



**Robert C.
Martin**

Mit Beiträgen von
James Grenning
und Simon Brown

Vorwort von
Kevlin Henney

Nachwort von
Jason Gorman

Deutsche Ausgabe

Clean Architecture

**Das Praxis-Handbuch
für professionelles Softwaredesign**

**Regeln und Paradigmen
für effiziente Softwarestrukturierung**

Inhaltsverzeichnis

	Vorwort	15
	Einleitung	19
	Über den Autor	23
	Danksagung	24
Teil I	Einführung	25
1	Was bedeuten »Design« und »Architektur«?	29
1.1	Das Ziel?	30
1.2	Fallstudie	31
1.2.1	Die Signatur des Chaos	32
1.2.2	Die Perspektive der Unternehmensleitung	33
1.2.3	Was ist schiefgelaufen?	34
1.3	Fazit	37
2	Die Geschichte zweier Werte	39
2.1	Verhalten	39
2.2	Architektur	40
2.3	Der größere Wert	41
2.4	Das Eisenhower-Prinzip	42
2.5	Der Kampf für die Architektur	43
Teil II	Die ersten Bausteine setzen: Programmierparadigmen	45
3	Die Paradigmen	47
3.1	Strukturierte Programmierung	47
3.2	Objektorientierte Programmierung	48
3.3	Funktionale Programmierung	48
3.4	Denkanstöße	49
3.5	Fazit	49

4	Strukturierte Programmierung	51
4.1	Die Beweisführung	52
4.2	Eine »schädliche« Proklamation	54
4.3	Funktionale Dekomposition	55
4.4	Keine formalen Beweise	55
4.5	Wissenschaft als Rettung	55
4.6	Tests	56
4.7	Fazit	57
5	Objektorientierte Programmierung	59
5.1	Datenkapselung [?]	60
5.2	Vererbung [?]	63
5.3	Polymorphie	65
5.3.1	Die Macht der Polymorphie	67
5.3.2	Abhängigkeitsumkehr	68
5.4	Fazit	71
6	Funktionale Programmierung	73
6.1	Quadrierung von Integern	74
6.2	Unveränderbarkeit und Architektur	75
6.3	Unterteilung der Veränderbarkeit	76
6.4	Event Sourcing	77
6.5	Fazit	79
Teil III	Designprinzipien	81
7	SRP: Das Single-Responsibility-Prinzip	85
7.1	Symptom 1: Versehentliche Duplizierung	86
7.2	Symptom 2: Merges	88
7.3	Lösungen	89
7.4	Fazit	90
8	OCP: Das Open-Closed-Prinzip	91
8.1	Ein Gedankenexperiment	92
8.2	Richtungssteuerung	95
8.3	Information Hiding	95
8.4	Fazit	96
9	LSP: Das Liskov'sche Substitutionsprinzip	97
9.1	Gesteuerte Nutzung der Vererbung	98

9.2	Das Quadrat-Rechteck-Problem	98
9.3	Das LSP und die Softwarearchitektur	99
9.4	Beispiel für einen Verstoß gegen das LSP	99
9.5	Fazit	101
10	ISP: Das Interface-Segregation-Prinzip	103
10.1	Das ISP und die Programmiersprachen	104
10.2	Das ISP und die Softwarearchitektur	105
10.3	Fazit	105
11	DIP: Das Dependency-Inversion-Prinzip	107
11.1	Stabile Abstraktionen	108
11.2	Factories	109
11.3	Konkrete Komponenten	110
11.4	Fazit	110
Teil IV	Komponentenprinzipien	111
12	Komponenten	113
12.1	Eine kurze Historie der Komponenten	114
12.2	Relokatierbarkeit	116
12.3	Linker	117
12.4	Fazit	119
13	Komponentenkohäsion	121
13.1	REP: Das Reuse-Release-Equivalence-Prinzip	121
13.2	CCP: Das Common-Closure-Prinzip	123
13.2.1	Ähnlichkeiten mit dem SRP	124
13.3	CRP: Das Common-Reuse-Prinzip	124
13.3.1	Relation zum ISP	125
13.4	Das Spannungsdiagramm für die Komponentenkohäsion	125
13.5	Fazit	127
14	Komponentenkopplung	129
14.1	ADP: Das Acyclic-Dependencies-Prinzip	129
14.1.1	Der wöchentliche Build	130
14.1.2	Abhängigkeitszyklen abschaffen	131
14.1.3	Auswirkung eines Zyklus in einem Komponenten- abhängigkeitsgraphen	133
14.1.4	Den Zyklus durchbrechen	134
14.1.5	Jitters (Fluktuationen)	135

14.2	Top-down-Design	136
14.3	SDP: Das Stable-Dependencies-Prinzip	137
14.3.1	Stabilität.	137
14.3.2	Stabilitätsmetriken	139
14.3.3	Nicht alle Komponenten sollten stabil sein	141
14.3.4	Abstrakte Komponenten	143
14.4	SAP: Das Stable-Abstractions-Prinzip	143
14.4.1	Wo werden die übergeordneten Richtlinien hinterlegt?	143
14.4.2	Einführung in das SAP (Stable-Abstractions-Prinzip)	143
14.4.3	Bemessung der Abstraktion.	144
14.4.4	Die Hauptreihe	144
14.4.5	Die »Zone of Pain«.	145
14.4.6	Die »Zone of Uselessness«	146
14.4.7	Die Ausschlusszonen vermeiden	147
14.4.8	Abstand von der Hauptreihe	147
14.5	Fazit	149
Teil V	Softwarearchitektur	151
15	Was ist Softwarearchitektur?	153
15.1	Entwicklung	155
15.2	Deployment	155
15.3	Betrieb.	156
15.4	Instandhaltung.	157
15.5	Optionen offenhalten.	157
15.6	Geräteunabhängigkeit	159
15.7	Junk Mail	161
15.8	Physische Adressierung	162
15.9	Fazit	163
16	Unabhängigkeit	165
16.1	Use Cases	165
16.2	Betrieb.	166
16.3	Entwicklung	167
16.4	Deployment	167
16.5	Optionen offenhalten.	168
16.6	Layer entkoppeln	168
16.7	Use Cases entkoppeln	169
16.8	Entkopplungsmodi	170

16.9	Unabhängige Entwickelbarkeit.	171
16.10	Unabhängige Deploybarkeit.	171
16.11	Duplizierung.	171
16.12	Entkopplungsmodi (zum Zweiten)	172
16.12.1	Welcher Modus ist am besten geeignet?	173
16.13	Fazit	174
17	Grenzen: Linien ziehen.	175
17.1	Ein paar traurige Geschichten	176
17.2	FitNesse	179
17.3	Welche Grenzen sollten Sie ziehen – und wann?	181
17.4	Wie verhält es sich mit der Ein- und Ausgabe?	184
17.5	Plug-in-Architektur	185
17.6	Das Plug-in-Argument	186
17.7	Fazit	187
18	Anatomie der Grenzen	189
18.1	Grenzüberschreitungen	189
18.2	Der gefürchtete Monolith	190
18.3	Deployment-Komponenten.	192
18.4	Threads.	192
18.5	Lokale Prozesse	193
18.6	Services.	193
18.7	Fazit	194
19	Richtlinien und Ebenen	195
19.1	Ebene	196
19.2	Fazit	199
20	Geschäftsregeln	201
20.1	Entitäten.	202
20.2	Use Cases	203
20.3	Request-and-Response-Modelle	205
20.4	Fazit	206
21	Die schreiende Softwarearchitektur	207
21.1	Das Thema einer Architektur	208
21.2	Der Zweck einer Softwarearchitektur	208
21.3	Aber was ist mit dem Web?	209
21.4	Frameworks sind Tools, keine Lebenseinstellung	209
21.5	Testfähige Architekturen	210
21.6	Fazit	210

22	Die saubere Architektur	211
22.1	Die Abhängigkeitsregel (Dependency Rule)	213
22.1.1	Entitäten	213
22.1.2	Use Cases	213
22.1.3	Schnittstellenadapter	214
22.1.4	Frameworks und Treiber	214
22.1.5	Nur vier Kreise?	215
22.1.6	Grenzen überschreiten	215
22.1.7	Welche Daten überqueren die Grenzlinien?	216
22.2	Ein typisches Beispiel	216
22.3	Fazit	217
23	Presenters und »Humble Objects«	219
23.1	Das Pattern »Humble Object«	219
23.2	Presenters und Views	220
23.3	Das Testen und die Softwarearchitektur	221
23.4	Datenbank-Gateways	221
23.5	Data Mappers	221
23.6	Service Listeners	222
23.7	Fazit	222
24	Partielle Grenzen	223
24.1	Den letzten Schritt weglassen	224
24.2	Eindimensionale Grenzen	224
24.3	Fassaden	225
24.4	Fazit	226
25	Layer und Grenzen	227
25.1	Hunt the Wumpus	228
25.2	Saubere Architektur?	229
25.3	Datenstromüberschreitungen	231
25.4	Datenströme teilen	232
25.5	Fazit	234
26	Die Komponente Main	235
26.1	Das ultimative Detail	235
26.2	Fazit	239
27	Services – große und kleine	241
27.1	Servicearchitektur?	241

27.2	Vorteile der Services?	242
27.2.1	Denkfalle: Entkopplung	242
27.2.2	Denkfalle: Unabhängige Entwickel- und Deploybarkeit	243
27.3	Das Kätzchen-Problem	243
27.4	Objekte als Rettung	245
27.5	Komponentenbasierte Services	247
27.6	Cross-Cutting Concerns	248
27.7	Fazit	248
28	Die Testgrenze	249
28.1	Tests als Systemkomponenten	249
28.2	Design für Testfähigkeit	250
28.3	Die Test-API	251
28.3.1	Strukturelle Kopplung	251
28.3.2	Sicherheit	252
28.4	Fazit	252
29	Saubere eingebettete Architektur	253
29.1	App-Eignungstest	256
29.2	Der Flaschenhals der Zielhardware	259
29.2.1	Eine saubere eingebettete Architektur ist eine testfähige eingebettete Architektur.	260
29.2.2	Offenbaren Sie dem HAL-User keine Hardwaredetails	263
29.3	Fazit	269
Teil VI	Details	271
30	Die Datenbank ist ein Detail	273
30.1	Relationale Datenbanken	274
30.2	Warum sind Datenbanksysteme so weit verbreitet?	274
30.3	Was wäre, wenn es keine Festplatten gäbe?	275
30.4	Details	276
30.5	Und was ist mit der Performance?	276
30.6	Anekdote	277
30.7	Fazit	278
31	Das Web ist ein Detail	279
31.1	Der immerwährende Pendelausschlag	280
31.2	Quintessenz	281
31.3	Fazit	282

32	Ein Framework ist ein Detail	283
32.1	Framework-Autoren	284
32.2	Asymmetrische Ehe	284
32.3	Die Risiken	285
32.4	Die Lösung	285
32.5	Hiermit erkläre ich euch zu ...	286
32.6	Fazit	286
33	Fallstudie: Software für den Verkauf von Videos	287
33.1	Das Produkt	287
33.2	Use-Case-Analyse	288
33.3	Komponentenarchitektur	289
33.4	Abhängigkeitsmanagement	291
33.5	Fazit	291
34	Das fehlende Kapitel	293
34.1	Package by Layer	294
34.2	Package by Feature	295
34.3	Ports and Adapters	297
34.4	Package by Component	299
34.5	Der Teufel steckt in den Implementierungsdetails	304
34.6	Organisation vs. Kapselung	305
34.7	Andere Entkopplungsmodi	308
34.8	Fazit: Der fehlende Ratschlag	309
A	Architekturarchäologie	311
A.1	Das Buchhaltungssystem für die Gewerkschaft	312
A.2	Zurechtschneiden mit dem Laser	317
A.3	Monitoring von Aluminiumspritzguss	321
A.4	4-TEL	322
	A.4.1 Service Area Computer	326
	A.4.2 Ermittlung des Wartungsbedarfs	327
	A.4.3 Architektur	327
	A.4.4 Die große Neugestaltung	329
	A.4.5 Europa	330
	A.4.6 SAC: Fazit	330
A.5	Die Programmiersprache C	331
	A.5.1 C	332
A.6	BOSS	332

A.7	Projekt CCU	333
A.7.1	Denkfalle: Die Planung	334
A.8	DLU/DRU	335
A.8.1	Architektur	336
A.9	VRS	337
A.9.1	Der Name	338
A.9.2	Architektur	338
A.9.3	VRS: Fazit	339
A.10	Der Elektronische Rezeptionist	340
A.10.1	Der Untergang des ER	341
A.11	Craft Dispatch System	342
A.12	Clear Communications	344
A.12.1	Die Gegebenheiten	345
A.12.2	Uncle Bob	346
A.12.3	Das Telefongespräch	346
A.13	ROSE	347
A.13.1	Fortsetzung der Debatten	348
A.13.2	... unter anderem Namen	348
A.14	Prüfung zum eingetragenen Architekten	349
A.15	Fazit	352
B	Nachwort	353
	Stichwortverzeichnis	357

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 wunderbare, unter riesigen Schlammlawinen begrabene archäologische Fundschichten. Dass Softwarestrukturen in der gleichen Art und Weise unserer Intuition folgen, wie dies auch bei Gebäudestrukturen der Fall ist, lässt sich häufig nur schwer erkennen.

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

Da stehen lauter Schildkröten aufeinander.¹

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

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

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

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

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

Das ist das Ungheuerliche in der Liebe, dass der Wille unendlich ist und die Ausführung beschränkt, dass das Verlangen grenzenlos ist und die Tat ein Sklave der Beschränkung.

– William Shakespeare²

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

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

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

– Grady Booch

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

Wenn Sie denken, eine gute Architektur sei teuer, dann probieren Sie es mal mit einer schlechten.

– Brian Foote und Joseph Yoder

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

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

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

– Ralph Johnson

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

An diesem Punkt verzweigt der Weg in viele Richtungen.

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

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

Architektur ist eine Hypothese, die durch Implementierung und Bewertung bewiesen werden muss.

– Tom Gilb

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

Der einzige Weg, um schnell voranzukommen, ist gut voranzukommen.

– Robert C. Martin

Genießen Sie die Reise.

Kevlin Henney, Mai 2017

Einleitung

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

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

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

Dasselbe gilt für Singlethread-Anwendungen, Multithread-Anwendungen, Anwendungen mit wenigen schwergewichtigen Prozessen, Anwendungen mit vielen leichtgewichtigen Prozessen, Multiprozessoranwendungen, Datenbankanwendungen, mathematische Anwendungen, algorithmische Geometrie-Anwendungen 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 Computerprogrammiererin¹ per Zeitreise in das Jahr 2016 befördert, sie an mein MacBook mit der *IntelliJ-IDE* (Integrated Development Environment, integrierte Entwicklungsumgebung) gesetzt und ihr Java gezeigt, hätte sie sicherlich 24 Stunden gebraucht, um sich von dem Schock zu erholen. Aber unmittelbar danach wäre sie bereits in der Lage gewesen, Code zu schreiben – bei genauerer Betrachtung unterscheidet sich Java nämlich gar nicht so sehr von C oder auch von Fortran.

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

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

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

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

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

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

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

Hinweis

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

Über den Autor



Robert C. Martin (auch bekannt als »Uncle Bob«) ist seit 1970 als Softwareprogrammierer tätig. Er ist Mitbegründer der Organisation *cleancoders.com*, die Online-Videotrainings für Softwareentwickler bereitstellt. Des Weiteren gründete er das Unternehmen *Uncle Bob Consulting LLC*, das Dienstleistungen in den Bereichen

IT-Beratung, Training und Skill Development für große Firmen weltweit anbietet. Außerdem war er als Master Craftsman in dem in Chicago ansässigen IT-Consulting-Unternehmen *8th Light Inc.* beschäftigt. Martin veröffentlichte Dutzende von Artikeln in diversen Handelsmagazinen und tritt regelmäßig als Redner bei internationalen Konferenzen und Kongressen auf. Er war drei Jahre lang Chefredakteur des Computermagazins *C++ Report* und ist zudem Erster Vorsitzender der *Agile Alliance*, einer gemeinnützigen Organisation zur Förderung der im *Agile Manifesto* festgelegten Konzepte der Agilen Softwareentwicklung.

Martin hat zahlreiche Bücher geschrieben sowie redaktionell bearbeitet, darunter Titel wie *Clean Code: 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++ Applications Using the Booch Method* (Prentice Hall, 1995).



Danksagung

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

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

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

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

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 Software-designs? 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!

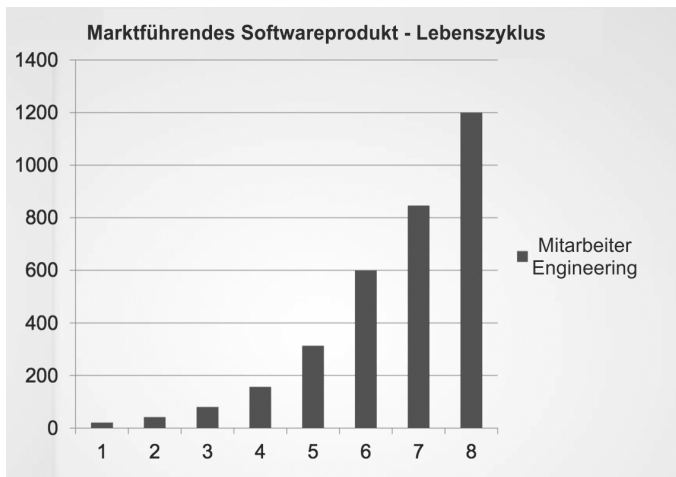


Abb. 1.1: Personalwachstum im Engineering-Bereich
(Abdruck mit freundlicher Genehmigung von Jason Gorman)

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).

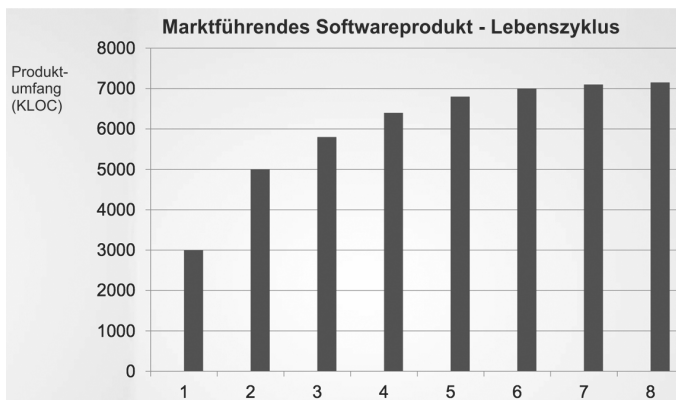


Abb. 1.2: Produktivität über denselben Zeitraum hinweg

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.

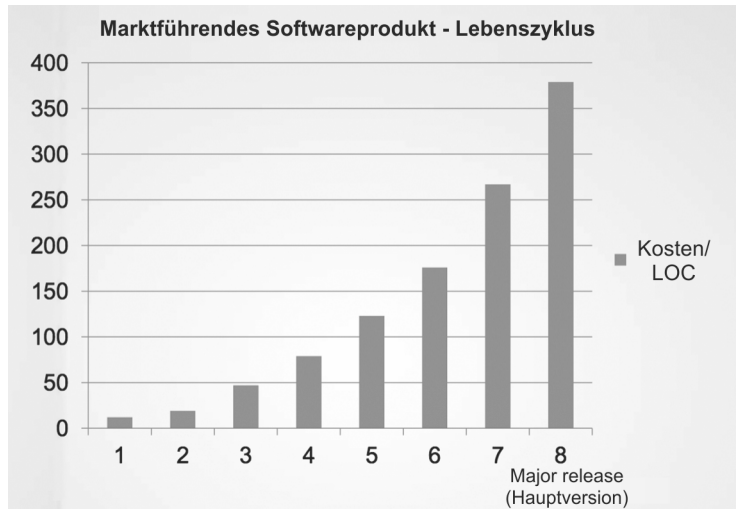


Abb. 1.3: Kosten pro Codezeile im Zeitverlauf

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.

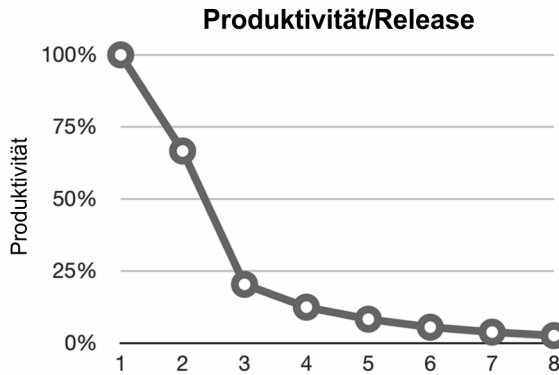


Abb. 1.4: Produktivität je Release

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!

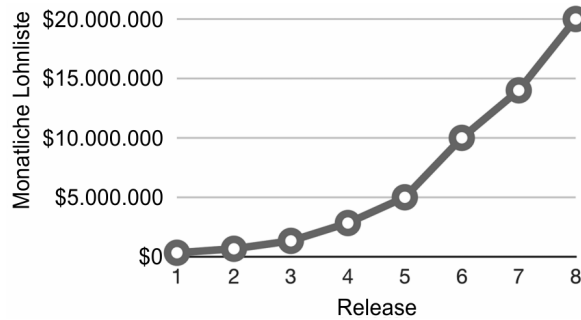


Abb. 1.5: Entwicklung der monatlichen Gehaltszahlungen je Release

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 Dämmer Schlaf – 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 zielstrebigsten asymptotischen Verlauf gegen null.

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

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

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

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

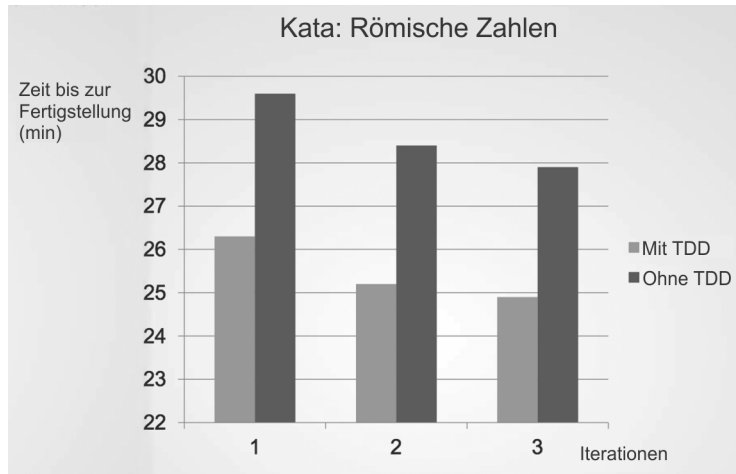


Abb. 1.6: Zeit bis zur Fertigstellung nach Iterationen und Anwendung/Nichtanwendung der TDD-Methode

Auffällig ist in Abbildung 1.6 die Lernkurve: An den späteren Tagen war die Arbeit schneller erledigt als an den vorausgehenden. Beachten Sie auch, dass die Programmierung an den TDD-Tagen um etwa 10 % schneller voranging als an den Tagen ohne Anwendung des TDD-Verfahrens und dass Jason selbst am langsamsten TDD-Tag schneller vorwärtscam 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 Architekturen¹ und Designs aussehen, damit Softwareentwickler Systeme bauen können, die eine lange und gewinnbringende Lebensdauer erzielen.

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

Stichwortverzeichnis

3DBB (Three-Dimensional Black Board) 343
4-TEL 322, 334, 337
8085-Hardware 331
8085-Mikroprozessor 323
8086-Mikrocomputer 329
.jar-Datei
 Download-and-Go-Regel 179
 Entkopplungsmodus auf Quellcode-
 Ebene 192
 Komponenten als .jar-Datei 113
 Komponentenarchitektur 290
 komponentenbasierte Services designen
 247
 Komponentendefinition 303
 partielle Grenze erstellen 224
.NET
 Komponenten als DLLs 113

A

Abhängigkeiten 70
 architektonisches Framework für Richt-
 linien 196
 Auswirkungen auf Software 254
 Beispiel OCP 92
 Common-Reuse-Prinzip 124
 Stabilitätsmetriken berechnen 139
 transitive 95
 unerwünschte verwalten 109
 unverwaltete 254
 von Komponenten verstehen 137
Abhängigkeiten, ADP *siehe* ADP (Acyclic-
 Dependencies-Prinzip)
Abhängigkeitsgraph *siehe* Komponentenab-
 hängigkeitsgraph
Abhängigkeitsmanagement
 Abhängigkeitsregel 291
 für vollwertige architektonische Grenzen
 223
 Vererbungsbeziehungen 291
Abhängigkeitsregel
 Abhängigkeitsmanagement 291
 Beispiel saubere Architektur 217
 Definition 110
 Entitäten 213

 Frameworks 214
 Framework-Verstöße 285
 Grenzen überschreiten 215
 grenzüberschreitende Daten 216
 im Adventurespiel Hunt the Wumpus
 229
 objektorientierter Ansatz für Cross-Cut-
 ting Concerns 248
 Schnittstellenadapter 214
 Services 242
 Services zur Einhaltung der 248
 Tests gemäß der 250
 Treiber 214
 Übersicht 213
 und saubere Architektur 213
 Use Cases 213
Abhängigkeitsumkehr
 objektorientierte Programmierung 68
Abhängigkeitsumkehrung 317
Abhängigkeitsverwaltung
 mittels Polymorphie in monolithischen
 Systemen 190
Abhängigkeitszyklus *siehe* Komponentenab-
 hängigkeitsgraph
Abstract Factories 109
Abstrakte Klassen
 Dependency-Inversion-Prinzip und 107
 Fazit 149
 in Zone of Uselessness übrig gebliebene
 146
 Services in Java 247
 übergeordnete Richtlinien hinterlegen
 143
Abstrakte Komponenten 143
Abstraktionen
 stabile 108
 und Quellcode-Abhängigkeiten 107
ADP (Acyclic-Dependencies-Prinzip)
 Abhängigkeitszyklen abschaffen 131
 Auswirkung eines Zyklus im Kompo-
 nentenabhängigkeitsgraphen 133
 Jitters (Fluktuationen) 135
 Übersicht 129
 wöchentlicher Build 130
 Zyklus durchbrechen 134

Agile Manifesto 23
 Akteure 86
 Aktualisierungen
 nebenläufige 75
 Aluminiumspritzguss 321
 Anpassung
 einfache 40
 Anwendungsfälle *siehe* Use Cases
 Anwendungsspezifische Geschäftsregeln
 Use Cases 204, 213
 APIs
 Tests 251
 App-Eignungstest 256
 Architektur
 als Systemwert 40
 Eisenhower-Prinzip 42
 Microservice- 342
 richtig hinbekommen 27
 serviceorientierte 341
 und ISP 105
 und LSP 99
 und Unveränderbarkeit 75
 vor Funktionalität 41
 vs. Design 29
 vs. Funktionalität 41
 Archive
 als Komponentenaggregation 113
 ASC Tabulating 312
 Asymmetrische Ehe
 mit Framework-Autoren 284
 Ausführbare Dateien
 Deployment von Monolithen 190
 Komponenten verknüpfen als ausführ-
 bare Dateien 113
 Ausgehende Abhängigkeiten
 Stabilitätsmetriken 139
 Ausmaß
 von Systemänderungen 40
 Ausschlusszonen
 Beziehung zwischen Abstraktion und
 Stabilität 145
 vermeiden 147
 Automat
 endlicher 342
 Automatisierte Systeme
 Geschäftsregeln 203
 Autoren
 Framework 284
B
 Basisklassen
 Frameworks 284

BCE(Boundary Control Entity)-Systemarchi-
 tektur 211
 Beck, Kent 256
 Beispiel
 Kätzchen-Problem 243
 Betriebssystem (OS)
 als Detail 266
 Beweisführung 52
 in der strukturierten Programmierung 55
 Bibliotheken
 Adresse des Quellcodes 115
 relokatierbare Binärdateien 117
 Binärdateien
 Relokatierbarkeit 116
 Böhm, Corrado 53
 Booch-Diagramm 347
 BOSS (Basic Operating System and
 Scheduler 332
 Boston Systems Office 332
 BPEL (Business Process Execution
 Language) 343
 Brooks, Fred 256
 Buchhaltungssystem 312
 Build
 wöchentlicher 130

C

C 331, 332
 Datenkapselung 60
 Polymorphie in 65
 Vererbung 63
 C4 Software Architecture Model 304
 C#
 Abhängigkeitsumkehr 69
 abstrakte Komponenten 143
 Anweisungen für Abhängigkeiten ver-
 wenden 196
 geminderte Datenkapselung 62
 C++
 Abhängigkeiten 140
 Ehe mit STL eingehen 286
 geminderte Datenkapselung 61
 Polymorphie in 67
 Vererbung 64
 C-Compiler 332
 CCP (Common-Closure-Prinzip)
 Layer entkoppeln 168
 Modifikationen lokal halten 136
 Richtlinien in Komponenten gruppieren
 198
 Spannungsdiagramm 126
 Übersicht 123

- und SDP (Stable-Dependency-Prinzip) 136
- vs. SRP (Single-Responsibility-Prinzip) 124
- CDS (Craft Dispatch System) 342
- Church, Alonzo 48, 73
- Clean Code 82
- cleancoders.com 23, 287
- Clear Communications 344
- Clojure 74, 77
- Cockburn, Alistair 211
- Codd, Edgar 274
- Code
 - Kostenanstieg der Gehälter 33
 - Produktivität vs. Kosten 31
 - Signatur des Chaos 32
 - Unvernunft der Selbstüberschätzung 34
- Code, Quellcode-Abhängigkeiten *siehe* Quellcode-Abhängigkeiten
- Codeorganisation
 - andere Entkopplungsmodi 308
 - Implementierungsdetails 304
 - Package by Component 299
 - Package by Feature 295
 - Package by Layer 294
 - Ports and Adapters 297
 - Übersicht 293
 - vs. Kapselung 305
- Codezeilen 31, 32
- Common-Closure-Prinzip *siehe* CCP (Common-Closure-Prinzip)
- Common-Reuse-Prinzips *siehe* CRP (Common-Reuse-Prinzip)
- Compare-and-Swap-Algorithmus 77
- Compiler
 - Adresse des Quellcodes 114
 - Architekturprinzipien umsetzen 307
 - relokatierbare Binärdateien 116
- Constantine, Larry 55
- Controller
 - Beispiel saubere Architektur 216
 - Grenzen überschreiten 215
 - saubere Architektur 214
- Conway's Law *siehe* Gesetz von Conway
- Coplien, James 211
- Cross-Cutting Concerns 248
- CRP (Common-Reuse-Prinzip)
 - Einfluss auf die Komponentenzusammenstellung 137
 - Spannungsdiagramm 126
 - Übersicht 124

D

- DAG (Directed Acyclic Graph)
 - architektonisches Framework für Richtlinien 196
 - Definition 132
 - Komponentenzyklus durchbrechen 134
- Dahl, Ole Johan 48
- Data Mappers 221
- Datanet 30 312
- Dateisysteme
 - Verzögerungen verringern 275
- Daten
 - Abhängigkeitsregel 216
 - Beispiel saubere Architektur 216
 - von Funktionen trennen 89
- Daten- und Informationsabfragemodelle *siehe* Request-and-Response-Modelle
- Datenbank
 - Abhängigkeitsregel 214
 - als Option offenhalten 209
 - Beispiel saubere Architektur 216
 - Detail 273
 - Gateways 221
 - gegen Geschäftsregeln abgrenzen 180
 - im Adventurespiel Hunt the Wumpus 228
 - in der Entwicklungsphase als Option offenhalten 158
 - Komponenten abgrenzen 181
 - Layer entkoppeln 169
 - objektorientierte 348
 - Plug-in-Architektur 185
 - relationale 274
 - Schema in der Zone of Pain 146
 - testfähige Architektur erstellen 210
 - unabhängige Entwickelbarkeit 70
 - unabhängige saubere Architektur 212
 - Use Cases entkoppeln 169
- Datenbank als Detail
 - Anekdote 277
 - Bedeutung von Datenbanksystemen 274
 - Details 276
 - Fazit 278
 - ohne Festplatten 275
 - Performance 276
 - relationale Datenbanken 274
 - Übersicht 273
- Datenkapselung
 - objektorientierte Programmierung 60
- Datenmanagement
 - als Aspekt der Softwarearchitektur 49
- Datenmodell
 - vs Datenbank 273

- Datenspeicherung
 - Bedeutung von Datenbanksystemen dank Festplatten 274
- Datenströme
 - teilen 232
 - überschreiten 231
 - und saubere Architektur 231
- DCI(Data Context Interaction)-Systemarchitektur 211
- Deadlocks
 - durch veränderbare Variablen 75
- Deep Thought 329
- DeMarco, Tom 55
- Dependency Rule *siehe* Abhängigkeitsregel
- Dependency-Injection-Framework
 - Main-Komponente 236
- Dependency-Inversion-Prinzip *siehe* DIP (Dependency-Inversion-Prinzip)
- Deployment
 - architekturgestütztes 167
 - Einfluss der Architektur 155
 - Komponenten 192
 - Komponenten als Einheiten 113
- Deployment-Ebene
 - Entkopplungsmodi 173
- Design
 - Flüchtigkeit von Schnittstellen reduzieren 108
 - Produktivität vs. Kosten 30
 - richtig hinbekommen 27
 - Signatur des Chaos 32
 - SOLID-Prinzipien 82
 - vs. Architektur 29
 - Zielsetzung 30
- Design für Testfähigkeit 250
- Design Pattern Humble Object
 - Data Mappers 221
 - Datenbank-Gateways 221
 - Presenters 219
 - Presenters und Views 220
 - Testen und Softwarearchitektur 221
- Design Pattern Strategy
 - objektorientierter Ansatz für Cross-Cutting Concerns 246
 - partielle Grenzen 225
- Design Pattern Template Method
 - objektorientierter Ansatz für Cross-Cutting Concerns 246
- Design vs. Architektur
 - Fazit 37
 - Übersicht 29
- Designprinzipien 81
- Detail
 - Betriebssystem 266
 - Datenbank 273
 - Frameworks 283
 - Hardware 261, 263
 - Prozessor 263
 - Web 279
- Details
 - von Richtlinien trennen 158
- Dijkstra, Edsger Wybe 51
 - Beweisführung in der Softwareprogrammierung 52
 - Entdeckung der strukturierten Programmierung 47
 - Proklamation zu goto-Anweisungen 54
 - Softwaretests 56
- DIP (Dependency-Inversion-Prinzip) 84
 - Entitäten ohne Kenntnis der Use Cases 205
 - Factories 109
 - Fazit 110
 - Grenzen überschreiten 215
 - Grenzlinien ziehen 187
 - in guter Softwarearchitektur 92
 - Komponentenzyklus durchbrechen 134
 - konkrete Komponenten 110
 - nicht alle Komponenten sollten stabil sein 142
 - stabile Abstraktionen 108
 - Übersicht 107
 - und Stable-Abstractions-Prinzip 144
- D-Metrik
 - Abstand von der Hauptreihe 147
- do/while/until-Anweisungen 47
- Don't Repeat Yourself *siehe* DRY-Prinzip (Don't Repeat Yourself)
- Dreischichtige »Architektur«
 - als Topologie 176
- Dringlichkeit
 - Eisenhower-Prinzip 42
- DRY-Prinzip (Don't Repeat Yourself)
 - bedingte Kompilierungsrichtlinien 269
- Duplizierung
 - echte 171
 - echte vs. versehentliche 171
 - versehentliche 86
- Dynamisch typisierte Sprachen
 - und DIP 107
 - und ISP 104
- Dynamisch verknüpfte Libraries
 - als architektonische Grenze 192

E

Ebenen
 und Richtlinien 195
 Echte Duplizierungen 171
 Echtzeitbetriebssystem
 als Detail 266
 Educational Testing Service (ETS) 349
 Ein-/Ausgabe
 Definition der Richtlinienebene als
 Abstand zwischen Ein-/Ausgabe 196
 Geschäftsregeln für Use Cases 205
 Ein-/Ausgabe (I/O)
 Komponenten abgrenzen 184
 Eindimensionale Grenzen 224
 Eingehende Abhängigkeiten
 Stabilitätsmetriken 139
 Eisenhower-Prinzip 42
 Wichtigkeit vs. Dringlichkeit 42
 Elektronischer Rezeptionist 340
 Endlicher Automat 342
 Entitäten
 Abhängigkeitsregel 213
 Beispiel saubere Architektur 216
 Risiken durch Frameworks 285
 testfähige Architektur erstellen 210
 und Geschäftsregeln 202
 vs. Use Cases 202
 Entkopplung
 auf Quellcode-Ebene 190
 Beispiel Kätzchen-Problem 243
 Denkfalle bei Services 242
 Layer 168
 Modi 170, 172
 objektorientierter Ansatz für Cross-Cutting Concerns 245
 Quellcode-Abhängigkeiten 308
 Test-API 251
 unabhängige Deploybarkeit 171
 unabhängige Entwickelbarkeit 171, 242
 unabhängiges Deployment 242
 Use Cases 169
 Entkopplungsmodus
 auf Deployment-Ebene 192
 Entwickler
 Architektur vs. Funktionalität 41
 Eisenhower-Prinzip 43
 Signatur des Chaos 32
 Unvernunft der Selbstüberschätzung 34
 Entwicklung
 architekturgestützte 167
 Einfluss der Architektur 155
 Enumeration
 Beweis für Sequenz und Selektion 53

EPROM-Brenner 324
 Event Sourcing
 Transaktionen speichern 78
 Externe Definition
 Compiler 117
 Externe Komponenten
 unabhängige saubere Architektur 212
 Externe Referenz
 Compiler 117
 Extreme Programming 353

F

Fallstudie Videoverkaufssoftware
 Abhängigkeitsmanagement 291
 Abläufe/Entscheidungen eines guten
 Softwarearchitekten 287
 Fazit 291
 Komponentenarchitektur 289
 Produkt 287
 Übersicht 287
 Use-Case-Analyse 288
 Fan-in/Fan-out-Metriken
 Komponentenstabilität 139
 Feathers, Michael 83
 Fernschreiber 314
 Festplatten
 Aussterben 275
 Bedeutung für Datenbanksysteme 274
 Firewalls
 Grenzüberschreitungen mittels
 Firewalls 189
 Systemschutz durch Plug-in-Architektur
 187
 Firmware
 Auswirkungen veralteter Hardware auf
 Firmware 253
 Definitionen 254
 Flaschenhals der Zielhardware 259
 reduzieren 256
 verschwommene Grenze zwischen Software und Firmware 261
 FitNesse-Programm
 partielle Grenze 224
 Übersicht 179
 FLD (Field Labeled Data) 344
 Flüchtige Komponenten
 Design für Testfähigkeit 251
 flüchtige Software hinterlegen 143
 problematisch in der Zone of Pain 146
 Stable-Dependencies-Prinzip 137
 und Abhängigkeitsgraph 135, 137
 Form
 von Systemänderungen 40
 Fowler, Martin 294

- Fragile Tests Problem 251
- Framework als Detail
 - Fazit 286
 - Framework-Autoren 284
 - Lösung 285
 - Popularität 283
 - Risiken 285
 - Übersicht 283
- Frameworks
 - Abhängigkeitsregel 214
 - als Option offenhalten 209
 - als Tools 209
 - Detail 283
 - feste Bindung 286
 - in Softwarearchitekturen vermeiden 208
 - testfähige Architektur erstellen 210
 - unabhängige saubere Architektur 212
- Freeman, Steve 211
- Funktion
 - Prinzip einer einzigen Aufgabe 85
- Funktionale Dekomposition
 - als Best Practice in der Programmierung 57
 - in der strukturierten Programmierung 55
- Funktionale Programmierung
 - Beispiel Quadrierung von Integern 74
 - Entstehungsgeschichte 48
 - Event Sourcing 77
 - Fazit 79
 - Übersicht 73
 - Unterteilung der Veränderbarkeit 76
 - Unveränderbarkeit 75
- Funktionalität
 - als Aspekt der Softwarearchitektur 49
 - vs. Architektur 41
- Funktionen
 - Beispiele SRP 89
 - gliedern 55
 - konkrete 109
 - nicht überschreiben 109
 - von Daten trennen 89
- Funktionsaufrufe
 - Services als Funktionsaufrufe 241
- Funktionszeiger
 - Entstehungsgeschichte 48
 - in der objektorientierten Programmierung 48
- G**
- Gateways
 - Datenbank 221
- GE Danet 30 312
- Geheimnisprinzip *siehe* Information Hiding
- Geräteunabhängigkeit 68
 - Beispiel 159
 - Beispiel physische Adressierung 162
 - Definition 159
 - I/O-Gerät des UI 282
- Gerichteter azyklischer Graph *siehe* DAG (Directed Acyclic Graph)
- Geschäftsdaten
 - kritische 202
- Geschäftsregeln
 - andocken 185
 - Design für Testfähigkeit 250
 - Entitäten erstellen 202
 - Fazit 206
 - im Adventurespiel Hunt the Wumpus 228
 - Komponenten abgrenzen 181
 - kritische 202
 - Layer entkoppeln 168
 - mit Richtlinien-Anweisungen berechnen 195
 - nah bei den Daten halten 90
 - Request-and-Response-Modelle 205
 - saubere Architektur 212
 - Übersicht 201
 - unabhängige Entwickelbarkeit 70
 - und GUI abgrenzen 184
 - Use Cases 203, 213
 - Use Cases entkoppeln 169
 - von UI entkoppeln 281
- Gesetz von Conway 167
- goto-Anweisungen
 - Dijkstras Proklamation 54
 - durch iterative Kontrollstrukturen ersetzen 53
 - Entfernung aus der strukturierten Programmierung 47
- Grenzanatomie
 - Deployment-Komponenten 192
 - Fazit 194
 - gefürchteter Monolith 190
 - lokale Prozesse 193
 - Services 193
 - Threads 192
 - Überschreitungen 189
 - Übersicht 189
- Grenze 316
- Grenzen
 - Abhängigkeitsregel 215
 - Abhängigkeitsregel für Daten 216
 - Beispiele für architektonische Misserfolge 176

- Ein- und Ausgabe (I/O) 184
 - Fazit 187
 - lokale Prozesse 193
 - partiell und eindimensional 224
 - partielle 223
 - Plug-in-Architektur 185
 - Plug-in-Argument 186
 - Services als grenzüberschreitende Funktionsaufrufe 242
 - Services in Komponenten unterteilen 248
 - Tests 249
 - Übersicht 175
 - wo und wann ziehen 181
 - Grenzüberschreitung
 - Beispiel saubere Architektur 217
 - saubere Architektur 215
 - Growing Object-Oriented Software with Tests (Freeman & Pryce) 211
 - GUI (Graphical User Interface)
 - Design für Testfähigkeit 251
 - Ein-/Ausgabe und Grenzlinien 184
 - Plug-in-Architektur 185
 - Plug-in-Argument 187
 - Unit-Test 220
 - von Geschäftsregeln abgrenzen 181
 - von Geschäftsregeln entkoppeln 281
 - Web als Detail 281
 - GUI (Graphical User Interface) *siehe auch* UI (User Interface)
- H**
- HAL (Hardware Abstraction Layer)
 - Betriebssystem als Detail 266
 - DRY-bedingte Kompilierungsrichtlinien 269
 - Flaschenhals der Zielhardware mit Layern vermeiden 262
 - Grenze zwischen Software und Firmware 262
 - Hardwaredetails verbergen 263
 - veraltete Firmware 262
 - Hardware Abstraction Layer *siehe* HAL (Hardware Abstraction Layer)
 - Hardware-Abstraktionsschicht *siehe* HAL (Hardware Abstraction Layer)
 - Hauptreihe
 - Abstand messen 147
 - Ausschlusszonen vermeiden 147
 - Beziehung zwischen Abstraktion und Stabilität 144
 - Zone of Pain 144
 - Zone of Uselessness 146
 - Hawking, Stephen W. 15
 - Header-Dateien
 - Schnittstellen programmieren mit Header-Dateien 263
 - Hexagonale Architektur (Ports and Adapters) 211
 - Höhlenerkundung *siehe* Spelunking
 - Hunt the Wumpus
 - Main-Komponente 236
 - Hunt, Andrew 269
- I**
- I/O-Gerät
 - UNIX-Funktionen 66
 - Web 282
 - IBM 2314 316
 - IBM System/7 321
 - IBM-3270-Display 321
 - IBM-370-System 321
 - if/then/else-Anweisungen 47
 - Induktionsverfahren
 - Beweis für Iteration 53
 - Information Hiding
 - Open-Closed-Prinzip 95
 - Instandhaltung
 - Einfluss der Architektur 157
 - Integer
 - Beispiel funktionale Programmierung 74
 - Integration
 - Problemstellung beim wöchentlichen Build 130
 - Interfaces *siehe* Schnittstellen
 - Interface-Segregation-Prinzip *siehe* ISP (Interface-Segregation-Prinzip)
 - Isolation
 - Tests 250
 - ISP (Interface-Segregation-Prinzip) 84
 - Fazit 105
 - Übersicht 103
 - und Architektur 105
 - und Sprachtypen 104
 - vs. CRP (Common-Reuse-Prinzip) 125
 - Issue-Tracking-System 26
 - Iteration 53
 - als Programmkontrollstruktur 53
- J**
- Jacobson, Ivar 208, 211
 - Jacopini, Giuseppe 53
 - Java
 - abstrakte Komponenten 143
 - Beispiel ISP 104

- Beispiel Quadrierung von Integern 74
 - Bindung an Standard-Library-Framework 286
 - geminderte Datenkapselung 62
 - import-Anweisungen für Abhängigkeiten 196
 - Komponenten als .jar-Dateien in 113
 - Modul-Frameworks 308
 - Package by Layer 294
 - und DIP 107
- Jitters (Fluktuationen)
 - Komponentenzyklus durchbrechen 135
- JSON 344
- Junk Mail
 - Beispiel 161
- K**
- Kabeldefekt 337
- Kapselung *siehe auch* Datenkapselung
 - übermäßige Verwendung von public und Kapselung 305
 - vs. Codeorganisation 305
- Kerncode
 - Frameworks vermeiden 286
- Klasse
 - Definition 82
- Klassen
 - Beispiele SRP 89
 - Common-Reuse-Prinzip 124
 - Gliederung von Prozessen in 93
 - LSP für gesteuerte Nutzung der Vererbung 98
 - Reuse-Release-Equivalence-Prinzip 122 und DIP 107
- Klassen, abstrakte *siehe* Abstrakte Klassen
- Kohäsion
 - Single-Responsibility-Prinzip 86
- Kommunikation
 - als Funktionsaufrufe zwischen Komponenten in Monolithen 194
 - Gesetz von Conway 167
 - grenzüberschreitend, Deployment-Komponenten 193
 - grenzüberschreitend, lokale Prozesse 193
 - grenzüberschreitend, Services 194
 - in verschiedenen Entkopplungsmodi 173
 - über entkoppelte Grenzen auf Quellcode-Ebene 192
- Kompilierte Programmiersprachen 113
- Komponenten
 - abstrakte 109
 - Deployment 192
 - Entstehungsgeschichte 114
 - Fazit 119
 - konkrete 109, 110
 - Linker 117
 - Prinzipien 111
 - Prozesse in Klassen gliedern 93
 - Relokatierbarkeit 116
 - Tests 249
 - Übersicht 113
- Komponentenabhängigkeitsgraph
 - Auswirkung eines Zyklus 133
 - wieder als DAG einsetzen 134
- Komponentenarchitektur
 - Fallstudie Videoverkaufssoftware 289
- Komponentenbasierte Systeme
 - Funktionsaufrufe 242
 - objektorientierter Ansatz für Cross-Cutting Concerns 246
 - Services mit SOLID-Prinzipien designen 247
 - skalierbar errichten 245
- Komponentenkohäsion
 - Common-Closure-Prinzip 123
 - Common-Reuse-Prinzip 124
 - Fazit 127
 - Reuse-Release-Equivalence-Prinzip 121
 - Spannungsdiagramm 125
 - Übersicht 121
- Komponentenkopplung
 - Fazit 149
 - Fragile Tests Problem 251
 - Stable-Dependencies-Prinzip 137
 - Top-down-Design 136
 - Übersicht 129
- Komponentenkopplung, ADP *siehe auch* ADP (Acyclic-Dependencies-Prinzip)
- Komponentenstabilität
 - bemessen 139
 - und Abstraktion 144
 - verstehen 137
- Komponententeilung
 - als Aspekt der Softwarearchitektur 49
- Konkrete Komponenten
 - Dependency-Inversion-Prinzip 110
- Kontrollfluss 68
 - Abhängigkeitsmanagement 291
 - dynamische Polymorphie 190
 - Grenzen überschreiten 215
- Kontrollstrukturen
 - in Programmen 53
- Kontrollübertragung
 - direkte 47
 - indirekte 48

Kopplung

- durch Framework vermeiden 284
- verfrühte Entscheidungen 175

Kopplung *siehe* Komponentenkopplung

Kritische Geschäftsdaten 202

Kritische Geschäftsregeln 202

Kupferleitungen 334

L

λ -Kalkül *siehe* Lambda-Kalkül

Lambda-Kalkül 48, 73

Laser 318

Layer

- Ansatz für die Codeorganisation 294
- Duplizierung 171
- entkoppeln 168
- Flaschenhals der Zielhardware vermeiden 262
- saubere Architektur 211
- unabhängige Entwickelbarkeit 171

Layer und Grenzen

- Datenströme teilen 232
- Datenstromüberschreitungen 231
- Fazit 234
- saubere Architektur 229
- Übersicht 227

Lebenszyklus

- architekturgestützter 154

Leiningen-Tool

- Modulmanagement 121

Lines of Code *siehe* Codezeilen

Linker

- von Loader separieren 118

Liskov, Barbara 97

Liskov'sches Substitutionsprinzip *siehe* LSP
(Liskov'sches Substitutionsprinzip)

LISP

- Beispiel Quadrierung von Integern 74
- funktionale Programmierung 48

Loader

- relokatierbare Binärdateien 117
- verknüpfen 117

Locks 75

LSP (Liskov'sches Substitutionsprinzip) 84

- Fazit 101
- gesteuerte Nutzung der Vererbung 98
- Quadrat-Rechteck-Problem 98
- Übersicht 97
- und Architektur 99
- Verstoß 99

M

M365 318

Mailboxes

- Kommunikation zwischen lokalen Prozessen 193

Main-Komponente

- als ultimatives Detail 235
- Fazit 239
- geringfügige Releasefolgen 132
- Übersicht 235

main-Komponente

- als konkrete Komponente 110
- objektorientierte Programmierung 64
- Polymorphie 68

Management

- Architektur vs. Funktionalität 41
- Eisenhower-Prinzip 43

Marketingkampagnen

- Datenbankanbieter 278

Mathematik

- Beweisführung 52
- vs. Wissenschaft 55

Maven-Tool

- Modulmanagement 121

McCarthy, John 48

Merges

- Beispiele SRP 88

Metriken

- Abstand von der Hauptreihe 147
- Abstraktion 144

Metriken für das Abhängigkeitsmanagement

siehe ADP (Acyclic-Dependencies-Prinzip)

Meyer, Bertrand 84, 91

Microservice-Architektur 342

- Beliebtheit 241
- Deployment-Strategie 156
- Entkopplungsmodi 170

Minecraft 119

Modem 328

Modul

- Definition 85

Module

- Common-Reuse-Prinzip 124
- public vs. published 308
- Reuse-Release-Equivalence-Prinzip 122

Monolithen

- Deployment 190
- Funktionsaufrufe 242
- lokale Prozesse als statisch verknüpfte 193
- skalierbare Systeme errichten 243
- Threads 192
- vs. Deployment-Ebene 193

Moore'sches Gesetz 118
 Morning-After-Syndrom
 durch die Abschaffung von Abhängig-
 keitszyklen vermeiden 131
 Problemstellung beim wöchentlichen
 Build 130
 Übersicht 129
 zu vermeidende Abhängigkeiten verwal-
 ten 131
 MP/M-86 341

N

National Council of Architects Registry Board
 (NCARB) 349
 Nebenläufige Aktualisierungen 75
 Netnews 345
 Nygaard, Kristen 48

O

Object-Oriented Software Engineering
 (Jacobson) 208, 211
 Object-Relational Mapping *siehe* ORM
 Objektorientierte Datenbank 348
 Objektorientierte Programmierung
 Abhängigkeitsumkehr 68
 Datenkapselung 60
 Deployment von Monolithen 190
 Entstehungsgeschichte 48
 Fazit 71
 für Cross-Cutting Concerns 245
 Hauptkomponenten 60
 Macht der Polymorphie 67
 Polymorphie 65
 Übersicht 59
 Vererbung 63
 Vererbungsbeziehungen 70
 Objektrelationale Abbildung *siehe* ORM
 OCP (Open-Closed-Prinzip) 84, 343
 Abhängigkeitsmanagement 291
 Entstehungsgeschichte 160
 Fazit 96
 Gedankenexperiment 92
 Information Hiding 95
 komponentenbasierte Services designen
 247
 Richtungssteuerung 95
 Übersicht 91
 vs. CCP (Common-Closure-Prinzip) 123
 Open-Closed-Prinzip *siehe* OCR (Open-Clo-
 sed-Prinzip)
 Operating System Abstraction Layer (OSAL)
 saubere eingebettete Architektur 267
 Optionen

 durch Entkopplungsmodi 170
 operationsfördernde Architektur 166,
 168
 zur Erleichterung von Systemänderun-
 gen offenhalten 168
 Zweck einer Softwarearchitektur 154,
 208

ORM (Object-Relational Mapping) 221
 OS (Betriebssystem)
 als Detail 266
 OSAL (Operating System Abstraction Layer)
 saubere eingebettete Architektur 267
 Outboard Marine Corporation 321
 Overlay-System 316

P

Package by Feature 295
 Package by Layer
 horizontale Schichtenarchitektur 294
 Zugriffsmodifikatoren 304
 Packages
 Organisation vs. Kapselung 305
 Page-Jones, Meilir 55
 Partielle Grenzen
 eindimensional 224
 Fassaden 225
 Fazit 226
 implementieren 223
 letzten Schritt weglassen 224
 Übersicht 223
 PDP-11/60 331
 PDP-8 318
 Pendelstationen
 Web als eine von vielen Pendelstationen
 279
 Performance
 tiefschichtiger Mechanismus 276
 Périphérique-Ports-and-Adapters-Anti-Pat-
 tern 309
 Personelle Ressourcen
 Minimierung durch Softwarearchitektur
 175
 Physische Adressierung
 Beispiel 162
 Plug-in-Architektur
 Annahme einer Plug-in-Architektur 185
 für Geräteunabhängigkeit 68
 Grenzen entlang der Modifikationsachse
 ziehen 187
 Main-Komponente 239
 von untergeordneten Komponenten in
 übergeordneten Komponenten 198
 Policy *siehe* Richtlinien

- Policy *siehe* Übergeordnete Richtlinien
 - Polymorphie
 - Abhängigkeitsumkehr 68
 - dynamische 190
 - Grenzen überschreiten 215
 - in der objektorientierten Programmierung 48
 - Kontrollfluss in dynamischer Polymorphie 190
 - Macht der Polymorphie 67
 - objektorientierte Programmierung 65
 - statische vs. dynamische 190
 - Ports and Adapters
 - Abhängigkeiten mit Quellcode-Bäumen entkoppeln 308
 - Périphérique-Ports-and-Adapters-Anti-Pattern 309
 - Zugriffsmodifikatoren 306
 - Positionale Stabilität
 - Komponente 139
 - Presentation Domain Data Layering (Fowler) 294
 - Presenters
 - Beispiel saubere Architektur 216
 - Grenzen überschreiten 215
 - Komponentenarchitektur 290
 - saubere Architektur 214
 - Presenters und Humble Objects
 - Data Mappers 221
 - Datenbank-Gateways 221
 - Design Pattern Humble Object 219
 - Fazit 222
 - Presenters und Views 220
 - Service Listeners 222
 - Testen und Softwarearchitektur 221
 - Übersicht 219
 - Problematik fragiler Tests *siehe* Fragile Tests Problem
 - Produkt
 - Fallstudie Videoverkaufssoftware 287
 - Produktivität
 - Signatur des Chaos 32
 - vs. Kosten 31
 - Programmierparadigmen
 - Entstehungsgeschichte 45
 - Fazit 49
 - Übersicht 47
 - Programmierparadigmen *siehe auch* Funktionale Programmierung
 - Programmierparadigmen, funktionale Programmierung *siehe* Funktionale Programmierung
 - Programmierparadigmen, objektorientierte Programmierung *siehe* Objektorientierte Programmierung
 - Programmierparadigmen, strukturierte Programmierung *siehe* Strukturierte Programmierung
 - Programmiersprachen
 - dynamisch typisierte 107
 - Komponenten 113
 - statisch typisierte 107
 - und abstrakte Komponenten 143
 - und ISP 104
 - Variablen in funktionalen Sprachen 75
 - Proxies
 - mit Frameworks nutzen 286
 - Prozesse
 - Gliederung in Klassen 93
 - Prozessor
 - als Detail 263
 - und Unveränderbarkeit 75
 - Prüfung zum eingetragenen Architekten 350
 - Pryce, Nat 211
 - public
 - übermäßige Verwendung 304
 - vs. published 308
 - published
 - vs. public 308
 - Python
 - und DIP 107
 - und ISP 104
- Q**
- Quadrat-Rechteck-Problem
 - LSP 98
 - Quellcode
 - Kompilierung 115
 - Quellcode-Abhängigkeiten
 - Abhängigkeitsumkehr 68
 - Beispiel OCP 93
 - entkoppeln 197
 - Entkopplung 308
 - Grenzen überschreiten 215
 - Grenzüberschreitungen mittels 189
 - lokale Prozesse als 193
 - nur auf Abstraktionen beziehen 107
 - Wiederverwendung von Spielregeln durch UI-Komponenten 228
 - Quellcode-Bäume
 - Abhängigkeiten entkoppeln 308
 - Quellcode-Ebene
 - Entkopplungsmodi 172

R

- Race Conditions
 - durch veränderbare Variablen 75
 - Schutz gegen nebenläufige Aktualisierungen 76
- RAM
 - als Festplattenersatz 275
- RAM-Speicher 77
- RDBMS (Relational Database Management Systems) 275
- Real-Time Operating System (RTOS) *siehe* Echtzeitbetriebssystem
- Reenskaug, Trygve 211
- Relational Database Management Systems *siehe* RDBMS
- Relationale Datenbankverwaltungssysteme *siehe* RDBMS
- Relaxed Layered Architecture 300
- Releases
 - Abhängigkeitszyklen abschaffen 131
 - Auswirkung eines Zyklus im Komponentenabhängigkeitsgraphen 133
 - neue Komponenten nummerieren 131
 - Reuse-Release-Equivalence-Prinzip für neue Releases 122
- REP (Reuse-Release-Equivalence-Prinzip) Übersicht 121
- Request-and-Response-Modelle
 - Geschäftsregeln 205
- ReSharper
 - Plug-in-Argument 186
- REST-Schnittstelle
 - Entwicklungsoptionen offenhalten 158
- Rezeptionist
 - Elektronischer 340
- Richtlinien
 - Datenströme teilen 232
 - Fazit 199
 - in sauberer Architektur 213
 - in Softwaresystemen abbilden 195
 - Übersicht 195
- Richtlinien *siehe auch* Übergeordnete Richtlinien
- Richtungssteuerung
 - Open-Closed-Prinzip 95
- Risiken
 - Kosten minimieren 157
 - von Frameworks 285
- ROSE 347
- RTOS (Real-Time Operating System) *siehe* Echtzeitbetriebssystem

Ruby

- Komponenten als .gem-Dateien 113
- und DIP 107
- und ISP 104

RVM-Tool

- Modulmanagement 121

S

- SAP (Stable-Abstractions-Prinzip) 143
 - Abstand von der Hauptreihe 147
 - Abstraktionsmetriken 144
 - Ausschlusszonen vermeiden 147
 - Einführung 143
 - Grenzlinien ziehen 187
 - Hauptreihe 144
 - übergeordnete Richtlinien hinterlegen 143
- Saubere Architektur
 - Layer und Grenzen 229
- Saubere eingebettete Architektur
 - App-Eignungstest 256
 - Betriebssystem als Detail 266
 - DRY-bedingte Kompilierungsrichtlinien 269
 - Fazit 269
 - Flaschenhals der Zielhardware 259
 - Hardware als Detail 261
 - keine Hardwaredetails für HAL-User 263
 - Layer 260
 - Programmierung für Schnittstellen und Substituierbarkeit 268
 - testfähige eingebettete Architektur 260
 - Übersicht 253
- Sauberer Code *siehe* Clean Code
- Schichtenarchitektur
 - Gründe gegen 299
 - Package by Layer-Codeorganisation 294
- Schichtenarchitektur, lässige *siehe* Relaxed Layered Architecture
- Schmidt, Doug 253
- Schnittstellen
 - Anwendung des LSPs 99
 - Interface-Segregation-Prinzip 104
- Schnittstellenadapter
 - Abhängigkeitsregel 214
- Schutzhierarchie
 - auf Ebenengrundlage 95
- SDP (Stable-Abstractions-Prinzip) und Stable-Abstractions-Prinzip 144
- SDP (Stable-Dependencies-Prinzip)
 - Acyclic-Dependencies-Prinzip 137

- nicht alle Komponenten sollten stabil sein 141
- Stabilität 137
- Stabilitätsmetriken 139
- und abstrakte Komponenten 143
- SDP (Stable-Dependency-Prinzip und CCP (Common-Closure-Prinzip) 136
- Selbstüberschätzung
 - Unvernunft der Selbstüberschätzung 34
- Selektion
 - als Programmkontrollstruktur 53
- Sequenz
 - als Programmkontrollstruktur 53
- Service-Ebene
 - Entkopplungsmodi 173
- Serviceorientierte Architektur *siehe* SOA (Service-Oriented Architecture)
- Services
 - als Funktionsaufrufe vs. Architektur 241
 - als strikteste Grenze 193
 - Cross-Cutting Concerns 248
 - Denkfalle Entkopplung 242
 - Denkfalle unabhängige Entwickel-/Deploybarkeit 243
 - Fazit 248
 - Humble-Object-Grenzen 222
 - Kätzchen-Problem 243
 - komponentenbasierte 247
 - Objekte als Rettung 245
 - Übersicht 241
 - Vorteile 242
- Sicherheit
 - Test-API 251
- Single-Responsibility-Prinzip *siehe* SRP (Single-Responsibility-Prinzip)
- Skalierbarkeit
 - Kätzchen-Problem und Services 243
 - Services nicht die einzige Option 243
- SOA (Service-Oriented Architecture)
 - Beliebtheit 241
 - Entkopplungsmodi 170
- Sockets
 - Kommunikation zwischen lokalen Prozessen 193
- Software
 - Flaschenhals der Zielhardware vermeiden 261
 - Isolierung vom OS durch saubere eingebettete Architektur 267
 - richtig hinbekommen 26
 - SOLID-Prinzipien 83
 - verschwommene Grenze zwischen Firmware und Software 261
- Softwarearchitekten
 - Zielsetzung der personellen Ressourcenminimierung 175
- Softwarearchitektur
 - Beispiel Junk Mail 161
 - Beispiel physische Adressierung 162
 - Definition 153
 - Deployment 155
 - Details von Richtlinien trennen 158
 - drei Hauptaspekte 49
 - Entwicklung 155
 - Fazit 163
 - Geräteunabhängigkeit 159
 - hexagonale 211
 - Optionen offenhalten 157
 - Plug-in- 185
 - Stabilität 143
 - Systembetrieb 156
 - Testen 221
 - Unabhängigkeit 165
 - zur Unterstützung des Systems 154
- Softwarearchitektur, saubere
 - Abhängigkeitsregel 213
 - Fazit 217
 - Framework-Verstöße 285
 - typisches Beispiel 216
 - Übersicht 211
- Softwarearchitektur, schreiende
 - Fazit 210
 - Frameworks als Tools 209
 - testfähige Architekturen 210
 - übergeordnetes Thema 208
 - Übersicht 207
 - Web 209
 - Zweck 208
- Softwareentität
 - OCP 91
- Softwareentwickler
 - als Stakeholder 43
- Softwareentwicklung
 - Architektur vor Funktionalität 43
 - Ausmaß vs. Form von Systemänderungen 40
 - Tests 250
 - wissenschaftliche Beweisführung 56
- Softwarewiederverwendung
 - Common-Reuse-Prinzip 123
 - Reuse-Release-Equivalence-Prinzip 122
 - und wiederverwendbare Komponenten 122

- SOLID-Prinzipien
 - Entstehungsgeschichte 82
 - komponentenbasierte Services designen 247
 - objektorientierter Ansatz für Cross-Cutting Concerns 245
- SOLID-Prinzipien, Dependency-Inversion-Prinzip *siehe* DIP (Dependency-Inversion-Prinzip)
- SOLID-Prinzipien, Interface-Segregation-Prinzip *siehe* ISP (Interface-Segregation-Prinzip)
- SOLID-Prinzipien, Liskov'sches Substitutionsprinzip *siehe* LSP (Liskov'sches Substitutionsprinzip)
- SOLID-Prinzipien, Open-Closed-Prinzip *siehe* OCP (Open-Closed-Prinzip)
- SOLID-Prinzipien, Single-Responsibility-Prinzip *siehe* SRP (Single-Responsibility-Prinzip)
- Sparcstation 344
- Speicher
 - und lokale Prozesse 193
- Speicher *siehe* RAM-Speicher
- Speicherreferenzadressen
 - relokatierbare Binärdateien 116
- Speicherzuordnung
 - frühes Layout 115
- Spelunking
 - Kosten minimieren 157
- Sperren *siehe* Locks
- Sprachausgabe 337
- Sprachen
 - im Adventurespiel Hunt the Wumpus 228
 - und saubere Architektur 229
- Sprachverarbeitungskarte 340
- SQL (Structured Query Language) 339
- SRP (Single Responsibility-Prinzip)
 - Modifikationen lokal halten 136
- SRP (Single-Responsibility-Prinzip) 83, 86
 - Abhängigkeitsmanagement 288, 291
 - Fazit 90
 - Grenzen ziehen 187
 - in guter Softwarearchitektur 92
 - Layer entkoppeln 168
 - Lösungen 89
 - Merges 88
 - Richtlinien in Komponenten gruppieren 198
 - Übersicht 85
 - Use-Case-Analyse 288
- Stabile Komponenten
 - abstrakte Komponenten als 143
 - harmlos in der Zone of Pain 146
 - nicht alle Komponenten sollten stabil sein 141
- Stable-Abstractions-Prinzip 143
 - übergeordnete Richtlinien hinterlegen 143
- Stakeholder
 - Architektur vor Funktionalität 43
 - Ausmaß vs. Form von Systemänderungen 40
 - Systemwerte für Stakeholder 39
- Statische Analysetools
 - Architekturverstöße 302
- Störfallmeldung 342
- Struktur
 - als Systemwert 39
- Struktur *siehe* Architektur
- Strukturelle Kopplung
 - Test-API 251
- Strukturierte Programmierung 47
 - Beweisführung 55
 - Dijkstras Proklamation zu goto-Anweisungen 54
 - Entstehungsgeschichte 47
 - Fazit 57
 - funktionale Dekomposition 55
 - Rolle der Wissenschaft 55
 - Softwaretests 56
 - Übersicht 51
 - Wert 57
- Substitution
 - Programmierung für Schnittstellen und 268
- Substitution *siehe* LSP (Liskov'sches Substitutionsprinzip)
- Subtypen
 - Definition 97
- Sun Microsystems 344
- Systembetrieb
 - architekturgestützter Systembetrieb 156, 166
 - Auswirkungen von Modifikationen auf Use Cases 214
 - Use Cases entkoppeln 170
- Systemverhalten 154
- Systemwerte
 - Architektur 40
 - Architektur vor Funktionalität 43
 - Eisenhower-Prinzip 42
 - Übersicht 39
 - Verhalten 39

T

T1-Leitung 344
 TDD *siehe* Testgetriebene Entwicklung
 Telefonsysteme
 Europa und USA 330
 Teradyne Applied Systems 317
 Teradyne M365 318
 Terminal mit Kathodenstrahlröhre 314
 Test-Driven Development *siehe* Testgetriebene Entwicklung
 Testen
 mit dem Design Pattern Humble Object 219
 Presenters und Views 220
 und Softwarearchitektur 221
 Testfähige Architektur
 saubere eingebettete Architektur als 260
 Übersicht 210
 Testfähige Softwarearchitektur
 saubere Architektur erstellen 212
 Testgetriebene Entwicklung 36
 Testgrenze
 als Systemkomponente 249
 Design für Testfähigkeit 250
 Fazit 252
 Fragile Tests Problem 251
 Test-API 251
 Übersicht 249
 Tests
 in der strukturierten Programmierung 56
 Thomas, David 269
 Threads
 Ausführungszeitplan/-reihenfolge 192
 und Unveränderbarkeit 75
 Ticket 342
 Ticketsystem 342
 Top-down-Design
 Komponentenstruktur 136
 Tramp-Daten 205
 Transaktionaler Speicher 76
 Transaktionen
 speichern 78
 Transitive Abhängigkeiten
 Verstoß gegen Softwareprinzipien 96
 Treiber
 Abhängigkeitsregel 214
 Turing, Alan 46, 48

U

Übergeordnete Richtlinien
 Datenströme teilen 232
 hinterlegen 143

 von Details trennen 157
 von untergeordneten Ein-/Ausgabe-Richtlinien entkoppeln 198
 UI (User Interface)
 Flüchtigkeit reduzieren 108
 im Adventurespiel Hunt the Wumpus 228
 Programmierung für UI 268
 unabhängige Entwickelbarkeit 70
 unabhängige saubere Architektur 212
 Use Cases entkoppeln 168
 von Geschäftsregeln entkoppeln 281
 UI (User Interface) *siehe auch* GUI (Graphical User Interface)
 UML (Unified Modeling Language) 347
 UML-Klassendiagramm
 Ports and Adapters 299
 Relaxed Layered Architecture 300
 Unabhängige Deploybarkeit
 Beispiel Kätzchen-Problem 243
 Denkfalle bei Services 243
 objektorientierter Ansatz für Cross-Cutting Concerns 245
 Übersicht 171
 Unabhängige Entwickelbarkeit
 Beispiel Kätzchen-Problem 243
 Denkfalle bei Services 243
 objektorientierter Ansatz für Cross-Cutting Concerns 245
 Übersicht 171
 UI und Datenbank 70
 Unabhängige Komponenten
 Stabilitätsmetriken berechnen 139
 verstehen 137
 Unabhängiges Deployment
 Tests 250
 Unabhängigkeit
 Deployment 167
 Duplizierung 171
 Entkopplungsmodi 170, 172
 Entwickelbarkeit 171
 Entwicklung 167
 Fazit 174
 Layer entkoppeln 168
 Optionen offenhalten 168
 Systembetrieb 166
 Übersicht 165
 unabhängige Deploybarkeit 171
 Use Cases 165
 Use Cases entkoppeln 169
 Uncle Bob 346
 Unerwünschte Werbung *siehe* Junk Mail

Unit-Test

- Auswirkung von Zyklen im Komponentenabhängigkeitsgraphen 134
- mit dem Design Pattern Humble Object 219
- testfähige Architektur erstellen 210

UNIX

- I/O-Gerätetreiberfunktionen 66

Unveränderbarkeit 75

Upgrades

- Risiken durch Frameworks 285

Use Cases 203

- Abhängigkeitsregel 213
- als Grundlage für gute Softwarearchitektur 208
- architekturgestützte 165
- Duplizierung 171
- Entkopplungsmodi 169, 170
- Fallstudie Videoverkaufssoftware 288
- Geschäftsregeln 203
- Grenzen überschreiten 215
- Kopplung an verfrühte Entscheidungen 175
- testfähige Architektur erstellen 210
- und unabhängige Entwickelbarkeit 171

User Interface *siehe* GUI (Graphical User Interface)

Utility Library

- Zone of Pain 146

uucp-Verbindung 345

V

van Wijngaarden, Adriaan 52

Variablen

- funktionale Sprachen 75
- veränderbare 75, 76

Varian 620/f 315

Veränderbare Variablen 75, 76

Vererbung

- objektorientierte Programmierung 63

Vererbungsbeziehung

- Abhängigkeitsmanagement 291
- Abhängigkeitsumkehr 70
- gesteuerte Nutzung 98
- Grenzen überschreiten 215
- objektorientierte Programmierung 70

Verfrühte Entscheidungen

- Kopplung 175

Verhalten

- als Systemwert 39
- Architektur vor Funktionalität 43
- Architektur vs. Funktionalität 41

Eisenhower-Prinzip 42

- Optionen offenhalten 157

Verklemmungen *siehe* Deadlocks

Versehentliche Duplizierung 86, 171

View Model

- Presenters und Views 220

Views

- Komponentenarchitektur 290
- saubere Architektur 214
- und Presenters 220

Visual Studio

- Plug-in-Argument 186

Von-Neumann-Architekturen 67

VT100-Terminal 331

W

Wartungsbedarf 327

Web

- Abhängigkeitsregel 214
- als Bereitstellungssystem für die Anwendung 209
- Detail 279
- Web als Detail
- Fazit 282
- GUI (Graphical User Interface) 281
- Übersicht 279

Webserver

- als Option offenhalten 209
- eigenen schreiben 179
- in der Entwicklungsphase als Option offenhalten 158
- testfähige Architektur erstellen 210

Wettlaufsituationen *siehe* Race Conditions

Whitesmiths 332

Wichtigkeit

- Eisenhower-Prinzip 42

Wiederverwendbares Framework 351

Wiederverwendbarkeit *siehe* CRP (Common-Reuse-Prinzip)

Wiederverwendbarkeit von Code 350

Wiki-Text

- architektonische Erfolgsgeschichte 179

Wissenschaftliche Methode

- Anweisungen widerlegen 55

Wöchentlicher Build 130

X

X Window System 344

XML 344

Y

Yourdon, Ed 55

Z

- Zeiger 343
 - zur Erzeugung polymorphen Verhaltens 67
- Zone of Pain 145
- Zone of Uselessness 146
- Zugriffsmodifikatoren
 - architektonische Packages 306
- Zustände
 - Speichern von Transaktionen ohne Zustände 78
- Zustandsänderungen
 - Probleme durch Nebenläufigkeit 75
- Zuweisung
 - funktionale Programmierung 48
- Zyklen
 - Abhängigkeiten abschaffen 131
 - durchbrechen 134
 - im Komponentenabhängigkeitsgraphen 131
 - Problemstellung beim wöchentlichen Build 130
- Zyklus durchbrechen
 - Acyclic-Dependencies-Prinzip 134