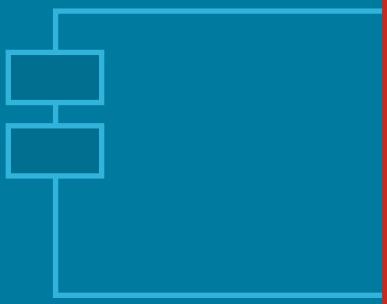
Philipp **FRIBERG**

SOFTWARE-ARCHITEKTUR PRAGMATISCH

Der Weg von der Software- in die Unternehmens-Architektur



Zusatzmaterial unter
plus.hanser-fachbuch.de

HANSER

Friberg

Softwarearchitektur pragmatisch

Ihr Plus – digitale Zusatzinhalte!

Auf unserem Download-Portal finden Sie zu diesem Titel kostenloses Zusatzmaterial.

Geben Sie auf **plus.hanser-fachbuch.de** einfach diesen Code ein:

plus-4ft7p-S32gm

Bleiben Sie auf dem Laufenden!

Unser **Computerbuch-Newsletter** informiert Sie monatlich über neue Bücher und Termine. Profitieren Sie auch von Gewinnspielen und exklusiven Leseproben. Gleich anmelden unter:

www.hanser-fachbuch.de/newsletter

Philipp Friberg

Softwarearchitektur pragmatisch

Der Weg von der Software- in die Unternehmens-Architektur

HANSER

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.

Lektorat: Brigitte Bauer-Schiewek
Copy editing: Petra Kienle, Fürstenfeldbruck
Umschlagdesign: Marc Müller-Bremer, München, *www.rebranding.de*
Umschlagrealisation: Max Kostopoulos
Titelmotiv: © stock.adobe.com/monsitj
Satz: Manuela Treindl, Fürth
Druck und Bindung: Hubert & Co. GmbH & Co. KG BuchPartner, Göttingen
Printed in Germany

Print-ISBN: 978-3-446-47370-6
E-Book-ISBN: 978-3-446-47437-6
E-Pub-ISBN: 978-3-446-47555-7

Inhalt

Einführung ..... **XI**
Was erwartet Sie in diesem Buch? XIII
Zusatzmaterial XV
Danksagung XV

Der Autor ..... **XVI**

I Teil 1: Architektur entdecken **1**

1 Einführung in die Software-Architektur **3**
1.1 Geheimnisprinzip 4
1.2 Aufgaben eines Betriebssystems 6
1.3 Strukturierung 10
1.3.1 Komponenten 10
1.3.2 Modul 12
1.3.3 Bibliotheken und Frameworks 12
1.3.4 Modularisierung 13
1.3.5 API 18
1.3.6 Systemcall und API live 20
1.3.7 Offen-Geschlossen-Prinzip 24
1.4 Schichtenarchitektur 26
1.5 Trennung von Strategie und Mechanismus 29
1.6 Zusammenfassung 30

2 Betriebssystemarchitekturen **32**
2.1 Monolithische Systeme 33
2.1.1 Verbesserte monolithische Systeme 35
2.1.2 Beispiel Unix System V 37
2.1.3 Pipe-Muster und Orthogonalität-Prinzip 39
2.1.4 Datei-Subsystem 40
2.2 Mikrokernel-Systeme 42
2.2.1 Mikrokernel 42
2.2.2 Beispiel QNX 44

2.3 Hybridkernel-Systeme 46
2.3.1 Hybridkernel 46
2.3.2 Beispiel Windows 46
2.4 Vergleich 48
2.5 Mobiles Betriebssystem Android 49
2.6 Zusammenfassung 52

3 Der Raum und die Zeit 53
3.1 Was ist ein Prozess? 53
3.2 Der Raummultiplex 57
3.2.1 Prozesserzeugung 58
3.2.2 Prozesskommunikation 60
3.2.3 Threads 61
3.3 Der Zeitmultiplex 63
3.3.1 Prozessunterbrechung 64
3.3.2 Prozesszustände 65
3.4 Scheduling 66
3.4.1 Grundalgorithmen 67
3.4.1.1 First In First Out (FIFO) 67
3.4.1.2 Round Robin (RR) 69
3.4.1.3 Prioritäten 71
3.4.1.4 Shortest Job First (SJF) 73
3.4.1.5 Shortest Remaining Time (SRT) 74
3.4.2 Multiprozessorsysteme 74
3.4.3 Beispiel Windows 75
3.5 Zusammenfassung 76

II Teil 2: Entwerfen einer Architektur 79

4 Einflussfaktoren der Architektur 81
4.1 Ziele 83
4.2 Stakeholder 85
4.3 Randbedingungen 88
4.3.1 Bereiche 88
4.3.2 Werkzeugkoffer 91
4.4 Systemkategorien 93
4.5 Anforderungen 95
4.6 Scotland Trading 98
4.7 Kontextsicht 100
4.8 Qualitätsanforderungen 103
4.8.1 ISO 25010 103
4.8.2 Qualitätsszenario 110

4.8.2.1 Benutzbarkeit . . . 110
4.8.2.2 Sicherheit . . . 111
4.8.2.3 Wartbarkeit . . . 111
4.8.2.4 Portabilität . . . 111
4.8.2.5 Zuverlässigkeit . . . 111
4.8.2.6 Funktionale Eignung . . . 112
4.8.2.7 Leistungseffizienz . . . 112
4.8.2.8 Kompatibilität . . . 112
4.8.2.9 Skalierbarkeit . . . 112
4.9 Technische Schulden . . . 113
4.10 Zusammenfassung . . . 115

5 TOGAF und ArchiMate . . . 117
5.1 TOGAF . . . 119
5.2 TOGAF ADM . . . 122
5.2.1 Vorbereitungsphase (Preliminary) . . . 124
5.2.2 A: Architekturvision (Architecture Vision) . . . 125
5.2.3 B: Geschäftsarchitektur (Business Architecture) . . . 126
5.2.4 C: Informationssystemarchitekturen (Information Systems Architectures) 127
5.2.5 D: Technologiearchitektur (Technology Architecture) . . . 129
5.2.6 E: Chancen und Lösungen (Opportunities and Solutions) . . . 130
5.2.7 F: Migrationsplanung (Migration Planning) . . . 130
5.2.8 G: Steuerung und Implementierung (Implementation Governance) . . . 131
5.2.9 H: Architekturveränderungen (Architecture Change Management) . . . 131
5.2.10 AM: Anforderungsmanagement (Requirements Management) . . . 131
5.2.11 Architektur-Repository (Architecture Content) . . . 132
5.3 Modellierungssprache ArchiMate . . . 133
5.3.1 View und Viewpoint . . . 134
5.3.2 Applikations-Layer Grundmuster . . . 135
5.3.3 Geschäfts-Layer . . . 137
5.3.4 Ableitungen . . . 144
5.3.5 Layered View . . . 146
5.3.6 Applikations-Layer Scotland Trading . . . 148
5.3.7 Technologie-Layer . . . 155
5.3.8 Implementation und Migration . . . 161
5.3.9 Strategie und Motivation . . . 162
5.4 Architekturprinzipien . . . 164
5.4.1 TOGAF-ADM-Techniken . . . 164
5.4.2 Architekturprinzipien erklärt . . . 164
5.4.3 Die 21 Prinzipien . . . 165
5.4.4 Prinzip 22: Buy-Configure-Build . . . 166
5.5 Architektur-Board . . . 167
5.6 Zusammenfassung . . . 170

6 Applikationsarchitektur 172
6.1 Bausteinsicht 173
6.1.1 Idee 173
6.1.2 Dynamischer Preisbilder 174
6.1.3 Separation-Of-Concerns-Prinzip 177
6.2 Monolith und Services 179
6.2.1 Monolith 179
6.2.2 Microservices 179
6.2.3 Nanoservice 180
6.2.4 Orchestrierung oder Choreografie? 180
6.3 Realisationssicht 183
6.3.1 Idee 183
6.3.2 Logische Gruppierung 183
6.3.3 Analyse pro Gruppe 185
6.3.4 Realisierungssicht der Scotland Trading 190
6.3.5 Muster und Prinzipien 195
6.3.6 Qualitätsszenarien 196
6.3.6.1 Wartbarkeit 197
6.3.6.2 Zuverlässigkeit 197
6.3.6.3 Leistungseffizienz 198
6.3.6.4 Skalierbarkeit 198
6.3.7 Adapter-Muster 199
6.3.8 Muster Backend for Frontend (BFF) 199
6.3.9 Native-Cloud-Muster 201
6.3.10 Hinweise 201
6.4 Referenzarchitekturen 202
6.5 Querschnittliche Konzepte 203
6.6 Pace-layered Application Strategy 205
6.6.1 Modell 205
6.6.2 Scotland Trading 208
6.6.3 Erfahrungen 209
6.7 Laufzeitsicht 210
6.8 Zustandssicht 212
6.9 Datensicht 216
6.9.1 Datensicht von Scotland Trading 216
6.9.2 Correlation IDs Prinzip 221
6.9.3 Event-Sourcing-Prinzip 221
6.10 Dokumentation – arc42 222
6.10.1 Gliederungsvorschlag 222
6.10.2 Glossar 225
6.11 Zusammenfassung 226

7 Integrationsarchitektur 229
7.1 Enterprise Application Integration Pattern (EAIP) 229
7.2 Adapter (Message Endpoints) 232
7.2.1 Push oder Pull? 232
7.2.2 Synchron und asynchron 233
7.2.3 Scotland Trading 235
7.3 Nachrichten (Message Constructs) 235
7.4 Kanäle, Transformation und Routing 236
7.4.1 Punkt-zu-Punkt-Kanal 237
7.4.2 Transformation 242
7.4.3 Vermittler, Routing 244
7.4.4 Nachrichtenkanal 245
7.4.5 Publish-Subscribe 246
7.4.6 ESB und BPMS 250
7.5 API-Gateway 250
7.6 Zusammenfassung 254

8 Scotland Trading 255
8.1 Hilfsmittel 261
8.2 ArchiMate 261

Anhang 265
Checkliste Architektur 265
Glossar 267
Literatur 271
Teil 1: Architektur entdecken 271
Teil 2: Entwerfen einer Architektur 271

Stichwortverzeichnis 273

Einführung

Wenn eine Gruppe von Menschen zusammenleben muss, kann einiges schiefgehen – sei es in einer Partnerschaft, in der Familie oder einer größeren Zweckgemeinschaft eines Wohnblocks. Für ein friedliches Zusammenleben braucht es eine Ordnung, die je nach Situation expliziter oder impliziter definiert ist. Ein frisch verliebtes Paar braucht nicht unbedingt eine Hausordnung, viele Regeln ergeben sich durch die gegenseitige Rücksichtnahme. In einem Mehrfamilienhaus hingegen wird es bereits schwieriger, besonders wenn es gemeinsame Ressourcen, wie zum Beispiel eine Waschküche, zu teilen und zu organisieren gibt. Nicht selten führen sogenannte Kleinigkeiten zu großen Problemen. Je mehr Menschen mit unterschiedlichen Ansichten und Lebenseinstellungen aufeinandertreffen, desto wichtiger wird eine explizite Ordnung.

Analog gilt dies auch für Software-Systeme. Ein PC zu Hause kann ad-hoc betrieben werden. In einem größeren Unternehmen gibt es eine Vielzahl von Systemen, auf denen die unternehmenskritischen und weniger kritischen Applikationen laufen. Sie müssen miteinander Daten austauschen, sind prozess-technisch voneinander abhängig und die Schlagader eines Unternehmens. Diese Komponenten brauchen für die korrekte Funktionsweise genauso eine Ordnung wie wir Menschen, besonders da diese von Menschen entwickelt und betrieben werden. Mit der Software-Architektur werden solche IT-Systeme beschrieben und mittels Prinzipien deren Funktionieren festgelegt. Deshalb gefällt mir die Definition des Begriffs „Software-Architektur“ von Wilhelm Hasselbring sehr gut:

> *Die grundlegende Organisation eines Systems, dargestellt durch dessen Komponenten, deren Beziehungen zueinander und zur Umgebung sowie den Prinzipien, die den Entwurf und die Evolution des Systems bestimmen.*
>
> Wilhelm Hasselbring

Dieses Buch wird Ihnen dabei helfen, sich in den Architekturen zurechtzufinden und zu bewegen. Es wird Ihnen helfen, gesamtheitlichere Architekturen zu entwerfen.

Architekturen

Diese Ordnung gibt es auf verschiedenen Ebenen, in verschiedenen Architekturen, wie z. B. die Architektur der Software, der Infrastruktur, der Netzwerke, aber auch der Geschäftsprozesse. Diese Architekturen bilden im Unternehmen wiederum eine Zweckgemeinschaft, die das Ziel hat, mit der IT die Erreichung der Geschäftsziele des Unternehmens zu unterstützen. Dafür gibt es die Unternehmensarchitektur oder auch Enterprise-Architektur (siehe folgendes Bild). Wir werden verschiedene Ebenen der Architektur durchlaufen, wobei der Fokus auf der Applikationsarchitektur, der Software, liegt.

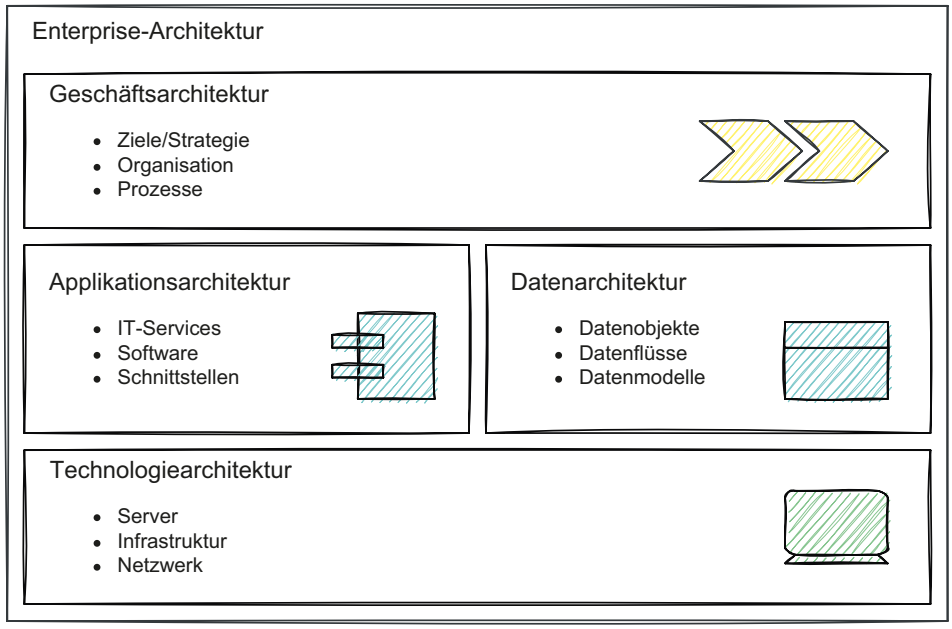


Bild 1 Einordnung der Architekturen

Struktur

Betrachten wir die Definition von Software-Architektur von Wilhelm Hasselbring genauer: Es geht darum, einen Plan, eine Struktur zu haben. Um solch einen Plan zu erhalten, müssen die Systeme in Komponenten zerlegt und beschrieben werden. Wichtig sind deshalb dessen Beziehungen zueinander. Dieser Plan schafft also eine Ordnung und einen Überblick. Vergleichen Sie es mit einem Stadtplan für Touristen (widerspiegelt die Organisation). In diesen Plänen sind die Sehenswürdigkeiten (Komponenten) prominent dargestellt. Je nach Interessengebiet des Besuchers gibt es dann Empfehlungen, in welcher Reihenfolge welche Sehenswürdigkeiten besucht werden könnten. Diese werden also zueinander in Beziehung gestellt und mittels Wegbeschreibungen verknüpft. Für den Tourist überflüssige Informationen, z. B. die Börse, werden aus Platzgründen weggelassen. Das heißt nicht, dass diese unwichtig sind, aber für den Zweck eines Reiseführers sind sie etwas weniger relevant. *Architektur ist also eine Sicht auf Systeme, die sich auf die relevanten Aspekte konzentriert.* Die nicht benötigten Informationen für einen Adressaten werden weggelassen. Der Touristenplan hat das Ziel, dem Tourist die schönsten Orte der Stadt zu zeigen. Ein Plan für öffentliche Verkehrsmittel verfolgt ein anderes Prinzip. Dort wird das Prinzip der übersichtlichen schematischen Darstellung der Transportwege angewendet.

Gernot Starke hat hierzu folgende Punkte in seinem Buch „Effektive Software-Architekturen“ [Sta18] aufgelistet:

- *Architektur enthält Strukturen.*
- *Architektur beschreibt eine Lösung und basiert auf Entscheidungen.*
- *Architektur schafft Ordnung und einen Überblick.*

- *Softwarearchitekturen machen Komplexität von Systemen beherrschbar und verständlich.*
- *Architektur lässt nicht benötigte Informationen gezielt weg.*

Je nach beruflicher Erfahrung dürften die Erläuterungen unterschiedliche Assoziationen hervorrufen. Ein Software-Entwickler wird einen Plan von Klassen und Komponenten, entworfen in UML, vor sich sehen. Ein Applikationsarchitekt sieht einen Plan von ganzen Software-Systemen vor sich, entworfen in ArchiMate. Und Sie?

Was erwartet Sie in diesem Buch?

Als Architekt einer Teilarchitektur, einer bestimmten Domäne, dürfen Sie sich als Vertreter an den Unternehmensdiskussionen zur Unternehmensarchitektur beteiligen. Es macht Freude, sich aktiv an Veränderungen beteiligen zu können. Vielleicht sind Sie auf dem Weg dazu oder wurden es kürzlich. Leider ist es nicht ganz einfach, alle Architekturzusammenhänge zu sehen und zu verstehen. Dafür werden tüchtige Architekten mit Unternehmenssicht gebraucht. Mit Weitsicht und der Fähigkeit, über den Tellerrand zu sehen! Es werden pragmatische Architekten gebraucht.

Das Wort „pragmatisch“ kommt aus dem Griechischen und heißt **geschäftskundig**, **tüchtig**. Es ist eine *Einstellung*, bei der der Pragmatiker auf die sachlichen Gegebenheiten und auf *praktisches Handeln* ausgerichtet ist. In diesem Buch geht es also um das *Tun*. Wie können Sie die sachlichen Gegebenheiten Ihrer Tätigkeiten praxistauglich anwenden? Aber aufgepasst, mit pragmatisch meine ich nicht waghalsig, also schnelle Entscheidungen ohne Risikoabwägung zu tätigen!

Egal, ob Sie bereits Architekt sind oder einer werden möchten und egal, aus welchem Bereich der IT Sie kommen: Ein paar *Muster* und *Prinzipien* der Software-Architektur helfen Ihnen, unternehmensweite Entscheidungen zu treffen. Für diese Muster müssen Sie nicht programmieren können. Wir lernen vor allem diese Muster zu lesen. Sie zu *analysieren*.

Es geht dann weiter in die *Applikationsarchitektur* und werden das Unternehmen *Scotland Trading* in Ihrer Transformation aktiv begleiten. Dabei stehen die Zusammenhänge im Mittelpunkt. Wir *beschäftigen* uns mit der Architektur und denken uns durch ein Projekt, bevor die Teilarchitekturen die Arbeit weiterführen. Wie in der Architektur üblich, gibt es für eine Problemstellung oder Situation kein „richtig“ oder „falsch“, sondern eher ein mehr oder weniger zutreffend. Deshalb ist mir das Wort *beschäftigen* wichtig. Lernen Sie, welche Best Practises in welcher Situation angebracht sind. Um dies zu lernen, hilft es, mit offenen Augen durch den IT-Dschungel zu gehen. Es gibt immer etwas zu entdecken.

Das Buch ist folglich in zwei Teile gegliedert: Architektur entdecken und aktiv eine Architektur entwerfen.

Teil 1: Architektur entdecken

Im ersten Teil entdecken wir die Software-Architektur am Beispiel der Betriebssysteme. Diese sind die Basis der Informatik. Sie regeln zum Beispiel den Zugriff auf gemeinsame Güter und ermöglichen ein Zusammenleben mehrerer Prozesse. Interessant ist für die Architekturanalyse die Evolution – von Unix aus den 60er-Jahren zu mobilen Betriebssystemen, die uns in den letzten Jahren eroberten. Wie kann diese Veränderung mit der Architektur vollzogen werden? Wir wollen Software analysieren und Architekturmerkmale identifizieren. So ergibt sich ein Katalog aus *Architekturprinzipien* und *Architekturmustern*, die wir in anderen Architekturen wieder antreffen.

Teil 2: Entwerfen einer Architektur

Aus den spezifischen Details wachsen wir, wie soft im Leben, heraus und sind plötzlich für größere Zusammenhänge mitverantwortlich, zum Beispiel, wenn der IT-Infrastruktur-Spezialist zum Architekten wird. Er soll dann in wichtigen Unternehmensprojekten mit all den anderen Architekten eine optimale gemeinsame Architektur mitdefinieren.

Im zweiten Teil wachsen wir also als Architekt von der Software-Architektur in die Unternehmensarchitektur hinein. Wir wechseln unsere Rolle vom Analysten zum Architekten. Die Einflussfaktoren und Qualitätsmerkmale prägen dessen Arbeit stark. Um das Thema konkreter anzugehen, durchlaufen wir die Architektur anhand eines Transformationsprojekts: Das Unternehmen *Scotland Trading* will sich neu positionieren und einen dynamischen Preisbilder bauen. Dazu durchlaufen wir gewisse TOGAF-Phasen, entdecken ArchiMate und vertiefen uns in der *Applikations- und Informationsarchitektur*. Dabei lernen wir verschiedene Sichten kennen, die uns helfen, die Architektur zu beschreiben. Mit den konkreten Architekturkonzepten aus dem ersten Teil sind wir gut gewappnet für diese Aufgabe.

Ein wichtiger Teil der Architektur ist die *Integrationsarchitektur*. Dort fügen wir die verschiedenen Bausteine zu einem Kommunikationsnetz zusammen. Wir überlegen uns, wie wir Tausende von Punkt-zu-Punkt-Verbindungen eleganter umsetzen können.

Ich achte hier wieder auf den pragmatischen Weg, der den Übergang der Sichtweise fördert.

Buchaufbau

Zusammengefasst ist das Buch folgendermaßen aufgebaut:

Teil 1: Architektur entdecken	Kapitel 1 bis 3	*Thema:* Architektur kennenlernen *Sicht:* Architekt beim Analysieren *Beispiel:* Betriebssysteme *Resultate:* Architekturprinzipien und Muster
Teil 2: Entwerfen einer Architektur	Kapitel 4 bis 8	*Thema:* Mitarbeit in der Unternehmens- und Applikationsarchitektur *Sicht:* Architekt beim Modellieren *Beispiel:* Transformationsprojekt der *Scotland Trading* *Resultate:* Architekturbausteine und Tools, in denen die Muster und Prinzipien von Teil 1 angewendet werden können

In jedem Kapitel werden wir Hinweise zu anderen Situationen erhalten und erfahren, wie wir Pragmatiker mit den Informationen umgehen können. Am Ende jedes Kapitels merken wir uns, in Form einer Checkliste, an was wir denken sollten. Oft sind das Fragen, die wir von Zeit zu Zeit wieder hervorholen können und über die wir unsere aktuelle Arbeit hinterfragen können ...

Dieses Buch wendet sich ausdrücklich an alle Geschlechter und Menschen jeglicher Geschlechtsidentität. Für die bessere Lesbarkeit wird die männliche Bezeichnung verwendet.

Zusatzmaterial

Unter

https://plus.hanser-fachbuch.de/

finden Sie den Quellcode der Programme aus dem ersten Teil und die ArchiMate-Modelle aus dem zweiten Teil. Mit der Eingabe des Codes

`plus-4ft7p-S32gm`

können Sie diese herunterladen.

Unter

https://arch.xapps.ch

finden Sie Unterlagen, die ich in meinem Unterricht zu diesem Thema verwende.

Danksagung

Ich durfte viele Diskussionen mit Kollegen, Studenten und ehemaligen Kunden zu diesem Thema führen. Vieles ist in dieses Buch auf die eine oder andere Art eingeflossen. Dafür danke ich und hoffe auf weitere spannende Diskussionen.

Einen besonderen Dank möchte ich meinem Architekturkollegen Ferdinand Moosmann geben. Er hat mir viele Tipps und Anregungen während des Entstehens des Buchs gegeben.

Wenn ich nicht für die Schule, an der ich Lehrbeauftragter bin, dieses Modul aufgebaut hätte, wäre dieses Buch wohl nicht entstanden. Dafür möchte ich mich bei Beat Hartmann bedanken.

Der Autor

Philipp Friberg arbeitet bei Interdiscount|microspot.ch als SAP-Architekt, ist Product Owner (PO) und Mitglied im Architektur-Board. Zuvor arbeitete er als SAP-Berater, seine dortigen Arbeitsschwerpunkte waren die Architekturberatung sowie Entwicklung und Projektleitung für kundenindividuelle Softwarelösungen. Darüber hinaus vermittelt er an der TBZ Höheren Fachschule Zürich sein Wissen im Fach Software-Architekturen. Er ist als Autor von verschiedenen Fachartikeln und Büchern bekannt und an Konferenzen als Sprecher anzutreffen. Philipp Friberg absolvierte ein Studium zum Software-Engineer FH an der Hochschule Rapperswil und zum Master of Science in Business Information Systems an der Hochschule Liechtenstein.

Kontakt: *philipp@xapps.ch*

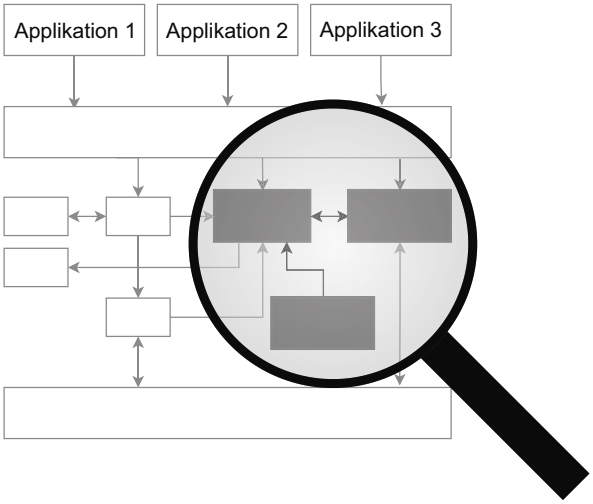
Twitter: @friibiiCH

WWW: *https://www.xapps.ch*

Hinweis

In diesem Buch wurde aufgrund der besseren Lesbarkeit auf eine gendergerechte Sprache verzichtet. Selbstverständlich spricht der Autor aber alle Personen jeglichen Geschlechts gleichermaßen an.

Teil 1: Architektur entdecken



1 Einführung in die Software-Architektur

Welche Fragen werden in diesem Kapitel beantwortet?

- Welche Aufgabe hat ein Betriebssystem?
- Weshalb ist ein Geheimnis hilfreich?
- Wie erfolgt die Strukturierung eines Systems?
- Welche Regeln gelten bei einer Schichtenarchitektur?
- Was wird unter der Trennung von einer Strategie und einem Mechanismus verstanden?

In den ersten drei Kapiteln wollen wir anhand von Betriebssystemen möglichst konkret Muster und Prinzipien des Software-Designs kennenlernen. Weshalb verwenden wir Betriebssysteme als Beispiel? Einerseits sind Betriebssysteme allgegenwärtig, andererseits zeigt ihre Evolution die Veränderungen der Architektur über die Zeit auf. Um Parallelen außerhalb des Beispiels der Betriebssysteme aufzuzeigen, werde ich in Kästen Vergleiche zur SAP S/4HANA-Architektur vornehmen.

Was ist SAP S/4HANA?

SAP schreibt auf ihrer Homepage[1] dazu: „SAP S/4HANA ist ein zukunftsfähiges ERP-System (Enterprise Resource Planning) mit integrierten intelligenten Technologien, einschließlich KI, maschinellem Lernen und erweiterten Analysen. Es transformiert Geschäftsprozesse mit intelligenter Automatisierung und läuft auf SAP HANA – eine der marktführenden In-Memory-Datenbanken."

Es beinhaltet neben der klassischen Buchhaltung und dem Controlling auch Materialwirtschaft, Verkauf, Service, Fertigung, Einkauf & Beschaffung, Lieferkette, Marketing, Personalwesen und vieles mehr. Das System lässt sich durch Konfiguration (Customizing) auf die Kundenprozesse einstellen und durch Entwicklungen erweitern.

Das System ist technisch gesehen ein Applikations-Server mit Programmcode. Die Programme, Daten und Prozesse sind in der Programmiersprache ABAP (objektorientiert oder strukturiert) entwickelt und werden direkt auf dem Applikations-Server ausgeführt. Als Frontend wird ein Rich-Client und ein Web-Frontend verwendet.

[1] *https://www.sap.com/swiss/products/enterprise-management-erp.html*

1.1 Geheimnisprinzip

Im täglichen Leben teilen viele Leute über die sozialen Plattformen, wie zum Beispiel Facebook, einiges von sich. Wir geben dabei bewusst Informationen über uns preis. Aber nicht alles wollen wir öffentlich machen, jeder möchte doch auch sein Geheimnis haben. Wenn wir Besuch zu Hause empfangen, wollen wir ihm wirklich einen Einblick in unser Schlafzimmer gewähren? Wahrscheinlich nicht jedem. Deshalb ist es ein eigenes Zimmer, das durch eine Tür abgegrenzt ist, die man bewusst schließen und öffnen kann.

Auch bei Betriebssystemen müssen oder wollen wir nicht alles im Detail wissen. Wichtig ist, dass wir wissen, wie wir dem Betriebssystem unseren Wunsch, z. B. mehr Speicher zu erhalten, mitteilen können. Wie dieser Speicher ermittelt und reserviert wird, ist uns egal. Wir trennen also das *Was* vom *Wie*.

Bei Software allgemein gibt es das Prinzip, dass das Innenleben eines Bausteins vor der Außenwelt verborgen bleiben soll. Folgend nennen wir solche Bausteine *Komponenten*. Dafür wird eine öffentliche Schnittstelle definiert, die beschreibt, was die Komponente macht. Das *Wie* dürfen wir getrost vor dem äußeren Zugriff verborgen halten. Es ist also eine *Blackbox*, die eine Dienstleistung für uns erbringt. So vermeiden wir ungewollte Abhängigkeiten zwischen Komponenten. Bild 1.1 stellt die beiden Konzepte „mit Geheimnisprinzip“ und „ohne Geheimnisprinzip“ einander gegenüber. Oben greift eine Komponente beliebig auf Subkomponenten in der Hauptkomponente zu, das heißt, das Innenleben der Hauptkomponente ist bekannt. Daraus ergibt sich eine hohe, direkte Abhängigkeit zwischen der aufrufenden Komponente und den inneren Subkomponenten. Im unteren Teil des Bilds ist das Blackbox-Prinzip erfüllt: Es gibt eine Schnittstelle, über die auf das Innenleben zugegriffen wird. Die Abhängigkeit ist über diese Schnittstelle definiert. Nehmen wir nun an, dass die Hauptkomponente einen Algorithmus implementiert und dieser geändert werden soll. Dies kann der Hersteller nun problemlos machen, solange die Schnittstelle gleich bleibt. Bei einer hohen Abhängigkeit wäre das nicht ohne weitere Anpassungen in den aufrufenden Komponenten möglich. Dieses Prinzip nennen wir *Geheimnisprinzip* oder auch „*Trenne Belange von Verantwortlichkeiten*“.

Beim *Geheimnisprinzip* sollen die Interna einer Komponente außerhalb der Komponente nicht sichtbar sein, sie soll eine Blackbox darstellen, die über ein API angesprochen wird.

Synonyme für dieses Prinzip sind Prinzip der Geheimhaltung, Information Hiding, Prinzip der Kapselung (engl. encapsulation).

Oft wird bei der Umsetzung dieses Prinzips das *Fassaden-Muster* angewendet. Bei diesem wird eine Komponente vor die anderen Komponenten gesetzt, die die Kommunikation nach außen und das Handling der internen Komponenten übernimmt.

Das Strukturmuster *Fassade* bietet eine einheitliche Schnittstelle zu einer Menge von Schnittstellen eines Subsystems.

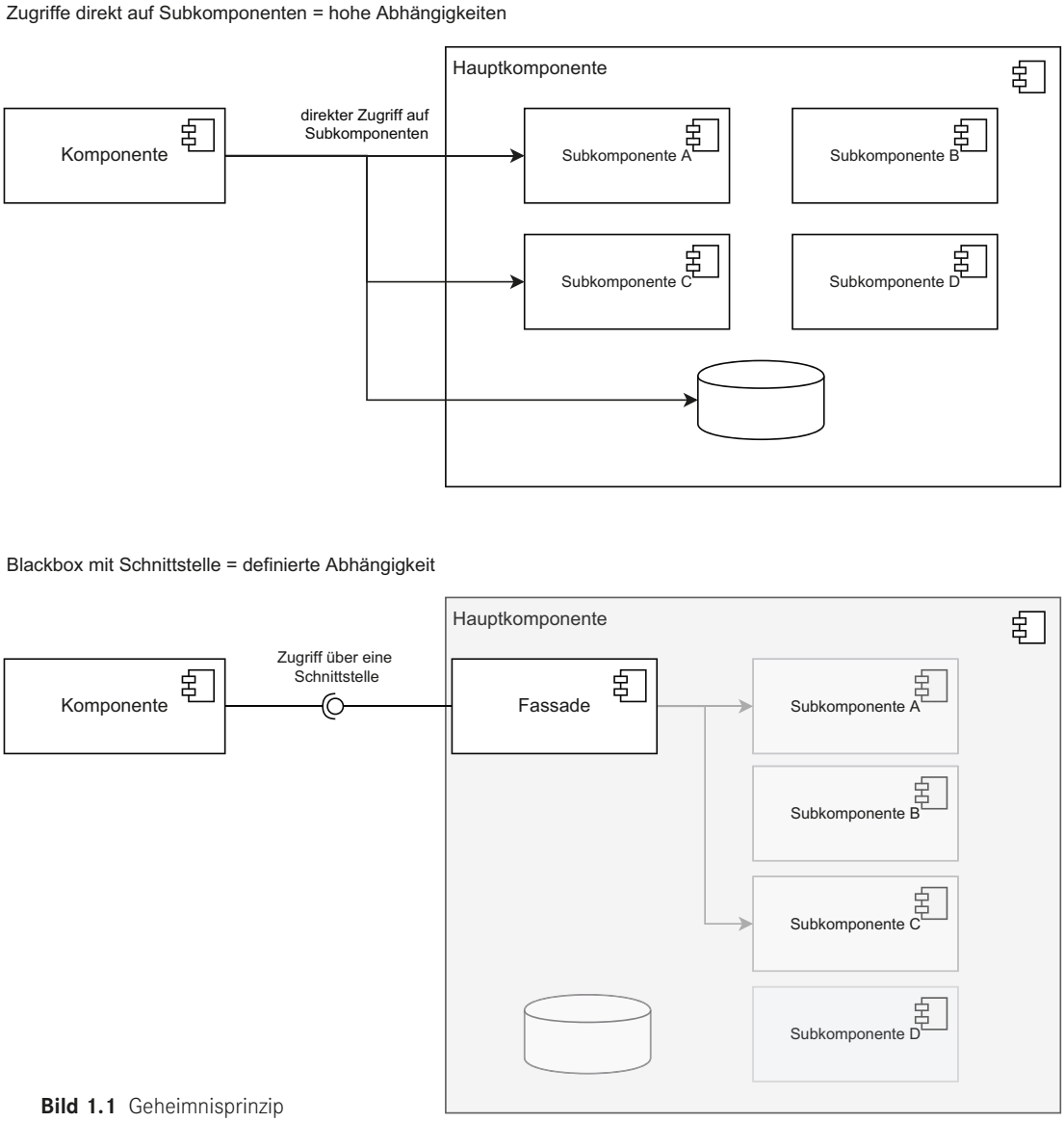


Bild 1.1 Geheimnisprinzip

Dieses Muster ist wie ein Pförtner, der genau kontrolliert, wer auf das Areal darf, wer nicht darf und wohin diese dürfen. Das Areal selbst ist eine Blackbox. Solche Muster, eine Blackbox mit einer wohldefinierten Schnittstelle und einer Fassade, treffen wir in der Informatik öfter an. Ich denke da zum Beispiel an die Netzwerkarchitektur: Der Reverse-Proxy, der das Netzwerk vor der Außenwelt verbirgt, ist der Pförtner, der den Verkehr kontrolliert und dann intern gezielt an einen Server weitergibt. Sie finden viele Beispiele, besonders in der Integrationsarchitektur, und wir werden ein paar auch in diesem Buch antreffen. Beim Betriebssystem haben wir das auch. Die Blackbox, die die Tastaturbefehle korrekt umsetzt, oder die eine Schnittstelle bereitstellt, um Dateien zu schreiben, egal, welches Dateisystem verwendet wird.

1.2 Aufgaben eines Betriebssystems

Diese Blackbox beim Betriebssystem wollen wir folgend genauer betrachten, dafür sollten wir uns aber zuerst Gedanken zu den Aufgaben eines Betriebssystems machen.

Ein Betriebssystem ist auf jedem unseren Notebooks, Workstations, Server, Mobilfunkgerät oder gar IoT-(Internet of Things-)Gerät und Embedded-System installiert. Immer in einer etwas anderen Form mit leicht anderer Funktionalität. Die erforderlichen Aufgaben können je nach Anwendungsfall sehr unterschiedlich sein. Schlussendlich benötigen alle Programme ein Betriebssystem und dessen Services für die Ausführung. Ein Betriebssystem hat also die Aufgabe, Programme zu steuern und den Betrieb des Rechnersystems zu regeln. Dazu stellt ein Betriebssystem verschiedene Services bereit, konkret beispielsweise:

- *Verwaltung der Prozesse*, d. h. das Laden der Prozesse in den Hauptspeicher (gemeinsames Gut), die Steuerung der Abarbeitungsreihenfolge der Prozesse (Scheduling) und Kontrolle des Unterbrechungsmechanismus für Prozesse (Interrupts[2])
- *Kommunikation* zwischen den Prozessen
- *Zuteilung der Betriebsmittel* (gemeinsames Gut) und deren Rücknahme
- *Verteilung des Speichers* auf die Prozesse. Anderseits müssen für gemeinsame Daten Prozesse auf gemeinsamen Speicher zugreifen können.
- *Zugriff auf Dateien*, egal wo sich die Datei physisch befindet
- *Schutzkonzepte* zu gewährleisten, wie Speicheradressierung, Programmausführung, Zugriff auf Hardware, Berechtigungen

Das Betriebssystem wird also zum Verwalter der Hardware, regelt das Zusammenleben der Prozesse und stellt eine gemeinsame Basis für die Ausführung zur Verfügung. Ist das nicht recht ähnlich zur Rolle vom Staat? Er stellt uns die Infrastruktur als Basis zur Verfügung, regelt mit Gesetzen das Zusammenleben und ist Verwalter des öffentlichen Raums. Die Länder haben dabei zum Teil unterschiedliche Probleme, je nach Lage und Größe: Zum Beispiel hat ein Land mit Meer andere Herausforderungen als ein Binnenland mit hohen Bergen. In diesem Sinne gibt es auch Betriebssysteme für unterschiedliche Situationen, die optimiert sind auf die zu lösenden Aufgaben.

Es gibt verschiedene Definitionen von Betriebssystemen, auch die Norm DIN 44 300, die aber sehr schwer lesbar ist. Aus verschiedenen Quellen zusammengeführt, gefällt mir diese:

Ein Betriebssystem ist ein System, bestehend aus verschiedenen Programmen, das die Infrastruktur verständlich als Service zur Verfügung stellt und das Zusammenleben der Programme regelt.

[2] Ein Interrupt ist eine Programmunterbrechung, die von einem Programm oder von Hardware ausgelöst wird. So kann Code außerhalb des normalen Programmablaufs ausgeführt werden.

Diese Definition eines Betriebssystems kann auch auf ein ERP-System, z. B. dem SAP S/4HANA, übertragen werden:

Das S/4HANA System besteht aus verschiedenen Programmen, die die Geschäftsprozesse verständlich als Services zur Verfügung stellen und die Prozessabfolge regeln. Um einen Geschäftsprozess erfolgreich abzuwickeln, braucht es ein *gemeinsames Gut*. Das kann ein Artikel oder ein Kunde sein.

Wenn *kein* Betriebssystem vorhanden ist, wie zum Beispiel bei der Arduino-Plattform, müssen die Hardware-Komponenten direkt aus dem Anwenderprogramm heraus angesprochen werden. Verschiedene Peripherie-Geräte haben dann immer Anpassungen am Programm zur Folge, da sie spezifische physische Kanäle benutzen. Mit einem Betriebssystem übernimmt dieses die spezifischen Umsetzungen und stellt einheitlich definierte Schnittstellen, folgend *logische Kanäle* genannt, zur Verfügung (Bild 1.2). Ein klassisches Beispiel ist die Tastatur:

- Ein Programm will nicht die technischen Codes einer Tastatur verwenden, sondern den gedrückten Buchstaben, unabhängig vom benutzten Tastaturlayout.
- Ob es sich um eine USB-Tastatur, eine Bluetooth-Tatstatur oder sogar eine Bildschirmtastatur handelt, soll dem Programm genauso egal sein.

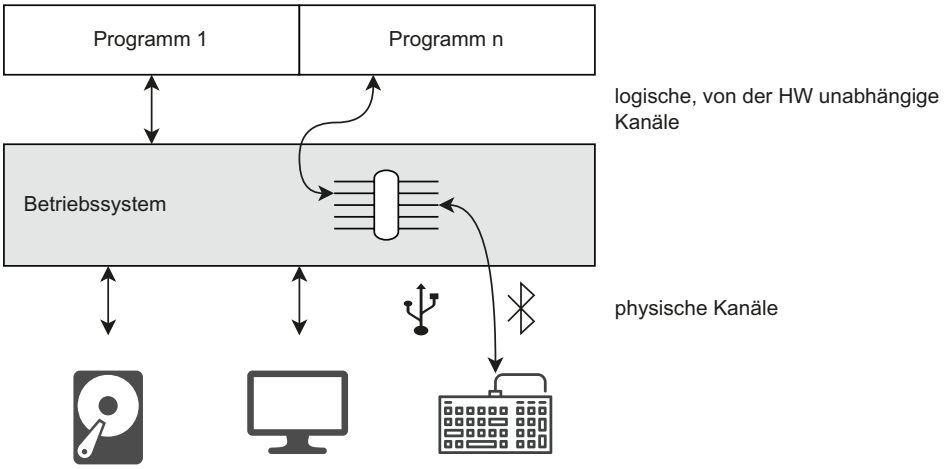


Bild 1.2 Logische und physische Kanäle

Aus eigener Erfahrung wissen Sie vielleicht, dass diese Umsetzung bei Betriebssystemen mittels Gerätetreibern gelöst ist. Diese steuern die Hardware anhand von logischen Befehlen. Dabei wird das strukturelle Muster, das wir später immer wieder antreffen werden, angewendet: Eine Komponente übersetzt ein Format in ein anderes Format (Adapter-Muster) oder stellt ein standardisiertes API für eine abstrakte Blackbox zur Verfügung.

Damit das Betriebssystem sich weiterentwickeln kann, müssen diese Aufgaben mit dem Geheimnisprinzip „geschützt“ werden. In Bild 1.3 ist dies mit dem großen schwarzen Block visualisiert. Die einheitliche Schnittstelle wird Systemcall-Interface genannt, da es sich um einen Systemaufruf handelt.

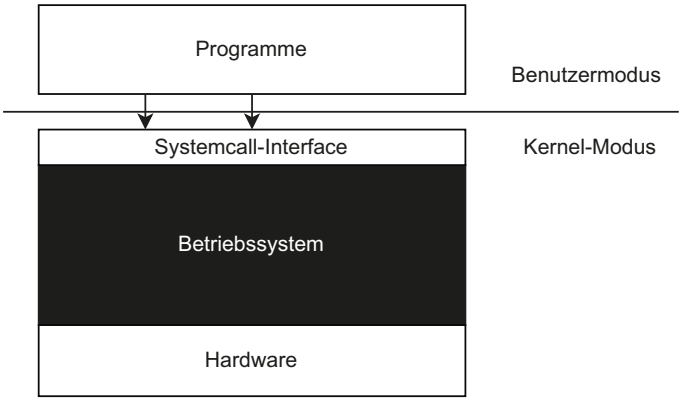


Bild 1.3
Geheimnisprinzip beim Betriebssystem

Um zu verhindern, dass Programme die Blackbox, das Betriebssystem, umgehen, stellt die CPU einen zweistufigen Sicherheitsmechanismus zu Verfügung:

- Der Betriebssystem-Kern stellt die grundlegende Infrastruktur für Prozesse bereit. In diesem erfolgt der Hardware-Zugriff, also die physikalischen und organisatorischen Aspekte. Dieser Modus wird *Kernel-Modus* genannt. In Bild 1.3 ist das der schwarze Bereich. Im Kernel-Modus ist jeder beliebige Befehl der CPU zur Ausführung zugelassen, wie zum Beispiel der direkte Adresszugriff in den Speicherbereich. In der CPU wird dies durch ein Steuer- oder Kontrollregister gekennzeichnet. Das Betriebssystem arbeitet üblicherweise im Kernel-Modus und hat somit alle Möglichkeiten, seine definierten Aufgaben zu erfüllen.
- Die eigentlichen Funktionen der Applikationen werden in einem separaten Bereich, dem *Benutzermodus*, erbracht. Hier erfolgt der Zugriff über die logischen, hardwareunabhängigen Kanäle. In diesem Modus sind nicht mehr alle CPU-Befehle erlaubt, so kann nicht auf alle Speicherbereiche zugegriffen werden.

Die Tabelle 1.1 fasst die unterschiedlichen Rechte für den Benutzer- und Kernel-Modus zusammen.

Tabelle 1.1 Kernel- und Benutzermodus

	Benutzermodus	Kernel-Modus
Maschinenbefehle (CPU-Befehle)	Begrenzte Auswahl	Zugriff auf alle
Hardwarezugriff	Nein, nur über Systemcalls	Ja
Zugriff auf Systemcode bzw. Daten	Nein, nur über Systemcalls	Ja

Wie der Kernel-Modus umgesetzt ist und wie er etwas macht, interessiert das Programm nicht. Es ist also wie ein Geheimnis oder eine Trennung der Verantwortlichkeit.

Kernel- und Benutzermodus visualisieren

Ein Betriebssystem zeigt in der Regel an, wie viel Leistung im Kernel- oder Benutzermodus erbracht wurde – unter Windows ist diese Funktion im „Task-Manager“ mit der rechten Maustaste auf dem Diagramm mit „Kernelzeiten aktivieren“ zu aktivieren. Das kann dann wie in Bild 1.4 aussehen. Die eingefärbte Fläche ist die erbrachte Leistung im Kernel-Modus.

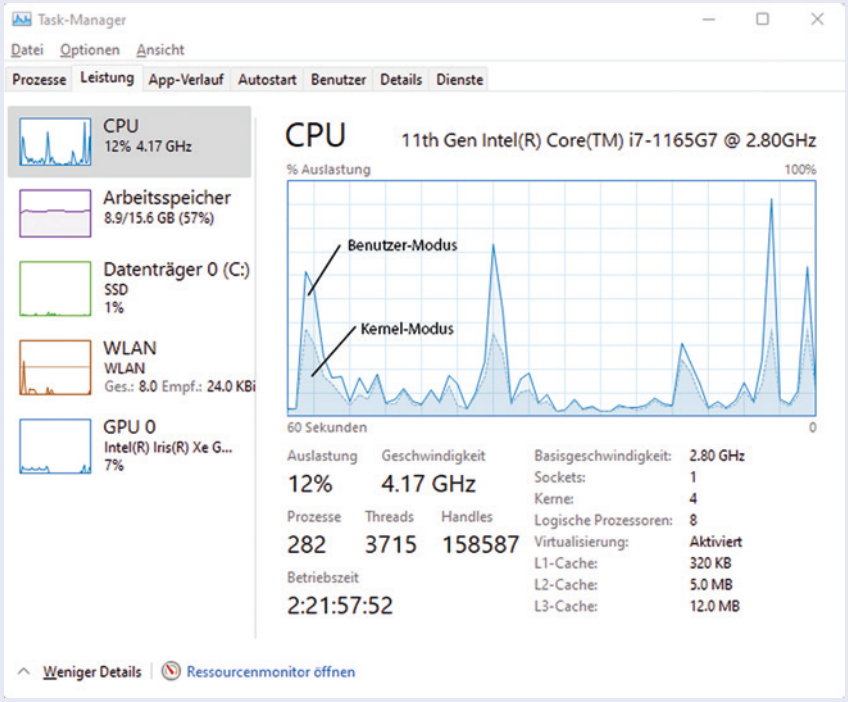


Bild 1.4 Leistung im Kernel- und Benutzermodus unter Windows

Wie machen das andere Systeme?

Ein Betriebssystem kennt als Schutzmechanismus den Kernel- und Benutzermodus. Solche grundlegenden Schutzmechanismen, integriert in die Architektur eines Software-Systems, helfen, die Stabilität und Sicherheit zu gewährleisten.
Es gibt Software-Systeme, die zwischen öffentlichen und privaten APIs unterscheiden und den Zugriff mit Black-/White-Listen prüfen. Bei Systemen, bei denen der Sourcecode zugänglich ist, kann dieser mittels eines Schlüssels geschützt werden. Nur wenn der Entwickler über den entsprechenden Schlüssel verfügt, kann er den Sourcecode ändern.

1.3 Strukturierung

Die Strukturierung von Software und Software-Landschaften gehört zu den wichtigsten Aufgaben der Architektur. Das sagt schon die Definition von Starke [Sta18] aus: *Architektur enthält Strukturen*.

1.3.1 Komponenten

Für diese Strukturierung werden Komponenten verwendet. Der Begriff kommt aus der komponentenbasierten Entwicklung. Die dortige Definition beinhaltet bei einer Komponente einen Rahmen für Struktur, Verknüpfungen, Persistenz, Sicherheit, Versionen usw. Abgeleitet auf die allgemeine Architektur ist eine Komponente ein Teil eines Systems, das eine Schnittstelle hat und austauschbar ist.

Eine Komponente ist ein Teil eines Systems mit folgenden Eigenschaften:

- Komponenten haben eine *definierte Aufgabe* zu erfüllen. Diese kann vom Gesamtsystem unabhängig dokumentiert werden.
- Komponenten haben eine *definierte Schnittstelle*, die den privaten Inhalt kapselt (Geheimnisprinzip) und nur über diese Schnittstelle eine Abhängigkeit aufweist (lose Kopplung).
- Komponenten können anhand der Schnittstellen *isoliert getestet* werden.
- Komponenten sollen immer *austauschbar* sein.
- Komponenten können *unabhängig* voneinander entwickelt werden und im Extremen auch unabhängig ausgeliefert werden.

Zu der obigen Definition ein paar Beispiele:

Definition	Beispiel
... definierte Aufgabe zu erfüllen.	Eine Verschlüsselung einer Zeichenkette
... definierte Schnittstelle, ...	Egal, welche Algorithmen verwendet werden (zum Beispiel abhängig von den Gesetzen im Land) und welche Vor- und Nacharbeiten erfolgen müssen, die Schnittstelle bleibt gleich: Schlüsselmanagement, Zeichenkonvertierungen, Verschleierungen, Hash-Wert-Berechnungen zu Überprüfungen etc.
... isoliert getestet werden.	Ob die Verschlüsselung funktioniert, kann in einem Blackbox-Verfahren jederzeit getestet werden. Vereinfacht gesagt: Bei einer bestimmten Eingabe wird eine bestimmte Ausgabe erwartet, der verschlüsselte Text. Ideal für automatisierte Testfälle.

Definition	Beispiel
... austauschbar sein.	Soll der Algorithmus geändert werden, kann die Komponente ausgetauscht werden. Die Schnittstelle muss gleich definiert bleiben.
... können unabhängig voneinander entwickelt werden ...	Diese Komponente kann im Gesamtsystem unabhängig von anderen Komponenten von einem separaten, dafür spezialisierten Team, entwickelt werden. Einzig die Schnittstelle als Vertrag muss definiert werden.

Komponenten, die eine Software erweitern, werden in manchen Fällen auch als Add-on oder Plug-in bezeichnet. Im Beispiel oben können vielleicht weitere Algorithmen über ein Plug-in-System später hinzugefügt werden. Auch ein Add-on zum Browser ist eine Komponente.

Eine Komponente wird mit dem UML[3]-Symbol in Bild 1.5 dargestellt. Das Symbol zwischen den Komponenten symbolisiert die Schnittstelle. Diese wird öfter auch nur durch einen geraden Strich „abgekürzt“.

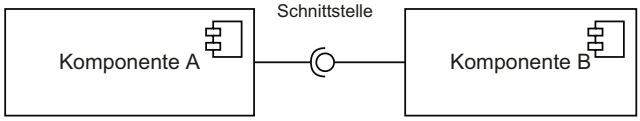


Bild 1.5
Komponente und Schnittstelle

Eine Komponente hat auch eine Struktur in sich selbst. Diese kann aus weiteren Komponenten bestehen. Daraus ergibt sich eine Kapselung wie in Bild 1.6. Die Hauptkomponente delegiert die Schnittstelle an die Fassade, die dann die Komponente B verwendet. Diese verwendet wiederum die Komponente C.

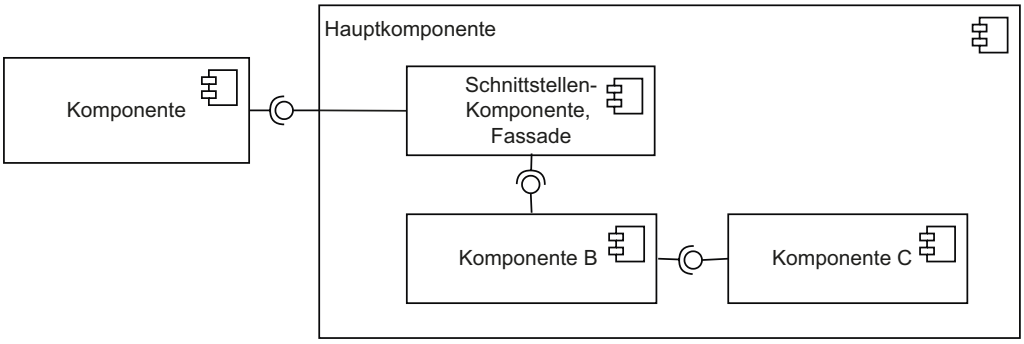


Bild 1.6 Komponentenkopplung

[3] UML: Unified Modeling Language ist eine grafische Notation zur Spezifikation, Konstruktion, Dokumentation und Visualisierung von Software-Teilen und anderen Systemen: *https://de.wikipedia.org/wiki/Unified_Modeling_Language*

1.3.2 Modul

Ein weiteres Gruppierungselement ist das *Modul*. Im Gegensatz zu einer Komponente, die eine geschlossene Einheit bildet, ist ein Modul ein Baustein innerhalb einer Komponente. Die Module sind untereinander abhängiger. Oft wird ein Modul im Kontext der modularen Programmierung verwendet. Sie stellt dann eine Entität dar, die durch die Programmiersprache ausgedrückt werden kann. Technisch kann es in einer Programmiersprache eine Klasse oder Gruppe von Codeeinheiten, die ein Geheimnisprinzip zulassen, sein. Eine Prozedur oder Methode ist kein Modul, da kein Geheimnisprinzip umgesetzt werden kann. Ein Java-Package ist auch kein Modul, da es sich dabei eher um eine Gruppierung handelt. Wichtig ist, dass ein Modul für sich testbar ist. Eine überschaubare Modulgröße und Komplexität hilft bei den Testfällen und führt zu weniger Fehlern.

Die Herausforderung bei Modulen ist, dass dieselbe oder ähnliche Logik in unterschiedlichen Modulen implementiert werden kann. Diese werden dann im gleichen Software-Projekt mehrfach eingesetzt. So ist dann unabsichtlich plötzlich dieselbe Umsetzung mehrfach vorhanden. Dies ist aus Wartungs- und Qualitätssicht zu vermeiden.

Aus meiner Sicht das wichtigste ist, dass eine Modularisierung des Systems überhaupt durchgeführt wurde. Im Unterschied zu einer Komponente ist ein Modul eine „Hierarchieebene tiefer“ angesiedelt. Die Komponente kombiniert die Funktionalitäten von Modulen zu fachspezifischen Diensten. Im Sprachgebrauch werden Module und Komponenten häufig undifferenziert verwendet, was schnell zu Unklarheiten führen kann.

1.3.3 Bibliotheken und Frameworks

Entwickler sprechen oft von *Bibliotheken*. Diese beinhalten eine Sammlung von Modulen, die in einer gekapselten Weise verwendet werden. Eine wichtige Eigenschaft von Bibliotheken ist, dass sie wiederverwendet werden können und in einem *nichtflüchtigen* Speicher eingesetzt werden. Eine Bibliothek kann zum Beispiel eine Ansammlung von mathematischen Funktionen sein oder eine Schnittstelle zu Geräten kapseln, so dass der Entwickler diese einfacher ansprechen kann. Eine Bibliothek wird von einem Entwickler benutzt, er steuert also den Ablauf.

Ein *Framework* kann in Form von einer oder mehreren Bibliotheken bereitgestellt werden. Der Unterschied besteht darin, dass ein Framework eine weitgehend vollständige, in sich gekapselte, aber anpassungsfähige und erweiterbare Lösung darstellt. Der Entwickler, der das Framework einsetzt, kann sich somit auf die domänen- und projektspezifischen Probleme konzentrieren. Bei einem Framework wird die Steuerung umgekehrt angewendet: Das Framework steuert den Ablauf und der Entwickler gibt Funktionalitäten hinzu. Diese Funktionalität wird beim Framework registriert und später von diesem aufgerufen. Dies wird *Inversion of Control* (IoC) genannt.