, sodass sich jeder Use Case eines anderen Aspekts der Benutzerschnittstelle und der Datenbank bedient, dann wird das Hinzufügen neuer Use Cases aller Wahrscheinlichkeit nach keine Auswirkungen auf die bereits vorhandenen haben.

16.8 Entkopplungsmodi

Überlegen Sie nun einmal, was all diese Entkopplungen für den zweiten Teil von Punkt 1 unserer eingangs erwähnten Liste zur Folge haben: die Instandhaltung des Systems. Wenn die verschiedenen Aspekte der Use Cases separiert werden, dann sind diejenigen, die mit einem hohen Durchsatz laufen, wahrscheinlich bereits von denjenigen, die mit einem niedrigen Durchsatz laufen, entkoppelt. Und wenn die Benutzerschnittstellen und die Datenbank von den Geschäftsregeln getrennt wurden, dann können sie auf verschiedenen Servern ausgeführt werden. Was eine höhere Bandbreite braucht, kann somit auf viele Server repliziert werden.

Kurz: Die Entkopplung, die wir hier im Sinne der Use Cases vorgenommen haben, ist auch im Bereich des Systembetriebs hilfreich. Um jedoch von dem operativen Vorteil profitieren zu können, muss die Entkopplung im passenden Modus erfolgen. Damit sie auf separaten Servern genutzt werden können, dürfen die separierten Komponenten nicht davon abhängig sein, dass sie sich gemeinsam in demselben Adressbereich eines Prozessors befinden. Stattdessen müssen sie unabhängige Services sein, die über irgendeine Art von Netzwerk kommunizieren.

Viele Softwarearchitekten nennen solche Komponenten in vager Anlehnung an die Zeilenzählung »Services« oder »Microservices«. Tatsächlich wird eine auf Services

aufbauende Softwarearchitektur oft als *serviceorientierte Architektur* (*SOA*, **S**ervice-**O**riented **A**rchitecture) bezeichnet.

Sollte diese Nomenklatur in irgendeiner Weise alarmierend für Sie klingen, dann kann ich Sie beruhigen: Ich werde Ihnen nicht erzählen, dass die SOA die bestmögliche Softwarearchitektur ist oder dass Microservices das Modell der Zukunft darstellen. An dieser Stelle soll lediglich deutlich werden, dass wir unsere Komponenten manchmal bis hin zur Serviceebene trennen müssen.

Denken Sie immer daran: Eine gute Softwarearchitektur lässt Optionen offen. *Und der Entkopplungsmodus ist eine dieser Optionen.*

Bevor wir aber nun weiter in dieses Thema vordringen, sollen zunächst auch noch die letzten beiden Punkte der anfangs erwähnten Liste betrachtet werden.

16.9 Unabhängige Entwickelbarkeit

Bei dem dritten Aspekt geht es um die Entwicklungsarbeit. Wenn die Komponenten stark entkoppelt sind, mildert dies die gegenseitige Beeinträchtigung zwischen den Teams ohne Frage ab. Solange die Geschäftsregeln keine Kenntnis von der Benutzerschnittstelle haben, kann die Arbeit eines Teams, das sich auf Letztere konzentriert, keine großen Auswirkungen auf die Arbeit eines Teams haben, das hauptsächlich mit den Geschäftsregeln befasst ist. Und wenn die Use Cases selbst voneinander entkoppelt sind, dann ist es eher unwahrscheinlich, dass sich ein Team, das sich auf den Anwendungsfall `addOrder` konzentriert, und ein anderes Team, das seinerseits wiederum an dem Anwendungsfall `deleteOrder` arbeitet, ins Gehege kommen.

Solange die Layer und Use Cases entkoppelt sind, unterstützt die Systemarchitektur die Teamorganisation, unabhängig davon, ob sie in Featureteams, Komponententeams, Layerteams oder irgendeine andere Teamkonstellation eingeteilt sind.

16.10 Unabhängige Deploybarkeit

Das Entkoppeln der Use Cases und Layer gewährleistet außerdem eine hohe Flexibilität in Bezug auf das Deployment. Wenn die Entkopplung gut durchgeführt wird, sollte es sogar durchaus möglich sein, Layer und Use Cases im laufenden Systembetrieb auszutauschen. Die Ergänzung eines neuen Use Case könnte sich dann so simpel darstellen, dass einfach ein paar neue `.jar`-Dateien oder Services zum System hinzugefügt werden, während die übrigen Elemente komplett unangetastet

bleiben.

16.11 Duplizierung

Softwarearchitekten tappen oft in die Falle – eine Falle, die auf ihrer Angst vor Duplizierungen fußt.

Grundsätzlich sind Duplizierungen im Softwarebereich immer eine schlechte Sache. Programmierer mögen keinen duplizierten Code. Und wenn es dann doch wirklich einmal passiert ist, dann ist es für uns, als professionelle Softwareentwickler, Ehrensache, solche Doppelungen zu beheben oder zumindest zu minimieren.

Allerdings gibt es verschiedene Arten von Duplizierungen. Zum einen wäre da die *echte Duplizierung* zu nennen, was bedeutet, dass jede Modifikation an einer Instanz dieselbe Änderung an jedem Duplikat dieser Instanz erforderlich macht. Und zum anderen kommen auch *falsche* oder *versehentliche Duplizierungen* vor. Wenn sich zwei offenbar duplizierte Codeabschnitte in verschiedene Richtungen entwickeln – sei es nun, dass sie unterschiedlich oft angepasst werden oder wegen unterschiedlicher Anforderungen –, *dann handelt es sich nicht um echte Duplizierungen*. Und wenn Sie sich die beiden Abschnitte nach ein paar Jahren noch mal anschauen, dann werden Sie feststellen, dass sie sich sehr voneinander unterscheiden.

Stellen Sie sich nun einmal zwei Use Cases mit strukturell sehr ähnlichen Bildschirmausgaben vor. Die Softwarearchitekten wären sicherlich stark versucht, den Code für diese Struktur gemeinsam zu nutzen. Aber sollten sie das auch tun? Wäre das eine echte Duplizierung? Oder eine versehentliche?

Höchstwahrscheinlich wäre es eher ein Versehen. Vermutlich würden die beiden Bildschirmausgaben mit der Zeit immer weiter voneinander abweichen und schließlich sehr unterschiedlich aussehen. Deshalb muss darauf geachtet werden, dass sie nicht zusammengefasst werden – andernfalls würde eine spätere Separierung zur Herausforderung werden.

Sobald Sie die Use Cases vertikal separieren, werden Sie mit diesem Problem und somit auch der Versuchung konfrontiert werden, diese Anwendungsfälle zu koppeln, weil sie eine strukturell ähnliche Bildschirmausgabe, ähnliche Algorithmen oder ähnliche Datenbankabfragen und/oder -schemata aufweisen. Seien Sie damit jedoch vorsichtig! Widerstehen Sie der Versuchung, die »Sünde« der reflexartigen Beseitigung von Duplizierungen zu begehen – und vergewissern Sie sich stattdessen, dass die Duplizierung echt ist.

Das Gleiche gilt auch für die horizontale Trennung der Layer. Hier werden Sie möglicherweise feststellen, dass die Datenstruktur eines bestimmten Datensatzes

innerhalb der Datenbank einer bestimmten Bildschirmausgabe sehr ähnlich ist. Dementsprechend könnten Sie in diesem Fall versucht sein, den Datensatz einfach an die Benutzerschnittstelle weiterzuleiten, statt ein gleichartiges *View Model* (Ansichtsmodell) zu erstellen und die Elemente dort hineinzukopieren. Auch hier ist also Vorsicht geboten: Eine solche Duplikation erfolgt in aller Regel versehentlich. Das Erstellen eines separaten View Models ist keine allzu aufwendige Angelegenheit – und darüber hinaus ist es sehr nützlich, um die saubere Entkopplung der Layer zu gewährleisten.

16.12 Entkopplungsmodi (zum Zweiten)

Nun aber zurück zu den Modi. Es gibt viele Möglichkeiten, die Layer und Use Cases zu entkoppeln: auf der Quellcode-Ebene, auf der Ebene des Binärcodes (Deployment) und auf der Ausführungsebene (Service).

- **Quellcode-Ebene.** Die Abhängigkeiten zwischen den Quellcode-Modulen können in der Form gesteuert werden, dass Modifikationen an einem Modul keine weiteren Anpassungen oder eine Neukompilierung der anderen Module (z.B. Ruby Gems) erzwingen.

  In diesem Entkopplungsmodus werden alle Komponenten in demselben Adressbereich ausgeführt und kommunizieren über einfache Funktionsaufrufe miteinander. Es wird nur eine einzige ausführbare Datei in den Computerspeicher geladen. Dieser Aufbau wird oft als *monolithische Struktur* bezeichnet.

- **Deployment-Ebene.** Die Abhängigkeiten zwischen deploybaren Einheiten wie etwa `.jar`-Dateien, DLLs oder Shared Libraries (dynamische Bibliotheken) lassen sich so steuern, dass Modifikationen am Quellcode eines Moduls keine Anpassungen oder ein erneutes Deployment anderer Module erzwingen.

  Viele der Komponenten können weiterhin in demselben Adressbereich existieren und durch Funktionsaufrufe miteinander kommunizieren. Andere Komponenten können in anderen Prozessen desselben Prozessors existieren und die Kommunikation interprozedural, über Sockets oder mittels gemeinsam genutztem Speicher bewerkstelligen. Der springende Punkt ist hier, dass die entkoppelten Komponenten in unabhängig voneinander deploybare Einheiten wie `.jar`-Dateien, `.gem`-Dateien oder DLLs partitioniert werden können.

- **Service-Ebene.** Man kann die Abhängigkeiten bis auf die Ebene der Datenstrukturen reduzieren und die Kommunikation ausschließlich über Netzwerkpakete abwickeln, sodass Modifikationen am Quell- oder Binärcode anderer Elemente (z.B. Services oder Microservices) keinerlei Einfluss auf die

einzelnen Ausführungseinheiten haben.

16.12.1 Welcher Modus ist am besten geeignet?

Die Antwort auf diese Frage lautet, dass es schwierig ist herauszufinden, welcher Modus in den Frühphasen eines Projekts der beste ist. Im Grunde genommen verhält es sich sogar so, dass sich der optimale Modus im Laufe des Projekts verändern kann.

Beispielsweise ist es nicht schwierig, sich vorzustellen, dass ein System, das zu einem bestimmten Zeitpunkt wunderbar auf einem einzelnen Server läuft, zu einem späteren Zeitpunkt in der Zukunft so umfangreich wird, dass einige seiner Komponenten auf separate Server ausgelagert werden müssen. Solange dieses System auf einem einzelnen Server läuft, mag die Entkopplung auf Quellcode-Ebene noch hinreichend sein – später könnte hingegen durchaus auch eine Entkopplung in deploybare Einheiten oder Services notwendig werden.

Eine Lösung (die zurzeit sehr beliebt zu sein scheint) besteht darin, das System einfach standardmäßig auf der Service-Ebene zu entkoppeln. Allerdings stellt sich hierbei das Problem, dass dieser Ansatz kostenintensiv ist und eine grobgranulare Entkopplung begünstigt. Egal, wie »mikro« die Microservices auch werden, die Entkopplung wird höchstwahrscheinlich dennoch nicht feingranular genug sein.

Ein weiteres Problem mit der Entkopplung auf der Service-Ebene ist zudem seine Kostenintensität sowohl hinsichtlich der Entwicklungszeit als auch der Systemressourcen. Der Umgang mit Serviceeinschränkungen, wo keine benötigt werden, ist eine Verschwendung von Arbeitszeit, Speicher und Zyklen. Und ja, ich weiß, dass die beiden letztgenannten Faktoren preiswert sind – der erste jedoch keineswegs.

Die von mir persönlich bevorzugte Variante ist, die Entkopplung bis zu dem Punkt voranzutreiben, an dem bei Bedarf ein Service gebildet werden könnte, die Komponenten aber dann so lange wie möglich in demselben Adressbereich zu belassen. So bleibt die Option für einen Service offen.

Bei dieser Vorgehensweise werden zunächst die Komponenten auf der Quellcode-Ebene separiert. Das kann für die gesamte Lebenszeit des Projekts ausreichen. Sollten jedoch irgendwelche Deployment- oder Entwicklungsprobleme auftreten, kann es – zumindest für eine Weile – auch reichen, einen Teil der Entkopplung lediglich auf der Deployment-Ebene vorzunehmen.

Wenn die Entwicklungs-, Deployment- und operativen Probleme zunehmen, entscheide ich sorgfältig, welche deploybaren Einheiten in Services gewandelt werden sollten, und verlagere das System dann schrittweise in diese Richtung.

Mit der Zeit können sich die operativen Anforderungen an das System auch reduzieren: Was zuvor noch auf der Service-Ebene entkoppelt werden musste, kann dann möglicherweise nur noch eine Entkopplung auf der Deployment- oder Quellcode-Ebene erfordern.

Eine gute Softwarearchitektur ermöglicht es, dass ein System ursprünglich als monolithische Struktur startet, die in einer einzigen Datei deployt wird, dann jedoch zu einem Satz von unabhängig voneinander deploybaren Einheiten und schließlich zu unabhängigen Services und/oder Microservices anwächst. Wenn sich die Sachlage später ändert, sollte es außerdem möglich sein, diese Progression wieder umzukehren und zur monolithischen Struktur zurückzukehren.

Außerdem schützt eine gute Softwarearchitektur den Großteil des Quellcodes vor den erwähnten Änderungen. Der Entkopplungsmodus bleibt als Option bestehen, sodass große Deployments einen Modus und kleine Deployments einen anderen Modus nutzen können.

16.13 Fazit

Zugegeben: Das Ganze ist etwas kompliziert. Und ich behaupte auch nicht, dass die Änderung der Entkopplungsmodi eine triviale Konfigurationsoption sein sollte (obwohl das manchmal tatsächlich angebracht ist). Vielmehr will ich mit den Ausführungen in diesem Kapitel deutlich machen, dass der Entkopplungsmodus eines Systems eins der Dinge ist, die sich mit der Zeit verändern können – und ein guter Softwarearchitekt sieht solche Änderungen voraus und erleichtert sie entsprechend.

Kapitel 17

Grenzen: Linien ziehen

Die Softwarearchitektur ist die Kunst, *Grenzen* zu setzen. Sie separieren die Softwareelemente voneinander und halten diejenigen auf der einen Seite davon ab, Kenntnis von denen auf der anderen Seite zu erhalten. Einige solcher Grenzlinien werden bereits sehr frühzeitig in einem Projekt gezogen – noch bevor überhaupt Code geschrieben wurde. Andere kommen erst sehr viel später dazu. Grenzlinien, die frühzeitig gezogen werden, dienen dazu, Entscheidungen so lange wie möglich hinauszuzögern und zu verhindern, dass sie die Kerngeschäftslogik beeinträchtigen.

Rufen Sie sich in Erinnerung, dass das Ziel eines Softwarearchitekten darin besteht, die für den Build und die Instandhaltung des benötigten Systems erforderlichen personellen Ressourcen zu minimieren. Was ist es, das diese Art von Antriebskraft schwächt? Die Kopplung – und insbesondere die Sorte Kopplung, die zu voreiligen bzw. verfrühten Entscheidungen führt.

Aber was wären das für Entscheidungen? Solche, die nichts mit den geschäftlichen Anforderungen – den Use Cases – an das System zu tun haben. Dazu gehören Entscheidungen über Frameworks, Datenbanken, Webserver, Utility Libraries, die Dependency Injection und so weiter. Eine gute Systemarchitektur entsteht dann, wenn

Entscheidungen wie diese ergänzend getroffen und zurückgestellt werden können. Eine gute Systemarchitektur ist nicht von derartigen Entscheidungen abhängig. Eine gute Systemarchitektur ermöglicht es, dass diese Entscheidungen im letzten Moment und ohne einschneidende Auswirkungen getroffen werden können.

17.1 Ein paar traurige Geschichten

An dieser Stelle will ich Ihnen einmal die traurige Geschichte des Unternehmens P erzählen, die Ihnen im Hinblick auf das Fällen verfrühter Entscheidungen als Warnung dienen soll. In den 1980er-Jahren schrieben die Gründer von P eine simple monolithische Desktopanwendung. Wie sich herausstellte, war das Produkt ein durchschlagender Erfolg und entwickelte sich bis zu den 1990er-Jahren zu einer sehr beliebten GUI-Desktopanwendung.

Doch dann, in den späten 1990er-Jahren, gewann das Internet zunehmend an Bedeutung. Urplötzlich musste jeder eine Weblösung haben, und P machte da keine Ausnahme. Die Kunden des Unternehmens verlangten nach einer internettauglichen Version des Produkts. Und um dieser Nachfrage gerecht zu werden, stellte P eine Reihe von erstklassigen jungen Java-Programmierern ein und schickte sich an, seine Anwendung für das Web aufzubereiten.

Diese Java-Jungs hatten eine Menge Flausen von Serverfarmen im Kopf und adaptierten daher eine vollgepackte dreischichtige »Architektur«[1], die sie mithilfe besagter Serverfarmen verteilen konnten. Ihrer Vorstellung nach sollte es Server für das GUI, für die Middleware und für die Datenbank geben. Na klar!

Die Programmierer entschieden bereits sehr frühzeitig, dass alle Domänenobjekte drei Instanzen haben sollten: eine in der GUI-Schicht, eine in der Middleware-Schicht und eine in der Datenbank-Schicht. Da diese Instanziierungen auf verschiedenen Maschinen erfolgten, wurde ein umfangreiches interprozedurales und schichtübergreifendes Kommunikationssystem aufgebaut. Methodenaufrufe zwischen den einzelnen Schichten wurden in Objekte konvertiert, serialisiert und dem *Marshalling* (zu Deutsch etwa »Umwandlung«) für die drahtgebundene Übermittlung unterzogen.

Nun stellen Sie sich einmal vor, welcher Aufwand betrieben werden muss, um ein einfaches Feature, etwa ein neues Datenfeld, in einen vorhandenen Datensatz zu implementieren: Das betreffende Feld müsste in die Klassen aller drei Schichten ergänzt werden sowie in einige der schichtübergreifenden Meldungen. Da die Datenübermittlung in beide Richtungen stattfinden würde, müssten vier Nachrichtenprotokolle entworfen werden. Jedes dieser Protokolle hätte eine sendende und eine empfangende Seite, sodass acht Protokoll-Handler erforderlich wären.

Außerdem müssten drei ausführbare Dateien mit jeweils drei aktualisierten Geschäftsobjekten, vier neuen Nachrichten und acht neuen Handlern erstellt werden.

Und jetzt überlegen Sie, was diese ausführbaren Dateien tun müssten, um die einfachsten Funktionen zu implementieren. Bedenken Sie dabei all die Objektinstanziierungen, Serialisierungen, das Marshalling und das Demarshalling, das Erstellen und Parsen von Nachrichten, die Socket-Kommunikation, die Timeout-Manager, die Retry-Szenarien sowie all die anderen Extras, die durchgeführt werden müssten, um eine einfache Sache zu erledigen.

Natürlich verfügten die Programmierer der Firma P während der gesamten Entwicklungszeit ihres Systems nicht über eine Serverfarm, sondern führten einfach alle drei ausführbaren Dateien in drei verschiedenen Prozessen auf einer einzigen Maschine aus. So gingen sie einige Jahre lang zu Werke, und trotzdem waren sie überzeugt, dass ihre Systemarchitektur richtig war. Und deshalb nahmen sie, obwohl sie die Dateien auf einer einzigen Maschine ausführten, dennoch sämtliche Objektinstanziierungen, Serialisierungen, das gesamte Marshalling und Demarshalling, das Erstellen und Parsen von Meldungen, die Socket-Kommunikation und alle Extras auch weiterhin auf nur einer Maschine vor.

Die Ironie an dieser Geschichte ist, dass das Unternehmen niemals ein System verkauft hat, das eine Serverfarm benötigt hätte. Vielmehr war jedes von ihnen deployte System ein einzelner Server. Und auf diesem Server wurden in Erwartung einer Serverfarm, die nie existierte – und nie existieren würde –, weiterhin alle drei ausführbaren Dateien, sämtliche Objektinstanziierungen, alle Serialisierungen, das gesamte Marshalling und Demarshalling, das komplette Erstellen und Parsen von Nachrichten, die gesamte Socket-Kommunikation sowie alle anderen Extras abgewickelt.

Besonders tragisch daran ist, dass die Softwarearchitekten den Entwicklungsaufwand durch das Fällen ihrer voreiligen Entscheidung enorm erhöht hatten.

Und dabei ist die Geschichte des Unternehmens P keineswegs ein Einzelfall. Ich selbst habe sie viele Male in vielen verschiedenen Situationen erlebt – P repräsentiert im Grunde genommen nur eine Überlagerung all dieser Fälle.

Aber es gibt noch schlimmere Schicksale als das dieses Unternehmens.

Betrachten wir zum Beispiel einmal den Fall der Firma W, eines regionalen Unternehmens, das sich mit der Verwaltung von Firmenwagenflotten befasst. Hier wurde soeben ein »Softwarearchitekt« eingestellt, um das Chaos des hauseigenen Softwaresammelsuriums in den Griff zu bekommen. Und ich kann Ihnen versichern: »In den Griff bekommen« war der zweite Vorname dieses Mannes. Schnell war er zu dem Schluss gekommen, dass eine ausgewachsene *unternehmenstaugliche serviceorientierte »SOFTWAREARCHITEKTUR«* genau das war, was dieser kleine Betrieb brauchte. Also erstellte er ein riesiges Domänenmodell all der verschiedenen

»Objekte« des Unternehmens, entwarf eine Service-Suite zur Verwaltung dieser Domänenobjekte und schickte sämtliche verfügbaren Softwareentwickler schnurstracks auf den Weg in Richtung Hölle.

Nehmen Sie als einfaches Beispiel einmal an, Sie wollten den Namen, die Anschrift und die Telefonnummer zu dem Verkaufsdatensatz einer Kontaktperson ergänzen. Dazu müssten Sie in der `ServiceRegistry` die Service-ID von `ContactService` abfragen. Dann müssten Sie eine `CreateContact`-Meldung an `ContactService` senden. Natürlich würde diese Meldung Dutzende von Feldern mit gültigen Daten enthalten – Daten, auf die der Programmierer keinen Zugriff hätte, denn ihm stünden lediglich ein Name, eine Anschrift und eine Telefonnummer zur Verfügung. Nachdem er also entsprechende Platzhalterdaten angegeben hat, würde der Programmierer die ID des neu erstellten Kontakts in den Verkaufsdatensatz einpflegen und die `UpdateContact`-Meldung an den `SaleRecordService` senden.

Um das Ganze nun zu testen, müssten Sie anschließend nacheinander alle notwendigen Services starten, den Message Bus und den *BPEL-Server* (**B**usiness **P**rocess **E**xecution **L**anguage) hochfahren und noch einiges mehr. Das hätte allerdings wiederum Laufzeitverzögerungen zur Folge, während diese Meldungen von Service zu Service wandern und in einer Warteschlange nach der anderen landen.

Und wenn Sie dann ein neues Feature ergänzen wollten – nun ja, Sie können sich die Kopplung all dieser Services und den schieren Umfang der *WSDLs* (**W**eb **S**ervices **D**escription **L**anguage), die modifiziert werden müssten, sowie alle erneuten Deployments, die dies mit sich bringen würden, sicher vorstellen ...

Im Vergleich dazu erscheint die Hölle geradezu wie ein kuscheliges Örtchen.

Grundsätzlich ist an einem Softwaresystem, das um Services herum strukturiert wird, erst einmal nichts auszusetzen. Der Fehler bei dem Unternehmen W war jedoch die vorzeitige Adaption und Umsetzung einer Tool-Suite, die SoA versprach – sprich die verfrühte Einführung einer umfangreichen Suite von Domänenobjekt-Services, denn: In der Folge schlug sich der damit einhergehende Aufwand auf reine Arbeitsstunden nieder, die vom SoA-Strudel regelrecht fortgerissen wurden – und zwar zuhauf.

Ich könnte Ihnen aus dem Stegreif noch jede Menge weitere Geschichten von gescheiterten Softwarearchitekturen erzählen, aber lassen Sie uns stattdessen doch lieber mal einen Blick auf echte architektonische Erfolgsmodelle werfen.

17.2 FitNesse

Im Jahr 2001 begannen mein Sohn Micah und ich gemeinsam mit der Arbeit an dem Testframework *FitNesse*. Die Idee dabei war, ein einfaches Wiki zu erstellen, das

Ward Cunninghams *FIT-Tool* (**F**ramework for **I**ntegrated **T**est) zum Schreiben von Akzeptanztests umfasste.

Das war zu einem Zeitpunkt, bevor Maven das Problem mit den `.jar`-Dateien »gelöst« hatte. Ich bestand darauf, dass nichts von dem, was wir erzeugten, es erforderlich machen dürfte, dass mehr als eine `.jar`-Datei heruntergeladen werden müsste. Ich nannte diese Regel »Download and Go«. Sie bildete die Basis für viele unserer Entscheidungen.

Eine der ersten dieser Entscheidungen war, unseren eigenen Webserver zu schreiben, der speziell auf die Bedürfnisse von FitNesse zugeschnitten war. Das mag zwar absurd klingen, denn natürlich gab es auch 2001 schon jede Menge Open-Source-Webserver, derer wir uns hätten bedienen können. Dennoch erwies sich diese Wahl als richtig, weil ein Barebone-Webserver lediglich das Schreiben einer simplen kleinen Software erfordert und es uns ermöglichte, alle Entscheidungen bezüglich des Webframeworks bis zu einem sehr viel späteren Zeitpunkt aufzuschieben.[2]

Eine weitere frühe Überlegung bestand darin, erst einmal nicht weiter über eine Datenbank nachzudenken. Wir hatten zwar MySQL im Hinterkopf, verschoben diese Entscheidung jedoch ganz bewusst, indem wir ein Design nutzten, das sie vorerst irrelevant machte und einfach darin bestand, dass wir eine Schnittstelle zwischen alle Datenzugriffe und den eigentlichen Datenspeicher setzten.

Die Methoden für den Datenzugriff legten wir in eine Schnittstelle namens `WikiPage`. Diese Methoden stellten die komplette für das Auffinden, Abrufen und Speichern von Seiten erforderliche Funktionalität bereit. Selbstverständlich haben wir sie zunächst nicht implementiert, sondern einfach als *Stubs* (eine Art stellvertretende Proxies) angelegt, während wir an Features arbeiteten, die keinen Datenabruf und keine Datenspeicherung erforderten.

Tatsächlich haben wir drei Monate nur daran gewerkelt, den Wiki-Text in HTML zu übersetzen. Dafür war keinerlei Datenspeicherung nötig, deshalb erstellten wir eine Klasse mit Namen `MockWikiPage`, die die Datenzugriffsmethoden einfach als Stubs beließ.

Schließlich erwiesen sich diese vorgehaltenen Methoden für die Features, die wir schreiben wollten, jedoch als unzureichend. Wir brauchten echten Datenzugriff, keine Stubs. Also erstellten wir eine neue Ableitung von `WikiPage`, die wir `InMemoryPage` nannten. Diese Ableitung implementierte die Datenzugriffsmethode so, dass sie eine Hashtabelle der Wiki-Seiten verwaltete, die wir im RAM bereithielten.

Auf diese Weise war es uns möglich, ein ganzes Jahr lang Feature um Feature zu schreiben. Eigentlich brachten wir dadurch die gesamte erste Version des FitNesse-Programms ans Laufen. Wir konnten Seiten erstellen, Links zu anderen Seiten bereitstellen, die gesamte Wiki-typische Formatierung vornehmen und sogar Tests mit

FIT durchführen. Was wir jedoch nicht konnten, war, unsere Arbeit zu speichern.

Als es dann an der Zeit war, etwas mehr Dauerhaftigkeit zu implementieren, zogen wir noch einmal MySQL in Betracht, beschlossen aber letztlich, dass dies kurzfristig nicht nötig war, weil die Hashtabellen wirklich sehr einfach in *Flat Files* (reine, unstrukturierte Datendateien) zu schreiben waren. Also implementierten wir stattdessen `FileSystemWikiPage`, um einfach nur die Funktionalität in Flat Files zu verschieben, und fuhren anschließend damit fort, weitere Features zu entwickeln.

Drei Monate später kamen wir zu dem Schluss, dass die Flat-File-Lösung gut genug war – und verwarfen somit die Idee von MySQL gänzlich. Faktisch haben wir diese Entscheidung damit sozusagen bis in die Nichtexistenz aufgeschoben und nie zurückgeschaut.

Und damit wäre die Geschichte eigentlich auch schon am Ende, wenn nicht einer unserer Kunden beschlossen hätte, dass er das Wiki für seine eigenen Zwecke in MySQL einbauen müsse. *Nur einen Tag später* berichtete er uns schon, dass das System in MySQL lief: Er hatte einfach eine `MySqlWikiPage`-Ableitung geschrieben und es so zum Funktionieren gebracht.

Wir haben diese Option mit FitNesse gebündelt, allerdings war sie nie von irgendwem genutzt worden, deshalb haben wir sie am Ende wieder verworfen. Und selbst der Kunde, der die Ableitung geschrieben hatte, nahm später wieder Abstand davon.

Zu Beginn der Entwicklung von FitNesse hatten wir eine Grenze zwischen Geschäftsregeln und Datenbanken gezogen. Diese Grenzlinie verhinderte, dass die Geschäftsregeln, abgesehen von den einfachen Datenzugriffsmethoden, Kenntnis von der Datenbank hatten. Aufgrund dieser Entscheidung war es uns möglich, die endgültige Auswahl und Implementierung der Datenbank weit über ein Jahr hinauszuzögern. Und das wiederum bot uns Gelegenheit, die Dateisystemoption auszuprobieren und eine andere Richtung einzuschlagen, als wir eine bessere Lösung erkannten. Es hinderte jedoch niemanden daran – oder hatte irgendwelche negativen Auswirkungen darauf –, auch in die ursprünglich angedachte Richtung weiterzugehen, wenn dies gewünscht war.

Die Tatsache, dass wir 18 Monate lang überhaupt keine Datenbank hatten, bedeutete, dass wir in dieser Zeit auch keine schemabezogenen Schwierigkeiten sowie Abfrage-, Datenbankserver-, Passwort-, Verbindungszeit- oder irgendwelche anderen lästigen Probleme hatten, die sich normalerweise von ihrer hässlichen Seite zeigen, wenn man eine Datenbank startet. Und es bedeutete auch, dass alle unsere Tests schnell liefen, weil es ja keine Datenbank gab, die sie verlangsamen konnte.

Kurz: Das Errichten von Grenzen hat uns geholfen, Entscheidungen hinauszuzögern bzw. aufzuschieben – was uns wiederum letzten Endes jede Menge Zeit und Kopfschmerzen erspart hat. Und genau das sollte eine gute Softwarearchitektur auch

tun.

17.3 Welche Grenzen sollten Sie ziehen – und wann?

Grundsätzlich gilt es, Grenzlinien zwischen jenen Aspekten des Systems zu ziehen, die von Belang sind, und solchen, die es nicht sind. Das GUI spielt für die Geschäftsregeln keine Rolle, daher sollte hier eine Linie gezogen werden. Die Datenbank spielt für das GUI keine Rolle, deshalb sollte auch hier eine Grenze gezogen werden. Und da die Datenbank ihrerseits für die Geschäftsregeln ohne Belang ist, gilt hier dasselbe.

Nun mag der eine oder andere von Ihnen diese Aussagen ganz oder teilweise zurückweisen – insbesondere die über die Geschäftsregeln und die Datenbank. Viele Softwareentwickler haben ursprünglich einmal gelernt, dass die Datenbank untrennbar mit den Geschäftsregeln verbunden ist. Und manche sind sogar davon überzeugt, dass sie gar die Verkörperung der Geschäftsregeln darstellt.

Wie Sie jedoch später noch sehen werden, ist diese Vorstellung fehlgeleitet. Die Datenbank ist im Wesentlichen vielmehr ein Tool, das die Geschäftsregeln indirekt nutzen können. Letztere müssen keine Kenntnis von dem zugrunde liegenden Schema oder der Sprache oder irgendwelchen anderen Einzelheiten hinsichtlich der Datenbank haben. Alles, was den Geschäftsregeln bekannt sein muss, ist, dass es einen Satz von Funktionen gibt, der verwendet werden kann, um Daten abzurufen oder zu speichern. Und das ermöglicht es uns Softwareentwicklern, die Datenbank hinter eine Schnittstelle zu setzen.

In Abbildung 17.1 ist dies deutlich zu erkennen: Die Klasse `BusinessRules` nutzt `DatabaseInterface` zum Laden und Speichern der Daten, während die `DatabaseAccess`-Klasse die Schnittstelle implementiert und den Betrieb der eigentlichen Datenbank `Database` steuert.

Die Klassen und Schnittstellen in diesem Diagramm dienen lediglich als symbolische Beispiele. In einer realen Anwendung würde es natürlich zahlreiche Geschäftsregelklassen, viele Datenbankschnittstellenklassen und jede Menge Datenbankzugriffsimplementierungen geben, die jedoch allesamt ungefähr dem gleichen Muster folgen.

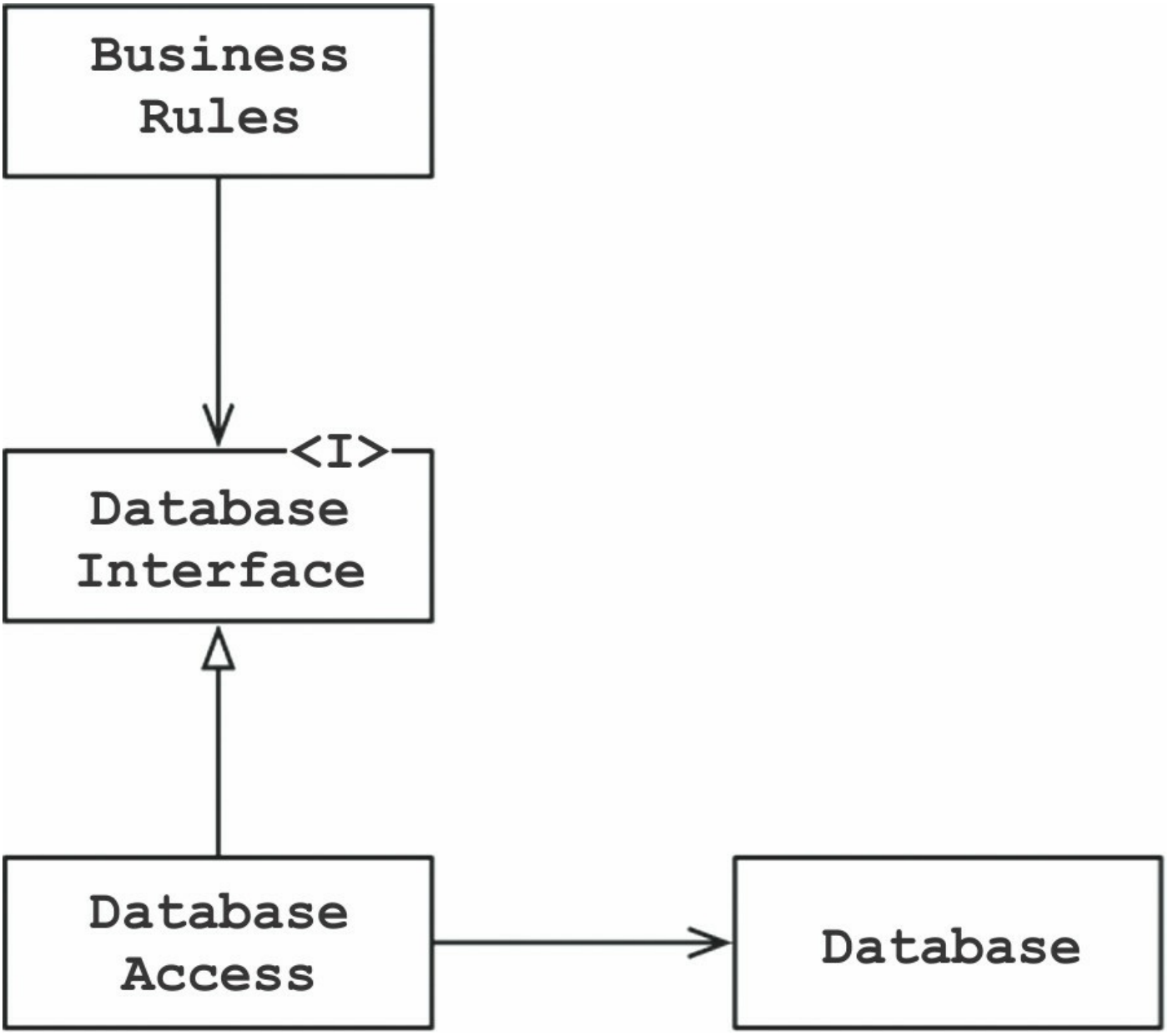


Abb. 17.1: Die Datenbank hinter einer Schnittstelle

Doch wo befindet sich denn nun die Grenzlinie? Sie verläuft unmittelbar geradewegs über die Vererbungsbeziehung hinweg, unmittelbar unterhalb der Klasse `DatabaseInterface` (siehe Abbildung 17.2).

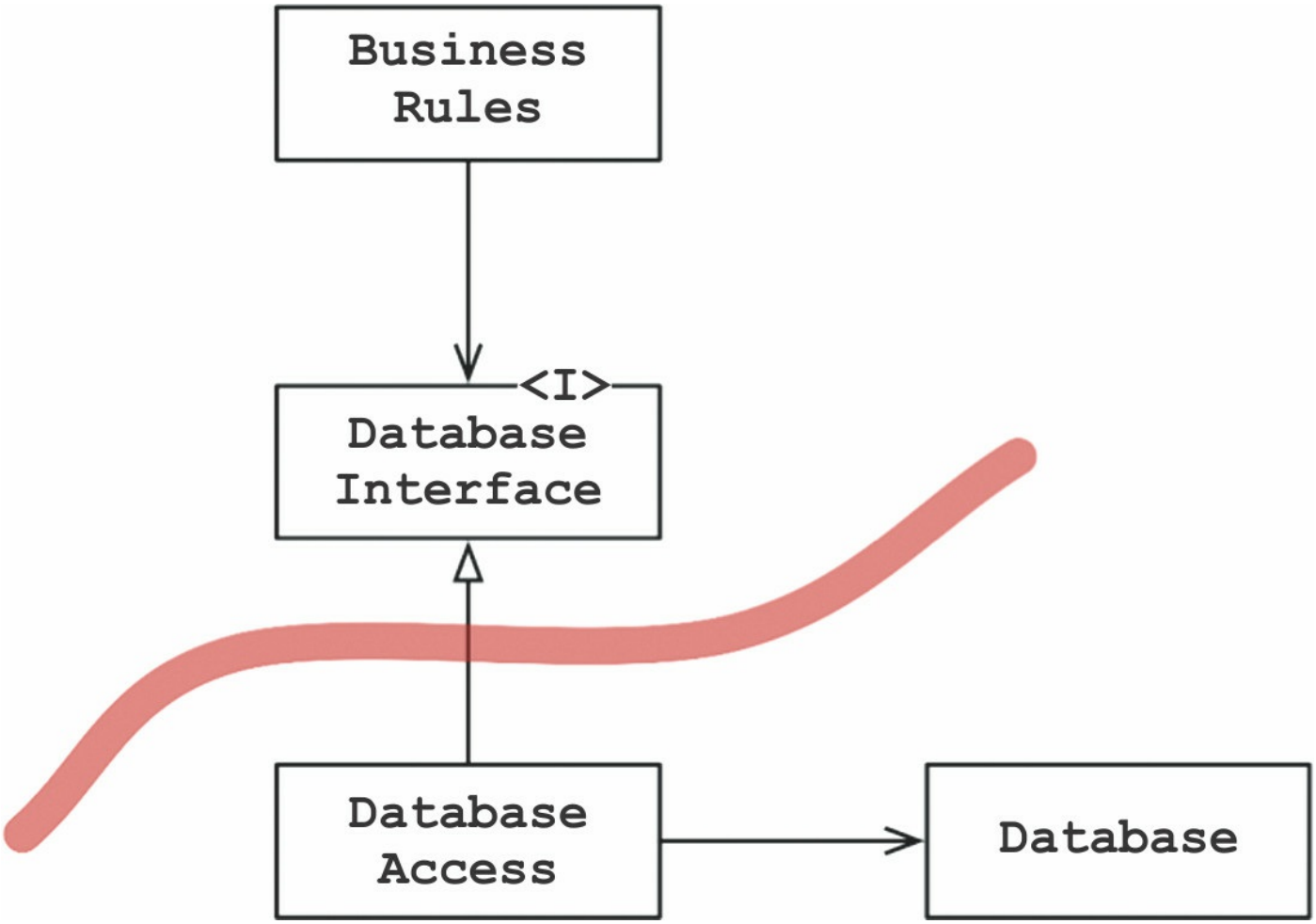


Abb. 17.2: Die Grenzlinie

Beachten Sie hier die zwei von der Klasse `DatabaseAccess` abgehenden Pfeile. Sie weisen von ihr weg – was so viel bedeutet, dass keine der anderen betreffenden Zielklassen (in diesem Fall `DatabaseInterface` und `Database`) Kenntnis von der Existenz der `DatabaseAccess`-Klasse hat.

Treten wir als Nächstes einmal einen Schritt zurück und betrachten die Komponente, die zahlreiche Geschäftsregeln enthält, und die Komponente, die die Datenbank und alle ihre Zugriffsklassen enthält (siehe Abbildung 17.3).

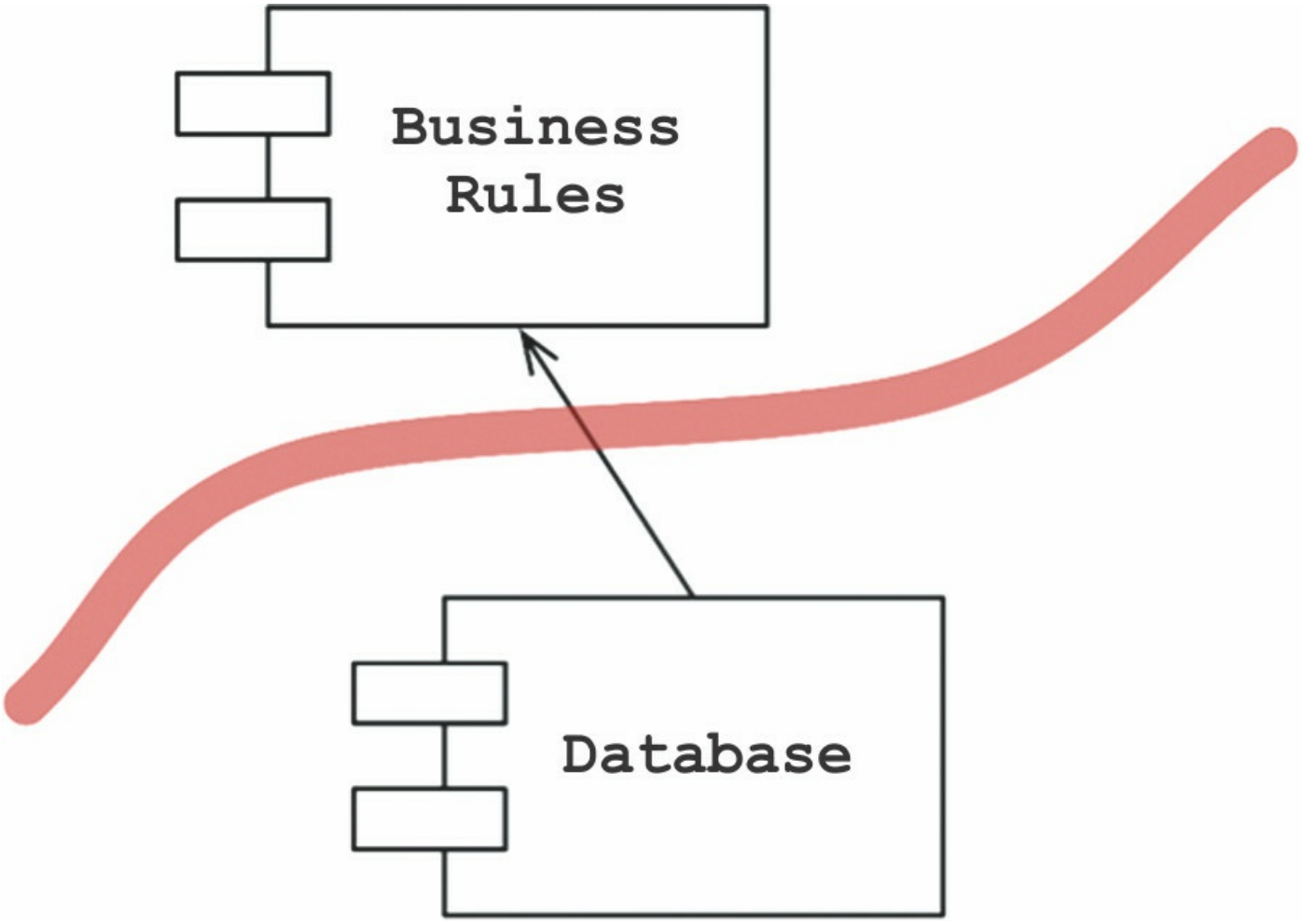


Abb. 17.3: Die Komponenten für die Geschäftsregeln und die Datenbank

Beachten Sie auch hier die Pfeilrichtung. In diesem Fall hat die Komponente `Database` Kenntnis von der Komponente `BusinessRules` – umgekehrt weiß die Komponente `BusinessRules` aber nichts von der Komponente `Database`. Damit wird deutlich, dass die `DatabaseInterface`-Klassen in der `BusinessRules`-Komponente enthalten sind, während sich die `DatabaseAccess`-Klassen in der `Database`-Komponente befinden.

Die Ausrichtung dieser Grenzlinie ist insofern von großer Bedeutung, als sie darlegt, dass die Komponente `Database` zwar nicht für die Komponente `BusinessRules` von Belang ist, andererseits aber ohne die Komponente `BusinessRules` nicht existieren kann.

Wenn Ihnen das seltsam vorkommen sollte, erinnern Sie sich an Folgendes: Die Komponente `Database` enthält den Code, der die von `BusinessRules` ausgehenden Aufrufe in die Abfragesprache der Datenbank übersetzt. Und deshalb besitzt dieser Übersetzungscode auch Kenntnis von der Komponente `BusinessRules`.

Angesichts der Grenzlinie zwischen den beiden Komponenten und der Ausrichtung des Pfeils auf die Komponente `BusinessRules` wird nun deutlich, dass diese faktisch jede beliebige Art von Datenbank nutzen könnte. Demnach könnte `Database` also durch viele verschiedene Implementierungen ersetzt werden, ohne dass dies für die

Komponente `BusinessRules` eine Rolle spielen würde.

Die Datenbank könnte mit Oracle, MySQL, Couch, Datomic oder auch simplen Flat Files implementiert werden. Das ist für die Geschäftsregeln ohne Belang und führt in der Konsequenz dazu, dass die Entscheidung hinsichtlich der Datenbank aufgeschoben werden kann und Sie sich stattdessen zunächst einmal darauf konzentrieren können, die Geschäftsregeln schriftlich zu formulieren und zu testen, bevor Sie sich endgültig auf eine Datenbank festlegen.

17.4 Wie verhält es sich mit der Ein- und Ausgabe?

Softwareentwickler und Kunden haben häufig eine falsche Auffassung davon, was das System als solches eigentlich ist. Sie sehen die Benutzerschnittstelle (GUI) und betrachten sie als gleichbedeutend mit dem System. Im Prinzip definieren sie ein System anhand des GUI und unterstellen, dass es damit auch sofort funktionieren sollte. Allerdings verkennen sie dabei einen maßgeblichen Grundsatz: *Die Ein-/Ausgabe (I/O) ist in diesem Zusammenhang irrelevant.*

Das mag zunächst einmal schwer zu verstehen sein, weil das Systemverhalten oftmals mit dem I/O-Verhalten gleichgesetzt wird. Ein gutes Beispiel hierfür sind Videospiele, bei denen die User Experience zweifellos von der Benutzerschnittstelle dominiert wird: der Grafik, der Maussteuerung, der Tastensteuerung und der Soundausgabe. Allerdings darf man dabei nicht vergessen, dass hinter dieser Schnittstelle ein Modell – ein hochentwickelter Satz von Datenstrukturen und -funktionen – steckt, der all dies steuert. Und was noch wichtiger ist: Dieses Modell ist nicht auf die Schnittstelle angewiesen, sondern würde seine Aufgaben, sprich die Modellierung sämtlicher Spielereignisse, auch dann erledigen, wenn das Spiel überhaupt nicht auf dem Bildschirm ausgegeben würde. Die Schnittstelle spielt für das Modell – die Geschäftsregeln – also überhaupt keine Rolle.

Und so wird auch hier wieder deutlich, dass die Komponenten `GUI` und `BusinessRules` durch eine Grenzlinie voneinander getrennt sind (siehe Abbildung 17.4) und dass die weniger relevante Komponente von der stärker relevanten Komponente abhängig ist. Die Pfeilausrichtung zeigt, welche Komponente Kenntnis von der jeweils anderen hat, und gibt damit auch an, welche Komponente für die andere von Belang ist. In diesem Beispiel ist die Komponente `BusinessRules` für die Komponente `GUI` von Bedeutung.

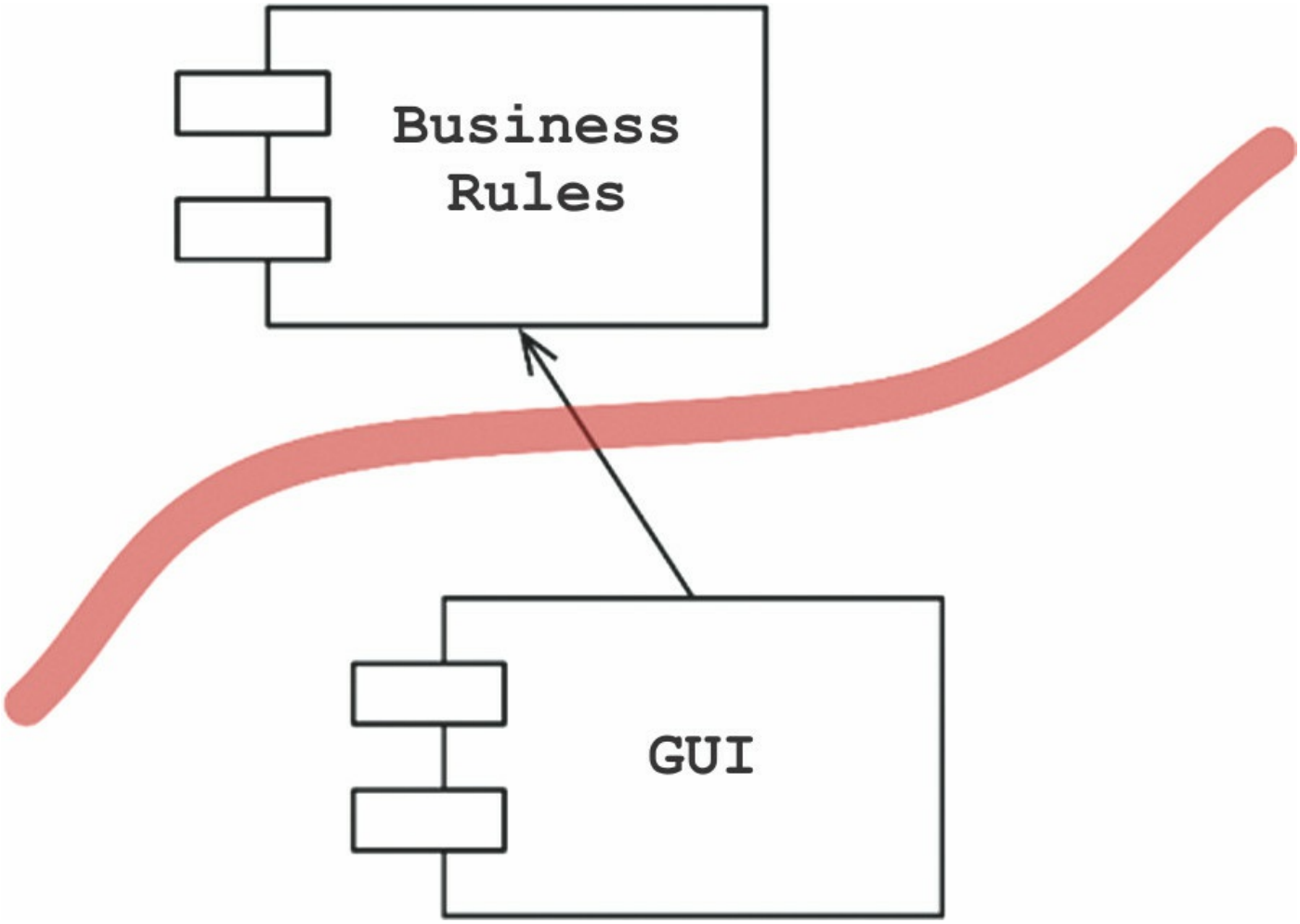


Abb. 17.4: Die Grenzlinie zwischen den Komponenten `GUI` und `BusinessRules`

Durch das Ziehen dieser Grenze und die Ausrichtung des Pfeils lässt sich nun ebenfalls erkennen, dass die Komponente `GUI` durch eine beliebige andere Schnittstelle ersetzt werden könnte – ohne dass sich dies auf die Komponente `BusinessRules` auswirken würde.

17.5 Plug-in-Architektur

Zusammengenommen ergeben diese beiden Entscheidungen in Bezug auf die Datenbank und die Benutzerschnittstelle eine Art Muster für die Ergänzung weiterer Komponenten. Dieses Muster entspricht demjenigen, das von Systemen genutzt wird, die Third-Party-Plug-ins zulassen.

Tatsächlich geht es in der Softwareentwicklung von jeher um die Frage, wie sich die möglichst praktischen Plug-ins zur Errichtung einer skalierbaren und wartbaren Systemarchitektur erstellen lassen. Die zentralen Geschäftsregeln werden separat und unabhängig von denjenigen Komponenten gehalten, die entweder optional sind oder in vielen verschiedenen Formen implementiert werden können (siehe Abbildung 17.5).

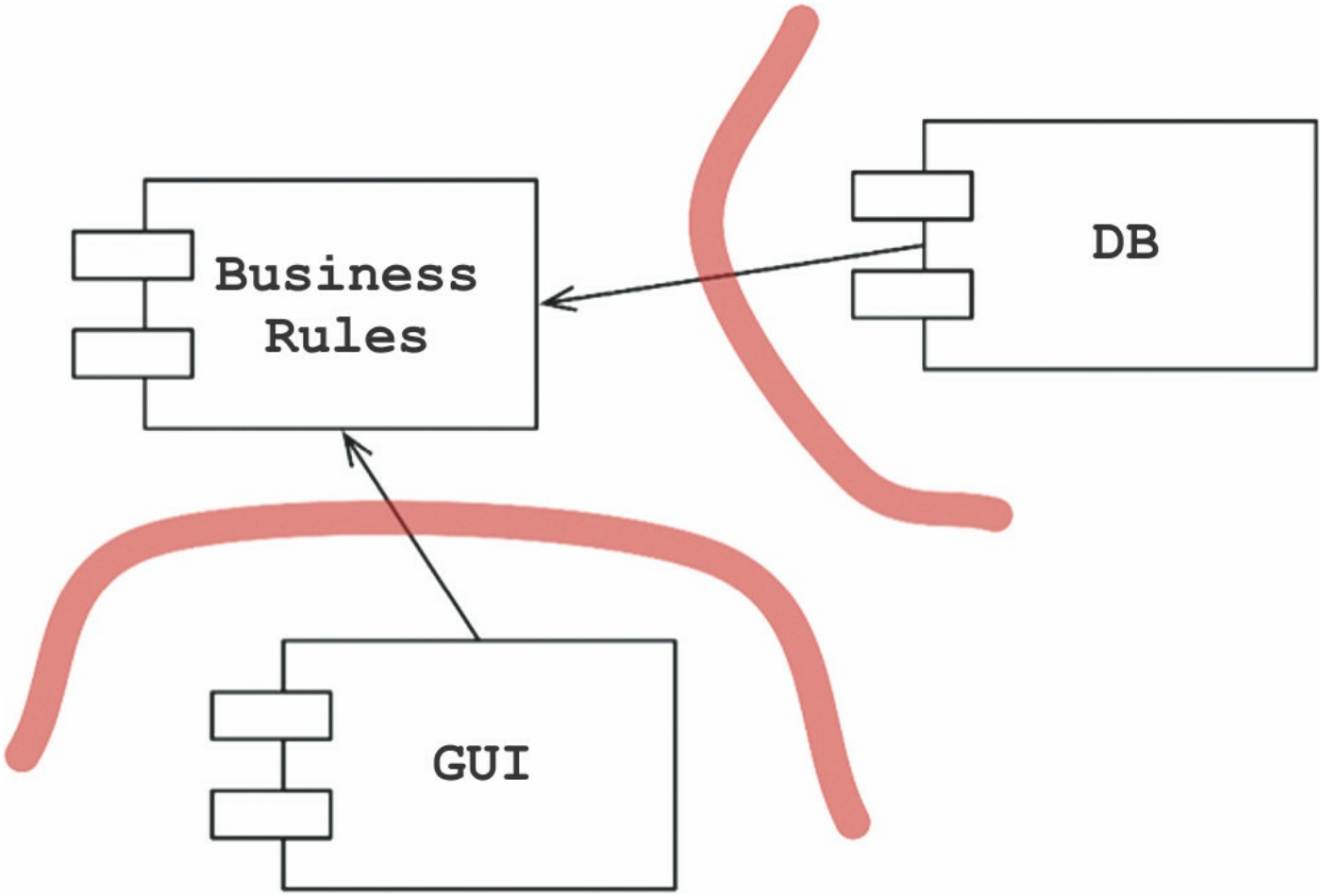


Abb. 17.5: Andocken an die Geschäftsregeln

Weil die Benutzerschnittstelle in diesem Design als Plug-in zu betrachten ist, bleibt hier die Möglichkeit offen, an viele verschiedene Arten von GUIs anzudocken. Diese könnten webbasiert, Client/Server-basiert, SOA-basiert oder konsolenbasiert sein oder auf jeder beliebigen anderen Art von Benutzerschnittstellentechnologie aufbauen.

Dasselbe gilt auch für die Datenbank. Da wir uns in diesem Beispiel entschieden haben, sie als Plug-in zu behandeln, können wir sie mit jeder der diversen SQL-Datenbanken, einer NOSQL-Datenbank, einer dateisystembasierten Datenbank oder jeder beliebigen anderen Art von Datenbanktechnologie ersetzen, sollte dies in Zukunft nötig werden.

Solche Ersetzungen sind allerdings nicht trivial. Wenn das ursprüngliche Deployment des Systems webbasiert war, dann könnte das Schreiben des Plug-ins für eine Client/Server-basierte Benutzerschnittstelle zur Herausforderung werden. Es ist wahrscheinlich, dass die Kommunikation zwischen den Geschäftsregeln und der neuen Benutzerschnittstelle teilweise überarbeitet werden muss – dennoch wurde durch die ursprüngliche Annahme einer Plug-in-Struktur eine solche Anpassung zumindest praktikabel gestaltet.

17.6 Das Plug-in-Argument

Vergleichen Sie nun einmal die Situation zwischen ReSharper und Visual Studio. Diese Softwarekomponenten werden von völlig verschiedenen Entwicklungsteams in völlig verschiedenen Unternehmen produziert. JetBrains, die Softwareschmiede, die ReSharper hervorgebracht hat, ist in Russland ansässig. Microsoft hingegen ist natürlich bekanntermaßen in Redmond, Washington, zu Hause. Zwei Entwicklungsteams, die weiter voneinander entfernt arbeiten, sind kaum vorstellbar.

Welches dieser Teams kann nun der Arbeit des anderen in die Quere kommen? Und welches ist dem anderen Team gegenüber »immun«? Das Abhängigkeitsdiagramm in Abbildung 17.6 liefert die Antworten auf diese Fragen: Der ReSharper-Quellcode ist von dem Quellcode von Visual Studio abhängig. Dementsprechend besteht keine Gefahr, dass das ReSharper-Team etwas tun könnte, das die Arbeit des Teams von Visual Studio beeinträchtigen würde. Andererseits könnte das Visual-Studio-Team das Team von ReSharper dagegen sogar komplett außer Gefecht setzen, wenn es dies denn wollte.

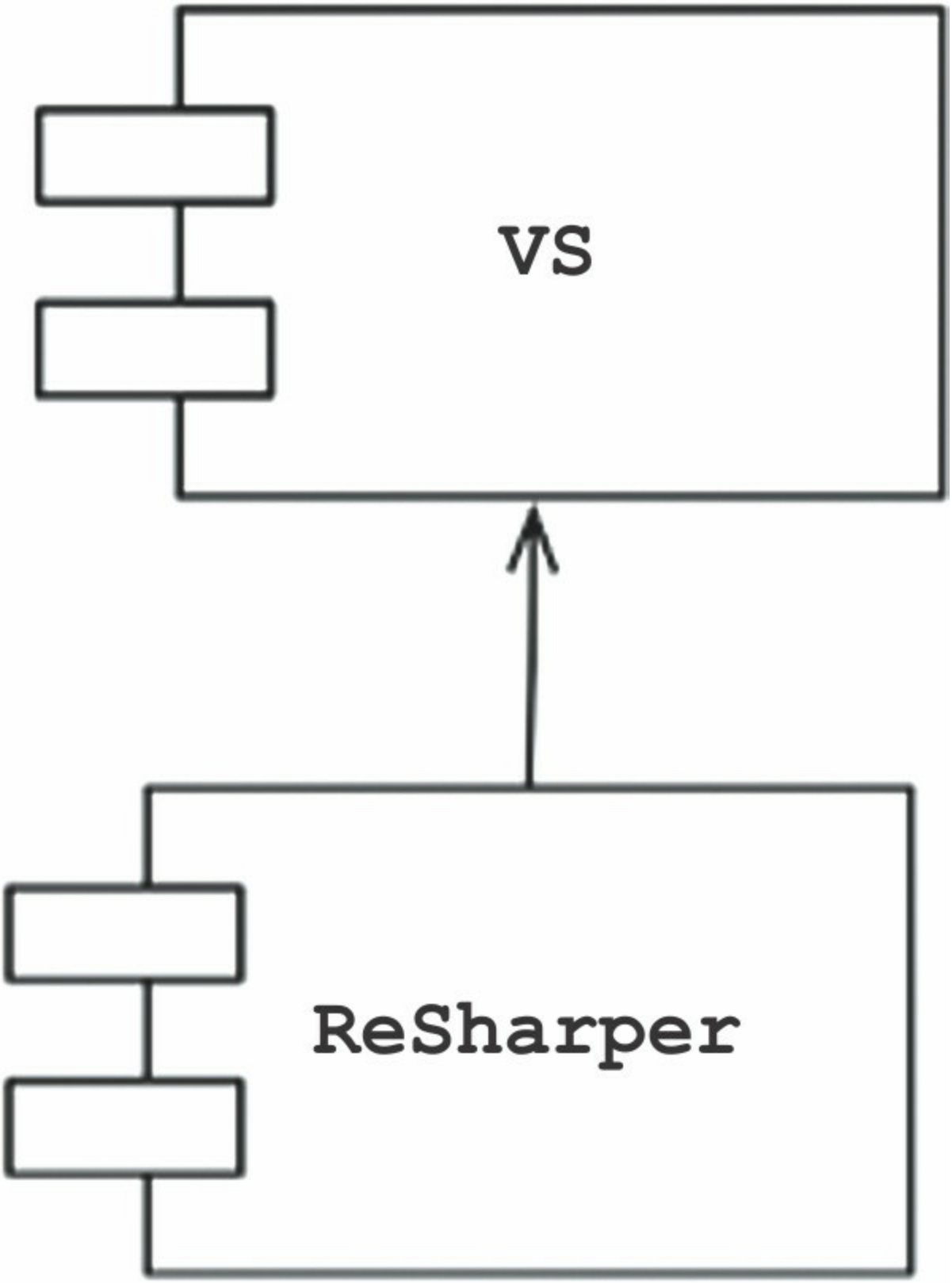


Abb. 17.6: ReSharper ist von Visual Studio abhängig.

Hierbei handelt es sich um eine zutiefst asymmetrische Beziehung, die wir uns auch für unsere eigenen Systeme wünschen. Bestimmte Module sollten anderen gegenüber immun sein. Beispielsweise ist es keinesfalls wünschenswert, dass die Geschäftsregeln von Formatänderungen an einer Webseite oder dem Datenbankschema beeinträchtigt werden. Grundsätzlich sollten Modifikationen an einem Teilbereich des Systems nicht dazu führen, dass andere, nicht verwandte Bereiche des Systems dadurch Schaden

nehmen bzw. nicht mehr funktionieren. Unsere Systeme sollten diese Art von Fragilität nicht aufweisen.

Der Aufbau der Systeme als Plug-in-Architekturen schützt sie durch eine Firewall, die Modifikationen nicht durchdringen können. Wenn das GUI an die Geschäftsregeln »andockt«, dann können daran vorgenommene Modifikationen diese Geschäftsregeln somit nicht beeinträchtigen.

Grenzen werden entlang der »Modifikationsachse« gezogen, soll heißen: Die Komponenten auf der einen Seite der Grenzlinie verändern sich zu anderen Zeitpunkten und aus anderen Gründen als die Komponenten auf der anderen Seite.

GUIs ändern sich zu anderen Zeitpunkten und mit einer anderen Frequenz als die Geschäftsregeln, deshalb sollten sie voneinander abgegrenzt werden. Die Geschäftsregeln wiederum ändern sich zu anderen Zeitpunkten und aus anderen Gründen als die Dependency-Injection-Frameworks, demzufolge sollte auch hier eine Grenzlinie gezogen werden.

Dieses Vorgehen entspricht im Wesentlichen einfach wieder dem *Single-Responsibility-Prinzip*, das aufzeigt, wo die Grenzen zu ziehen sind.

17.7 Fazit

Um in einer Softwarearchitektur Grenzen zu ziehen, teilen Sie das System zunächst in Komponenten auf. Einige davon repräsentieren die Kerngeschäftsregeln, andere sind Plug-ins, die notwendige, nicht in direktem Zusammenhang mit dem Kerngeschäft stehende Funktionen enthalten. Dann ordnen Sie den Code nach diesen Komponenten, sodass die ihnen zugewiesenen Pfeile in eine einzige Richtung weisen – zum Kerngeschäft.

Sie sollten diese Vorgehensweise als eine Anwendung des *Dependency-Inversion-Prinzips* und des *Stable-Abstractions-Prinzips* verstehen. Die Abhängigkeitspfeile werden so ausgerichtet, dass sie von den untergeordneten Details auf die übergeordneten Abstraktionen zeigen.

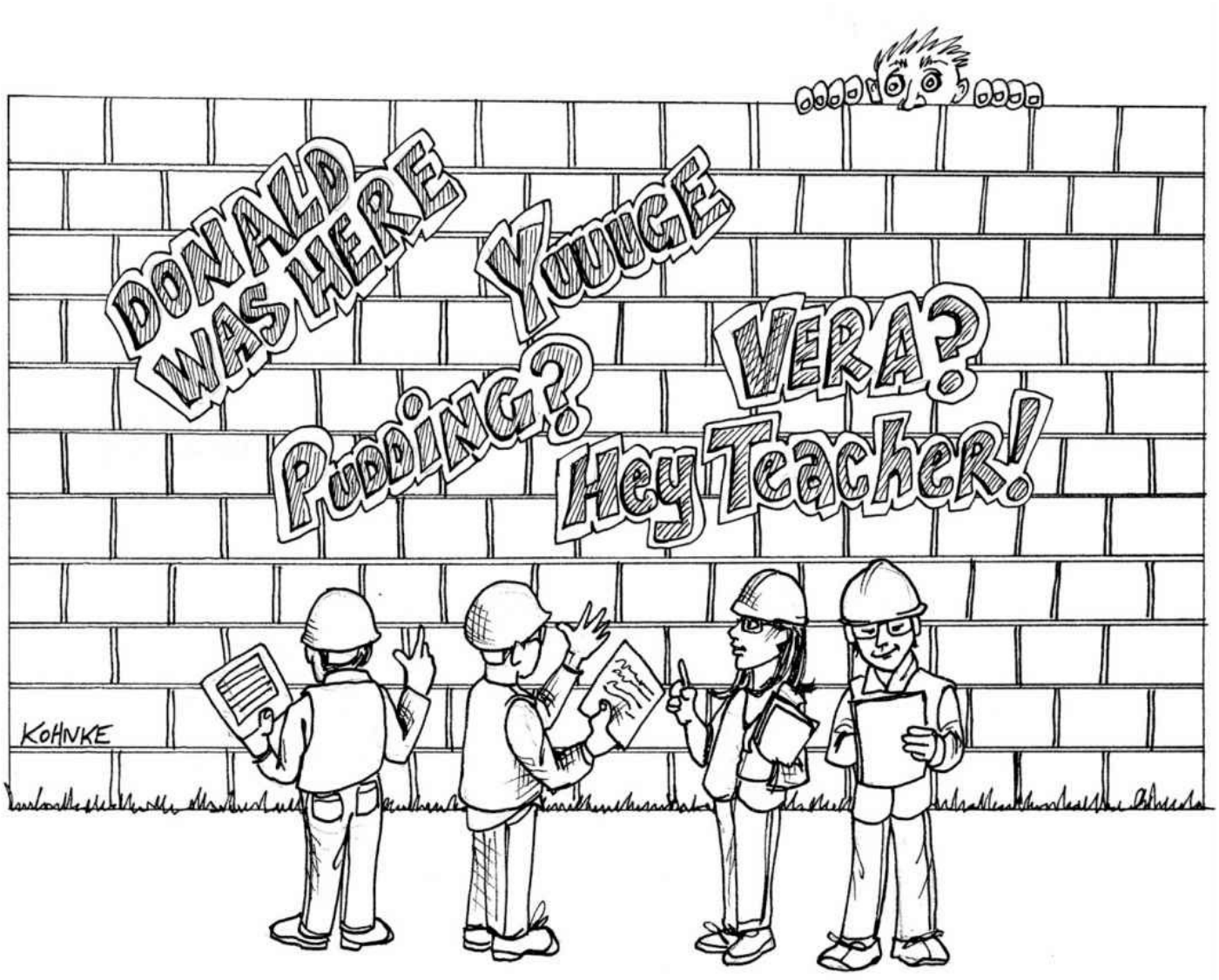
---

[1] Der Begriff »Architektur« ist hier in Anführungszeichen gesetzt, weil ein dreischichtiges Modell nicht wirklich eine Architektur ist, sondern vielmehr eine Topologie. Dies ist exakt die Art von Entscheidung, die in einer guten Softwarearchitektur aufgeschoben können werden sollte.

[2] Viele Jahre später waren wir in der Lage, das Velocity-Framework in FitNesse

einzubringen.

Kapitel 18
Anatomie der Grenzen



Die Architektur eines Systems definiert sich durch einen Satz von Softwarekomponenten sowie die Grenzlinien, die sie voneinander separieren. Diese Grenzen können verschiedene Formen annehmen, daher werden wir in diesem Kapitel die gebräuchlichsten von ihnen etwas näher betrachten.

18.1 Grenzüberschreitungen

Zur Laufzeit bedeutet eine Grenzüberschreitung nichts anderes, als dass eine Funktion auf der einen Seite der Grenzlinie eine Funktion auf der anderen Seite aufruft und

einige Daten übermittelt. Der Trick bei der Erstellung einer angemessenen Grenzüberschreitung besteht in der Verwaltung der Quellcode-Abhängigkeiten.

Aber warum Quellcode? Weil in dem Fall, dass ein Quellcodemodul geändert wird, auch andere Quellcodemodule modifiziert oder neu kompiliert und dann erneut deployt werden müssen. Und das ist es, worum es beim Ziehen dieser Grenzen geht: die Errichtung und Verwaltung von Firewalls gegen solche Modifikationen.

18.2 Der gefürchtete Monolith

Die einfachsten und üblichsten architektonischen Grenzen besitzen keine strenge physische Form. Vielmehr handelt es sich um eine disziplinierte Trennung von Funktionen und Daten innerhalb eines einzigen Prozessors und eines einzigen Adressbereichs. In Kapitel 16 habe ich dies als »Entkopplung auf Quellcode-Ebene« bezeichnet.

Aus Sicht des Deployments ergibt sich dadurch nichts anderes als eine einzige ausführbare Datei – der sogenannte *Monolith*. Diese Datei kann ein statisch verknüpftes C- oder C++-Projekt sein oder eine Reihe von Java-Klassendateien, die in eine ausführbare `.jar`-Datei eingebunden sind, oder ein Satz von .NET-Binärdateien, die in einer einzelnen `.exe`-Datei zusammengefasst sind, usw.

Die Tatsache, dass die Grenzlinien während des Deployments eines Monolithen nicht sichtbar sind, bedeutet jedoch nicht, dass sie nicht präsent und bedeutsam wären. Selbst wenn sie statisch in eine einzelne ausführbare Datei verlinkt sind, ist die damit gebotene Fähigkeit, die Entwicklung und das Marshalling der verschiedenen Komponenten für den finalen Einbau unabhängig voneinander durchführen zu können, von enormem Wert.

Solche Softwarearchitekturen sind zur Verwaltung ihrer internen Abhängigkeiten fast immer von einer Form von dynamischer Polymorphie[1] abhängig. Das ist einer der Gründe, weshalb die objektorientierte Entwicklung in den vergangenen Jahrzehnten so ein bedeutsames Paradigma geworden ist. Ohne die Objektorientierung oder eine vergleichbare Form der Polymorphie müssten die Softwareentwickler auf die risikobehaftete Praxis der Verwendung von Funktionszeigern zurückgreifen, um eine geeignete Entkopplung zu erzielen. Die meisten Softwarearchitekten halten den produktiven Einsatz von Funktionszeigern jedoch für zu riskant und sind deshalb gezwungen, jegliche Art von Komponententrennung aufzugeben.

Die einfachste mögliche Variante der Grenzüberschreitung ist ein Funktionsaufruf vonseiten eines untergeordneten Clients an einen übergeordneten Service. Sowohl die Abhängigkeit zur Laufzeit als auch die Abhängigkeit zur Kompilierzeit weisen in

dieselbe Richtung, nämlich auf die übergeordnete Komponente.

In Abbildung 18.1 kreuzt der Kontrollfluss die Grenzlinie von links nach rechts. Der `Client` ruft die Funktion `f()` aus dem `Service` auf und übergibt eine Instanz von `Data`. Der `<DS>`-Marker weist lediglich auf eine Datenstruktur hin. `Data` darf nicht als Funktionsargument oder mittels irgendeiner anderen aufwendigeren Methode übergeben werden. Beachten Sie, dass sich die Definition von `Data` auf der *aufzurufenden Seite* der Grenzlinie befindet.

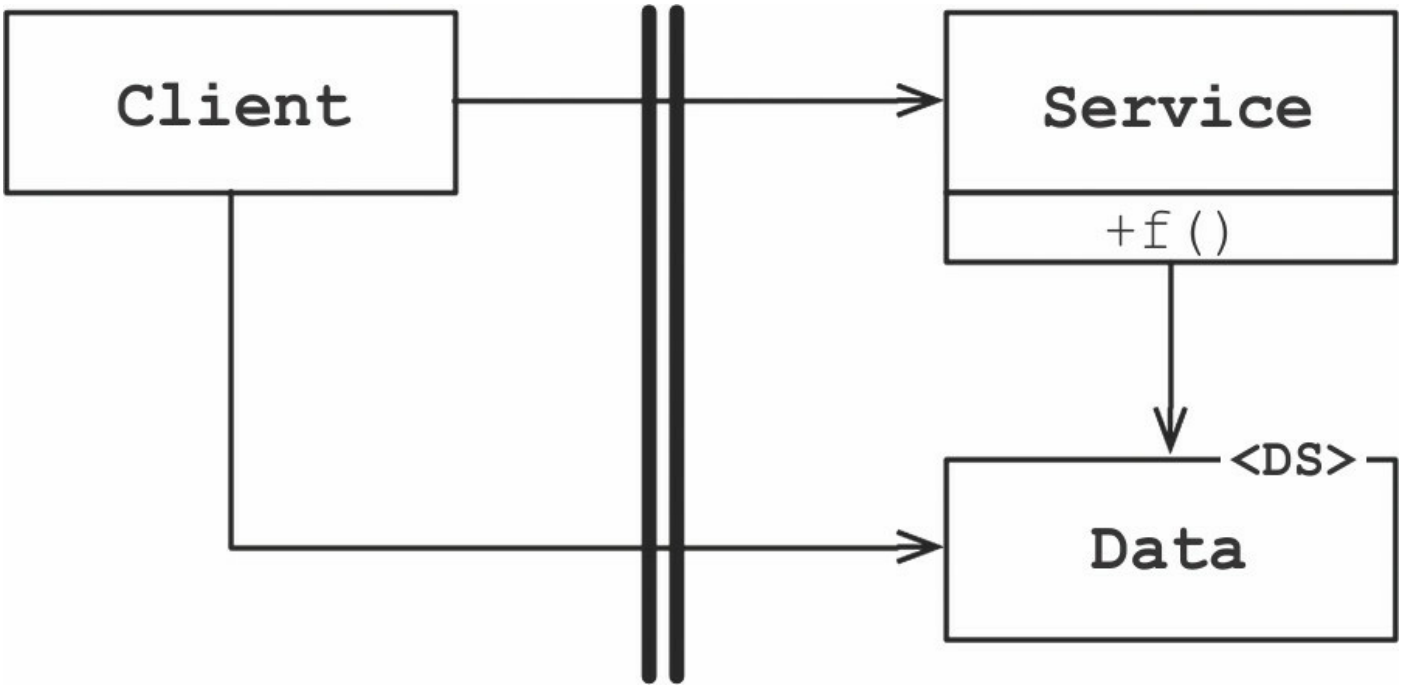


Abb. 18.1: Der Kontrollfluss überquert die Grenzlinie von einer niedrigen Ebene hin zu einer höheren Ebene.

Wenn ein übergeordneter Client einen untergeordneten Service aufrufen muss, dann wird die dynamische Polymorphie genutzt, um die Abhängigkeit gegen den Kontrollfluss zu invertieren. Die Abhängigkeit zur Laufzeit steht der Abhängigkeit zur Kompilierzeit gegenüber.

In Abbildung 18.2 kreuzt der Kontrollfluss die Grenzlinie wie zuvor wieder von links nach rechts. Der übergeordnete `Client` ruft die Funktion `f()` der untergeordneten Komponente `ServiceImpl` über die Schnittstelle `Service` auf. Beachten Sie in diesem Fall jedoch, dass alle Abhängigkeiten von rechts nach links *in Richtung der übergeordneten Komponente* über die Grenzlinie hinweg ausgerichtet sind. Anders ist hier außerdem, dass sich die Definition der Datenstruktur auf der *aufrufenden* Seite der Grenze befindet.

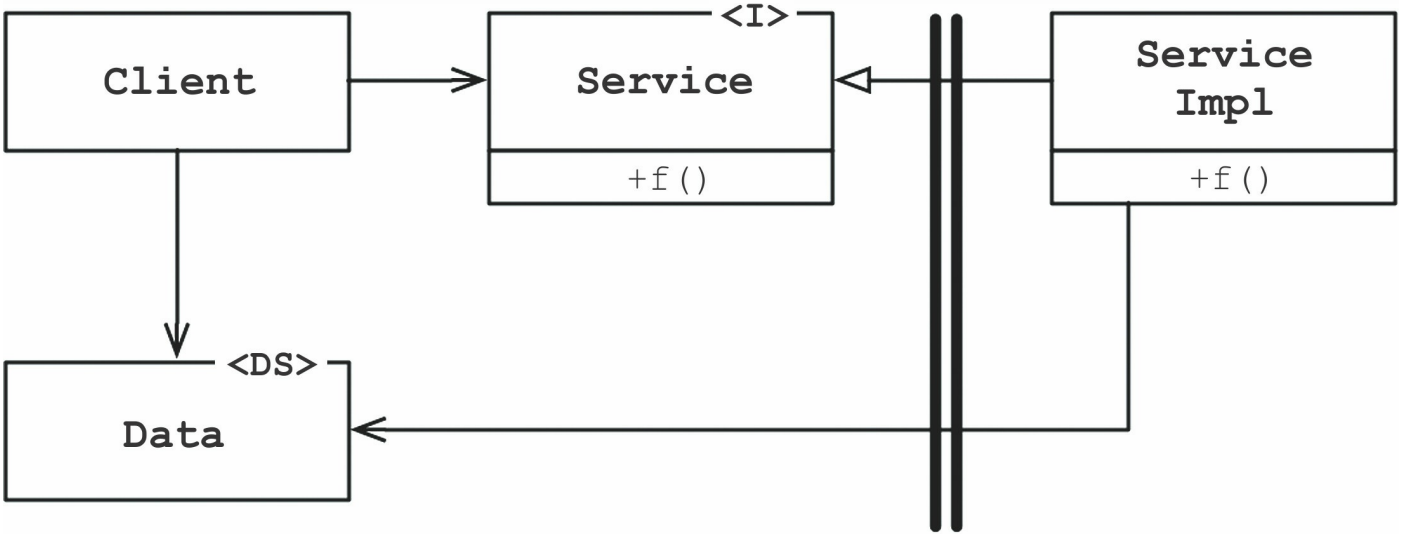


Abb. 18.2: Überquerung der Grenzlinie entgegen dem Kontrollfluss

Auch in einer monolithischen, statisch verknüpften ausführbaren Datei kann diese Art von disziplinierter Trennung eine große Hilfe beim Entwickeln, Testen und Deployen des Projekts sein. Die Teams können jeweils unabhängig voneinander an ihren eigenen Projekten arbeiten, ohne sich gegenseitig ins Gehege zu kommen. Und die übergeordneten Komponenten bleiben von den untergeordneten Details unabhängig.

Die Kommunikation zwischen den Komponenten in einem monolithischen System verläuft sehr schnell und kostengünstig. Dabei handelt es sich im Allgemeinen einfach nur um *Funktionsaufrufe*, und dementsprechend kann die Kommunikation über die auf Quellcode-Ebene entkoppelten Grenzen hinweg sehr lebhaft sein.

Da das Deployment von Monolithen normalerweise eine Kompilierung sowie eine statische Verlinkung erfordert, werden Komponenten in diesen Systemen üblicherweise als Quellcode bereitgestellt.

18.3 Deployment-Komponenten

Die einfachste physische Repräsentation einer architektonischen Grenze ist eine dynamisch verknüpfte Bibliothek wie eine .Net-DLL, eine Java-`.jar`-Datei, eine Ruby-Gem oder eine Shared Libary in UNIX. Das Deployment selbst beinhaltet keine Kompilierung, stattdessen werden die Komponenten in binärer oder einer ähnlich deploybaren Form bereitgestellt. Bei dieser Vorgehensweise handelt es sich um den Entkopplungsmodus auf Deployment-Ebene. Der Vorgang des Deployments selbst besteht einfach darin, besagte deploybaren Einheiten in einer geeigneten Weise zu sammeln, etwa in einer `.war`-Datei oder auch bloß in einem Verzeichnis.

Abgesehen von dieser einen Ausnahme sind Komponenten auf Deployment-Ebene das Gleiche wie Monolithen. Die Funktionen existieren im Allgemeinen alle im selben

Prozessor und Adressbereich, und die Strategien in Bezug auf die Trennung der Komponenten sowie die Verwaltung ihrer Abhängigkeiten sind ebenfalls identisch.[2]

18.4 Threads

Sowohl Monolithen als auch Deployment-Komponenten können sich *Threads* zunutze machen. Threads sind keine architektonischen Grenzen oder Deployment-Einheiten, sondern eher eine Möglichkeit, den Zeitplan und die Reihenfolge der Ausführung zu organisieren. Sie können vollständig in einer Komponente enthalten oder auch über viele Komponenten verteilt sein.

18.5 Lokale Prozesse

Eine viel striktere physische architektonische Grenze stellt dagegen der *lokale Prozess* dar, der normalerweise über die Befehlszeile oder einen gleichwertigen Systemaufruf erstellt wird. Lokale Prozesse werden im selben Prozessor oder in derselben Prozessorgruppe eines Mehrkernsystems, aber in separaten Adressbereichen ausgeführt. Der Speicherschutz verhindert im Allgemeinen die gemeinsame Speichernutzung durch diese Prozesse, obwohl häufig auch Shared-Memory-Partitionen verwendet werden.

In den meisten Fällen kommunizieren lokale Prozesse unter Verwendung von Sockets oder irgendeiner anderen Art von betriebssystemseitiger Kommunikationseinrichtung wie Mailboxes oder Nachrichtenwarteschlangen miteinander.

Jeder lokale Prozess kann ein statisch verknüpfter Monolith oder aus dynamisch verknüpften Deployment-Komponenten zusammengesetzt sein. Im ersten Fall können mehrere monolithische Prozesse dieselben darin kompilierten und verknüpften Komponenten enthalten. Im letzteren Fall können sie dieselben dynamisch verknüpften Deployment-Komponenten gemeinsam haben.

Sie können sich einen lokalen Prozess wie eine Art Überkomponente vorstellen: Er besteht aus untergeordneten Komponenten, die ihre Abhängigkeiten mittels dynamischer Polymorphie verwalten.

Die Strategie zur Separierung der lokalen Prozesse ist die Gleiche wie für die Monolithen und die binären Komponenten. Quellcode-Abhängigkeiten weisen in die gleiche Richtung über die Grenzlinie hinweg und immer zu der übergeordneten Komponente hin.

Für lokale Prozesse bedeutet das, dass der Quellcode der übergeordneten Prozesse weder die Namen noch die physischen Adressen oder Registrierungsschlüssel der untergeordneten Prozesse enthalten darf. Vergessen Sie nicht, dass das architektonische Ziel darin besteht, untergeordnete Prozesse als Plug-ins für übergeordnete Prozesse bereitzustellen.

Die über die Grenzen lokaler Prozesse hinausgehende Kommunikation umfasst Betriebssystemaufrufe, Daten-Marshalling und -Entschlüsselung sowie interprozedurale Kontextwechsel, die moderat kostenintensiv sind. Das »Geplapper« sollte daher sorgsam eingeschränkt werden.

18.6 Services

Die strikteste Grenze bildet ein *Service*. Hierbei handelt es sich um einen Prozess, der im Allgemeinen über die Befehlszeile oder einen gleichwertigen Systemaufruf initiiert wird. Services sind nicht von ihrem jeweiligen physischen Standort abhängig. Zwei miteinander kommunizierende Services können im selben physischen Prozessor oder in derselben Prozessorgruppe eines Mehrkernsystems betrieben werden, müssen es aber nicht. Die Services setzen voraus, dass sämtliche Kommunikation über das Netzwerk erfolgt.

Allerdings erfolgt die grenzüberschreitende Kommunikation bei den Services im Verhältnis zu den Funktionsaufrufen nur sehr schleppend: Die Bearbeitungszeiten können im Bereich von Zehntelmillisekunden bis Sekunden liegen. Insofern muss hier darauf geachtet werden, dass »Geplapper« möglichst vermieden wird, denn auf dieser Ebene muss die Kommunikation hohe Latenzzeiten bewältigen.

Ansonsten gelten für Services aber dieselben Regeln wie für lokale Prozesse. Untergeordnete Services sollten an übergeordnete Services »andocken«, und der Quellcode von übergeordneten Services darf keine spezifische physische Kenntnis von irgendeinem untergeordneten Service (z.B. von seinem URI) haben.

18.7 Fazit

In den meisten Systemen, ausgenommen monolithischen, kommt mehr als nur eine Grenzstrategie zum Einsatz. Ein System, das Service-Grenzlinien nutzt, kann auch über einige Grenzbereiche für lokale Prozesse verfügen. Tatsächlich ist ein Service oftmals nur eine Fassade für einen Satz von interagierenden lokalen Prozessen. Ein Service, oder auch ein lokaler Prozess, wird mit ziemlicher Sicherheit entweder ein aus Quellcode-Komponenten bestehender Monolith oder ein Satz dynamisch verknüpfter

Deployment-Komponenten sein.

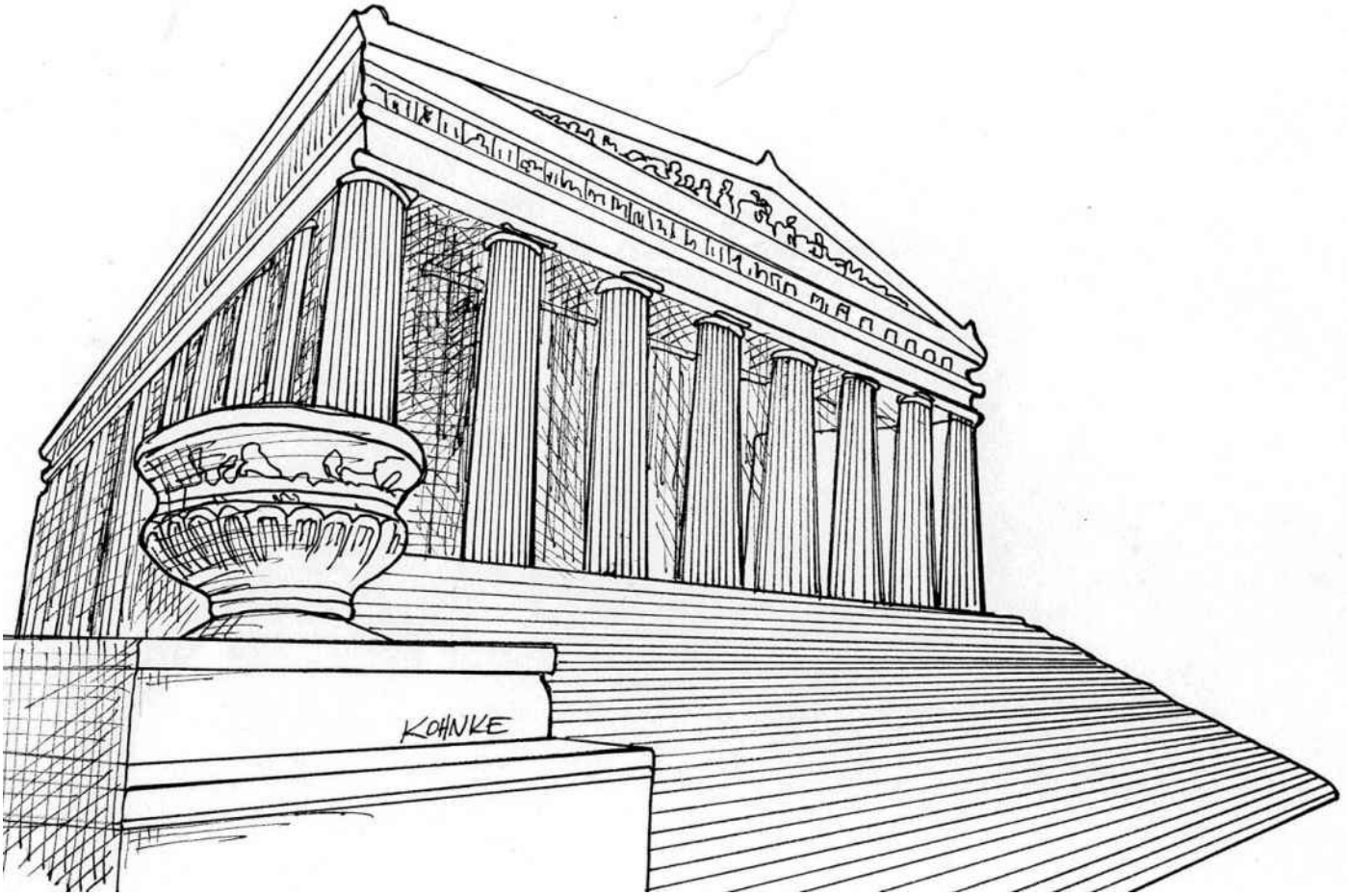
Daraus folgt auch, dass die Grenzlinien in einem System häufig eine Mischung aus »plappernden« lokalen und eher an der Latenz orientierten Grenzen darstellen.

---

[1] Statische Polymorphie (z.B. generische Typen oder Templates) können mitunter ein gangbares Mittel für die Abhängigkeitsverwaltung in monolithischen Systemen darstellen, insbesondere in Programmiersprachen wie C++. Allerdings kann die Entkopplung mittels generischer Typen nicht in der Art und Weise vor der Notwendigkeit einer Neukompilierung und eines erneuten Deployments schützen, wie es die dynamische Polymorphie kann.

[2] Wenngleich die statische Polymorphie in diesem Fall keine Option ist.

Kapitel 19
Richtlinien und Ebenen



Softwaresysteme bilden die ihnen zugrunde liegenden Richtlinien (engl. *Policy*) ab. Das ist im Kern auch schon alles, was ein Computerprogramm tatsächlich darstellt: eine detaillierte Beschreibung der Richtlinien, auf deren Grundlage Eingaben in Ausgaben umgewandelt werden.

In den meisten nichttrivialen Systemen lassen sich diese Richtlinien in viele verschiedene kleinere Anweisungen unterteilen: Einige davon beschreiben, wie bestimmte Geschäftsregeln berechnet werden, andere geben vor, wie bestimmte Berichte formatiert werden sollen, und wieder andere legen fest, wie Eingabedaten zu validieren sind.

Zum Teil besteht die Kunst der Entwicklung einer Softwarearchitektur darin, die zugrunde liegenden Richtlinien sorgfältig voneinander abzugrenzen und entsprechend der Motivationslage für deren voraussichtliche Modifikationen neu zu gruppieren: Richtlinien, die sich aus denselben Gründen und zur selben Zeit ändern, befinden sich auf derselben Ebene und gehören gemeinsam in ein und dieselbe Komponente.

Richtlinien, die sich aus unterschiedlichen Gründen und zu unterschiedlichen Zeitpunkten ändern, befinden sich dagegen auf verschiedenen Ebenen und sollten dementsprechend auch auf verschiedene Komponenten aufgeteilt werden.

Ein weiterer Aspekt der Kunst der Softwarearchitektur ist häufig außerdem die Anordnung der neu gruppierten Komponenten in einen gerichteten azyklischen Graphen. Dabei werden die Knotenpunkte von den Komponenten gebildet, die gleichrangige Richtlinien enthalten. Die gerichteten Kanten repräsentieren dagegen die Abhängigkeiten zwischen diesen Komponenten und verbinden solche Komponenten, deren Richtlinien sich auf unterschiedlichen Ebenen befinden.

Bei besagten Abhängigkeiten handelt es sich um Quellcode – es sind Abhängigkeiten, die zur Kompilierzeit bestehen. In Java werden sie durch `import`-Anweisungen dargestellt, in C# sind es `using`-Anweisungen, und in Ruby werden sie durch `require`-Anweisungen repräsentiert. All diese Abhängigkeiten sind für die Funktionsfähigkeit des Compilers erforderlich.

In einer guten Softwarearchitektur basiert die Ausrichtung der Abhängigkeiten auf der Ebene der Komponenten, die sie miteinander verbinden. In jedem Fall sind untergeordnete Komponenten so angelegt, dass sie von übergeordneten Komponenten abhängig sind.

19.1 Ebene

Eine strenge Definition des Begriffs »Ebene« lautet: »Der Abstand zwischen Eingaben und Ausgaben.« Je weiter eine Richtlinie sowohl von den Eingaben als auch den Ausgaben des Systems entfernt ist, desto höher die Ebene. Mit anderen Worten: Die Richtlinien, die Ein- und Ausgaben verwalten, bilden die tiefschichtigsten untergeordneten Ebenen des Systems.

Das Datenflussdiagramm in Abbildung 19.1 veranschaulicht ein einfaches Verschlüsselungsprogramm, das Zeichen von einem Eingabegerät ausliest, sie mithilfe einer Tabelle übersetzt und die übersetzten Zeichen dann an ein Ausgabegerät schreibt. Der Datenfluss wird hier in Form von durchgehenden gebogenen Pfeilen angezeigt, die zugehörigen Quellcode-Abhängigkeiten sind durch gerade gestrichelte Pfeile gekennzeichnet.

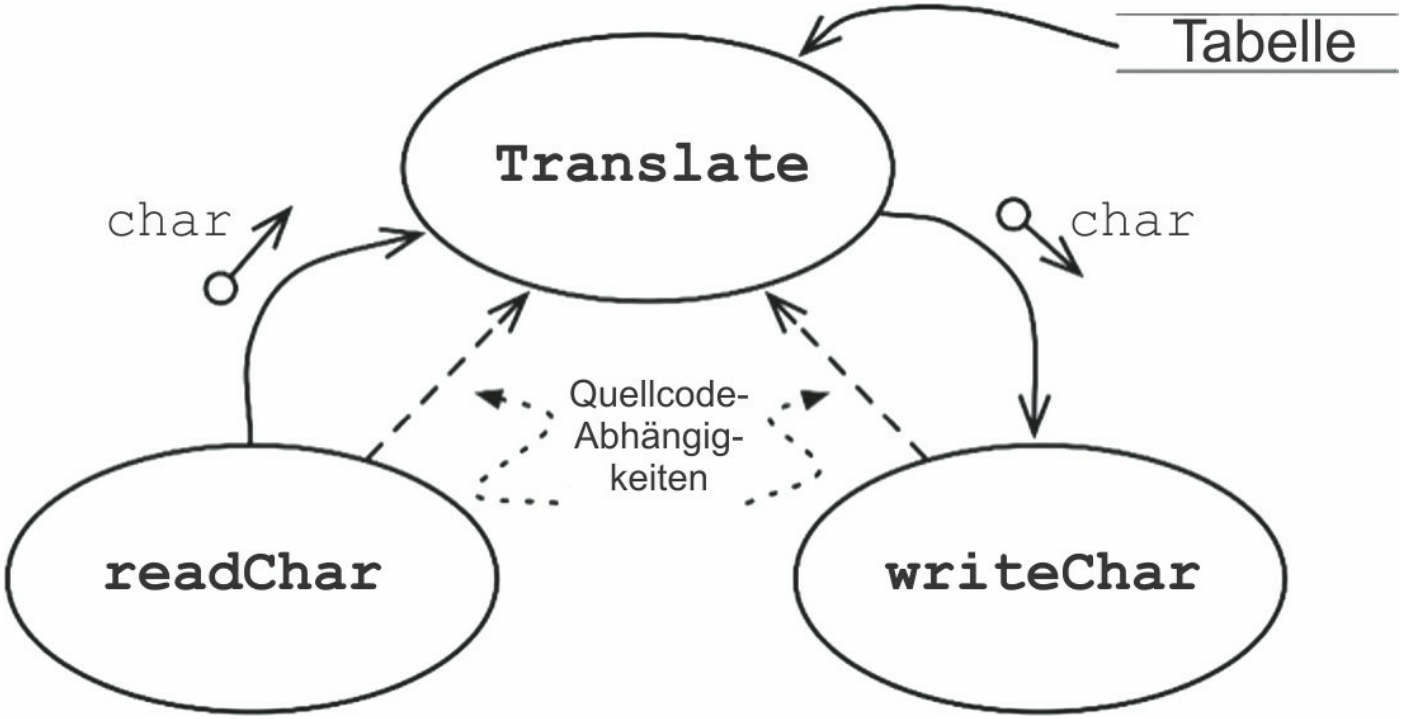


Abb. 19.1: Ein einfaches Verschlüsselungsprogramm

Die Komponente `Translate` ist in diesem System die übergeordnete Komponente, weil sie am weitesten von den Eingaben und Ausgaben entfernt ist.[1]

Beachten Sie, dass der Datenfluss und die Quellcode-Abhängigkeiten nicht immer in dieselbe Richtung weisen. Auch das ist wiederum ein Aspekt der kunstvollen Softwarearchitektur: Die Quellcode-Abhängigkeiten sollen vom Datenfluss entkoppelt und *entsprechend der Ebene gekoppelt* sein.

Ein Verschlüsselungsprogramm wie das Folgende würde dagegen eine fehlerhafte Softwarearchitektur nach sich ziehen:

```
function encrypt() {
   while(true)
      writeChar(translate(readChar()));
}
```

Und zwar deshalb, weil die übergeordnete `encrypt`-Funktion in diesem Fall von den untergeordneten Funktionen `readChar` und `writeChar` abhängig wäre.

Eine deutlich bessere Softwarearchitektur für dieses System ist dagegen in dem Klassendiagramm in Abbildung 19.2 dargestellt. Beachten Sie den gestrichelten Rahmen, der die `Encrypt`-Klasse sowie die beiden Schnittstellen `CharWriter` und `CharReader` umschließt: Alle Abhängigkeiten, die diese Grenzlinie kreuzen, weisen nach innen – und damit repräsentiert diese Einheit das hochschichtigste Element im System.

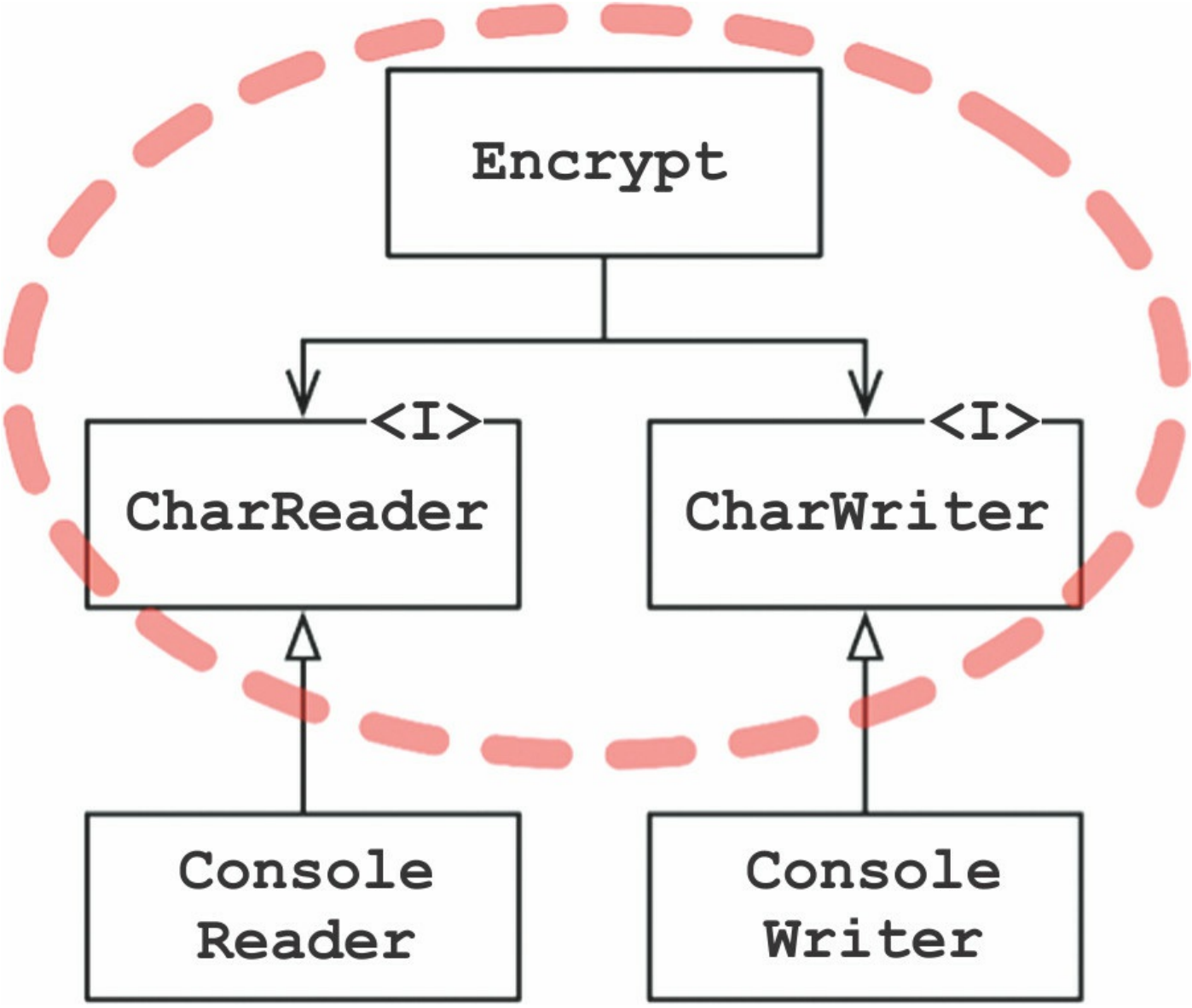


Abb. 19.2: Dieses Klassendiagramm zeigt eine bessere Softwarearchitektur für das System.

`ConsoleReader` und `ConsoleWriter` werden als Klassen ausgewiesen. Da sie sich nah an den Ein- und Ausgaben befinden, sind sie untergeordnet.

Bemerkenswert ist hier außerdem, wie diese Struktur die übergeordnete Verschlüsselungsrichtlinie von den untergeordneten Ein-/Ausgabe-Richtlinien entkoppelt – so wird die Verschlüsselungsrichtlinie für eine große kontextuelle Bandbreite nutzbar. Wenn Modifikationen an den Richtlinien für die Ein- und Ausgabe vorgenommen werden, haben sie mit großer Wahrscheinlichkeit keinerlei Auswirkungen auf die Verschlüsselungsrichtlinie.

Vergessen Sie nicht, dass Richtlinien unter Berücksichtigung ihrer potenziellen Modifikationsgrundlagen eingruppiert werden: Richtlinien, die sich aus denselben Gründen und zu denselben Zeitpunkten ändern, werden entsprechend des SRPs und des CCPs zusammengefasst. Übergeordnete Richtlinien – also diejenigen, die am weitesten von den Eingaben und Ausgaben entfernt sind – ändern sich in der Regel

weniger häufig und aus wichtigeren Gründen als untergeordnete Richtlinien. Letztere wiederum – also die Richtlinien, die sich am nächsten an den Ein- und Ausgaben befinden – ändern sich dagegen häufiger und mit einer größeren Dringlichkeit, aber aus weniger wichtigen Gründen.

So ist es selbst in dem vorerwähnten trivialen Beispiel des Verschlüsselungsprogramms sehr viel wahrscheinlicher, dass sich die I/O-Geräte ändern, als dass der Verschlüsselungsalgorithmus modifiziert wird. Und sollte der Verschlüsselungsalgorithmus doch verändert werden, dann voraussichtlich aus einem triftigeren Grund, als dies bei einem Wechsel eines der I/O-Geräte der Fall wäre.

Die Richtlinien getrennt voneinander zu halten, und zwar so, dass alle Quellcode-Abhängigkeiten in Richtung der übergeordneten Richtlinien weisen, verringert den Einfluss der vorzunehmenden Modifikationen. Triviale, aber dringende Anpassungen auf den untersten Ebenen des Systems wirken sich so nur minimal bis gar nicht auf die höheren und bedeutsameren Ebenen aus.

Eine weitere Herangehensweise an diese Problemstellung ist, die untergeordneten Komponenten als Plug-ins für die übergeordneten Komponenten bereitzustellen. Das Komponentendiagramm in Abbildung 19.3 veranschaulicht diese Anordnung: Die Komponente `Encryption` hat keine Kenntnis von der Komponente `IODevices` – die Komponente `IODevices` ist hingegen von der Komponente `Encryption` abhängig.

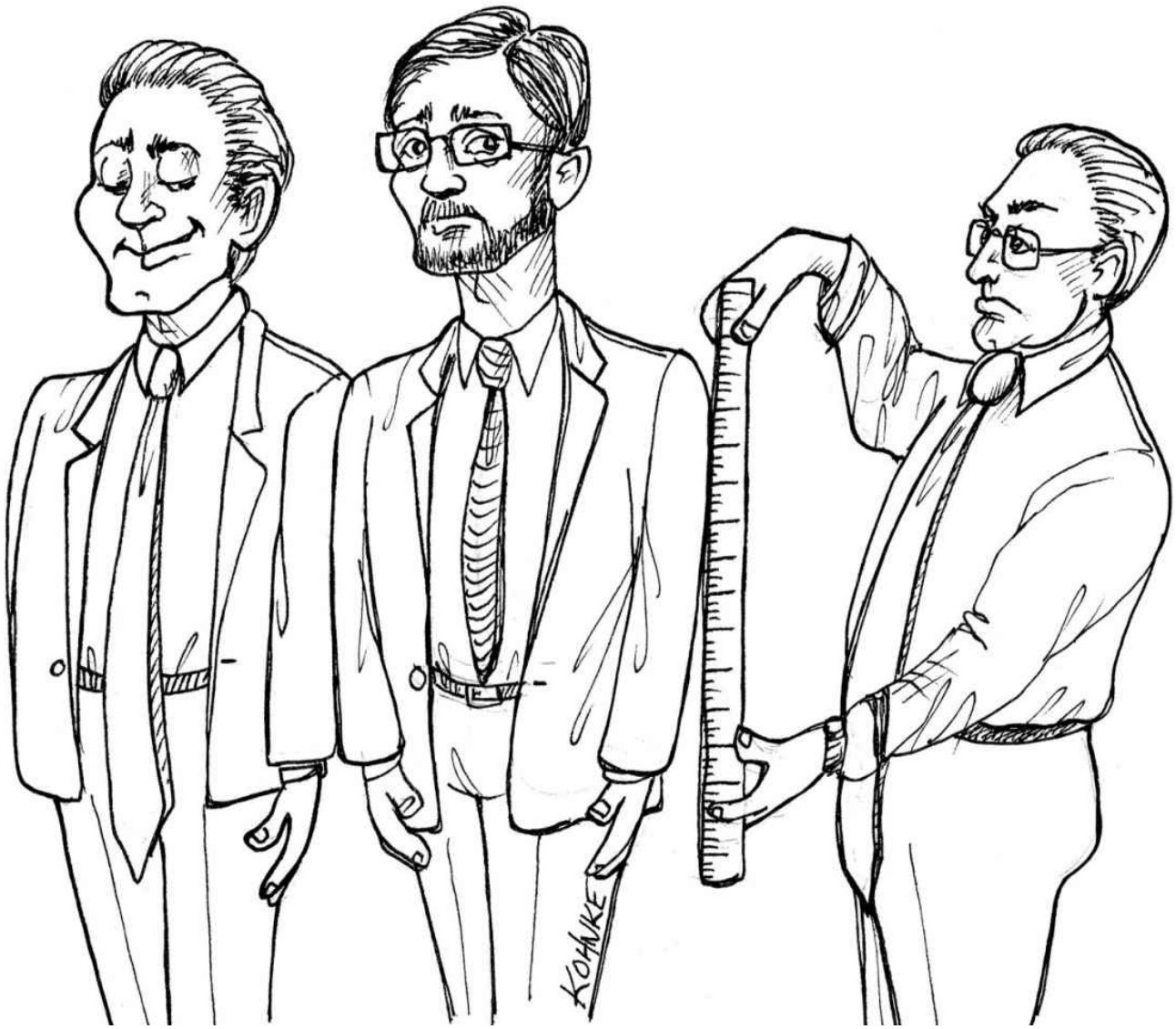
Abb. 19.3: Untergeordnete Komponenten sollten an übergeordnete Komponenten andocken.

19.2 Fazit

In dieser Diskussion der Richtlinien wurden Bestandteile des *Single-Responsibility-Prinzips*, des *Open-Closed-Prinzips*, des *Common-Closure-Prinzips*, des *Dependency-Inversion-Prinzips*, des *Stable-Dependencies-Prinzips* und des *Stable-Abstractions-Prinzips* angeführt. Blicken Sie doch noch einmal etwas genauer auf das Kapitel zurück und versuchen Sie festzustellen, wo genau jedes einzelne Prinzip Anwendung fand und warum.

[1] Meilir Page-Jones bezeichnet diese Komponente in seinem Buch *The Practical Guide to Structured Systems Design*, 2. Auflage (Yourdon Press, 1988) als »Central Transform«, zu Deutsch etwa »Zentrale Transformation«.

Kapitel 20
Geschäftsregeln



Wenn Sie Ihre Anwendung in Geschäftsregeln und Plug-ins unterteilen, dann sollten Sie natürlich auch wissen, was Geschäftsregeln eigentlich sind. Tatsächlich treten sie in mehreren verschiedenen Formen in Erscheinung.

Im Kern handelt es sich bei den Geschäftsregeln um Richtlinien oder Verfahren, die zur Erwirtschaftung oder Einsparung von Geschäftskapital führen. Streng genommen würden sie auch dann für Zugewinne und Kostenersparnisse sorgen, wenn sie nicht auf einem Computer implementiert wären. Mit anderen Worten: Selbst bei einer nicht maschinengestützten Umsetzung dieser Regeln würde das Unternehmen durch ihre Einhaltung dennoch Geld verdienen oder einsparen.

So ist beispielsweise die Vorgabe, dass eine Bank x % Zinsen für ein Darlehen berechnet, eine Geschäftsregel, die dem Geldinstitut einen Profit verschafft. Ob die Zinsen aber nun von einem Computer oder von einem Bankangestellten berechnet werden, der zu diesem Zweck einen altertümlichen Abakus benutzt, spielt dabei keine Rolle.

Vorgaben bzw. Richtlinien dieser Art werden im Folgenden als *kritische Geschäftsregeln* bezeichnet, weil sie in der Tat entscheidend für das Unternehmen selbst sind und auch dann Anwendung finden würden, wenn es kein System gäbe, mit dem sie sich automatisieren ließen.

Kritische Geschäftsregeln erfordern üblicherweise ein paar Daten, mit denen gearbeitet werden kann. Zum Beispiel werden für das erwähnte Darlehen ein Darlehensstand, ein Zinssatz und ein Zahlungsplan benötigt. Hierbei handelt es sich um die sogenannten *kritischen Geschäftsdaten*. Diese würden ebenfalls auch dann existieren, wenn das System nicht automatisiert wäre. Die kritischen Geschäftsregeln und die kritischen Daten sind untrennbar miteinander verbunden, sodass sie ideale Anwärter für ein Objekt sind – das in diesem Fall als *Entität*[1] bezeichnet wird.

20.1 Entitäten

Eine *Entität* ist ein Objekt innerhalb eines Computersystems, das einen kleinen Satz kritischer Geschäftsregeln repräsentiert, die auf kritischen Geschäftsdaten basieren. Dieses Objekt enthält entweder die kritischen Geschäftsdaten selbst oder es verfügt über schnellen Zugriff darauf. Die Schnittstelle der Entität besteht wiederum aus den Funktionen, die die mit diesen Daten operierenden kritischen Geschäftsregeln implementieren.

Als Beispiel zeigt Abbildung 20.1, wie die vorgenannte Entität `Darlehen` als Klasse in UML aussehen könnte. Sie enthält drei Bestandteile der kritischen Geschäftsdaten und präsentiert in ihrer Schnittstelle drei kritische Geschäftsregeln.

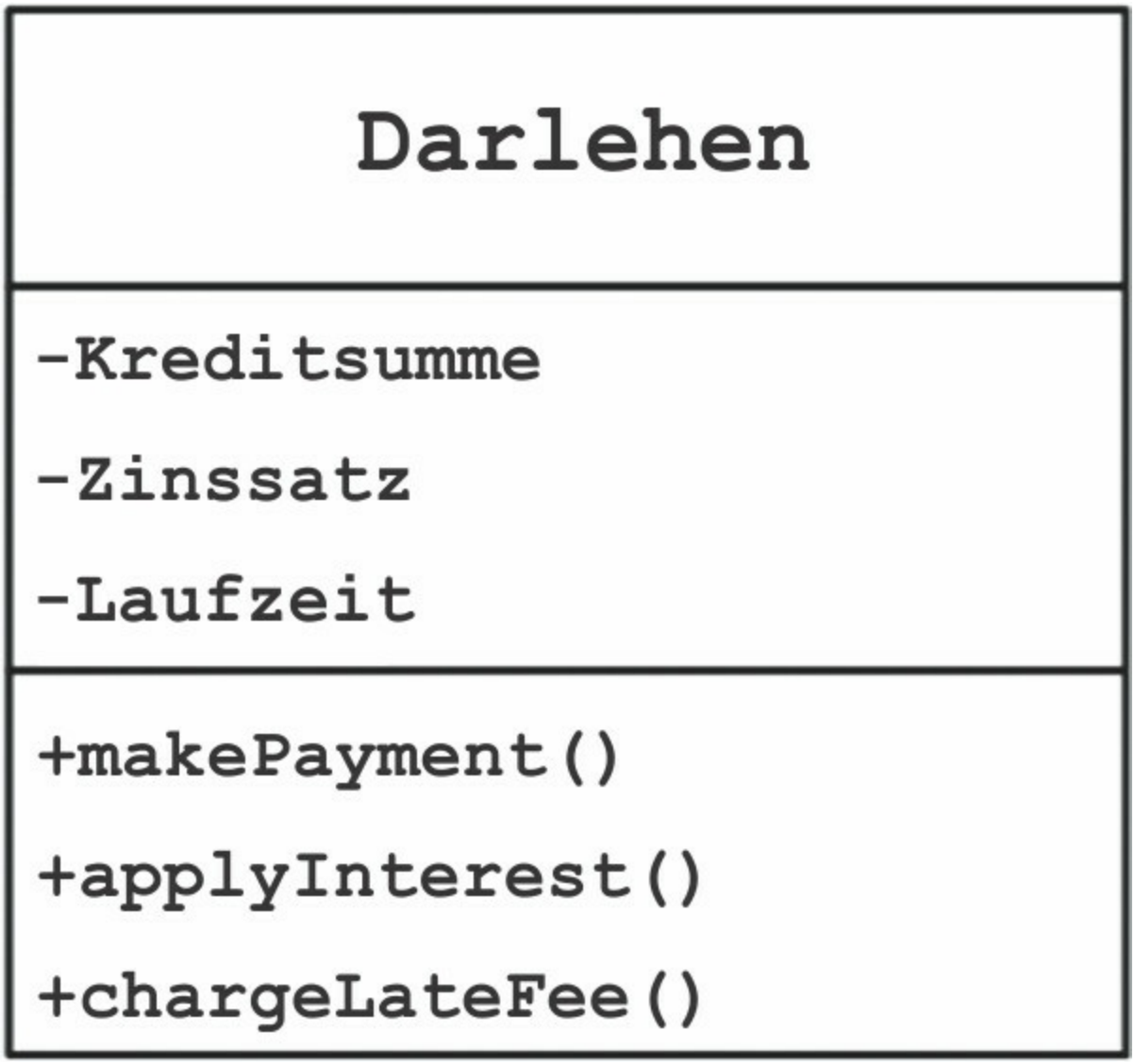


Abb. 20.1: Entität `Darlehen` als Klasse in UML

Für die Erstellung dieser Art von Klasse stellen die Programmierer die Software zusammen, die ein geschäftskritisches Konzept implementiert, und separieren sie sorgsam von allen anderen Aspekten und Belangen in dem automatisierten System, das sie gerade entwickeln. Diese Klasse ist der alleinige Repräsentant der Geschäftstätigkeit. Sie bleibt von jeder Überlegung hinsichtlich Datenbanken, Benutzerschnittstellen oder Frameworks von Drittanbietern unangetastet. Stattdessen lässt sie sich für die eigentlichen Geschäftsabläufe in jedem beliebigen System nutzen, ungeachtet des Systemaufbaus, der Datenspeicherungsmethode oder der Anordnung der Rechnerstruktur innerhalb dieses Systems. Die Entität bezieht sich einzig und allein auf die reine Geschäftstätigkeit – und nichts anderes.

Vielleicht wundern Sie sich, dass hier von einer »Klasse« die Rede ist – aber keine Sorge: Sie müssen keine objektorientierte Sprache verwenden, um eine Entität zu erstellen. Es geht ausschließlich darum, die kritischen Geschäftsdaten und die kritischen Geschäftsregeln gemeinsam in einem einzigen, separaten Softwaremodul zusammenzuführen.

20.2 Use Cases

Allerdings sind nicht alle Geschäftsregeln so puristisch wie Entitäten. Einige von ihnen sorgen dafür, dass das Unternehmen Gewinne erwirtschaftet oder Kapital einspart, indem sie die Art und Weise definieren und einschränken, in der ein *automatisiertes* System arbeitet. Diese Regeln würden in einer nicht automatisierten Umgebung natürlich keine Anwendung finden, weil sie lediglich als Bestandteil eines automatisierten Systems Sinn machen.

Stellen Sie sich beispielsweise eine Anwendung vor, die von den Bankangestellten zum Anlegen eines neuen Darlehensvertrags genutzt wird. Die Bank könnte entscheiden, dass die Kreditsachbearbeiter keine Ratenpläne erstellen können sollen, ohne zuvor die Kontaktdaten des Antragstellers einzupflegen und zu validieren sowie sicherzustellen, dass die Kreditwürdigkeit des potenziellen Darlehensnehmers bei 500 Punkten oder höher liegt. So könnte die Bank vorgeben, dass das System den Ratenplan erst öffnet, *nachdem* das Kontaktdatenformular vollständig ausgefüllt wurde, die Daten den Verifizierungsvorgang durchlaufen haben und die Kreditwürdigkeit als über die Mindestbonität hinausgehend bestätigt ist.

Hierbei handelt es sich um einen *Use Case*[2]. Ein solcher »Anwendungsfall« repräsentiert eine Beschreibung der Art und Weise, wie ein automatisiertes System verwendet wird. Er spezifiziert die von dem User vorzunehmende Eingabe, die an den User zurückzugebende Ausgabe sowie die Verarbeitungsschritte, die für die Erzeugung dieser Ausgabe erforderlich sind. Im Gegensatz zu den kritischen Geschäftsregeln innerhalb der Entitäten beschreibt ein Use Case die *anwendungsspezifischen* Geschäftsregeln.

Abbildung 20.2 zeigt ein Beispiel für solch einen Use Case. Beachten Sie hier, dass in der letzten Zeile der Kunde als solcher erwähnt wird. Diese Referenz bezieht sich auf die Entität `Kunde`, die die kritischen Geschäftsregeln enthält, die die Beziehung zwischen der Bank und ihren Kunden regulieren.

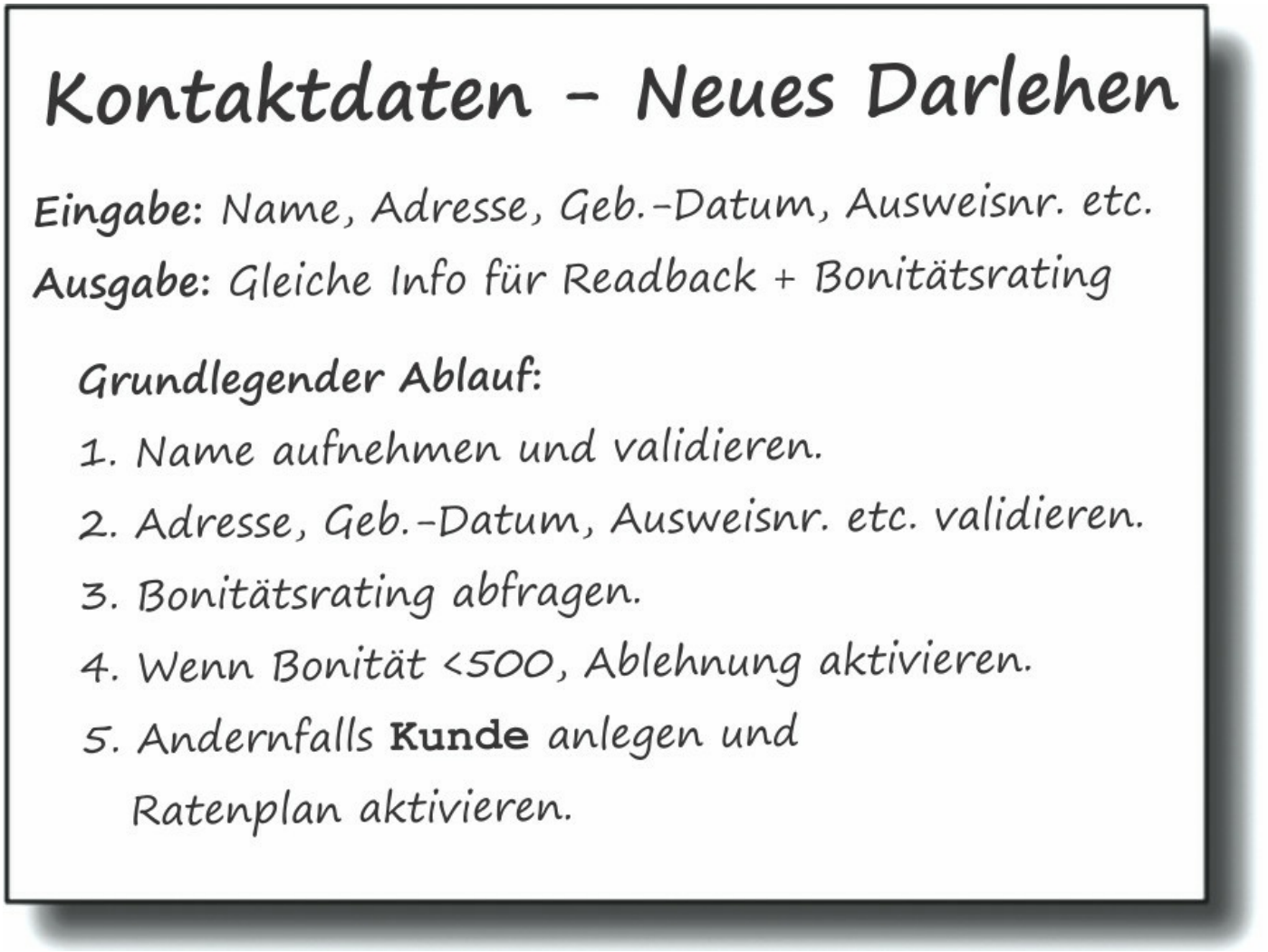


Abb. 20.2: Use-Case-Beispiel

Use Cases enthalten die Richtlinien, die spezifizieren, wie und wann die kritischen Geschäftsregeln innerhalb der Entitäten ausgelöst werden. Sie steuern somit den »Tanz der Entitäten«.

Beachten Sie auch, dass der Use Case die Benutzerschnittstelle nur insoweit beschreibt, als er informell die Daten spezifiziert, die von dieser Schnittstelle kommen, sowie die Daten, die wieder durch dieselbe Schnittstelle hinausgehen. Es ist nicht möglich, anhand des Use Case festzustellen, ob die Anwendung im Web, auf einem Fat Client, einer Konsole oder als reiner Service bereitgestellt wird.

Das ist ein sehr wichtiger Punkt. Use Cases beschreiben nicht, wie sich das System für den User darstellt, sondern weisen lediglich die anwendungsspezifischen Regeln zur Regulierung der Interaktion zwischen den Usern und den Entitäten aus. Wie die Dateien in das System hinein- und wieder herausgelangen, ist für die Use Cases irrelevant.

Ein Use Case ist ein Objekt, das eine oder mehrere Funktionen besitzt, die die anwendungsspezifischen Geschäftsregeln implementieren. Außerdem verfügt es über Datenelemente, zu denen auch die Eingabedaten, die Ausgabedaten sowie die

Referenzen auf die zugehörigen Entitäten gehören, mit denen es interagiert.

Entitäten haben keine Kenntnis davon, dass sie von den Use Cases gesteuert werden. Hierbei handelt es sich um ein weiteres Beispiel für die Ausrichtung der Abhängigkeiten gemäß dem *Dependency-Inversion-Prinzip*: Übergeordnete Konzepte, wie zum Beispiel Entitäten, besitzen keine Kenntnis von untergeordneten Konzepten, wie etwa den Use Cases. Umgekehrt haben die untergeordneten Use Cases jedoch Kenntnis von den übergeordneten Entitäten.

Warum aber sind Entitäten übergeordnet und Use Cases untergeordnet? Weil die Use Cases spezifisch auf eine einzelne Anwendung ausgelegt sind und sich daher näher an den Ein- und Ausgaben des Systems befinden. Entitäten hingegen sind allgemeingültig und können in vielen verschiedenen Anwendungen zum Einsatz kommen. Insofern sind sie auch weiter von den Ein- und Ausgaben des Systems entfernt. Use Cases sind von Entitäten abhängig – umgekehrt sind Entitäten jedoch nicht von Use Cases abhängig.

20.3 Request-and-Response-Modelle

Use Cases erwarten Eingabedaten und erzeugen Ausgabedaten. Ein gut gestaltetes Use-Case-Objekt sollte allerdings nicht den Hauch einer Ahnung davon haben, wie diese Daten dem User oder einer anderen Komponente übermittelt werden. Und natürlich sollte der Code innerhalb der Use-Case-Klasse auch definitiv keine Kenntnis von HTML oder SQL haben!

Die Use-Case-Klasse nimmt einfache Abfragedatenstrukturen als Eingaben entgegen und gibt einfache Antwortdatenstrukturen wieder aus. Diese Datenstrukturen sind vollkommen unabhängig und werden nicht von Standard-Framework-Schnittstellen wie `HttpRequest` und `HttpResponse` abgeleitet. Sie haben weder Kenntnis vom Web, noch stehen sie in irgendeinem Zusammenhang mit dem, was die Benutzeroberfläche bereitstellen mag.

Dieses völlige Fehlen von Abhängigkeiten ist ein ausschlaggebender Faktor, denn: Wenn die Request-and-Response-Modelle nicht unabhängig sind, dann werden die Use Cases, die wiederum von ihnen abhängig sind, indirekt an alle Abhängigkeiten gebunden, die die Modelle mit sich bringen.

Nun könnten Sie versucht sein, die Datenstrukturen mit Referenzen auf die Entitätsobjekte zu füllen. Vielleicht halten Sie dies ja für sinnvoll, weil die Entitäten und die Request-and-Response-Modelle so viele Daten teilen. Dieser Versuchung sollten Sie jedoch widerstehen, denn die beiden Objekte erfüllen sehr unterschiedliche Zwecke! Sie werden sich im Laufe der Zeit aus sehr unterschiedlichen Gründen

ändern, sodass es unweigerlich einen Verstoß gegen das *Common-Closure*- und das *Single-Responsibility-Prinzip* bedeuten würde, sie in irgendeiner Weise miteinander zu verknüpfen. Dies hätte jede Menge Tramp-Daten – also solche, die ziellos im System umherwandern – sowie zahllose bedingte Verzweigungen in Ihrem Code zur Folge.

20.4 Fazit

Im Wesentlichen sind die Geschäftsregeln der eigentliche Grund für die Existenz eines Softwaresystems. Sie bilden den Kern der Funktionalität und enthalten den Code, der Gewinne erwirtschaftet oder Einsparungen fördert. Im übertragenen Sinne sind sie sozusagen die Familienjuwelen.

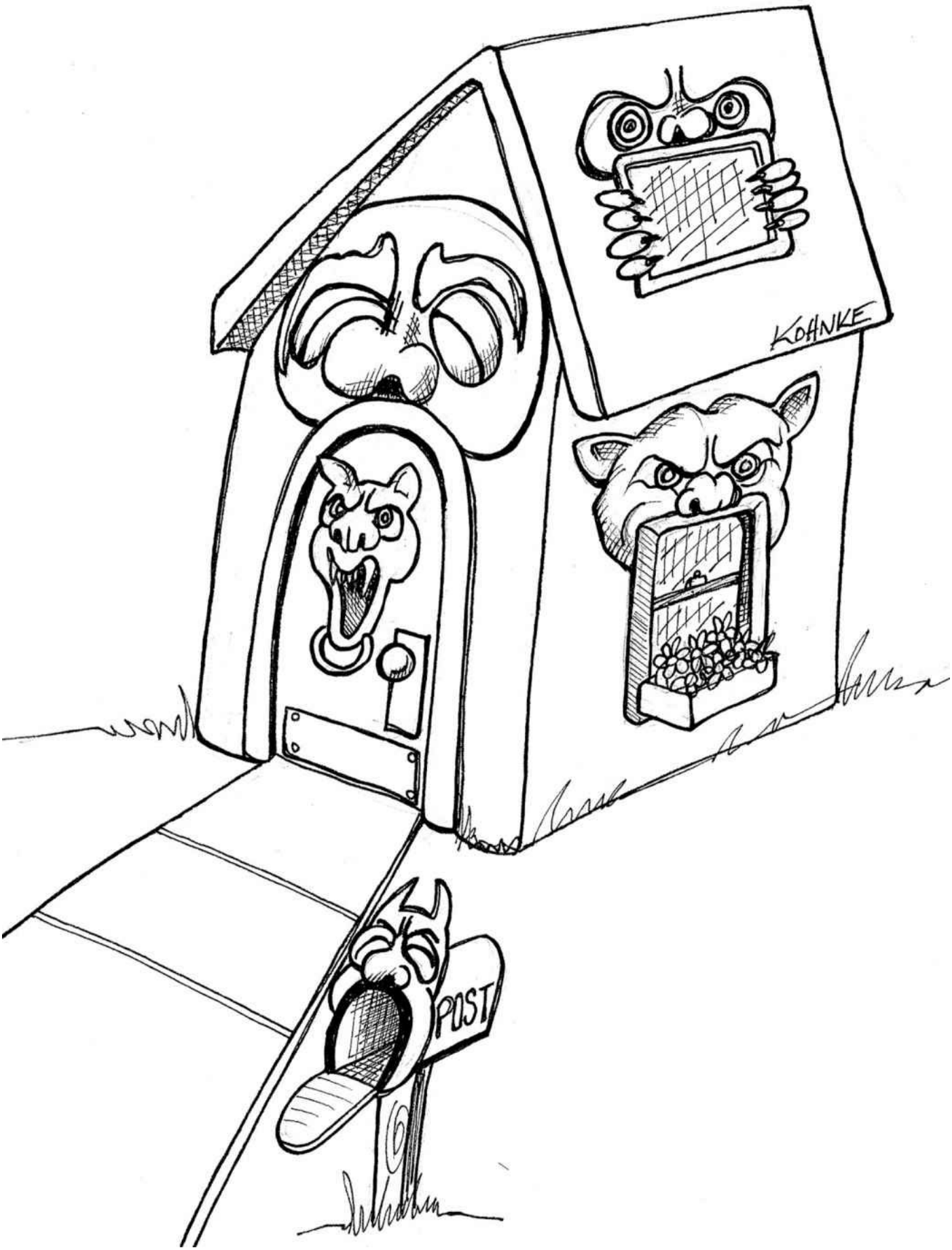
Die Geschäftsregeln sollten unangetastet bleiben und nicht durch grundlegende Aspekte wie die Benutzerschnittstelle oder die verwendete Datenbank beeinträchtigt werden. Im Idealfall sollte der Code, der sie repräsentiert, das Herzstück des Systems darstellen, wo sie nur von sehr wenigen Faktoren tangiert werden. Außerdem sollten die Geschäftsregeln der unabhängigste und wiederverwendbarste Code im System sein.

---

[1] Der Name dieses Konzepts geht auf Ivar Jacobsen zurück (*Object Oriented Software Engineering: A Use Case Driven Approach*, Ivar Jacobson et al, Addison-Wesley Professional, 1992).

[2] Ebd.

Kapitel 21
Die schreiende Softwarearchitektur



Stellen Sie sich einmal vor, Sie betrachteten die Baupläne eines Gebäudes. Diese Blaupausen, von einem Architekten angefertigt, zeigen den Entwurf des zukünftigen

Bauwerks. Was sagen Ihnen diese Baupläne?

Wenn es sich um den Grundriss eines Einfamilienhauses handelt, dann werden Sie vermutlich einen Hauseingang entdecken, einen Eingangsbereich, der in ein Wohnzimmer führt, und vielleicht auch ein Esszimmer. Nicht weit davon entfernt sollte eine Küche zu sehen sein, normalerweise in der Nähe des Esszimmers. Möglicherweise gibt es auch nur eine Essecke und ggf. einen angrenzenden Familienraum. Bei der Betrachtung eines solchen Planungsentwurfs steht sofort außer Frage, dass es sich um ein Einfamilienhaus handelt. Der Architekturplan schreit förmlich: »HEIM«.

Nun unterstellen wir einmal, Sie würden sich stattdessen die Architekturpläne einer Bibliothek anschauen. Sie würden aller Wahrscheinlichkeit nach eine große Eingangshalle vorfinden, einen Servicebereich, wo die Leihbücher ausgehändigt und wieder abgegeben werden können, Lesebereiche, kleine Konferenzräume und eine Galerie mit jeder Menge aneinandergereihten Bücherregalen. Dieser Architekturplan schreit also förmlich: »BIBLIOTHEK«.

Und was »schreit« der Architekturplan Ihrer Anwendung? Wenn Sie die übergeordnete Verzeichnisstruktur betrachten und die Quelldateien im Package der höchsten Ebene, schreien diese dann »Krankenhausinformationssystem« oder »Buchhaltungssystem« oder »Lagerhaltungssystem«? Oder bloß »Rails« oder »Spring/Hibernate« oder »ASP«?

21.1 Das Thema einer Architektur

Rufen Sie sich Ivar Jacobsons bahnbrechendes Werk über die Softwarearchitektur in Erinnerung (oder lesen Sie es ggf. noch einmal): *Object-Oriented Software Engineering*[1]. Besonders bemerkenswert ist der Untertitel dieses Buches: *A Use Case Driven Approach* – ein Use-Case-getriebener Ansatz. Jacobson macht hier sehr deutlich, dass Softwarearchitekturen vor allem Strukturen sind, die die Use Cases des Systems unterstützen. So, wie die Baupläne eines Hauses oder einer Bibliothek die Use Cases ebendieser Gebäude hinausschreien, so sollte auch die Architektur einer Softwareanwendung die Use Cases der Architektur laut und deutlich kundtun.

Bei Softwarearchitekturen geht es nicht um Frameworks (zumindest sollte es das nicht), und sie sollten auch nicht von Frameworks bereitgestellt werden. Frameworks sind Tools, die es zu nutzen gilt, aber keine Architekturen, an die sich angepasst werden muss. Wenn Ihre Softwarearchitektur auf Frameworks basiert, dann kann sie nicht auf Ihren Use Cases basieren.

21.2 Der Zweck einer Softwarearchitektur

Gute Softwarearchitekturen konzentrieren sich auf die Use Cases, damit die Softwarearchitekten alle Strukturen, die diese Use Cases unterstützen, zielsicher beschreiben können, ohne sich auf Frameworks, Tools und Umgebungen festlegen zu müssen. Denken Sie auch hierbei wieder an das Beispiel der Baupläne für ein Gebäude: Die erste Priorität des Architekten besteht darin, sicherzustellen, dass das Haus zweckdienlich nutzbar ist – und nicht, dass es aus Ziegelsteinen besteht. Tatsächlich ist der Architekt sogar sehr bemüht, seine Entwürfe in der Form zu gestalten, dass der Hausbesitzer später selbst entscheiden kann, welche Materialien er für die Außenfassade verwenden will (Klinker, Stein oder Zedernholz), *nachdem* die Planung gewährleistet, dass den jeweiligen Use Cases entsprochen wird.

Eine gute Softwarearchitektur ermöglicht es, Entscheidungen über Frameworks, Datenbanken, Webserver und andere umgebungsbezogene Aspekte und Tools hinauszuzögern und aufzuschieben. *Frameworks sind Optionen, die offen bleiben müssen.* Eine gute Softwarearchitektur macht es erst sehr viel später in einem Projekt erforderlich, sich für Rails, Spring, Hibernate, Tomcat oder MySQL zu entscheiden. Und sie lässt Ihnen die Möglichkeit, Ihre Meinung hinsichtlich dieser Entscheidungen problemlos wieder zu ändern. Mit anderen Worten: Eine gute Softwarearchitektur stellt die Use Cases in den Mittelpunkt und entkoppelt sie von allen peripheren Aspekten.

21.3 Aber was ist mit dem Web?

Ist das Web eine Architektur? Diktiert der Umstand, dass Ihr System ggf. im Web bereitgestellt wird, auch die Architektur Ihres Systems? Natürlich nicht! Tatsächlich ist das Web lediglich ein Bereitstellungsmechanismus, ein I/O-Gerät (*Input/Output*, Ein-/Ausgabe) – und genau so sollte die Architektur Ihrer Anwendung es auch behandeln. Der Umstand, dass die Anwendung im Web bereitgestellt wird, ist lediglich ein Detail, das Ihre Systemstruktur jedoch keinesfalls dominieren darf. Im Gegenteil: Die Entscheidung, ob Ihre Anwendung über das Web verfügbar gemacht wird oder nicht, sollte sich ohne Schwierigkeiten aufschieben lassen. Für Ihre Systemarchitektur sollte die Frage, wie sie letztendlich bereitgestellt wird, so irrelevant wie möglich sein. Stattdessen sollten Sie in der Lage sein, sie ohne unnötige Komplikationen oder Modifikationen an der grundlegenden Architektur ganz nach Wunsch als Konsolen-App, als Web-App, als Fat-Client-App oder auch als Webservice-App anzubieten.

21.4 Frameworks sind Tools, keine

Lebenseinstellung

Frameworks können sehr mächtig und nützlich sein – und ihre Schöpfer sind oftmals in einem tiefen, fast schon religiös anmutenden Glauben an ihre Werke verhaftet. Die Beispiele, die sie für die Verwendung ihrer Frameworks liefern, sind aus der Sicht echter »Framework-Jünger« geschrieben. Auch andere Autoren, die über Frameworks schreiben, neigen dazu, glühende Anhänger dieses Konstrukts zu sein. Sie weisen Ihnen den Weg zum Einsatz des Frameworks als solches – und vermitteln dabei nicht selten eine geradezu omnipräsente und über die Maßen vordringliche »Überlassen Sie einfach alles dem Framework«-Haltung.

Doch das ist nicht die Einstellung, die Sie dazu haben sollten.

Betrachten Sie jedes Framework aus einer unvoreingenommenen, neutralen Perspektive. Seien Sie skeptisch. Ja, es könnte durchaus nützlich sein – aber zu welchem Preis? Fragen Sie sich, wie Sie es einsetzen würden und wie Sie sich selbst dagegen schützen könnten. Denken Sie darüber nach, wie Sie den Schwerpunkt Ihrer Softwarearchitektur auf den Use Cases belassen können. Entwickeln Sie eine Strategie, die verhindert, dass das Framework Ihre Architektur überlagert und am Ende möglicherweise sogar dominiert.

21.5 Testfähige Architekturen

Wenn sich Ihre Systemarchitektur voll und ganz auf die Use Cases konzentriert und Sie Ihre Frameworks auf einer gesunden Distanz halten, dann sollte es möglich sein, Unit-Tests ohne jegliche Frameworks für all diese Anwendungsfälle durchzuführen. Für die Ausführung Ihrer Tests sollten Sie keinen Webserver benötigen. Auch eine Datenbank sollten Sie hierfür nicht brauchen. Ihre Entitätsobjekte sollten einfache alte Objekte sein, die keine Abhängigkeiten von Frameworks, Datenbanken oder anderen Erschwernissen aufweisen. Ihre Use-Case-Objekte sollten Ihre Entitätsobjekte koordinieren. Und schließlich sollte alles zusammen unmittelbar testfähig sein, ohne die durch Frameworks bedingten Komplikationen.

21.6 Fazit

Ihre Softwarearchitektur sollte den Betrachtern etwas über das System selbst preisgeben – und nicht über die Frameworks, die Sie in Ihrem System verwenden. Wenn Sie ein Krankenhausinformationssystem (kurz KIS) entwickeln, dann sollten neue Programmierer gleich bei einem ersten Blick auf die Versionsverwaltung

feststellen können: »Oh, das ist ja ein KIS.« Sie sollten außerdem in der Lage sein, ohne jegliche Kenntnis der Bereitstellungsform des Systems sämtliche Use Cases des Systems zu überblicken. So könnten sie beispielsweise anmerken:

»Nun, wir sehen hier zwar einige Dinge, die wie Modelle aussehen – aber wo sind die Views und Controller?«

Und dann sollten Sie erwidern können:

»Ach, das sind Details, mit denen wir uns im Moment noch gar nicht befassen müssen. Darüber entscheiden wir später.«

---

[1] *Object-Oriented Software Engineering: A Use Case Driven Approach*, Ivar Jacobson et al., Addison-Wesley Professional, 1992.

Kapitel 22
Die saubere Architektur

In den letzten Jahrzehnten kamen eine ganze Reihe von Ideen bezüglich der Architektur von Softwaresystemen auf, darunter auch Folgende:

- **Hexagonale Architektur** (später auch als »Ports and Adapters« bezeichnet). Entwickelt von Alistair Cockburn und von Steve Freeman und Nat Pryce in ihrem wundervollen Buch *Growing Object-Oriented Software with Tests*[1] adaptiert.
- **DCI** (**D**ata **C**ontext **I**nteraction). Entwickelt von James Coplien und Trygve Reenskaug.
- **BCE** (**B**oundary **C**ontrol **E**ntity). Vorgestellt von Ivar Jacobson in seinem Buch *Object-Oriented Software Engineering: A Use Case Driven Approach*.[2]

Auch wenn sich alle diese Architekturvarianten im Detail ein wenig unterscheiden, so

sind sie sich doch insgesamt sehr ähnlich. Ebenso verfolgen sie dieselbe Zielsetzung: die Separierung unterschiedlicher Systemaspekte. Und sie alle erreichen dies durch die Unterteilung der Software in Layer. In jedem Fall existiert mindestens ein Layer für Geschäftsregeln sowie je ein weiterer Layer für die Benutzer- und Systemschnittstellen. Jede dieser Softwarearchitekturen führt letztendlich zu Systemen, die folgende Charakteristika aufweisen:

- *Framework-unabhängig*. Die Architektur ist nicht von der Existenz irgendeiner Library mit funktionsbeladener Software abhängig. Dadurch steht es Ihnen frei, solche Frameworks als Tools einzusetzen, ohne dass Sie gezwungen sind, Ihr System an deren eingeschränkte Möglichkeiten anzupassen.

- *Testfähig*. Die Geschäftsregeln können ohne Benutzeroberfläche, Datenbank, Webserver oder sonstige externe Elemente getestet werden.

- *UI-unabhängig*. Die Benutzeroberfläche kann problemlos modifiziert werden, ohne dass dadurch Anpassungen am übrigen System notwendig werden. So ließe sich beispielsweise ein Web-UI ohne jegliche Veränderung der Geschäftsregeln durch ein Konsolen-UI ersetzen.

- *Datenbankunabhängig*. Oracle oder SQL Server lassen sich durch Mongo, BigTable, CouchDB oder eine beliebige andere Datenbank austauschen. Ihre Geschäftsregeln sind nicht an die Datenbank gebunden.

- *Unabhängig von jeglichen sonstigen externen Komponenten*. Tatsächlich haben Ihre Geschäftsregeln keinerlei Kenntnis von den Schnittstellen zur Außenwelt.

Die Grafik in Abbildung 22.1 illustriert den Versuch, all diese Softwarearchitekturen als ein einziges umsetzbares Konzept zu integrieren.

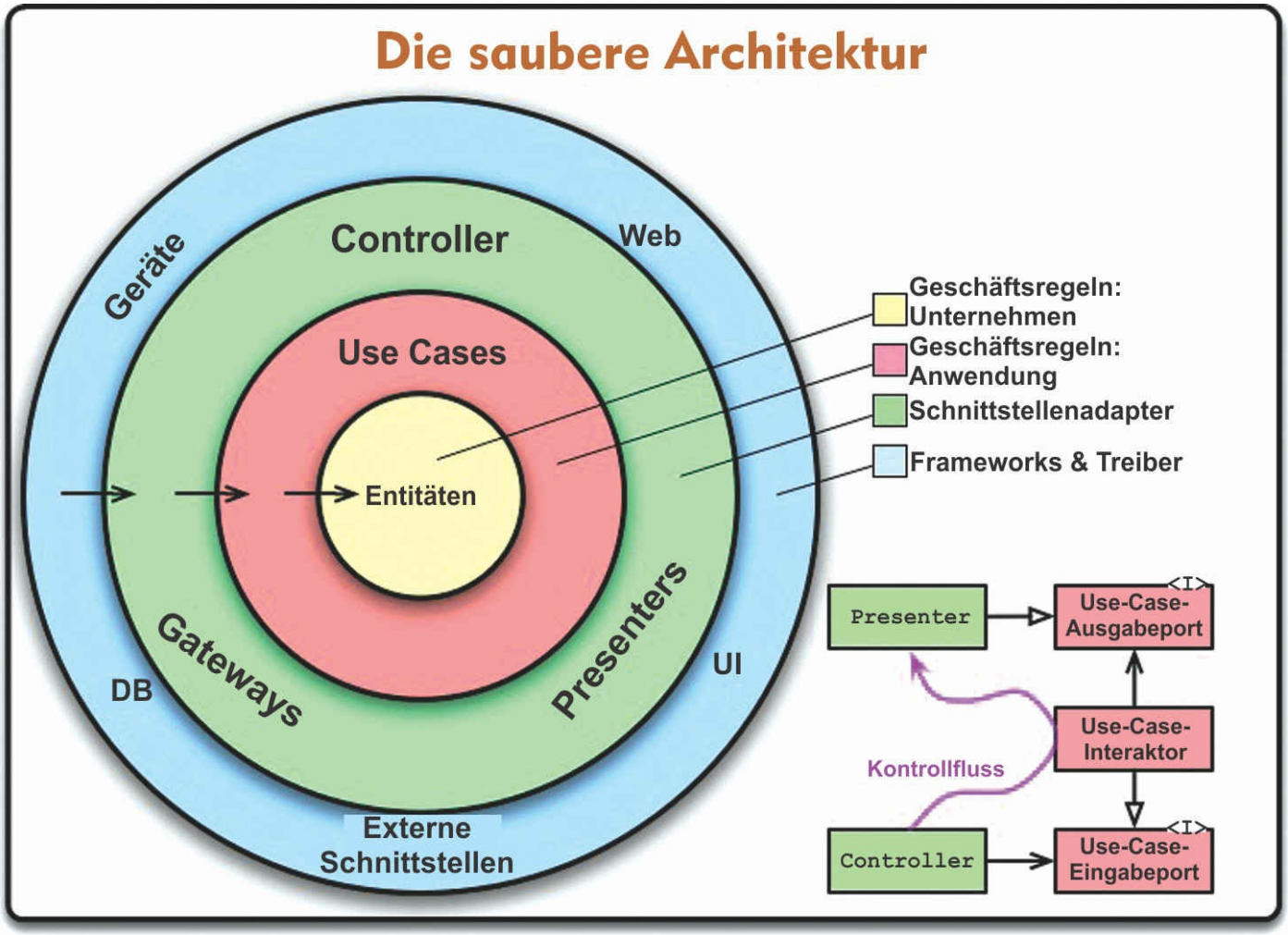


Abb. 22.1: Die saubere Architektur

22.1 Die Abhängigkeitsregel (Dependency Rule)

Die konzentrischen Kreise in Abbildung 22.1 repräsentieren verschiedene Bereiche der Software. Üblicherweise steigt die Softwareebene von außen nach innen an. Die äußeren Kreise beherbergen Mechanismen, bei den inneren Kreisen handelt es sich um Richtlinien.

Die übergeordnete Regel, die diese Architektur funktionieren lässt, ist die *Abhängigkeitsregel*:

> *Quellcode-Abhängigkeiten dürfen nur nach innen in Richtung der übergeordneten Richtlinien weisen.*

Keine Komponente in einem inneren Kreis darf jedwede Kenntnis von irgendeiner Komponente in einem der äußeren Kreise haben. Insbesondere darf der Name einer Komponente, die in einem äußeren Kreis deklariert ist, nicht im Code einer Komponente in einem der innen liegenden Kreise erwähnt werden. Das schließt auch Funktionen, Klassen, Variablen sowie alle anderen benannten Softwareentitäten mit

ein.

Aus demselben Grund sollten Datenformate, die in einem äußeren Kreis deklariert sind, ebenfalls nicht von einer Komponente in einem inneren Kreis verwendet werden, insbesondere dann nicht, wenn diese Formate von einem Framework in einem äußeren Kreis generiert werden. In den äußeren Kreisen ist somit schlichtweg alles, was die inneren Kreise beeinflussen könnte, unerwünscht.

22.1.1 Entitäten

Entitäten kapseln *unternehmensweite* kritische Geschäftsregeln. Eine Entität kann ein Objekt mit Methoden sein oder auch ein Satz von Datenstrukturen und Funktionen. Solange die Entitäten von vielen verschiedenen Anwendungen innerhalb des Unternehmens verwendet werden können, spielt das keine Rolle.

Wenn Sie kein Unternehmen haben und einfach eine Einzelanwendung schreiben, dann sind diese Entitäten die Geschäftsobjekte der Anwendung. Sie kapseln die allgemeingültigen und hochschichtigsten Regeln. Die Wahrscheinlichkeit, dass sie sich aufgrund externer Modifikationen ändern, ist hier am geringsten. Beispielsweise steht nicht zu erwarten, dass diese Objekte von einer Modifikation an der Seitennavigation oder Systemsicherheit betroffen wären. Keine betriebsbezogene Modifikation an einer bestimmten Anwendung sollte Auswirkungen auf den Entitäten-Layer haben.

22.1.2 Use Cases

Die Software im *Use-Cases-Layer* enthält *anwendungsspezifische* Geschäftsregeln. Hier werden alle Use Cases des Systems gekapselt und implementiert. Diese Use Cases verwalten den Datenfluss zu und von den Entitäten und weisen sie an, ihre kritischen Geschäftsregeln anzuwenden, um die Zielsetzungen des jeweiligen Use Case zu erfüllen.

Es steht nicht zu erwarten, dass sich Modifikationen innerhalb dieses Layers auf die Entitäten auswirken. Ebenso wenig steht zu erwarten, dass er von Modifikationen an externen Komponenten wie der Datenbank, der Benutzeroberfläche oder irgendeinem der üblichen Frameworks beeinflusst wird. Der Use-Cases-Layer ist von derartigen Faktoren isoliert.

Andererseits ist allerdings durchaus zu erwarten, dass betriebsbezogene Modifikationen an der Anwendung Einfluss auf die Use Cases haben – und damit ebenfalls auf die Software in diesem Layer. Sobald sich die Details eines Use Case ändern, ist definitiv auch Code innerhalb dieses Layers davon betroffen.

22.1.3 Schnittstellenadapter

Bei der im Layer für die *Schnittstellenadapter* enthaltenen Software handelt es sich um einen Satz von Adaptern, die die Daten aus dem für die Use Cases und Entitäten geeigneten Format in das jeweils passende Format für diverse externe Komponenten wie die Datenbank oder das Web konvertieren. In diesem Layer ist beispielsweise die komplette MVC(**M**odel **V**iew **C**ontroller)-Architektur eines GUI untergebracht. Die Presenters, Views und Controller gehören allesamt in den Layer für die Schnittstellenadapter. Die zugehörigen Modelle sind in der Regel einfach nur Datenstrukturen, die von den Controllern an die Use Cases weitergeleitet und dann von den Use Cases an die Presenters und Views zurückgegeben werden.

In ähnlicher Weise werden die Daten in diesem Layer außerdem von der für Entitäten und Use Cases geeigneten Form in diejenige konvertiert, die am besten für jedes beliebige verwendete Persistenz-Framework (z.B. die Datenbank) passt. Kein Code innerhalb dieses Kreises sollte Kenntnis von der Datenbank haben. Wenn es sich dabei z.B. um eine SQL-Datenbank handelt, dann sollte also sämtlicher SQL-Code auf diesen Layer beschränkt sein – und insbesondere auf die darin enthaltenen Bereiche, die mit der Datenbank zu tun haben.

22.1.4 Frameworks und Treiber

Der in dem Modell in Abbildung 22.1 am weitesten außen liegende Layer setzt sich im Allgemeinen aus Frameworks und Tools wie der Datenbank und dem Web-Framework zusammen. Grundsätzlich wird nicht allzu viel Code in diesen Layer-Kreis geschrieben, abgesehen von dem »Glue Code«, der die Kommunikation mit dem nächsten innenliegenden Kreis abwickelt.

Der Layer für die *Frameworks und Treiber* ist der Ort, an dem alle Details Platz finden. So ist beispielsweise das Web ein Detail, ebenso wie die Datenbank. All diese Faktoren werden außen vorgehalten, wo sie den geringsten Schaden anrichten können.

22.1.5 Nur vier Kreise?

Die in Abbildung 22.1 dargestellten Kreise geben lediglich eine schematische Darstellung wieder – eventuell werden Sie für Ihr System aber auch mehr als nur diese vier Kreise benötigen. Es gibt keine Regel, die besagt, dass Sie sich immer nur auf diese vier Kreise beschränken müssen – die Abhängigkeitsregel findet allerdings in jedem Fall Anwendung. Quellcode-Abhängigkeiten müssen immer nach innen weisen. Je weiter Sie in das Innere vordringen, desto weiter steigt das Niveau der Abstraktionen und der Richtlinien. Der äußerste Kreis besteht aus untergeordneten konkreten Details. Und mit jedem weiteren Schritt nach innen wird die Software

zunehmend abstrakter und kapselt übergeordnete Richtlinien. Der zentrale Kreis stellt somit die allgemeinste und höchste Ebene dar.

22.1.6 Grenzen überschreiten

Unten rechts in dem Diagramm in Abbildung 22.1 findet sich ein Beispiel dazu, wie die Grenzlinien der Kreise überschritten werden. Hier wird die Kommunikation der Controller und Presenters mit den Use Cases im nächsten Layer demonstriert. Beachten Sie an dieser Stelle den Kontrollfluss: Er hat seinen Ursprung im `Controller`, geht dann über den `Use Case` weiter und endet schließlich mit der Ausführung im `Presenter`. Beachten Sie auch die Quellcode-Abhängigkeiten: Jede von ihnen weist nach innen in Richtung der Use Cases.

Normalerweise wird dieser offensichtliche Widerspruch durch die Anwendung des *Dependency-Inversion-Prinzips* aufgelöst. In einer Sprache wie Java würden z.B. die Schnittstellen und Vererbungsbeziehungen so angeordnet, dass die Quellcode-Abhängigkeiten genau an den richtigen Punkten entlang der Grenzlinien entgegen dem Kontrollfluss verlaufen.

Unterstellen wir einmal, dass der Use Case den `Presenter` aufrufen muss. Dieser Aufruf darf nicht direkt erfolgen, denn das wäre ein Verstoß gegen die Abhängigkeitsregel: Kein Name einer Komponente in einem der äußeren Kreise darf in einem der inneren Kreise Erwähnung finden. Der `Use Case` ruft also eine Schnittstelle (in Abbildung 22.1 mit `Use-Case-Ausgabeport` bezeichnet) im inneren Kreis auf und der `Presenter` im äußeren Kreis implementiert sie.

Die gleiche Technik wird auch angewendet, um sämtliche Grenzlinien in den Softwarearchitekturen zu überwinden. Hierbei werden die Vorzüge der dynamischen Polymorphie ausgenutzt, um dem Kontrollfluss entgegenlaufende Quellcode-Abhängigkeiten zu erzeugen, damit unabhängig von der Richtung des Kontrollflusses die Einhaltung der Abhängigkeitsregel gewährleistet ist.

22.1.7 Welche Daten überqueren die Grenzlinien?

Im Normalfall bestehen die grenzüberschreitenden Daten aus einfachen Datenstrukturen. Sie können also ganz nach Belieben grundlegende Konstrukte oder einfache Datentransferobjekte verwenden. Auch einfache Argumente in Funktionsaufrufen sind möglich. Oder Sie packen die Daten in eine Hashmap oder setzen sie zu einem Objekt zusammen. Wichtig hierbei ist nur, dass es isolierte, einfache Datenstrukturen sein müssen, die die Grenzen überqueren. Sie sollten nicht schummeln und etwa Entitätsobjekte oder Datensätze bzw. Tupel auf die Reise schicken. Die Datenstrukturen dürfen keinerlei Abhängigkeiten aufweisen, die gegen

die Abhängigkeitsregel verstoßen.

Beispielsweise beantworten viele Datenbank-Frameworks eine Abfrage in einem praktischen Datenformat. Nennen wir es einen »Datensatz«. Diese Datensatzstruktur sollte die Grenzlinien in Richtung der inneren Kreise allerdings nicht überschreiten, denn das würde gegen die Abhängigkeitsregel verstoßen, weil in diesem Fall zwangsläufig eine Komponente in einem inneren Kreis Kenntnis von einer Komponente in einem äußeren Kreis haben müsste.

Wenn also Daten über eine Grenzlinie hinweg übermittelt werden sollen, dann immer in der für den inneren Kreis geeignetsten Form.

22.2 Ein typisches Beispiel

Das in Abbildung 22.2 dargestellte Schema zeigt ein typisches Szenario für ein webbasiertes Java-System, das eine Datenbank nutzt. Der Webserver nimmt die Eingabedaten vom User in Empfang und leitet sie an den `Controller` oben links weiter. Der `Controller` packt diese Daten in ein simples Java-Objekt (*POJO*, **P**lain **O**ld **J**ava **O**bject) und übermittelt dieses dann über die Komponente `InputBoundary` an `UseCaseInteractor`. Der `UseCaseInteractor` interpretiert die empfangenen Daten und verwendet sie, um den »Tanz der Entitäten«, der `Entities`, zu steuern. Außerdem macht er sich das `DataAccessInterface` zunutze, um die von diesen Entitäten verwendeten Daten in den Speicher der Komponente `Database` zu übertragen. Sobald das geschehen ist, sammelt der `UseCaseInteractor` Daten von den `Entities` und erstellt die Komponente `OutputData` als ein weiteres einfaches Java-Objekt. Und schließlich werden die `OutputData` dann über die `OutputBoundary`-Schnittstelle an den `Presenter` weitergegeben.

Die Aufgabe der Komponente `Presenter` besteht darin, die `OutputData` in einer darstellbaren Form als `ViewModel` neu zusammenzufassen, was wiederum selbst bloß ein normales Java-Objekt ist. Das `ViewModel` enthält überwiegend `Strings` und Flags, die `View` zur Anzeige der Daten verwendet. Während `OutputData` `Date`-Objekte enthalten kann, lädt der `Presenter` das `ViewModel` mit entsprechenden `Strings`, die bereits in für den User geeigneter Weise formatiert sind. Das Gleiche gilt auch für `Currency`-Objekte oder sonstige geschäftsorientierte Daten. `Button`- und `MenuItem`-Bezeichnungen werden ebenso in die Komponente `ViewModel` platziert wie die Flags, die `View` anweisen, ob die betreffenden Systemelemente ausgegraut werden sollen.

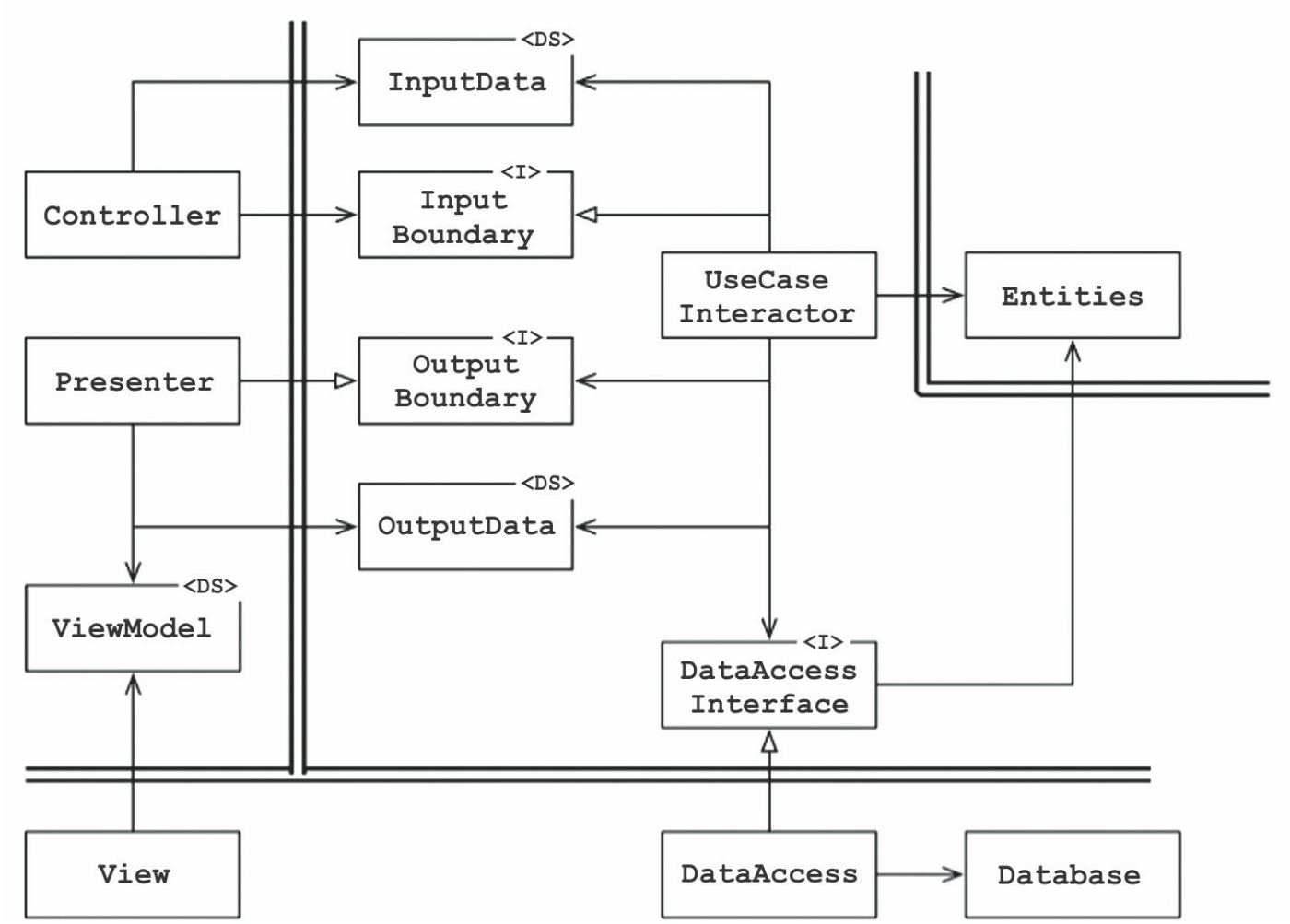


Abb. 22.2: Ein typisches Szenario für ein webbasiertes Java-System unter Verwendung einer Datenbank

Damit bleibt für `View` nichts weiter zu tun, als die Daten vom `ViewModel` in die `HTML`-Seite zu verschieben.

Beachten Sie auch hier wieder die Ausrichtung der Abhängigkeiten: Sie alle kreuzen die Grenzlinien in Richtung des Kreisinnern – ganz im Sinne der Abhängigkeitsregel.

22.3 Fazit

Die vorgenannten einfachen Regeln einzuhalten, ist nicht schwierig – und es wird Ihnen im weiteren Verlauf Ihrer Arbeit eine Menge Kopfzerbrechen ersparen. Sofern Sie die Software in Layer separieren und die Abhängigkeitsregel einhalten, werden Sie ein System erschaffen, das mit allen damit einhergehenden Vorteilen komplett testfähig ist. Sollten dann irgendwelche externen Systemkomponenten, wie etwa die Datenbank oder das Web-Framework, hinfällig werden, lassen sich diese Elemente mit minimalem Aufwand problemlos ersetzen.

[1] *Growing Object-Oriented Software with Tests*, Steve Freeman und Nat Pryce, Addison-Wesley Professional, 2009.

[2] *Object-Oriented Software Engineering: A Use Case Driven Approach*, Ivar Jacobson et al., Addison-Wesley Professional, 1992.

Kapitel 23

Presenters und »Humble Objects«

In Kapitel 22 wurde das Konzept der *Presenters* bereits erwähnt. Bei diesen Objekten handelt es sich um eine Ausführung des Design Patterns *Humble Object (Einfaches Objekt)*, das zur Identifizierung und zum Schutz architektonischer Grenzen beiträgt. Die saubere Architektur im vorigen Kapitel enthielt bereits eine Fülle von *Humble Object*-Implementierungen.

23.1 Das Pattern »Humble Object«

Das Pattern *Humble Object*[1] ist ein Entwurfsmuster, das ursprünglich als eine Methode entwickelt wurde, um die Separierung von schwierig und einfach zu testendem Systemverhalten im Rahmen von Unit-Tests zu erleichtern. Dahinter steckt eine sehr simple Idee: Die Verhaltensweisen werden in zwei Module oder Klassen aufgeteilt. Das eine Modul ist *humble* (zu Deutsch etwa »einfach, bescheiden«) und enthält alle auf ihren Kerngehalt herunterreduzierten schwer zu testenden Verhaltensweisen. Und das andere Modul enthält alle testfähigen Verhaltensweisen, um die das Humble Object reduziert wurde.

Zum Beispiel gestalten sich Unit-Tests für GUIs (**G**raphical **U**ser **I**nterfaces) normalerweise recht schwierig, weil es kompliziert ist, Tests zu schreiben, mit denen sich die Bildschirmausgabe erkennen und die darin angezeigten Elemente überprüfen lassen. Andererseits ist das Verhalten eines GUIs größtenteils einfach zu testen. Durch die Anwendung des Patterns *Humble Object* lassen sich diese zwei Arten von Verhalten in unterschiedliche Klassen namens `Presenter` und `View` trennen.

23.2 Presenters und Views

Die Komponente `View` ist das schwer zu testende Humble Object. Der darin enthaltene Code wird so einfach wie möglich gehalten. Dieses Objekt verschiebt Daten in das GUI, ohne sie jedoch zu verarbeiten.

Die Komponente `Presenter` repräsentiert dagegen das testfähigere Objekt. Ihre Aufgabe besteht darin, Daten von der Anwendung in Empfang zu nehmen und sie zu Präsentationszwecken zu formatieren, sodass die Komponente `View` sie einfach auf dem Bildschirm ausgeben kann. Fordert die Anwendung die Ausgabe eines Datums in einem Anzeigefeld an, dann übergibt sie der Komponente `Presenter` ein `Date`-Objekt. Der `Presenter` formatiert die zugehörigen Daten als passenden String und platziert diesen anschließend in eine einfache Datenstruktur, das *View Model*, wo die Komponente `View` sie aufspüren kann.

Fordert die Anwendung dagegen die Bildschirmausgabe eines Geldbetrags an, dann könnte sie stattdessen ein `Currency`-Objekt an den `Presenter` übermitteln. Dieser formatiert das Objekt als Nächstes mit den entsprechenden Dezimalstellen und Währungssymbolen und erzeugt so einen String, den er ebenfalls in das View Model platzieren kann. Soll der Zahlenwert für den Fall, dass er negativ ist, rot gefärbt werden, dann wird einfach zusätzlich noch ein passendes boolesches Flag im View Model gesetzt.

Jede Schaltfläche auf dem Bildschirm hat eine Bezeichnung. Dieser Name wird von

der Komponente `Presenter` als String im View Model hinterlegt. Sollen die betreffenden Schaltflächen ausgegraut dargestellt werden, setzt der `Presenter` bei der Übergabe an das View Model hierzu ebenfalls ein passendes boolesches Flag. Ebenso werden auch die Bezeichnungen jedes einzelnen Menüeintrags vom `Presenter` im View Model platziert und finden sich dort dann als Strings wieder. Das Gleiche gilt für die Bezeichnungen jedes Optionsfelds, Kontrollkästchens und Textfelds: Der `Presenter` wandelt all diese Angaben in entsprechende Strings und boolesche Werte und lädt sie in das View Model. Und schließlich werden auch Zahlenkolonnen, die auf dem Bildschirm ausgegeben werden sollen, in Form von korrekt formatierten Strings als Tabellen vom `Presenter` in das View Model eingespeist.

Mit anderen Worten: Alles, was auf dem Bildschirm erscheint und in irgendeiner Form der Kontrolle der Anwendung unterliegt, findet sich im View Model als String, boolescher oder numerischer Wert wieder. Damit fällt der Komponente `View` lediglich noch die Aufgabe zu, die Daten aus dem View Model zu laden und auf den Bildschirm auszugeben. Und das bedeutet, dass die Komponente `View` »humble«, also einfach ist.

23.3 Das Testen und die Softwarearchitektur

Die Testfähigkeit ist bekanntermaßen eine der charakteristischen Eigenschaften von gelungenen Softwarearchitekturen. Das Pattern *Humble Object* ist ein gutes Beispiel dafür, weil die Unterteilung der Verhaltensweisen in testfähige und nicht testfähige Bestandteile häufig eine architektonische Grenze definiert. Die `Presenter/View`-Grenzlinie ist dabei nur ein Vertreter ihrer Art, es gibt aber noch zahlreiche weitere.

23.4 Datenbank-Gateways

Zwischen den Use-Case-Interaktoren und der Datenbank befinden sich die *Datenbank-Gateways*.[2] Bei diesen Gateways handelt es sich um polymorphe Schnittstellen, die Methoden für jede Erstellungs-, Lese-, Aktualisierungs- oder Löschoperation enthalten, die vonseiten der Anwendung in der Datenbank ausgeführt werden kann. Fordert die Anwendung beispielsweise die Nachnamen aller User an, die sich am Vortag eingeloggt haben, dann nutzt die `UserGateway`-Schnittstelle eine Methode namens `getLastNamesOfUsersWhoLoggedInAfter`, die mithilfe des Arguments `Date` eine Liste der gesuchten Nachnamen zurückgibt.

Denken Sie daran, dass im Use-Cases-Layer kein SQL zulässig ist. Stattdessen kommen hier Gateway-Schnittstellen mit geeigneten Methoden zur Anwendung. Diese Gateways werden durch Klassen im Datenbank-Layer implementiert. Und diese Implementierung ist dann das Humble Object, das schlicht und ergreifend SQL oder

eine sonstige gegebene Schnittstelle zu der Datenbank verwendet, um auf die von den einzelnen Methoden benötigten Daten zuzugreifen. Die Interaktoren sind demgegenüber nicht »humble«, weil sie anwendungsspezifische Geschäftsregeln kapseln – sie sind aber dennoch testfähig, weil die Gateways hier mit entsprechenden Stubs und stellvertretenden Proxies ersetzt werden können.

23.5 Data Mappers

Kommen wir nun noch einmal zum Thema Datenbanken zurück. Was glauben Sie, in welchen Layer *ORM-Frameworks* (**O**bject-**R**elational **M**apping, objektrelationale Abbildung) wie Hibernate hineingehören?

Lassen Sie mich an dieser Stelle zunächst etwas klarstellen: So etwas wie objektrelationale Abbildungen (ORMs) gibt es im Grunde genommen gar nicht – und zwar aus einem einfachen Grund: Objekte sind keine Datenstrukturen. Zumindest nicht aus der Sicht der User. Die User eines Objekts können die Daten nicht sehen, weil sie alle privat sind. Sie sehen nur die öffentlichen Methoden des betreffenden Objekts. Insofern ist ein Objekt aus der Perspektive des Users betrachtet einfach nur ein Satz von Operationen.

Im Gegensatz dazu ist eine Datenstruktur ein Satz öffentlicher Datenvariablen, die kein implizites Verhalten aufweisen. ORMs sollten daher besser als *Data Mappers* bezeichnet werden, weil sie Daten aus relationalen Datenbanktabellen in Datenstrukturen laden.

Aber zurück zu meiner eingangs gestellten Frage: Wo sollten sich solche ORM-Systeme befinden? Im Datenbank-Layer natürlich! Faktisch bilden ORMs eine andere Art von Humble-Object-Grenze zwischen den Gateway-Schnittstellen und der Datenbank.

23.6 Service Listeners

Und wie steht es mit Services? Wenn Ihre Anwendung mit anderen Services kommunizieren muss oder einen Satz von Services zur Verfügung stellt, erstellt das Pattern *Humble Object* dann eine Servicegrenze?

Natürlich! Die Anwendung lädt die Daten in einfache Datenstrukturen und leitet diese dann über die Grenzlinie hinweg an Module weiter, die die Daten zweckgemäß formatieren und an externe Services übermitteln. Auf der Eingangsseite erhalten solche *Service Listeners* Daten von der Serviceschnittstelle und formatieren sie in eine

einfache Datenstruktur, die von der Anwendung genutzt werden kann. Und anschließend wird diese Datenstruktur dann über die Servicegrenze weitergereicht.

23.7 Fazit

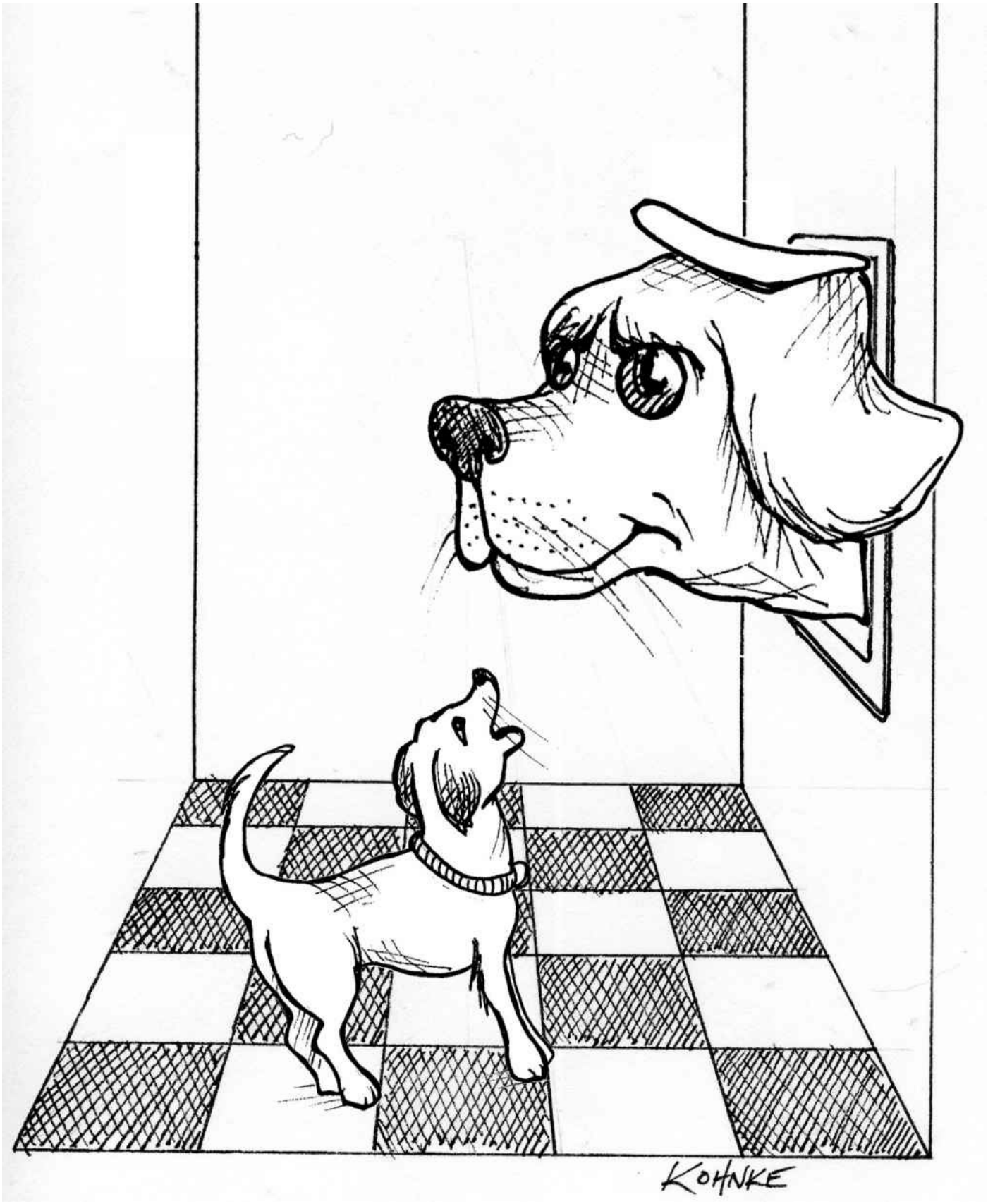
Wo es eine architektonische Grenze gibt, ist mit hoher Wahrscheinlichkeit auch das Design Pattern *Humble Object* nicht weit. Die Kommunikation über diese Grenzlinie hinweg bedingt fast immer irgendeine Art von einfacher Datenstruktur. Die Grenze selbst wird in der Regel die schwierig zu testenden Komponenten von den einfach zu testenden trennen. Generell trägt der Einsatz dieses Patterns an architektonischen Grenzen in erheblichem Maße zur Testfähigkeit des gesamten Systems bei.

---

[1] *xUnit Test Patterns: Refactoring Test Code*, Gerard Meszaros, Addison, 2007, S. 695.

[2] *Patterns für Enterprise Application-Architekturen*, Martin Fowler et al., mitp Professional, 2003.

Kapitel 24
Partielle Grenzen



Vollwertige architektonische Grenzen sind kostenintensiv. Sie erfordern reziproke polymorphe Grenzschnittstellen, Ein-/Ausgabe-Datenstrukturen sowie das nötige umfassende Abhängigkeitsmanagement, um die beiden Bereiche in unabhängig voneinander kompilierbare und deploybare Komponenten zu isolieren. Das bedeutet viel Arbeit – auch in Bezug auf die Instandhaltung.

Aus diesem Grund befinden gute Softwarearchitekten den Aufwand für das Ziehen einer solchen Grenze in vielen Situationen zunächst für zu hoch – möchten sich aber dennoch die Möglichkeit offenhalten, dies bei Bedarf später nachzuholen.

Ein derartig vorausschauendes Design wird von der agilen Community oftmals als Verstoß gegen das YAGNI-Prinzip missbilligt: »**Y**ou **A**ren't **G**oing to **N**eed **I**t« (zu Deutsch etwa: »Du wirst es nicht brauchen«). Die Softwarearchitekten sehen das mitunter jedoch etwas anders und halten dagegen: »Sicher, aber vielleicht brauche ich sie ja doch irgendwann.« In solchen Fällen können sie alternativ auch eine *partielle Grenze* implementieren.

24.1 Den letzten Schritt weglassen

Eine Möglichkeit, eine partielle Grenze einzurichten, besteht darin, alle notwendigen Vorbereitungen für die Erstellung unabhängig voneinander kompilierbarer und deploybarer Komponenten zu treffen und sie dann einfach gemeinsam in einer Komponente abzulegen. Auf diese Weise sind sowohl die reziproken Schnittstellen als auch die Datenstrukturen für die Ein-/Ausgabe vorhanden, und alles Notwendige steht bereit – wird aber als eine einzige Komponente kompiliert und deployt.

Zwar bleibt der Umfang des Codes und der vorbereitenden Designarbeit für eine solche partielle Grenze natürlich derselbe wie bei einer vollwertigen Grenzlinie – was jedoch wegfällt, ist der Aufwand für die Verwaltung mehrerer Komponenten. Die Last der Versionskontrolle und des Releasemanagements entfällt. Und dieser auf den ersten Blick vielleicht eher unscheinbare Unterschied ist nicht zu unterschätzen.

Genau diese Strategie verfolgten wir anfangs auch bei unserer Arbeit an dem Testframework *FitNesse*. Wir entwarfen die Webserver-Komponente in der Form, dass sie von dem Wiki- und dem eigentlichen Testbereich getrennt werden konnte. Der Leitgedanke dahinter war, dass wir gegebenenfalls noch andere webbasierte Anwendungen mit dieser Web-Komponente erstellen könnten. Außerdem wollten wir vermeiden, dass die User zwei Komponenten herunterladen mussten – denn wie Sie sich vielleicht erinnern werden, lautete eins unserer Designziele »Download and Go«. Geplant war also, dass die User nur eine einzige `.jar`-Datei herunterladen und ausführen sollten, ohne gezwungen zu sein, darüber hinaus noch weiteren Dateien nachzujagen, die Versionskompatibilität sicherzustellen und Ähnliches.

Das Beispiel des FitNesse-Testframeworks zeigt allerdings auch die Tücken dieses Ansatzes: Als im Laufe der Zeit immer klarer wurde, dass eine separate Web-Komponente niemals benötigt werden würde, rückte damit bei der weiteren Entwicklungsarbeit auch die Separierung der Web- und Wiki-Komponenten zunehmend in den Hintergrund. Und das hatte letztlich zur Folge, dass die

Abhängigkeiten die Grenzlinie ganz allmählich auch in die falsche Richtung kreuzten – sie heute, also nachträglich, wieder sauber voneinander zu trennen, wäre deshalb eine ziemlich mühsame Aufgabe.

24.2 Eindimensionale Grenzen

Die vollwertige architektonische Grenze macht sich reziproke Grenzschnittstellen zunutze, um die Isolation in beiden Richtungen zu gewährleisten. Die Aufrechterhaltung der Separierung in beide Richtungen ist sowohl bei der Ersteinrichtung als auch in der laufenden Instandhaltung aufwendig bzw. kostenintensiv.

Eine einfachere Struktur, die als Platzhalter für eine spätere Erweiterung zu einer vollwertigen Grenze dient, ist in Abbildung 24.1 dargestellt. Sie veranschaulicht das traditionelle Design Pattern *Strategy (Strategie)*. Der Client nutzt hier eine `ServiceBoundary`-Schnittstelle, die ihrerseits von der Klasse `ServiceImp` implementiert wird.

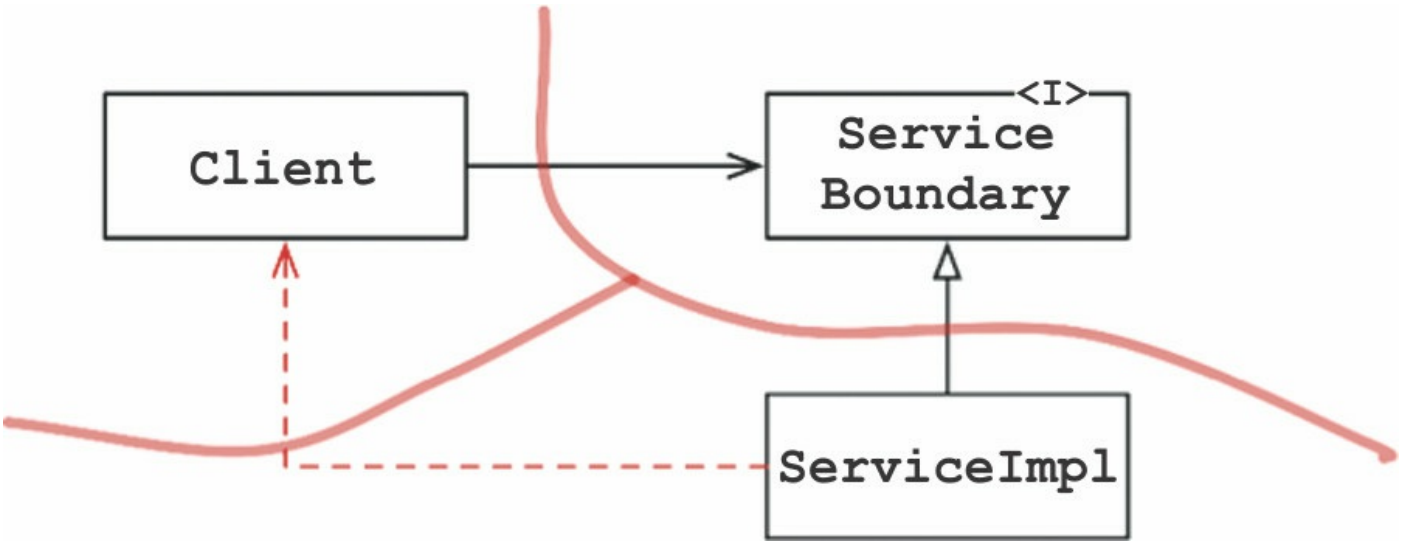


Abb. 24.1: Das Design Pattern *Strategy (Strategie)*

Es ist klar erkennbar, dass an dieser Stelle die »Bühne« für eine zukünftige architektonische Grenze bereitet wird. Hier greift die unerlässliche Abhängigkeitsumkehr, um den `Client` von der Klasse `ServiceImp` zu isolieren. Zudem wird anhand des gestrichelten Pfeils ebenfalls deutlich, dass die Separierung relativ schnell abfallen kann. Ohne reziproke Schnittstellen gibt es abgesehen von der Sorgfalt und der Disziplin der Softwareentwickler und Softwarearchitekten keinerlei Schutz gegen diese Art von *Backchannel* (Rückkanal).

24.3 Fassaden

Eine noch simplere Grenze erstellt das Design Pattern *Facade (Fassade)*, das in Abbildung 24.2 illustriert ist. In diesem Fall wird auch die Abhängigkeitsumkehr geopfert. Die Grenzlinie wird einfach durch die Klasse `Facade` definiert, die alle Dienste als Methoden auflistet und die Serviceaufrufe an Klassen deployt, auf die der Client nicht zugreifen soll.

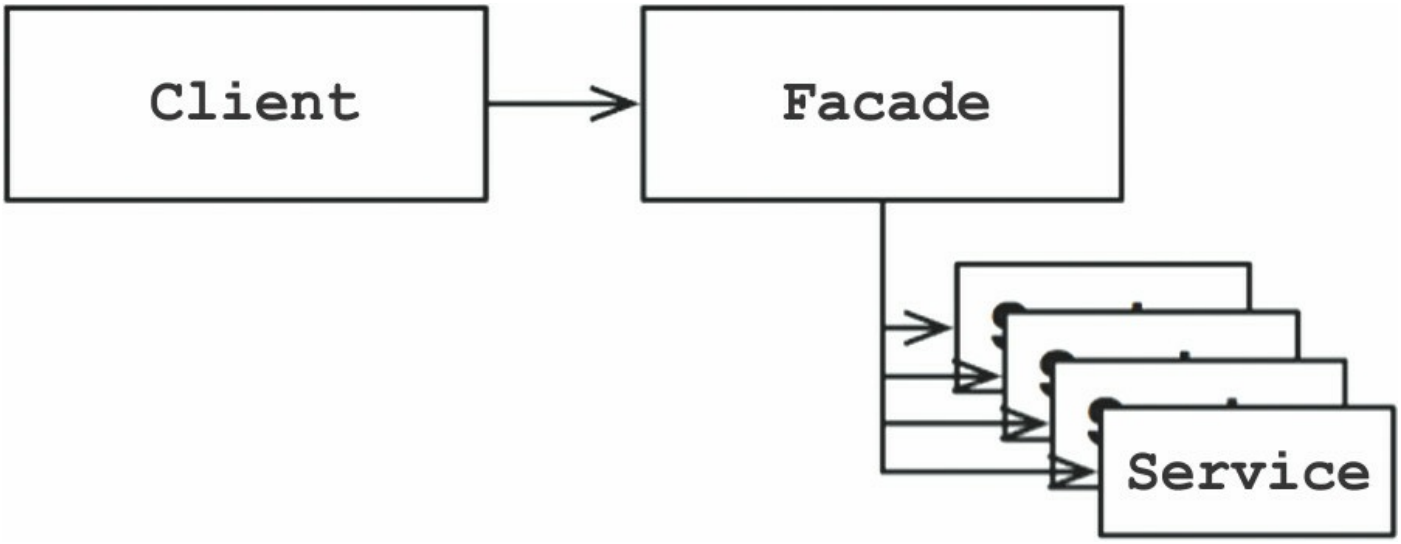


Abb. 24.2: Das Design Pattern *Facade (Fassade)*

Beachten Sie jedoch, dass der `Client` in einer transitiven Abhängigkeit zu all diesen `Service`-Klassen steht. In statischen Programmiersprachen würde eine Modifikation am Quellcode einer der `Service`-Klassen den `Client` zur Neukompilierung zwingen – und in Anbetracht dessen fällt es nicht allzu schwer, sich vorzustellen, wie schnell bei einem derartigen strukturellen Aufbau Rückkanäle entstehen können.

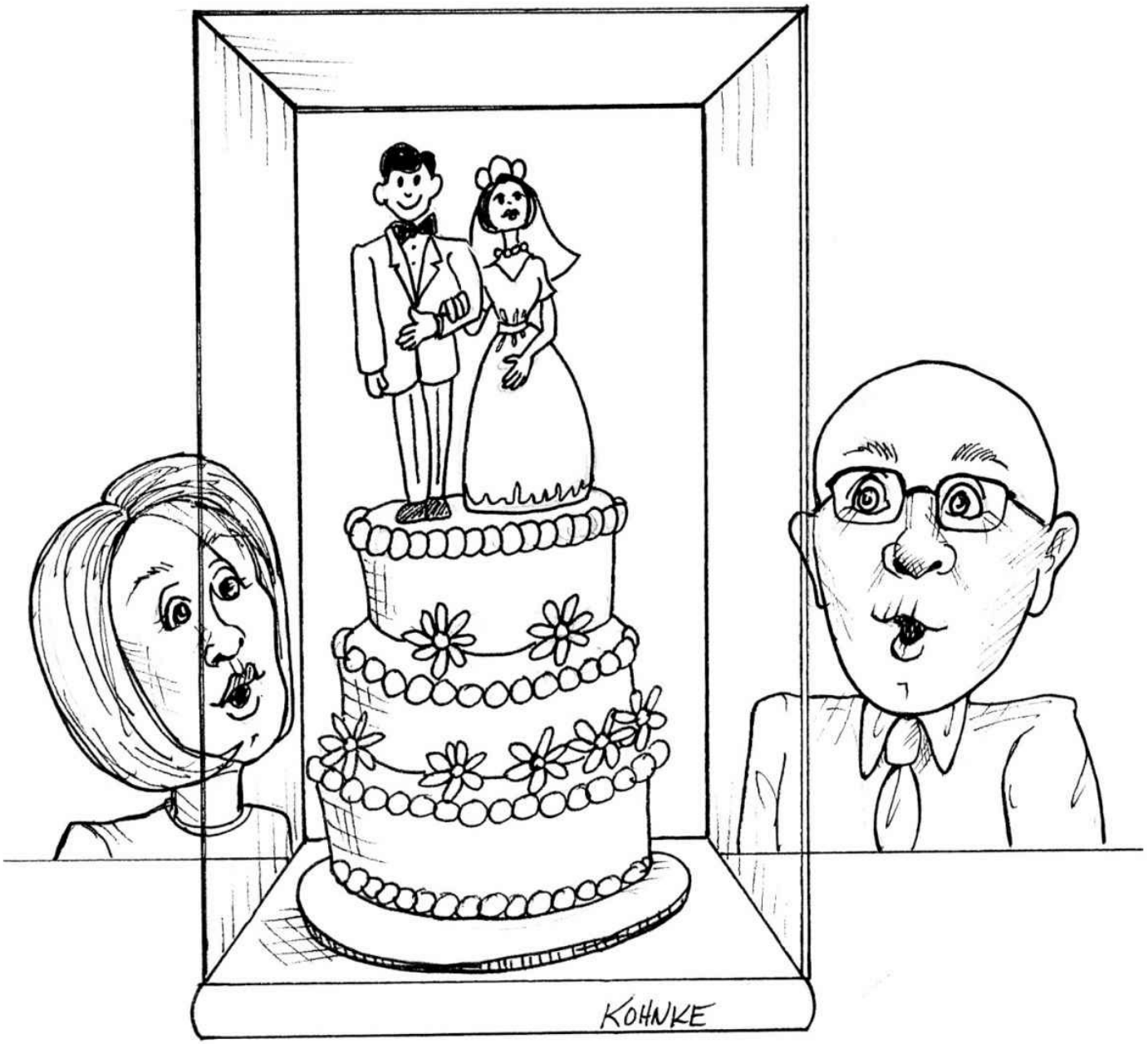
24.4 Fazit

In diesem Kapitel wurden drei einfache Wege aufgezeigt, die partielle Implementierung einer architektonischen Grenze zu realisieren. Diese drei Strategien sollen Ihnen als Beispiele dienen.

Jeder der hier vorgestellten Ansätze hat seine individuellen Vor- und Nachteile, aber sie alle sind in bestimmten Kontexten als Platzhalter für eine mögliche vollwertige Grenze geeignet. Und sollte diese Grenze zukünftig doch nicht notwendig werden, dann kann die gewählte Vorgehensweise bei Bedarf auch einfach wieder vernachlässigt werden.

Die Entscheidung, wo möglicherweise eines Tages eine architektonische Grenze existieren soll und ob diese Grenzlinie vollständig oder partiell zu integrieren ist, liegt im Endeffekt bei dem Softwarearchitekten des Systems.

Kapitel 25
Layer und Grenzen



Es ist durchaus vorstellbar, dass Softwaresysteme aus nur drei Komponenten bestehen: der Benutzerschnittstelle, den Geschäftsregeln und der Datenbank. Für einfache Systeme ist das auch ausreichend, in den meisten findet sich allerdings eine größere Zahl von Komponenten.

Denken Sie zum Beispiel an ein einfaches Computerspiel. Hier könnten die drei Grundkomponenten doch vielleicht ausreichen: Die Benutzerschnittstelle transportiert alle Eingaben des Spielers an die Spielregeln, und die Spielregeln speichern den Status des Spiels wiederum in irgendeiner Form von permanenter Datenstruktur. Aber ist das wirklich schon genug?

25.1 Hunt the Wumpus

Betrachten wir dazu einmal ein konkretes Beispiel, und zwar das altgediente Adventurespiel »Hunt the Wumpus« aus dem Jahr 1972. Bei diesem textbasierten Spiel kommen sehr einfache Befehlseingaben wie GO EAST (gehe nach Osten) und SHOOT WEST (schieße in Richtung Westen) zum Einsatz. Der Spieler gibt den gewünschten Befehl ein und der Computer reagiert bzw. antwortet darauf, indem er den Spieler etwas sehen, hören und erleben lässt. Bei der »Jagd nach dem Wumpus« geht es darum, in einem Höhlenlabyrinth eine virtuelle Kreatur, den Wumpus, aufzuspüren und dabei Fallen, Gruben und anderen lauernden Gefahren zu entgehen. (Falls Sie sich für die genauen Spielregeln interessieren, werden Sie im Internet schnell fündig.)

Angenommen, wir behalten nun die textbasierte Benutzerschnittstelle bei, entkoppeln sie aber von den Spielregeln, damit unsere Version auf verschiedenen Märkten in unterschiedlichen Sprachen genutzt werden kann. Die Spielregeln sollen über eine sprachunabhängige Programmierschnittstelle (*API*, **A**pplication **P**rogramming **I**nterface) mit der UI-Komponente kommunizieren, und die Benutzerschnittstelle soll die API in die jeweils gewünschte menschliche Sprache übersetzen.

Sofern die Quellcode-Abhängigkeiten vernünftig verwaltet werden (siehe Abbildung 25.1), können ein und dieselben Spielregeln von einer beliebigen Anzahl von UI-Komponenten wiederverwendet werden. Die Spielregeln selbst haben keine Kenntnis davon – und sind auch nicht davon beeinflusst –, welche menschliche Sprache verwendet wird.

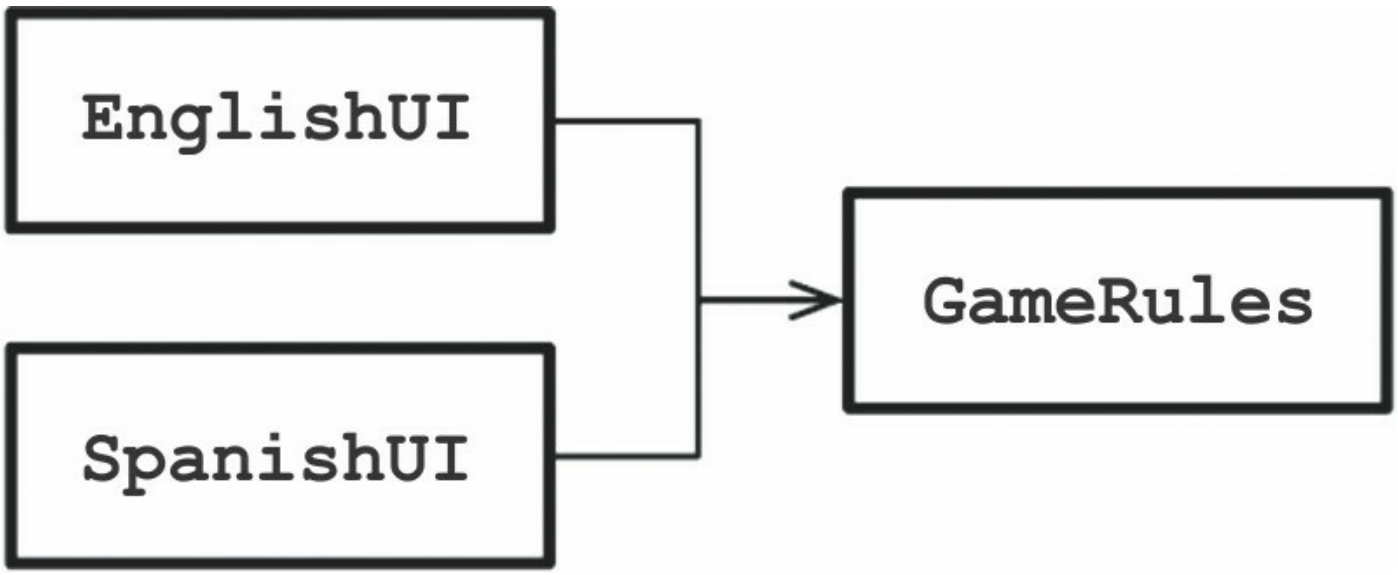


Abb. 25.1: Die Spielregeln können von einer beliebigen Anzahl von UI-Komponenten wiederverwendet werden.

Gehen wir nun weiter davon aus, dass der Zustand des Spiels in einem permanenten Speicher gehalten wird – beispielsweise in einem Flashspeicher, in der Cloud oder einfach nur im RAM. In allen Fällen soll ausgeschlossen sein, dass die Spielregeln

Kenntnis von den Details haben – deshalb wird auch hier wieder eine API erstellt, die die Spielregeln für die Kommunikation mit der Datenspeicherungskomponente nutzen kann.

Weil die Spielregeln darüber hinaus auch keinerlei Kenntnis von den verschiedenen Datenspeichervarianten haben sollen, müssen die Abhängigkeiten sorgfältig entsprechend der Abhängigkeitsregel ausgerichtet sein, wie in Abbildung 25.2 dargestellt.

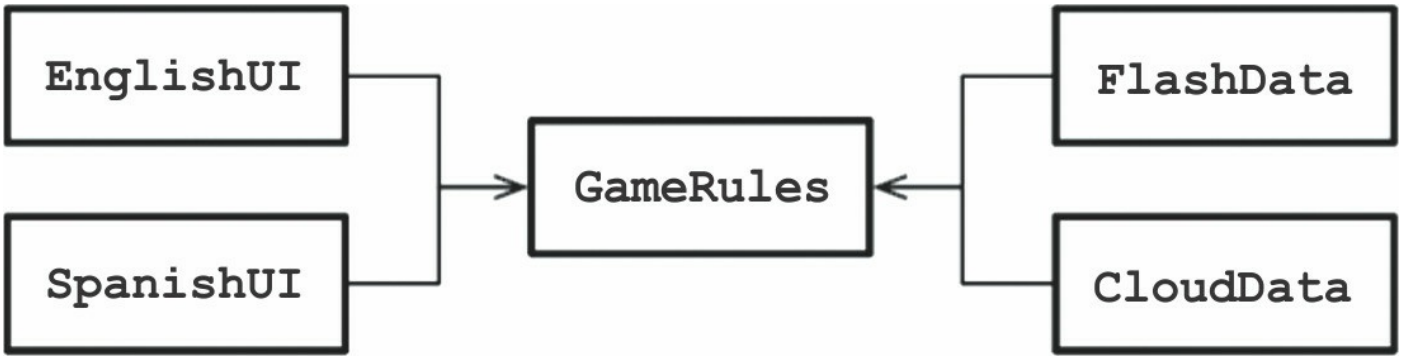


Abb. 25.2: Einhaltung der Abhängigkeitsregel

25.2 Saubere Architektur?

Es steht außer Frage, dass sich das Konzept der sauberen Softwarearchitektur angesichts all der Use Cases, Grenzen, Entitäten sowie der entsprechenden Datenstrukturen in diesem Kontext problemlos anwenden lassen würde.[1] Aber wurden hier auch wirklich alle relevanten architektonischen Grenzen aufgespürt?

Beispielsweise ist die Sprache nicht die einzige Modifikationsachse für die Benutzerschnittstelle. Die Mechanismen, mit deren Hilfe der Text kommuniziert wird, sollten ebenfalls variieren. Zum Beispiel könnten eine herkömmliche Eingabeaufforderung, Textmeldungen oder eine Chat-Anwendung zum Einsatz kommen. Es gibt viele verschiedene Möglichkeiten.

Das bedeutet, dass es eine potenzielle architektonische Grenze gibt, die von dieser Modifikationsachse definiert wird. Vielleicht sollten wir eine API errichten, die diese Grenze überschreitet und die Sprache von dem Kommunikationsmechanismus isoliert. Ein solches Konzept ist in Abbildung 25.3 illustriert.

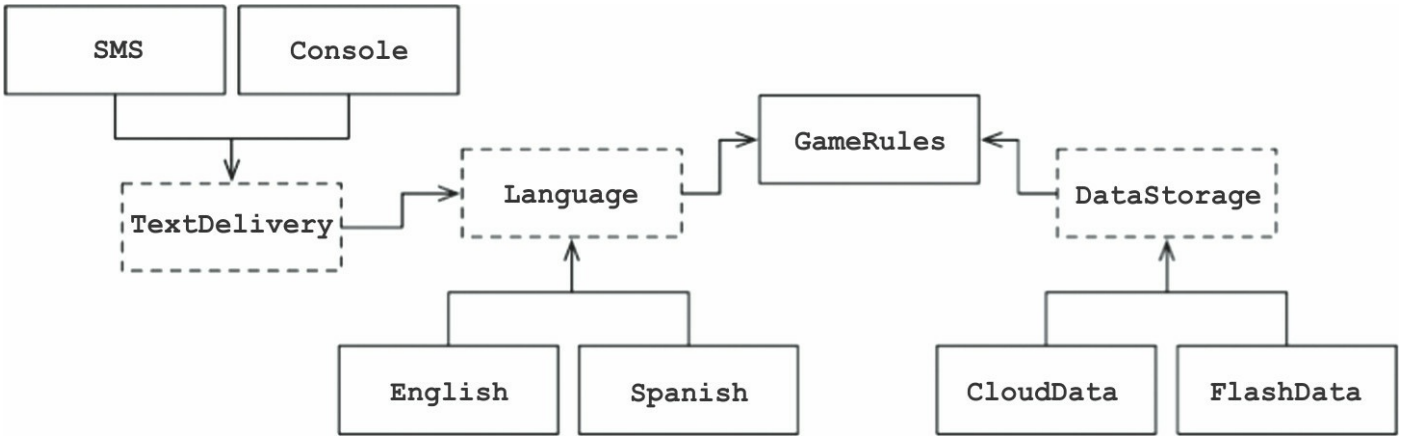


Abb. 25.3: Das umstrukturierte Diagramm

Die schematische Darstellung in Abbildung 25.3 ist ein wenig komplexer als das vorige Diagramm, sollte aber keine großen Überraschungen enthalten. Die gestrichelten Kästen kennzeichnen abstrakte Komponenten, die eine API definieren, die von den darüber- oder darunterliegenden Komponenten implementiert ist. So wird zum Beispiel die `Language`-API von `English` und `Spanish` implementiert.

`GameRules` kommuniziert mit `Language` über eine API, die von `GameRules` definiert und von `Language` implementiert wird. Die Komponente `Language` kommuniziert ihrerseits mit `TextDelivery` über eine API, die von `Language` definiert, aber von `TextDelivery` implementiert wird. Die API befindet sich im Besitz des Users und wird auch von diesem definiert, nicht von der implementierenden Instanz.

Ein Blick in die Komponente `GameRules` würde polymorphe `Boundary`-Schnittstellen erkennen lassen, die von dem Code in `GameRules` verwendet und von dem Code in `Language` implementiert werden. Außerdem wären hier polymorphe `Boundary`-Schnittstellen zu finden, die von der Komponente `Language` verwendet und von dem Code in `GameRules` implementiert werden.

Bei einem Blick in die Komponente `Language` böte sich ein ähnliches Bild: Hier gäbe es polymorphe `Boundary`-Schnittstellen, die von dem Code in `TextDelivery` implementiert werden, sowie polymorphe `Boundary`-Schnittstellen, die von der Komponente `TextDelivery` verwendet und von `Language` implementiert werden.

In jedem Fall befindet sich die von diesen `Boundary`-Schnittstellen definierte API im Besitz der Upstream-Komponente.

Die Variationen, wie etwa `English`, `SMS` und `CloudData`, werden von in der abstrakten API-Komponente definierten polymorphen Schnittstellen bereitgestellt und von den konkreten Komponenten, die ihnen zuarbeiten, implementiert. So könnten Sie zum Beispiel davon ausgehen, dass in der Komponente `Language` definierte polymorphe Schnittstellen von `English` und `Spanish` implementiert werden.

Vereinfacht lässt sich dieses Diagramm darstellen, indem sämtliche Variationen außer

Acht gelassen und nur die API-Komponenten berücksichtigt werden. Das Ergebnis ist in Abbildung 25.4 zu sehen.

Wie Sie sehen, ist das Diagramm in Abbildung 25.4 jetzt so angeordnet, dass alle Pfeile nach oben weisen und `GameRules` somit an die Spitze gerückt ist. Diese Anordnung macht auch Sinn, denn die Komponente `GameRules` enthält in diesem Beispiel die hochschichtigsten übergeordneten Richtlinien.

Beachten Sie hier auch die Richtung des Informationsflusses. Sämtliche Eingaben seitens des Users erfolgen mithilfe der Komponente `TextDelivery` unten links. Diese Informationen durchlaufen die `Language`-Komponente, die sie in Befehle übersetzt und an `GameRules` weiterleitet. Als Nächstes werden die Usereingaben von der Komponente `GameRules` verarbeitet, die danach die passenden Daten an die Komponente `DataStorage` unten rechts übermittelt.

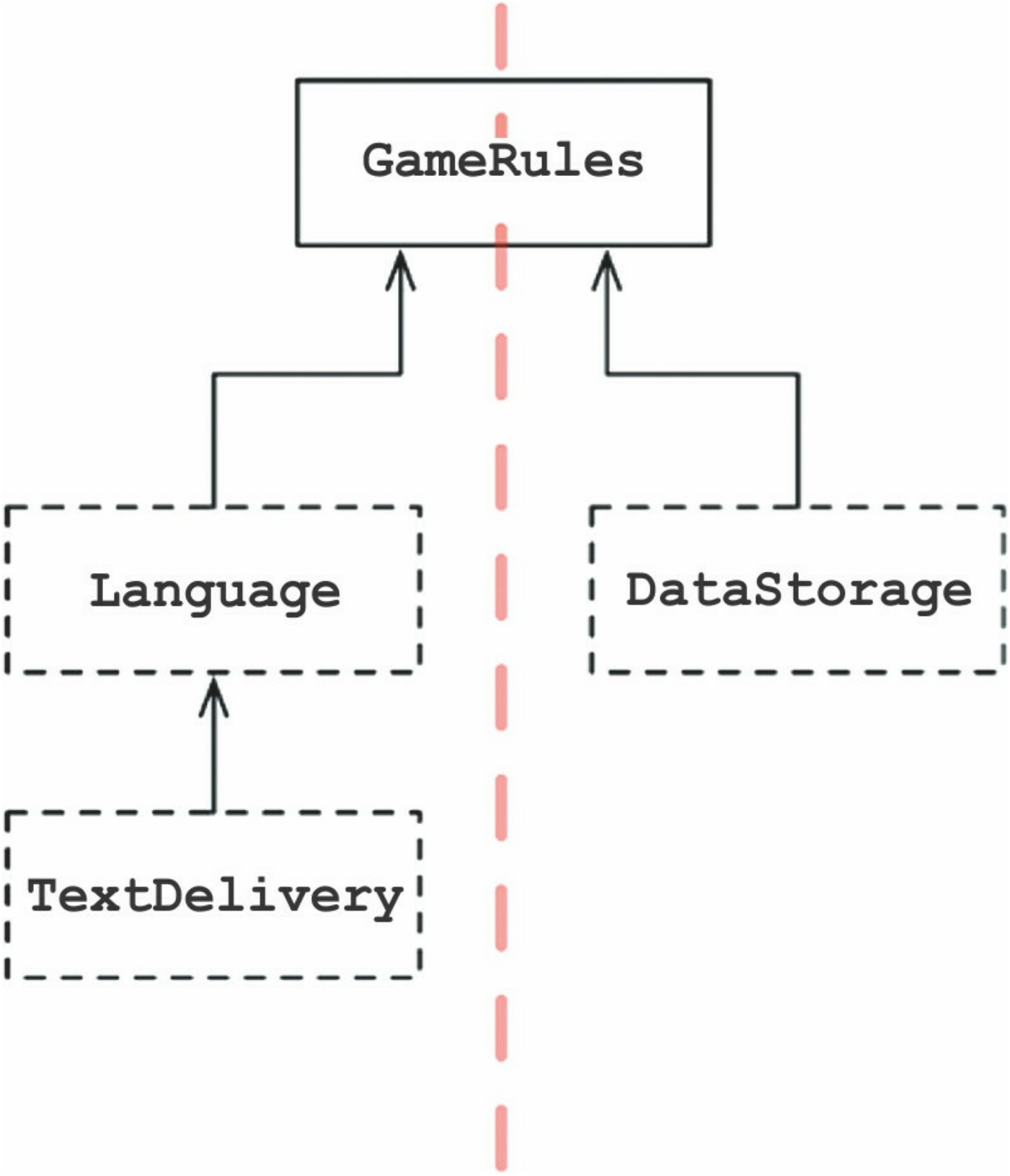


Abb. 25.4: Vereinfachte schematische Darstellung

Und schließlich gibt `GameRules` die Ausgabe nach unten an die Komponente `Language` zurück, die die API in die jeweils passende Sprache zurückübersetzt und über die Komponente `TextDelivery` an den User übermittelt.

Bei dieser Anordnung wird der Datenfluss in effizienter Weise in zwei Datenströme unterteilt:[2] Der Datenstrom auf der linken Seite konzentriert sich auf die

Kommunikation mit dem User, und der Datenstrom auf der rechten Seite befasst sich mit der Datenpersistenz. Beide Datenströme laufen an der Spitze[3] bei der Komponente `GameRules` zusammen, die ultimativ für die Verarbeitung der Daten zuständig ist, die beide Datenströme durchlaufen.

25.3 Datenstromüberschreitungen

Gibt es immer zwei Datenströme wie in diesem Beispiel? Nein, absolut nicht. Stellen Sie sich einmal vor, Sie wollten »Hunt the Wumpus« im Multiplayer-Modus mit anderen Spielern im Internet spielen. In diesem Fall wäre, wie in Abbildung 25.5 gezeigt, eine Netzwerkkomponente erforderlich. Bei der hier dargestellten schematischen Anordnung wird der Datenfluss in drei Datenströme unterteilt, die alle von der Komponente `GameRules` gesteuert werden.

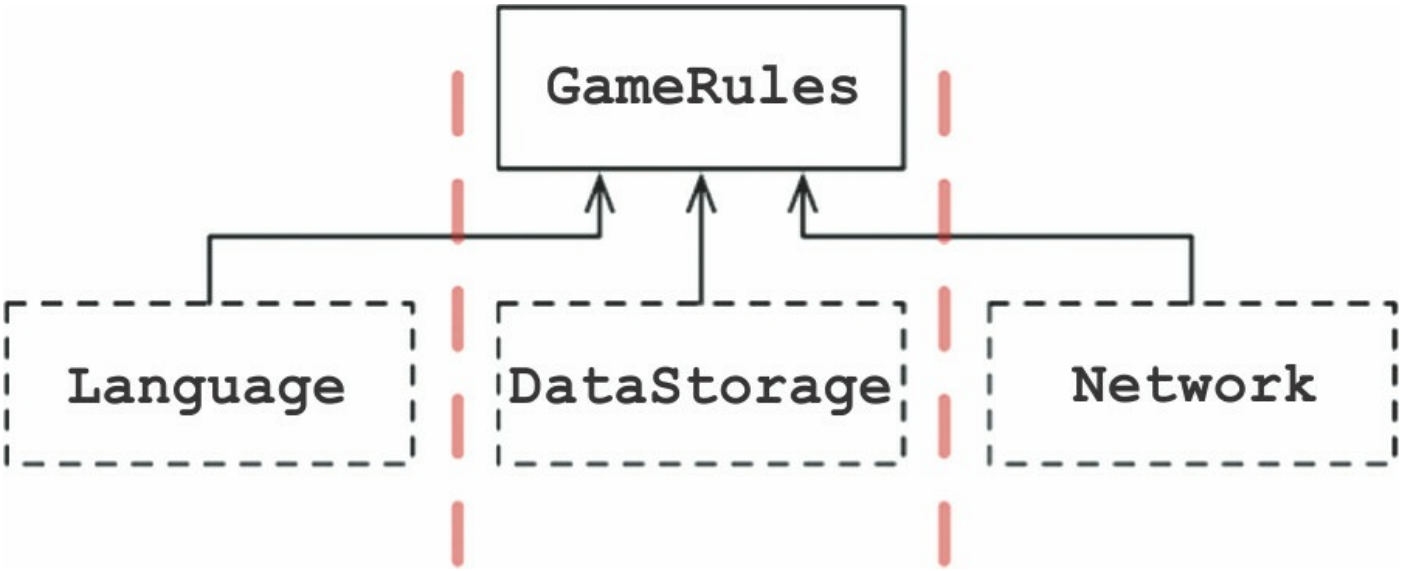


Abb. 25.5: Ergänzung einer Netzwerkkomponente

Je komplexer ein System wird, in umso mehr solcher Datenströme verzweigt die Komponentenstruktur.

25.4 Datenströme teilen

An dieser Stelle mögen Sie vielleicht denken, dass alle Datenströme letztendlich in einer einzigen Komponente zusammenlaufen. Wenn es doch nur so einfach wäre! Die Realität ist allerdings ein bisschen komplexer.

Nehmen wir zum Beispiel die `GameRules`-Komponente des Spiels »Hunt the Wumpus«. Ein Teil der Spielregeln bezieht sich auf die Mechanik der Karte. Sie legen fest, wie die Gänge in dem Labyrinth miteinander verbunden sind und welche Gegenstände sich in jeder Höhle befinden. Ebenso bestimmen sie, wie sich der Spieler von Höhle zu Höhle fortbewegen kann bzw. darf und wann die verschiedenen Ereignisse ausgelöst werden, die der Spieler meistern muss.

Diesen Regeln ist jedoch noch ein weiterer Satz Richtlinien übergeordnet – die wiederum die Lebensenergie des Spielers sowie die Kosten bzw. den Nutzen eines bestimmten Ereignisses regulieren. Sie veranlassen, dass der Spieler nach und nach Lebensenergie verliert oder aber durch das Auffinden und Einsammeln von Nahrungsgegenständen hinzugewinnt. Die untergeordneten Richtlinien zur Spielmechanik deklarieren Ereignisse wie `FoundFood` (Nahrung gefunden) oder `FellInPit` (in Grube gefallen) an diesen übergeordneten Richtliniensatz, der daraufhin seinerseits anhand der übermittelten Daten den Zustand des Spielers verwaltet (wie in Abbildung 25.6 veranschaulicht). Und in der Konsequenz entscheiden die übergeordneten Richtlinien schließlich auch darüber, ob der Spieler das Spiel gewinnt oder verliert.

Handelt es sich hierbei um eine architektonische Grenze? Wird eine API benötigt, die die Komponenten `MoveManagement` und `PlayerManagement` separiert? Nun, gestalten wir das Ganze doch noch ein wenig interessanter, indem wir ein paar Microservices ergänzen.

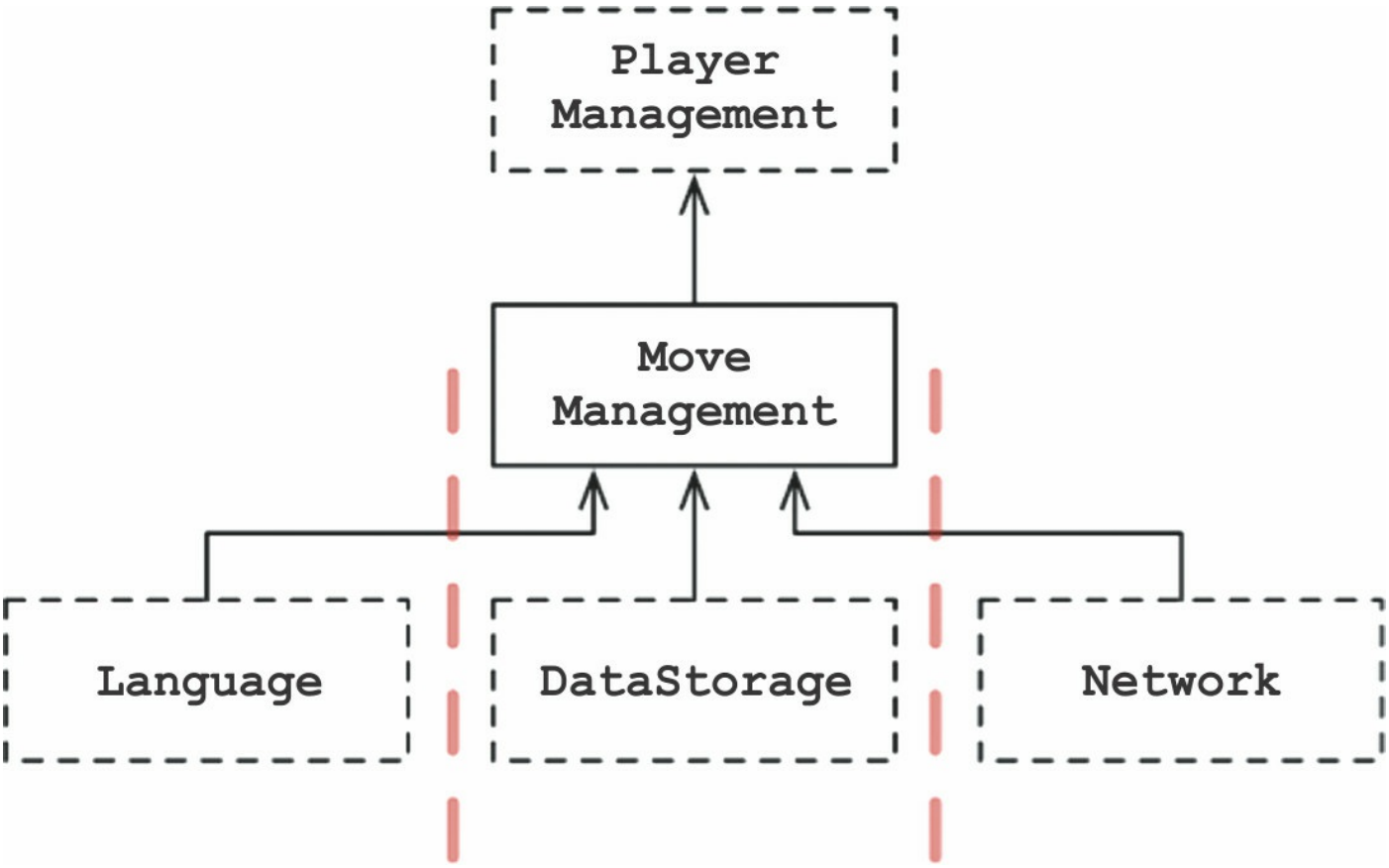


Abb. 25.6: Der übergeordnete Richtliniensatz verwaltet den Spielerstatus.

Unterstellen wir einmal, wir hätten es mit einer riesigen Multiplayer-Version von »Hunt the Wumpus« zu tun. Das `MoveManagement` wird lokal auf dem Computer des Spielers abgewickelt, das `PlayerManagement` hingegen findet auf einem Server statt. Nun bietet `PlayerManagement` allen angeschlossenen `MoveManagement`-Komponenten eine Microservice-API an.

Das Diagramm in Abbildung 25.7 gibt dieses Szenario in einer leicht gekürzten Form wieder. Die `Network`-Elemente sind ein wenig komplexer als hier dargestellt – der Grundgedanke wird aber dennoch deutlich. In diesem Fall existiert eine vollwertige architektonische Grenze zwischen `MoveManagement` und `PlayerManagement`.

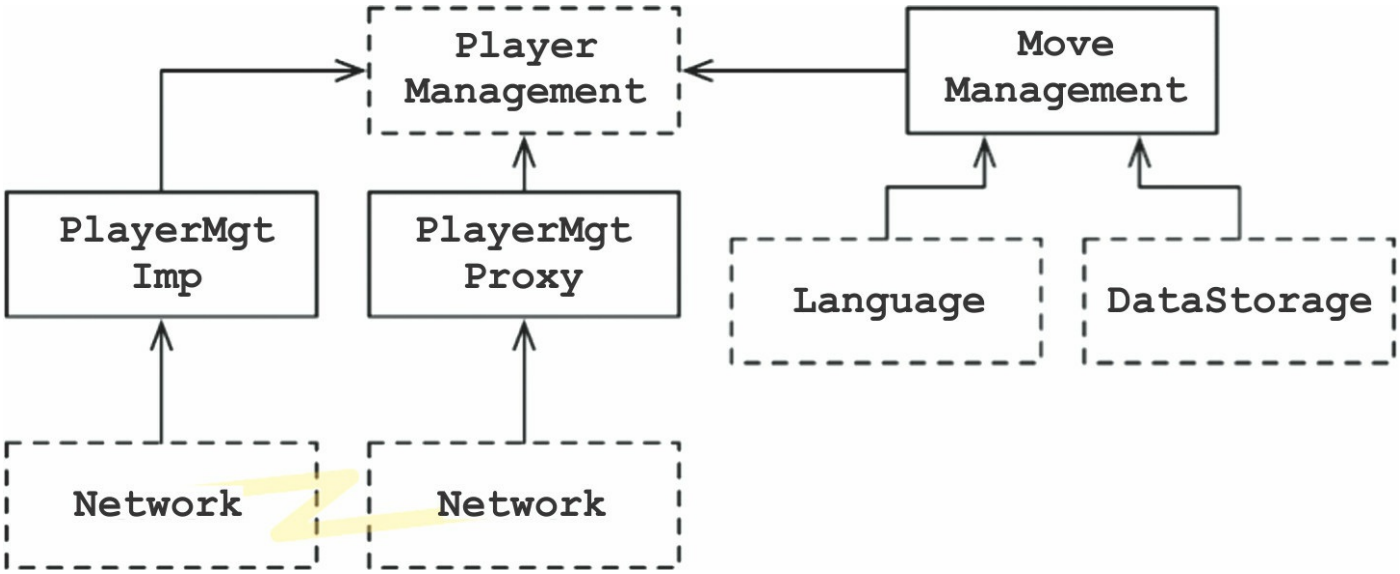


Abb. 25.7: Ergänzung einer Microservice-API

25.5 Fazit

Was hat das alles eigentlich zu bedeuten? Warum habe ich ein absurd einfaches Programm, das mit 200 Kornshell-Codezeilen implementiert werden kann, mit all diesen verrückten architektonischen Grenzen extrapoliert?

Dieses Beispiel soll demonstrieren, dass es buchstäblich überall architektonische Grenzen gibt – und als Softwarearchitekt müssen Sie mit aller gebotenen Sorgfalt prüfen, wann sie benötigt werden. Darüber hinaus sollten Sie sich auch darüber im Klaren sein, dass diese Grenzen, sobald sie vollständig implementiert werden, kostenintensiv sind.

Gleichzeitig müssen Sie aber auch in Ihre Überlegungen mit einbeziehen, dass es ein noch kostenintensiveres Unterfangen wäre, diese Grenzen zunächst zu ignorieren und erst später zu ergänzen – selbst im Rahmen von umfangreichen Testsuiten und Refactoring-Maßnahmen.

Also was tun die Softwarearchitekten? Die Antwort auf diese Frage fällt leider etwas unbefriedigend aus: Einerseits haben uns sehr kluge Leute über die Jahre hinweg eingebläut, dass wir keine Notwendigkeit für Abstraktionen unterstellen sollten. Das ist die Philosophie des YAGNI-Prinzips: »Du wirst es nicht brauchen.« Es steckt viel Wahrheit in dieser Aussage, denn das Overengineering ist im Vergleich zum Underengineering tatsächlich häufig das größere Übel. Wenn Sie jedoch andererseits feststellen, dass Sie wirklich eine architektonische Grenze ziehen müssten, wo keine ist, dann können die Kosten und Risiken für deren nachträgliche Ergänzung sehr hoch sein.

So sieht es also aus: Als Softwarearchitekt müssen Sie in die Zukunft schauen können! Sie müssen wohldurchdachte, intelligente Einschätzungen vornehmen. Und Sie müssen die Kosten abwägen und feststellen, wo die architektonischen Grenzen liegen, welche von ihnen vollständig umgesetzt, welche teilweise umgesetzt und welche ignoriert werden sollten.

Allerdings handelt es sich hierbei keineswegs um einmalige Entscheidungen. Es ist nicht damit getan, einfach nur zu Beginn eines Projekts zu bestimmen, welche Grenzen implementiert und welche ignoriert werden sollen. Vielmehr müssen Sie ständig weiter *beobachten*, wie sich das System entwickelt. Stellen Sie fest, wo gegebenenfalls weitere Grenzen benötigt werden, und achten Sie sorgfältig auf jedes Anzeichen von aufgrund fehlender Grenzen entstehenden Reibungspunkten.

An diesem Punkt wägen Sie dann die Kosten der Implementierung dieser Grenzen gegen die Kosten ab, die entstehen würden, wenn Sie sie ignorierten – und stellen Sie Ihre zuvor getroffenen Entscheidungen regelmäßig auf den Prüfstand. Ihr Ziel sollte stets sein, die Grenzen genau an dem Wendepunkt zu implementieren, an dem die Implementierungskosten die Kosten des Ignorierens unterschreiten.

Und dafür braucht es ein wachsames Auge.

---

[1] Ebenso steht aber auch außer Frage, dass das Konzept der sauberen Architektur nicht auf etwas so Triviales wie dieses Spiel angewendet werden würde – immerhin besteht das ganze Programm nur aus höchstens 200 Codezeilen. In diesem Beispielfall wird lediglich ein einfaches Programm stellvertretend für ein weit größeres System mit relevanten architektonischen Grenzen verwendet.

[2] Wenn Sie sich über die Ausrichtung der Pfeile wundern, bedenken Sie bitte, dass sie stets die Quellcode-Abhängigkeiten ausweisen und nicht die Richtung des Datenflusses.

[3] Früher wurde die Komponente an der Spitze eines Datenstroms als »Zentrale Transformation« bezeichnet. Siehe *The Practical Guide to Structured Systems Design*, Meilir Page-Jones, 2. Auflage, Yourdon Press, 1988.

Kapitel 26
Die Komponente Main

In jedem System findet sich wenigstens eine Komponente, die alle übrigen erstellt, koordiniert und überwacht. Ich nenne diese Komponente üblicherweise `Main`.

26.1 Das ultimative Detail

Die `Main`-Komponente repräsentiert das ultimative Detail eines Systems – die tiefschichtigste untergeordnete Richtlinie. Sie stellt den initialen Eintrittspunkt in das System dar. Abgesehen von dem Betriebssystem bestehen ansonsten keinerlei Abhängigkeiten von dieser Komponente. Ihre Aufgabe besteht darin, alle Factories, Strategien sowie andere globale Instanzen zu erstellen und diese dann der Kontrolle der übergeordneten abstrakten Systembereiche zu überantworten.

Sämtliche vom *Dependency-Injection-Framework* reglementierten Abhängigkeiten sollten in ebendiese `Main`-Komponente »injiziert« werden. Und anschließend sollte `Main` diese Abhängigkeiten ohne Verwendung des Frameworks normal weiterverteilen.

Stellen Sie sich die Komponente `Main` am besten als die unsauberste der unsauberen Komponenten vor.

Die nachfolgend vorgestellte `Main`-Komponente entstammt einer aktuellen Version des testbasierten Adventurespiels »Hunt the Wumpus«. Beachten Sie hier insbesondere, wie all die Strings geladen werden, von denen der Hauptteil des Codes keine Kenntnis haben soll.

```
public class Main implements HtwMessageReceiver {
   private static HuntTheWumpus game;
   private static int hitPoints = 10;
   private static final List<String> caverns = new ArrayList<>();
   private static final String[] environments = new String[]{
      "bright",
      "humid",
      "dry",
      "creepy",
      "ugly",
      "foggy",
      "hot",
      "cold",
      "drafty",
      "dreadful"
   };

   private static final String[] shapes = new String[] {
      "round",
      "square",
      "oval",
      "irregular",
      "long",
      "craggy",
      "rough",
      "tall",
      "narrow"
   };
```

```
   private static final String[] cavernTypes = new String[] {
      "cavern",
      "room",
      "chamber",
      "catacomb",
      "crevasse",
      "cell",
      "tunnel",
      "passageway",
      "hall",
      "expanse"
   };

   private static final String[] adornments = new String[] {
      "smelling of sulfur",
      "with engravings on the walls",
      "with a bumpy floor",
      "",
      "littered with garbage",
      "spattered with guano",
      "with piles of Wumpus droppings",
      "with bones scattered around",
      "with a corpse on the floor",
      "that seems to vibrate",
      "that feels stuffy",
      "that fills you with dread"
   };
```

Als Nächstes sehen Sie den Aufbau der Funktion `Main`. Beachten Sie in diesem Fall, wie `HtwFactory` zur Erstellung des Spiels genutzt wird. Die Funktion übergibt den Namen der Klasse, `htw.game.HuntTheWumpusFacade`, weil diese noch unsauberer ist als `Main` selbst. Dadurch wird verhindert, dass Modifikationen an dieser Klasse eine Neukompilierung bzw. das erneute Deployment von `Main` zur Folge haben.

```
public static void main(String[] args) throws IOException {
   game = HtwFactory.makeGame("htw.game.HuntTheWumpusFacade", new
Main());
   createMap();
   BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
   game.makeRestCommand().execute();
   while (true) {
      System.out.println(game.getPlayerCavern());
      System.out.println("Health: " + hitPoints + " arrows: " +
game.getQuiver());
      HuntTheWumpus.Command c = game.makeRestCommand();
      System.out.println(">");
      String command = br.readLine();
      if (command.equalsIgnoreCase("e"))
         c = game.makeMoveCommand(EAST);
      else if (command.equalsIgnoreCase("w"))
         c = game.makeMoveCommand(WEST);
      else if (command.equalsIgnoreCase("n"))
```

```
            c = game.makeMoveCommand(NORTH);
         else if (command.equalsIgnoreCase("s"))
            c = game.makeMoveCommand(SOUTH);
         else if (command.equalsIgnoreCase("r"))
            c = game.makeRestCommand();
         else if (command.equalsIgnoreCase("sw"))
            c = game.makeShootCommand(WEST);
         else if (command.equalsIgnoreCase("se"))
            c = game.makeShootCommand(EAST);
         else if (command.equalsIgnoreCase("sn"))
            c = game.makeShootCommand(NORTH);
         else if (command.equalsIgnoreCase("ss"))
            c = game.makeShootCommand(SOUTH);
         else if (command.equalsIgnoreCase("q"))
            return;
         c.execute();
      }
   }
```

Beachten Sie auch, dass `Main` den Eingabestream erzeugt und die Hauptschleife des Spiels enthält, indem es die einfachen Eingabebefehle interpretiert, die gesamte Verarbeitung dann anschließend jedoch anderen, hochschichtigeren Komponenten überlässt.

Und jetzt werfen Sie zum Schluss noch einen Blick darauf, wie `Main` die Karte generiert.

```
private static void createMap() {
   int nCaverns = (int) (Math.random() * 30.0 + 10.0);
   while (nCaverns-- > 0)
      caverns.add(makeName());

   for (String cavern : caverns) {
      maybeConnectCavern(cavern, NORTH);
      maybeConnectCavern(cavern, SOUTH);
      maybeConnectCavern(cavern, EAST);
      maybeConnectCavern(cavern, WEST);
   }

      String playerCavern = anyCavern();
      game.setPlayerCavern(playerCavern);
      game.setWumpusCavern(anyOther(playerCavern));
      game.addBatCavern(anyOther(playerCavern));
      game.addBatCavern(anyOther(playerCavern));
      game.addBatCavern(anyOther(playerCavern));

      game.addPitCavern(anyOther(playerCavern));
      game.addPitCavern(anyOther(playerCavern));
      game.addPitCavern(anyOther(playerCavern));

      game.setQuiver(5);
   }
```

```
    // Es wurde viel Code entfernt ...
}
```

Der springende Punkt ist hierbei, dass `Main` ein unsauberes untergeordnetes Modul im äußersten Kreis der sauberen Architektur ist. Es lädt alle notwendigen Ressourcen für das hochschichtige System und überstellt es dann seiner Kontrolle.

26.2 Fazit

Stellen Sie sich `Main` am besten als Plug-in für die Anwendung vor – ein Plug-in, das die anfänglichen Zustände und Konfigurationen einrichtet, alle externen Ressourcen sammelt und dann die Kontrolle darüber an die übergeordnete Richtlinie der Anwendung überträgt. Aufgrund seiner Eigenschaft als Plug-in ist es auch möglich, mehrere `Main`-Komponenten vorzuhalten – eine für jede Konfiguration Ihrer Anwendung.

So könnten Sie zum Beispiel ein `Main`-Plug-in für *Dev*, ein weiteres für *Test* und wieder ein anderes für *Production* einrichten. Ebenso gut könnten Sie auch jeweils ein `Main`-Plug-in für jedes Land, in dem die Anwendung deployt werden soll, für jeden Zuständigkeitsbereich oder für jeden Kunden verwenden.

Wenn Sie `Main` als eine Plug-in-Komponente betrachten, die sich hinter einer architektonischen Grenze befindet, werden sich die Probleme, die sich im Zusammenhang mit der Konfiguration ergeben, sehr viel leichter lösen lassen.

Kapitel 27
Services – große und kleine

Sowohl serviceorientierte als auch Microservice-»Architekturen« sind in der jüngeren Vergangenheit immer populärer geworden. Die Gründe für ihre gegenwärtige Beliebtheit sind unter anderem in folgenden vorgeblichen Vorteilen zu suchen:

- *Services scheinen stark voneinander entkoppelt zu sein.* Wie Sie im Folgenden sehen werden, trifft dies allerdings nur zum Teil zu.

- *Services scheinen eine weitreichendere Unabhängigkeit hinsichtlich der Entwicklung und des Deployments eines Systems zu fördern.* Auch was diesen Punkt angeht, wird sich nachstehend noch zeigen, dass er nur der halben Wahrheit entspricht.

27.1 Servicearchitektur?

Als Erstes soll an dieser Stelle jedoch auf die Annahme eingegangen werden, die Verwendung von Services stelle schon von Haus aus eine Systemarchitektur dar. Diese Mutmaßung ist eindeutig falsch! Die Architektur eines Systems wird durch Grenzlinien definiert, die übergeordnete Richtlinien von untergeordneten Details

separieren und den Abhängigkeitsregeln entsprechen. Services, die einfach nur die Verhaltensweisen einer Anwendung voneinander abgrenzen, sind hingegen kaum mehr als kostenintensive Funktionsaufrufe und deshalb auch nicht unbedingt architekturrelevant.

Das soll allerdings nicht heißen, dass alle Services architekturrelevant sein *sollten*. Die Bereitstellung von Services, die die Funktionalität über Prozesse und Plattformen hinweg voneinander trennen, bringt oftmals erhebliche Vorteile mit sich – ob sie nun der Abhängigkeitsregel folgen oder nicht. Allerdings definieren Services an sich noch keine Architektur.

Eine hilfreiche Analogie hierzu findet sich in der Organisation von Funktionen. Die Architektur eines monolithischen oder komponentenbasierten Systems wird durch bestimmte Funktionsaufrufe definiert, die architektonische Grenzen überschreiten und der Abhängigkeitsregel folgen. Viele andere Funktionen in diesen Systemen separieren jedoch lediglich Verhaltensweisen voneinander und sind somit aus architektonischer Sicht nicht relevant.

So ist das mit den Services. Letztlich handelt es sich bloß um prozess- und plattformübergreifende Funktionsaufrufe. Einige dieser Services sind im architektonischen Kontext bedeutsam, andere sind es nicht – und dieses Kapitel wird sich mit Ersteren befassen.

27.2 Vorteile der Services?

Die Formulierung dieser Überschrift als Frage deutet bereits darauf hin, dass im nachfolgenden Abschnitt einige Aspekte der zurzeit allgegenwärtigen Orthodoxie der Servicearchitektur infrage gestellt werden. Analysieren wir also den Nutzen der Services der Reihe nach.

27.2.1 Denkfalle: Entkopplung

Einer der mutmaßlichen großen Vorteile der Unterteilung eines Systems in Services besteht darin, dass Letztere strikt voneinander entkoppelt sind. Immerhin wird jeder Service in einem anderen Prozess betrieben oder sogar auf einem anderen Prozessor – und deshalb haben sie auch keinen Zugriff auf die Variablen der jeweils anderen Services. Zudem muss jede Serviceschnittstelle klar definiert sein.

Das ist sicherlich in gewisser Weise richtig – trifft aber längst nicht in jedem Fall zu. Ja, Services sind auf der Ebene der einzelnen Variablen entkoppelt. Dennoch können sie durch gemeinsam genutzte Ressourcen in einem Prozessor oder im Netzwerk

gekoppelt sein. Und auch durch die gemeinsam genutzten Daten entsteht eine starke Kopplung.

Wird zum Beispiel ein Datensatz, der zwischen den Services transferiert wird, um ein neues Datenfeld erweitert, dann muss jeder für dieses neue Feld ausgeführte Service dementsprechend angepasst werden. Darüber hinaus müssen die Services auch bei der Interpretation der Felddaten weitestgehend aufeinander abgestimmt sein. Insofern sind die Services allesamt nicht nur stark an den Datensatz, sondern indirekt auch miteinander gekoppelt.

Das gilt zum einen sicherlich für klar definierte Schnittstellen – zum anderen aber nicht minder auch für Funktionen. Serviceschnittstellen sind nicht in irgendeiner Weise formeller oder strenger oder besser definiert als Funktionsschnittstellen. Dieser vermeintliche Vorteil ist also offensichtlich illusorisch.

27.2.2 Denkfalle: Unabhängige Entwickel- und Deploybarkeit

Ein weiterer vorgeblicher Vorteil der Services ist, dass sie der Eigentümerschaft und Verwaltung eines bestimmten Teams überantwortet werden können. Damit ist dann ein Team im Rahmen einer DevOps-Strategie (**Dev**elopment and IT-**Op**eration**s**) für die Weiterentwicklung, die Instandhaltung und den Betrieb des Services verantwortlich. Die damit einhergehende Unabhängigkeit in Bezug auf die Entwicklung und das Deployment wird als *skalierbar* angenommen. Und dies führt wiederum zu der Annahme, dass große Unternehmenssysteme aus Dutzenden, Hunderten oder gar Tausenden von unabhängig entwickelbaren und deploybaren Services erstellt werden können, die sich hinsichtlich Entwicklung, Instandhaltung und Betrieb auf eine vergleichbare Anzahl von unabhängig arbeitenden Teams aufteilen lassen.

Nun mag diese Annahme zum Teil auch begründet sein – aber eben nur zum Teil. Erstens hat sich in der Vergangenheit gezeigt, dass große Unternehmenssysteme sowohl aus Monolithen und komponentenbasierten Systemen als auch aus servicebasierten Systemen erstellt werden können. Insofern sind Services nicht die einzige Option für die Errichtung skalierbarer Systeme.

Und zweitens gilt für die Denkfalle der Entkopplung, dass Services nicht immer unabhängig entwickelt, deployt und betrieben werden können – denn die Entwicklung, das Deployment und der Betrieb des Systems müssen trotzdem in dem Ausmaß, in dem sie durch Daten oder Verhalten gekoppelt sind, koordiniert werden.

27.3 Das Kätzchen-Problem

Als Beispiel für die beiden vorgenannten Denkfallen soll an dieser Stelle noch einmal das in Kapitel 9 erwähnte Aggregatorsystem für Taxidienste herhalten. Wie Sie sich erinnern werden, verwaltet dieses System mehrere Taxizentraldienste in einer Stadt und ermöglicht den Kunden die bequeme Buchung von Beförderungsfahrten. In diesem Beispielfall unterstellen wir einmal, dass die Kunden die Taxis auf der Grundlage verschiedener Kriterien auswählen können, wie etwa der Abholzeit, den Fahrtkosten, dem Komfort und der Erfahrung des Fahrers.

Das System soll skalierbar sein, deshalb entscheiden Sie sich, es aus vielen kleinen Microservices zusammenzubauen. Die Softwareentwickler werden in mehrere kleine Teams aufgeteilt, die jeweils die Verantwortung für die Entwicklung, die Instandhaltung und den Betrieb einer entsprechend kleinen Anzahl von Services tragen.[1]

Das Diagramm in Abbildung 27.1 zeigt eine fiktive Anordnung der Services zur Implementierung dieser Anwendung. Der Service `TaxiUI` befasst sich mit den Kunden bzw. Usern, die ihre Beförderungsanfragen per Mobilgerät übermitteln. Der `TaxiFinder`-Service überprüft die Bestände der verschiedenen Taxidienste (`TaxiSupplier`) und ermittelt, welche Taxis mögliche Kandidaten (`CandidateTaxis`) für den betreffenden Kunden sind. Anschließend hinterlegt er die geeigneten Anwärter in einem Kurzzeitdatensatz, der dem anfordernden User zugeordnet wird. Der Service `TaxiSelector` übernimmt die Kriterien des Kunden hinsichtlich Kosten, Zeitpunkt, Komfort usw. und wählt ein passendes Taxi aus der Liste der Kandidaten aus. Dann übergibt er den ermittelten Wagen an den `TaxiDispatcher`-Service, der das betreffende Taxi schließlich ordert.

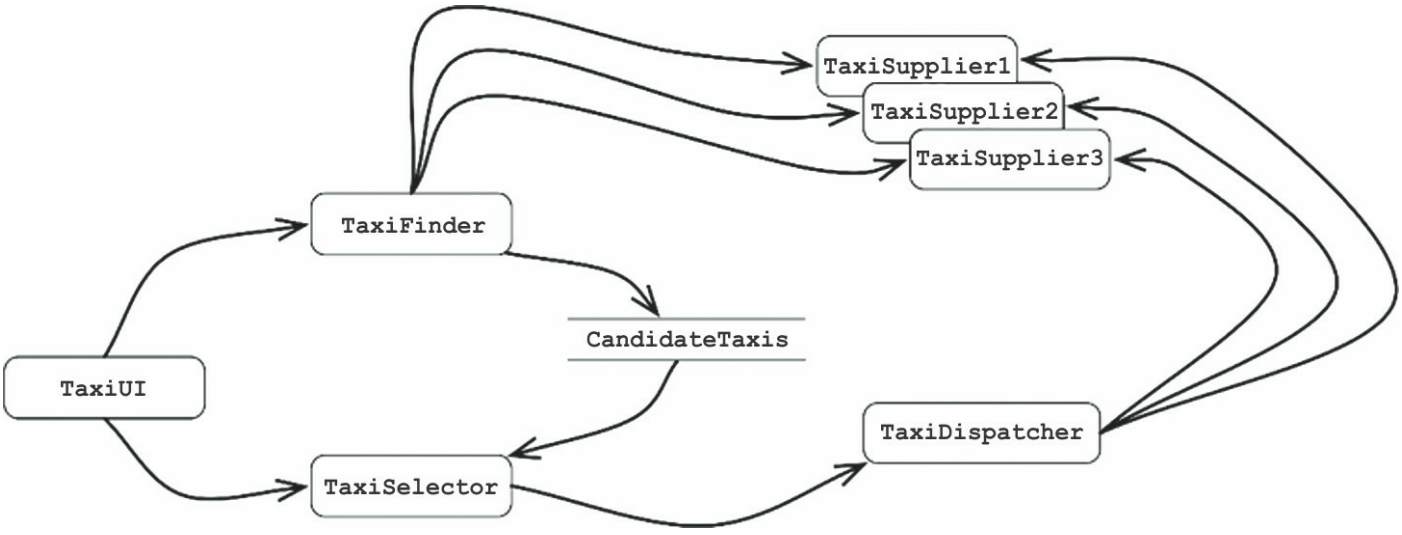


Abb. 27.1: Anordnung der Services zur Implementierung des Aggregatorsystems für Taxidienste

Unterstellen wir nun, dass das System bereits länger als ein Jahr in Betrieb ist und die Entwickler während des Betriebs und der Instandhaltung all dieser Services fröhlich

neue Features entwickelt haben.

Eines schönen Tages lädt dann die Marketingabteilung das Entwicklungsteam zu einem Meeting, in dem verkündet wird, dass ein Katzenlieferdienst für das Stadtgebiet eingerichtet werden soll. Die Kunden können also Kätzchen bestellen, die zu ihnen nach Hause oder an den Arbeitsplatz geliefert werden.

Das Unternehmen wird mehrere Katzenaufnahmestationen in der ganzen Stadt einrichten. Und sobald eine Katzenbestellung eingeht, wird ein in der Nähe einer Station befindliches Taxi ausgewählt, um ein Kätzchen von dort abzuholen und anschließend an die Kundenadresse zu liefern.

Einer der Taxidienste hat seine Teilnahme an diesem Programm bereits zugesagt, und es steht zu erwarten, dass ihm noch weitere folgen werden. Wieder andere könnten jedoch auch ablehnen.

Es ist natürlich möglich, dass der ein oder andere Fahrer allergisch auf Katzen reagiert und dementsprechend nicht für diesen Service zur Auswahl steht. Ebenso werden einige der Fahrgäste ähnliche Allergien haben, deshalb sollte ein Fahrzeug, das in den vorausgehenden drei Tagen für die Lieferung eines Kätzchens eingesetzt wurde, nicht für Kunden ausgewählt werden, die entsprechende Allergien angeben.

Schauen Sie sich das Diagramm der Services noch einmal an. Wie viele davon müssen zur Implementierung dieses Features modifiziert werden? Die richtige Antwort lautet: *Alle!* Es steht also außer Frage, dass die Entwicklung und das Deployment des Kätzchen-Features sehr sorgfältig koordiniert werden müssen.

Soll heißen: Die Services sind allesamt miteinander gekoppelt und können *nicht* unabhängig voneinander entwickelt, deployt und instand gehalten werden.

Und genau das ist das Problem mit den *Cross-Cutting Concerns* (zu Deutsch etwa »querschnittliche Belange«). Jedes Softwaresystem muss sich dieser Problematik stellen, ob es nun serviceorientiert ist oder nicht. Funktionale Dekompositionen der Art, wie sie in Abbildung 27.1 dargestellt sind, sind in Bezug auf die Einführung neuer Features, die sich über alle diese funktionalen Verhaltensweisen erstrecken, sehr anfällig.

27.4 Objekte als Rettung

Doch wie hätten wir dieses Problem in einer komponentenbasierten Systemarchitektur lösen können? Nun, die sorgfältige Einhaltung der SOLID-Designprinzipien würde uns veranlassen, einen Satz von polymorph erweiterbaren Klassen zu erstellen, um neue Features zu handhaben.

Das Diagramm in Abbildung 27.2 illustriert diese Strategie. Die Klassen stimmen grob mit den in Abbildung 27.1 gezeigten Services überein. Beachten Sie hier jedoch die Grenzlinien sowie die Ausrichtung der Abhängigkeiten, die der Abhängigkeitsregel folgen.

Der überwiegende Teil der den ursprünglichen Services zugrunde liegenden Logik bleibt mit den Basisklassen des Objektmodells erhalten. Allerdings wurde der für die *Fahrten* spezifische Teil nun in eine eigene Komponente `Rides` extrahiert. Und das neue Feature für die Kätzchen wurde ebenfalls in eine eigene Komponente `Kittens` platziert. Diese beiden Komponenten überschreiben die abstrakten Basisklassen in den ursprünglichen Komponenten mithilfe eines Design Patterns wie *Template Method (Schablonenmethode)* oder *Strategy (Strategie)*.

Beachten Sie auch, dass die beiden neuen Komponenten `Rides` und `Kittens` der Abhängigkeitsregel entsprechen. Außerdem werden die Klassen, die diese Features implementieren, von Factories erzeugt, die der Kontrolle der Benutzerschnittstelle unterliegen.

Dieser schematische Aufbau lässt keinen Zweifel daran, dass `TaxiUI` modifiziert werden muss, sobald das Kätzchen-Feature implementiert wird – ansonsten sind aber keine weiteren Anpassungen erforderlich. Stattdessen wird eine neue `.jar`- oder `.gem`-Datei oder eine DLL in das System ergänzt und dynamisch zur Laufzeit geladen.

Somit ist das Kätzchen-Feature entkoppelt und unabhängig entwickel- und deploybar.

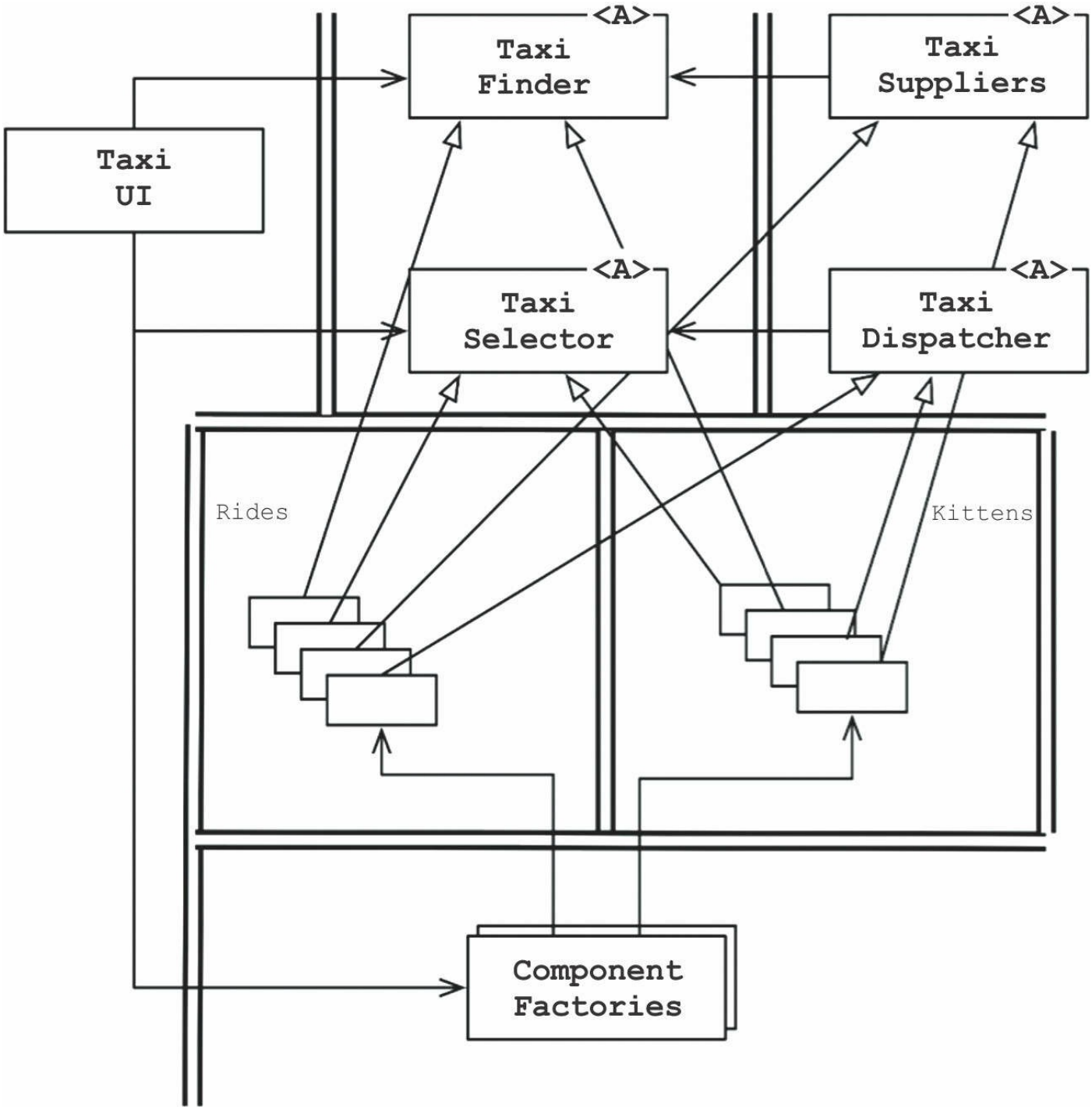


Abb. 27.2: Objektorientierter Ansatz für den Umgang mit Cross-Cutting Concerns

27.5 Komponentenbasierte Services

Die Frage, die sich hier offenkundig stellt, ist: Lässt sich das für die Services hinbekommen? Und die Antwort lautet natürlich: Ja! Services brauchen keine kleinen Monolithen zu sein. Vielmehr können sie unter Einhaltung der SOLID-Prinzipien entworfen und mit einer Komponentenstruktur ausgestattet werden, sodass sie ohne jegliche Modifikationen an den bereits im Service vorhandenen Komponenten um neue Komponenten ergänzt werden können.

Stellen Sie sich einen Service in Java als einen Satz abstrakter Klassen in einer oder mehreren `.jar`-Dateien vor – und jedes neue Feature oder jede Feature-Erweiterung als eine weitere `.jar`-Datei, die ihrerseits Klassen enthält, die die abstrakten Klassen in den ersten `.jar`-Dateien erweitert. Das Deployment eines neuen Features ist dann keine Frage des erneuten Deployments der Services mehr, sondern erfordert lediglich die *Ergänzung* neuer `.jar`-Dateien zum Laden der Verzeichnisse dieser Services. Mit anderen Worten: Das Hinzufügen neuer Features erfolgt in Konformität mit dem *Open-Closed-Prinzip*.

Das Diagramm in Abbildung 27.3 illustriert die Struktur. Die Services sind nach wie vor vorhanden, allerdings hat nun jeder von ihnen sein eigenes internes Komponentendesign, das die Ergänzung neuer Features in Form neuer abgeleiteter Klassen ermöglicht. Diese Derivate existieren jeweils innerhalb ihrer eigenen Komponenten.

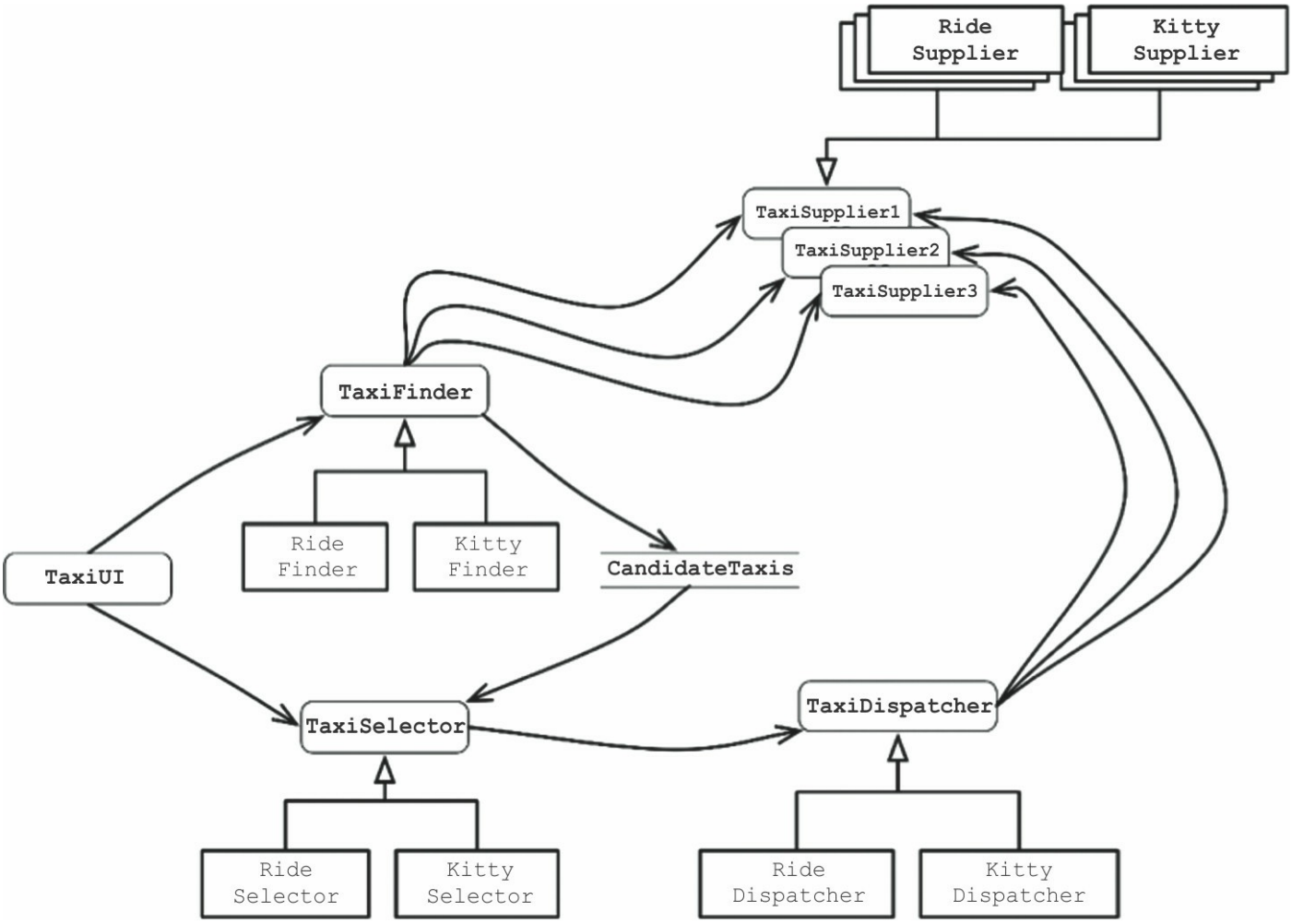


Abb. 27.3: Jeder Service hat sein eigenes internes Komponentendesign, das die Ergänzung neuer Features als neue abgeleitete Klassen ermöglicht.

27.6 Cross-Cutting Concerns

Wie Sie erfahren haben, befinden sich die architektonischen Grenzlinien nicht *zwischen* den Services, sondern verlaufen *durch* die Services und teilen sie in Komponenten auf.

Für die Handhabung der *Cross-Cutting Concerns*, mit denen alle wichtigen Softwaresysteme konfrontiert sind, müssen die Services mit internen Komponentenarchitekturen entworfen werden, die der Abhängigkeitsregel entsprechen, wie in Abbildung 27.4 dargestellt. Diese Services definieren nicht die architektonischen Grenzen in dem System – das tun die Komponenten innerhalb der Services.

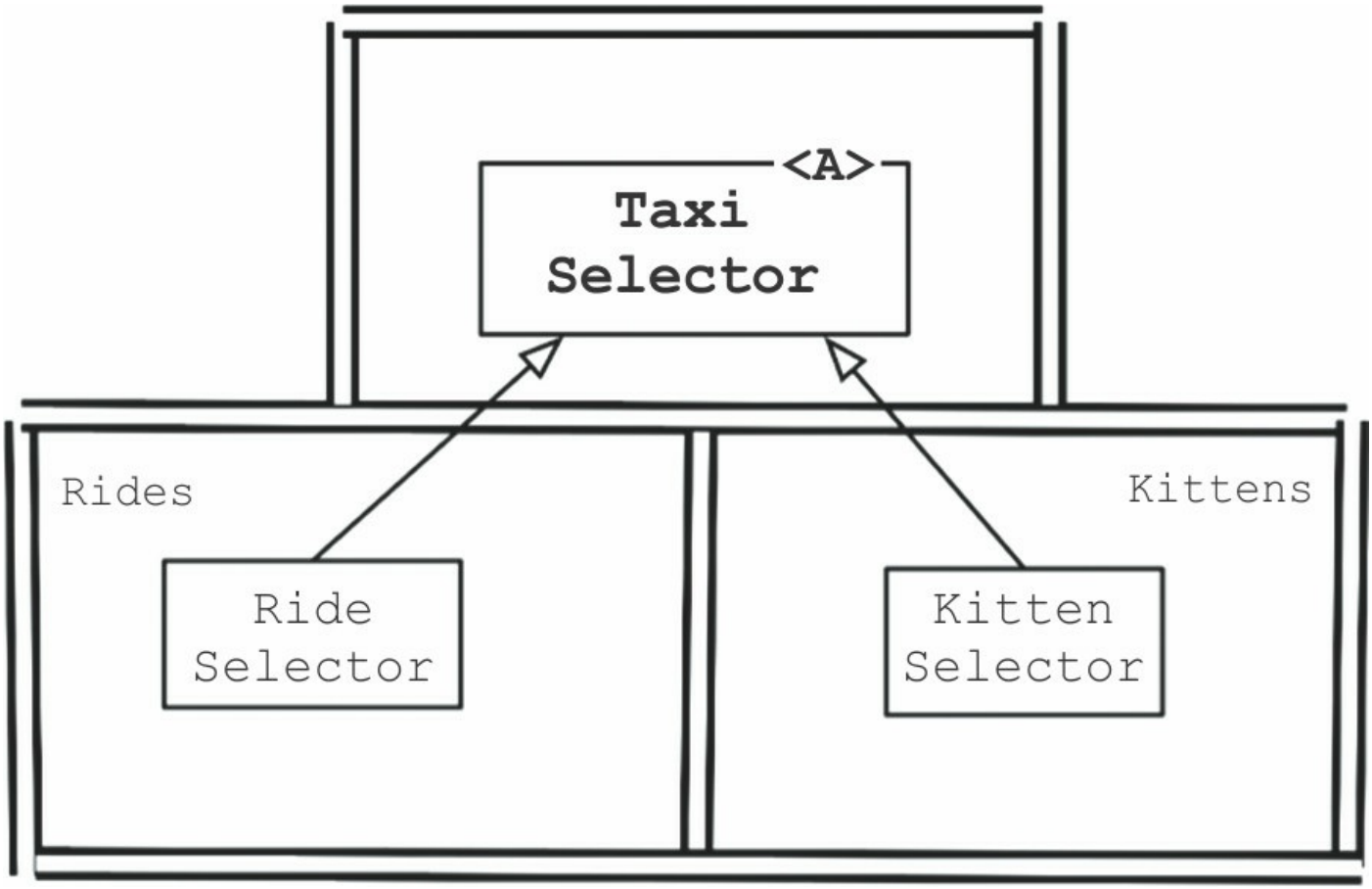


Abb. 27.4: Services müssen mit internen Komponentenarchitekturen entworfen werden, die der Abhängigkeitsregel folgen.

27.7 Fazit

So nützlich wie Services für die Skalierbarkeit und Entwickelbarkeit eines Systems auch sind, stellen sie für sich genommen trotzdem keine architektonisch relevanten Elemente dar. Die Systemarchitektur wird durch die innerhalb dieses Systems gezogenen Grenzen definiert sowie durch die Abhängigkeiten, die diese Grenzlinien überschreiten – nicht jedoch durch die physischen Mechanismen, über die Elemente kommunizieren und ausgeführt werden.

Ein Service kann eine einzelne Komponente sein, die vollständig von einer architektonischen Grenze umgeben ist. Alternativ kann er auch aus mehreren Komponenten bestehen, die durch architektonische Grenzen voneinander getrennt sind. In seltenen Fällen[2] können Clients und Services zudem so miteinander gekoppelt sein, dass sie überhaupt keine architektonische Relevanz aufweisen.

---

[1] Die Anzahl der Microservices würde somit in etwa der Anzahl der Programmierer entsprechen.

[2] Zumindest steht zu hoffen, dass diese Fälle selten sind. Leider zeigt die Erfahrung, dass sie doch häufiger vorkommen, als sie sollten.

Kapitel 28
Die Testgrenze

Ja, Sie haben richtig gelesen: *Tests sind ein Bestandteil des Systems* – und sie partizipieren wie jeder andere Bestandteil des Systems auch an der Softwarearchitektur. In gewisser Hinsicht ist diese Beteiligung völlig normal, in anderer Hinsicht kann sie allerdings ziemlich einzigartig sein.

28.1 Tests als Systemkomponenten

In Bezug auf die Systemtests herrscht oftmals große Verwirrung. Sind sie Bestandteil des Systems? Sind sie gesondert vom System zu sehen? Welche Arten von Tests gibt es? Sind Komponententests und Integrationstests verschiedene Dinge? Und wie steht es mit Akzeptanztests, Funktionstests, Cucumber-Tests, TDD-Tests (**T**est **D**riven **D**evelopment, testgetriebene Entwicklung), BDD-Tests (**B**ehavior **D**riven **D**evelopment, verhaltensgetriebene Softwareentwicklung), Komponententests usw.?

Sinn und Zweck dieses Buches ist nicht, in die spezielle Debatte zu diesem Thema einzugreifen – und glücklicherweise ist das auch gar nicht notwendig. Aus architektonischer Sicht kommt allen Tests die gleiche Bedeutung zu. Ob es sich nun um kleinere TDD-Tests handelt oder um umfangreichere Tests wie FitNesse, Cucumber, SpecFlow oder JBehave: Sie alle sind die Architektur betreffend gleichwertig.

Ihrem Wesen nach folgen die Tests der Abhängigkeitsregel: Sie sind sehr detailliert und konkret, und ihre Abhängigkeitsverhältnisse sind immer nach innen auf den zu testenden Code ausgerichtet. Bildlich ausgedrückt kann man sie sich als den äußersten Kreis der Softwarearchitektur vorstellen. Es sind keine Systemkomponenten von den Tests abhängig, während sie selbst immer nach innen gerichtet von den Komponenten des Systems abhängig sind.

Darüber hinaus sind sie auch unabhängig deploybar. Faktisch werden sie meist eher in Testsystemen deployt als in Produktionssystemen. Das bedeutet: Selbst in Systemen, in denen ansonsten gar kein unabhängiges Deployment erforderlich ist, werden sie trotzdem unabhängig deployt.

Die Tests sind die am stärksten isolierte Systemkomponente. Sie sind für den Systembetrieb nicht zwingend erforderlich. Kein User ist von ihnen abhängig. Ihre Aufgabe besteht darin, die Entwicklungsarbeit zu unterstützen, nicht den Betrieb des Systems. Und doch sind die Tests genauso eine Systemkomponente wie jede andere. Tatsächlich verkörpern sie in vielerlei Hinsicht sogar ein Modell, dem alle anderen Systemkomponenten folgen sollten.

28.2 Design für Testfähigkeit

Die extreme Isolation der Tests gepaart mit dem Umstand, dass sie normalerweise nicht deployt werden, verleitet Softwareentwickler häufig dazu, anzunehmen, dass sie außerhalb des Systemdesigns zu sehen sind. Diese Annahme ist jedoch ganz und gar unzutreffend. Tests, die nicht gut in das Design des Systems integriert sind, neigen zu Fragilität und tragen dazu bei, dass das System unflexibel und schwer modifizierbar wird.

Die Hauptursache dieses Problems ist natürlich die Kopplung: Tests, die stark an das System geknüpft sind, müssen sich *mit* dem System ändern. Selbst die geringfügigste Modifikation an einer Systemkomponente kann dazu führen, dass viele gekoppelte Tests nicht funktionieren oder Anpassungen erfordern.

Und das kann sich zu einer akuten Situation auswachsen. Änderungen an allgemeinen Systemkomponenten können zur Folge haben, dass Hunderte oder sogar Tausende von

Testfehlern auftreten. Dieses Phänomen ist unter der Bezeichnung *Fragile Tests Problem* (zu Deutsch etwa »Problematik fragiler Tests«) bekannt.

Wie es dazu kommen kann, ist unschwer zu erkennen. Denken Sie zum Beispiel an eine Testsuite, die zur Verifizierung der Geschäftsregeln auf das GUI zurückgreift. Solche Tests können im Anmeldebildschirm anfangen und dann durch die Seitenstruktur navigieren, bis sie bestimmte Geschäftsregeln prüfen können. Jede Modifikation am Anmeldebildschirm oder der Navigationsstruktur könnte somit eine enorme Anzahl von Testfehlern hervorrufen.

Fragile Tests haben oft den paradoxen Effekt, das System starr und unflexibel zu machen. Wenn Softwareentwickler erkennen, dass einfache Änderungen am System massive Testfehler verursachen, dann könnte sie das davon abhalten, diese Modifikationen überhaupt vorzunehmen. Stellen Sie sich beispielsweise nur einmal die Diskussion zwischen dem Entwicklungsteam und einem Marketingteam vor, das eine einfache Anpassung an der Seitennavigationsstruktur angefordert hat, die in der Folge jedoch 1.000 Testfehler hervorruft.

Die Lösung ist hier ein auf Testfähigkeit ausgerichtetes Design. Die erste Regel des Softwaredesigns – ob nun zum Zweck der Testfähigkeit oder zu irgendeinem anderen Zweck – bleibt immer dieselbe: *Erzeugen Sie keine Abhängigkeiten von flüchtigen Komponenten.* GUIs sind flüchtige Komponenten. Testsuiten, die das System über das GUI betreiben, *müssen fragil sein*. Gestalten Sie daher sowohl das System als auch die Tests so, dass die Geschäftsregeln ohne den Einsatz des GUIs getestet werden können.

28.3 Die Test-API

Um dieses Ziel zu erreichen, sollten Sie eine spezifische API erzeugen, die Tests zur Verifizierung der Geschäftsregeln nutzen kann. Diese API sollte über »Superkräfte« verfügen, die es den Tests ermöglichen, Sicherheitsbeschränkungen zu vermeiden, kostenintensive Ressourcen (wie Datenbanken) zu umgehen und das System in bestimmte testfähige Zustände zu zwingen. Eine solche API dient als Obermenge der *Interaktoren* und *Schnittstellenadapter*, die vom UI verwendet werden.

Der Zweck der Test-API ist die Entkopplung der Tests von der Anwendung. Diese Separierung umfasst allerdings etwas mehr als nur das Lösen der Tests vom UI: Sie zielt darauf ab, die Struktur der Tests von der *Struktur* der Anwendung zu entkoppeln.

28.3.1 Strukturelle Kopplung

Die strukturelle Kopplung ist eine der stärksten und heimtückischsten Formen der

Testkopplung. Stellen Sie sich eine Testsuite vor, die eine Testklasse für jede Produktionsklasse und einen Satz von Testmethoden für jede Produktionsmethode enthält. Eine solche Testsuite ist eng mit der Struktur der Anwendung gekoppelt.

Sobald sich eine der besagten Produktionsmethoden oder -klassen ändert, muss auch eine große Anzahl von Tests entsprechend angepasst werden – und damit sind diese Tests fragil und machen den Produktionscode starr und unflexibel.

Die Aufgabe der Test-API besteht darin, die Struktur der Anwendung vor den Tests zu verbergen. Dadurch ist die Anpassung und Weiterentwicklung des Produktionscodes möglich, ohne dass die Tests davon beeinflusst werden. Zudem können umgekehrt auch die Tests umgestaltet und weiterentwickelt werden, ohne dass dies Einfluss auf den Produktionscode hat.

Diese Art von separierter Softwareentwicklung ist deshalb notwendig, weil die Tests im Laufe der Zeit immer konkreter und spezifischer werden, während der Produktionscode im Gegensatz dazu eher dahin tendiert, immer abstrakter und allgemeingültiger zu werden. Eine starke strukturelle Kopplung verhindert diese notwendige Entwicklung oder beeinträchtigt sie zumindest derart, dass die weitestgehend allgemeine und flexible Gestaltung des Produktionscodes nicht mehr möglich ist.

28.3.2 Sicherheit

Der Einsatz der »Superkräfte« der Test-API kann im Falle des Deployments in einem Produktionssystem riskant sein. Sollte dieser Risikofaktor bestehen, dann sollten die Test-API bzw. die risikoträchtigen Bestandteile ihrer Implementierung in einer separaten, unabhängig deploybaren Komponente gehalten werden.

28.4 Fazit

Tests sind nicht außerhalb des Softwaresystems zu sehen – vielmehr sind sie Bestandteile des Systems, die ordentlich konzipiert werden müssen, wenn sie die gewünschten Vorteile hinsichtlich Stabilität und Regression bieten sollen. Tests, die nicht als Bestandteil des Systems entworfen wurden, sind in der Regel fragil und schwer instand zu halten. Solche Tests fallen oft durch das Raster der Instandhaltung – und werden am Ende ganz verworfen, weil sie schlicht zu schwierig zu warten sind.

Kapitel 29
Saubere eingebettete Architektur

Vor einiger Zeit stieß ich in dem Blog des Informatik-Professors Doug Schmidt auf einen Artikel mit dem Titel *»The Growing Importance of Sustaining Software for the DoD«* (zu Deutsch etwa »Die zunehmende Bedeutung nachhaltiger Software für das US-Verteidigungsministerium«).[1] Darin macht Doug Schmidt folgende Feststellung:

> *Wenngleich sich Software in aller Regel nicht abnutzt, so gilt für Firmware und Hardware jedoch, dass sie sich selbst überleben und veralten, was in der Konsequenz dazu führt, dass Anpassungen der Software notwendig werden.*

Das war ein erhellender Moment für mich. Doug Schmidt führte hier zwei Tatsachen an, die im Grunde genommen offensichtlich sein sollten – aber vielleicht sind sie das ja gar nicht: *Software* kann eine lange, produktive Lebensdauer haben, wohingegen *Firmware* in dem Maße veraltet, in dem sich die Hardware weiterentwickelt. Wenn Sie sich schon mal ein wenig mit der Entwicklung von eingebetteten Systemen (auch als *Embedded Systems* bezeichnet) befasst haben, dann wissen Sie, dass Hardware im Allgemeinen einem kontinuierlichen Neuerungs- und Verbesserungsprozess unterliegt.

Und parallel dazu wird neue »Software« ständig mit weiteren, zunehmend komplexeren Funktionen ausgestattet.

Insofern möchte ich Doug Schmidts Aussage ein wenig umformulieren:

> *Wenngleich sich Software in aller Regel nicht abnutzt, kann sie aufgrund von nicht verwalteten Firmware- und Hardwareabhängigkeiten dennoch aus sich selbst heraus unbrauchbar werden.*

Nicht selten bleibt eingebetteter Software eine potenziell lange Lebensdauer verwehrt, weil sie mit Abhängigkeitsverhältnissen zu der verwendeten Hardware »infiziert« ist.

Mir persönlich gefällt Doug Schmidts Firmware-Definition, es gibt aber durchaus noch andere Auslegungen, von denen ich Ihnen nachfolgend einmal einige zusammengestellt habe:

- »Unter Firmware (engl. firm, ›fest‹) versteht man Software, die in elektronischen Geräten eingebettet ist. Sie ist zumeist in einem Flash-Speicher, einem EPROM, EEPROM oder ROM gespeichert und durch den Anwender nicht oder nur mit speziellen Mitteln bzw. Funktionen austauschbar.« (Wikipedia, https://de.wikipedia.org/wiki/Firmware)

- »[Firmware ist die] Gesamtheit der zur Hardware eines Computers gehörenden, vom Hersteller auf Festwertspeicher abgelegten und vom Benutzer nicht veränderbaren Programme.« (Duden, https://www.duden.de/rechtschreibung/Firmware)

- »Unter dem Begriff Firmware versteht man die fest eingebaute Software in einem technischen Gerät, das zum Betrieb desselben erforderlich ist.« (IT-Service24, https://www.it-service24.com/lexikon/f/firmware/)

- Firmware ist eine Software, die dauerhaft in einem Festwertspeicher, einem Read Only Memory (ROM), Programmable Read Only Memory (PROM) oder Erasable PROM (EPROM) eines Computers gespeichert ist. Sie kann auch in einem Flash-Speicher abgelegt sein. (ITWissen.info, http://www.itwissen.info/Firmware-firmware.html)

Doug Schmidts Aussage machte mir bewusst, dass all diese gängigen und anerkannten Definitionen des Begriffs »Firmware« im Grunde genommen falsch sind – oder zumindest überholt. Firmware bedeutet nicht, dass Code im ROM existiert. Bei dieser speziellen Softwareausprägung geht es nicht etwa um die Frage, wo sie sich befindet, sondern vielmehr darum, wovon sie abhängig ist und inwieweit es möglich ist, sie an die Weiterentwicklungen der Hardware anzupassen. Hardware unterliegt einem ständigen, dem technischen Fortschritt geschuldeten Wandel (denken Sie nur mal an Ihr Handy), deshalb sollte eingebetteter Code auch immer vor dem Hintergrund dieser

Realität strukturiert werden.

Generell habe ich nichts gegen Firmware, geschweige denn Firmware-Entwickler einzuwenden (immerhin habe ich selbst auch schon Firmware geschrieben). Dennoch bin ich der Auffassung, dass wir tatsächlich weniger Firmware und mehr Software brauchen – und ehrlich gestanden finde ich es etwas enttäuschend, dass Programmierer so viel Firmware schreiben.

Sogar die Entwickler nicht eingebetteter Software fabrizieren Firmware! Im Prinzip schreiben wir alle Firmware, sobald wir SQL in unseren Code einbringen oder Plattformabhängigkeiten darin verankern. Auch die Entwickler von Android-Apps erzeugen Firmware, sofern sie ihre Geschäftslogik nicht klar von der Android-API trennen.

Ich selbst war an einer Reihe von Projekten beteiligt, bei denen die Grenze zwischen dem Produktcode (der Software) und dem Code, der mit der Hardware des Produkts (der Firmware) interagiert, bis zur Unkenntlichkeit verschwamm. Beispielsweise hatte ich in den späten 1990er-Jahren das Vergnügen, an dem Redesign eines Kommunikationssubsystems mitwirken zu dürfen, das vom Zeitmultiplexverfahren (*TDM*, **T**ime **D**ivision **M**ultiplexing) in VoIP (**V**oice-**o**ver-**IP**) überführt wurde. Inzwischen ist Voice-over-IP natürlich Standard, in den 1950er- und 1960er-Jahren wurde jedoch TDM als State of the Art betrachtet und erfuhr daher in den 1980er- und 1990er-Jahren weite Verbreitung.

Wann immer wir dem leitenden Systementwickler eine Frage in Bezug darauf stellten, wie eine Telefonverbindung auf eine bestimmte Situation reagieren sollte, verschwand er zunächst für eine Weile und kam dann etwas später mit einer detaillierten Antwort zurück. »Wo hast du plötzlich diese Informationen her?«, fragten wir. Woraufhin er antwortete: »Aus dem aktuellen Produktcode.« Wie sich herausstellte, hielt der verworrene Legacy-Spaghetticode als Spezifikation für das neue Produkt her! In der vorhandenen Implementierung existierte allerdings keine Trennung des TDMs und der Geschäftslogik für den Ablauf einer Telefonverbindung. Das Produkt war von vorn bis hinten hardware-/technologieabhängig und ließ sich auch nicht entwirren – somit war es vollumfänglich zu Firmware geworden.

Hier noch ein weiteres Beispiel: Befehlsnachrichten wurden über den seriellen Port an dieses System übermittelt. Selbstredend war natürlich auch ein Message Processor/Dispatcher vorhanden, der das Format der Nachrichten kannte und sie analysieren sowie anschließend an den Code senden konnte, der die Anfrage verarbeiten würde. Nichts davon war irgendwie ungewöhnlich, abgesehen davon, dass sich der Message Processor/Dispatcher in derselben Datei befand wie der Code, der mit einer UART2-Hardware[2] interagierte und demzufolge mit UART-Details durchsetzt war. Grundsätzlich hätte es sich hierbei um eine Software mit einer potenziell langen und nützlichen Lebenserwartung handeln können, stattdessen war er aber Firmware. Mit anderen Worten: Dem Message Processor blieb die Möglichkeit

verwehrt, als Software zu existieren – und das ist einfach nicht richtig!

Über die Notwendigkeit einer dauerhaften Separierung von Hardware und Software war ich mir schon seit Langem im Klaren gewesen und es leuchtete mir auch ein, warum das sinnvoll ist. Trotzdem machten mir Doug Schmidts Worte erst wirklich bewusst, wie man die Begriffe »Software« und »Firmware« in Relation zueinander setzen muss.

Für Softwareentwickler ist das eine klare Botschaft: Hört auf, so viel Firmware zu schreiben, und gebt eurem Code die Chance auf eine lange, nutzbringende Lebensdauer. Diese Forderung allein wird sicher noch nicht dafür sorgen, dass es auch so kommen wird. Aber schauen wir uns doch einmal an, wie sich die Architektur der eingebetteten Software sauber halten lässt, damit die Software eine Chance auf eine möglichst lange Lebenserwartung erhält.

29.1 App-Eignungstest

Warum ist die Wahrscheinlichkeit so hoch, dass eingebettete Software zu Firmware »mutiert«? Es scheint, als würden sich die Softwareentwickler schwerpunktmäßig darauf konzentrieren, den eingebetteten Code zum Laufen zu bringen, und weniger darauf, sie für eine lange, nutzbringende Lebensdauer zu strukturieren. Kent Beck beschreibt drei Kerndirektiven für die Erzeugung von Software (der zitierte Text gibt Kent Becks Worte wieder, meine Interpretationen dazu sind kursiv gesetzt):

1. »First make it work.« *Wenn es nicht funktioniert, sind Sie Ihren Job los.*

2. »Then make it right.« *Überarbeiten Sie den Code so, dass Sie selbst und andere ihn verstehen und bedarfsweise weiterentwickeln können oder er zumindest verständlicher wird.*

3. »Then make it fast.« *Gestalten Sie den Code zielgerichtet für die »nötige« Performance.*

Ein Großteil der eingebetteten Systeme, die ich auf dem freien Markt zu Gesicht bekomme, scheint sich hauptsächlich an der ersten Direktive »Bring sie zum Laufen!« zu orientieren – und darüber hinaus mitunter auch in geradezu obsessiver Manier an der Zielsetzung der dritten Direktive, »Krieg es schnell hin!«, die durch die Ergänzung von Mikrooptimierungen bei jeder sich bietenden Gelegenheit erreicht wird. In seinem Buch *»Vom Mythos des Mann-Monats«* stellt der Informatiker Fred Brooks die These auf, dass wir Systeme als »Wegwerfprodukte« planen. Sowohl Kent Beck als auch Fred Brooks geben im Prinzip die gleiche Empfehlung: Finden Sie heraus, was funktioniert, und schaffen Sie dann eine bessere Lösung.

Eingebettete Software ist im Hinblick auf diese Problemstellungen nicht unbedingt ein Sonderfall. Die meisten nicht eingebetteten Apps sind ebenfalls in erster Linie darauf ausgelegt, einfach nur zu funktionieren, ohne dass allzu viel Aufhebens um die Langlebigkeit des Codes gemacht wird.

Eine App zum Funktionieren zu bringen, ist ein Vorgang, den ich als *App-Eignungstest* für Programmierer bezeichne. Ob sie nun an eingebetteten oder nicht eingebetteten Systemen arbeiten: Softwareentwickler, die sich nur darauf konzentrieren, ihre App zum Laufen zu bringen, tun ihren Produkten und Auftraggebern damit keinen Gefallen. Zur Kunst des Programmierens gehört weitaus mehr, als eine Anwendung bloß lauffähig zu machen.

Betrachten Sie als Beispiel für einen Code, der im Rahmen eines solchen App-Eignungstests erzeugt wurde, die folgenden Funktionen, die in diesem Fall in einer einzigen Datei eines kleinen eingebetteten Systems untergebracht waren:

```
ISR(TIMER1_vect) { ... }
ISR(INT2_vect) { ... }
void btn_Handler(void) { ... }
float calc_RPM(void) { ... }
static char Read_RawData(void) { ... }
void Do_Average(void) { ... }
void Get_Next_Measurement(void) { ... }
void Zero_Sensor_1(void) { ... }
void Zero_Sensor_2(void) { ... }
void Dev_Control(char Activation) { ... }
char Load_FLASH_Setup(void) { ... }
void Save_FLASH_Setup(void) { ... }
void Store_DataSet(void) { ... }
float bytes2float(char bytes[4]) { ... }
void Recall_DataSet(void) { ... }
void Sensor_init(void) { ... }
void uC_Sleep(void) { ... }
```

Die Liste dieser Funktionen ist hier in der Reihenfolge wiedergegeben, in der ich sie in der Quelldatei vorgefunden habe. Wenn man diese Funktionen separiert und nach bestimmten Bereichen gruppiert, sieht das folgendermaßen aus:

- Funktionen, die die Domänenlogik enthalten:

```
float calc_RPM(void) { ... }

void Do_Average(void) { ... }

void Get_Next_Measurement(void) { ... }

void Zero_Sensor_1(void) { ... }
```

```
void Zero_Sensor_2(void) { ... }
```

- Funktionen, die die Hardwareplattform bereiten:

```
ISR(TIMER1_vect) { ... }*

ISR(INT2_vect) { ... }

void uC_Sleep(void) { ... }
Funktionen, die auf die Betätigung der On/Off-Taste reagieren

void btn_Handler(void) { ... }

void Dev_Control(char Activation) { ... }
Eine Funktion, die A/D(Analog/Digital)-Eingabewerte von der
Hardware abfragen kann

static char Read_RawData(void) { ... }
```

- Funktionen, die Werte im permanenten Speicher hinterlegen:

```
char Load_FLASH_Setup(void) { ... }

void Save_FLASH_Setup(void) { ... }

void Store_DataSet(void) { ... }

float bytes2float(char bytes[4]) { ... }

void Recall_DataSet(void) { ... }
```

- Funktionen, die nicht das tun, was ihre Bezeichnungen vermuten lassen:

```
void Sensor_init(void) { ... }
```

Bei näherer Betrachtung einiger anderer Dateien derselben Anwendung stieß ich auf zahlreiche Hindernisse, die das Verständnis des Codes erschwerten. Außerdem fiel mir eine Dateistruktur auf, die ausschließlich eine Möglichkeit zuließ, den Code zu testen: im Zusammenspiel mit der eingebetteten Zielhardware. Buchstäblich jedes Bit dieses Codes »wusste«, dass es sich in einer speziellen Mikroprozessorarchitektur befand, die »erweiterte« C-Konstrukte[3] nutzte, um den Code an eine bestimmte Toolkette und einen Mikroprozessor zu binden. Somit bestand für diese Software keine Aussicht auf eine lange Lebensdauer, es sei denn, das Produkt müsste zu keinem Zeitpunkt in eine abweichende Hardwareumgebung portiert werden.

Trotz allem hat der Softwareentwickler den App-Eignungstest bestanden, denn die Anwendung funktionierte – aber man kann nicht behaupten, dass sie über eine saubere eingebettete Architektur verfügen würde.

29.2 Der Flaschenhals der Zielhardware

Es gibt eine Menge spezieller Aspekte, mit denen sich die Entwickler eingebetteter Systeme im Gegensatz zu den Entwicklern nicht eingebetteter Systeme auseinandersetzen müssen – so zum Beispiel begrenzte Speicherkapazitäten, Echtzeitbeschränkungen und Deadlines, eingeschränkte I/O, unkonventionelle Benutzerschnittstellen sowie Sensoren und Anbindungen an die reale Welt. Meist wird die Hardware zeitgleich mit der Software und der Firmware entwickelt. Als Softwareentwickler eines derartigen Systems haben Sie zunächst möglicherweise keine Gelegenheit, den Code auch tatsächlich auszuführen. Und wenn das noch nicht schlimm genug ist: Sobald Sie dann die Hardware bekommen, wird sich höchstwahrscheinlich herausstellen, dass sie ihre ganz eigenen Schwachstellen hat, was den Entwicklungsfortschritt Ihrer Software zusätzlich noch über das normale Maß hinaus verzögert.

Ja, eingebetteter Code ist speziell. Und die Entwickler eingebetteter Software sind auch speziell. Andererseits ist die eingebettete Entwicklung aber wiederum auch nicht *so* speziell, dass sich die in diesem Buch vorgestellten Prinzipien nicht ebenso gut auf eingebettete Systeme anwenden ließen.

Einer dieser besonderen Problemfälle bei der Entwicklung eingebetteter Software ist der *Flaschenhals der Zielhardware*. Wenn eingebetteter Code ohne Berücksichtigung der Praktiken und Prinzipien der sauberen Architektur strukturiert wird, werden Sie oftmals in die Situation geraten, dass sich der Code ausschließlich unter Zuhilfenahme der Zielhardware testen lässt. Und wenn diese Hardware den einzigen gangbaren Weg zum Testen darstellt, wird dieser Flaschenhals Ihre Fortschritte verlangsamen.

29.2.1 Eine saubere eingebettete Architektur ist eine testfähige eingebettete Architektur

Deshalb befassen wir uns an dieser Stelle einmal mit der Anwendung der architektonischen Prinzipien auf eingebettete Software und Firmware, um dem Flaschenhals der Zielhardware zu entgehen.

Layer

Die Errichtung von Layer-Systemen hat viele Facetten. Hier sollen zunächst drei Layer zum Einsatz kommen, wie in Abbildung 29.1 dargestellt. Die Hardware befindet sich auf dem untersten Layer. Wie Doug Schmidt eindringlich zu bedenken gibt, wird sich die Hardware aufgrund technologischer Fortschritte und des mooreschen Gesetzes im Laufe der Zeit unvermeidlich ändern. Zum Teil wird sie überflüssig werden oder veralten, während neue Hardware weniger Strom verbraucht, eine bessere Performance bietet oder preiswerter ist. Und als Entwickler eingebetteter Software will man sich die Arbeit für den Fall, dass der Hardwarewechsel schließlich, aus welchem Grund auch immer, stattfindet, nicht unnötig schwerer machen als nötig.

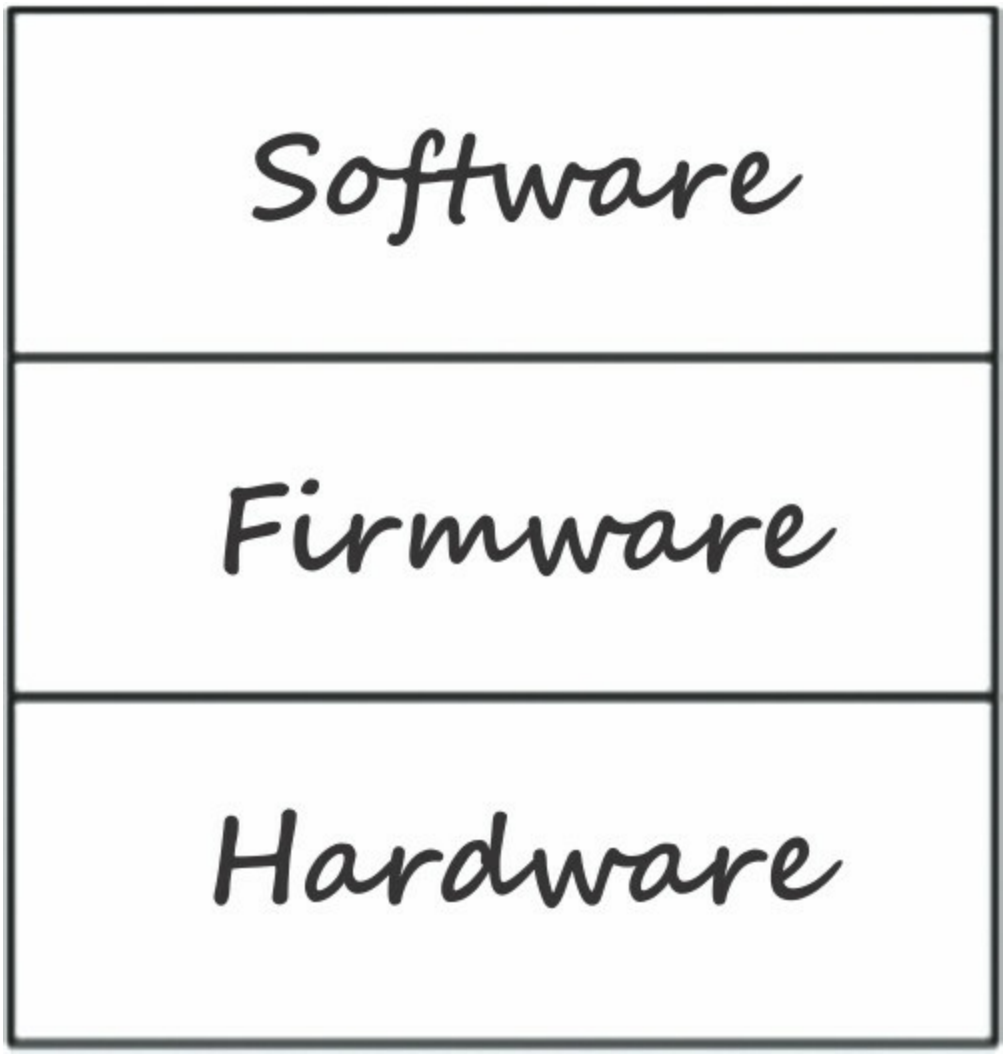


Abb. 29.1: Drei Layer

Die Separierung der Hardware vom restlichen System ist eine Grundvoraussetzung – zumindest sobald sie definiert ist (siehe Abbildung 29.2). Das ist oftmals der Punkt, an dem die Probleme beim Meistern des App-Eignungstests anfangen. Aber auch das Wissen um die zu verwendende Hardware garantiert noch nicht, dass der gesamte Code nicht dennoch verunreinigt wird. Wenn Sie nicht darauf achten, wo Sie die einzelnen Systemelemente platzieren und inwieweit ein Modul Kenntnis von einem anderen Modul haben darf, wird sich der Code später nur sehr schwer anpassen lassen – und damit meine ich nicht nur Anpassungen an neue Hardware, sondern auch die

Modifikationen, die vonseiten des Users angefragt oder zur Behebung eines Bugs erforderlich werden.

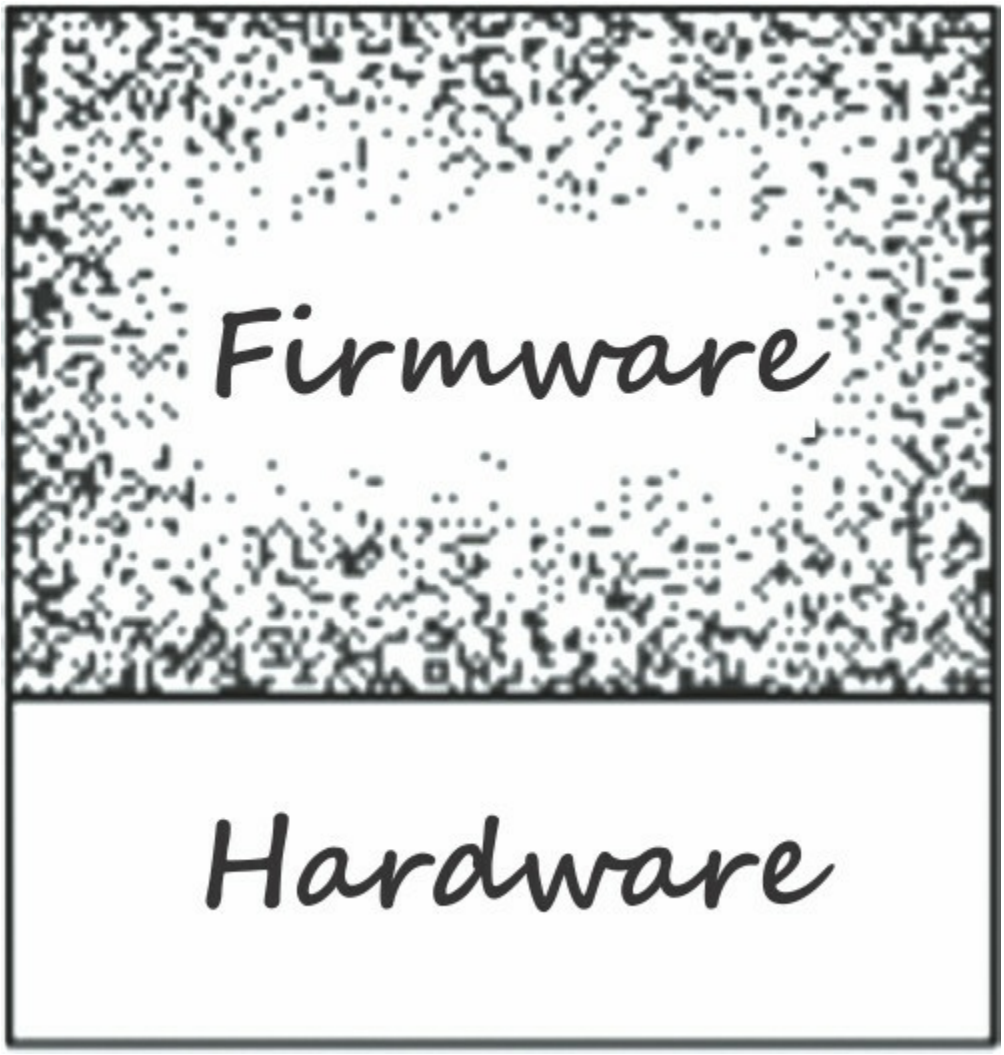


Abb. 29.2: Die Hardware muss vom übrigen System separiert werden.

Die Vermischung von Software und Firmware kommt einem Anti-Pattern gleich. Code, der dieses Anti-Pattern aufweist, wird Modifikationen gegenüber standhaft bleiben. Darüber hinaus werden Änderungen in dieser Konstellation riskant und haben häufig unbeabsichtigte Folgen – selbst für geringfügige Anpassungen werden somit vollständige Regressionstests am gesamten System notwendig. Sofern Sie keine extern instrumentierten Tests erstellt haben, sollten Sie davon ausgehen, dass Sie sich mit manuellen Tests herumschlagen müssen – und dann können Sie auch gleich neue Fehlerberichte erwarten.

Die Hardware ist ein Detail

Die Grenzlinie zwischen Software und Firmware ist typischerweise nicht so klar definiert wie die Grenze zwischen Code und Hardware (siehe Abbildung 29.3).

Abb. 29.3: Die Grenzlinie zwischen Software und Firmware verschwimmt ein wenig mehr als die Grenze zwischen Code und Hardware.

Eine der Aufgaben eines Entwicklers eingebetteter Software besteht darin, diese Grenzlinie zu festigen. Die Grenze zwischen Software und Firmware wird als *Hardware Abstraction Layer* (*HAL*, Hardware-Abstraktionsschicht) bezeichnet (siehe Abbildung 29.4). Dieses Konzept ist allerdings keineswegs neu: Im Bereich der PCs existierte es schon lange vor Windows.

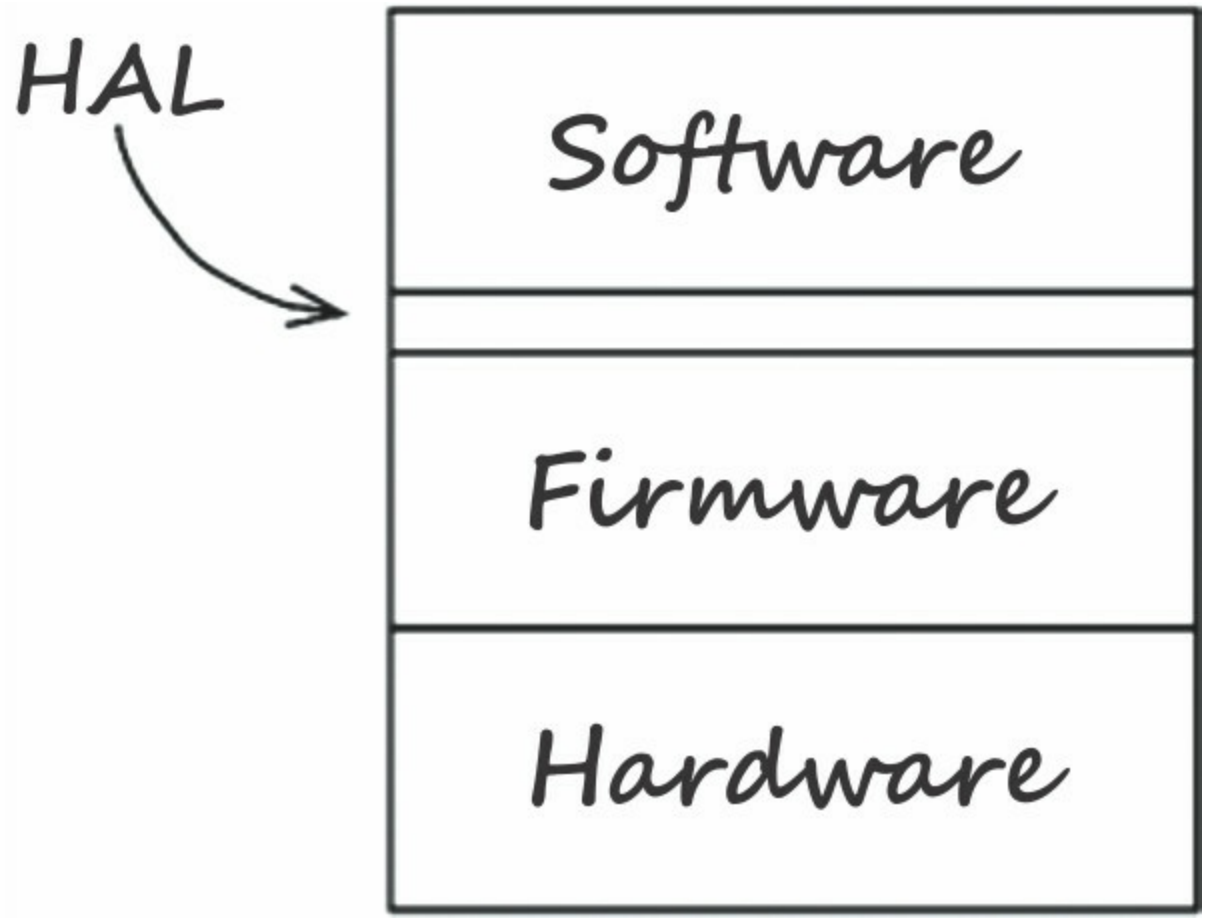


Abb. 29.4: Der *Hardware Abstraction Layer (HAL)*

Der HAL ist für die Software zuständig, die sich unmittelbar über dieser Schicht befindet, und dementsprechend sollte seine API auch auf die Bedürfnisse dieser Software zugeschnitten sein. Zum Beispiel kann die Firmware Bytes und Bytearrays im Flash-Speicher ablegen. Im Gegensatz dazu muss die Anwendung Name-Wert-Paare in einem permanenten Speicher unterbringen und auch von dort auslesen. Die Software sollte es jedoch nicht interessieren, ob die Name-Wert-Paare in einem Flash-Speicher, auf einer Disk, in der Cloud oder im Kernspeicher hinterlegt sind. Der HAL stellt daher einen entsprechenden Service bereit, teilt der Software allerdings nicht mit, wie der Vorgang vonstattengeht. Die Flash-Implementierung ist ein Detail, das vor der Software verborgen bleiben sollte.

Im nächsten Beispiel wird eine LED an ein *GPIO-Bit* (**G**eneral **P**urpose **I**nput/**O**utput, Allzweckeingabe/-ausgabe) angebunden. Die Firmware könnte den Zugriff auf die GPIO-Bits ermöglichen, wo wiederum ein HAL die Komponente `Led_TurnOn(5)` bereitstellen könnte. Das ist ein relativ tiefschichtiger Hardware Abstraction Layer. Angenommen, wir würden die Abstraktionsebene ausgehend von der Hardware in Richtung Softwareware/Produkt erhöhen. Was zeigt die LED an? In diesem Fall soll sie einen schwachen Batteriezustand angeben. Bei einem bestimmten Grenzwert könnte die Firmware (oder ein *Board Support Package (BSP)*) die Komponente `LED_TurnOn(5)` und der HAL die Komponente `Indicate_LowBattery()` bereitstellen. Wie hier zu erkennen ist, drückt der HAL Services aus, die von der Anwendung

benötigt werden. Außerdem zeigt dieses Beispiel, dass Layer ihrerseits wiederum Layer enthalten können. Es handelt sich eher um ein sich wiederholendes Fraktalmuster als um einen eingeschränkten Satz vordefinierter Layer. Die GPIO-Zuweisungen sind Details, die vor der Software verborgen bleiben sollten.

29.2.2 Offenbaren Sie dem HAL-User keine Hardwaredetails

Die Software einer sauberen eingebetteten Architektur ist auch *ohne* die Zielhardware testfähig. Ein guter HAL stellt die Nahtstelle oder einen Satz von Substitutionspunkten bereit, die ein von der Zielhardware unabhängiges Testen ermöglichen.

Der Prozessor ist ein Detail

Wenn Ihre eingebettete Anwendung eine spezialisierte Toolkette verwendet, wird sie Ihnen häufig Header-Dateien als Hilfestellung[4] zur Seite stellen. Diese Compiler machen oftmals von den Freiheiten der Programmiersprache C Gebrauch und ergänzen neue Schlüsselwörter, um auf ihre Prozessorfunktionen zuzugreifen. Der Code wird wie C aussehen, ist aber tatsächlich kein C mehr.

Manchmal sind in den herstellerseitig gelieferten C-Compilern so etwas Ähnliches wie globale Variablen für den direkten Zugriff auf Prozessorregister, I/O-Ports, Taktgeber, I/O-Bits, Interrupt-Controller und andere Prozessorfunktionen enthalten. Sicherlich ist ein einfacher Zugang zu diesen Elementen recht hilfreich, Sie sollten dabei jedoch beachten, dass sämtlicher Code, der diese Funktionen nutzt, kein C-Code mehr ist – und sich somit nicht für einen anderen Prozessor und ggf. auch nicht mit einem anderen Compiler für denselben Prozessor kompilieren lässt.

Ich gehe nicht davon aus, dass Halbleiterhersteller und Tool-Provider gezielt darauf aus sind, Ihr Produkt an den Compiler zu binden, sondern im Zweifelsfall tatsächlich nur helfen wollen. Letztlich ist es jedoch an Ihnen selbst, diese Hilfestellung auch wirklich in einer Form zu nutzen, dass sie Ihre zukünftigen Pläne nicht durchkreuzt. Die Beschränkung der Dateien, die die C-Erweiterungen kennen dürfen, liegt also ganz bei Ihnen.

Das folgende Beispiel zeigt eine Header-Datei, die für die fiktive ACME-Familie von *DSPs* (**D**igital **S**ignal **P**rocessor, digitaler Signalprozessor) entworfen wurde – Sie wissen schon, DSPs der Art, wie sie von Wile E. Coyote genutzt werden:

```
#ifndef _ACME_STD_TYPES
#define _ACME_STD_TYPES
#if defined(_ACME_X42)
      typedef unsigned int       Uint_32;
      typedef unsigned short     Uint_16;
```

```
      typedef unsigned char      Uint_8;

      typedef int                Int_32;
      typedef short              Int_16;
      typedef char               Int_8;

#elif defined(_ACME_A42)
      typedef unsigned long      Uint_32;
      typedef unsigned int       Uint_16;
      typedef unsigned char      Uint_8;

      typedef long               Int_32;
      typedef int                Int_16;
      typedef char               Int_8;
#else
      #error <acmetypes.h> wird für diese Umgebung nicht unterstützt
#endif

#endif
```

Die Header-Datei `acmetypes.h` sollte nicht direkt verwendet werden – andernfalls würde Ihr Code an einen der ACME-DSPs gebunden. Nun mögen Sie entgegnen: »Aber ich nutze doch einen ACME-DSP, was sollte denn da schon passieren?« Nun, Sie können Ihren Code nicht kompilieren, es sei denn, Sie schließen diesen Header mit ein. Wenn Sie den Header verwenden und `_ACME_X42` oder `_ACME_A42` definieren, werden Ihre Integer nicht die richtige Größe haben, um Ihren Code ohne die Zielhardware testen zu können. Schlimmer noch: Eines Tages werden Sie Ihre Anwendung auf einen anderen Prozessor portieren wollen und dann feststellen, dass Sie sich diese Aufgabe selbst schwer gemacht haben, indem Sie sich gegen die Portabilität entschieden und keine Einschränkungen hinsichtlich der Dateien, die von ACME Kenntnis haben dürfen, vorgenommen haben.

Statt die `acmetypes.h`-Datei einzusetzen, sollten Sie daher lieber versuchen, einen standardisierteren Weg zu gehen und die Header-Datei `stdint.h` zu nutzen. Und wenn der Ziel-Compiler keine `stdint.h` bereitstellt? In diesem Fall können Sie sie auch selbst schreiben. Eine solche individuell erstellte Header-Datei `stdint.h` für Ziel-Builds verwendet die `acmetypes.h`-Datei für Zielkompilierungen wie folgt:

```
#ifndef _STDINT_H_
#define _STDINT_H_

#include <acmetypes.h>

typedef Uint_32 uint32_t;
typedef Uint_16 uint16_t;
typedef Uint_8  uint8_t;

typedef Int_32  int32_t;
typedef Int_16  int16_t;
typedef Int_8   int8_t;
```

```
#endif
```

Der Einsatz der `stdint.h`-Header-Datei in Ihrer eingebetteten Software und Firmware unterstützt Sie dabei, Ihren Code sauber und portabel zu halten. Dabei steht außer Frage, dass jegliche *Software* prozessorunabhängig sein sollte – im Fall von *Firmware* ist das allerdings nicht immer möglich. Der folgende Codeabschnitt bedient sich spezieller C-Erweiterungen, die Ihrem Code Zugriff auf die Peripheriefunktionen des Mikrocontrollers gewähren. Es ist wahrscheinlich, dass Ihr Produkt diesen Mikrocontroller verwendet, um dessen integrierte Peripheriefunktionen nutzen zu können. Die im nachstehenden Listing dargestellte Funktion erzeugt eine Zeile, die ein »`hi`« an den seriellen Ausgabeport ausgibt. (Dieses Beispiel basiert auf einem realen Code.)

```
void say_hi()
{
   IE = 0b11000000;
   SBUF0 = (0x68);
   while(TI_0 == 0);
   TI_0 = 0;
   SBUF0 = (0x69);
   while(TI_0 == 0);
   TI_0 = 0;
   SBUF0 = (0x0a);
   while(TI_0 == 0);
   TI_0 = 0;
   SBUF0 = (0x0d);
   while(TI_0 == 0);
   TI_0 = 0;
   IE = 0b11010000;
}
```

Diese kleine Funktion bringt zahlreiche Probleme mit sich, von denen eins sofort ins Auge springt: `0b11010000`. Grundsätzlich ist diese Binärnotation zwar in Ordnung – aber kann C auch etwas damit anfangen? Leider nicht. Zudem finden sich in dem hier angeführten Codebeispiel noch ein paar andere problematische Stellen, die in direktem Zusammenhang mit der Verwendung der standardmäßigen C-Erweiterungen stehen:

- `IE`: *Interrupt Enable Bits* (Interrupt-Freigaberegister-Bits).
- `SBUF0`: *Serial Output Buffer* (serieller Ausgabepuffer).
- `TI_0`: *Serial Transmit Buffer Empty Interrupt* (serieller Sendepuffer). Das Einlesen einer 1 bedeutet, dass der Pufferspeicher leer ist.

Die in Großbuchstaben geschriebenen Variablen greifen auf integrierte Peripheriefunktionen des Mikrocontrollers zu. Wenn Sie die Interrupts und

Ausgabezeichen steuern wollen, müssen Sie diese Peripheriefunktionen nutzen. Sicher, das ist komfortabel – aber es ist nicht C.

Eine saubere eingebettete Architektur würde diese Zugriffsregister nur an sehr wenigen Stellen direkt einsetzen und sie zudem ausschließlich auf die *Firmware* beschränken. Alle Systemelemente, die Kenntnis von diesen Registern haben, werden zu *Firmware* und sind dementsprechend an den Halbleiter gebunden – das wird Ihnen allerdings dann Probleme bereiten, wenn Sie den Code zum Laufen bringen wollen, bevor Sie über stabile Hardware verfügen. Abgesehen davon wird es Ihnen eine solche Anbindung auch erschweren, Ihre eingebettete Anwendung auf einem anderen Prozessor zu betreiben.

Wenn Sie den Mikrocontroller auf die beschriebene Weise nutzen, könnte Ihre Firmware solche tiefschichtigen Funktionen mithilfe eines *Processor Abstraction Layers* (*PAL*, Prozessor-Abstraktionsschicht) isolieren. Dadurch wird die Firmware oberhalb des PALs ein bisschen weniger »firm« und kann dann auch ohne die Zielhardware getestet werden.

Das Betriebssystem ist ein Detail

Ein HAL ist also unerlässlich – aber reicht das auch aus? In eingebetteten Bare-Metal-Systemen kann ein HAL tatsächlich schon alles sein, was Sie brauchen, damit Ihr Code nicht zu sehr von der Betriebsumgebung abhängig ist. Doch wie steht es mit eingebetteten Systemen, die ein Echtzeitbetriebssystem (*RTOS*, **R**eal-**T**ime **O**perating **S**ystem) oder eine eingebettete Linux- oder Windows-Version nutzen?

Um Ihrem eingebetteten Code eine gute Aussicht auf eine lange Lebensdauer zu verschaffen, müssen Sie das Betriebssystem als Detail behandeln und entsprechende Abhängigkeiten ausschließen.

Der softwareseitige Zugriff auf die Services der Betriebsumgebung erfolgt über das Betriebssystem. Und das Betriebssystem bildet wiederum einen eigenen Layer, der die Software von der Firmware separiert (siehe Abbildung 29.5). Die direkte Verwendung eines Betriebssystems kann somit Probleme bereiten. Was würde zum Beispiel passieren, wenn der Hersteller Ihres Echtzeitbetriebssystems von einem anderen Unternehmen aufgekauft wird und in der Folge die Lizenzgebühren erhöht werden oder die Qualität des Systems nachlässt? Was, wenn sich Ihre Anforderungen ändern und Ihr Echtzeitbetriebssystem die von Ihnen benötigten Fähigkeiten nicht mehr aufbringt? Solche Fälle machen weitreichende Anpassungen Ihres Codes erforderlich – und dabei wird es sich nicht nur um einfache syntaktische Modifikationen aufgrund der neuen Betriebssystem-API handeln. Wahrscheinlicher ist vielmehr, dass Sie den Code semantisch an die diversen Fähigkeiten und Grundfunktionen des neuen Betriebssystems anpassen müssen.

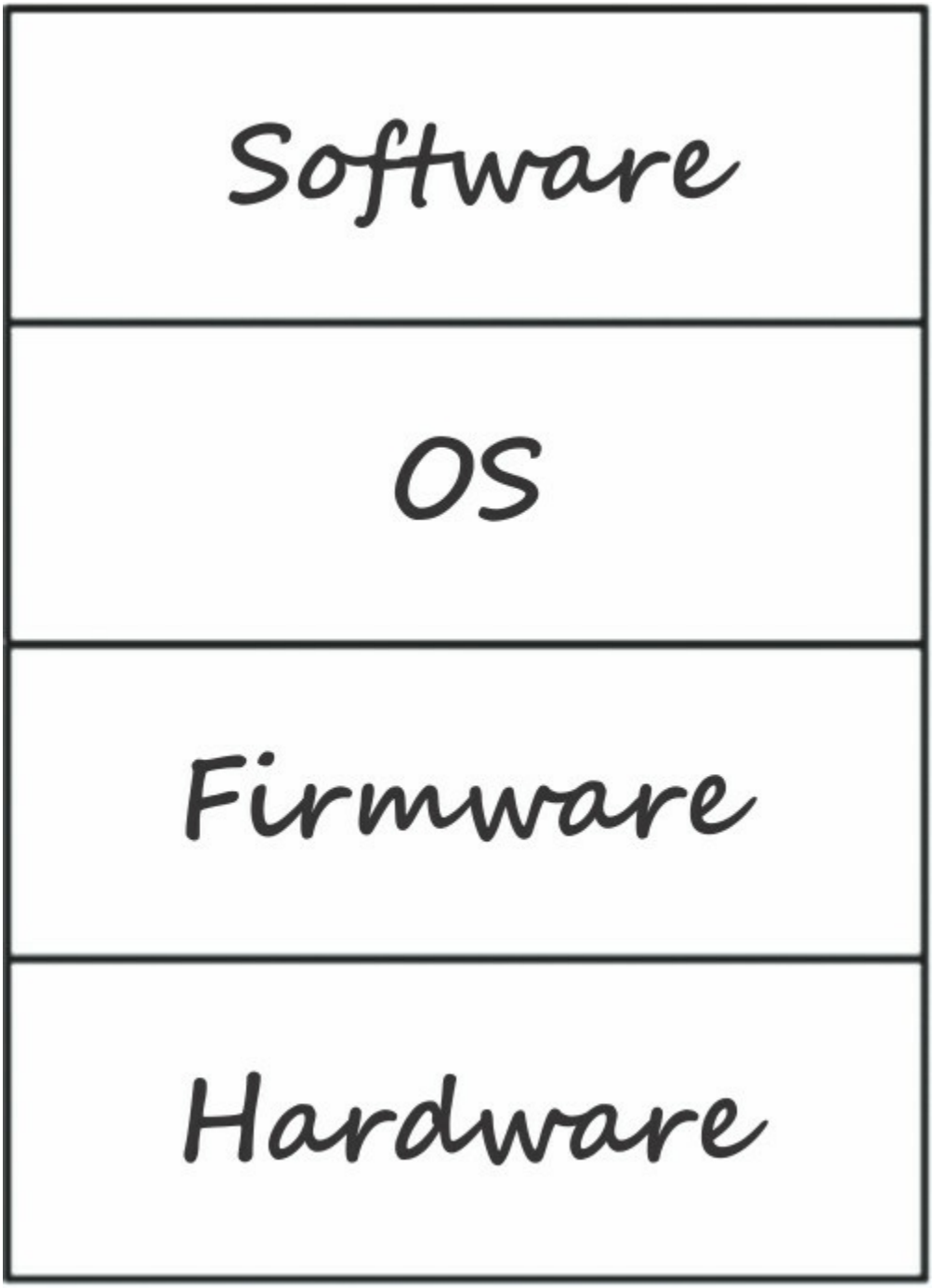


Abb. 29.5: Ergänzung eines Betriebssystems (OS)

Eine saubere eingebettete Architektur isoliert die Software mithilfe eines *Operating System Abstraction Layers* (*OSAL*, Betriebssystem-Abstraktionsschicht) von dem Betriebssystem (siehe Abbildung 29.6). In einigen Fällen mag die Implementierung dieses Layers einfach sein und lediglich die Namensänderung einer Funktion bedeuten – in anderen Fällen könnten jedoch auch gleich mehrere Funktionen zusammengefasst werden müssen.

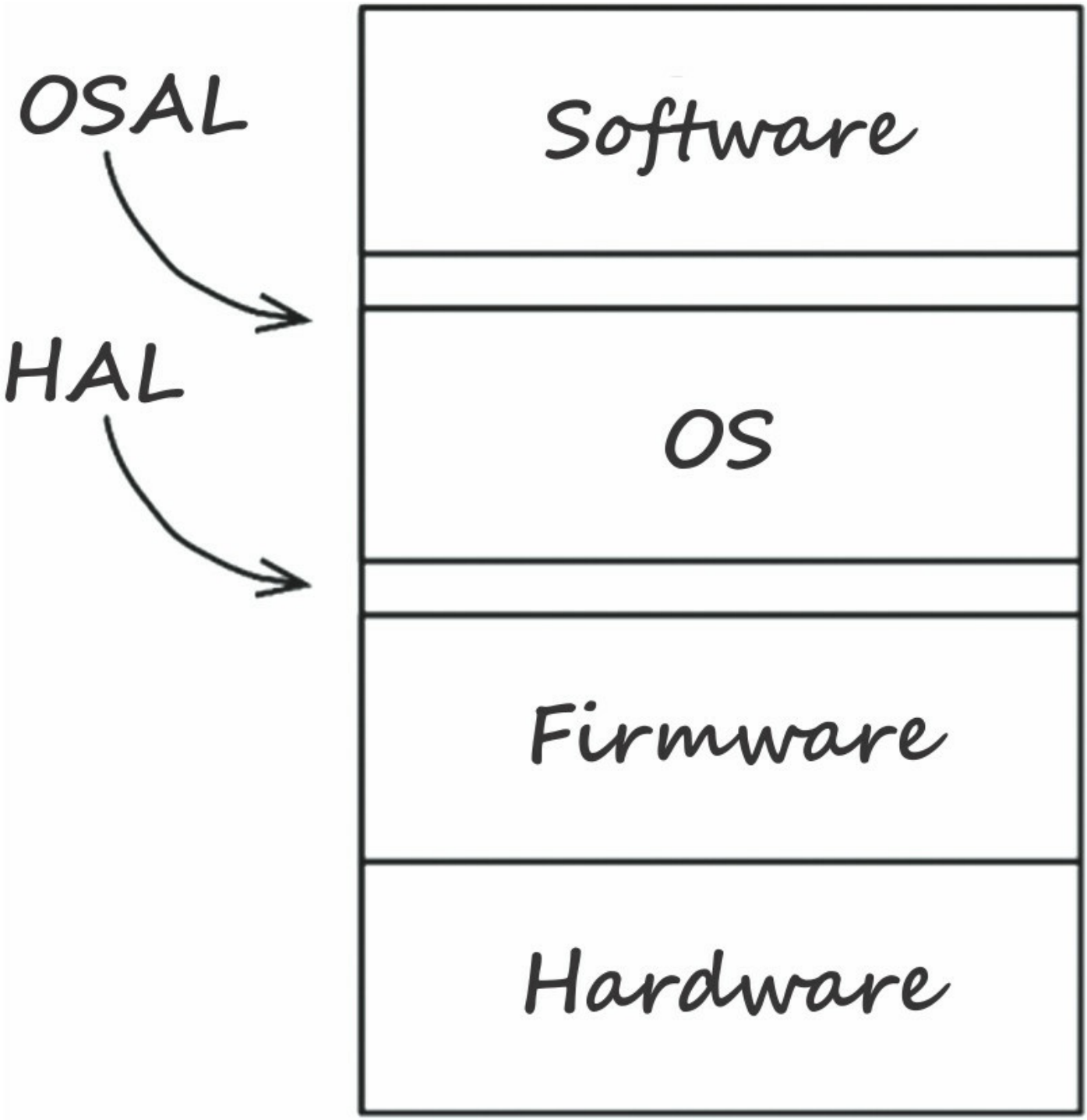


Abb. 29.6: Der *Operating System Abstraction Layer (OSAL)*

Sollten Sie Ihre Software schon jemals von einem Echtzeitbetriebssystem auf ein anderes portiert haben, werden Sie wissen, wie schwierig das ist. Wäre sie allerdings nicht direkt vom Betriebssystem abhängig, sondern von einem OSAL, dann würden Sie im Wesentlichen nur ein neues OSAL schreiben müssen, das mit dem vorherigen kompatibel ist. Was wäre also Ihnen lieber: einen umfangreichen Teil Ihres komplexen, bereits vorhandenen Codes anzupassen oder neuen Code für eine definierte Schnittstelle und ein definiertes Verhalten zu erzeugen? Das ist keineswegs eine Fangfrage. Ich persönlich würde mich jedenfalls jederzeit für Letzteres entscheiden.

Nun fragen Sie sich womöglich, ob Ihr Code nicht unnötigerweise zu sehr aufblähen könnte. Faktisch wird der Layer jedoch zu eben dem Standort, an dem der überwiegende Teil der auf das Betriebssystem zurückzuführenden Duplizierungen isoliert wird. Diese Duplizierungen müssen nicht unbedingt mit einem großen Aufwand einhergehen. Wenn Sie einen OSAL definieren, können Sie Ihre Anwendungen auch dazu veranlassen, einer gemeinsamen Struktur zu folgen. So könnten Sie beispielsweise Mechanismen für den Nachrichtenaustausch einrichten, statt dass jeder Thread ein eigenes Nebenläufigkeitsmodell verfolgt.

Der OSAL kann Sie dabei unterstützen, Testpunkte zu setzen, damit der wertvolle Anwendungscode im Software-Layer auch ohne die Zielhardware und das Betriebssystem getestet werden kann. In einer sauberen eingebetteten Architektur ist die Software auch ohne das Zielbetriebssystem testfähig. Und ein vernünftiger OSAL stellt diese Nahtstelle oder einen Satz von Substitutionspunkten zur Verfügung, die das Testen ohne die Zielhardware ermöglichen.

Programmierung für Schnittstellen und Substituierbarkeit

Zusätzlich zur Ergänzung eines HALs und ggf. eines OSALs in jedem der Hauptlayer (*Software, OS (Betriebssystem), Firmware* und *Hardware*) können und sollten Sie natürlich auch die in diesem Buch beschriebenen Prinzipien anwenden. Sie fördern die Trennung der einzelnen Systemfaktoren sowie der Programmierung für Schnittstellen und Substituierbarkeit.

Das Konzept einer Schichtenarchitektur baut auf der Idee der Programmierung für Schnittstellen auf. Sobald ein Modul über eine Schnittstelle mit einem anderen Modul interagiert, können Sie problemlos einen Service-Provider durch einen anderen ersetzen. Möglicherweise haben Sie Ihre eigene kleine Version der Ausgabefunktion `printf` für das Deployment in der Zielhardware geschrieben. Solange die Schnittstelle zu Ihrer `printf`-Funktion der Standardversion von `printf` entspricht, können Sie jeden Service mit einem anderen überschreiben.

Eine Faustregel ist hierbei, Header-Dateien als Schnittstellendefinitionen zu verwenden. Wenn Sie sich daran halten, müssen Sie jedoch mit dem Inhalt dieser Header-Datei etwas vorsichtig sein. Beschränken Sie ihn auf Funktionsdeklarationen sowie die von der Funktion benötigten Konstanten und Strukturnamen.

Überfrachten Sie die Header-Dateien für die Schnittstellen nicht mit Datenstrukturen, Konstanten und Typdefinitionen, die nur von der Implementierung benötigt werden. Das ist nicht nur eine Frage der Ordnung: Eine solche Überfrachtung wird unweigerlich auch zu unerwünschten Abhängigkeiten führen. Beschränken Sie außerdem die Sichtbarkeit der Implementierungsdetails und richten Sie sich darauf ein, dass sich die Implementierungsdetails ändern werden. Je weniger Stellen es gibt, an denen der Code Kenntnis von den Details hat, desto weniger Stellen wird es auch

geben, an denen der Code entsprechend durchforstet und modifiziert werden muss.

Eine saubere eingebettete Architektur ist innerhalb der Layer testfähig, weil die Module über Schnittstellen interagieren. Jede Schnittstelle stellt eine solche Nahtstelle oder einen Satz von Substitutionspunkten zur Verfügung, die das Testen ohne die Zielhardware ermöglichen.

DRY-bedingte Kompilierungsrichtlinien

Ein oft verkanntes Einsatzgebiet der Substituierbarkeit bezieht sich darauf, wie eingebettete C- und C++-Programme mit unterschiedlicher Zielhardware oder Betriebssystemen umgehen. Es ist eine Tendenz zur Verwendung der bedingten Kompilierung zwecks Zu- und Abschaltung einzelner Codeabschnitte zu beobachten. Ich erinnere mich in diesem Zusammenhang an einen besonders problematischen Fall, bei dem die Anweisung `#ifdef BOARD_V2` in einer Telekommunikationsanwendung mehrere Tausend Mal erwähnt wurde.

Diese Form der Codewiederholung verstößt jedoch gegen das sogenannte *DRY-Prinzip*[5] (**D**on't **R**epeat **Y**ourself, zu Deutsch etwa »Wiederhole dich nicht«). Wenn ich eine Anweisung wie `#ifdef BOARD_V2` einmal sehe, ist das nicht wirklich ein Problem. Finde ich sie hingegen gleich *sechstausend Mal* vor, dann ist das allerdings ein extremes Problem! Die bedingte Kompilierung zur Identifizierung des Typs der Zielhardware wird in eingebetteten Systemen häufig wiederholt. Doch was gibt es für eine Alternative?

Wie wäre es denn mit einem Hardware Abstraction Layer? Der Hardwaretyp würde somit zu einem unter dem HAL verborgenen Detail. Und wenn der HAL nun einen Schnittstellensatz bereitstellen würde, statt die bedingte Kompilierung zu nutzen, könnten Sie den Linker oder eine Form der Laufzeitanbindung einsetzen, um die Software mit der Hardware zu koppeln.

29.3 Fazit

Entwickler eingebetteter Software können in hohem Maße von den auf dem Gebiet der nicht eingebetteten Software gemachten Erfahrungen profitieren. Sollten Sie also im Bereich der eingebetteten Softwareentwicklung tätig sein und dieses Buch lesen, dann werden Sie hier eine Fülle von nützlichem Wissen und hilfreichen Anregungen vorfinden.

Wenn Sie Ihren gesamten Code als Firmware gestalten, ist das im Hinblick auf eine langfristige Lebensdauer Ihres Produkts nicht sinnvoll. Auch Tests nur im

Zusammenspiel mit der Zielhardware ausführen zu können, zahlt sich diesbezüglich nicht aus. Eine saubere eingebettete Architektur ist der langfristigen Lebensdauer Ihres Produkts dagegen sehr zuträglich.

---

[1] https://insights.sei.cmu.edu/sei_blog/2011/08/the-growing-importance-of-sustaining-software-for-thedod.html

[2] Das Hardwaregerät, das den seriellen Port steuert.

[3] Einige Halbleiterhersteller ergänzen Schlüsselwörter zur C-Programmiersprache, um den Zugang zu Registern und I/O-Ports aus C heraus zu erleichtern. In solchen Fällen handelt es sich dann aber leider nicht mehr um C-Code.

[4] Die HTML-Tags wurden an dieser Stelle absichtlich eingefügt.

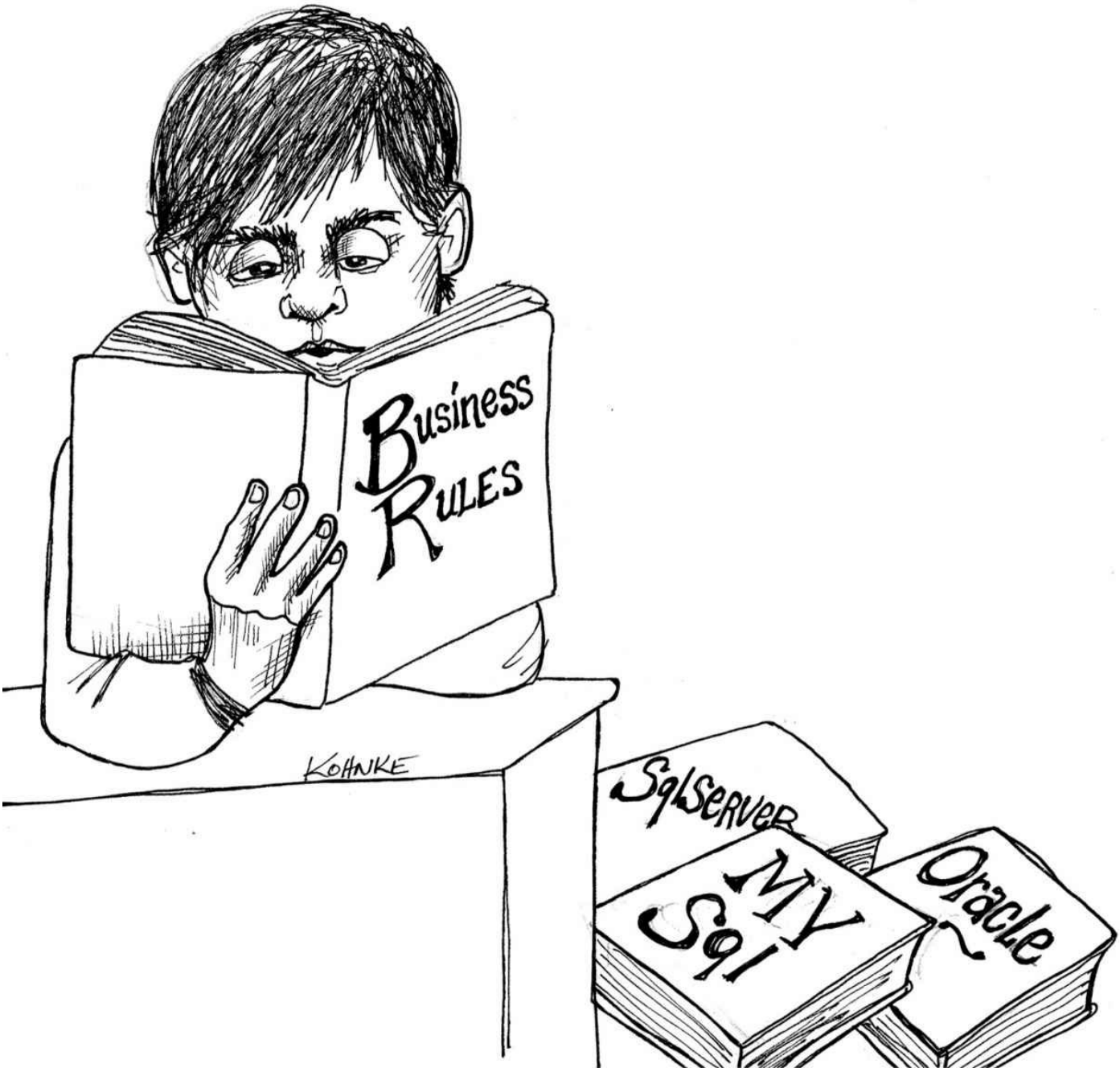
[5] David Thomas und Andrew Hunt, *Der Pragmatische Programmierer*, Hanser Fachbuch, 2003.

Teil VI
Details

In diesem Teil:

Kapitel 30

Die Datenbank ist ein Detail



Aus architektonischer Sicht ist die Datenbank eine Nicht-Entität – sie ist ein Detail, das die Ebene eines architektonischen Elements nicht erreicht. Man könnte auch sagen: Die Datenbank steht zu der Architektur eines Softwaresystems in einem ähnlichen Verhältnis wie ein Türknauf zu der Architektur Ihres Hauses.

Mir ist bewusst, dass es sich hierbei um eine provokante Aussage handelt – und glauben Sie mir, ich habe mich der daraus folgenden Debatte mehr als nur einmal gestellt. Deshalb lassen Sie mich Folgendes klarstellen: Ich schreibe hier nicht vom Datenmodell. Die Struktur, die Sie den Daten innerhalb Ihrer Anwendung geben, ist für die Architektur Ihres Systems selbstverständlich von entscheidender Bedeutung. Aber die Datenbank ist nicht das Datenmodell. Sie ist lediglich eine Software, ein Utility-Programm, das den Zugriff auf die Daten ermöglicht. Aus architektonischer Sicht ist dieses Utility-Programm insofern irrelevant, als es sich um ein tiefschichtiges

Detail handelt – einen Mechanismus. Und ein guter Softwarearchitekt lässt nicht zu, dass tiefschichtige Mechanismen seine Systemarchitektur verunreinigen.

30.1 Relationale Datenbanken

Nachdem Edgar Codd 1970 die Prinzipien der relationalen Datenbanken definiert hatte, entwickelte sich das relationale Modell bis Mitte der 1980er-Jahre zum führenden Verfahren für die Datenspeicherung. Und dafür gab es gute Gründe: Das relationale Datenbankmodell ist elegant, diszipliniert und robust – kurzum eine hervorragende Technologie für die Datenspeicherung und den Datenzugriff.

Doch unabhängig davon, wie brillant, nützlich und mathematisch fundiert eine Technologie auch sein mag: Sie ist und bleibt »nur« eine Technologie – und das bedeutet, sie ist ein Detail.

Während relationale Tabellen für bestimmte Arten des Datenzugriffs zweifellos komfortabel sind, hat das Anordnen von Daten in Zeilen innerhalb von Tabellen keinerlei architektonische Relevanz. Die Use Cases Ihrer Anwendung sollten weder Kenntnis von derartigen Verfahren haben, noch sollten sie für sie von Belang sein. Tatsächlich sollte die Kenntnis von der Tabellenstruktur der Daten sogar nur auf die tiefschichtigsten Ebenen der Utility-Funktionen in den äußeren Kreisen der Systemarchitektur beschränkt sein.

Viele Frameworks für den Datenzugriff ermöglichen die systemweite Weitergabe von Datenbankzeilen und -tabellen in Objektform. Dies zuzulassen, ist in Bezug auf die Systemarchitektur jedoch ein Fehler, denn: Eine solche Weitergabemöglichkeit koppelt die Use Cases, Geschäftsregeln und in einigen Fällen sogar die Benutzerschnittstelle an die relationale Struktur der Daten.

30.2 Warum sind Datenbanksysteme so weit verbreitet?

Es stellt sich die Frage, warum Softwaresysteme und Softwareunternehmen so sehr von Datenbanksystemen dominiert werden. Was verleiht Oracle, MySQL und SQL Server eine solche Vorrangstellung? Die kurze Antwort auf diese Frage lautet: Es sind die Festplatten.

Die rotierende magnetische Festplatte bildete über fünf Jahrzehnte hinweg das Fundament der Datenspeicherung. Gleich mehrere Generationen von

Softwareentwicklern kannten überhaupt keine andere Form der Datenspeicherung. Im Laufe der Jahre entwickelte sich die Festplattentechnologie von riesigen Stapeln massiver Platten mit einem Durchmesser von 48 Zoll, die Tausende Kilos wogen und 20 MByte Speicherkapazität boten, bis hin zu einzelnen dünnen Scheiben von lediglich 3 Zoll Durchmesser, die nur ein paar Gramm wiegen und ein Terabyte und mehr Speicherkapazität zu bieten haben. *Es war ein rasanter Entwicklungsprozess.* Und doch plagten sich die Programmierer die ganze Zeit über mit einer fatalen Eigenschaft der Festplattentechnologie: Diese Datenträger sind *langsam*.

Auf einer Festplatte werden die Daten in kreisförmigen Spuren gespeichert. Diese Spuren sind in Sektoren unterteilt, die eine zweckmäßige Anzahl von Bytes enthalten, meist 4 KB oder genauer 4.096 Bytes. Jede in der Festplatte verbaute Datenträgerscheibe (auch *Platter* genannt) kann Hunderte von Spuren beherbergen, wobei ein Dutzend oder mehr dieser Scheiben vorhanden sein können. Um ein bestimmtes Byte von der Festplatte auszulesen, muss der Schreib-/Lesekopf auf die richtige Spur verschoben werden, dann wird gewartet, bis die Festplatte in den richtigen Sektor rotiert ist, sämtliche 4 KB dieses Sektors in das RAM eingelesen wurden und der RAM-Speicher indiziert ist, bis schließlich das gewünschte Byte gefunden wird. Und das braucht Zeit – genauer ausgedrückt Millisekunden.

Nun scheinen Millisekunden nicht gerade ein hoher Zeitaufwand zu sein, dennoch ist eine Millisekunde immerhin eine Million Mal langsamer als die Zykluszeit der meisten Prozessoren. Wenn sich die Daten also nicht gerade auf einer Festplatte befinden würden, könnte der Zugriff darauf demnach eigentlich binnen Nanosekunden statt Millisekunden erfolgen.

Zur Verringerung der von den Festplatten verursachten zeitlichen Verzögerung bedarf es Indizes, Caches und optimierter Abfragemodelle – sowie eines gleichförmigen Verfahrens zur Darstellung der Daten, damit diese Indizes, Caches und Abfragemodelle wissen, womit sie arbeiten. Kurz: Sie brauchen ein Datenzugriffs- und -verwaltungssystem. Im Laufe der Jahre haben sich zwei Arten solcher Systeme etablieren können: Dateisysteme und relationale Datenbankverwaltungssysteme (*RDBMS*, **R**elational **D**atabase **M**anagement **S**ystems).

Dateisysteme sind dokumentenbasiert. Sie bieten eine intuitive und komfortable Möglichkeit, ganze Dokumente zu speichern. Wenn es darum geht, einen Satz Dokumente nach Namen zu speichern und abzurufen, funktionieren sie wirklich gut – sobald die Inhalte der Dokumente durchsucht werden müssen, sind sie allerdings keine allzu große Hilfe. Eine Datei namens LOGIN.C zu finden, ist in diesen Systemen ein Leichtes, deutlich schwieriger und langwieriger wird es dagegen, wenn alle .c-Dateien aufgespürt werden müssen, die eine Variable mit der Bezeichnung x enthalten.

Beide Systemvarianten organisieren die Daten auf der Festplatte so, dass sie entsprechend ihrer jeweiligen speziellen Zugriffsanforderungen so effizient wie möglich gespeichert und abgerufen werden können. Jedes dieser Systeme verfügt über

ein eigenes Schema zum Indizieren und Ordnen der Daten. Außerdem werden die relevanten Daten in beiden Fällen schließlich im RAM abgelegt, wo sie schnell manipuliert werden können.

30.3 Was wäre, wenn es keine Festplatten gäbe?

Trotz ihrer früheren Vormachtstellung sind Festplatten mittlerweile eine »aussterbende Art« und werden bald denselben Weg gehen, den auch schon Bandlaufwerke, Diskettenlaufwerke und CDs beschreiten mussten: Sie werden durch das RAM ersetzt.

Stellen Sie sich doch mal folgende Frage: Wenn es keine Festplatten mehr gäbe und all Ihre Daten im RAM abgelegt wären, wie würden Sie dann Ihre Datenbestände organisieren? Würden Sie sie in Tabellen anordnen und den Zugriff darauf mit SQL realisieren? Oder würden Sie sie in Dateien organisieren und über ein Verzeichnis darauf zugreifen?

Natürlich nicht. Sie würden sie in verzweigte Listen, Baumstrukturen, Hashtabellen, Stacks, Warteschlangen oder irgendeiner der Myriaden von möglichen Datenstrukturen organisieren und den Zugriff mithilfe von Pointern oder Referenzierungen ermöglichen – weil *es das ist, was Programmierer machen*.

Wenn Sie einmal darüber nachdenken, werden Sie feststellen, dass Sie im Grunde genommen schon jetzt genau so vorgehen. Obwohl die Daten in einer Datenbank oder einem Dateisystem gespeichert sind, lesen Sie sie in das RAM ein und organisieren sie dann nach Bedarf in Listen, Datensätze, Stacks, Warteschlangen, Baum- oder sonstige Datenstrukturen, die Ihnen gerade sinnvoll erscheinen. Dass Sie die Daten in Datei- oder Tabellenform belassen, ist wohl eher unwahrscheinlich.

30.4 Details

Eben diese realen Gegebenheiten sind auch der Grund dafür, dass ich eine Datenbank als ein Detail betrachte. Sie stellt lediglich einen Mechanismus dar, den wir uns zunutze machen, um die Daten zwischen der Oberfläche der Festplatte und dem Arbeitsspeicher hin- und herzubewegen. Tatsächlich ist eine Datenbank nichts anderes als ein großes, mit Bits gefülltes Behältnis, in dem wir unsere Daten langfristig speichern – dass wir die Daten auch in dieser Form verwenden, kommt jedoch nur selten vor.

Aus architektonischer Sicht sollten wir uns daher nicht um die Form kümmern, in der die Daten auf der Oberfläche einer rotierenden magnetischen Festplatte vorliegen. Stattdessen sollten wir die Existenz der Festplatte in diesem Kontext einfach außer Acht lassen.

30.5 Und was ist mit der Performance?

Ist die Performance denn kein architektonischer Aspekt? Natürlich ist sie das – aber wenn es um die Datenspeicherung geht, ist sie ein Aspekt, der vollständig gekapselt und von den Geschäftsregeln separiert werden kann. Sicher, wir müssen die Daten schnell in den Datenspeicher hinein- und wieder herausbekommen, trotzdem handelt es sich dabei um ein nachrangiges Problem, dem wir mit tiefschichtigen Datenzugriffsmechanismen begegnen können. Mit der Gesamtarchitektur unserer Systeme hat das jedoch überhaupt nichts zu tun.

30.6 Anekdote

In den späten 1980er-Jahren leitete ich ein Team von Softwareingenieuren in einem Start-up-Unternehmen, das im Begriff war, ein Netzwerkmanagementsystem für Messungen der Kommunikationsintegrität von T1-Telekommunikationsleitungen zu errichten und auf den Markt zu bringen. Dieses System rief Daten von den Geräten an den Endpunkten der Leitungen ab und führte dann eine Reihe von Vorhersagealgorithmen aus, um Probleme aufzuspüren und zu melden.

Wir nutzten Unix-Plattformen und speicherten unsere Daten in einfachen Random-Access-Dateien (Dateien mit wahlfreiem Zugriff). Eine relationale Datenbank brauchten wir nicht, weil die Daten nur wenige inhaltsbezogene Beziehungen aufwiesen – und deshalb war es sinnvoller, sie in Baumstrukturen und verknüpften Listen in den Random-Access-Dateien aufzubewahren. Kurz: Wir hielten die Daten in einer Form vor, in der sie sich am komfortabelsten in das RAM laden ließen, wo sie schließlich manipuliert werden konnten.

Das Start-up-Unternehmen stellte bald einen Marketingmanager ein – einen netten, sachkundigen Typen –, der mir gleich als Erstes zu verstehen gab, dass wir eine relationale Datenbank im System haben müssten. Das war keine Option und auch kein technisches Anliegen – es war eine Marketingfrage.

Für mich ergab das keinen Sinn. Warum in aller Welt sollte ich meine verknüpften Listen und Baumstrukturen in einen Haufen Zeilen und Tabellen umordnen, auf die über SQL zugegriffen würde? Warum sollte ich all den Aufwand und die Kosten für

ein riesiges RDBMS in Kauf nehmen, wenn doch ein einfaches Random-Access-Dateisystem mehr als ausreichend war? Also habe ich mich mit Händen und Füßen dagegen zur Wehr gesetzt.

Wir hatten in diesem Unternehmen allerdings auch einen Hardwareingenieur, der in das gleiche RDBMS-Horn stieß. Er war ebenfalls überzeugt, dass unser Softwaresystem aus technischen Gründen ein RDBMS brauchte. Also hielt er hinter meinem Rücken Meetings mit den Führungskräften der Firma ab, malte ein auf einem wackeligen Pfahl balancierendes Haus mit einem Strichmännchen darauf an das Whiteboard und fragte in die versammelte Runde: »Würden Sie ein Haus auf einer wackeligen Stange bauen?« Die Botschaft, die er damit vermitteln wollte, war, dass ein RDBMS, das seine Tabellen in Random-Access-Dateien aufbewahrt, irgendwie zuverlässiger wäre als die Random-Access-Dateien, die wir schon benutzten.

Also stellte ich mich auch ihm entgegen. Und dem Marketingtypen. Angesichts so viel unfassbarer Ignoranz blieb ich meinen technischen Prinzipien standhaft treu – und stemmte mich vehement dagegen.

Letzten Endes wurde der Hardwareentwickler über meinen Kopf hinweg zum Softwaremanager befördert. Und sie haben ein RDBMS in das arme System eingepflanzt. Und sie hatten absolut recht damit, während ich falsch gelegen hatte.

Aber wohlgemerkt nicht aus technischen Gründen: Damit hatte ich recht. Es war richtig, dass ich mich dagegen gewehrt hatte, ein RDBMS in den architektonischen Kern des Systems einzubauen. Womit ich falsch lag, war die Tatsache, dass unsere Kunden eine relationale Datenbank von uns erwarteten. Nicht, dass sie gewusst hätten, was sie damit anfangen sollten. Faktisch hatten sie überhaupt keine realistische Möglichkeit, sie in unserem System zu nutzen. Doch das spielte gar keine Rolle: Unsere Kunden erwarteten ein RDBMS. Es war einfach ein Punkt auf der Wunschliste all unserer Softwarekäufer geworden. Dahinter stand keinerlei technisches Verständnis – Rationalität hatte damit nichts zu tun. Es war schlicht und ergreifend ein irrationales, von außen kommendes und vollkommen unbegründetes Bedürfnis, das deswegen aber nicht minder real war.

Doch woher kam dieses Bedürfnis? Es war durch die damals sehr effektiven Marketingkampagnen der Datenbankanbieter entstanden. Sie hatten es geschafft, hochrangige Führungskräfte davon zu überzeugen, dass ihre unternehmenseigenen »Datenbestände« geschützt werden müssten und dass die von den entsprechenden Herstellern angebotenen Datenbanksysteme genau das richtige Mittel waren, um diesen Schutz zu gewährleisten.

Genau die gleichen Marketingstrategien sehen wir auch heute. Die Begriffe »Enterprise« oder »Unternehmen« und das Konzept der »serviceorientierten Architektur« haben viel mehr mit Marketing zu tun als mit den realen Gegebenheiten.

Was also hätte ich in dieser lange zurückliegenden Situation tun *sollen*? Ich hätte an der Peripherie des Systems ein RDBMS sowie einen schmalen und sicheren Datenzugriffskanal für das System bereitstellen und die Random-Access-Dateien im Systemkern beibehalten sollen. Doch was *habe* ich gemacht? Ich habe gekündigt und wurde Berater.

30.7 Fazit

Die Organisationsstruktur der Daten, das Datenmodell, ist aus architektonischer Sicht sehr bedeutsam – die Technologien und Systeme, die Daten auf und von einer rotierenden magnetischen Oberfläche bewegen, sind es dagegen nicht. Relationale Datenbanksysteme, die die Organisation der Daten in Tabellen erzwingen und auf die mit SQL zugegriffen wird, haben viel mehr mit Letzterem als mit Ersterem zu tun. Die Daten sind signifikant – die Datenbank ist jedoch ein Detail.

Kapitel 31
Das Web ist ein Detail

Waren Sie vielleicht schon in den 1990er-Jahren als Softwareentwickler tätig? Erinnern Sie sich noch, wie das Internet alles auf den Kopf stellte? Und wissen Sie auch noch, wie verächtlich wir damals angesichts der neuen, glanzvollen Technologie des World Wide Webs auf unsere alten Client-Server-Architekturen herabgeschaut haben?

Tatsächlich hat das Web gar nichts auf den Kopf gestellt – oder zumindest hätte es das nicht sollen. Das Internet ist lediglich die neueste Station in einer Reihe von

Pendelausschlägen, die unsere Industrie seit den 1960er-Jahren durchlebt hat. Dabei wechseln die Pendeltendenzen ständig zwischen der Bereitstellung der gesamten Computerleistung auf zentralen Servern und der Herausnahme der gesamten Computerleistung von den Terminals hin und her.

Einige Pendelstationen sind erst in den letzten etwa zehn Jahren hinzugekommen, seit das Web immer mehr an Bedeutung gewann. So hielt man es zunächst für sinnvoll, sämtliche Computerleistung in Serverfarmen zu stecken und die Browser »dumm« zu lassen. Als Nächstes wurden dann die Browser mit Applets versehen. Das gefiel den Softwareentwicklern dann aber doch nicht mehr so gut, also verlagerten sie die dynamischen Inhalte wieder auf die Server zurück. Nun stellte sich heraus, dass dies ebenfalls nicht das Richtige war, deshalb wurde das Web 2.0 erfunden und nun mit Ajax und JavaScript wiederum dem Browser eine Menge Verarbeitungslast zugeschustert. Das ging dann so weit, dass die Softwareentwickler riesige in sich geschlossene Anwendungen schrieben, die unmittelbar in den Browsern ausgeführt wurden. Und jetzt sind wir wieder an einem Punkt angelangt, an dem wir uns voller Enthusiasmus daran begeben, mithilfe der Plattform Node abermals JavaScript auf den Server zu bringen.

(Seufz.)

31.1 Der immerwährende Pendelausschlag

Natürlich wäre es falsch zu glauben, dass diese Pendelbewegungen mit dem Internet begonnen hätten. Vor dem Web gab es bereits die Client-Server-Architekturen. Und davor hatten wir zentrale Minicomputer mit reihenweise »dummen« Terminals. Diese waren ihrerseits die Nachfolger der Mainframes mit den intelligenten »Green-Screen-Terminals« (die den modernen Browsern sehr ähnlich waren) – denen wiederum Computerräume und Lochkarten vorausgegangen waren ...

Und so geht die Geschichte immer weiter. Offenbar können wir uns nicht so recht entscheiden, wo wir die Computerleistung haben wollen. Stattdessen schwanken wir unaufhörlich zwischen Zentralisierung und Verteilung hin und her. Und ich vermute, das wird auch noch eine Weile so bleiben.

Wenn man den Verlauf der IT-Geschichte betrachtet, hat das Web im Grunde genommen überhaupt nichts verändert. Vielmehr war es einfach eine weitere von vielen Pendelstationen im Rahmen eines Bestrebens, das seinen Anfang nahm, bevor die meisten von uns geboren wurden – und fortdauern wird, wenn wir uns schon längst in den Ruhestand verabschiedet haben.

Als Softwarearchitekten müssen wir jedoch vorausschauend arbeiten und uns

langfristig orientieren. Die vorerwähnten Pendelausschläge adressieren lediglich kurzlebige Belange, die wir daher aus dem zentralen Kern unserer Geschäftsregeln heraushalten sollten.

An dieser Stelle möchte ich Ihnen kurz die Geschichte des Unternehmens Q erzählen, das ein äußerst populäres persönliches Finanzsystem entwickelt hatte. Es handelte sich dabei um eine Desktop-App mit einem sehr gefälligen und hilfreichen GUI. Ich nutzte die Anwendung selbst und fand sie großartig.

Dann kam das Internet – und schon mit dem nächsten Release veränderte das Unternehmen das GUI so, dass es wie ein Browser aussah und sich auch so verhielt. Ich war wie vom Donner gerührt! Welches Marketing-Genie hatte entschieden, dass eine persönliche Finanzsoftware, die auf einem Desktop lief, das Look-and-Feel eines Webbrowsers haben sollte?

Wie Sie sicher schon vermuten, gefiel mir die neue Schnittstelle ganz und gar nicht. Und offenbar ging es den anderen Usern genauso, denn schon nach wenigen weiteren Releases wandte sich Q allmählich wieder von dem browserorientierten Look ab und versah sein persönliches Finanzsystem erneut mit dem normalen Desktop-GUI.

Nun stellen Sie sich einmal vor, Sie wären als Softwarearchitekt in dem Unternehmen Q tätig und irgendein Marketingmensch überzeugt das gehobene Management der Firma davon, dass die gesamte Benutzerschnittstelle verändert werden muss, um mehr nach Internet auszusehen. Was würden Sie tun? Oder vielmehr: Was hätten Sie vor diesem Punkt tun sollen, um Ihre Anwendung vor der Einflussnahme seitens dieses Marketing-Genies zu schützen?

Sie hätten die Geschäftsregeln von Ihrer Benutzerschnittstelle entkoppeln sollen. Ob die Softwarearchitekten des Unternehmens Q so vorgegangen sind, weiß ich leider nicht, allerdings würde ich mir ihre Geschichte nur zu gern einmal anhören. Wäre ich zu jener Zeit dort beschäftigt gewesen, hätte ich sicherlich nachdrücklich dafür plädiert, die Geschäftsregeln von dem GUI zu isolieren – weil man ja nie wissen kann, womit die Marketingleute als Nächstes um die Ecke kommen.

Ein weiteres Beispiel hierfür ist auch das Unternehmen A, das ein großartiges Smartphone anbietet. Erst kürzlich wurde ein Upgrade für dessen »Betriebssystem« releast. (Ist es nicht merkwürdig, dass wir tatsächlich über ein Betriebssystem in einem Telefon sprechen können?) Unter anderem wurde im Rahmen dieses »Betriebssystem«-Upgrades auch das Look-and-Feel sämtlicher Anwendungen verändert. Warum? Vermutlich weil irgendein Marketing-Genie es so wollte.

Nun bin ich kein Experte für die Software in einem solchen Gerät, deshalb kann ich auch nicht beurteilen, ob diese Modifikationen den Programmierern der Apps, die auf einem Smartphone der Firma A laufen, größere Schwierigkeiten bereitet haben. Ich hoffe aber, dass die Softwarearchitekten des Unternehmens A ebenso wie die

Softwarearchitekten der Apps ihre Benutzerschnittstellen und Geschäftsregeln jeweils voneinander isolieren – denn wie erwähnt, es gibt immer irgendwelche Marketingleute, die nur darauf warten, sich auf das nächste bisschen Kopplung zu stürzen, das Sie erschaffen haben.

31.2 Quintessenz

Die Quintessenz des Ganzen ist simpel: Das GUI ist ein Detail. Und das Web ist ein GUI – ergo ist das Web ein Detail. Als Softwarearchitekt liegt es in Ihrem eigenen Interesse, Details hinter Grenzlinien zu platzieren, die sie von Ihrer Kerngeschäftslogik trennen.

Betrachten Sie es doch einmal so: *Das Web ist ein I/O-Gerät.* In den 1960er-Jahren haben wir gelernt, wie wichtig es ist, geräteunabhängige Anwendungen zu schreiben. An der Motivation für die Gewährleistung dieser Unabhängigkeit hat sich seitdem nichts geändert – und das Web bildet da keine Ausnahme.

Oder doch? Man könnte argumentieren, dass ein GUI, wie das Web, so einzigartig und reichhaltig ist, dass es absurd wäre, eine geräteunabhängige Systemarchitektur zu verfolgen. Angesichts der Feinheiten der JavaScript-Validierung oder Drag-and-Drop-AJAX-Aufrufe oder irgendeines der zahlreichen anderen Widgets und Gadgets, die man auf einer Webseite bereitstellen kann, ist es leicht zu argumentieren, dass die Geräteunabhängigkeit unpraktisch ist.

Und in gewisser Weise stimmt das auch. Die »Lebhaftigkeit« der Interaktion zwischen der Anwendung und dem GUI richtet sich spezifisch nach der Art des GUI, das Sie verwenden. Das Zusammenspiel zwischen einem Browser und einer Webanwendung unterscheidet sich von dem zwischen einem Desktop-GUI und seiner Anwendung. Und zu versuchen, dieses Zusammenspiel in der Art zu abstrahieren, wie Geräte aus UNIX abstrahiert werden, scheint kaum möglich zu sein.

Eine andere Grenzlinie zwischen der Benutzerschnittstelle und der Anwendung lässt sich dagegen abstrahieren: Die Geschäftslogik kann als eine Suite von Use Cases betrachtet werden, die jeweils eine oder mehrere Funktionen im Auftrag eines Users ausführen. Jeder Use Case kann auf der Grundlage der Eingabedaten, der ausgeführten Verarbeitungsprozesse und der Ausgabedaten beschrieben werden.

An irgendeinem Punkt des Zusammenspiels zwischen der Benutzerschnittstelle und der Anwendung können die Eingabedaten dann als vollständig betrachtet werden, sodass die Ausführung des Use Case starten kann. Und anschließend können die resultierenden Daten wieder in das Zusammenspiel zwischen Benutzerschnittstelle und Anwendung eingebracht werden.

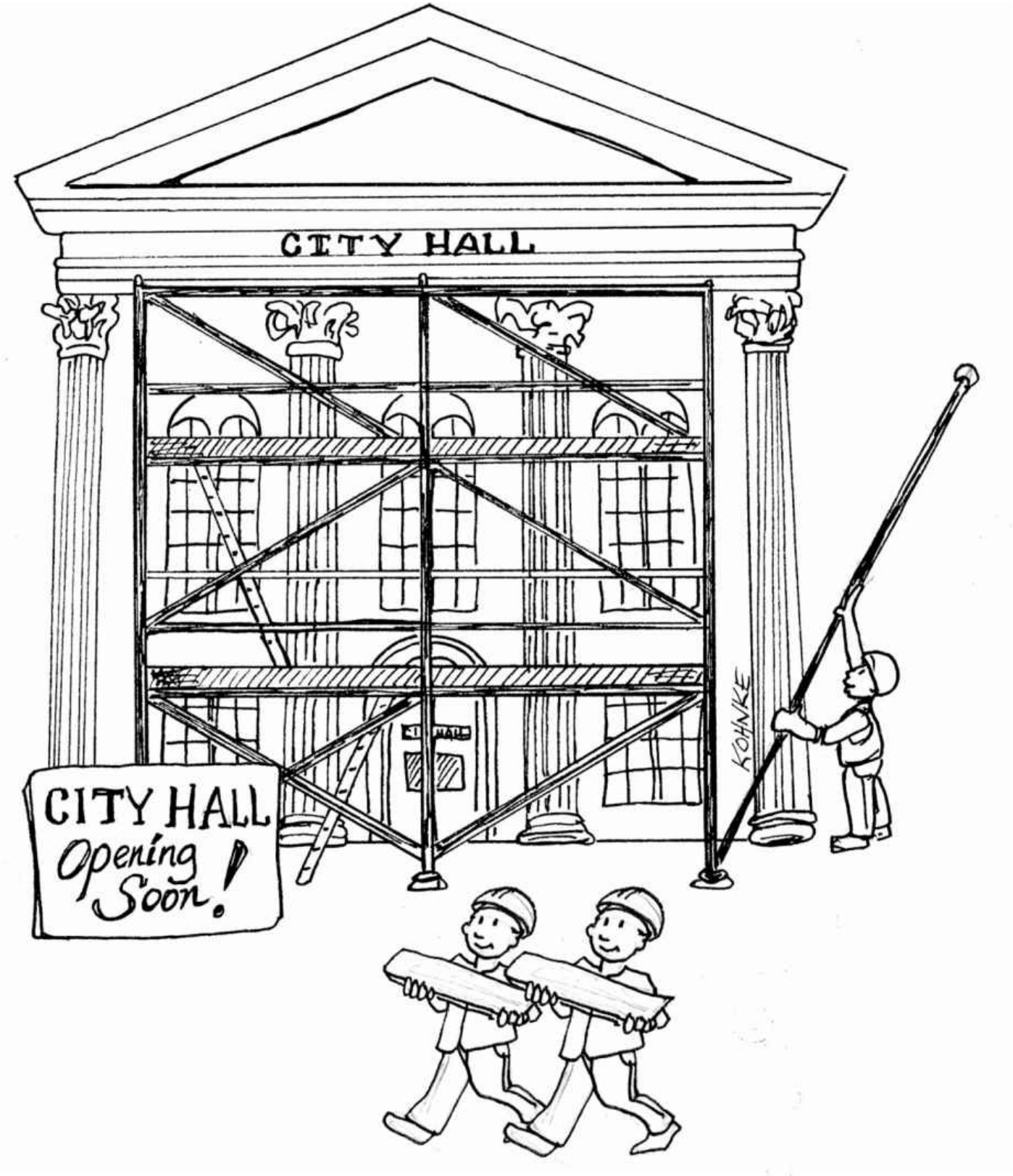
Die vollständigen Eingabedaten sowie die resultierenden Ausgabedaten können in Datenstrukturen platziert und als Eingabe- und Ausgabewerte für einen Prozess verwendet werden, der den Use Case ausführt. Bei diesem Ansatz kann davon ausgegangen werden, dass jeder Use Case das I/O-Gerät der Benutzerschnittstelle in einer geräteunabhängigen Art und Weise betreibt.

31.3 Fazit

Diese Art von Abstraktion ist nicht einfach durchzuführen, und es wird vermutlich auch mehrere Iterationen brauchen, um sie richtig hinzubekommen. Aber es ist machbar. Und da es jede Menge Marketing-Genies auf dieser Welt gibt, fällt es nicht allzu schwer, Gründe dafür zu finden, dass dies in der Regel auch absolut notwendig ist.

Kapitel 32

Ein Framework ist ein Detail



Frameworks sind zunehmend beliebter geworden – und generell ist das auch eine gute Sache. Oftmals werden sie kostenlos angeboten und sind sehr zweckdienlich und leistungsstark.

Dennoch handelt es sich bei Frameworks nicht um Architekturen – auch wenn einige das nur zu gerne wären und auch vorgeben, es zu sein.

32.1 Framework-Autoren

Die meisten Framework-Autoren bieten ihre Werke kostenlos an, weil sie die Community unterstützen möchten. Sie wollen sich erkenntlich zeigen und ihr auf diese Weise etwas zurückgeben, was wirklich sehr lobenswert ist. Unabhängig von ihren edlen Motiven verfolgen sie jedoch nicht immer auch *Ihre* besten Interessen – das können sie gar nicht, weil sie weder Sie noch die Probleme kennen, mit denen Sie konfrontiert sind.

Framework-Autoren wissen lediglich um ihre eigenen Problemstellungen sowie die ihrer Kollegen und Freunde. Und insofern schreiben sie ihre Frameworks vorrangig, um *diese* Probleme zu lösen – und nicht Ihre.

Andererseits ergeben sich dabei natürlich oftmals zahlreiche Überschneidungen – andernfalls wären diese Rahmenstrukturen nicht so beliebt. Und in dem Maße, in dem die adressierten Problemstellungen übereinstimmen, können Frameworks selbstverständlich auch wirklich sehr hilfreich sein.

32.2 Asymmetrische Ehe

Die Beziehung, die Sie mit dem jeweiligen Framework-Autor eingehen, ist außerordentlich asymmetrisch: Sie selbst müssen sich voll und ganz auf das Framework einlassen, der Autor des Frameworks geht hingegen keinerlei Verpflichtung ein.

Denken Sie einmal sorgfältig über diesen Punkt nach. Wenn Sie ein Framework nutzen, lesen Sie sich zunächst die Dokumentation durch, die der Autor zur Verfügung stellt. Darin finden sich nicht nur Hinweise des Autors, sondern auch Empfehlungen von anderen Usern des betreffenden Frameworks bezüglich der Integrationsmöglichkeiten Ihrer Software. Üblicherweise läuft dies darauf hinaus, dass Sie Ihre Architektur um das Framework herum gestalten. Die Empfehlung des Autors lautet, dass Sie Ihre Funktionen von den Basisklassen seines Frameworks ableiten und dessen Funktionen in Ihre Geschäftsobjekte importieren. Er fordert Sie im Wesentlichen also auf, Ihre Anwendung so eng wie möglich an das Framework zu *koppeln*.

Für den Framework-Autor ist die Kopplung an sein eigenes Framework mit keinerlei Risiko verbunden. Im Gegenteil: Er *will* diese Kopplung, weil er letzten Endes die absolute Kontrolle über das Framework behält.

Darüber hinaus liegt es aber auch in seinem ureigensten Interesse, dass *Sie* Ihr gesamtes System an das Framework koppeln, weil es so sehr schwierig wird, es später

wieder davon zu lösen. Nichts verschafft einem Framework-Autor mehr Bestätigung, als wenn User ihre Funktionen bereitwillig und untrennbar von seinen Basisklassen ableiten.

Im Grunde genommen fordert Sie der Autor auf, eine Ehe mit seinem Framework zu schließen – sprich eine bedeutsame, langfristige Bindung mit dieser Rahmenstruktur einzugehen. Und doch geht der Autor seinerseits unter keinen Umständen dieselbe Verpflichtung Ihnen gegenüber ein. Es ist eine Art einseitige Eheschließung: Sie übernehmen alle Risiken und Lasten, während der Framework-Autor überhaupt nichts auf sich nimmt.

32.3 Die Risiken

Was sind die Risiken bei der ganzen Sache? Nun, hier sind einige Aspekte, die Sie bedenken sollten:

- Die Architektur des Frameworks ist häufig nicht besonders sauber. Frameworks neigen dazu, gegen die Abhängigkeitsregel zu verstoßen. Die Framework-Autoren fordern Sie dazu auf, ihren Code in Ihre Geschäftsobjekte zu übernehmen – Ihre Entitäten! Sie wollen, dass ihr Framework in diesen innersten Kreis eingekoppelt wird. Und wenn es da erst mal drin ist, geht es nie wieder heraus. Der Ehering steckt an Ihrem Finger – und wird auf ewig dort bleiben.

- Das Framework mag Ihnen bei der Bereitstellung einiger der anfänglichen Features Ihrer Anwendung behilflich sein. Sobald Ihr Produkt jedoch heranreift, könnte es den Möglichkeiten des Frameworks entwachsen – und wenn Sie sich den Ehering an den Finger haben stecken lassen, dann könnten Sie im Laufe der Zeit feststellen, dass das Framework immer mehr gegen Sie arbeitet.

- Das Framework kann sich in eine Richtung entwickeln, die Sie weniger nützlich finden. Womöglich müssen Sie ständig auf neue Versionen upgraden, die Ihnen trotzdem nicht weiterhelfen. Und vielleicht stellen Sie sogar fest, dass alte Features, die Sie genutzt haben, plötzlich nicht mehr vorhanden sind oder in einer Weise verändert wurden, mit der Sie nur schwer Schritt halten können.

- Es könnte ein neueres, besseres Framework auf den Plan treten, zu dem Sie gerne wechseln würden.

32.4 Die Lösung

Was ist die Lösung?

Gehen Sie keine Ehe mit dem Framework ein!

Selbstverständlich können Sie das Framework *nutzen* – aber koppeln Sie Ihre Software nicht damit. Halten Sie es eine Armlänge auf Abstand. Behandeln Sie das Framework wie ein Detail, das in einen der äußeren Kreise Ihrer Architektur gehört. Lassen Sie es nicht in die inneren Kreise!

Wenn das Framework Sie dazu drängt, Ihre Geschäftsobjekte von seinen Basisklassen abzuleiten, sagen Sie strikt: Nein! Leiten Sie stattdessen Proxies ab und behalten Sie diese in Komponenten, die als *Plug-ins* für Ihre Geschäftsregeln dienen.

Lassen Sie das Framework nicht in Ihren Kerncode. Integrieren Sie es stattdessen unter Einhaltung der Abhängigkeitsregel in Komponenten, die an Ihren Kerncode andocken.

Vielleicht gefällt Ihnen ja zum Beispiel das wirklich gute Dependency-Injection-Framework Spring. Möglicherweise nutzen Sie es, um Ihre Abhängigkeiten automatisch zu erstellen. Das ist auch in Ordnung, nur sollten Sie keine `@autowired`-Notationen in Ihren Geschäftsobjekten verteilen. Ihre Geschäftsobjekte sollten keinerlei Kenntnis von Spring haben.

Stattdessen können Sie Spring verwenden, um Abhängigkeiten in Ihre `Main`-Komponente zu injizieren. Wenn `Main` Kenntnis von Spring hat, ist das in Ordnung, denn sie repräsentiert die unsauberste und tiefschichtigste Komponente in der Softwarearchitektur.

32.5 Hiermit erkläre ich euch zu ...

Es gibt einige Frameworks, mit denen man einfach eine feste Bindung eingehen muss. Wenn Sie beispielsweise C++ verwenden, werden Sie höchstwahrscheinlich die Ehe mit STL eingehen müssen – das lässt sich kaum vermeiden. Sollten Sie hingegen Java nutzen, werden Sie mit ziemlicher Sicherheit die Standard-Library »ehelichen« müssen.

Das ist normal – dennoch sollte es eine *Entscheidung* sein. Es muss Ihnen klar sein, dass Sie, wenn Sie ein Framework in Ihre Anwendung einbringen, für die restliche Lebensdauer dieser Software daran gebunden sein werden. In guten wie in schlechten Tagen, in Krankheit und Gesundheit, in Reichtum und Armut, bis dass der Tod euch scheidet, *werden* Sie dieses Framework nutzen (müssen). Und eine solche Verpflichtung sollte nicht leichtfertig eingegangen werden.

32.6 Fazit

Wenn Sie mit einem Framework konfrontiert werden, versuchen Sie, nicht gleich eine Ehe mit ihm einzugehen. Schauen Sie, ob nicht die Möglichkeit besteht, sich erst mal für eine Weile zu verabreden, bevor Sie den ultimativen Schritt wagen. Halten Sie das Framework so lange wie möglich hinter einer architektonischen Grenze. Vielleicht finden Sie ja einen Weg, die Milch zu bekommen, ohne gleich die Kuh kaufen zu müssen.

Kapitel 33

Fallstudie: Software für den Verkauf von Videos

Jetzt ist es an der Zeit, all die in diesem Buch erläuterten Regeln und Theorien zum Thema Softwarearchitektur einmal anhand einer Fallstudie zu demonstrieren. Der nachstehend beschriebene Beispielfall wird kurz und simpel sein, aber dennoch sowohl die Abläufe als auch die Entscheidungen, die ein guter Softwarearchitekt anbringen würde, in nachvollziehbarer Weise darstellen.

33.1 Das Produkt

Ich habe für diese Fallstudie ein Produkt gewählt, mit dem ich sehr gut vertraut bin: eine Software für eine Website, die Videos verkauft. Ähnlichkeiten mit der Site `cleancoders.com`, auf der ich meine Software-Tutorial-Videos anbiete, sind daher durchaus möglich.

Die Grundidee hinter diesem Konzept ist trivial: Wir haben eine Reihe von Videos, die wir verkaufen möchten – und zwar über das Web und sowohl an Privatpersonen als auch an Geschäftsleute. Privatkunden können die Videos gegen eine Gebühr streamen oder für ein höheres Entgelt downloaden und damit dauerhaft kaufen. Geschäftslizenzen sind dagegen ausschließlich für das Streaming gedacht und werden im Bundle erworben, wobei auch Mengenrabatte möglich sind.

Privatkunden werden normalerweise in Betrachter und Käufer unterschieden. Im Fall der Geschäftskunden werden die Videos meist von Einzelpersonen geordert und dann von anderen Leuten betrachtet.

Die Video-Autoren müssen neben ihren Videodateien und schriftlichen Beschreibungen dazu auch Prüfungsaufgaben, Beispiele für Problemstellungen, mögliche Lösungswege, Quellcode und andere Materialien bereitstellen.

Die Aufgabe der Administratoren besteht darin, neue Videoserien einzupflegen, einzelne Videos zu den Serien hinzuzufügen und ggf. daraus zu entfernen sowie die Preise für die verschiedenen Lizenzen festzulegen.

Der erste Schritt zur Festlegung der grundlegenden Architektur des Systems ist, die Akteure und die Use Cases zu bestimmen.

33.2 Use-Case-Analyse

Abbildung 33.1 zeigt eine typische Use-Case-Analyse.

Die vier Hauptakteure sind hier klar erkennbar. Entsprechend des *Single-Responsibility-Prinzips* stellen sie die primären Quellen für mögliche Systemänderungen dar. Jedes Mal, wenn ein neues Feature ergänzt oder ein vorhandenes Feature modifiziert wird, findet diese Maßnahme statt, um einen dieser vier Akteure zufriedenzustellen. Insofern macht es Sinn, das System so zu gliedern, dass sich eine Modifikation, die einen bestimmten Akteur betrifft, nicht auch auf die anderen Akteure auswirkt.

Natürlich sind die in Abbildung 33.1 dargestellten Use Cases nicht vollzählig,

beispielsweise fehlen diejenigen für die An- und Abmeldung. Dass sie in diesem Schema nicht angegeben sind, ist einfach darauf zurückzuführen, dass im Rahmen dieses Buches lediglich ein überschaubares Beispiel angeführt werden kann – denn die Auflistung sämtlicher möglichen Use Cases würde problemlos ein eigenes Buch füllen.

Beachten Sie die gestrichelt umrandeten Use Cases in der Mitte der Abbildung. Hierbei handelt es sich um *abstrakte*[1] Use Cases. Solche Anwendungsfälle geben allgemeine Richtlinien vor, an denen sich mindestens ein anderer Use Case orientiert. Wie Sie sehen können, erben die Use Cases *Katalog als Betrachter sichten* und *Katalog als Käufer sichten* beide von dem abstrakten Use Case *Katalog sichten*.

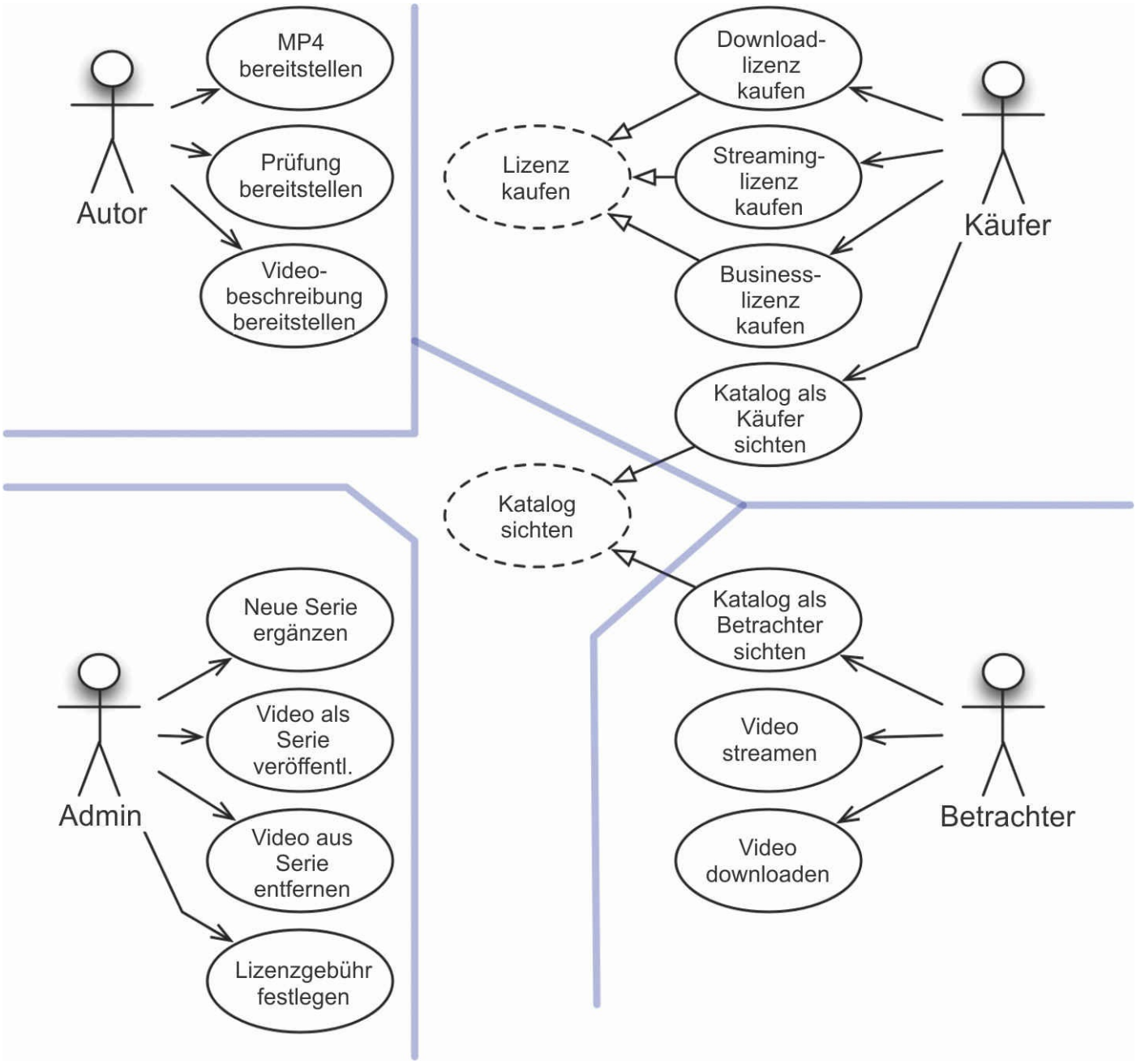


Abb. 33.1: Eine typische Use-Case-Analyse

Einerseits war es hier nicht unbedingt notwendig, die Abstraktion zu erzeugen. Ich hätte den abstrakten Use Case genauso gut auch aus dem Schema weglassen können, ohne dadurch irgendwelche anderen Features des Gesamtprodukts zu beeinträchtigen. Andererseits sind sich die beiden Use Cases allerdings so ähnlich, dass ich es für sinnvoll hielt, dies entsprechend zu berücksichtigen und einen Weg zu finden, sie von vornherein in der Analyse zu vereinheitlichen.

33.3 Komponentenarchitektur

Nachdem die Akteure und Use Cases jetzt bekannt sind, kann als Nächstes eine

vorläufige Komponentenarchitektur erstellt werden (siehe Abbildung 33.2).

Die doppelten Linien in dem Diagramm kennzeichnen wie üblich die architektonischen Grenzen. Darüber hinaus ist auch die typische Unterteilung in `Views`, `Presenters`, `Interactors` und `Controllers` zu erkennen. Außerdem habe ich jede dieser Kategorien nach ihren entsprechenden Akteuren gegliedert.

Die einzelnen in Abbildung 33.2 dargestellten Komponenten repräsentieren jeweils eine potenzielle `.jar`- oder `.dll`-Datei. Und jede Komponente enthält die Views, Presenters, Interaktoren und Controller, die ihr zugewiesen wurden.

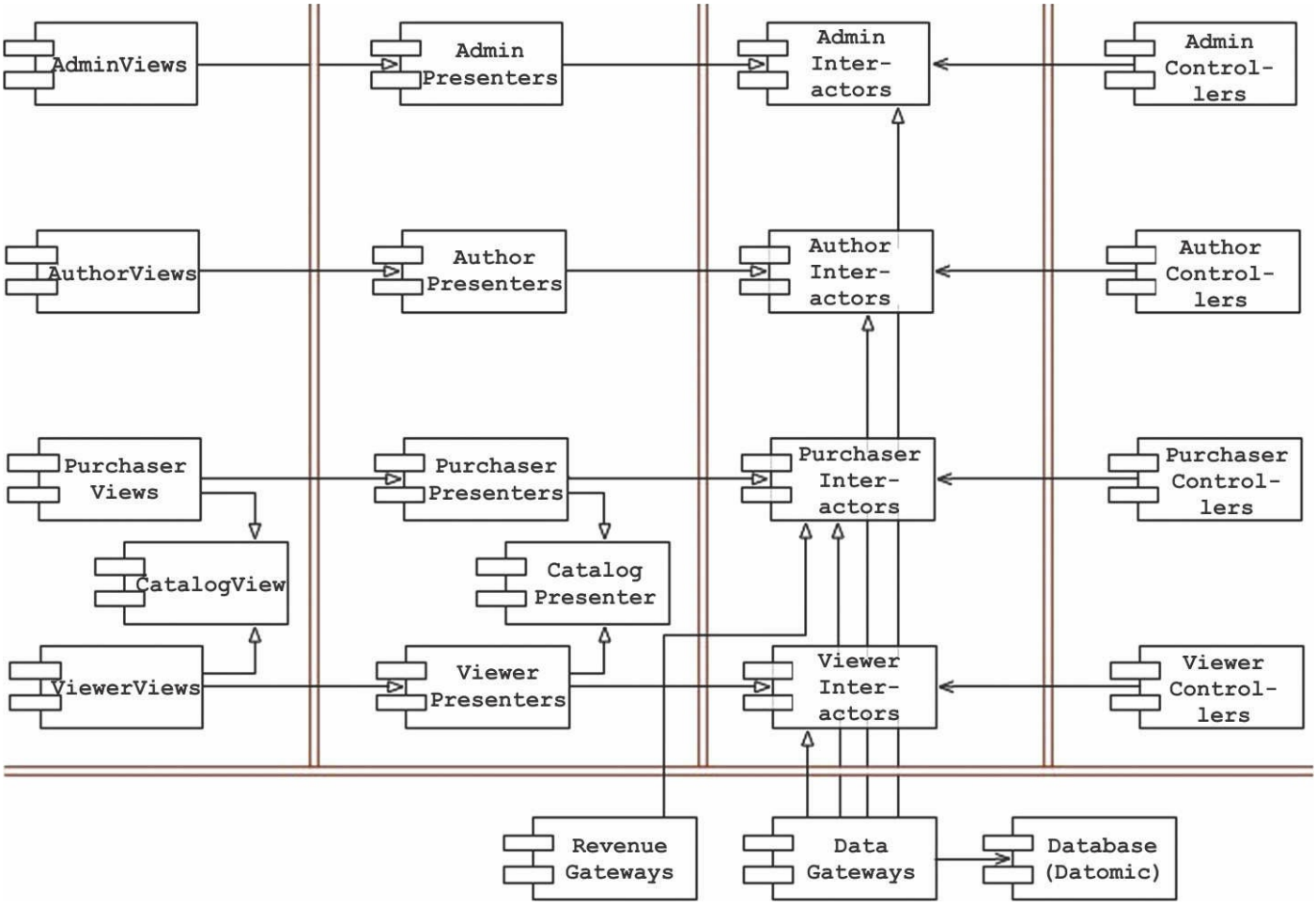


Abb. 33.2: Eine vorläufige Komponentenarchitektur

Beachten Sie die speziellen Komponenten für `CatalogView` und `CatalogPresenter` im unteren Drittel des Schemas, mit denen ich auf den Umgang mit dem abstrakten Use Case *Katalog sichten* abziele. Dabei unterstelle ich, dass diese Views und Presenters in abstrakte Klassen innerhalb der betreffenden Komponenten programmiert werden und dass die erbenden Komponenten `View`- und `Presenter`-Klassen enthalten werden, die ihrerseits von diesen abstrakten Klassen erben.

Würde ich das System tatsächlich in all diese Komponenten gliedern und sie als `.jar`- oder `.dll`-Dateien bereitstellen? Ja und nein. Zweifellos würde ich die Kompilier- und

Build-Umgebung derart unterteilen, dass ich unabhängige Deliverables wie diese erzeugen *könnte*. Zudem würde ich mir auch die Möglichkeit offenhalten, die diversen Komponenten bei Bedarf zu einer insgesamt geringeren Anzahl Deliverables zusammenzufassen. Beispielsweise wäre es angesichts der Gliederung in Abbildung 33.2 ein Leichtes, sie zu fünf `.jar`-Dateien zu kombinieren – je eine für die Views, die Presenters, die Interaktoren, die Controller und die Utilities. So könnte man dann diejenigen Komponenten, die sich höchstwahrscheinlich unabhängig voneinander ändern, auch unabhängig voneinander deployen.

Eine weitere mögliche Unterteilung bestünde darin, die Views und Presenters gemeinsam in einer `.jar`-Datei zusammenzufassen und die Interaktoren, die Controller und die Utilities in einer weiteren. Und noch eine, wenn auch primitivere Gruppierungsvariante wäre die Erzeugung zweier `.jar`-Dateien mit den Views und Presenters in der einen und dem kompletten Rest in der zweiten Datei.

Sich diese Optionen offenzuhalten, ermöglicht die Adaption des System-Deployments entsprechend der Systemänderungen, die sich im Laufe der Zeit ergeben werden.

33.4 Abhängigkeitsmanagement

Der in Abbildung 33.2 dargestellte Kontrollfluss verläuft von rechts nach links. Die Eingabe erfolgt an den Controllern und wird dann von den Interaktoren verarbeitet. Das daraus resultierende Ergebnis wird daraufhin von den Presenters formatiert und schließlich von den Views in der entsprechenden Präsentationsform ausgegeben.

Bemerkenswert ist hier, dass die Pfeile nicht alle von rechts nach links weisen. Tatsächlich zeigen die meisten von ihnen von links nach rechts. Der Grund hierfür ist, dass die Architektur der *Abhängigkeitsregel* folgt. Alle Abhängigkeiten kreuzen die Grenzlinien in eine Richtung und weisen dabei immer auf die Komponenten, die die übergeordneten Richtlinien enthalten.

Beachten Sie auch, dass die *nutzenden Beziehungen* (offene Pfeile) *in Richtung* des Kontrollflusses weisen, während die *Vererbungsbeziehungen* (geschlossene Pfeile) *gegen* den Kontrollfluss gerichtet sind. Hier spiegelt sich die Anwendung des *Open-Closed-Prinzips* wider, durch die sichergestellt wird, dass die Abhängigkeiten in die richtige Richtung verlaufen und dass sich Modifikationen an untergeordneten Details nicht nach oben hin auf die übergeordneten Richtlinien auswirken.

33.5 Fazit

Die in Abbildung 33.2 gezeigte Architektur umfasst zwei Dimensionen der Komponententrennung. Im ersten Fall erfolgt die Separierung der Akteure auf der Grundlage des *Single-Responsibility-Prinzips*, im zweiten Fall basiert sie auf der *Abhängigkeitsregel*. Die Zielsetzung beider Varianten ist, die Komponenten, die sich aus unterschiedlichen Gründen und zu unterschiedlichen Zeitpunkten ändern, voneinander abzugrenzen. Dabei beziehen sich die unterschiedlichen Gründe auf die Akteure und die unterschiedlichen Zeitpunkte auf die verschiedenen Richtlinienebenen.

Wenn Sie Ihren Code auf die beschriebene Weise strukturieren, sind Sie damit jederzeit in der Lage, die Komponenten so zu kombinieren und anzuordnen, wie Sie das System auch tatsächlich deployen möchten. Und sollten sich die Bedingungen bzw. Umstände im System später ändern, können Sie die Zusammenstellung der Komponenten problemlos modifizieren und sie auf jede sinnvolle Art in deploybare Deliverables gruppieren.

---

[1] Hierbei handelt es sich um meine eigene Notation für »abstrakte« Anwendungsfälle. Ein UML-Stereotyp wie `<<abstract>>` würde vermutlich eher dem Standard entsprechen, allerdings halte ich die Einhaltung solcher Standards heutzutage nicht mehr für sehr zweckmäßig.

Kapitel 34
Das fehlende Kapitel

Sämtliche Hinweise und Ratschläge, die Sie in diesem Buch bis hierhin erhalten haben, werden Ihnen sicherlich dabei behilflich sein, Ihre aus Klassen und Komponenten mit klar definierten Grenzen, eindeutigen Verantwortlichkeiten und kontrollierten Abhängigkeiten bestehende Software besser zu gestalten. Der Teufel steckt jedoch in den Implementierungsdetails – und es ist schnell passiert, dass man an dieser letzten Hürde doch noch scheitert, wenn man ihr nicht ebenfalls ein wenig Beachtung schenkt.

Nehmen Sie einmal an, Sie würden eine Online-Buchhandlung aufbauen wollen und einer der geforderten Use Cases wäre, dass die Kunden die Möglichkeit haben, den Status ihrer Bestellungen abzufragen. Das im Folgenden angeführte Beispiel ist zwar in Java geschrieben, die hier zugrunde liegenden Prinzipien lassen sich aber gleichermaßen auch in allen anderen Programmiersprachen anwenden. Lassen wir die saubere Softwarearchitektur an dieser Stelle einmal für einen Moment beiseite und betrachten wir stattdessen zunächst nur eine Reihe von Ansätzen für das Design und die Organisation des Codes.

34.1 Package by Layer

Der erste und vielleicht simpelste Designansatz ist die traditionelle horizontale Schichtenarchitektur, bei der der Code auf der Grundlage dessen separiert wird, was er aus technischer Sicht bewirkt. Diese Vorgehensweise wird häufig als *Package by Layer* (zu Deutsch etwa »Paket nach Layer«) bezeichnet. Abbildung 34.1 zeigt, wie dieses Konzept als UML-Klassendiagramm aussehen könnte.

Die hier dargestellte typische Schichtenarchitektur enthält jeweils einen Layer für den Webcode, für die "Geschäftslogik" und für die Datenpersistenz. Der Code wird also in horizontale Layer separiert, die im Wesentlichen zur Gruppierung gleichartiger Elemente benutzt werden. In einer "strengen Schichtenarchitektur" sollten die Layer nur von dem jeweils unmittelbar angrenzenden untergeordneten Layer abhängig sein. In Java werden Layer üblicherweise als Packages implementiert. Wie in Abbildung 34.1 zu sehen ist, weisen alle Abhängigkeiten zwischen den Layern (*Packages*) nach unten. In diesem Beispiel sind folgende Java-Typen vorhanden:

- `OrdersController`: Ein Webcontroller, ähnlich wie ein Spring-MVC-Controller, der die Anfragen aus dem Web bearbeitet.
- `OrdersService`: Eine Schnittstelle, die die auf Bestellungen bezogene »Geschäftslogik« definiert.
- `OrdersServiceImpl`: Die Implementierung des Bestellservice.[1]
- `OrdersRepository`: Eine Schnittstelle, die definiert, wie der Zugriff auf dauerhafte Bestellinformationen realisiert wird.
- `JdbcOrdersRepository`: Eine Implementierung der `Repository`-Schnittstelle.

In seinem Artikel *»Presentation Domain Data Layering«*[2] erklärt Martin Fowler, dass die Adaption einer solchen Schichtenarchitektur ein guter Anfang ist. Und mit dieser Auffassung steht er nicht allein. Viele der Bücher, Tutorials, Trainingskurse und

Beispielcodes, die es zu diesem Thema gibt, tendieren ebenfalls in diese Richtung. Es ist eine schnelle Methode, um ein System ohne allzu große Komplexität auf die Beine zu stellen und in Gang zu bringen. Wie Martin Fowler betont, ist das Problem dabei, dass Sie dann, wenn Ihre Software an Umfang und Komplexität zunimmt, schnell feststellen werden, dass drei große Behältnisse für Ihren Code nicht ausreichen und Sie deshalb über eine weitere Modularisierung nachdenken müssen.

Ein zweites Problem ist, dass eine Schichtenarchitektur, wie Uncle Bob bereits ausgeführt hat, nichts über die Geschäftsdomäne »herausschreit«. Wenn Sie den Code für zweischichtige Architekturen aus zwei sehr unterschiedlichen Geschäftsbereichen nebeneinanderstellen, werden sich diese höchstwahrscheinlich erschreckend ähnlich sehen: Web, Services und Repositories. Und dann gibt es da noch ein Problem mit den Schichtenarchitekturen, zu dem ich aber erst später kommen werde.

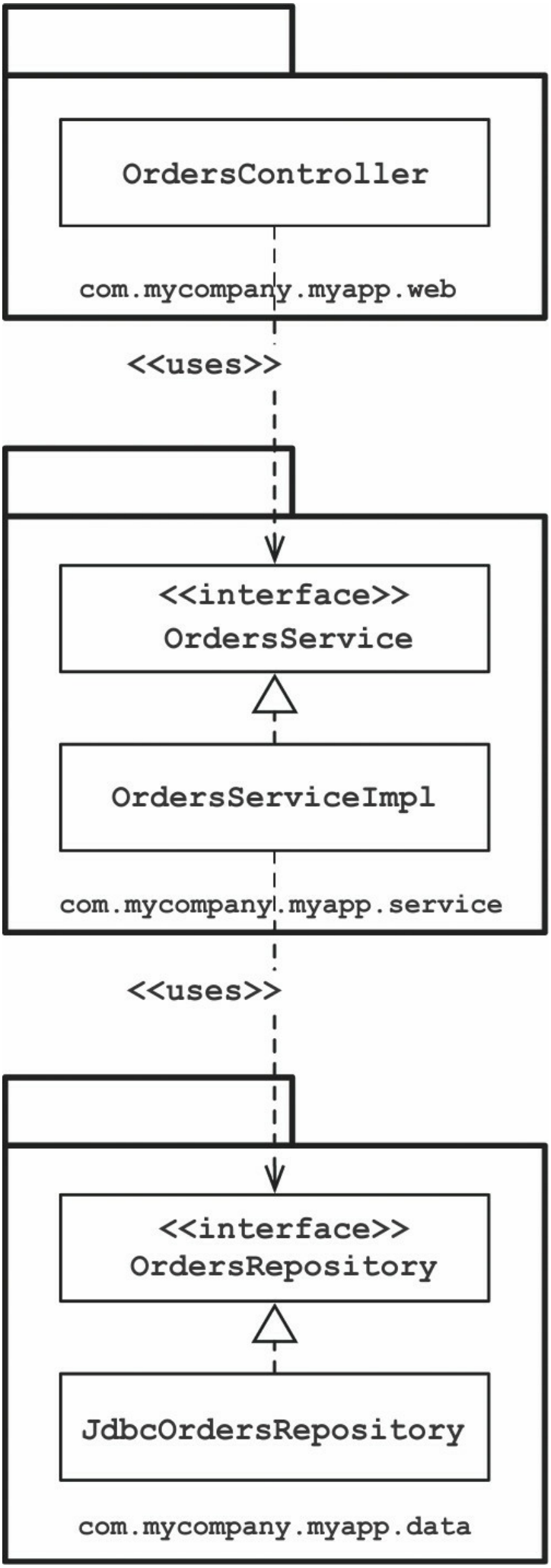

OrdersController
com.mycompany.myapp.web
<<uses>>
<<interface>>
OrdersService
OrdersServiceImpl
com.mycompany.myapp.service
<<uses>>
<<interface>>
OrdersRepository
JdbcOrdersRepository
com.mycompany.myapp.data

Abb. 34.1: *Package by Layer*

34.2 Package by Feature

Eine andere Möglichkeit, Ihren Code zu organisieren, ist die Adaption eines *Package by Feature*-Stils (zu Deutsch etwa »Paket nach Merkmal«). Hierbei handelt es sich um eine vertikale Analyse, die auf verwandten Merkmalen, Domänenkonzepten oder, um die Terminologie des Domain-driven Designs zu bemühen, Aggregate Roots (primären Objekten) basiert. In den typischen Implementierungen, die mir geläufig sind, werden alle Typen in ein einziges Java-Package platziert, dessen Namensgebung das gruppierte Konzept erkennen lässt.

Bei diesem Ansatz, wie in Abbildung 34.2 dargestellt, finden sich dieselben Schnittstellen und Klassen wie zuvor, allerdings sind sie nun alle in einem einzigen Java-Package zusammengefasst und nicht mehr auf drei Packages aufgeteilt.

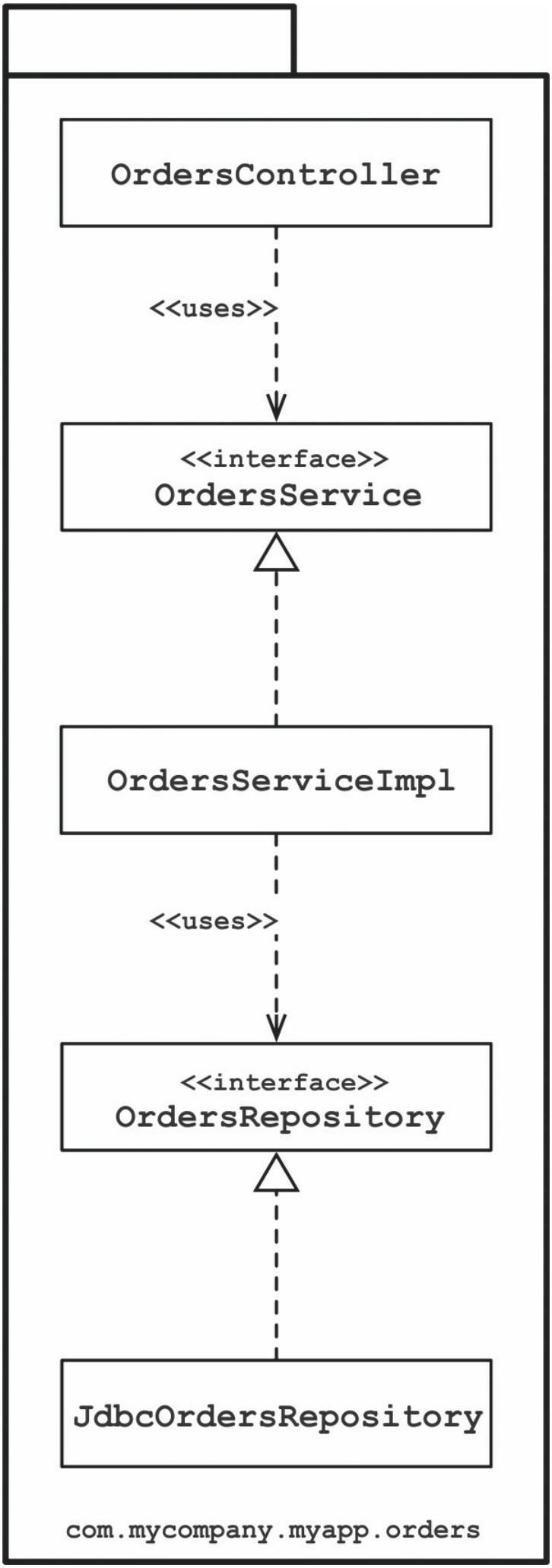

OrdersController
<<uses>>
<<interface>>
OrdersService
OrdersServiceImpl
<<uses>>
<<interface>>
OrdersRepository
JdbcOrdersRepository
com.mycompany.myapp.orders

Abb. 34.2: *Package by Feature*

Ich habe schon oft erlebt, dass Entwicklungsteams Probleme mit dem horizontalen Layering (*Package by Layer*) hatten und deshalb zum vertikalen Layering (*Package by Feature*) wechselten. Meiner Meinung nach sind beide Verfahren suboptimal. Wenn Sie dieses Buch bis hierhin durchgelesen haben, wird Ihnen vermutlich der Gedanke kommen, dass das doch deutlich besser gehen müsste – und damit liegen Sie vollkommen richtig.

Hierbei handelt es sich um ein sehr simples Refactoring des *Package by Layer*-Stils, dennoch »schreit« die übergeordnete Organisation des Codes nun etwas über die Geschäftsdomäne heraus. In diesem Fall ist gleich zu erkennen, dass die Codebasis etwas mit Bestellungen statt mit dem Web, Services und Repositories zu tun hat. Ein weiterer Vorteil besteht hierbei darin, dass es potenziell einfacher ist, sämtlichen Code aufzuspüren, den Sie für den Fall, dass sich der `View Orders`(*Bestellungen ansehen*)-Use-Case ändert, anpassen müssen. Alles dafür Notwendige ist in einem einzigen Java-Paket enthalten und nicht mehr überall verteilt.[3]

34.3 Ports and Adapters

Wie Uncle Bob richtig sagte, zielen Ansätze wie »Ports and Adapters«, die »hexagonale Architektur«, »Grenzen, Controller, Entitäten« und so weiter auf die Errichtung von Systemarchitekturen ab, in denen geschäfts-/domänenfokussierter Code von den technischen Implementierungsdetails wie Frameworks und Datenbanken separiert wird und unabhängig bleibt. Zusammengefasst zeigen sich häufig solche wie in Abbildung 34.3 dargestellten Codebasen, die aus einem »Innenbereich« (der *Domäne*) und einem »Außenbereich« (der *Infrastruktur*) bestehen.

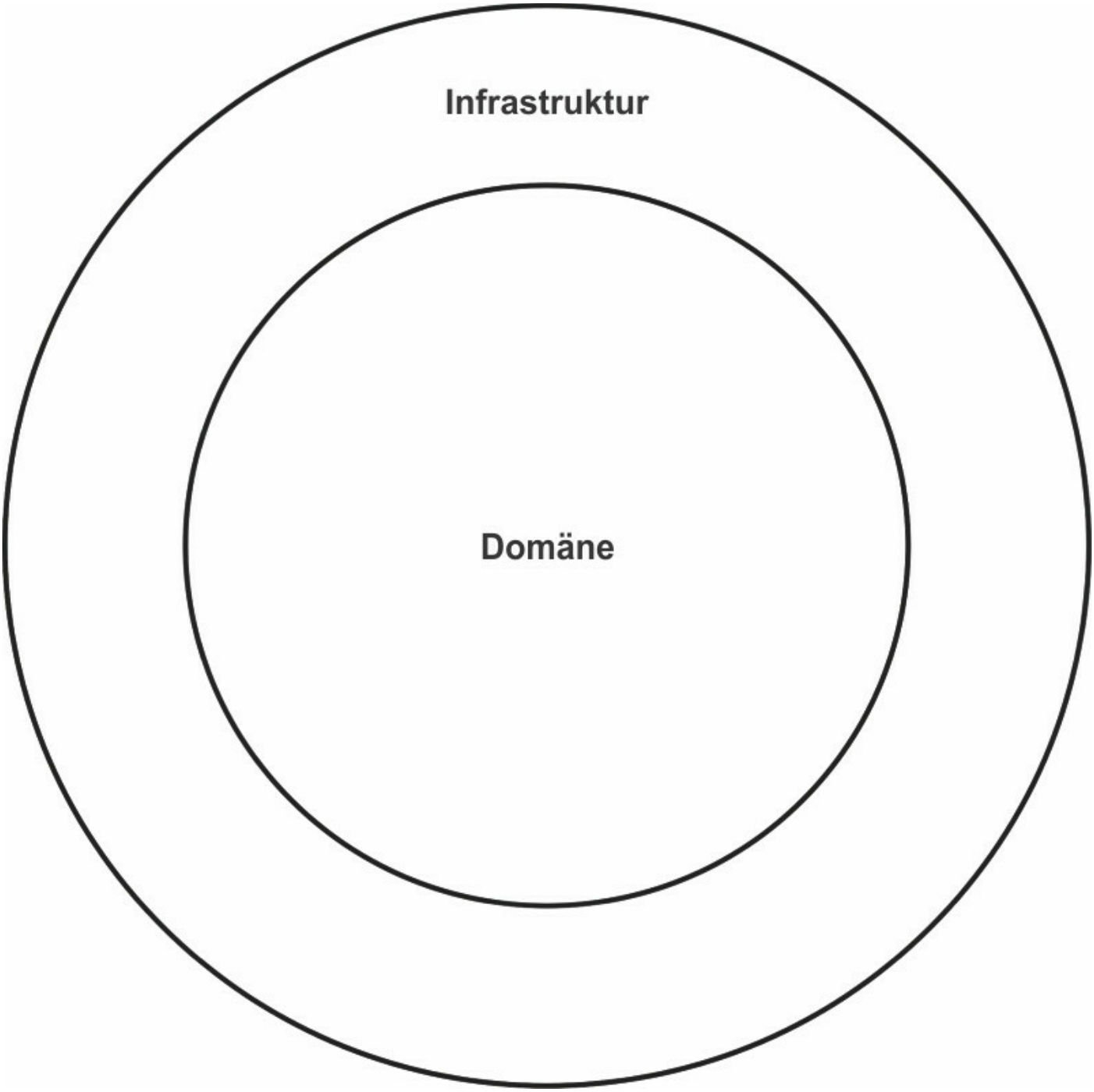


Abb. 34.3: Eine Codebasis mit einem *Innenbereich* und einem *Außenbereich*

Der »Innenbereich« beherbergt alle Domänenkonzepte, während der »Außenbereich« die Interaktionen mit der äußeren Umgebung enthält (z.B. UIs, Datenbanken, Integrationen von Drittanbietern). Die wichtigste Regel ist hier, dass der »Außenbereich« vom »Innenbereich« abhängig ist – niemals umgekehrt! Abbildung 34.4 zeigt eine mögliche Variante, wie der Use Case `View Orders` implementiert werden könnte.

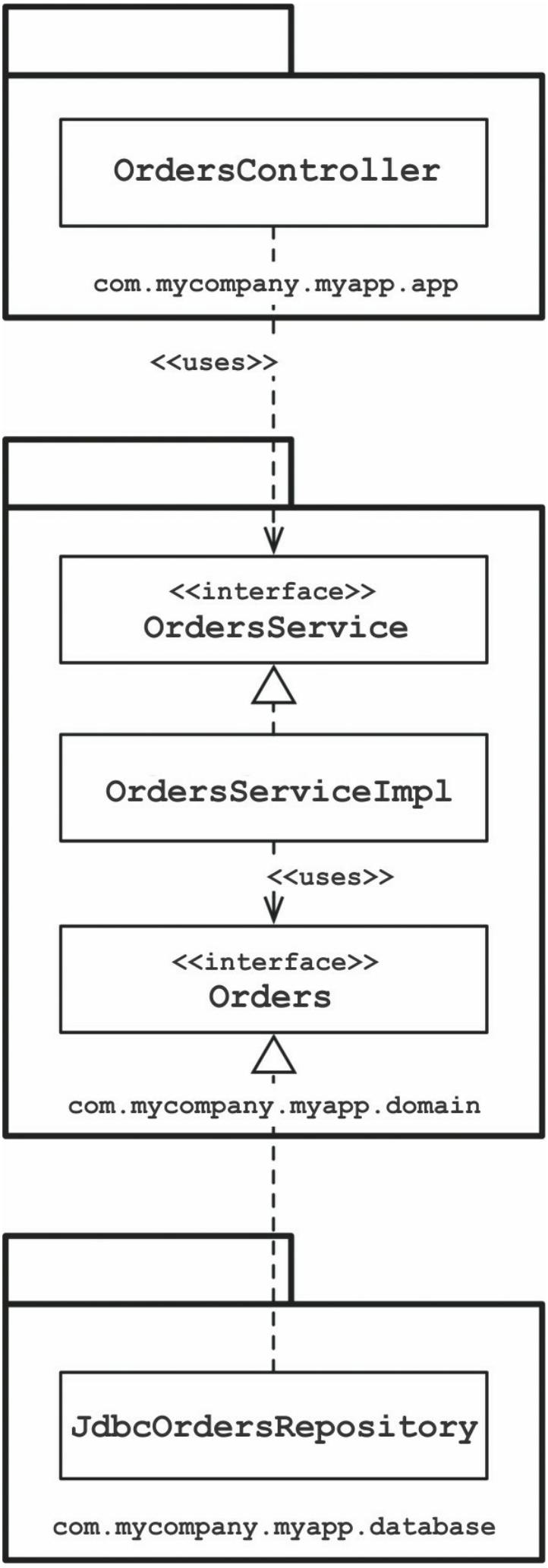

OrdersController
com.mycompany.myapp.app
<<uses>>
<<interface>>
OrdersService
OrdersServiceImpl
<<uses>>
<<interface>>
Orders
com.mycompany.myapp.domain
JdbcOrdersRepository
com.mycompany.myapp.database

Abb. 34.4: Der Use Case `View Orders`

Das Package `com.mycompany.myapp.domain` befindet sich hier im »Innenbereich«, die übrigen Packages sind im »Außenbereich«. Beachten Sie, dass die Abhängigkeiten zum »Innenbereich« hin ausgerichtet sind. Der aufmerksame Leser wird bemerken, dass die Komponente `OrdersRepository` aus den vorangegangenen Diagrammen in diesem Schema einfach in `Orders` umbenannt wurde. Diese Namensänderung wurde der Welt des Domain-driven Designs entliehen, wo die Empfehlung gilt, dass die Benennung aller Elemente im »Innenbereich« entsprechend der »ubiquitären Domänensprache« erfolgen sollte – also in der Sprache, die in der Softwareerstellung allgemeine Verwendung findet. Oder anders ausgedrückt: Wenn wir über die Domäne reden, sprechen wir von `Orders` (den Bestellungen), und nicht vom `OrdersRepository` (der Bestellverwaltung).

Natürlich soll an dieser Stelle nicht unerwähnt bleiben, dass es sich hier um die vereinfachte Version eines möglichen UML-Klassendiagramms handelt. Elemente wie Interaktoren und Objekte für das Marshalling der Daten über die Abhängigkeitsgrenzen hinweg fehlen in diesem Fall.

34.4 Package by Component

Obwohl ich voll und ganz hinter den Ausführungen zu den Themen SOLID, REP, CCP und CRP sowie den meisten Empfehlungen in diesem Buch stehe, komme ich hinsichtlich der Frage, wie der Code organisiert werden sollte, doch zu einem etwas anderen Schluss. Deshalb möchte ich Ihnen im Folgenden noch eine weitere Option vorstellen, die ich als *Package by Component* (zu Deutsch etwa »Paket nach Komponente«) bezeichne. Als Hintergrundinformation möchte ich vorausschicken, dass ich den größten Teil meiner Karriere damit zugebracht habe, hauptsächlich in Java geschriebene Enterprise-Software für eine Reihe verschiedener Geschäftsbereiche zu erstellen. Auch diese Softwaresysteme haben eine immense Entwicklung durchlaufen. Sehr viele davon sind webbasiert, bei anderen handelt es sich um Client-Server-,[4] verteilte, nachrichtenbasierte oder sonstige Systeme. Aber auch wenn sich die Technologien unterscheiden, haben die meisten dieser Softwaresysteme doch eins gemeinsam: Ihre Architektur baut in der Regel auf einem traditionellen Schichtenmodell auf.

Ich habe ja bereits einige Gründe dafür genannt, warum Schichtenarchitekturen als weniger geeignet betrachtet werden können – aber das ist noch nicht die ganze Geschichte. Der Zweck einer Schichtenarchitektur besteht darin, Code, der eine vergleichbare Funktion erfüllt, gemeinsam abzugrenzen. Alles, was mit dem Web zu tun hat, wird von der Geschäftslogik getrennt, die wiederum vom Datenzugriff

separiert wird. Wie Sie anhand des UML-Klassendiagramms sehen können, entspricht ein Layer aus Sicht der Implementierung einem Java-Package. Aus der Perspektive des Codezugriffs betrachtet, muss die `OrdersService`-Schnittstelle als `public` (öffentlich) gekennzeichnet sein, damit die Komponente `OrdersController` von ihr abhängig sein kann. In gleicher Weise muss die Schnittstelle `OrdersRepository` ebenfalls als `public` markiert sein, damit sie außerhalb des Repository-Packages für die `OrdersServiceImpl`-Klasse sichtbar ist.

In einer strengen Schichtenarchitektur sollten die Abhängigkeitspfeile stets nach unten weisen, wobei die Layer jeweils nur von dem nächsten benachbarten Layer darunter abhängig sind. So entsteht ein schöner, sauberer azyklischer Abhängigkeitsgraph, der durch die Einführung einiger Regeln zu der Frage erreicht wird, wie Elemente in einer Codebasis voneinander abhängen sollten. Das große Problem dabei ist, dass man hier schummeln und auch ein paar unerwünschte Abhängigkeiten einbringen kann, aber dennoch einen schönen azyklischen Abhängigkeitsgraphen erhält.

Nehmen Sie einmal an, Sie stellen ein neues Teammitglied ein und beauftragen den Neuankömmling mit der Implementierung eines weiteren bestellungsbezogenen Use Cases. Um Eindruck zu machen, möchte er den Anwendungsfall natürlich so schnell wie nur irgend möglich implementieren – und schon nach ein paar Minuten Denkpause bei einer Tasse Kaffee fällt ihm auf, dass bereits eine `OrdersController`-Klasse existiert, die nach seinem Dafürhalten genau der richtige Ort für den Code der neuen `Orders`-orientierten Webpage sein könnte. Allerdings braucht die Site einige Bestelldaten aus der Datenbank. Plötzlich geht dem neuen Mitarbeiter ein Licht auf: »Oh, da gibt es ja auch schon eine `OrdersRepository`-Schnittstelle. Dann kann ich die Implementierung ja einfach per Dependency Injection in meinen Controller leiten. Perfekt!« Und bereits nach wenigen Minuten Tipparbeit läuft die Webseite – aber das daraus resultierende UML-Klassendiagramm sieht nun wie in Abbildung 34.5 aus.

Die Abhängigkeitspfeile weisen immer noch nach unten, darüber hinaus umgeht `OrdersController` nun aber zusätzlich für einige Use Cases die Komponente `OrdersService`. Diese Anordnung wird häufig als *Relaxed Layered Architecture* (zu Deutsch etwa »lässige Schichtenarchitektur«) bezeichnet, weil es den Layern gestattet ist, ihre direkt angrenzenden Nachbarn zu überspringen. In manchen Situationen wird genau das auch bezweckt, z.B. wenn Sie versuchen, das *CQRS-Pattern*[5] (**C**ommand **Q**uery **R**esponsibility **S**egregation, zu Deutsch etwa »Kommando-Abfrage-Zuständigkeit-Trennung«) anzuwenden. In vielen anderen Fällen ist die Umgehung des Layers, der die Geschäftslogik enthält, jedoch nicht erwünscht – insbesondere dann nicht, wenn diese Geschäftslogik dafür zuständig ist, beispielsweise den autorisierten Zugriff auf individuelle Datensätze sicherzustellen.

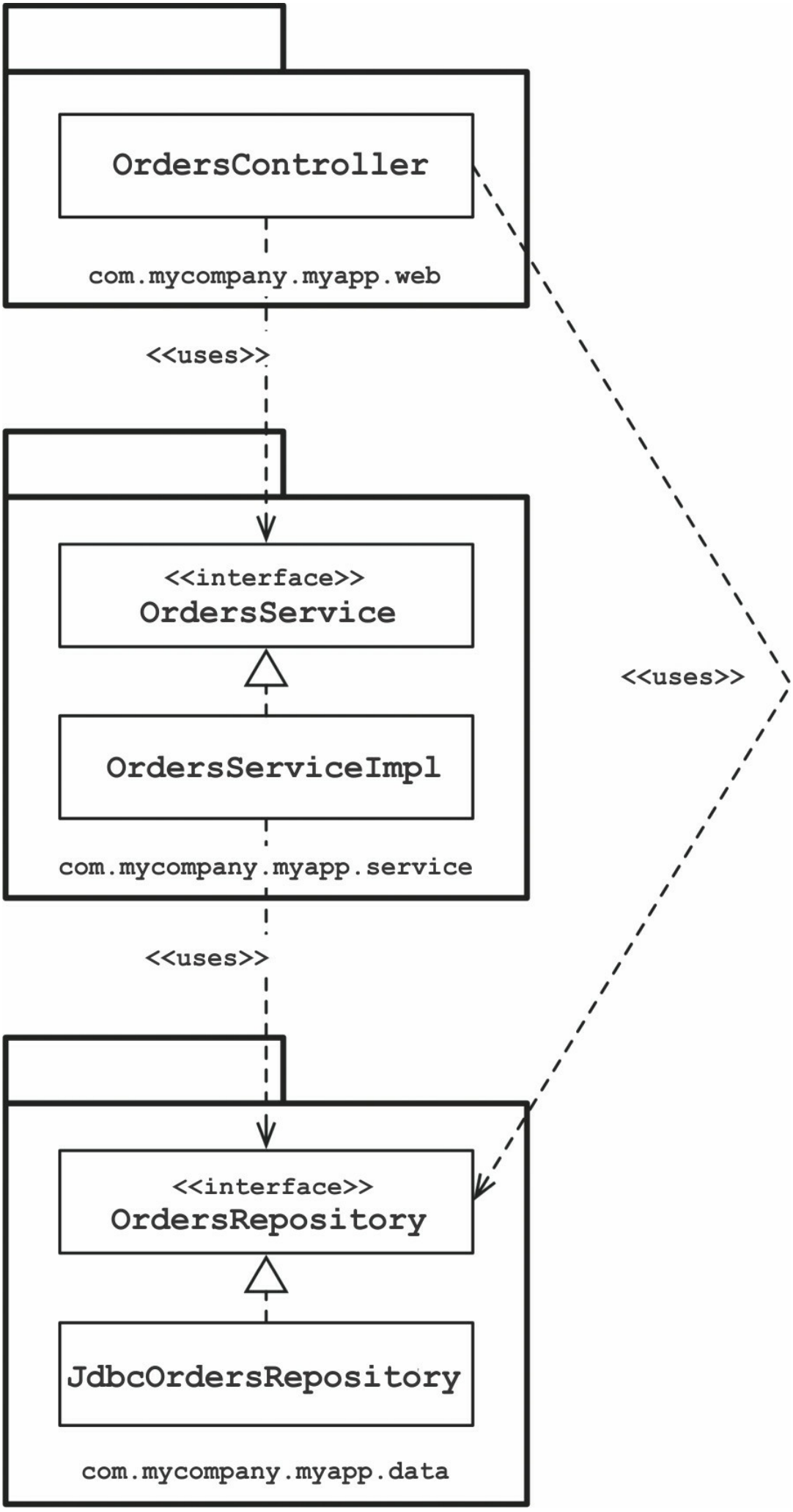

OrdersController
com.mycompany.myapp.web
<<uses>>
<<interface>>
OrdersService
OrdersServiceImpl
com.mycompany.myapp.service
<<uses>>
<<uses>>
<<interface>>
OrdersRepository
JdbcOrdersRepository
com.mycompany.myapp.data

Abb. 34.5: *Relaxed Layered Architecture*

Der neue Use Case funktioniert also zwar, er ist aber vielleicht nicht in der Form implementiert, wie man das erwartet hätte. Ich habe so etwas in vielen Teams erlebt, die ich als Berater betreut habe, und es fällt normalerweise erst dann auf, wenn die Teams anfangen, die Struktur ihrer Codebasis zu visualisieren – oftmals zum allerersten Mal überhaupt.

Was hier vonnöten ist, ist eine Richtlinie – ein architektonisches Prinzip, das etwa Folgendes besagt: »Webcontroller sollten niemals direkt auf Repositories zugreifen.« Bleibt natürlich die Frage der Umsetzung. Viele Teams, mit denen ich zu tun hatte, stehen einfach auf dem Standpunkt: »Wir setzen bei der Einhaltung dieses Prinzips auf Disziplin und Code-Reviews, weil wir unseren Softwareentwicklern vertrauen.« So viel Zuversicht ist wirklich schön – aber wir wissen doch alle, was passiert, wenn die Budgets langsam zur Neige gehen und die Deadlines immer näher kommen.

Eine deutlich kleinere Gruppe von Teams verlässt sich in dieser Hinsicht lieber auf statische Analysetools (z.B. NDepend, Structure101, Checkstyle etc.), um die Systemstruktur auf architektonische Verstöße zur Build-Zeit zu prüfen und sie automatisch zu beheben. Vielleicht haben Sie ja selbst schon Erfahrung mit solchen Richtlinien gemacht: Sie stellen sich normalerweise als reguläre Ausdrücke oder Platzhalterstrings dar, die angeben, dass »Typen im Package `**/web` nicht auf Typen in `**/data`« zugreifen sollen, und werden nach jedem Kompilierungsschritt ausgeführt.

Diese Vorgehensweise ist ein wenig plump, kann aber durchaus ihren Zweck erfüllen, indem Verstöße gegen die Architekturprinzipien, die Sie als Team definiert haben, gemeldet werden und (so steht zumindest zu hoffen) nicht in den Build gelangen. Das Problem mit beiden Ansätzen ist, dass sie fehlbar sind und die Feedbackschleife länger ist, als sie sein sollte. Und sofern es nicht kontrolliert wird, kann dieses Verfahren eine Codebasis auch in einen »Big Ball of Mud«[6] (zu Deutsch etwa »große Schlammkugel«) verwandeln. Ich persönlich bevorzuge daher, wenn irgend möglich, den Einsatz des Compilers, um meine Architekturvorstellungen umzusetzen.

Und damit kommen wir nun wieder zur *Package by Component*-Option. Dieser hybride Ansatz richtet sich auf alles, was wir bisher gesehen haben, mit dem Ziel, alle Verantwortlichkeiten im Zusammenhang mit einer einzelnen grobkörnigen Komponente in einem einzelnen Java-Package zu bündeln. Dabei geht es darum, eine servicezentrierte Sicht auf ein Softwaresystem einzunehmen – etwas, das wir auch bei Microservice-Architekturen sehen. Auf dieselbe Art, wie *Ports and Adapters* das Web lediglich als einen weiteren Liefermechanismus behandeln, wird beim *Package by Component*-Ansatz die Benutzerschnittstelle von diesen grobkörnigen Komponenten getrennt gehalten. Abbildung 34.6 illustriert, wie der `View Orders`-Use-Case aussehen

könnte.

Im Wesentlichen werden hier die »Geschäftslogik« und der Persistenzcode in einem einzelnen Element gebündelt, das ich als *Komponente* bezeichne. Uncle Bob hat seine Definition von einer »Komponente« bereits in Kapitel 12 dieses Buches erläutert und festgestellt:

> *Komponenten sind Deployment-Einheiten. Sie repräsentieren die kleinsten Entitäten, die als Teil eines Systems deployt werden können. In Java sind dies* `.jar`*-Dateien.*

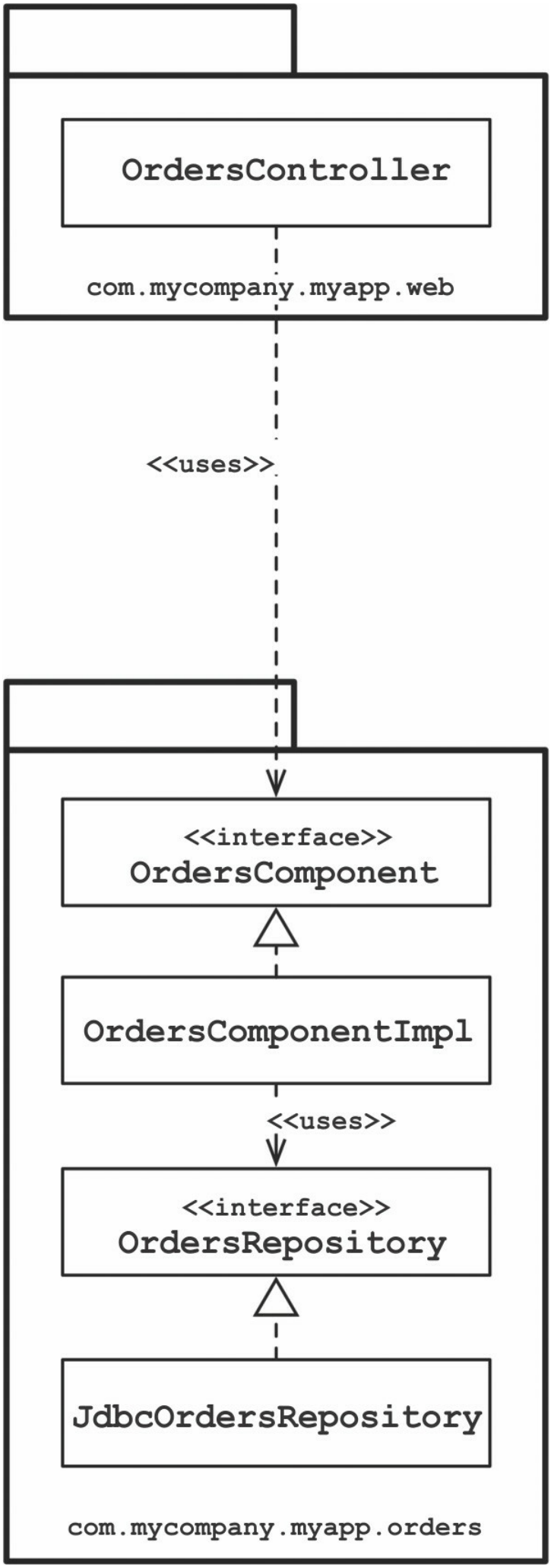

OrdersController
com.mycompany.myapp.web
<<uses>>
<<interface>>
OrdersComponent
OrdersComponentImpl
<<uses>>
<<interface>>
OrdersRepository
JdbcOrdersRepository
com.mycompany.myapp.orders

Abb. 34.6: `View Orders`-Use-Case

Meine Definition einer Komponente lautet etwas anders: »Eine Gruppierung verwandter Funktionalität hinter einer netten, sauberen Schnittstelle, die sich in einer Ausführungsumgebung wie einer Anwendung befindet.« Diese Definition entstammt meinem »C4 Software Architecture Model«[7], das eine einfache hierarchische Denkweise hinsichtlich der statischen Strukturen eines Softwaresystems in Bezug auf Container, Komponenten und Klassen (oder Code) darstellt. Es besagt, dass ein Softwaresystem aus einem oder mehreren Containern (z.B. Webanwendungen, mobilen Apps, Standalone-Anwendungen, Datenbanken, Dateisystemen) besteht, von denen jeder eine oder mehrere Komponenten enthält, die wiederum von einer oder mehreren Klassen (oder Code) implementiert werden. Ob sich jede Komponente in einer separaten `.jar`-Datei befindet, ist eine orthogonale Frage.

Ein maßgeblicher Vorteil des *Package by Component*-Ansatzes ist, dass Sie, wenn Sie Code schreiben, der etwas mit `Orders` anfangen muss, nur einen Ort aufsuchen müssen – die Komponente `OrdersComponent`. Innerhalb dieser Komponente bleibt die Trennung der einzelnen Bestandteile weiterhin erhalten, sodass die Geschäftslogik von der Datenpersistenz separiert ist – was aber wiederum ein Implementierungsdetail der Komponente darstellt, von dem die Verbraucher nichts wissen müssen. Im Endeffekt ist dies mit dem vergleichbar, was Sie möglicherweise erreichen würden, wenn Sie eine an Microservices oder Services orientierte Architektur adaptieren würden: eine separate Komponente `OrdersService`, die alles kapselt, was mit der Bestellbearbeitung zu tun hat. Der Hauptunterschied ist hier der Entkopplungsmodus. Sie können sich gut definierte Komponenten in einer monolithischen Anwendung als ein Sprungbrett zu einer Microservices-Architektur vorstellen.

34.5 Der Teufel steckt in den Implementierungsdetails

Auf den ersten Blick sehen die vier Ansätze nach sehr verschiedenen Möglichkeiten der Codeorganisation aus und könnten daher gegebenenfalls auch als unterschiedliche Architekturstile betrachtet werden. Sobald sich falsche Implementierungsdetails einschleichen, zeigt sich jedoch, dass diese Wahrnehmung nicht ganz richtig ist.

Eine Sache, die mir regelmäßig auffällt, ist eine übermäßig liberale Verwendung des `public`-Zugriffsmodifikators in Sprachen wie Java. Es scheint fast so, als ob wir Softwareentwickler das Schlüsselwort `public` instinktiv nutzen, ohne weiter darüber nachzudenken. Irgendwie hat es sich in unserem Muskelgedächtnis manifestiert. Wenn Sie mir nicht glauben, dann werfen Sie doch mal einen Blick auf die Codebeispiele für

Bücher, Tutorials und Open-Source-Frameworks auf GitHub. Diese Tendenz ist offensichtlich, unabhängig davon, auf welchen architektonischen Stil eine Codebasis abzielt – horizontale Layer, vertikale Layer, Ports und Adapter oder irgendetwas anderes.

Die Kennzeichnung all Ihrer Typen als `public` bringt jedoch letztlich mit sich, dass Sie die Möglichkeiten, die Ihre Programmiersprache in Bezug auf die Kapselung bietet, nicht zu Ihrem Vorteil nutzen. In einigen Fällen gibt es buchstäblich nichts, wodurch sich verhindern ließe, dass jemand Code zur direkten Instanziierung einer konkreten Implementierungsklasse schreibt, womit dann gegen den beabsichtigten Architekturstil verstoßen wird.

34.6 Organisation vs. Kapselung

Man kann dieses Problem aber auch aus einer anderen Perspektive betrachten: Wenn Sie alle Typen in Ihrer Java-Anwendung als `public` deklarieren, sind die Packages einfach ein Organisationsmechanismus (eine Gruppierung wie Ordner), statt der Kapselung zu dienen. Da `public`-Typen von jeder beliebigen Stelle in einer Codebasis aus nutzbar sind, können die Packages effektiv ignoriert werden, weil sie nur sehr wenig realen Wert bieten. Und daraus folgt, dass es, wenn Sie Packages ignorieren (weil sie keine Möglichkeit der Verkapselung oder des Verbergens bieten), nicht wirklich wichtig ist, welchen Architekturstil Sie erzeugen möchten. Wenn wir noch einmal auf die Beispiel-UML-Klassendiagramme zurückblicken, werden die Java-Packages zu einem irrelevanten Detail, wenn alle Typen als `public` gekennzeichnet sind. Im Wesentlichen sind bei übermäßiger Verwendung dieser Kennzeichnung alle vier in diesem Kapitel vorgestellten architektonischen Ansätze genau gleich (siehe Abbildung 34.7).

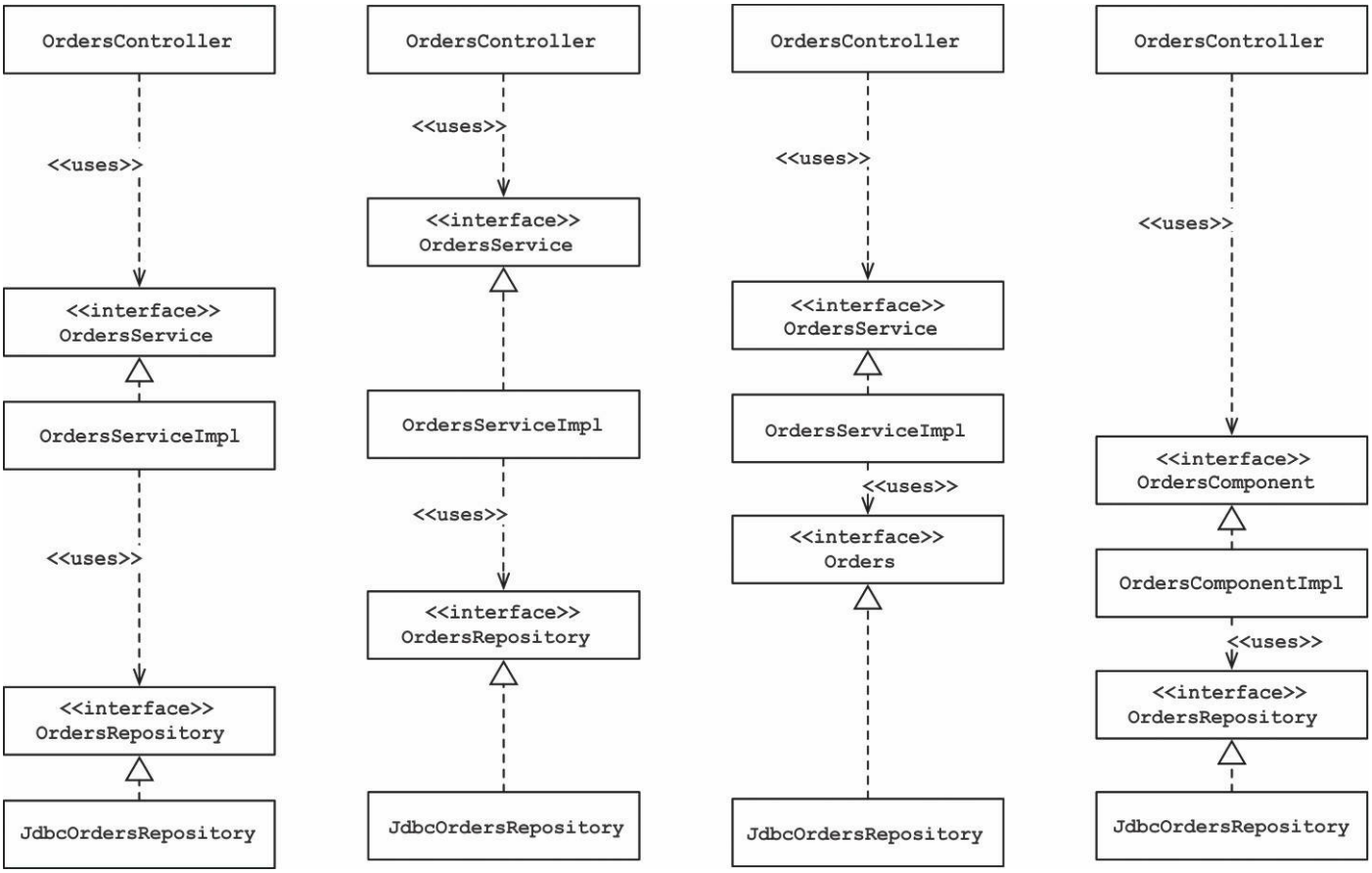


Abb. 34.7: Alle vier architektonischen Ansätze sind gleich.

Betrachten Sie dazu einmal die Pfeile zwischen den einzelnen in Abbildung 34.7 dargestellten Typen: Unabhängig davon, welchen architektonischen Ansatz Sie zu adaptieren versuchen, sind sie alle identisch. Konzeptionell unterscheiden sie sich zwar, syntaktisch sind sie jedoch identisch. Darüber hinaus könnte man argumentieren, dass letztlich nur vier Möglichkeiten existieren, eine traditionelle horizontale Schichtenarchitektur zu beschreiben, wenn Sie alle Typen als `public` deklarieren. Das ist ein netter Trick und normalerweise würde natürlich niemand jemals alle seine Java-Typen als `public` kennzeichnen, aber auch das kann vorkommen – ich habe es selbst schon erlebt.

Die Zugriffsmodifikatoren in Java sind nicht perfekt,[8] sie zu ignorieren würde andererseits jedoch unweigerlich Probleme heraufbeschwören. Die Art und Weise, wie Java-Typen in Packages platziert werden, kann tatsächlich einen enormen Unterschied in Bezug darauf machen, wie zugänglich (oder unzugänglich) diese Typen bei korrekter Anwendung der Java-Zugriffsmodifikatoren sind. Wenn ich die Packages noch einmal zurückhole und die Typen, bei denen der Zugriffsmodifikator restriktiver gestaltet werden kann, durch optisches Ausgrauen kennzeichne, bietet sich ein ziemlich interessantes Bild (siehe Abbildung 34.8).

Von links nach rechts verlaufend müssen die Schnittstellen `OrdersService` und `OrdersRepository` im *Package by Layer*-Ansatz `public` sein, weil sie eingehende Abhängigkeiten von den Klassen außerhalb ihres definierenden Packages aufweisen.

Im Gegensatz dazu können die Implementierungsklassen (`OrdersServiceImpl` und `JdbcOrdersRespository`) restriktiver gestaltet werden (paketgeschützt). Von ihnen muss niemand Kenntnis haben, sie sind ein Implementierungsdetail.

Beim *Package by Feature*-Ansatz bietet die Schnittstelle `OrdersController` den einzigen Eintrittspunkt in das Package, sodass alles andere als `protected` gekennzeichnet werden kann. Der große Nachteil hierbei ist, dass nichts anderes in der Codebasis außerhalb dieses Pakets auf Informationen zugreifen kann, die sich auf Bestellungen (`Orders`) beziehen, es sei denn, es durchläuft den Controller – und das kann entweder wünschenswert sein oder auch nicht.

Beim *Ports and Adapters*-Ansatz weisen die Schnittstellen `OrdersService` und `Orders` eingehende Abhängigkeiten von anderen Packages auf, deshalb müssen sie als `public` gekennzeichnet werden. Auch hier gilt, dass die Implementierungsklassen zur Laufzeit paketgeschützt sein und mit Dependency Injection versehen werden können.

Und schließlich hat die `OrdersComponent`-Schnittstelle beim *Package by Component*-Ansatz eine eingehende Abhängigkeit vom Controller, aber alles andere kann paketgeschützt werden. Je weniger Typen `public` sind, desto geringer ist die Anzahl der potenziellen Abhängigkeiten. Es besteht keine Möglichkeit,[9] dass Code außerhalb dieses Packages die Schnittstelle `OrdersRepository` oder die Implementierung direkt nutzen kann, sodass der Compiler dieses Architekturprinzip zuverlässig umsetzen wird. Das Gleiche lässt sich mit dem Schlüsselwort `internal` auch in .NET erreichen, wenngleich Sie in diesem Fall für jede Komponente eine separate Assembly erstellen müssten.

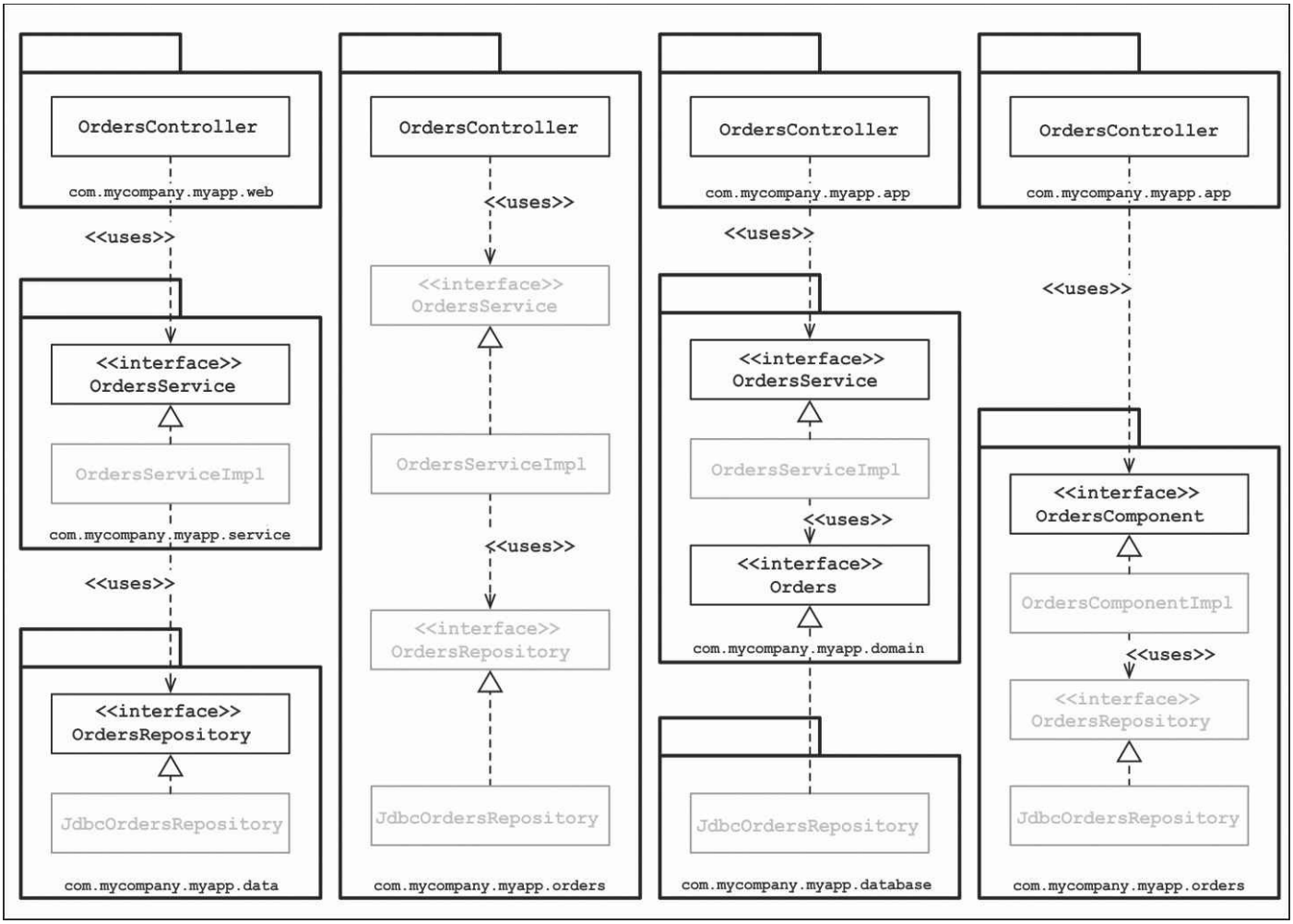


Abb. 34.8: Die ausgegrauten Typen geben an, wo der Zugriffsmodifikator restriktiver gestaltet werden kann.

Um es ganz klar zu sagen: Alles, was ich soeben beschrieben habe, bezieht sich auf eine monolithische Anwendung, in der sich sämtlicher Code in einem einzigen Quellcode-Baum befindet. Wenn Sie eine solche Anwendung erstellen (was viele Leute tun), würde ich Ihnen sicherlich dazu raten, sich zwecks Durchsetzung Ihrer architektonischen Prinzipien auf den Compiler zu berufen, statt sich auf Selbstdisziplin und den Tooleinsatz nach der Kompilierung zu verlassen.

34.7 Andere Entkopplungsmodi

Zusätzlich zu der verwendeten Programmiersprache gibt es oftmals noch andere Möglichkeiten, Ihre Quellcode-Abhängigkeiten zu entkoppeln. Im Fall von Java stehen Ihnen zu diesem Zweck Modul-Frameworks wie OSGi und das neue Java-9-Modulsystem zur Verfügung. Sofern sie richtig angewendet werden, können Sie mithilfe von Modulsystemen zwischen den als `public` und den als `published` (veröffentlicht) gekennzeichneten Typen unterscheiden. Beispielsweise könnten Sie ein Modul `Orders` erzeugen, in dem alle Typen als `public` gekennzeichnet sind, aber

lediglich eine kleine Teilmenge dieser Typen für die externe Nutzung veröffentlichen. Auch wenn es noch eine Weile auf sich warten lassen wird, bin ich dennoch guter Dinge, dass das Java-9-Modulsystem uns ein weiteres Tool an die Hand gibt, um bessere Software zu entwickeln und das Interesse der Leute am Designdenken neu zu entfachen.

Eine weitere Option ist, Ihre Abhängigkeiten auf der Quellcode-Ebene zu entkoppeln, indem Sie den Code über *verschiedene Quellcode-Bäume* verteilen. Wenn wir den *Ports and Adapters*-Ansatz als Beispiel nehmen, könnten wir drei Quellcode-Bäume einrichten:

- Den Quellcode für die geschäfts- und die domänenbezogenen Komponenten (d.h. alles, was nicht von technologischen und Framework-orientierten Entscheidungen abhängig ist): `OrdersService`, `OrdersServiceImpl` und `Orders`
- Den Quellcode für das Web: `OrdersController`
- Den Quellcode für die Datenpersistenz: `JdbcOrdersRepository`

Die letzten beiden Quellcode-Bäume weisen zur Kompilierzeit eine Abhängigkeit von dem Geschäfts- und Domänencode auf, die ihrerseits wiederum keine Kenntnis von dem Web- oder Datenpersistenzcode haben. Aus Sicht der Implementierung lässt sich das erreichen, indem Sie separate Module oder Projekte in Ihrem Build-Tool (z.B. Maven, Gradle, MSBuild) konfigurieren. Idealerweise würden Sie dieses Muster wiederholen und einen separaten Quellcode-Baum für jede einzelne Komponente in Ihrer Anwendung anlegen.

Das ist allerdings eine sehr idealistische Lösung, weil es in Bezug auf die Performance, Komplexität und Instandhaltung Probleme gibt, die mit einer derartigen Aufschlüsselung des Quellcodes in Zusammenhang stehen.

Ein einfacherer Ansatz, den manche Leute für ihren *Ports and Adapters*-Code verfolgen, ist hingegen, nur zwei Quellcode-Bäume anzulegen:

- den Domänencode (den »Innenbereich«)
- den Infrastrukturcode (den »Außenbereich«)

Das Diagramm in Abbildung 34.9 stellt dieses Vorgehen, das viele Softwareentwickler zum Zusammenfassen der *Ports and Adapters*-Architektur einsetzen, in gut überschaubarer Weise dar. Zur Kompilierzeit besteht eine Abhängigkeit der Infrastruktur von der Domäne.

Auch dieser Ansatz zur Organisation von Quellcode wird funktionieren, allerdings müssen Sie sich dabei eines potenziellen Abstrichs bewusst sein, den ich als das »Périphérique-Ports-and-Adapters-Anti-Pattern« bezeichne. Die französische

Hauptstadt Paris ist von einer ringförmigen Autobahn umgeben, dem sogenannten »Boulevard Périphérique«, die das Umfahren der City ermöglicht, ohne mit der komplexen Verkehrssituation in der Stadt selbst in Berührung zu kommen. Wenn Sie Ihren gesamten Infrastrukturcode in einer einzigen Quellcode-Struktur untergebracht haben, ist es in ähnlicher Weise potenziell möglich, dass Infrastrukturcode in einem Bereich Ihrer Anwendung (z.B. ein Webcontroller) Code in einem anderen Bereich Ihrer Anwendung direkt aufruft (z.B. ein Datenbank-Repository), ohne durch die Domain navigieren zu müssen. Dies gilt insbesondere dann, wenn Sie vergessen haben, entsprechende Zugriffsmodifikatoren auf diesen Code anzuwenden.

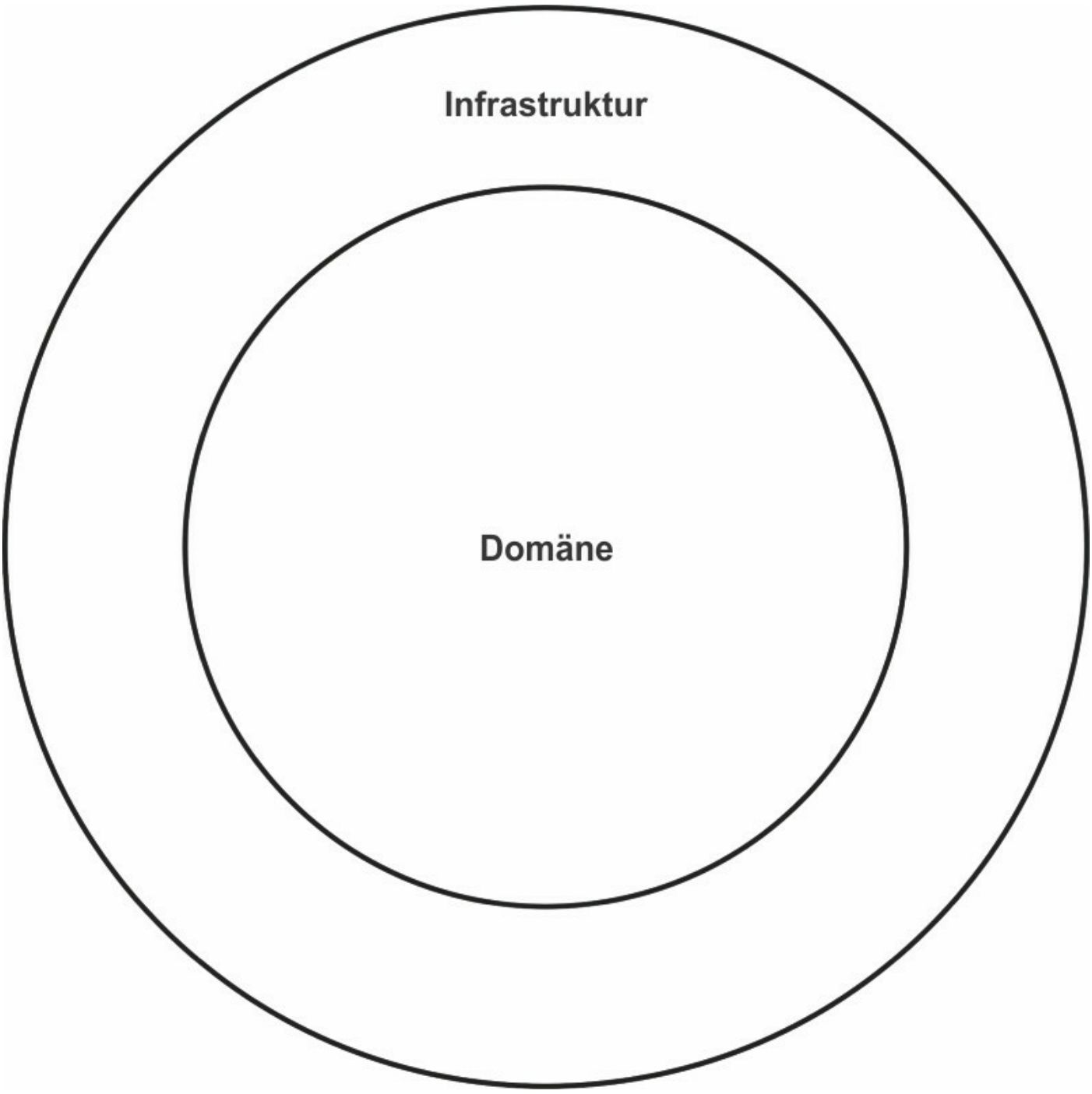


Abb. 34.9: Domänencode und Infrastrukturcode

34.8 Fazit: Der fehlende Ratschlag

Sinn und Zweck dieses Kapitels ist es, Ihnen aufzuzeigen, dass Ihre besten Designabsichten blitzschnell zunichtegemacht werden können, wenn Sie die Feinheiten der Implementierungsstrategie außer Acht lassen. Überlegen Sie, wie Sie Ihr gewünschtes Design auf Codestrukturen abbilden, wie Sie diesen Code organisieren und welche Entkopplungsmodi sich zur Laufzeit und zur Kompilierzeit anwenden lassen. Halten Sie sich wenn möglich Optionen offen, aber seien Sie auch pragmatisch und berücksichtigen Sie ebenso die Größe Ihres Teams, dessen Qualifikationsniveau sowie die Komplexität der Lösung in Verbindung mit dem Ihnen zur Verfügung stehenden Zeitrahmen und dem veranschlagten Budget. Denken Sie außerdem daran, Ihren Compiler einzusetzen, um den von Ihnen gewählten Architekturstil effizient umzusetzen, und achten Sie auf die Kopplung in anderen Bereichen wie beispielsweise Datenmodellen. Der Teufel steckt in den Implementierungsdetails.

---

[1] Diese Form der Benennung ist zwar grauenvoll; wie Sie später noch sehen werden, spielt das aber keine große Rolle.

[2] `https://martinfowler.com/bliki/PresentationDomainDataLayering.html`

[3] Dieser Vorteil verliert zwar mit den neuen Navigationsmöglichkeiten in modernen IDEs an Bedeutung, doch es scheint in letzter Zeit einen neuen Trend zur Rückkehr zu leichtgewichtigen Texteditoren zu geben. Die Beweggründe dafür kann ich allerdings nicht nachvollziehen, dafür bin ich eindeutig zu alt.

[4] Mein erster Job nach meinem Uniabschluss im Jahr 1996 war die Errichtung von Client-Server-Desktopanwendungen mithilfe einer Technologie namens PowerBuilder – einer superproduktiven 4GL, die in herausragender Weise für die Entwicklung von datenbankbasierten Anwendungen geeignet war. Ein paar Jahre später baute ich dann Java-basierte Client-Server-Anwendungen, wobei wir unsere eigene Datenbankkonnektivität (das war noch vor JDBC) und unsere eigenen, auf dem AWT aufsetzenden GUI-Toolkits einrichten mussten. Das ist »Fortschritt«!

[5] Das *Command Query Responsibility Segregation*-Pattern stellt Ihnen separate Patterns für die Aktualisierung und das Lesen der Daten zur Verfügung.

[6] `http://www.laputan.org/mud/`

[7] Für weitere Informationen siehe `https://www.structurizr.com/help/c4`.

[8] Obwohl wir dazu neigen, Packages beispielsweise in Java als hierarchisch zu

betrachten, ist es nicht möglich, Zugriffsbeschränkungen auf der Grundlage einer Package- und Subpackage-Beziehung einzurichten. Jede Hierarchie, die Sie erstellen, fußt immer ausschließlich auf diesen Packages und der Verzeichnisstruktur auf der Festplatte.

[9] Es sei denn, Sie schummeln und machen von Javas Reflexionsmechanismus Gebrauch ... aber bitte sehen Sie davon ab!

Anhang A

Architekturarchäologie

Um den Prinzipien der guten Softwarearchitektur detailliert auf den Grund zu gehen, möchte ich Sie im Folgenden auf eine kleine »archäologische« Erkundungsreise durch ein paar der Projekte mitnehmen, an denen ich seit 1970 beteiligt war. Einige davon sind aus architektonischer Sicht höchst aufschlussreich, während andere insofern interessant sind, als sie wertvolle Erkenntnisse zutage gefördert haben, die auf verschiedene Art in nachfolgende Projekte einflossen.

Naturgemäß weist dieser Anhang diverse autobiografische Züge auf. Dabei habe ich natürlich versucht, die Diskussion weitestgehend auf das Thema Softwarearchitektur zu fokussieren – aber wie in allen autobiografischen Abhandlungen ließ es sich auch in diesem Fall nicht gänzlich vermeiden, dass auch andere Faktoren zur Sprache kommen. ;-)

A.1 Das Buchhaltungssystem für die Gewerkschaft

Eine Firma namens ASC Tabulating schloss Ende der 1960er-Jahre mit der regionalen Fernfahrergewerkschaft einen Vertrag über die Bereitstellung eines Buchhaltungssystems ab. Für die Implementierung dieses Systems wählte ASC einen Computer mit der Bezeichnung GE Datanet 30 (siehe Abbildung A.1).

Abb. A.1: Der Computer GE Datanet 30
(Mit freundlicher Genehmigung von Ed Thelen, ed-thelen.org)

Wie Sie sehen, war diese Maschine wirklich riesig.[1] Sie benötigte einen eigenen, klimatisierten Raum.

Als dieser Computer gebaut wurde, gab es noch keine integrierten Schaltkreise. Stattdessen war er aus lauter einzelnen Transistoren aufgebaut. Er enthielt sogar einige Elektronenröhren (wenngleich auch nur in den Leseverstärkern der Magnetbandlaufwerke).

Nach heutigem Maßstab war die Maschine riesengroß, langsam, von bescheidener Leistung und ziemlich primitiv. Sie verfügte über 16 KB à 18 Bits Kernspeicher, und jeder Taktzyklus dauerte rund 7 Mikrosekunden.[2] Sie füllte einen großen, klimatisierten Raum aus und besaß ein siebenspuriges Magnetbandlaufwerk sowie eine Festplatte mit einer Kapazität in der Größenordnung von 20 Megabyte.

Diese Festplatte (Abbildung A.2) war ein wahres Monster, dessen tatsächliche Ausmaße die Abbildung kaum zu vermitteln vermag. Der Gehäuseschrank war mehr als mannshoch. Die Festplatten hatten einen Durchmesser von 91,44 Zentimetern und

waren 0,95 Zentimeter dick. Abbildung A.3 zeigt eine dieser Festplatten.

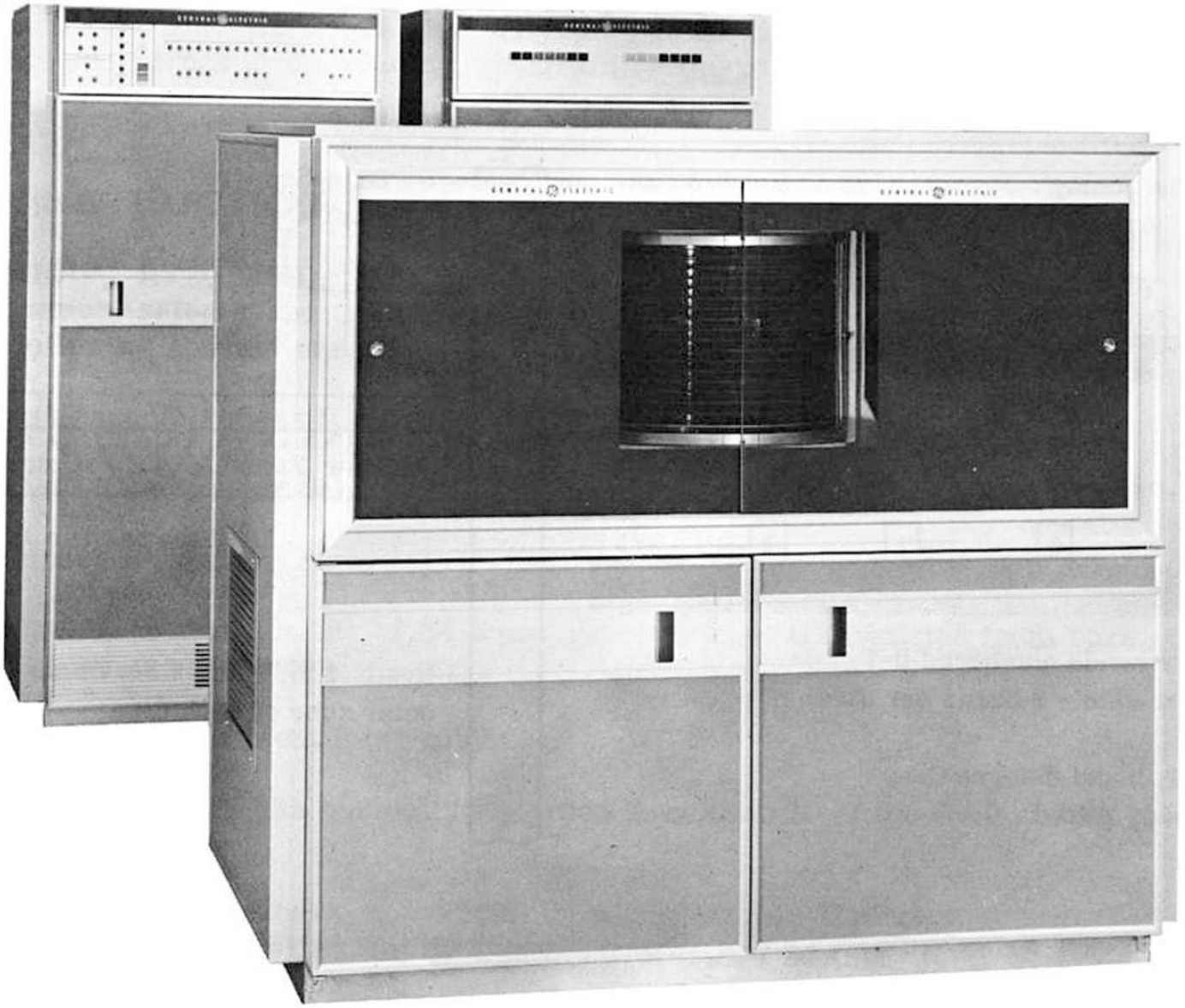


Abb. A.2: Der Massenspeicher mit den Festplatten
(Mit freundlicher Genehmigung von Ed Thelen, ed-thelen.org)

Wenn Sie einmal die Platten in Abbildung A.2 zählen, werden Sie auf mehr als ein Dutzend kommen. Jede einzelne von ihnen besaß einen eigenen, pneumatisch angetriebenen Schreib-/Lesekopf. Die Zugriffszeit lag bei etwa einer halben bis einer Sekunde.

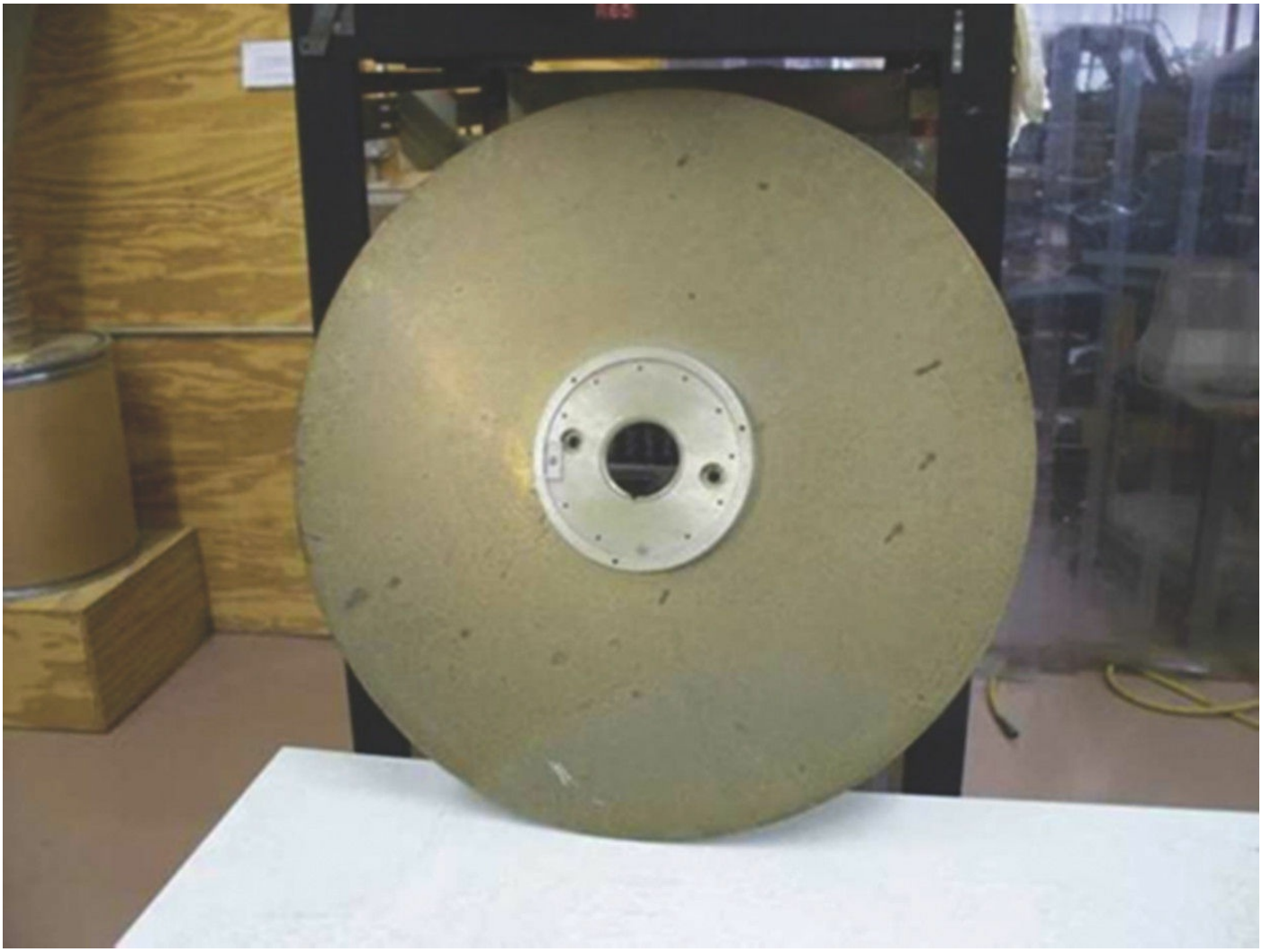


Abb. A.3: Eine der Festplatten: 0,95 Zentimeter dick und 91,44 Zentimeter im Durchmesser
(Mit freundlicher Genehmigung von Ed Thelen, ed-thelen.org)

Das Einschalten dieses Ungeheuers klang wie das Anwerfen eines Düsentriebwerks. Der Boden bebte und es rumpelte, bis das Gerät genügend Fahrt aufgenommen hatte. [3]

Die Datanet 30 wies die besondere Fähigkeit auf, dass sie eine Vielzahl asynchroner Terminals gleichzeitig mit vergleichsweise hoher Geschwindigkeit ansteuern konnte – und genau das war es, was ASC benötigte.

ASC war in Lake Bluff, Illinois, ansässig, das knapp 50 Kilometer nördlich von Chicago liegt. Die Büroräume der Gewerkschaft befanden sich dagegen in der Innenstadt von Chicago. Geplant war, dass rund ein Dutzend Datentypisten an Terminals mit Kathodenstrahlröhren[4] (Abbildung A.4) arbeiten sollten, um Daten für das System zu erfassen. Die Berichte sollten dann auf ASR35-Fernschreibern ausgegeben werden (Abbildung A.5).

Abb. A.4: Terminal mit Kathodenstrahlröhre von Datapoint
(Mit freundlicher Genehmigung von Bill Degnan, vintagecomputer.net)

Abb. A.5: Fernschreiber ASR35
(Mit freundlicher Genehmigung von Joe Mabel)

Die Terminals konnten 30 Zeichen pro Sekunde ausgeben. Ende der 1960er-Jahre war dies ein ziemlich guter Wert, denn die damaligen Modems waren noch nicht besonders weit entwickelt.

ASC mietete etwa ein Dutzend eigene Telefonleitungen und doppelt so viele 300-Baud-Modems bei der Telefongesellschaft, um die Datanet 30 mit den Terminals zu verbinden.

Diese Computer wurden ohne Betriebssystem ausgeliefert. Sie besaßen noch nicht einmal ein Dateisystem. Es stand lediglich ein Assembler zur Verfügung.

Wenn man Daten auf der Festplatte speichern wollte, wurden diese direkt auf die Platte geschrieben – nicht in eine Datei oder in ein Verzeichnis. Man musste ermitteln, auf welcher Platte, in welcher Spur und in welchem Sektor die Daten gespeichert werden sollten, und wies das Laufwerk dann an, sie dort abzulegen. Wir mussten also selbst einen Treiber für das Laufwerk schreiben.

In dem Buchhaltungssystem der Gewerkschaft gab es drei verschiedene Datensätze: Gewerkschaftsvertreter, Arbeitgeber und Mitglieder. Das System konnte diese Datensätze erstellen, lesen, aktualisieren und löschen, stellte aber auch Operationen für die Verwaltung von Beiträgen, Mitgliederkonten und Ähnliches zur Verfügung.

Das ursprüngliche System war von einem Berater in Assembler programmiert worden, der es irgendwie geschafft hatte, das Ganze in 16 KB unterzubringen.

Sie können sich sicherlich vorstellen, dass es ziemlich kostspielig war, eine große Maschine wie die Datanet 30 zu betreiben und zu warten. Und der Berater, der die Software auf dem Laufenden hielt, war auch nicht billig. Darüber hinaus kamen damals auch die ersten Minicomputer auf, die erheblich preiswerter waren.

Im Jahr 1971, ich war 18 Jahre alt, beauftragte ASC mich und zwei meiner technikbegeisterten Freunde, das gesamte Buchhaltungssystem der Gewerkschaft durch eins zu ersetzen, das auf einem Minicomputer des Typs Varian 620/f (Abbildung A.6) lief. Der Computer war preiswert – und wir waren ebenfalls günstig zu haben. Für ASC sah also alles nach einem guten Geschäft aus.

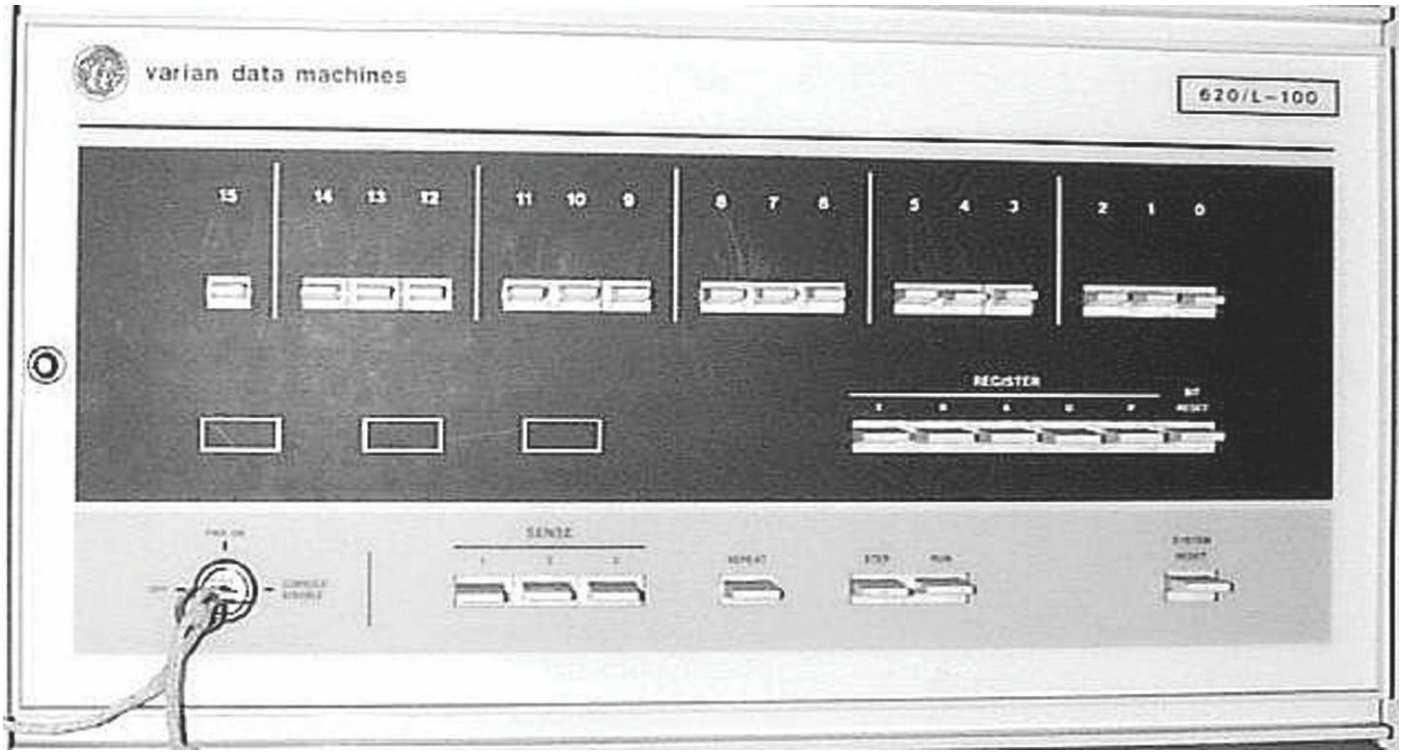


Abb. A.6: Der Minicomputer Varian 620/f (The Minicomputer Orphanage)

Die Maschine von Varian verfügte über einen 16-Bit-Bus, 32 KB×16 Kernspeicher, und der Taktzyklus dauerte rund eine Mikrosekunde. Dieser Computer war sehr viel leistungsfähiger als die Datanet 30. Er nutzte IBMs äußerst erfolgreiche 2314-Festplattentechnologie und ermöglichte es, 30 Megabyte auf Festplatten zu speichern, die einen Durchmesser von nur 35,56 Zentimetern besaßen und keine Betonwände durchdringen konnten!

Es gab natürlich noch immer weder ein Betriebssystem noch ein Dateisystem oder eine Hochsprache. Uns stand lediglich ein Assembler zur Verfügung, aber damit kamen wir zurecht.

Anstatt das gesamte System in den 32 KB einzupferchen, entwickelten wir ein Overlay-System. Die Anwendungen wurden von der Festplatte in einen für Overlays reservierten Speicherbereich geladen. Dort wurden sie auch ausgeführt und dann vorsorglich inklusive der lokalen RAM-Inhalte in einer Auslagerungsdatei auf der Festplatte gespeichert, um die Ausführung anderer Programme zu ermöglichen.

Die Programme wurden in den Overlay-Speicherbereich geladen, so lange ausgeführt, bis die Ausgabepuffer gefüllt waren, und anschließend wieder in eine Auslagerungsdatei geschrieben, sodass ein anderes Programm geladen und ausgeführt werden konnte.

Da die Benutzeroberfläche nur 30 Zeichen pro Sekunde darstellen konnte, befanden sich die Programme länger im Wartemodus. Wir hatten also genügend Zeit, die Programme einzulesen oder in die Auslagerungsdatei zu schreiben, um alle Terminals schnellstmöglich zu betreiben. Beschwerden über die Reaktionszeiten des Systems gab es nicht.

Wir programmierten einen präemptiven Supervisor, der Interrupts und die Ein-/Ausgabe handhabte. Und wir schrieben die Treiber für die Festplatte, die Terminals, die Magnetbandlaufwerke und für alles andere selbst. In diesem System gab es nichts, was wir nicht selbst geschrieben hatten. Es war zwar eine Plackerei mit viel zu vielen 80-Stunden-Wochen, aber wir konnten das Biest binnen acht bis neun Monaten zum Laufen bringen.

Die Architektur des Systems war einfach (Abbildung A.7). Nach dem Start einer Anwendung erzeugte sie so lange Ausgaben, bis der ihr zugewiesene Terminalpuffer voll war. Dann schrieb der Supervisor die Anwendung in die Auslagerungsdatei und las die nächste Applikation ein. Der Supervisor gab nun den Inhalt des Terminalpuffers mit 30 Zeichen pro Sekunde aus, bis er fast leer war. Anschließend wurde die Anwendung erneut geladen, um den Puffer wieder zu befüllen.

In diesem System fanden sich zwei Grenzen. Da war zum einen die Ausgabe der Zeichen. Der Anwendung war nicht bekannt, dass die Ausgabe an ein Terminal geliefert wurde, das 30 Zeichen pro Sekunde darstellte. Aus Sicht der Anwendung war die Ausgabe tatsächlich vollkommen abstrakt. Sie übergab die Zeichen einfach an den Supervisor, der sich darum kümmerte, die Puffer zu befüllen und die Anwendungen in den Arbeitsspeicher einzulesen bzw. sie auszulagern.

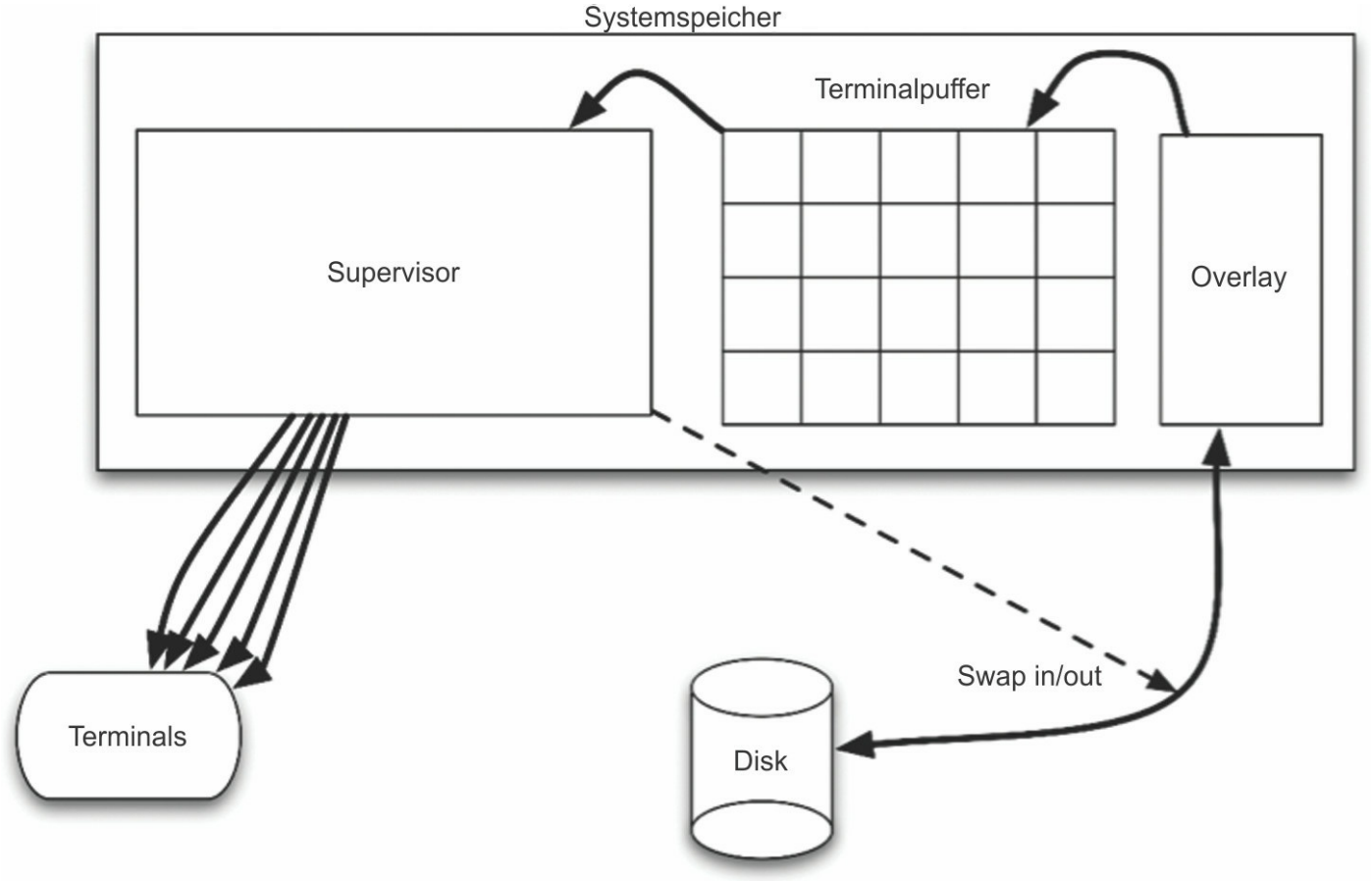


Abb. A.7: Die Systemarchitektur

Bei dieser Grenze lag ein normales Abhängigkeitsverhältnis vor – die Abhängigkeiten wiesen in *dieselbe* Richtung wie der Kontrollfluss. Die Anwendungen waren zur Kompilierzeit vom Supervisor abhängig, und die Steuerung wurde von den Anwendungen an den Supervisor übergeben. Diese Grenze verhinderte, dass die Anwendungen Kenntnis von der Art des Ausgabegeräts hatten.

Bei der zweiten Grenze lag eine Abhängigkeitsumkehrung vor. Der Supervisor konnte die Anwendungen starten, war zur Kompilierzeit jedoch nicht von ihnen abhängig. Die Steuerung wurde vom Supervisor an die Anwendungen übergeben. Die polymorphe Schnittstelle, die für die Umkehrung der Abhängigkeit sorgt, funktionierte folgendermaßen: Jede Anwendung wurde durch einen Sprung an genau dieselbe Speicheradresse innerhalb des Overlay-Speicherbereichs gestartet. Diese Grenze verhinderte, dass der Supervisor – außer der Startadresse – jegliche Kenntnis von den Anwendungen erlangte.

A.2 Zurechtschneiden mit dem Laser

1973 war ich bei einem Unternehmen namens Teradyne Applied Systems (TAS) in Chicago tätig. Hierbei handelte es sich um einen Bereich der Teradyne Inc., deren

Hauptniederlassung sich in Boston befand. Wir produzierten ein System, das vergleichsweise energiereiche Laser verwendete, um elektronische Bauteile sehr exakt zurechtzuschneiden.

Damals wandten die Hersteller Siebdruckverfahren an, um elektronische Bauteile auf keramischen Trägermaterialien zu produzieren. Diese Träger waren etwa 6 bis 7 Quadratzentimeter groß. Bei den Bauteilen handelte es sich typischerweise um Widerstände – Bauelemente, die den Fluss des elektrischen Stroms begrenzen.

Der Widerstand eines solchen Bauelements hängt von mehreren Faktoren ab, unter anderem vom Material und von der Geometrie. Je breiter das Bauteil ist, desto geringer ist der Widerstand.

Bei unserem System wurde das keramische Trägermaterial auf einem Arbeitstisch eingespannt und mit elektrisch leitenden Kontakten in Verbindung gebracht. Das System maß den Widerstand und verwendete den Laser, um Material von dem Bauteil abzutragen, das zunehmend dünner wurde, bis es den gewünschten Widerstandswert innerhalb einer Toleranz von circa einem Zehntelprozent erreicht hatte.

Wir verkauften diese Systeme an die Hersteller. Außerdem nutzten wir auch einige betriebsinterne Systeme, um Chargen relativ bescheidener Größe für kleinere Hersteller zu produzieren.

Bei dem verwendeten Computer handelte es sich um einen M365. Damals bauten viele Firmen ihre eigenen Computer: Teradyne baute den M365 und stellte ihn den verschiedenen Unternehmensbereichen zur Verfügung. Der M365 war eine verbesserte Version des PDP-8, einem seinerzeit weit verbreiteten Minicomputer.

Der M365 positionierte den Arbeitstisch, der das keramische Trägermaterial mit den Kontakten in Verbindung brachte. Außerdem steuerte er die Widerstandsmessung und den Laser. Der Laser wurde mithilfe von X-Y-Spiegeln positioniert, die programmgesteuert drehbar waren. Darüber hinaus konnte der Rechner auch die Leistung des Lasers regeln.

Die Entwicklungsumgebung des M365 war vergleichsweise primitiv. Ein Festplattenlaufwerk gab es nicht. Als Massenspeicher wurden Magnetbandkassetten verwendet, die den alten 8-spurigen Audiokassetten ähnelten. Die Bänder und die Laufwerke wurden von Tri-Data hergestellt.

Wie bei den 8-spurigen Audiokassetten war das Band als Schleife aufgewickelt. Das Laufwerk beförderte das Band nur in eine Richtung – ein Zurückspulen war nicht möglich! Wenn man etwas vom Anfang des Bandes lesen wollte, musste man das Gerät anweisen, zum »Startpunkt« zu spulen.

Das Band wurde mit einer Geschwindigkeit von ca. 30 Zentimetern pro Sekunde

transportiert – bei einem Band mit einer Länge von knapp 8 Metern konnte es also ganze 25 Sekunden dauern, bis zum Startpunkt zu spulen. Aus diesem Grund stellte Tri-Data Kassetten mit verschiedenen Bandlängen (ca. 3 bis 30 Meter) her.

Auf der Vorderseite des M365 befand sich eine Taste, die dazu diente, ein kleines Bootprogramm in den Arbeitsspeicher zu laden und auszuführen. Dieses Programm las den ersten Datenblock vom Band und führte ihn aus. Typischerweise enthielt dieser Block Code, der das auf dem übrigen Teil des Bandes befindliche Betriebssystem einlas.

Das Betriebssystem forderte den User auf, den Namen eines Programms einzugeben, das ausgeführt werden sollte. Diese Programme waren ebenfalls auf dem Band direkt nach dem Betriebssystem gespeichert. Wir gaben also den Namen eines Programms ein, zum Beispiel den des Editors ED-402, und das Betriebssystem suchte daraufhin auf dem Band danach, las es ein und führte es aus.

Als Konsole diente eine Kathodenstrahlröhre mit grüner Phosphorbeschichtung. Auf dem Bildschirm wurden 24 Zeilen mit jeweils 72 ASCII-Zeichen[5] (nur Großbuchstaben) angezeigt.

Zum Bearbeiten eines Programms musste man den Editor ED-402 laden und anschließend das Band einlegen, auf dem der Quellcode gespeichert war. Dann wurde ein Datenblock des Codes vom Band eingelesen und auf dem Bildschirm dargestellt. Ein Block konnte 50 Codezeilen speichern. Um den Code zu bearbeiten, musste man den Cursor auf dem Bildschirm bewegen und Zeichen eingeben – ganz ähnlich wie beim Editor `vi`. Nach der Bearbeitung musste der Code auf ein anderes Band geschrieben und der nächste Datenblock vom Quellcode-Band eingelesen werden. Das Ganze musste dann so lange wiederholt werden, bis alles erledigt war.

Es war nicht möglich, zum vorhergehenden Datenblock zurückzuscrollen. Die Bearbeitung eines Programms musste vollkommen linear erfolgen, von Anfang bis Ende. Wenn man zum Anfang des Programms zurückkehren wollte, musste man zunächst das Kopieren des Quellcodes auf das neue Band abschließen und anschließend mit diesem Band eine neue Sitzung zum Bearbeiten des Codes eröffnen. In Anbetracht dieser Einschränkungen ist es kaum verwunderlich, dass wir unsere Programme auf Papier ausdruckten, die vorzunehmenden Änderungen von Hand rot markierten und unsere Programme Block für Block anhand der Anmerkungen im Listing bearbeiteten.

Wenn die Bearbeitung abgeschlossen war, mussten wieder das Betriebssystem und der Assembler gestartet werden. Der Assembler las das Quellcode-Band, schrieb eine Binärdatei auf ein weiteres Band und gab außerdem ein Listing auf einem dafür vorgesehenen Zeilendrucker aus.

Man konnte sich nicht zu 100 % auf die Bänder verlassen, deshalb schrieben wir

immer zwei Bänder gleichzeitig. Auf diese Weise war die Wahrscheinlichkeit, dass zumindest eines der beiden Bänder fehlerlos lesbar war, ziemlich groß.

Unser Programm bestand aus etwa 20.000 Zeilen Code. Das Kompilieren dauerte fast 30 Minuten. Die Chance, dass es beim Lesen eines Bandes zu einer Fehlermeldung kam, stand ungefähr bei 1 zu 10. Wenn der Assembler auf einen Lesefehler traf, gab er auf der Konsole ein Tonsignal aus und druckte dann eine Reihe von Fehlermeldungen auf dem Drucker aus. Man konnte dieses unerträgliche Piepsen im gesamten Labor vernehmen. Außerdem konnte man den bedauernswerten Programmierer fluchen hören, dem soeben klar geworden war, dass die 30 Minuten dauernde Kompilierung erneut gestartet werden musste.

Die Architektur des Programms war für die damalige Zeit typisch. Es gab ein *Master Operating Program*, das treffenderweise als »MOP« bezeichnet wurde. Es hatte die Aufgabe, die grundlegenden Ein-/Ausgabefunktionen zu steuern und eine rudimentäre »Shell« auf der Konsole bereitzustellen. Der MOP-Quellcode wurde in vielen Unternehmensbereichen von Teradyne eingesetzt, sie alle hatten jedoch für eigene Zwecke Änderungen vorgenommen. Dementsprechend verteilten wir Aktualisierungen des Quellcodes in Form von mit Markierungen versehenen Listings untereinander, die wir dann von Hand (und äußerst vorsichtig) integrierten.

Die Hardware für die Widerstandsmessungen, den Arbeitstisch und den Laser wurde durch einen speziellen Hilfsprogramm-Layer gesteuert. Die Grenze zwischen diesem Layer und dem MOP war bestenfalls als unübersichtlich zu bezeichnen. Der Layer rief das MOP auf, das wiederum speziell für den Layer angepasst war und seinerseits den Layer aufrief. Tatsächlich betrachteten wir die beiden eigentlich gar nicht als verschiedene Layer. Für uns handelte es sich einfach nur um etwas hochgradig gekoppelten Code, den wir dem MOP hinzufügten.

Dann folgte der Isolations-Layer. Dieser Layer stellt in Form einer virtuellen Maschine eine Schnittstelle für Anwendungsprogramme zur Verfügung, die in einer völlig anderen domänenspezifischen datengetriebenen Sprache (DSL, *Domain-specific Data-driven Language*) geschrieben waren. Diese Sprache stellte verschiedene Operationen bereit, wie zum Beispiel das Positionieren des Lasers, das Verschieben des Arbeitstischs, das Ausführen von Schnitten, das Messen des Widerstands und so weiter. Unsere Kunden konnten ihre Anwendungsprogramme für das Zurechtschneiden mit dem Laser in dieser Sprache schreiben und der Isolations-Layer würde sie ausführen.

Dieser Ansatz war nicht als eine maschinenunabhängige Sprache für das Zurechtschneiden mit dem Laser gedacht. Tatsächlich wies die Sprache eine Vielzahl von Eigenheiten auf, die in hohem Maße an die darunter befindlichen Layer gekoppelt waren. Den Anwendungsprogrammierern sollte vielmehr eine »einfachere« Sprache als M365-Assembler zur Verfügung gestellt werden, in der sie die zu erledigenden Aufgaben programmieren konnten.

Die so programmierten Aufgaben konnten vom Band eingelesen und vom System ausgeführt werden. Im Grunde genommen war unser System ein Betriebssystem für Zurechtschneideanwendungen.

Das System war in M365-Assembler programmiert und erstellte in einem einzelnen Durchlauf Binärcode.

Die Grenzen waren bei dieser Anwendung bestenfalls als schwammig zu bezeichnen. Selbst die Grenze zwischen Systemcode und den in DSL programmierten Anwendungen wurde nicht streng umgesetzt – es gab überall Kopplungen.

Allerdings war das für Software zu Beginn der 1970er-Jahre durchaus typisch.

A.3 Monitoring von Aluminiumspritzguss

Als die OPEC Mitte der 1970er-Jahre ein Ölembargo verhängte und die Benzinknappheit dafür sorgte, dass verärgerte Autofahrer an den Tankstellen in Streit gerieten, begann meine Tätigkeit bei der Outboard Marine Corporation (OMC). Hierbei handelt es sich um die Muttergesellschaft von Johnson Motors und dem Rasenmäherhersteller Lawnboy.

In Waukegan, Illinois, betrieb OMC ein großes Werk, das im Spritzgussverfahren Aluminiumbauteile für die Motoren und die anderen Produkte des Unternehmens herstellte. Das Aluminium wurde in riesigen Schmelzöfen verflüssigt und in großen Wannen zu den mehreren Dutzend eigenständig betriebenen Aluminiumspritzgussmaschinen transportiert. Jede Maschine wurde von einem Arbeiter bedient, der für die Einrichtung der Gussformen, die Arbeitsgänge und die Entnahme der gegossenen Bauteile verantwortlich war. Die Bezahlung erfolgte entsprechend der Anzahl der produzierten Bauteile.

Ich war damit beauftragt, ein Projekt zur Automatisierung des Produktionsbereichs zu entwickeln. OMC hatte ein IBM System/7 erworben – IBMs Antwort auf den Minicomputer. Der Computer war mit den Spritzgussmaschinen verbunden, sodass wir die Arbeitsgänge der verschiedenen Maschinen zählen und deren Dauer messen konnten. Unsere Aufgabe war es, all diese Informationen zu sammeln und sie auf grünen IBM-3270-Displays anzuzeigen.

Als Programmiersprache kam erneut Assembler zum Einsatz. Und wieder einmal war der gesamte Code, der auf diesem Computer ausgeführt wurde, von uns selbst programmiert. Ein Betriebssystem gab es nicht, auch keine Bibliotheken oder Frameworks. Es handelte sich um reinen Code.

Der Code war Interrupt-gesteuert und wurde in Echtzeit ausgeführt. Nach jedem

Arbeitsgang einer Spritzgussmaschine mussten wir eine Reihe von Statistiken aktualisieren und die Daten an ein großes IBM-370-System übermitteln, auf dem ein CICS-COBOL-Programm (*Customer Information Control System*, ein Transaktionsmonitor von IBM) lief, das die Statistiken auf den grünen Bildschirmen anzeigte.

Ich hasste diesen Job. Meine Güte, wie habe ich ihn gehasst! Die eigentliche *Arbeit* hat durchaus *Spaß* gemacht, aber die Unternehmenskultur ... Es reicht wohl, zu erwähnen, dass ich *gezwungen* wurde, einen Schlips zu tragen.

Ach, ich habe mir wirklich Mühe gegeben – und wie! Aber es machte mich offensichtlich unglücklich, dort zu arbeiten, und meine Kollegen wussten das, denn ich konnte mir entscheidende Termine nicht merken oder schaffte es nicht, rechtzeitig aufzustehen, um an wichtigen Meetings teilzunehmen. Dies war meine einzige Anstellung als Programmierer, aus der man mich hinausgeworfen hat – und ich hatte es verdient.

Aus architektonischer Sicht kann man hieraus – mit einer Ausnahme – nicht viel lernen. Das System/7 verfügte über eine hochinteressante Anweisung namens *Set Program Interrupt* (*SPI*). Sie erlaubte es, einen Interrupt des Prozessors auszulösen, was es wiederum ermöglichte, die anderen in einer Warteschlange mit niedriger Priorität befindlichen Interrupts zu verarbeiten. Heutzutage wird dieses Verhalten in Java als `Thread.yield()` bezeichnet.

A.4 4-TEL

Nachdem ich im Oktober 1976 von OMC gefeuert worden war, kehrte ich zu einem anderen Geschäftsbereich von Teradyne zurück, bei dem ich die nächsten zwölf Jahre verbleiben sollte. Das Produkt, an dem ich arbeitete, hieß 4-TEL. Das Gerät hatte die Aufgabe, jede Nacht alle Telefonleitungen in einem bestimmten Versorgungsgebiet zu testen und einen Bericht zu erstellen, in dem sämtliche Leitungen aufgeführt wurden, an denen eine Reparatur erforderlich war. Darüber hinaus ermöglichte es den Mitarbeitern der Telefongesellschaft, ausgewählte Leitungen genauer zu untersuchen.

Dieses System besaß anfangs dieselbe Architektur wie das System zum Zuschneiden mit dem Laser. Es handelte sich um eine monolithische, in Assembler programmierte Anwendung ohne nennenswerte Grenzen. Zu dem Zeitpunkt, als ich dem Unternehmen beitrat, änderte sich das allerdings gerade.

Das System wurde von in einem *SC* (**S**ervice **C**enter) befindlichen Testern genutzt. Ein Servicecenter wiederum bestand aus mehreren *COs* (**C**entral **O**ffice, Telefonvermittlung), die jeweils nicht weniger als 10.000 Telefonleitungen bedienen

konnten. Die Hardware zum Wählen und Messen musste sich innerhalb des COs befinden, dort wurden also die M365-Computer aufgestellt. Wir nannten diese Rechner *COLTs* (**C**entral **O**ffice **L**ine **T**ester, Telefonvermittungsleitungstester). Im SC wurde ein weiterer M365-Computer eingerichtet, den wir als *SAC* (**S**ervice **A**rea **C**omputer, Versorgungsgebietcomputer) bezeichneten. Der SAC verfügte über mehrere Modems, die es ihm ermöglichten, die COLTs anzuwählen und in einer Geschwindigkeit von 300 Baud (30 Zeichen pro Sekunde) mit ihnen zu kommunizieren.

Zunächst erledigten die COLT-Computer sämtliche Aufgaben, inklusive der Steuerung der Konsole, der Anzeige der Menüs und des Erstellens von Berichten. Der SAC war lediglich ein einfacher Multiplexer, der die Ausgaben der COLTs entgegennahm und sie auf dem Bildschirm anzeigte.

Bei diesem Aufbau gab es jedoch das Problem, dass 30 Zeichen pro Sekunde wirklich sehr langsam waren. Den Testern gefiel es überhaupt nicht, die gemächlich eintröpfelnden Zeichen auf dem Bildschirm zu betrachten, zumal sie nur an einigen wenigen entscheidenden Daten interessiert waren. Darüber hinaus war der Kernspeicher des M365 damals kostspielig, und das Programm war ziemlich groß.

Die Lösung bestand darin, die Teile der Software zum Einwählen und Durchmessen der Leitungen von den Teilen zu trennen, die das Ergebnis analysierten und Berichte druckten. Letzteres wurde auf den SAC verlagert und Ersteres verblieb auf den COLTs. Auf diese Weise konnten kleinere Rechner mit sehr viel weniger Speicher als die COLTs eingesetzt werden, und die Reaktionszeiten der Terminals wurden erheblich beschleunigt, denn die Berichte wurden ja auf dem SAC erstellt.

Die Umstellung war außergewöhnlich erfolgreich. Die Aktualisierung der Bildschirmausgabe war sehr schnell (nachdem die Verbindung mit dem entsprechenden COLT zustande gekommen war), und der Speicherbedarf der COLTs verringerte sich erheblich.

Die Grenze war hier sehr scharf gezogen und sorgte für eine starke Entkopplung. Zwischen SAC und COLT wurden nur noch sehr kleine Datenpakete ausgetauscht. Dabei handelte es sich um eine sehr einfache Form der DSL, die aus schlichten Befehlen wie `DIAL XXXX` oder `MEASURE` bestand.

Der M365 las seine Daten von einem Magnetbandlaufwerk. Diese Laufwerke waren teuer und nicht besonders zuverlässig – schon gar nicht in dem industriellen Umfeld einer Telefonvermittlungszentrale. Darüber hinaus war der M365 im Vergleich zu der übrigen in den COLTs vorhandenen Elektronik kostspielig. Wir nahmen also ein Projekt in Angriff, das den M365 durch einen Rechner mit 8085-Mikroprozessor ersetzen sollte.

Der neue Computer bestand aus einer Platine mit einem 8085-Mikroprozessor, einer

RAM-Platine mit 32 KB Arbeitsspeicher und drei ROM-Platinen, die jeweils 12 KB Nur-Lese-Speicher enthielten. All diese Platinen waren in demselben Gehäuse verbaut, das auch die Hardware zum Messen enthielt, dadurch entfiel das voluminöse zusätzliche Gehäuse, das den M365 beherbergt hatte.

Auf den ROM-Platinen befanden sich zwölf Intel 2708-*EPROM-Chips* (**E**rasable **P**rogrammable **R**ead-**O**nly **M**emory, löschbarer programmierbarer Nur-Lese-Speicher).[6] Abbildung A.8 zeigt einen solchen Chip. Um die Chips mit Software zu beschreiben, mussten wir sie in ein spezielles Gerät einsetzen, das als EPROM-Brenner bezeichnet wird und von unserer Entwicklungsumgebung angesteuert wurde. Um die Chips wieder zu löschen, musste man sie intensivem Ultraviolettlicht aussetzen.[7]

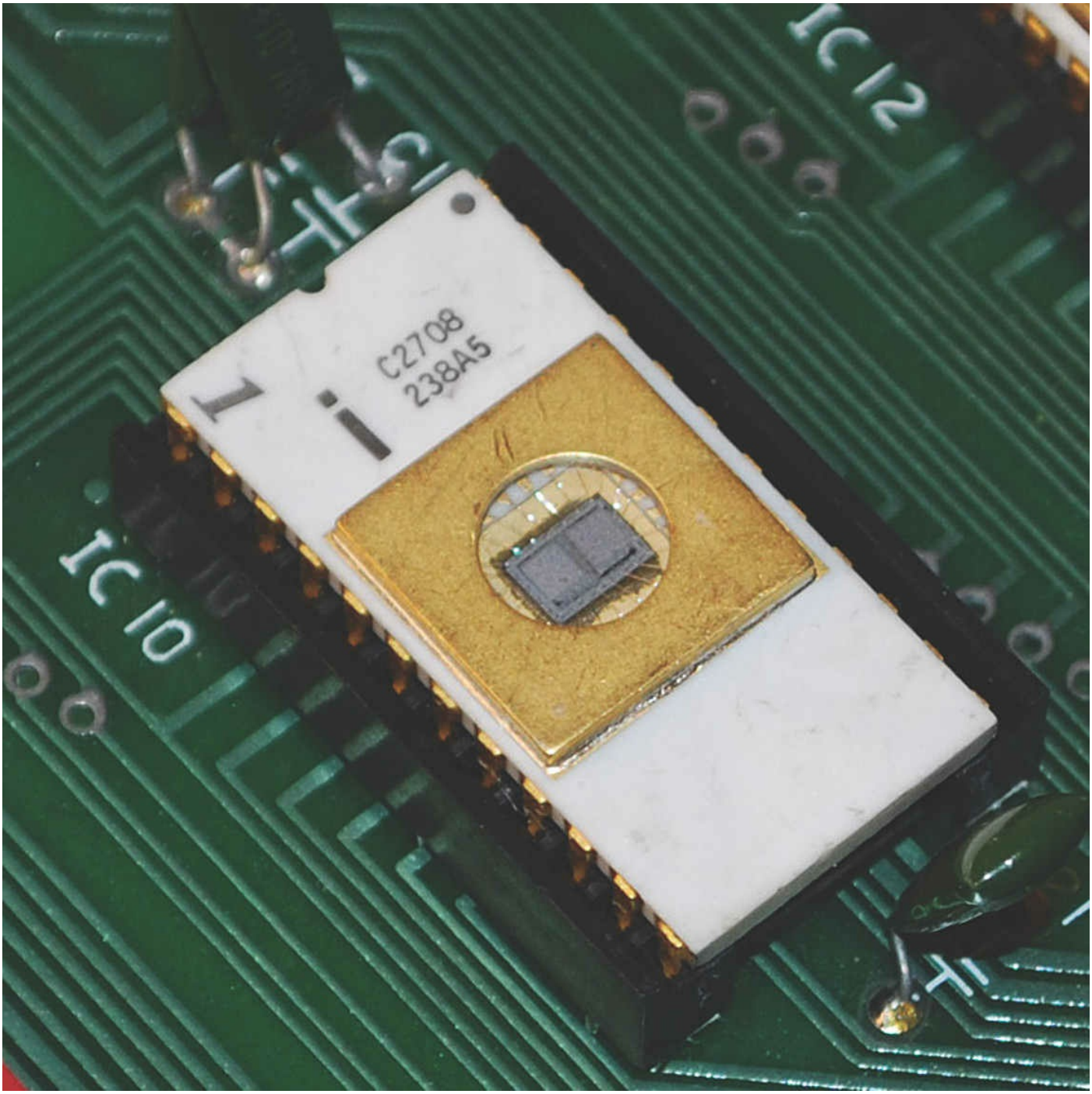


Abb. A.8: EPROM-Chip

Mein Kollege CK und ich übersetzten das Assemblerprogramm für den COLT in die 8085-Assemblersprache. Das geschah von Hand und dauerte rund sechs Monate. Das Ergebnis waren etwa 30 KB 8085-Code.

Unsere Entwicklungsumgebung verfügte über 64 KB RAM und besaß kein ROM, sodass wir die kompilierten Binärprogramme schnell ins RAM laden und testen konnten.

Nachdem wir das Programm zum Laufen bekommen hatten, verwendeten wir die EPROMs. Wir brannten 30 Chips und steckten sie in die passenden Steckplätze auf den drei ROM-Platinen. Die Chips waren gekennzeichnet, damit wir erkennen

konnten, welcher Chip in welchen Steckplatz gehörte.

Unser Programm war eine einzelne Binärdatei und 30 KB groß. Um die Chips brennen zu können, teilten wir die Binärdatei in 30 verschiedene 1 KB große Segmente auf und brannten die einzelnen Segmente auf die entsprechend gekennzeichneten Chips.

Das Ganze funktionierte sehr gut, also begannen wir mit der Massenproduktion der Hardware und brachten das System zum praktischen Einsatz.

Aber Software unterliegt nun einmal Veränderungen.[8] Wir mussten zusätzliche Features hinzufügen. Fehler mussten behoben werden. Als die Anzahl der Installationen wuchs, wurde die Logistik beim Update der Software durch das Brennen von 30 Chips pro Installation und die Notwendigkeit, jeweils vor Ort alle 30 Chips durch Wartungspersonal ersetzen zu lassen, zu einem Albtraum.

Es gab die unterschiedlichsten Probleme. Manchmal waren die Chips nicht richtig gekennzeichnet oder die Aufkleber mit der Kennzeichnung lösten sich von den Chips. Manchmal ersetzten die Wartungstechniker vor Ort den falschen Chip. Gelegentlich brach einer der Pins der neuen Chips ab, daher musste der Techniker alle Chips doppelt dabeihaben.

Warum aber mussten alle 30 Chips gleichzeitig ausgetauscht werden? Wenn wir unserer 30 KB großen ausführbaren Datei Code hinzufügten (oder Code daraus entfernten), änderten sich die Adressen, an denen die verschiedenen Befehle in den Speicher geladen wurden. Dabei änderten sich auch die Adressen von Subroutinen und Funktionen, die wir aufriefen. Es spielte daher keine Rolle, wie trivial eine Änderung war, es waren stets alle Chips davon betroffen.

Eines Tages kam mein Chef zu mir und bat mich, das Problem zu lösen. Er meinte, wir müssten eine Möglichkeit finden, Änderungen an der Firmware vorzunehmen, ohne jedes Mal alle 30 Chips auszutauschen. Wir grübelten eine Weile darüber nach und nahmen dann das Projekt »Vektorisierung« in Angriff. Ich war drei Monate damit beschäftigt.

Die zugrunde liegende Idee war wunderbar einfach: Wir teilten das 30 KB große Programm in 32 voneinander unabhängig kompilierbare Quellcodedateien auf, die jeweils kleiner als 1 KB waren. Am Anfang der Datei wurde dem Compiler mitgeteilt, an welcher Adresse er das resultierende Programm laden sollte (zum Beispiel `ORG C400` bei dem Chip, der in den Steckplatz C4 gehörte).

Am Anfang der Datei wurde außerdem eine einfache Datenstruktur fester Größe erzeugt, in der die Adressen aller Subroutinen des zugehörigen Chips gespeichert waren. Diese Datenstruktur war 40 Byte groß und konnte daher maximal 20 Adressen speichern, was bedeutete, dass kein Chip mehr als 20 Subroutinen enthalten konnte.

Anschließend erzeugten wir im RAM einen speziellen Speicherbereich, für den die Bezeichnung »Sprungvektoren« geläufig ist. Er enthielt 32 jeweils 40 Byte große Tabellen – gerade groß genug, um die Zeiger am Anfang der Chips zu speichern.

Und schließlich änderten wir noch alle Aufrufe von Subroutinen der Chips in indirekte Aufrufe durch den entsprechenden Sprungvektor im RAM.

Beim Booten durchsuchte der Prozessor die Chips und lud die Tabelle mit den Sprungvektoren am Anfang der Chips ins RAM. Anschließend sprang er zum Hauptprogramm.

All das funktionierte ausgezeichnet. Wenn wir nun einen Fehler behoben oder ein Feature hinzufügten, brauchten wir nur noch ein oder zwei Chips neu zu kompilieren und sie dem Wartungstechniker zu übergeben.

Wir hatten erreicht, dass die Chips *unabhängig voneinander deploybar* waren. Wir hatten die »polymorphe Versendung« der Chips ermöglicht. Wir hatten Objekte erdacht.

Hierbei handelte es sich im wahrsten Sinne des Wortes um eine Plug-in-Architektur. Wir entwickelten sie weiter, sodass ein neues Feature unseres Produkts installiert werden konnte, indem man den entsprechenden Chip in einen der freien Steckplätze einsetzte. Die Menüsteuerung erschien dann automatisch und die Einbindung in das Hauptprogramm erfolgte ebenfalls automatisch.

Natürlich waren uns die Konzepte der objektorientierten Programmierung damals noch unbekannt und wir wussten auch nichts über die Trennung von Benutzerschnittstelle und Geschäftsregeln. Aber die Grundlagen waren geschaffen – und sie erwiesen sich als äußerst leistungsfähig.

Dieser Ansatz brachte außerdem einen weiteren unerwarteten Vorteil mit sich: Wir konnten die Firmware über eine Einwählverbindung patchen. Wenn wir einen Fehler entdeckten, konnten wir uns über eine Einwahlverbindung mit den Geräten verbinden und das dort laufende Monitorprogramm verwenden, um den Sprungvektor der fehlerhaften Subroutine zu ändern, sodass er auf einen leeren Speicherbereich im RAM verwies. In diesem Speicherbereich legten wir die korrigierte Version der Subroutine ab, indem wir den hexadezimalen Maschinencode eintippten.

Für den Wartungsbetrieb und für unsere Kunden war das ein Segen. Wenn es Schwierigkeiten gab, mussten wir nicht mehr sofort einen neuen Chip liefern und eiligst einen Wartungstechniker entsenden. Das System konnte gepatcht werden und der neue Chip würde bei der nächsten geplanten Wartung installiert werden.

A.4.1 Service Area Computer

Der *SAC* (**S**ervice **A**rea **C**omputer, Versorgungsgebietcomputer) von 4-TEL basierte auf einem M365-Minicomputer. Dieses System kommunizierte mit sämtlichen angebundenen COLTs entweder über eigens dafür vorgesehene Modems oder über Einwählverbindungen. Es wies die COLTs an, die Telefonleitungen durchzumessen, empfing die Ergebnisse und führte anschließend eine komplizierte Analyse dieser Ergebnisse durch, um Fehler zu identifizieren und zu lokalisieren.

A.4.2 Ermittlung des Wartungsbedarfs

Zu den wirtschaftlichen Säulen dieses Systems gehörte es, die korrekte Entsendung des Wartungspersonals zu gewährleisten. Das Wartungspersonal war gemäß den Vorschriften der Gewerkschaft in drei Gruppen von Handwerkern unterteilt, die für die zentrale Telefonvermittlung, die Verkabelung und den Kundendienst vor Ort zuständig waren: Die in der Telefonvermittlung tätigen Handwerker kümmerten sich um Probleme der zentralen Telefonvermittlung. Die für die Verkabelung zuständigen Handwerker behoben Probleme mit den Leitungen, die Telefonvermittlung und Kunden verbanden. Der Kundendienst war für Aufgaben zuständig, die in den Räumlichkeiten des Kunden oder bei den Leitungen anfielen, die zu diesen Räumlichkeiten führten.

Wenn ein Kunde über Probleme klagte, konnte unser System dieses Problem analysieren und entscheiden, welche der drei Gruppen dafür zuständig war. Dadurch sparten die Telefongesellschaften eine Menge Geld, denn das Entsenden ungeeigneter Handwerker bedeutete eine erhöhte Wartezeit für den Kunden und überflüssige Anfahrten für die Handwerker.

Der Code für die Ermittlung des Wartungsbedarfs wurde von jemandem entworfen und geschrieben, der zwar sehr gewitzt war, aber miserable Kommunikationsfähigkeiten besaß. Der Prozess zur Erstellung des Codes wurde wie folgt beschrieben: »Ein drei Wochen langes Starren an die Decke und zwei Tage, an denen der Code aus allen Poren seines Körpers hervorsprudelte – anschließend kündigte er.«

Diesen Code konnte allerdings niemand verstehen. Jedes Mal, wenn wir versuchten, ein Feature hinzuzufügen oder einen Fehler zu beheben, funktionierte irgendetwas nicht mehr. Und da einer der wichtigsten wirtschaftlichen Vorteile unseres Systems auf diesem Code beruhte, war jeder neu auftauchende Fehler für das Unternehmen hochnotpeinlich.

Letztendlich wies das Management uns an, den Code nicht mehr anzutasten und keine Änderungen vorzunehmen. Der Code wurde *offiziell eingefroren*.

Diese Erfahrung hat mich gelehrt, klaren und sauberen Code wertzuschätzen.

A.4.3 Architektur

Dieses System wurde 1976 in M365-Assembler geschrieben. Es handelte sich um ein einziges monolithisches Programm, das aus rund 60.000 Zeilen Code bestand. Das selbst gebastelte, nicht-präemptive Betriebssystem wechselte je nach Abfrage den Kontext. Wir bezeichneten es als *MPS*, was für **M**ultiprocessing **S**ystem stand. Der M365-Computer verfügte nicht über einen integrierten Stack, deshalb wurden aufgabenspezifische Variablen in einem speziellen Speicherbereich abgelegt und bei jedem Kontextwechsel in eine Auslagerungsdatei geschrieben. Gemeinsam genutzte Variablen wurden durch Locks (Sperren) und Semaphoren gehandhabt. Die Eintrittsinvarianz und Race Conditions führten ständig zu Problemen.

Die Steuerungslogik der Geräte und die Benutzerschnittstelle waren in keinerlei Weise von den Geschäftsregeln isoliert. Die Befehle zum Ansteuern der Modems waren beispielsweise überall im Code der Benutzerschnittstelle und der Geschäftsregeln verteilt. Es wurde auch kein Versuch unternommen, diese Befehle in einem Modul zu vereinen oder eine abstrakte Schnittstelle dafür einzurichten. Die Modems wurden auf Bit-Ebene durch Code angesteuert, der im gesamten System verstreut war.

Dasselbe traf auch für die Benutzeroberfläche des Terminals zu. Der Code zur Ausgabe von Benachrichtigungen und zum Formatieren der Ausgabe war nicht isoliert. Er fand sich in den 60.000 Zeilen Code an fast jeder beliebigen Stelle.

Die von uns verwendeten Modemmodule waren dafür ausgelegt, auf der Platine des PCs montiert zu werden. Wir erwarben diese Geräte von einem Dritthersteller und kombinierten sie mit einer weiteren Schaltung auf einer Platine, die auf unsere selbst gebaute Busplatine passte. Diese Geräte waren richtig teuer, deshalb entschlossen wir uns nach einigen Jahren, die Modems selbst zu entwickeln. Wir in der Softwareabteilung flehten die Hardwaredesigner an, dasselbe Bitformat für die Steuerung des neuen Modems zu verwenden. Wir erklärten ihnen, dass der Code zum Ansteuern von Modems überall verteilt war und dass wir in Zukunft beide Modemarten handhaben mussten. Wir baten also inständig: »Bitte sorgt dafür, dass sich die neuen Modems genauso wie die alten ansteuern lassen.«

Aber als wir das neue Modem erhielten, war die Steuerung komplett anders – nicht nur etwas unterschiedlich, sie war voll und ganz anders.

Danke, Hardwareentwickler!

Was also sollten wir tun? Wir ersetzten ja nicht einfach nur alle alten Modems durch neue, wir hatten es vielmehr mit einer Mischung aus alten und neuen Modems in unserem System zu tun. Die Software musste mit beiden Modemtypen gleichzeitig zurechtkommen. Waren wir wirklich dazu verdammt, sämtliche Stellen im Code, an denen Modems angesteuert wurden, mit Statusbits und Tests auf Sonderfälle zu versehen? Davon gab es Hunderte!

Letztendlich entschieden wir uns für eine sogar noch schlechtere Lösung.

Eine bestimmte Subroutine schrieb Daten auf den seriellen Bus, der dazu diente, sämtliche Geräte zu steuern, auch die Modems. Wir änderten diese Subroutine so ab, dass sie die den alten Modems eigenen Bitmuster erkannte und sie in die für die neuen Modems erforderlichen Bitmuster konvertierte.

Das war allerdings gar nicht so einfach. Die Modembefehle bestanden aus einer Abfolge von Schreibzugriffen auf verschiedene IO-Adressen des seriellen Busses. Unser Hack musste diese Befehle der Reihe nach interpretieren und unter Verwendung unterschiedlicher IO-Adressen und Bitpositionen übersetzen und zeitlich korrekt abgestimmt weiterleiten.

Wir haben es zum Laufen gebracht, aber es war der übelste Hack, den man sich nur vorstellen kann. Aufgrund dieses Fiaskos habe ich es gelernt, die Trennung von Hardware und Geschäftsregeln sowie abstrakte Schnittstellen wertzuschätzen.

A.4.4 Die große Neugestaltung

Als die 1980er-Jahre näher rückten, kam die Vorstellung, sich einen eigenen Minicomputer mit eigener Computerarchitektur zu entwickeln, allmählich aus der Mode. Auf dem Markt waren viele Minicomputer verfügbar, und sie zum Laufen zu bringen, war preiswerter und üblicher, als weiterhin auf die Ende der 1960er-Jahre entwickelten proprietären Computerarchitekturen zu setzen. Diese Tatsache und die grauenvolle Architektur der SAC-Software veranlasste unser technisches Management, die Architektur des SAC-Systems komplett neu zu gestalten.

Das neue System sollte auf einem von Festplatte laufenden UNIX-Betriebssystem auf einem 8086-Mikrocomputer von Intel in der Programmiersprache C entwickelt werden. Die Hardware-Jungs machten sich an die Arbeit, den neuen Computer zu entwickeln, und eine auserlesene Gruppe von Softwareentwicklern, das »Tiger-Team«, wurde mit der Neuprogrammierung beauftragt.

Ich möchte Sie nicht mit den Einzelheiten des anfänglichen Desasters langweilen. Es reicht wohl zu erwähnen, dass das erste Tiger-Team auf ganzer Linie scheiterte, nachdem zwei bis drei Arbeitsjahre für ein Softwareprojekt in den Sand gesetzt wurden, das nie zustande kam.

Ein bis zwei Jahre später, so um 1982, wurde ein weiterer Versuch in Angriff genommen. Das Ziel war eine vollständige Neugestaltung des SAC in C unter UNIX, das auf unserer eigenen, neu entwickelten und beeindruckend leistungsfähigen 80286-Hardware laufen sollte. Wir tauften diesen Computer »Deep Thought«.

Es dauerte Jahre. Jahre über Jahre. Ich weiß nicht, wann der erste unter UNIX laufende

SAC schließlich zum Einsatz kam. Ich glaube, zu diesem Zeitpunkt hatte ich das Unternehmen bereits verlassen (1988). Tatsächlich bin ich mir noch nicht einmal sicher, ob er überhaupt jemals zum Einsatz kam.

Doch was hatte diese Verzögerung verursacht? Kurz und bündig: Für das Team, das die Neugestaltung durchführen sollte, war es äußerst schwierig, mit einer großen Gruppe von Programmierern mitzuhalten, die das alte System aktiv warteten. Hier nur ein einziges Beispiel dafür, mit welchen Schwierigkeiten das Team zu kämpfen hatte.

A.4.5 Europa

Ungefähr zu dem Zeitpunkt, als der SAC in C neu gestaltet wurde, begann das Unternehmen, seinen Vertrieb in Europa aufzubauen. Natürlich konnte man nicht abwarten, bis die Neugestaltung der Software abgeschlossen war, deshalb kamen in Europa die alten M365-Systeme zum Einsatz.

Das Problem bestand darin, dass sich die Telefonsysteme in Europa deutlich von den Telefonsystemen in den USA unterscheiden. Auch die Organisation der Telefongesellschaften und die Bürokratie sind verschieden. Einer unserer besten Programmierer wurde nach Großbritannien entsandt, um ein Team britischer Entwickler zu leiten, das die SAC-Software an die europäischen Verhältnisse anpassen sollte.

Ein ernsthafter Versuch, diese Änderungen in die US-Software zu integrieren, wurde natürlich nicht unternommen – zu diesem Zeitpunkt gab es noch keine Netzwerke, die es ermöglicht hätten, eine größere Codebasis auf die andere Seite des Atlantiks zu übertragen. Die britischen Entwickler verwendeten einfach den US-Code, erstellten einen Fork (also eine Abspaltung) und nahmen nach Bedarf Anpassungen vor.

Das führte natürlich zu Problemen. Auf beiden Seiten des Atlantiks wurden Fehler entdeckt, die auf der Gegenseite hätten behoben werden müssen. Die verschiedenen Module hatten sich jedoch beträchtlich verändert, daher war es sehr schwierig festzustellen, ob die in den USA durchgeführte Fehlerbehebung in Großbritannien überhaupt funktionieren würde.

Nach einigen Jahren, die viele Kopfschmerzen bereiteten, und der Einrichtung einer schnellen Datenleitung zur Verbindung der Büros in den USA und Großbritannien wurde schließlich ein ernsthafter Versuch unternommen, die beiden Forks wieder so zu vereinen, dass die Unterschiede nur eine Frage der Konfiguration waren. Diese Bemühungen scheiterten beim ersten, zweiten und dritten Versuch. Die beiden Codebasen waren zwar äußerst ähnlich, aber für eine Vereinigung dennoch zu verschieden – insbesondere in Anbetracht des sich schnell ändernden Markts, der seinerzeit vorherrschte.

Das »Tiger-Team«, das versuchte, das Ganze in C unter UNIX neu zu schreiben, hatte inzwischen bemerkt, dass es sich ebenfalls um diese Zweiteilung Europa/USA kümmern musste. Und das beschleunigte den Fortschritt ihrer Arbeit natürlich überhaupt nicht.

A.4.6 SAC: Fazit

Ich könnte Ihnen noch viele weitere Geschichten über dieses System erzählen, aber es ist für mich einfach zu deprimierend, damit fortzufahren. Ich stelle nur fest, dass ich viele der bitteren Lektionen meines Softwarelebens lernen musste, während ich in den grauenhaften Assemblercode des SAC versunken war.

A.5 Die Programmiersprache C

Die beim 4-TEL-Projekt verwendete 8085-Computerhardware stellte für uns eine relativ preiswerte Plattform für viele verschiedene Projekte dar, die sich im industriellen Umfeld integrieren ließen. Uns standen 32 KB RAM und weitere 32 KB ROM zur Verfügung, und es gab außerordentlich flexible und leistungsfähige Möglichkeiten, Peripheriegeräte zu steuern. Was fehlte, war eine flexible und komfortable Programmiersprache zur Programmierung der Maschine – es machte schlicht und einfach keinen Spaß, Code mit dem 8085-Assembler zu schreiben.

Obendrein hatten unsere eigenen Programmierer den von uns verwendeten Assembler geschrieben. Er lief auf unseren M365-Computern und nutzte das im Abschnitt *Zurechtschneiden mit dem Laser* beschriebene, auf Magnetband gespeicherte Betriebssystem.

Das Schicksal wollte es, dass unser leitender *Hardwareentwickler* unseren CEO davon überzeugte, dass wir einen *richtigen* Computer benötigten. Er wusste zwar nicht, was er damit hätte anfangen können, war jedoch ziemlich einflussreich. Also wurde eine PDP-11/60 beschafft.

Ich, damals nur ein einfacher Softwareentwickler, war begeistert. Ich wusste *haargenau*, was ich mit diesem Computer machen wollte – und ich war entschlossen, dass dies *mein* Computer sein sollte.

Als die Handbücher eintrafen – mehrere Monate vor der Lieferung des Computers –, nahm ich sie mit nach Hause und verschlang sie regelrecht. Als der Computer geliefert wurde, wusste ich schon, wie die Hardware bedient wird, und auch die Software war mir bereits gut vertraut – zumindest so gut vertraut, wie es durch das Lesen von Handbüchern sein kann.

Ich war außerdem beim Schreiben der Bestellung behilflich. So konnte ich die Größe des Festplattenspeichers des neuen Computers festlegen. Ich entschied, dass wir zwei Wechselplattenlaufwerke kaufen sollten, deren Wechselmedien jeweils 25 Megabyte speichern konnten.[9]

Fünfzig Megabyte! Diese Zahl erschien unermesslich! Ich kann mich noch erinnern, dass ich spätabends durch die Büroflure lief und wie die Böse Hexe des Westens im Buch »Der Zauberer von Oz« vor mich hingackerte: »Fünfzig Megabyte! Hahahahahahahahah!«

Ich bat den Hausmeister, einen kleinen Raum einzurichten, der sechs VT100-Terminals beherbergen sollte. Diesen Raum schmückte ich mit einigen Weltraumbildern. Hier würden unsere Softwareentwickler ihren Code schreiben und kompilieren.

Als die Maschine geliefert wurde, verbrachte ich mehrere Tage damit, sie aufzubauen, die Terminals zu verkabeln und alles zum Laufen zu bringen. Es war das reine Vergnügen – eine Art Liebesdienst.

Dann beschafften wir Standardassembler für den 8085 von Boston Systems Office und übersetzten den 4-TEL-Mikrocode in diese Syntax. Wir bauten einen EPROM-Brenner und entwickelten ein plattformübergreifendes System zum Kompilieren, das es ermöglichte, kompilierte Binärdateien von der PDP-11 auf unsere 8085-Entwicklungsumgebung herunterzuladen. Und siehe da: Alles lief wie am Schnürchen!

A.5.1 C

Wir hatten allerdings noch immer das Problem, 8085-Assembler verwenden zu müssen. Damit war ich alles andere als glücklich. Ich hatte davon gehört, dass es diese »neue« Programmiersprache gab, von der in den Bell-Laboratorien ausgiebig Gebrauch gemacht wurde. Sie nannten sie »C«. Also kaufte ich mir das Buch *Programmieren in C* von Kernighan und Ritchie. Dieses Buch verschlang ich ebenso begierig wie einige Monate zuvor die PDP-11-Handbücher.

Die schlichte Eleganz dieser Programmiersprache verblüffte mich. Sie ermöglichte den Zugang zu der Leistungsfähigkeit der Assemblersprache durch eine sehr viel komfortablere Syntax, ohne irgendwelche Vorteile des Assemblers zu opfern. Ich war völlig fasziniert.

Also kaufte ich mir einen C-Compiler von Whitesmiths, den ich auf der PDP-11 zum Laufen bringen konnte. Die Ausgabe des Compilers bestand aus einer Assemblersyntax, die mit dem 8085-Compiler von Boston Systems Office kompatibel war. Damit war der Weg von C zur 8085-Hardware frei! Jetzt konnten wir loslegen.

Nun gab es nur noch das Problem, einen Haufen eingefleischter Assemblerprogrammierer davon zu überzeugen, stattdessen C zu verwenden. Aber diese albtraumhafte Geschichte erzähle ich ein anderes Mal.

A.6 BOSS

Für unsere 8085-Plattform gab es kein Betriebssystem. Durch meine Erfahrungen mit dem MPS des M365 und den primitiven Interrupt-Mechanismen des IBM System 7 gelangte ich zu der Überzeugung, dass wir für den 8085 einen einfachen Kontextwechsel ermöglichen sollten. Deshalb konzipierte ich *BOSS*: **B**asic **O**perating **S**ystem and **S**cheduler (Grundlegendes Betriebssystem und Steuerprogramm).[10]

Der weitaus größte Teil von BOSS war in C geschrieben. Es gab die Möglichkeit, mehrere konkurrierende Tasks auszuführen. Derartige Tasks waren nicht präemptiv – der Wechsel zu einer anderen Aufgabe wurde nicht durch den Interrupt gesteuert, sondern musste wie beim MPS des M365 durch einen einfachen Abfragemechanismus veranlasst werden. Diese Abfrage fand immer dann statt, wenn die Ausführung eines Tasks aufgrund eines Ereignisses blockiert wurde.

Der BOSS-Aufruf zum Blockieren der Ausführung eines Tasks sah folgendermaßen aus:

```
block(eventCheckFunction);
```

Dieser Aufruf hielt die Ausführung des aktuellen Tasks an, platzierte `eventCheckFunction` in die Liste der Aufrufe und verknüpfte die Funktion mit dem soeben blockierten Task. Anschließend verblieb die Funktion in der Aufrufwarteschleife und führte die Funktionen in der Liste der Aufrufe aus, bis eine davon `true` zurücklieferte. In diesem Fall wurde der mit der Funktion verknüpfte Task ausgeführt.

Auf diese Weise wurde es, wie bereits erwähnt, ermöglicht, einen einfachen, nicht-präemptiven Kontextwechsel vorzunehmen.

In den kommenden Jahren bildete diese Software die Grundlage einer Vielzahl von Projekten. Eins der ersten davon war CCU.

A.7 Projekt CCU

Die späten 1970er- und frühen 1980er-Jahre waren für die Telefongesellschaften eine

turbulente Zeit. Die digitale Revolution war einer der Gründe dafür.

Die letzten hundert Jahre hatten die Leitungen zwischen der zentralen Telefonvermittlung und dem Telefon des Kunden aus einer Doppelader Kupferdraht bestanden. Diese Leitungen wurden zu Kabelsträngen zusammengebündelt und als riesiges Netzwerk kreuz und quer durchs Land verlegt. Sie verliefen zum Teil oberirdisch über Telefonmasten oder wurden unterirdisch verlegt.

Kupfer ist ein wertvolles Metall, das die Telefongesellschaften zur Abdeckung der Versorgungsgebiete buchstäblich tonnenweise verlegen mussten. Das darin investierte Kapital war immens. Ein Großteil dieser Investitionen konnte weiter genutzt werden, indem die Telefongespräche über digitale Verbindungen übertragen wurden. Ein einziges Adernpaar konnte Hunderte Gespräche in digitaler Form transportieren.

Daraufhin nahmen es die Telefongesellschaften in Angriff, die alten analogen Vermittlungsstellen durch moderne, digitale Geräte zu ersetzen.

Unser 4-TEL-Gerät testete jedoch Kupferleitungen, keine digitalen Verbindungen. Zwar gab es auch in einem digitalen Netz noch viele Kupferleitungen, diese waren allerdings viel kürzer als früher und befanden sich in unmittelbarer Umgebung der Telefone der Kunden. Das Signal wurde von der zentralen Vermittlungsstelle digital zu einem Verteilerkasten vor Ort übertragen. Dort wurde es wieder in ein analoges Signal umgewandelt und über herkömmliche Kupferleitungen an den Kunden weitergeleitet. Unser Messgerät musste sich also am Anfang der analogen Kupferleitung befinden, das Gerät zum Wählen hingegen musste in der zentralen Vermittlungsstelle verbleiben. Das Problem bestand nun darin, dass alle unsere COLTs sowohl das Gerät zum Wählen als auch das Messgerät in einem einzigen Gehäuse enthielten. (Wir hätten ein Vermögen sparen können, wenn wir diese offensichtliche architektonische Grenze einige Jahre früher erkannt hätten!)

Wir entwickelten also eine neue Architektur für unser Produkt: *CCU/CMU* (**C**OLT **C**ontrol **U**nit, COLT-Steuergerät und **C**OLT **M**easurement **U**nit, COLT-Messgerät). Die CCU wurde in der zentralen Telefonvermittlung aufgestellt und übernahm das Wählen über die zu testenden Telefonleitungen. Die CMU wurde beim Verteilerkasten vor Ort platziert und konnte die Kupferleitungen durchmessen, die zum Telefon des Kunden führten.

Das nächste Problem bestand darin, dass jede CCU eine Vielzahl von CMUs verwendete. Die Information, welche CMU für eine bestimmte Telefonnummer verwendet werden sollte, war in der digitalen Vermittlungsstelle selbst gespeichert. Die CCU musste also bei der Vermittlungsstelle abfragen, welches CMU gesteuert werden sollte.

Wir sagten den Telefongesellschaften zu, dass die neue Architektur rechtzeitig zur Inbetriebnahme der digitalen Geräte funktionieren würde. Wir wussten, dass es noch

Monate, wenn nicht sogar Jahre dauern würde, bis es soweit war, deshalb hatten wir keine Eile – wir wussten allerdings auch, dass es mehrerer Arbeitsjahre bedurfte, die neue CCU/CMU-Hard- und Software zu entwickeln.

A.7.1 Denkfalle: Die Planung

Im Laufe der Zeit stellten wir fest, dass es immer irgendwelche dringenden Dinge zu erledigen gab, die uns dazu zwangen, die Entwicklung der CCU/CMU-Architektur aufzuschieben. Wir machten uns wegen dieser Entscheidung keine Sorgen, denn die Telefongesellschaften verschoben die Einführung der digitalen Vermittlungsstellen ebenfalls ständig. Wenn wir uns die Planung ansahen, waren wir uns sicher, genügend Zeit zu haben, sodass wir die Entwicklung ständig vor uns herschoben.

Eines Tages rief mein Chef mich in sein Büro und sagte: »Einer unserer Kunden führt nächsten Monat eine digitale Vermittlungsstelle ein. Bis dahin muss die CCU/CMU-Architektur funktionieren.«

Ich war fassungslos! Wie sollten wir mehrere Arbeitsjahre Entwicklung in nur einem Monat leisten? Mein Chef hatte allerdings einen Plan ...

Tatsächlich benötigten wir gar keine vollständige CCU/CMU-Architektur. Die Telefongesellschaft, die die digitale Vermittlungsstelle einführen wollte, war sehr klein. Sie besaß nur eine einzige zentrale Telefonvermittlung und lediglich zwei Verteilerkästen vor Ort. Wichtiger war jedoch, dass die »ortsgebundenen« Verteiler gar nicht so ortsgebunden waren. Tatsächlich befanden sich darin herkömmliche analoge Schaltungen, die die Verbindungen zu einigen Hundert Kunden herstellten. Außerdem waren diese Schaltungen von einem Typ, der von einem normalen COLT angewählt werden konnte. Und noch besser: Die Telefonnummern der Kunden enthielten alle erforderlichen Informationen, um entscheiden zu können, welcher der Verteilerkästen verwendet werden musste: Wenn die Telefonnummer an einer bestimmten Stelle eine 5, 6 oder 7 enthielt, gehörte sie zum Verteilerkasten 1, andernfalls zum Verteilerkasten 2.

Mein Chef erklärte mir also, dass wir tatsächlich gar keine CCU/CMU-Architektur benötigten. Es wurde lediglich ein einfacher Computer in der Telefonvermittlung benötigt, der via Modemleitung mit zwei bei den Verteilerkästen stationierten herkömmlichen COLTs verbunden war. Der SAC würde mit unserem Computer in der Telefonvermittlung kommunizieren, der Computer würde die Telefonnummer decodieren und die Befehle zum Wählen und zum Durchmessen der Leitung an den COLT des jeweiligen Verteilerkastens weiterleiten.

Das Projekt CCU hatte das Licht der Welt erblickt!

Hierbei handelte es sich um das erste bei einem Kunden eingerichtete Produkt, das in

C programmiert war und BOSS verwendete. Für die Entwicklung benötigte ich ungefähr eine Woche. Diese Geschichte liefert zwar keine bedeutsamen architektonischen Einsichten, stellt aber eine passende Einleitung für das nächste Projekt dar.

A.8 DLU/DRU

Anfang der 1980er-Jahre gehörte eine Telefongesellschaft in Texas zu unseren Kunden, deren Versorgungsgebiete ziemlich groß waren. Tatsächlich waren die Gebiete so groß, dass in einem einzigen davon mehrere Standorte erforderlich waren, von denen aus Wartungstechniker entsandt werden konnten. In diesen Niederlassungen gab es Wartungstechniker, die Terminals für den Zugriff auf unseren SAC benötigten.

Nun könnte man meinen, dass sich dieses Problem doch leicht lösen ließe – aber denken Sie daran, dass sich diese Geschichte Anfang der 1980er-Jahre zugetragen hat. Fernbedienungsterminals waren nicht sehr verbreitet. Und noch schlimmer war, dass die Hardware des SAC lokale Terminals voraussetzte. Tatsächlich waren unsere Terminals über einen schnellen proprietären seriellen Bus angeschlossen.

Wir hatten zwar die Möglichkeit, Fernbedienungsterminals anzuschließen, allerdings nur über Modems, deren Geschwindigkeit Anfang der 1980er-Jahre im Allgemeinen auf 300 Bits pro Sekunde (bps) beschränkt war. Mit dieser langsamen Übertragungsrate waren unsere Kunden nicht zufrieden.

Hochgeschwindigkeitsmodems waren zwar schon verfügbar, aber teuer und erforderten besonders beschaffene Standleitungen. Die Leitungsqualität einer Einwählverbindung war für die schnellen Modems definitiv nicht gut genug.

Unsere Kunden verlangten nach einer Lösung. Unsere Antwort war DLU/DRU.

DLU/DRU stand für »**D**isplay **L**ocal **U**nit« (lokales Anzeigegerät) und »**D**isplay **R**emote **U**nit« (entferntes Anzeigegerät). Die DLU war eine Rechnerplatine, die in das Gehäuse des SAC eingesteckt wurde und einen Terminalmanager simulierte. Aber anstatt den seriellen Bus anzusteuern, übertrug die DRU den Zeichenstrom gebündelt per Modem über eine einzelne 9600-bps-Standleitung.

Die DRU befand sich am entfernten Standort beim Kunden und wurde am anderen Ende der 9600-bps-Standleitung angeschlossen. Das Gerät verfügte über die erforderliche Hardware, um die an unserem proprietären seriellen Bus angeschlossenen Terminals ansteuern zu können. Es entschlüsselte die gebündelt über die 9600-bps-Verbindung empfangenen Zeichen und leitete sie an die lokalen Terminals weiter.

Ist es nicht merkwürdig? Wir mussten eigens eine Lösung entwickeln, die heute so allgegenwärtig ist, dass man gar nicht mehr darüber nachdenkt. Aber damals ...

Wir mussten sogar ein eigenes Datenübertragungsprotokoll erfinden, denn seinerzeit waren die Standardprotokolle noch keine quelloffene Shareware. Tatsächlich hat sich all dies abgespielt, lange bevor irgendeine Art von Internetverbindung verfügbar war.

A.8.1 Architektur

Die Architektur dieses Systems war sehr einfach, aber es gibt einige interessante Eigenheiten, auf die ich hinweisen möchte. Zum einen verwendeten beide Geräte unsere 8085-Technologie und beide Systeme waren in C programmiert und nutzten BOSS. Damit haben die Gemeinsamkeiten aber auch schon ein Ende.

Wir arbeiteten zu zweit an diesem Projekt. Ich war der Projektleiter und Mike Carew war mein engster Mitarbeiter. Ich entwarf und programmierte die DLU und Mike kümmerte sich um die DRU.

Die Architektur des DLU beruhte auf einem Datenflussmodell. Alle Funktionen erledigten eine kleine, begrenzte Aufgabe und reichten ihre Ausgabe an die nächste in einer Warteschlange befindliche Funktion weiter. Stellen Sie sich das Ganze wie die Ausgabeumlenkung in UNIX vor. Die Architektur war kompliziert. Manche Funktionen befüllten eine Warteschlange, die von vielen anderen Services genutzt wurde. Andere befüllten eine Warteschlange, die nur von einem einzigen Service geleert wurde.

Stellen Sie sich eine Fertigungsstraße vor. An jeder Position wird eine einzige, einfache, klar definierte Aufgabe erledigt. Ist diese erledigt, wandert das zu fertigende Produkt eine Position weiter. Manche Fertigungsstraßen teilen sich in mehrere Wege auf, die sich manchmal wieder zu einer einzigen Fertigungslinie vereinen. So funktionierte die DLU.

Mikes DRU funktionierte auf bemerkenswert andere Weise. Er erzeugte pro Terminal einen Task und erledigte damit sämtliche für dieses Terminal anfallenden Aufgaben. Keine Warteschlangen. Kein Datenfluss. Es gab nur viele Tasks gleicher Größe, die jeweils ihr eigenes Terminal verwalteten.

Hierbei handelt es sich um das Gegenteil einer Fertigungsstraße. Bei dieser Analogie entspricht das vielen einzelnen Experten, die jeweils das gesamte Produkt selbst zusammenbauen.

Damals hielt ich meine Architektur für überlegen. Und Mike war natürlich der Ansicht, dass die Seine besser sei. Wir haben angeregt darüber diskutiert. Letzten Endes haben natürlich beide Architekturen ziemlich gut funktioniert. Ich habe daraus

die Lehre gezogen, dass sich Softwarearchitekturen himmelweit unterscheiden können und dennoch gleichermaßen effektiv sind.

A.9 VRS

Im Laufe der 1980er-Jahre erschienen immer wieder neue Technologien auf der Bildfläche. Eine davon war die computergesteuerte *Sprachausgabe*.

Zu den Features des 4-TEL-Systems gehörte die Fähigkeit, es dem Wartungstechniker zu ermöglichen, den Ort eines Defekts in einem Kabel aufzuspüren. Das Verfahren lief folgendermaßen ab:

- Der Tester in der zentralen Telefonvermittlung verwendete unser System, um den ungefähren Abstand zum Kabeldefekt zu ermitteln. Das Ergebnis war auf ca. 20 % genau. Der Tester entsandte dann einen Wartungstechniker zu einem entsprechenden Zugangspunkt in der Nähe des Defekts.

- Wenn der Techniker vor Ort war, rief er den Tester an und forderte ihn auf, die Lokalisierung des Defekts zu starten. Der Tester startete das Feature des 4-TEL-Systems zum Lokalisieren von Defekten. Es führte Messungen der elektronischen Eigenschaften des fehlerhaften Kabels durch und gab auf dem Bildschirm Meldungen aus, dass bestimmte Operationen durchgeführt werden sollen, wie zum Beispiel das Kabel kurzzuschließen oder den Kurzschluss wieder zu entfernen.

- Der Tester gab diese Meldungen an den Wartungstechniker weiter, der dem Tester Bescheid gab, wenn er die Operation erledigt hatte. Dann teilte der Tester dem System mit, dass die gewünschte Operation ausgeführt wurde, woraufhin das System mit dem Test fortfuhr.

- Nachdem dieser Ablauf zwei oder drei Mal wiederholt worden war, berechnete das System einen neuen Abstand zum Kabeldefekt. Der Wartungstechniker begab sich dann zu der neuen Position und das Verfahren ging von vorne los.

Versuchen Sie einmal sich vorzustellen, wie viel besser dieses Verfahren gewesen wäre, wenn der am Leitungsmast hochgekletterte oder auf einer Arbeitsplattform stehende Wartungstechniker das System selbst hätte bedienen können. Und genau das war es, was die neue Sprachausgabetechnologie uns ermöglichte. Die Wartungstechniker konnten sich direkt in unser System einwählen, dem System durch Tonwahl Anweisungen erteilen und sich das Ergebnis anhören, das von einer angenehmen Stimme vorgelesen wurde.

A.9.1 Der Name

Das Unternehmen veranstaltete einen kleinen Wettbewerb, um einen Namen für das neue System auszuwählen. Einer der kreativsten Vorschläge lautete SAM CARP, was für »**S**till **A**nother **M**anifestation of **C**apitalist **A**varice **R**epressing the **P**roletariat« (Eine weitere Manifestation der Unterdrückung des Proletariats durch kapitalistische Habgier) stand. Ich muss wohl kaum erwähnen, dass dieser Vorschlag nicht angenommen wurde.

»Teradyne Interactive Test System« (Teradyne Interaktives Testsystem) war ein weiterer Vorschlag, der nicht ausgewählt wurde.

Ebenfalls nicht ausgewählt wurde »Service Area Test Access Network« (Testzugriffsnetzwerk für das Versorgungsgebiet).

Gewonnen hat letztendlich *VRS*: »**V**oice **R**esponse **S**ystem« (Sprachantwortsystem).

A.9.2 Architektur

An diesem System habe ich nicht mitgearbeitet, aber davon gehört, was passierte. Die Geschichte, die ich nun erzählen werde, basiert auf Hörensagen, allerdings glaube ich, dass sie zumindest größtenteils stimmt.

Damals hatten Mikrocomputer, UNIX-Betriebssysteme, C und SQL-Datenbanken Hochkonjunktur. Wir waren entschlossen, all diese Technologien einzusetzen.

Aus der großen Vielfalt von Datenbankanbietern wählten wir schließlich UNIFY aus. UNIFY war ein Datenbanksystem, das unter UNIX lief, und somit perfekt für uns geeignet.

Darüber hinaus unterstützte UNIFY eine neue Technologie, die als *Embedded SQL* (eingebettetes SQL) bezeichnet wurde. Diese Technologie ermöglichte es, SQL-Befehle direkt in unserem C-Code als Strings einzubetten. Und diese Möglichkeit nutzten wir ausgiebig – überall.

Es war einfach unglaublich cool, dass man SQL-Befehle direkt im Code eingeben konnte, wo immer man wollte. Und an welchen Stellen wollten wir das tun? Praktisch überall! Auf diese Weise wurde der gesamte Code mit SQL-Befehlen durchsetzt.

Natürlich war SQL damals noch kaum als ausgereifter Standard zu bezeichnen. Es gab jede Menge herstellerabhängige Eigenheiten. Auf diese Weise fanden auch spezielle SQL-Befehle und der UNIFY-API eigene Aufrufe Eingang in den Code.

Es funktionierte jedenfalls großartig! Das System war ein voller Erfolg. Die

Wartungstechniker nutzten es gern und auch die Telefongesellschaften waren sehr angetan. Friede, Freude, Eierkuchen.

Und dann wurde das von uns genutzte UNIFY-Produkt eingestellt.

Oh je!

Wir entschlossen uns also, zu Sybase zu wechseln. Oder war es Ingress? Ich kann mich nicht erinnern. Jedenfalls mussten wir den gesamten C-Code nach eingebetteten SQL-Befehlen und speziellen API-Aufrufen durchsuchen und durch die entsprechenden Anweisungen für die neue Datenbank ersetzen.

Nachdem wir uns etwa drei Monate lang abgemüht hatten, gaben wir auf. Wir bekamen es einfach nicht zum Laufen. Wir hatten uns so eng an UNIFY gebunden, dass es hoffnungslos war, den Code mit vertretbarem Aufwand umzustrukturieren.

Also beauftragten wir einen externen Dienstleister damit, UNIFY für uns weiterzupflegen, und schlossen einen Wartungsvertrag ab. Der Preis für diesen Wartungsvertrag stieg natürlich jedes Jahr an.

A.9.3 VRS: Fazit

Ich habe unter anderem hieraus gelernt, dass Datenbanken zu den Bestandteilen eines Systems gehören, die vom eigentlichen geschäftlichen Zweck isoliert sein sollten. Darüber hinaus handelt es sich hierbei um einen der Gründe dafür, dass ich eine Abneigung gegen die enge Kopplung an die Software von Drittherstellern habe.

A.10 Der Elektronische Rezeptionist

Im Jahr 1983 stand unser Unternehmen vor dem Zusammenwachsen von Computer-, Telekommunikations- und Sprachausgabesystemen. Unser CEO war der Ansicht, dies sei ein günstiger Ausgangspunkt für die Entwicklung neuer Produkte. Um dieses Ziel zu erreichen, beauftragte er ein Dreierteam (dem ich angehörte), sich ein neues Produkt für das Unternehmen auszudenken, zu konzipieren und zu implementieren.

Es dauerte nicht lange, bis wir den Einfall hatten, den *Elektronischen Rezeptionisten* (ER) zu entwickeln.

Die zugrunde liegende Idee war ganz einfach: Wenn Sie eine Firma anriefen, würde sich der ER melden und Sie fragen, mit wem Sie sprechen möchten. Sie gäben dann per Tonwahl den Namen der entsprechenden Person ein und der ER würde Sie

verbinden. Die ER-Benutzer konnten anrufen und durch einfache Tonwahlbefehle angeben, unter welcher Telefonnummer die gewünschte Person weltweit erreichbar war. Das System konnte sogar mehrere verschiedene Nummern auflisten.

Wenn Sie den ER anriefen und RMART (meinen Code) eingaben, würde der ER die erste auf meiner Liste stehende Nummer wählen. Falls ich den Anruf nicht annähme und mich identifizierte, würde der ER die nächste Nummer wählen usw. Sollte ich nicht erreichbar sein, würde der ER eine Nachricht des Anrufers aufzeichnen.

Der ER würde anschließend in regelmäßigen Abständen versuchen, mich zu erreichen, um diese Nachricht und von anderen für mich hinterlassene Nachrichten zu übermitteln.

Hierbei handelte es sich um das erste Sprachnachrichtensystem, und wir[11] hatten ein Patent darauf angemeldet.

Wir entwickelten die gesamte Hardware dieses Systems selbst – die Computerplatine, die Speicherkarten, die Elektronik für Sprachausgabe und Telefonanbindung, alles. Als Hauptplatine kam das vorhin erwähnte *Deep Thought* mit einem 80286-Prozessor zum Einsatz.

Die Sprachverarbeitungskarten konnten jeweils eine Telefonleitung bedienen. Sie bestanden aus einem Intel-80186-Mikrocomputer und besaßen einen Telefonanschluss, eine Sprachverarbeitungskarte und etwas Arbeitsspeicher.

Die Software für die Hauptplatine war in C programmiert. Als Betriebssystem wurde MP/M-86 verwendet, eins der ersten kommandozeilengesteuerten simultanverarbeitungsfähigen Plattenbetriebssysteme. MP/M war sozusagen das UNIX des kleinen Mannes.

Die Software für die Sprachverarbeitungskarten war in Assembler programmiert, ein Betriebssystem gab es nicht. Die Kommunikation zwischen Deep Thought und den Sprachverarbeitungskarten fand über gemeinsam genutzten Speicher statt.

Heutzutage würde man bei diesem System von einer *serviceorientierten Architektur* (*SOA*) sprechen. Alle Telefonleitungen wurden von einem unter MP/M laufenden Listener-Prozess überwacht.

Wenn ein Anruf einging, wurde ein entsprechender Prozess gestartet, dem der Anruf übergeben wurde. Beim Durchlaufen der verschiedenen Phasen des Anrufs wurden die zugehörigen Serviceprozesse gestartet und übernahmen die Kontrolle.

Der Austausch von Nachrichten zwischen diesen Services fand über auf Festplatte gespeicherte Dateien statt. Der gerade laufende Service ermittelte, welcher Service als Nächstes an der Reihe war, und legte die erforderlichen Informationen über den

Zustand des Systems in einer Datei ab. Anschließend setzte er auf der Kommandozeile den Befehl zum Starten des nächsten Service ab und beendete sich.

Es war das erste Mal, dass ich ein System dieser Art entwickelt hatte. Tatsächlich war ich sogar das erste Mal der Hauptarchitekt eines kompletten Produkts. Ich war für alles zuständig, was irgendwie mit Software zu tun hatte – und es lief wie am Schnürchen.

Ich würde nicht sagen, dass die Architektur dieses Systems im Sinne dieses Buches »sauber« war – es handelte sich ja nicht um eine »Plug-in«-Architektur. Allerdings gab es eindeutig Anzeichen für das Vorhandensein echter Grenzen. Die Services waren voneinander unabhängig einsetzbar und liefen innerhalb ihres eigenen Zuständigkeitsbereichs. Es gab hoch- und tiefschichtige Prozesse, und viele Abhängigkeiten wiesen in die richtige Richtung.

A.10.1 Der Untergang des ER

Bedauerlicherweise lief das Marketing für dieses Produkt nicht besonders gut. Teradyne war ein Unternehmen, das Prüfgeräte verkaufte. Wir hatten keine Ahnung, wie wir auf dem Markt für Bürogeräte hätten Fuß fassen können.

Nach mehreren Versuchen gab unser CEO nach zwei Jahren auf und ließ – leider – die Patentanmeldung fallen. Das Patent wurde der Firma zugesprochen, die drei Monate nach uns einen Patentantrag eingereicht hatte. Wir gaben also den gesamten Markt für Sprachnachrichtensysteme und Anrufweiterleitungen auf.

Autsch!

Aber andererseits kann man mir nicht die Schuld für die nervtötenden Sprachverarbeitungssysteme geben, die uns heutzutage das Leben schwer machen.

A.11 Craft Dispatch System

Als kommerzielles Produkt war der ER gescheitert, aber wir verfügten ja noch immer über die ganze Hard- und Software, die wir zur Verbesserung der vorhandenen Produktlinie nutzen konnten. Darüber hinaus waren wir aufgrund des Marketingerfolgs mit dem VRS überzeugt, dass wir ein sprachgesteuertes interaktives System für Wartungstechniker anbieten sollten, das nicht von unseren Prüfgeräten abhängig war.

So erblickte *CDS*, das **C**raft **D**ispatch **S**ystem (etwa: System zur Entsendung von Wartungstechnikern), das Licht der Welt. Das CDS funktionierte im Wesentlichen wie der ER, es konzentrierte sich jedoch auf das scharf abgegrenzte Gebiet,

Wartungstechniker zur Reparatur vor Ort zu entsenden.

Wenn eine Telefonleitung Schwierigkeiten bereitete, wurde im Servicecenter eine Störfallmeldung erstellt, ein sogenanntes Ticket. Die Tickets wurden von einem automatisierten System verwaltet. Wenn ein Wartungstechniker seine Aufgabe vor Ort erledigt hatte, rief er im Servicecenter an, um sich nach dem nächsten Arbeitsauftrag zu erkundigen. Der Mitarbeiter des Servicecenters suchte nach dem nächsten Ticket und las es dem Wartungstechniker vor.

Wir machten uns daran, diesen Vorgang zu automatisieren. Unsere Zielsetzung lautete, dass der Wartungstechniker das CDS anrufen und um den nächsten Arbeitsauftrag bitten sollte. Das CDS sollte dann das Ticketsystem abfragen und das Ergebnis vorlesen. Das System sollte nachverfolgen, welchem Wartungstechniker ein bestimmtes Ticket zugewiesen worden war, und den Status der Reparatur im Ticketsystem vermerken.

Im Zusammenhang mit der Bedienung des Ticketsystems, der Betriebsführung und den automatisierten Testsystemen wies das CDS einige durchaus interessante Features auf.

Aufgrund der mit der serviceorientierten Architektur des ER gemachten Erfahrungen wollte ich dieselbe Idee etwas aggressiver umsetzen. Der endliche Automat (der die verschiedenen möglichen Zustände repräsentiert) eines Ticketsystems war sehr viel komplizierter als derjenige zur Handhabung eines Anrufs mit dem ER. Ich nahm es also in Angriff, etwas zu entwickeln, das heutzutage als *Microservice-Architektur* bezeichnet wird.

Das System startete bei allen Anrufen für jeden Übergang von einem Zustand zum anderen, wie unbedeutend er auch war, einen neuen Service. Tatsächlich wurde der endliche Automat sogar ausgelagert und in einer externen Textdatei gespeichert, auf die das System zugreifen konnte. Jedes über die Telefonleitung eingehende Ereignis führte zu einem Zustandsübergang des endlichen Automaten. Der vorhandene Prozess startete entsprechend des Zustands des endlichen Automaten einen neuen Prozess, der das eingehende Ereignis verarbeitete. Anschließend wurde der schon vorhandene Prozess beendet oder in eine Warteschlange eingereiht.

Dieser ausgelagerte endliche Automat ermöglichte es uns, den Ablauf der Anwendung zu steuern, ohne irgendwelchen Code zu ändern (das Open-Closed-Prinzip). Wir konnten problemlos einen neuen Service hinzufügen, der von allen anderen unabhängig war, und ihn in den Programmablauf einbauen, indem wir die Textdatei modifizierten, die den endlichen Automaten repräsentierte. Wir konnten das sogar während des laufenden Betriebs tun. Mit anderen Worten: Uns standen *Hot-Swapping* und eine effektive *BPEL* (**B**usiness **P**rocess **E**xecution **L**anguage, eine Sprache zur Beschreibung von Geschäftsprozessen) zur Verfügung.

Der frühere ER-Ansatz, zur Kommunikation zwischen den Services Dateien auf der Festplatte zu verwenden, war für den sehr viel schneller erfolgenden Wechsel von einem Service zum anderen zu langsam, deshalb entwickelten wir einen Mechanismus zur gemeinsamen Speichernutzung, den wir 3DBB[12] tauften. Das 3DBB erlaubte es, über den Namen auf Daten zuzugreifen. Als Namen verwendeten wir die den verschiedenen Instanzen des endlichen Automaten zugewiesenen Bezeichnungen.

Zum Speichern von Strings und Konstanten war das 3DBB bestens geeignet, zum Speichern komplexer Datenstrukturen konnte es jedoch nicht verwendet werden. Der Grund hierfür war technischer Art, aber leicht zu verstehen: Jeder MP/M-Prozess lief in einem eigenen Speicherbereich. Die Zeiger in einem dieser Speicherbereiche waren in einem der anderen bedeutungslos. Dementsprechend durften die Daten im 3DBB keine Zeiger enthalten. Strings waren kein Problem, aber Bäume, verkettete Listen oder andere Datenstrukturen, die Zeiger enthielten, funktionierten nicht.

Die Tickets im Ticketsystem entstammten vielen verschiedenen Quellen. Manche wurden automatisch erstellt, andere von Hand. Die manuellen Einträge wurden von Mitarbeitern erstellt, die sich mit den Kunden über die vorliegenden Probleme unterhielten. Während der Kunde seine Probleme beschrieb, gab der Mitarbeiter die Beschwerden und Beobachtungen in Form eines strukturierten Texts ein. Das sah dann etwa folgendermaßen aus:

```
/pno 8475551212 /noise /dropped-calls
```

Sie verstehen schon: Das Zeichen »/« markierte den Anfang eines neuen Eintrags. Nach dem Schrägstrich folgte ein Code und dem Code folgten Parameter wie »Störgeräusche« oder »Unterbrochene Gespräche«. Es gab *Tausende* solcher Codes, und ein einzelnes Ticket konnte in der Beschreibung Dutzende davon enthalten. Noch schlimmer war, dass die Codes oft Tippfehler enthielten oder falsch formatiert waren, weil sie ja von Hand eingegeben wurden. Sie waren dazu gedacht, von Menschen gelesen zu werden, und für eine maschinelle Verarbeitung ungeeignet.

Unsere Aufgabe war es, diese mehr oder weniger beliebig formatierten Zeichenfolgen zu entziffern, sie zu interpretieren, die Fehler zu korrigieren und sie als Sprachausgabe aufzubereiten, damit sie dem auf einem Leitungsmast befindlichen, mit einem Telefonhörer lauschenden Wartungstechniker vorgelesen werden konnten. Zu diesem Zweck war unter anderem eine äußerst flexible Interpretation und Repräsentierung der Daten erforderlich, die über das 3DBB übermittelt wurden, das nur Strings speichern konnte.

So kam es, dass ich auf einem Flug zu einem Kundenbesuch ein Verfahren entwickelte, dass ich *FLD* nannte (**F**ield **L**abeled **D**ata, vor Ort markierte Daten). Heutzutage würden wir das als XML oder JSON bezeichnen. Das Format ist ein

anderes, aber die zugrunde liegende Idee ist dieselbe. FLDs waren Binärbäume, die in einer rekursiven Hierarchie Bezeichnungen mit Daten verknüpften. Sie konnten durch ein einfaches API abgefragt und in ein komfortables Stringformat umgewandelt werden, das für das 3DBB ideal war.

Zusammengefasst: Microservices, die über gemeinsam genutzten Speicher wie über Sockets kommunizieren und ein XML-ähnliches Format verwenden – im Jahr 1985.

Es gibt nichts Neues unter der Sonne.

A.12 Clear Communications

1988 verließ eine Gruppe von Mitarbeitern Teradyne und gründete ein Start-up namens Clear Communications. Einige Monate später schloss ich mich ihnen an. Wir hatten vor, die Software für ein System zu entwickeln, das die Verbindungsqualität von T1-Leitungen überwachen sollte, den digitalen Leitungen für Fernverbindungen. Wir stellten uns einen riesigen Monitor vor, der eine Landkarte der USA mit den kreuz und quer verlaufenden T1-Leitungen anzeigte, die rot aufleuchteten, wenn sich die Leitungsqualität verschlechterte.

Denken Sie daran, dass grafische Benutzeroberflächen 1988 noch nagelneu waren. Der Apple Macintosh war noch keine fünf Jahre alt. Windows war damals ein schlechter Witz. Aber Sun Microsystems baute Sparcstations mit einer zuverlässigen grafischen Benutzeroberfläche, dem X Window System. Also entschieden wir uns für Sun – und damit auch für C und UNIX.

Wie erwähnt handelte es sich um ein Start-up. Wir arbeiteten 70 bis 80 Stunden pro Woche. Wir hatten eine Vision. Wir waren motiviert. Wir besaßen den Willen. Wir verfügten über die Tatkraft. Wir hatten das nötige Wissen. Wir konnten auf das erforderliche Eigenkapital zurückgreifen. Wir träumten davon, Millionäre zu werden. Wir hatten Rosinen im Kopf.

Der C-Code sprudelte nur so aus uns heraus. Wir knallten ihn hierhin und packten ihn dorthin. Wir bauten gigantische Wolkenschlösser. Es gab Prozesse, Nachrichtenwarteschlangen und prachtvolle, alles je Dagewesene übertreffende Architekturen. Wir programmierten von Grund auf einen Kommunikationsstack mit sieben Schichten – bis hin zur Sicherungsschicht.

Wir schrieben GUI-Code. GUUUI-CODE! Heiliges Kanonenrohr! Wir schrieben GUUUUUUI-Code!

Ich habe selbst eine 3.000 Zeilen lange C-Funktion namens `gi()` geschrieben. Das Kürzel stand für **G**raphic **I**nterpreter. Dabei handelte es sich um ein Paradebeispiel für

verworrenen, unübersichtlichen Code. Das war nicht der einzige schlampige Code, den ich bei Clear schrieb, aber wohl der schlimmste.

Architektur? Soll das ein Scherz sein? Es handelte sich um ein Start-up, da hatten wir doch keine Zeit für *Architektur*. Schreibt verdammt noch mal den Code! *Programmiert um euer Leben!*

Also programmierten wir. Und programmierten und programmierten. Doch drei Jahre später hatten wir es nicht geschafft, etwas zu verkaufen. Nun gut, es gab ein oder zwei Installationen. Aber der Markt war an unserer großen Vision nicht besonders interessiert, und unsere Risikokapitalgeber hatten die Nase voll.

Zu diesem Zeitpunkt habe ich mein Leben gehasst. Alle meine Anstrengungen waren für die Katz gewesen und meine Träume gingen den Bach runter. Ich hatte Ärger auf der Arbeit, wegen der Arbeit Ärger zu Hause und ärgerte mich über mich selbst.

Und dann erhielt ich einen Anruf, der alles ändern sollte.

A.12.1 Die Gegebenheiten

Zwei Jahre vor diesem Telefongespräch geschahen zwei bedeutsame Dinge:

Erstens bekam ich es hin, eine `uucp`-Verbindung (Unix to Unix Copy Protocol) zu einer nahegelegenen Firma einzurichten, die über eine weitere `uucp`-Verbindung mit einer anderen Niederlassung verbunden war, die wiederum über einen Internetzugang verfügte. Dabei handelte es sich natürlich um Einwählverbindungen. Unsere Sparcstation, die auf meinem Schreibtisch stand, nutzte ein 1.200-bps-Modem, um sich zwei Mal täglich bei unserem `uucp`-Host einzuwählen. Auf diese Weise erhielten wir E-Mails und Netnews (eins der ersten sozialen Netzwerke, in dem die Leute interessante Themen diskutierten).

Zweitens veröffentlichte Sun einen C++-Compiler. Ich war seit 1983 an C++ und objektorientierter Programmierung (OOP) interessiert, aber es war schwierig, an Compiler heranzukommen. Als sich mir die Gelegenheit bot, wechselte ich auf der Stelle die Programmiersprache. Ich ließ die 3.000 Zeilen C-Funktionen zurück und begann damit, bei Clear C++-Code zu schreiben. Und ich lernte ...

Ich las Bücher. Natürlich habe ich *The C++ Programming Language* und *The Annotated C++ Reference Manual* (abgekürzt *ARM*) von Bjarne Stroustroup gelesen. Ich las Rebecca Wirfs-Brocks wunderbares Buch *Designing Object Oriented Software*, *OOA* (Object Oriented Analysis), *OOD* (Object Oriented Design) und *OOP* (Object Oriented Programming) von Peter Coad, *Smalltalk-80* von Adele Goldberg, *Advanced C++ Programming Styles and Idioms* von James O. Coplien. Vielleicht am wichtigsten war die Lektüre von *Object Oriented Design with Applications* von Grady

Booch.

Welch ein Name! Grady Booch. Wie könnte man einen Namen wie diesen vergessen? Und außerdem war er *Technischer Direktor* bei einer Firma namens Rational! Ich wollte auch *Technischer Direktor* sein! Also las ich das Buch. Und lernte, und lernte, und lernte ...

Während des Lernens führte ich außerdem auf Netnews verschiedene Debatten, so wie die Leute heutzutage auf Facebook debattieren. Dabei ging es um C++ und OOP. Zwei Jahre lang linderte ich die Frustration, die sich bei der Arbeit aufbaute, indem ich mit Hunderten von Usenet-Nutzern über die besten Sprachmerkmale und die besten Designprinzipien diskutierte. Nach einiger Zeit ergaben meine Äußerungen sogar einen gewissen Sinn.

Während einer dieser Debatten wurden die Grundlagen der SOLID-Prinzipien gelegt.

Durch all diese Debatten und vielleicht sogar durch einige der Dinge, die Sinn ergaben, wurde ich schließlich wahrgenommen ...

A.12.2 Uncle Bob

Einer der Ingenieure bei Clear war ein junger Bursche namens Billy Vogel. Billy verpasste allen Leuten einen Spitznamen. Mich nannte er Uncle Bob. Nun heiße ich zwar tatsächlich Bob, aber ich habe den Verdacht, dass er damit eine flapsige Anspielung auf J.R. »Bob« Dobbs beabsichtigte (siehe https://en.wikipedia.org/wiki/File:Bobdobbs.png).

Zunächst habe ich das hingenommen. Doch als die Monate ins Land gingen, lief sich sein unablässiges Geplapper von »Uncle Bob, ... Uncle Bob«, wenn es um den Stress und die Enttäuschungen ging, die mit dem Start-up verbunden waren, doch allmählich tot.

Und dann klingelte eines Tages das Telefon.

A.12.3 Das Telefongespräch

Es meldete sich ein Personalvermittler. Er hatte meinen Namen von jemandem erhalten, der sich mit C++ und objektorientiertem Design auskannte. Ich bin mir nicht ganz sicher, aber ich glaube, dass es irgendetwas mit meiner Anwesenheit bei Netnews zu tun hatte.

Er sagte, er hätte ein Jobangebot im Silicon Valley, bei einer Firma namens Rational.

Sie benötigte Hilfe bei der Entwicklung eines CASE-Tools.[13]

Ich wurde leichenblass. Ich *wusste*, worum es ging. Mir war zwar nicht klar, warum, aber ich *wusste* es. Das war die Firma, für die *Grady Booch* arbeitete. Mir eröffnete sich die Möglichkeit, mit *Grady Booch* zusammenzuarbeiten!

A.13 ROSE

Im Jahr 1990 ging ich als Auftragsprogrammierer zu Rational. Ich arbeitete an einem Produkt namens ROSE. Dabei handelte es sich um ein Tool, das es Programmierern ermöglichte, Booch-Diagramme zu zeichnen – die Diagramme, über die Grady in seinem Buch *Object Oriented Design with Applications* geschrieben hatte (siehe Abbildung A.9).

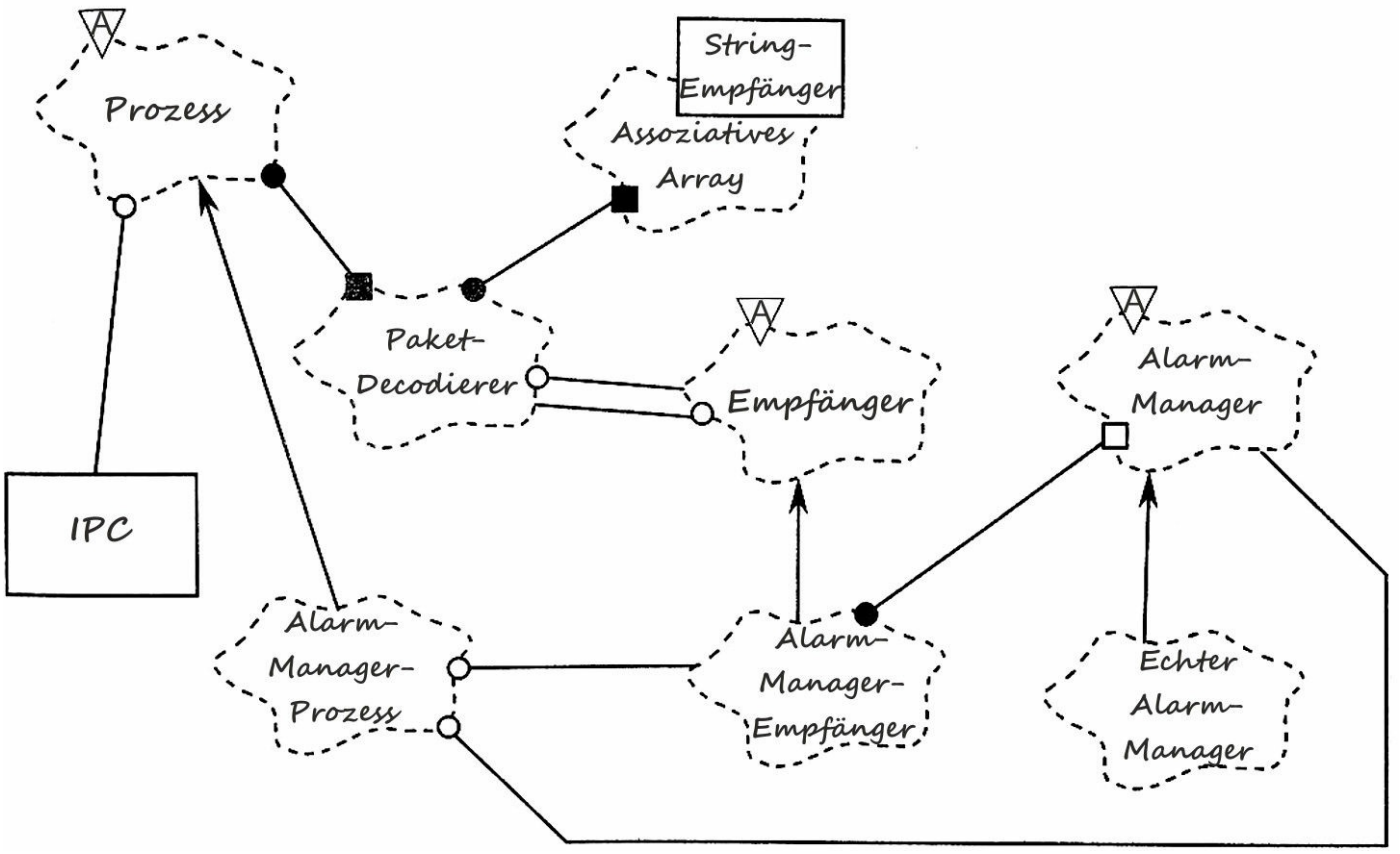


Abb. A.9: Beispiel für ein Booch-Diagramm

Die Booch-Notation war äußerst leistungsfähig und kann als Vorläufer von Modellierungssprachen wie UML betrachtet werden.

ROSE besaß eine Architektur – eine *richtige* Architektur. Sie bestand aus echten Layern, deren Abhängigkeiten voneinander vernünftig festgelegt wurden. Die Architektur sorgte dafür, dass ROSE releasefähig, entwickelbar und unabhängig

deploybar war.

Perfekt war die Architektur allerdings nicht. Viele Aspekte der architektonischen Prinzipien verstanden wir noch nicht.

Außerdem sind wir auf eine der unglückseligsten damaligen Modeerscheinungen hereingefallen: Wir verwendeten eine sogenannte *objektorientierte Datenbank*.

Aber alles in allem war es eine tolle Zeit. Ich arbeite anderthalb sehr erfreuliche Jahre zusammen mit dem Rational-Team an ROSE. Dies war eine der intellektuell anregendsten Erfahrungen meines Berufslebens.

A.13.1 Fortsetzung der Debatten ...

Ich habe natürlich weiterhin an den Debatten auf Netnews teilgenommen. Tatsächlich sogar deutlich öfter als früher. Ich schrieb die ersten Artikel für *C++-Report*. Und ich begann, mit Bradys Hilfe, an meinem ersten Buch zu arbeiten: *Designing Object-Oriented C++ Applications Using the Booch Method*.

Eine Sache ärgerte mich. Es war absurd, aber so war es. Niemand nannte mich noch »Uncle Bob«, und ich musste feststellen, dass ich das vermisste. Also machte ich den Fehler, meine E-Mails und meine Netnews-Beiträge mit »Uncle Bob« zu unterzeichnen. Der Name blieb hängen. Allmählich wurde mir klar, dass »Uncle Bob« ein ziemlich gutes Markenzeichen war.

A.13.2 ... unter anderem Namen

ROSE war eine riesige C++-Anwendung. Sie bestand aus verschiedenen Layern, die einer strengen Abhängigkeitsregel unterworfen waren. Dabei handelte es sich aber nicht um die in diesem Buch beschriebene Regel. Wir ließen die Abhängigkeiten *nicht* in Richtung hochschichtiger Richtlinien weisen. Stattdessen wiesen die Abhängigkeiten in die eher herkömmliche Richtung der Flusssteuerung. Die grafische Benutzeroberfläche wies auf die Repräsentierung, die ihrerseits auf die Änderungsregeln wies, die wiederum auf die Datenbank verwiesen. Dass die Ausrichtung der Abhängigkeiten auf die Richtlinien unterblieb, sorgte letztendlich für den allmählichen Untergang des Produkts.

Die Architektur von ROSE ähnelte der Architektur eines guten Compilers. Die grafische Notation wurde in eine interne Repräsentierung »übersetzt«, die durch bestimmte Regeln manipuliert und in einer objektorientierten Datenbank gespeichert wurde.

Objektorientierte Datenbanken waren ein relativ neues Konzept, und in der Welt der objektorientierten Programmierung herrschte in Anbetracht der Folgen helle Aufregung. Alle Programmierer wollten in ihren Systemen plötzlich objektorientierte Datenbanken einsetzen. Die zugrunde liegende Idee war vergleichsweise einfach und zutiefst idealistisch: Die Datenbank speicherte Objekte, nicht Tabellen, und sollte sich wie RAM verhalten. Wenn man auf ein Objekt zugriff, tauchte es einfach im Arbeitsspeicher auf. Wenn das fragliche Objekt auf ein anderes Objekt verwies, erschien es ebenfalls im Arbeitsspeicher, sobald man darauf zugriff. Es war wie Zauberei.

Die Datenbank war wohl unser größter in der Praxis spürbarer Fehler. Wir wollten die Zauberei, bekamen es aber mit einem umfangreichen, langsamen und teuren Framework eines Drittherstellers zu tun, das uns das Leben zur Hölle machte, indem es uns auf nahezu allen Ebenen darin hinderte, Fortschritte zu erzielen.

Die Datenbank war allerdings nicht der einzige Fehler, den wir begingen. Tatsächlich war die übertriebene Architektur der größte Fehler. Es gab sehr viel mehr Layer, als ich hier beschrieben habe, und jeder einzelne verursachte bei der Kommunikation eine eigene Art von Verwaltungsaufwand. Die Produktivität des Teams wurde dadurch erheblich beeinträchtigt.

Nach mehreren Mannjahren Arbeitsaufwand, immensen Anstrengungen und zwei halbherzigen Releases wurde das Tool als Ganzes über Bord geworfen und durch eine nette kleine Anwendung ersetzt, die ein kleines Team in Wisconsin geschrieben hatte.

Auf diese Weise musste ich erfahren, dass eine großartige Architektur manchmal auch zu grandiosem Scheitern führen kann. Sie muss flexibel genug sein, um sich an den Umfang einer Aufgabe anzupassen. Wenn die Architektur für ein Großunternehmen ausgelegt wird, obwohl man eigentlich nur ein hübsches kleines Hilfsprogramm für den Arbeitsplatzrechner benötigt, ist ein Misserfolg vorprogrammiert.

A.14 Prüfung zum eingetragenen Architekten

Anfang der 1990er-Jahre wurde ich ein »richtiger« Berater. Ich reiste durch die Welt und lehrte, was es mit dieser objektorientierten Programmierung auf sich hatte. Dabei konzentrierte ich mich vornehmlich auf das Design und die Architektur objektorientierter Systeme.

Zu meinen ersten Klienten gehörte der Educational Testing Service (ETS, eine gemeinnützige Organisation, die standardisierte Prüfungsverfahren anbietet). Das Unternehmen hatte einen Vertrag mit dem National Council of Architects Registry Board (NCARB) zur Durchführung der Prüfungen zum eingetragenen Architekten

abgeschlossen.

Wenn man sich in den Vereinigten Staaten oder in Kanada offiziell als Architekt (die Art von Architekt, die Gebäude entwirft) eintragen lassen möchte, muss man eine Prüfung ablegen. Bei dieser Prüfung muss der Kandidat eine Reihe von architektonischen Aufgaben lösen, die das Design von Gebäuden betreffen. Dem Kandidaten wurden manchmal auch verschiedene Anforderungen zum Beispiel für eine öffentliche Bibliothek, ein Restaurant oder eine Kirche vorgelegt. Anschließend musste er die dazugehörigen Architekturskizzen anfertigen.

Die Ergebnisse wurden gesammelt und verwahrt, bis eine Gruppe erfahrener Architekten die Zeit fand, sich zu treffen und die eingereichten Dokumente als Juroren zu bewerten. Diese Treffen waren große, kostspielige Veranstaltungen und führten oft zu widersprüchlichen Ansichten und Verzögerungen.

NCARB wollte dieses Prüfungsverfahren automatisieren, indem die Kandidaten die Prüfung am Computer absolvierten. Die Auswertung und die Benotung sollten ebenfalls von einem Computer erledigt werden. NCARB bat ETS, eine entsprechende Software zu entwickeln, und ETS beauftragte mich damit, ein Entwicklerteam dafür zusammenzustellen.

ETS hatte die Aufgaben in 18 Einzeltests aufgeteilt, für die jeweils eine CAD-ähnliche GUI-Anwendung erforderlich war, die der Prüfling zum Erstellen der Lösung verwendete. Eine zweite Anwendung nahm die Lösungen entgegen und lieferte die erreichte Punktzahl.

Mein Kollege Jim Newkirk und ich stellten fest, dass diese 36 Anwendungen sehr viel gemeinsam hatten. In den 18 GUI-Anwendungen kamen ähnliche Anweisungen und Mechanismen zum Einsatz. Und die 18 Benotungsanwendungen verwendeten dasselbe mathematische Verfahren. In Anbetracht dieser Gemeinsamkeiten beschlossen Jim und ich, ein wiederverwendbares Framework für die 36 Anwendungen zu entwickeln. Tatsächlich machten wir ETS diese Vorgehensweise schmackhaft, indem wir erklärten, dass wir für das Erstellen der ersten Anwendung zwar länger benötigen würden, die übrigen aber jeweils innerhalb weniger Wochen liefern könnten.

Jetzt sollten Sie die Hände über dem Kopf zusammenschlagen oder wenigstens die Stirn runzeln. Diejenigen Leser, die alt genug sind, erinnern sich vielleicht noch an das Versprechen der »Wiederverwendbarkeit« der objektorientierten Programmierung. Damals waren wir alle davon überzeugt, dass beim Schreiben sauberen objektorientierten C++-Codes quasi von allein jede Menge wiederverwendbarer Code entsteht.

Wir machten uns also daran, die erste Anwendung zu schreiben – die komplizierteste von allen. Sie hieß »Vignette Grande«.

Wir beide arbeiteten Vollzeit an Vignette Grande und achteten dabei darauf, wiederverwendbare Frameworks zu erstellen. Dafür benötigten wir ein Jahr. Am Ende dieses Jahres hatten wir 45.000 Zeilen Framework-Code und 6.000 Zeilen Anwendungscode geschrieben. Wir lieferten das Produkt an ETS aus, und sie beauftragten uns, schnellstens die übrigen 17 Anwendungen zu programmieren.

Jim und ich stellten ein Team von drei weiteren Entwicklern ein und begannen damit, an den nächsten Anwendungen zu arbeiten.

Aber irgendetwas ging schief. Wir mussten feststellen, dass unser wiederverwendbares Framework gar nicht besonders wiederverwendbar war. Es passte irgendwie nicht richtig zu den neu geschriebenen Anwendungen. Es gab kleine Reibungspunkte, die einfach nicht funktionieren wollten.

Es war zutiefst entmutigend, doch wir glaubten zu wissen, was wir unternehmen konnten. Wir suchten ETS auf und teilten ihnen mit, dass es eine Verzögerung geben würde. Das aus 45.000 Codezeilen bestehende Framework müsse umgeschrieben oder zumindest angepasst werden. Wir sagten ihnen, dass es etwas länger dauern würde, dies zu erledigen.

Ich muss Ihnen wohl kaum erzählen, dass ETS nicht gerade begeistert war.

Wir mussten also von vorn anfangen. Wir ließen das alte Framework außer Acht und schrieben vier Anwendungen gleichzeitig. Wir machten Anleihen beim alten Framework und übernahmen Teile des Codes, den wir so überarbeiteten, dass er ohne weitere Modifikationen mit allen vier Anwendungen funktionierte. Wir benötigten dafür ein weiteres Jahr. Das neue Framework bestand wieder aus 45.000 Codezeilen und die vier Anwendungen waren jeweils zwischen 3.000 und 6.000 Zeilen lang.

Es ist wohl unnötig zu erwähnen, dass die Beziehung zwischen den GUI-Anwendungen und dem Framework den Abhängigkeitsregeln folgte. Die Anwendungen waren Plug-ins für das Framework. Alle hochschichtigen Richtlinien der grafischen Benutzeroberfläche waren Bestandteil des Frameworks. Die Anwendungen waren nur »Glue Code«.

Die Beziehung zwischen den Benotungsanwendungen und dem Framework war etwas komplizierter. Die hochschichtigen Richtlinien der Benotung waren Teil der normalen Anwendung. Das Benotungs-Framework war ein Plug-in für die Benotungsanwendung.

Natürlich waren beide Anwendungen statisch verlinkte C++-Programme, daher spielte das Plug-in bei unseren Überlegungen überhaupt keine Rolle. Und dennoch befand sich die Ausrichtung der Abhängigkeiten in Einklang mit der Abhängigkeitsregel.

Nach der Auslieferung dieser vier Anwendungen nahmen wir die nächsten vier in

Angriff. Dieses Mal konnten wir wie vorhergesagt alle paar Wochen eine fertigstellen. Die Verzögerung hatte unsere Planung um fast ein Jahr zurückgeworfen, deshalb stellten wir zur Beschleunigung des Ablaufs einen weiteren Programmierer ein.

Wir konnten unsere Termine und Zusagen einhalten. Unser Kunde war zufrieden, und wir waren es auch. Das Leben war schön.

Es war uns jedoch eine Lehre: Man kann ein Framework nicht wiederverwendbar machen, ohne vorher ein wiederverwendbares Framework erstellt zu haben. Beim Erstellen eines wiederverwendbaren Frameworks ist es erforderlich, dieses zusammen mit *mehreren* Anwendungen zu entwickeln, die es verwenden.

A.15 Fazit

Wie ich eingangs bereits schrieb, trägt dieser Anhang autobiografische Züge. Ich habe die Höhepunkte der Projekte aufgezählt, die nach meinem Gefühl für die Architektur von Bedeutung waren. Darüber hinaus habe ich natürlich auch einige Begebenheiten erwähnt, die für den technischen Inhalt dieses Buches nicht relevant sind, aber dennoch wichtig waren.

Natürlich ist diese Chronik nicht vollständig. Ich habe im Laufe der Jahrzehnte an vielen anderen Projekten mitgewirkt. Außerdem habe ich diese Chronik ganz bewusst Anfang der 1990er-Jahre enden lassen – weil ich ein weiteres Buch über die Ereignisse schreiben möchte, die sich Ende der 1990er-Jahre zugetragen haben.

Ich kann mir nur wünschen, dass Ihnen dieser kleine Ausflug in meine Vergangenheit gefallen hat und dass Sie dabei etwas haben lernen können.

---

[1] Über diese bei ASC aufgestellte Maschine kursiert die Geschichte, dass sie im Rahmen eines Haushaltsmöbeltransports auf dem Anhänger eines Sattelschleppers angeliefert wurde. Während des Transports kollidierte der Sattelschlepper bei hoher Geschwindigkeit mit einer Brücke. Der Computer blieb heil, rutschte jedoch nach vorn und zermalmte die Möbel, von denen nur noch Holzsplitter übrig blieben.

[2] Heutzutage würden wir davon sprechen, dass die Taktfrequenz 142 kHz betrug.

[3] Versuchen Sie, sich eine Vorstellung von der Masse dieser Platten zu machen. Und von der kinetischen Energie! Eines Tages stellten wir fest, dass sich neben einem der Taster des Gehäuseschranks feine Metallspäne ansammelten. Also riefen wir den Wartungstechniker an, der uns empfahl, das Gerät erst einmal abzuschalten. Als er schließlich kam, um es zu reparieren, stellte er fest, dass eins der Lager verschlissen

war. Dann gab er Geschichten zum Besten, dass sich diese Platten, wenn sie nicht repariert würden, aus ihrer Verankerung losreißen könnten und sich ihren Weg durch Betonwände bahnen und in auf dem Parkplatz stehende Autos bohren würden.

[4] Monochrombildschirme mit grüner Schrift und ASCII-Anzeige

[5] Die Zahl 72 ist auf die Lochkarten von Hollerith zurückzuführen, die jeweils 80 Zeichen speicherten. Die letzten acht Zeichen waren allerdings für Sequenznummern reserviert, damit man die Karten wieder korrekt sortieren konnte, falls ein solcher Lochkartenstapel durcheinandergeriet.

[6] Mir ist schon klar, dass diese Bezeichnung ein Widerspruch in sich ist.

[7] Die Bauteile besaßen ein durchsichtiges Kunststofffenster, durch das man den Siliziumchip sehen konnte und das es ermöglichte, den Chip mit UV-Licht zu löschen.

[8] In ROM gebrannte Software wird eigentlich als Firmware bezeichnet, aber auch Firmware ist Veränderungen unterworfen.

[9] RKO7

[10] Später wurde diese Bezeichnung in »Bob’s Only Successful Software« (Bobs einzige erfolgreiche Software) umbenannt.

[11] Das Patent war auf die Firma angemeldet. In unseren Arbeitsverträgen war eindeutig festgelegt, dass die Firma die Rechte für alle unsere Erfindungen besaß. Mein Chef sagte mir: »Du hast es uns für einen Dollar verkauft, und wir haben dir diesen einen Dollar nicht gezahlt.«

[12] Three-Dimensional Black Board (Dreidimensionales schwarzes Brett)

[13] Computer Aided Software Engineering (rechnergestützte Softwareentwicklung)

Anhang B
Nachwort

Meine berufliche Laufbahn als Softwareentwickler nahm in den 1990er-Jahren ihren Anfang – zu einer Zeit, als die Dinosaurier großer Architekturen die Welt beherrschten. Man musste sich mit Objekten, Komponenten, Entwurfsmustern und UML (*Unified Modeling Language*, vereinheitlichte Modellierungssprache) befassen (oder mit deren Vorläufern), wenn man weiterkommen wollte.

Am Anfang von Projekten (meine Güte, sollten wir den Tag verfluchen, als wir uns für diese Bezeichnung entschieden haben?) standen lange Designphasen, in denen höherrangige Programmierer detaillierte Konzepte entwickelten, die von den nachrangigeren Programmierern umgesetzt werden sollten. Was sie natürlich nicht taten. Nie.

So kam es dazu, dass ich nach der Ernennung zum »Softwarearchitekten« (und später zum »Leitenden Softwarearchitekten«, »Obersoftwarearchitekten« und »Minister und Geheimrat für Softwarearchitektur« sowie all den anderen hochgestochenen Berufsbezeichnungen, die wir uns damals verliehen haben) offenbar dazu verdammt war, den Rest meines Lebens damit zu verbringen, verschiedene Kästchen durch Pfeile zu verbinden, PowerPoint zu »programmieren« und kaum noch irgendeinen Einfluss auf den eigentlichen Code nehmen zu können.

Mir wurde schlagartig klar, dass das Unsinn war. Für jede einzelne Codezeile muss mindestens eine Designentscheidung getroffen werden, und deshalb kann jeder Programmierer einen erheblich größeren Einfluss auf die Softwarequalität nehmen, als es einem PowerPoint-Guru wie mir jemals möglich gewesen wäre.

Und dann trat endlich die *Agile Softwareentwicklung* auf den Plan und erlöste Softwarearchitekten wie mich von ihrem Elend. Ich bin Programmierer. Ich mag es zu programmieren. Und die beste Methode, einen positiven Einfluss auf den Code auszuüben, ist immer noch, ihn selbst zu schreiben.

Die Dinosaurier der großen Softwarearchitekturen, die meist in den vorsintflutlichen Gefilden umfassender Prozesse umherstreiften, starben durch den Meteoriteneinschlag in Form des *Extreme Programmings* aus. Eine durchaus willkommene Erleichterung.

Die Entwicklerteams konnten sich endlich wieder auf das Wesentliche konzentrieren und sich mit den Dingen befassen, die einen echten Mehrwert bedeuteten. Statt Wochen oder Monate darauf zu warten, dass irgendein Architekturdokument vollendet wurde, konnten sie dies pflichtschuldigst ignorieren und den Code schreiben, den sie ohnehin schreiben wollten. Die Teams konnten mit ihren Kunden Tests vereinbaren,

um schnelle Designentscheidungen zu treffen und das erwünschte Verhalten zu erzielen, und schließlich den von vornherein geplanten Code liefern.

Die Dinosaurier der großen Architekturen waren verschwunden – und wir wurden durch kleine, flinke »So-gerade-eben-hinreichendes-Vorab-Design-mit-jeder-Menge-Refaktorisierungen«-Säugetiere ersetzt. Die Softwarearchitektur öffnete sich.

Nun ja, zumindest in der Theorie.

Die Architektur den Programmierern zu überlassen, bringt das Problem mit sich, dass sie in der Lage sein müssen, wie Architekten zu denken. Wie sich herausstellte, waren nicht alle Dinge, die wir in der Ära der großen Architekturen gelernt hatten, wertlos. Die Strukturierung der Software kann einen entscheidenden Einfluss auf unsere Fähigkeit haben, sie anzupassen und weiterzuentwickeln, sogar kurzfristig.

Bei jeder Designentscheidung muss man die Tür für zukünftige Anpassungen einen Spalt breit offen lassen. Wie beim Poolbilliard geht es nicht nur darum, die Kugel zu versenken, sondern auch darum, den nächsten Stoß vorzubereiten. Funktionierenden Code zu schreiben, der zukünftigem Code nicht im Weg steht, ist eine Fertigkeit, die alles andere als trivial ist. Es dauert Jahre, sie zu meistern.

Die Ära der großen Architekturen machte also für ein neues Zeitalter der fragilen Architekturen Platz: Designs, die schnell wuchsen, um frühzeitiger von Nutzen zu sein, es aber andererseits sehr schwierig machten, dieses Innovationstempo beizubehalten.

Es war immer davon die Rede, »Veränderungen willkommen zu heißen«, aber wenn es 500 Dollar kostet, eine Codezeile zu modifizieren, wird es keine Änderungen geben.

Bob Martins ursprüngliche Arbeiten über objektorientierte Designprinzipien haben mich als jungen Softwareentwickler stark beeinflusst. Ich betrachtete meinen Code aus einer neuen Perspektive und bemerkte Probleme, die mir vorher nie als solche erschienen waren.

Inzwischen wissen Sie, dass es möglich ist, Code zu schreiben, der schon heute von Nutzen ist, ohne zukünftigen Nutzen zu verhindern. Es ist Ihre Aufgabe, das in der Praxis umzusetzen, damit Sie diese Prinzipien auf Ihren eigenen Code anwenden können.

Wie das Fahrradfahren kann man Softwaredesign nicht lernen, indem man nur darüber liest. Sie müssen praktische Erfahrungen machen, um den größten Nutzen aus einem Buch wie diesem zu ziehen. Analysieren Sie Ihren Code und suchen Sie dabei nach den Problemen, die Bob aufzeigt. Refaktorisieren Sie den Code anschließend, um diese Probleme zu lösen. Und wenn die Refaktorisierung für Sie noch Neuland ist, wird diese Erfahrung von doppeltem Wert für Sie sein.

Finden Sie heraus, wie Sie diese Designprinzipien und eine saubere Architektur in Ihren Entwicklungprozess integrieren können, damit es weniger wahrscheinlich wird, dass neuer Code Ihnen Kopfschmerzen bereitet. Wenn Sie beispielsweise *TDD-Tests* (Test Driven Development, testgetriebene Entwicklung) einsetzen, sollten Sie das Design nach jedem bestandenen Test erneut in Augenschein nehmen und vor dem Fortfahren aufräumen. (Das ist erheblich kostengünstiger als schlechte Designs erst später zu korrigieren.) Oder bitten Sie einen Kollegen, den Code mit Ihnen zusammen zu überprüfen, bevor Sie ihn in die Versionsverwaltung einstellen. Ziehen Sie auch in Betracht, eine »Qualitätsschranke« in Ihre Build-Pipeline einzubinden, die einen letzten Schutz vor einer unsauberen Softwarearchitektur bietet. (Und falls Sie keine Build-Pipeline nutzen, ist jetzt vielleicht der richtige Zeitpunkt gekommen, eine einzurichten.)

Am allerwichtigsten ist jedoch, über die saubere Architektur zu diskutieren. Reden Sie mit Ihrem Team. Erörtern Sie das Thema mit der Entwicklergemeinde. Qualität geht jeden etwas an – und es ist von großer Bedeutung, einen Konsens über die Unterschiede zwischen guter und schlechter Architektur zu erreichen.

Sie müssen sich allerdings darüber im Klaren sein, dass die meisten Softwareentwickler der Architektur kaum Beachtung schenken – mir ging es vor 25 Jahren nicht anders, bis mich erfahrenere Entwickler darauf hingewiesen haben. Wenn Sie die saubere Architektur komplett verstanden haben, sollten Sie sich die Zeit nehmen, Ihr Wissen weiterzugeben.

Die Technologielandschaft der Entwickler unterliegt einem ständigen Wandel, während sich die hier beschriebenen fundamentalen Prinzipien jedoch nur selten ändern. Ich habe kaum einen Zweifel daran, dass dieses Buch noch viele Jahre seinen Platz in Ihrem Bücherregal haben wird – und kann mir nur wünschen, dass es Ihren Designfähigkeiten ebenso zuträglich sein wird, wie es Bobs ursprüngliche Arbeiten für die meinen waren.

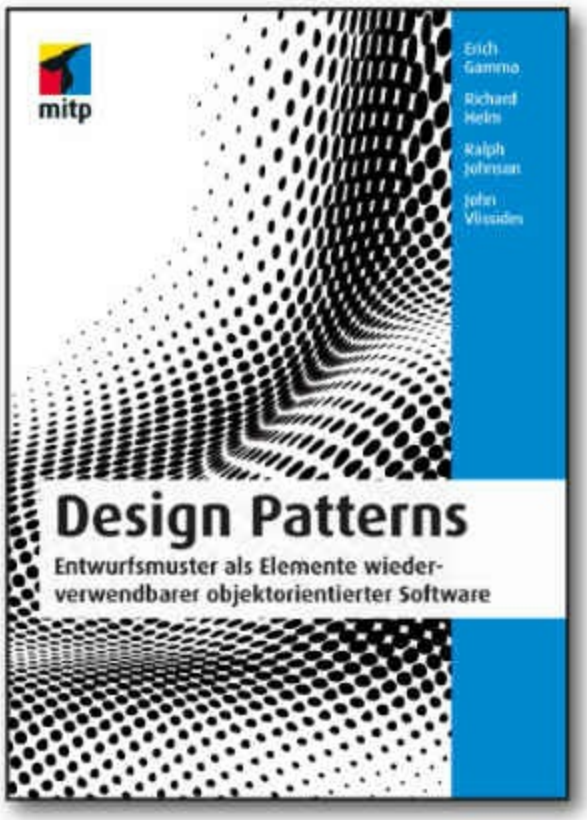
Die eigentliche Reise beginnt erst jetzt.

Jason Gorman, 26. Januar 2017

**Erich Gamma, Richard Helm,
Ralph Johnson, John Vlissides**

Design Patterns

Entwurfsmuster als Elemente wiederverwendbarer objektorientierter Software



- **Der Bestseller von Gamma und Co. in komplett neuer Übersetzung**
- **Das Standardwerk für die objektorientierte Softwareentwicklung**
- **Zeitlose und effektive Lösungen für wiederkehrende Aufgaben im Softwaredesign**

Mit Design Patterns lassen sich wiederkehrende Aufgaben in der objektorientierten Softwareentwicklung effektiv lösen. Die Autoren stellen einen Katalog einfacher und prägnanter Lösungen für häufig auftretende Aufgabenstellungen vor. Mit diesen 23 Patterns können Softwareentwickler flexiblere, elegantere und vor allem auch wiederverwendbare Designs erstellen, ohne die Lösungen jedes Mal aufs Neue selbst entwickeln zu müssen.

Die Autoren beschreiben zunächst, was Patterns eigentlich sind und wie sie sich beim Design objektorientierter Software einsetzen lassen. Danach werden die stets wiederkehrenden Designs systematisch benannt, erläutert, beurteilt und katalogisiert. Mit diesem Leitfaden lernen Sie, wie sich diese wichtigen Patterns in den Softwareentwicklungsprozess einfügen und wie sie zur Lösung Ihrer eigenen Designprobleme am besten eingesetzt werden.

Bei jedem Pattern ist angegeben, in welchem Kontext es besonders geeignet ist und welche Konsequenzen und Kompromisse sich aus der Verwendung des Patterns im Rahmen des Gesamtdesigns ergeben. Sämtliche Patterns entstammen echten Anwendungen und beruhen auf tatsächlich existierenden Vorbildern. Außerdem ist jedes Pattern mit Codebeispielen versehen, die demonstrieren, wie es in objektorientierten Programmiersprachen wie C++ oder Smalltalk implementiert werden kann.

Das Buch eignet sich nicht nur als Lehrbuch, sondern auch hervorragend als Nachschlagewerk und Referenz und erleichtert so auch besonders die Zusammenarbeit im Team.

Probekapitel und Infos erhalten Sie unter:

www.mitp.de/9700 ISBN 978-3-8266-9700-5

Sam Newman

Microservices

Konzeption und Design

- **Feingranulare Systeme mit Microservices aufbauen**
- **Design, Entwicklung, Deployment, Testen und Monitoring**
- **Sicherheitsaspekte, Authentifizierung und Autorisierung**

Verteilte Systeme haben sich in den letzten Jahren stark verändert: Große monolithische Architekturen werden zunehmend in viele kleine, eigenständige Microservices aufgespalten. Aber die Entwicklung solcher Systeme bringt Herausforderungen ganz eigener Art mit sich. Dieses Buch richtet sich an Softwareentwickler, die sich über die zielführenden Aspekte von Microservice-Systemen wie Design, Entwicklung, Testen, Deployment und Monitoring informieren möchten.

Sam Newman veranschaulicht und konkretisiert seine ganzheitliche Betrachtung der grundlegenden Konzepte von Microservice-Architekturen anhand zahlreicher praktischer Beispiele und Ratschläge. Er geht auf die Themen ein, mit denen sich Systemarchitekten und Administratoren bei der Einrichtung, Verwaltung und Entwicklung dieser Architekturen in jedem Fall auseinandersetzen müssen.

Aus dem Inhalt:

- Vorteile von Microservices
- Gestaltung von Services
- Ausrichtung der Systemarchitektur an der Organisationsstruktur
- Möglichkeiten zur Integration von Services
- Schrittweise Aufspaltung einer monolithischen Codebasis
- Deployment einzelner Microservices mittels Continuous Integration
- Testen und Monitoring verteilter Systeme
- Sicherheitsaspekte
- Authentifizierung und Autorisierung zwischen Benutzer und Service bzw. zwischen Services untereinander
- Skalierung von Microservice-Architekturen

Probekapitel und Infos erhalten Sie unter:
www.mitp.de/081

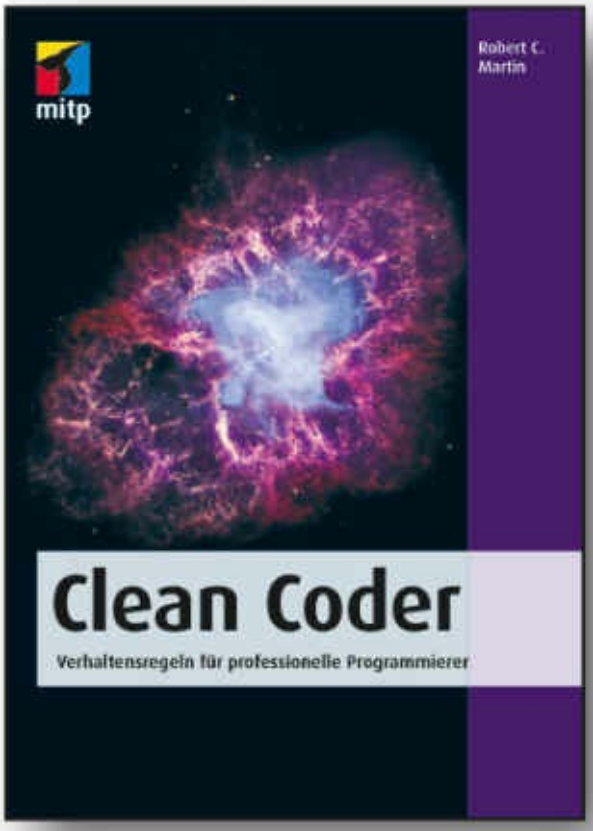
ISBN 978-3-95845-081-3

Robert C. Martin

Clean Coder

Verhaltensregeln für professionelle Programmierer

Der Nachfolger von „Uncle Bobs" erfolgreichem Bestseller *Clean Code*



Erfolgreiche Programmierer haben eines gemeinsam: Die Praxis der Software-Entwicklung ist ihnen eine Herzensangelegenheit. Auch wenn sie unter einem nicht nachlassenden Druck arbeiten, setzen sie sich engagiert ein. Software-Entwicklung ist für sie eine Handwerkskunst.

In Clean Coder stellt der legendäre Software-Experte Robert C. Martin die Disziplinen, Techniken, Tools und Methoden vor, die Programmierer zu Profis machen.

Dieses Buch steckt voller praktischer Ratschläge und behandelt alle wichtigen Themen vom professionellen Verhalten und Zeitmanagement über die Aufwandsschätzung bis zum Refactoring und Testen. Hier geht es um mehr als nur um Technik: Es geht um die innere Haltung. Martin zeigt, wie Sie sich als Software-Entwickler professionell verhalten, gut und sauber arbeiten und verlässlich kommunizieren und planen. Er beschreibt, wie Sie sich schwierigen Entscheidungen stellen und zeigt, dass das eigene Wissen zu verantwortungsvollem Handeln verpflichtet.

In diesem Buch lernen Sie:

- Was es bedeutet, sich als echter Profi zu verhalten
- Wie Sie mit Konflikten, knappen Zeitplänen und unvernünftigen Managern umgehen
- Wie Sie beim Programmieren im Fluss bleiben und Schreibblockaden überwinden
- Wie Sie mit unerbittlichem Druck umgehen und Burnout vermeiden
- Wie Sie Ihr Zeitmanagement optimieren
- Wie Sie für Umgebungen sorgen, in denen Programmierer und Teams wachsen und sich wohlfühlen
- Wann Sie „Nein" sagen sollten – und wie Sie das anstellen
- Wann Sie „Ja" sagen sollten – und was ein Ja wirklich bedeutet

Großartige Software ist etwas Bewundernswertes: Sie ist leistungsfähig, elegant, funktional und erfreut bei der Arbeit sowohl den Entwickler als auch den Anwender. Hervorragende Software wird nicht von Maschinen geschrieben, sondern von Profis, die sich dieser Handwerkskunst unerschütterlich verschrieben haben. Clean Coder hilft Ihnen, zu diesem Kreis zu gehören.

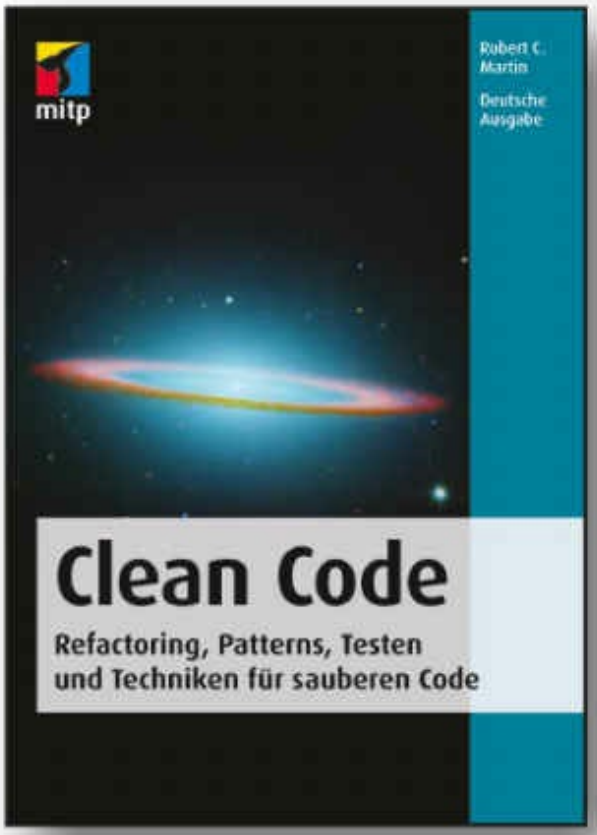
Probekapitel und Infos erhalten Sie unter:
www.mitp.de/9695

ISBN 978-3-8266-9695-4

Robert C. Martin

Clean Code

Refactoring, Patterns, Testen
und Techniken für sauberen Code



- Kommentare, Formatierung, Strukturierung
- Fehler-Handling und Unit-Tests
- Zahlreiche Fallstudien, Best Practices, Heuristiken und Code Smells

Selbst schlechter Code kann funktionieren. Aber wenn der Code nicht sauber ist, kann er ein Entwicklungsunternehmen in die Knie zwingen. Jedes Jahr gehen unzählige Stunden und beträchtliche Ressourcen verloren, weil Code schlecht geschrieben ist. Aber das muss nicht sein.

Mit Clean Code präsentiert Ihnen der bekannte Software-Experte Robert C. Martin ein revolutionäres Paradigma, mit dem er Ihnen aufzeigt, wie Sie guten Code schreiben und schlechten Code überarbeiten. Zusammen mit seinen Kollegen von Object Mentor destilliert er die besten Praktiken der agilen Entwicklung von sauberem Code zu einem einzigartigen Buch. So können Sie sich die Erfahrungswerte der Meister der Software-Entwicklung aneignen, die aus Ihnen einen besseren Programmierer machen werden – anhand konkreter Fallstudien, die im Buch detailliert durchgearbeitet werden.

Sie werden in diesem Buch sehr viel Code lesen. Und Sie werden aufgefordert, darüber nachzudenken, was an diesem Code richtig und falsch ist. Noch wichtiger: Sie werden herausgefordert, Ihre professionellen Werte und Ihre Einstellung zu Ihrem Beruf zu überprüfen.

Clean Code besteht aus drei Teilen: Der erste Teil beschreibt die Prinzipien, Patterns und Techniken, die zum Schreiben von sauberem Code benötigt werden. Der zweite Teil besteht aus mehreren, zunehmend komplexeren Fallstudien. An jeder Fallstudie wird aufgezeigt, wie Code gesäubert wird – wie eine mit Problemen behaftete Code-Basis in eine solide und effiziente Form umgewandelt wird. Der dritte Teil enthält den Ertrag und den Lohn der praktischen Arbeit: ein umfangreiches Kapitel mit Best Practices, Heuristiken und Code Smells, die bei der Erstellung der Fallstudien zusammengetragen wurden. Das Ergebnis ist eine Wissensbasis, die beschreibt, wie wir denken, wenn wir Code schreiben, lesen und säubern.

Dieses Buch ist ein Muss für alle Entwickler, Software-Ingenieure, Projektmanager, Team-Leiter oder Systemanalytiker, die daran interessiert sind, besseren Code zu produzieren.

Probekapitel und Infos erhalten Sie unter:
www.mitp.de/5548

ISBN 978-3-8266-5548-7

Table of Contents