HANSER

Leseprobe

zu

Softwarearchitektur pragmatisch

von Philipp Friberg

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

Weitere Informationen und Bestellungen unter
https://www.hanser-kundencenter.de/fachbuch/artikel/9783446473706
sowie im Buchhandel

© Carl Hanser Verlag, München

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.

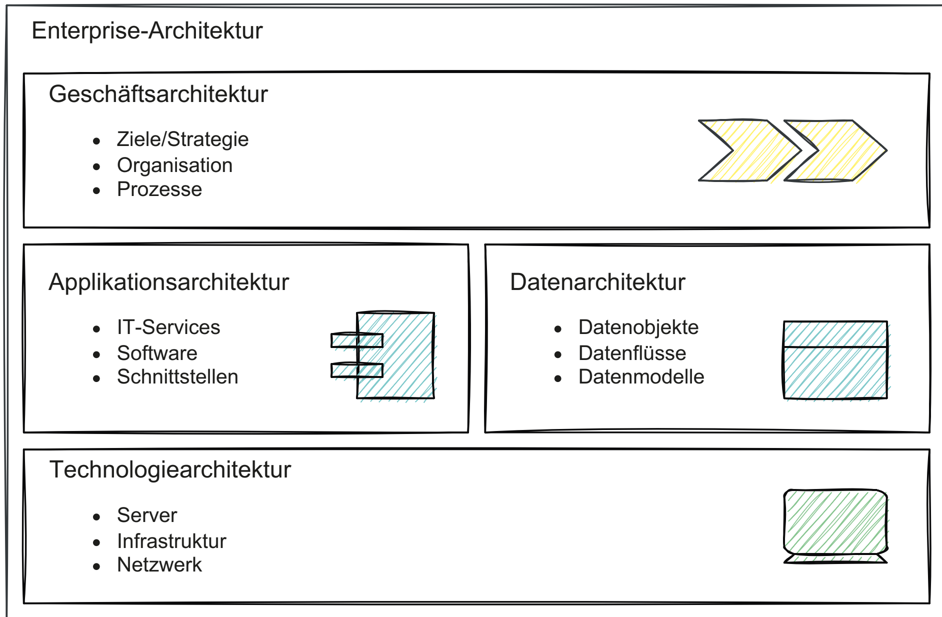


Bild 1 Einordnung der Architekturen

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-12abc-8xyz9

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.

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*.

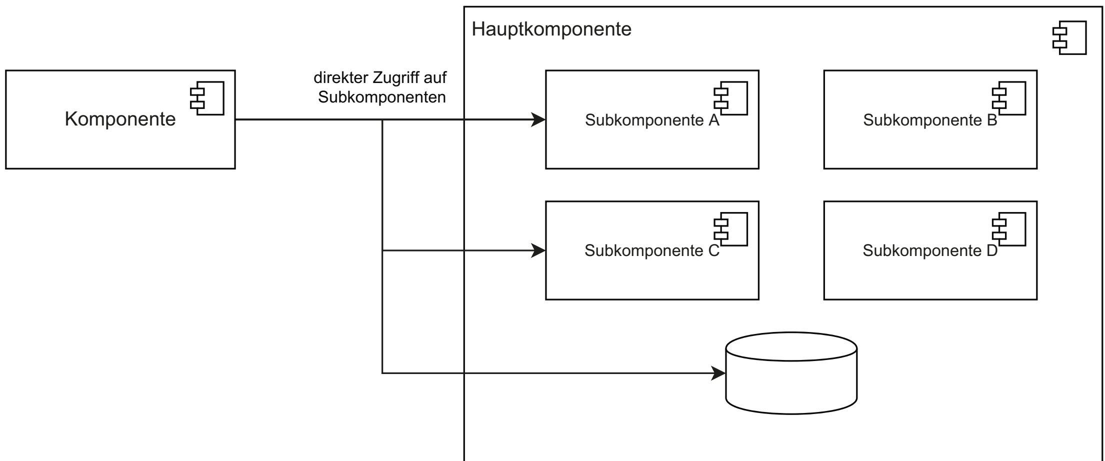
Bei Software allgemein gibt es das Prinzip, dass das Innenleben eines Bausteins vor der Außenwelt verborgen bleiben soll. Folgend nennen wir solche Bausteine *Komponenten*. Dafür wird eine öffentliche Schnittstelle definiert, die beschreibt, was die Komponente macht. Das *Wie* dürfen wir getrost vor 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.



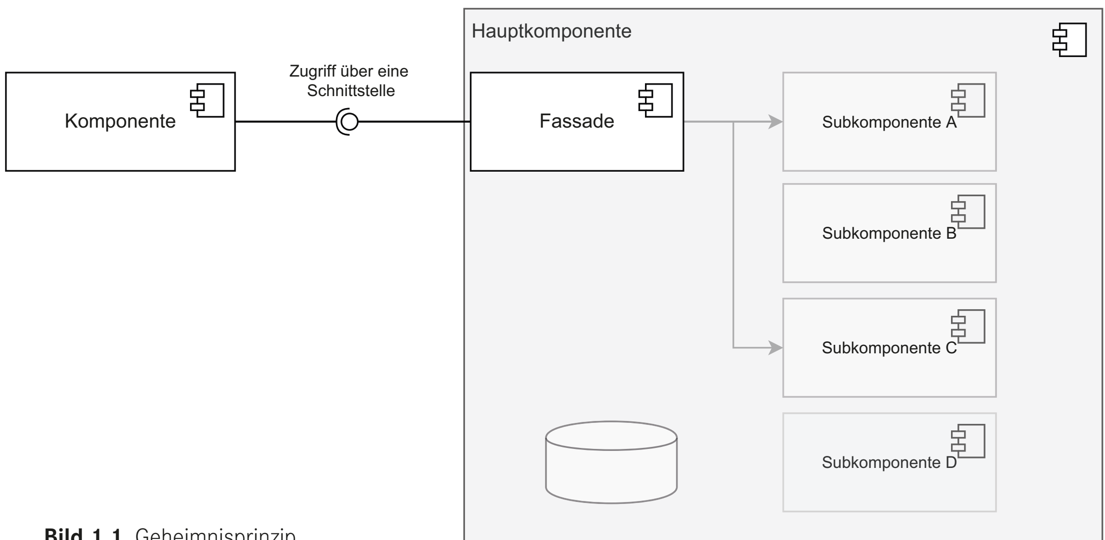


Bild 1.1 Geheimnisprinzip

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.

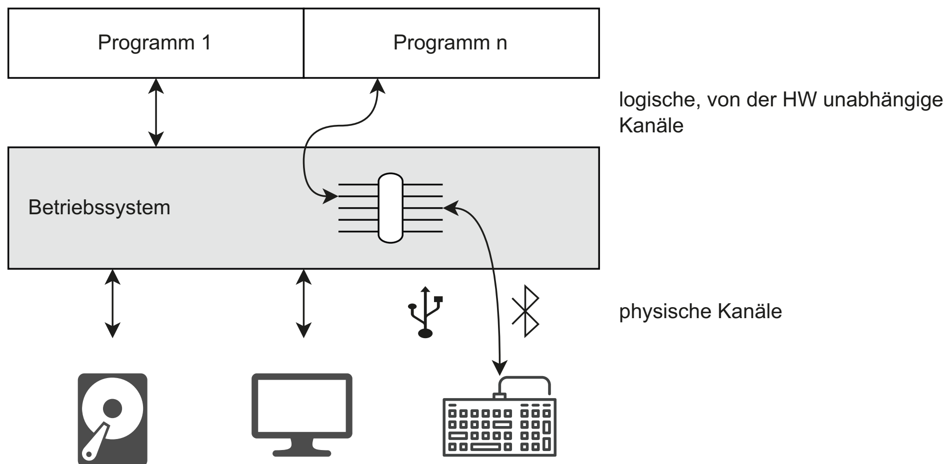


Bild 1.2 Logische und physische Kanäle

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.

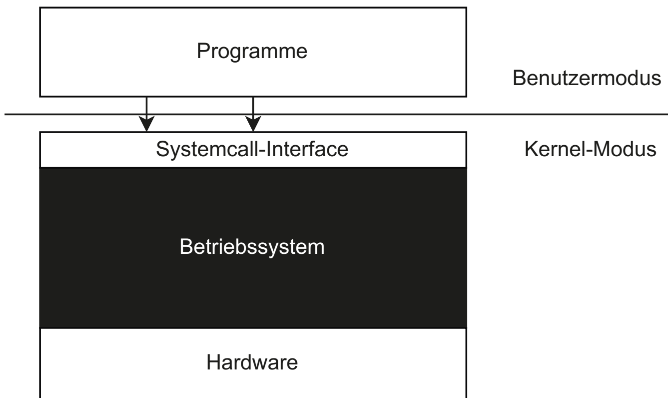


Bild 1.3 Geheimnisprinzip beim Betriebssystem

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.

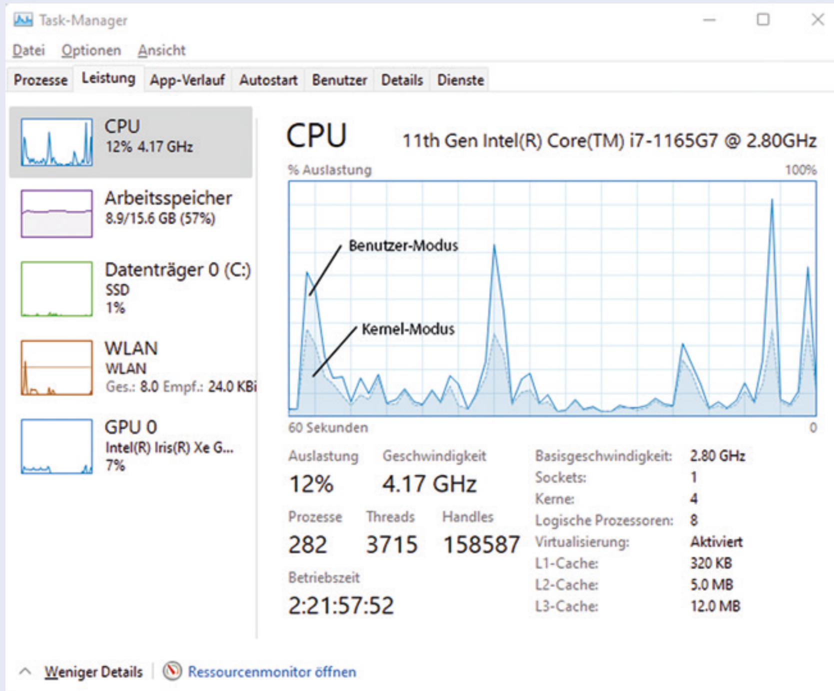


Bild 1.4 Leistung im Kernel- und Benutzermodus unter Windows

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“.

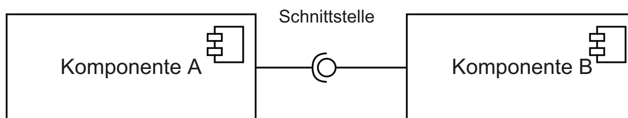


Bild 1.5 Komponente und Schnittstelle

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.

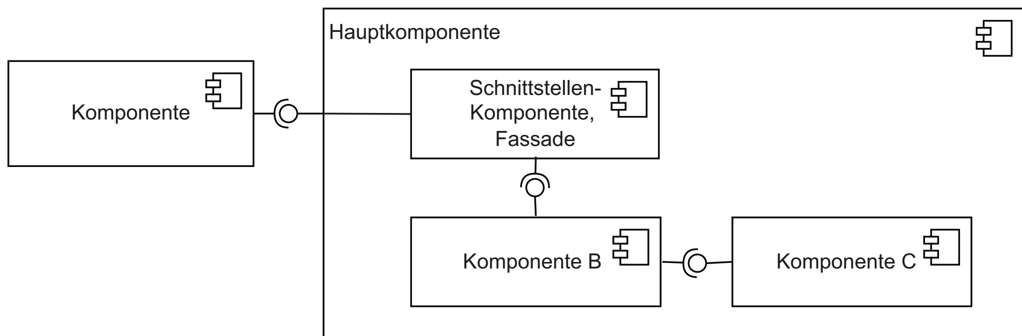


Bild 1.6 Komponentenkopplung

[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*

5.3 Modellierungssprache ArchiMate

Für die Dokumentation der Unternehmensarchitektur fehlt uns noch ein passender Ordnungsrahmen. Die Modellierungssprache ArchiMate®[11] ist ein Ordnungsrahmen für die grafische Beschreibung der Unternehmensarchitektur. Sie wird, wie das TOGAF, von *The Open Group* weiterentwickelt und unterstützt sämtliche Phasen des TOGAF ADM. Ein Vorteil von ArchiMate ist, dass es ein von Tool-Herstellern unabhängiger Standard ist. Mit der Version 3 gewinnt er immer mehr an Bedeutung und wird von verschiedenen Tools unterstützt. Die BOC Group[12] nennt folgende Punkte als gute Gründe für den Einsatz der Modellierungssprache ArchiMate:

- *Eine bewährte Modellierungssprache für Unternehmensarchitekturen.*
- *Adressiert sowohl die Interessen der Fachseite als auch die der IT.*
- *Ermöglicht eine Modellierung gemäß der TOGAF Architecture Development Method.*

Die Modellierungssprache ArchiMate gliedert sich in fünf Sichten, die sich den TOGAF-Phasen zuordnen lassen (Bild 5.6).

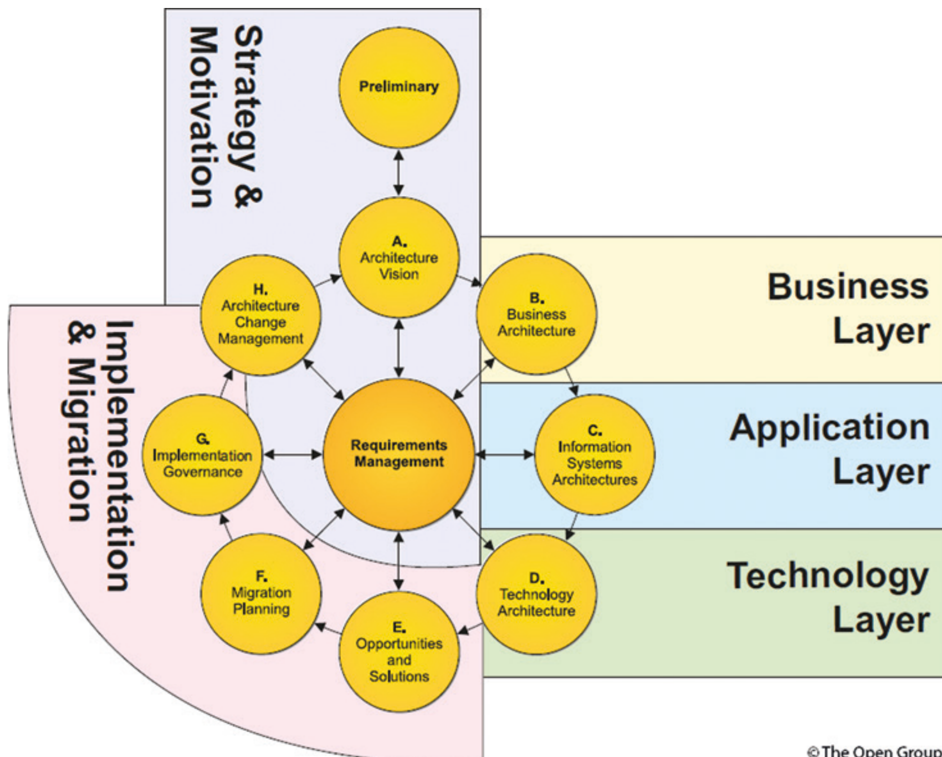


Bild 5.6 ArchiMate und die TOGAF-Phasen[13]

[11] *https://www.opengroup.org/archimate-home*
[12] Die BOC Group ist ein Hersteller von Architektur-Tools. *https://de.boc-group.com/standardsbest-practices/*
[13] Quelle: The Open Group, *www.opengroup.org/library/n190*

Die Schichten *Technologie*, *Applikation* und *Geschäft* werden als *Core-Layer* bezeichnet, da diese die meistbenutzten sind. Diese gibt es seit der ersten Version von ArchiMate. Mit der Version 2 kamen die *Implementierung und Migration* dazu. In der Version 3 folgten die *Strategie und Motivation*, die oft auch als zwei separate Schichten dargestellt werden. Als Software-Architekt dürften Sie sich in den *Core-Layer* am wohlsten fühlen und deshalb konzentrieren wir uns auf diese. Danach werde ich die anderen Layer nur kurz ansprechen.

Um ArchiMate richtig zu verstehen, hilft es, die Syntax, die Grammatik, zu verstehen. Verwenden wir folgenden Satz:

Im Programm Word schreiben wir ein Buch.

Dieser Satz beinhaltet das Subjekt **Programm Word**, das Verb **schreiben** und das Objekt **Buch**, also **Subjekt – Verb – Objekt**, oder in ArchiMate:

aktive Elemente – Verhaltenselemente – passive Elemente

Aktive Elemente sind Strukturelemente, beschreiben also etwas, das statisch ist. Das kann ein Stück Software, eine Komponente, eine Schnittstelle oder ein Gerät sein. *Verhaltenselemente* (behavior im Englischen) stellen ein Verhalten dar, das von einem aktiven Element ausgeführt werden kann, also eine Funktion, einen Prozess oder einen Service. In unserem Beispiel braucht es das **Programm Word**, um die Funktion **schreiben** auszuüben. *Passive Elemente* werden von einem Verhaltenselement benutzt. Sie haben kein eigenständiges Verhalten, sie sind etwas, das verwendet oder erzeugt wird. Das kann ein Dokument, ein Datensatz, eine Datei oder das **Buch** aus dem Beispiel sein. Vielleicht hilft die folgende Frage:

Wer macht was? Wer = aktiv, macht = Verhalten, was = passiv.

Reference Cards für ArchiMate

Als Hilfsmittel dienen die Reference Cards der Modellierungssprache ArchiMate: *https://publications.opengroup.org/n190*. Ich finde diese sehr hilfreich, um sich in ArchiMate einzuarbeiten und die folgenden Diagramme zu verstehen.

Bevor wir in die Technologie-, Applikations- und Geschäftsebenen abtauchen, sollten wir uns zuerst mit den zentralen ArchiMate-Begriffen *View* und *Viewpoint* auseinandersetzen.

5.3.1 View und Viewpoint

Ein wichtiges Merkmal der Modellierungssprache ArchiMate ist die *View*, die Sicht, und der *Viewpoint*, der Standpunkt. Was ist der Unterschied? Wie in der Einleitung erläutert, muss sich der Architekt bewusst sein, dass nicht alle Stakeholder alles interessiert! Deshalb gibt es die *View*. Das ist ein Teil der Architekturbeschreibung, die eine Reihe von zusammenhängenden Anliegen anspricht und auf bestimmte Stakeholder zugeschnitten ist. Ein *Viewpoint* spezifiziert eine Ansicht. Sie schreibt die Konzepte, Modelle, Analysetechniken und Visualisierungen vor, die in der Sicht angewendet werden. Somit gilt, dass eine Sicht (View) das ist, was gesehen wird, und der Standpunkt (Viewpoint) ist der Punkt, von dem aus die Sicht gesehen wird. Anders formuliert:

Ein Viewpoint ist eine Auswahl einer relevanten Teilmenge der Modellierungssprache ArchiMate und deren Beziehungen. Eine Sicht sind ein oder mehrere visuelle Diagramme, die einen Teil der Architekturmodelle darstellen, unter Verwendung der Konzepte und Beziehungen des entsprechenden Viewpoints.

Wichtig ist zu verstehen, dass Views, die dem Stakeholder relevante Aspekte der Architektur aufzeigen, das Modell nicht verändern. Genau dieser Aspekt hat mir in der Praxis sehr geholfen. So erstellten wir jeweils Stakeholder-bezogene Sichten der relevanten Systeme und deren Beziehungen aus dem bestehenden Modell.

Vertiefen wir uns folgend in verschiedene Sichten mit verschiedenen Viewpoints der Ist-Architektur. Im nächsten Kapitel werden wir das Modell um die neuen Komponenten erweitern. Ich empfehle, die oben erwähnten *Reference Cards* sowie die offizielle Dokumentation für die Vertiefung bereitzuhalten.

5.3.2 Applikations-Layer Grundmuster

Anhand des Applikations-Layer möchte ich das Grundkonzept der vorgestellten Grammatik zeigen. Im ADM wäre der Geschäfts-Layer der nächste Schritt. Aber ich denke, da Sie von einer Teilarchitektur herkommen dürften, ist es für Sie leichter, mit diesem Layer eine neue Modellierungssprache kennenzulernen.

Bild 5.7 zeigt das Grundmuster. Die Elemente auf dem Applikations-Layer werden oft blau dargestellt. Die aktiven und passiven Elemente sind eckig und die Verhaltenselemente haben abgerundete Ecken. Die Form und die Farbe helfen sehr, sich in einem Diagramm zurechtzufinden. Jedes Element hat ein Symbol und kann auch direkt mit dem Symbol selbst dargestellt werden.

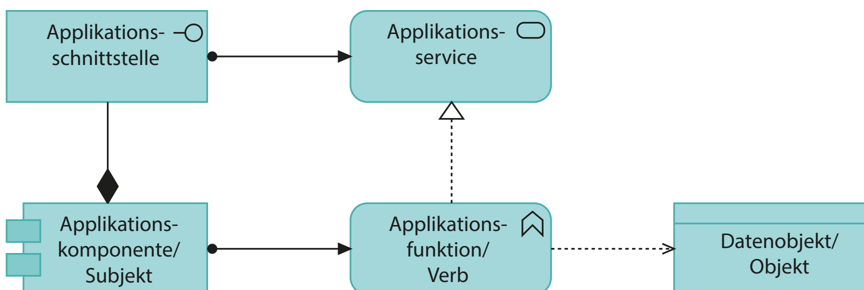


Bild 5.7 Grundmuster Applikations-Layer

Das Grundmuster beinhaltet fünf Elemente und fünf Beziehungen. Schauen wir uns diese genauer an.

Eine **Applikationskomponente** ist eine Komponente oder Applikation, die etwas ausführt. Komponenten ergeben eine Struktur in der IT, sie selbst haben aber zu diesem Zeitpunkt noch nichts gemacht. Wichtig ist, dass diese austauschbar sind. In unserem Grammatikbeispiel wäre die Komponente die **Applikation Word**.

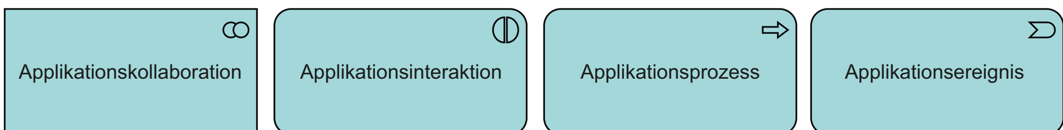
Eine **Applikationsfunktion** stellt ein automatisiertes Verhalten einer Applikationskomponente dar. Hier kommt also Bewegung in die Struktur rein. Im Beispiel stellt es das Verb **schreiben** dar, also eine Funktion von Word, etwas schreiben zu können.

Ein **Datenobjekt** ist etwas, das wir bearbeiten, erzeugen und löschen können, also ein Objekt, mit dem wir etwas machen. Das kann eine Kundenbestellung auf der Datenbank sein, aber es ist nicht die Datenbank selbst! Auf der anderen Seite sind es aber auch keine Variablen eines Programms, das wäre zu feingranular. In der Unternehmensarchitektur sind die Objekte und nicht die Attribute wichtig. Im Beispiel ist das **Buch** ein Datenobjekt. Als Grundregel gilt: Ein Objekt sollte auch außerhalb der Komponente vorhanden sein.

Die **Applikationsschnittstelle** repräsentiert einen Zugriffspunkt, der einer anderen Applikationskomponente zur Verfügung gestellt wird. Es muss sich nicht um eine technische Schnittstelle handeln, es kann auch eine Benutzerschnittstelle sein – in unserem Beispiel die grafische Benutzeroberfläche zum Benutzer, so dass das Buch in Word geschrieben werden kann. Technisch ist es in der Regel ein API, also eine wohlweislich definierte Schnittstelle. Es kann aktiv etwas an einer Applikationskomponente verändern, beeinflussen.

Ein **Applikationsservice** ist ein von außen zugreifbares Applikationsverhalten. Die Applikation offeriert einen Service, ein Verhalten, einer anderen Applikation oder dem Geschäftsprozess. Es ist technischer Natur.

Es gibt weitere Elemente auf dem Applikations-Layer, die wir erst später oder gar nicht benutzen werden:



- Mehrere Applikationskomponenten zusammen stellen eine *Applikationskollaboration* dar. Dies ist ein aktives Element.
- Die *Applikationsinteraktion* repräsentiert ein kollektives Verhalten. Es stellt das Verhalten zum passiven Element der Kollaboration dar.
- Der *Applikationsprozess* ist eine Sequenz von Applikationsverhalten. Ich verwende lieber die Applikationsfunktion. Im nächsten Kapitel werden wir ein Beispiel sehen, in dem ich dieses Element verwendet habe.
- Das *Applikationsereignis* wird für Statusänderungen verwendet.

Es ist an der Zeit, sich über die Relationen (Beziehungen) aus Bild 5.7 Gedanken zu machen. Betrachten wir dazu zusätzlich das Bild 5.8.

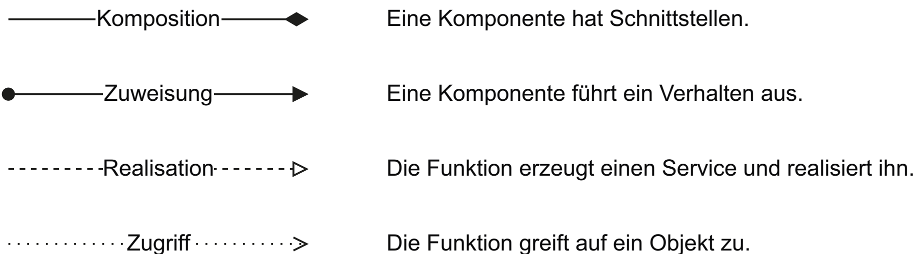


Bild 5.8 Relationen im Grundmuster

- Wenn Sie UML kennen, dürfte Ihnen die *Komposition* bekannt vorkommen: Eine Komponente hat mehrere Schnittstellen, diese können aber nicht ohne die Komponente existieren.
- Bei der *Zuweisung* führt das aktive Element, z. B. die Komponente, ein Verhaltenselement aus. Der Punkt wird auf der aktiven Seite platziert.
- Bei der *Realisation* erzeugt das Element ohne den Pfeil das Element, bei dem der Pfeil dargestellt wird, und stellt es somit zur Verfügung.
- Der *Zugriff* bedeutet, dass das Verhaltenselement auf das Datenobjekt zugreift – lesend, schreibend, erzeugend oder löschend.

Dieses Grundmuster werden wir in jedem Layer antreffen. Folgend gehen wir einzeln die Layer durch, auch nochmals den Applikations-Layer, und besprechen die Ist-Situation von *Scotland Trading*.

5.3.3 Geschäfts-Layer

In ADM ist der nächste Schritt die Geschäftsarchitektur. Diese wird in der Modellierungssprache ArchiMate der Geschäfts-Layer genannt. Um es vorwegzunehmen, ArchiMate hat nicht den Anspruch, eine Sprache zu sein, mit der Geschäftsprozesse vollumfänglich modelliert werden können. Die Idee ist mehr, dass die Basiselemente zur Verfügung gestellt werden, so dass eine Verlinkung mit anderen Modellen, zum Beispiel BPMN-Modellen, erfolgen kann. Da wir Pragmatiker aus der Technik sind, konzentriere ich mich auf die Elemente, die uns helfen, die Applikationsarchitektur verständlicher darzustellen.

Halten wir uns an das Grundmuster aus dem vorherigen Abschnitt und erweitern es analog mit den Geschäftsobjekten. Wir erhalten das Bild 5.9.

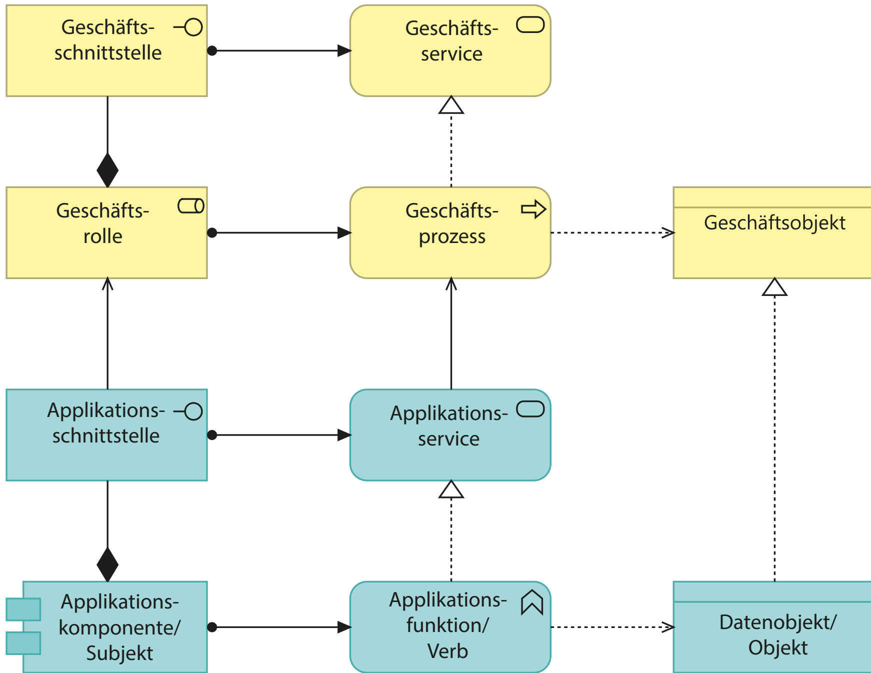
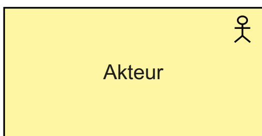


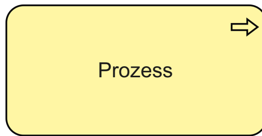
Bild 5.9 Grundmuster Geschäftsarchitektur

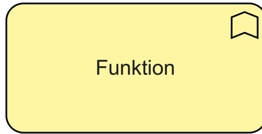
Gewisse der fünf neuen Elemente, die oft gelb dargestellt werden, erkennen wir aus der Applikationsarchitektur wieder, andere sind neu:



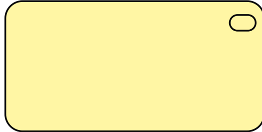
Die **Geschäftsrolle** ist eine abstrakte Einheit, die einen Geschäftsprozess ausführt. Die Rolle ist der abstrakte Akteur. ArchiMate kennt auch einen **Geschäftsakteur**, analog zum Akteur in UML. Akteure sind an einem Szenario beteiligt, z. B. der Lagermitarbeiter, der rüsten möchte. Bei *Scotland Trading* wäre die Rolle die Beschaffung (Einkauf) oder der Verkauf. Der Akteur wäre ein Produktmanager oder der Verkäufer.

Die **Geschäftsschnittstelle** ist ein Zugangspunkt für einen Geschäftsservice. Ein Zugangspunkt für die Geschäftsrolle Servicedesk kann zum Beispiel Telefon, Mail und Ticket-System sein.

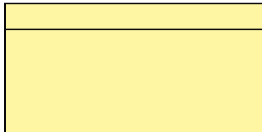




Auf der Verhaltensseite gibt es den **Geschäftsprozess**. Er ist eine Abfolge von Aktivitäten und führt zu einem bestimmten Ergebnis. Ohne eine Geschäftsrolle, die den Prozess ausführt, kann der Prozess nicht sein. Es gibt auch die **Geschäftsfunktion**, die sehr ähnlich zum Prozess ist. Sie wird häufig in Modellen, die im Internet zu finden sind, als Synonym verwendet. Wir verwenden die Funktion als übergeordnetes Element zu einem Prozess. Beispiel: Die Verkaufsfunktion beinhaltet die Prozessschritte Angebot, Verkauf, Lieferung und Fakturierung.



Ein **Geschäftsservice** stellt die Daseinsberechtigung des Prozesses dar. Wenn kein Service im Unternehmen erbracht werden muss, braucht es auch den Prozess nicht. Auf diesen kann von außen zugegriffen bzw. er kann beansprucht werden.



Auf der passiven Seite gibt es das **Geschäftsobjekt**. Im Unterschied zum Datenobjekt stellt es eher ein Konzept dar, also zum Beispiel eine Rechnung.

Zwischen dem Applikations-Layer und dem Geschäfts-Layer (Bild 5.9 Mitte) finden wir die neue Relation *Dient (Serving)*, einen einfachen Pfeil. Gemeint ist damit, dass der Applikationsservice vom Geschäftsprozess verwendet wird, also dient der Service dem Prozess. Damit wird die Abhängigkeit zwischen den Elementen verdeutlicht. Vor ArchiMate 3 wurde diese Relation *„is used by"* genannt, was mir persönlich einfacher gefallen ist. Die Lesart wäre dann, dass der Service vom Geschäftsprozess benutzt wird.

Folgend wollen wir den Geschäfts-Layer von *Scotland Trading* selbst modellieren. Dazu verwende ich das Open-Source-Tool *Archi*[14]. Das Tool ist eine Einzelplatzlösung, basierend auf Eclipse und für verschiedene Plattformen verfügbar. Da es kostenlos ist, laden Sie es doch herunter und sammeln Sie die ersten Erfahrungen. Im Unternehmensbereich gibt es verschiedene Tools. Sollten Sie noch eines evaluieren müssen, so beachten Sie dazu den Gartner® Magic Quadrant™ for Enterprise Architecture Tools[15]. Egal, welches Tool Sie verwenden, denken Sie an den Teil *Architecture Content* von TAGAF. ArchiMate hat ein definiertes *ArchiMate Model Exchange File Format*[16] definiert, das die Interoperabilität und somit einen späteren Tool-Wechsel erleichtern sollte. Gewisse Tools haben bewusst den ArchiMate-Standard mit proprietären Elementen und Relationen aufgeweicht. Ich bin kein Fan davon, aber das müssen Sie im Unternehmen klären, ob es sinnvoll ist, solche Modellierungselemente zu verwenden.

Die folgenden Modelle finden Sie im Buch-Download und können Sie im Programm *Archi* nachvollziehen. Zur Vereinfachung habe ich im Archi-Modell bei den Views die Bildnummer aus dem Buch angegeben.

[14] *https://www.archimatetool.com/*
[15] *https://content.ardoq.com/gartner-magic-quadrant-for-enterprise-architecture-tools*
[16] *https://www.opengroup.org/xsd/archimate/3.0/*

Die Ist-Architektur des Geschäfts-Layers ist in Bild 5.10 abgebildet. Der Service, den das Geschäft anbietet, ist die *Preisfestlegung*. Diese beinhaltet die Funktion *Preisbildung*, in der zuerst der marktübliche *Preis „gegoogelt“* wird. Zusammen mit dem Einkaufspreis werden dann die *Verkaufskonditionen festgelegt*. Diese Funktion wird von der Rolle *Verkauf* ausgeübt. In dieser Rolle ist der *Produktmanager* die verantwortliche Person.

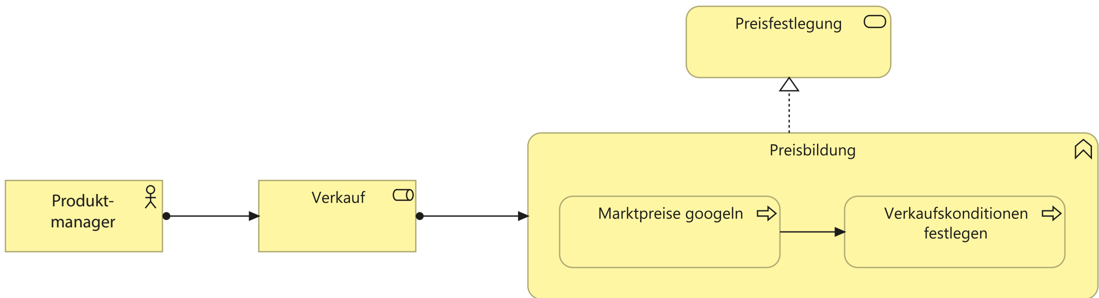


Bild 5.10 Geschäfts-Layer, Ist-Situation, Preisfestlegung

Ausgehend davon modellieren wir nun zusammen Schritt für Schritt die Soll-Architektur. In *Archi* legen Sie dazu eine neue Sicht an. Sie sehen unter Views (links in der Mitte in Bild 5.11) die verschiedenen Sichten.

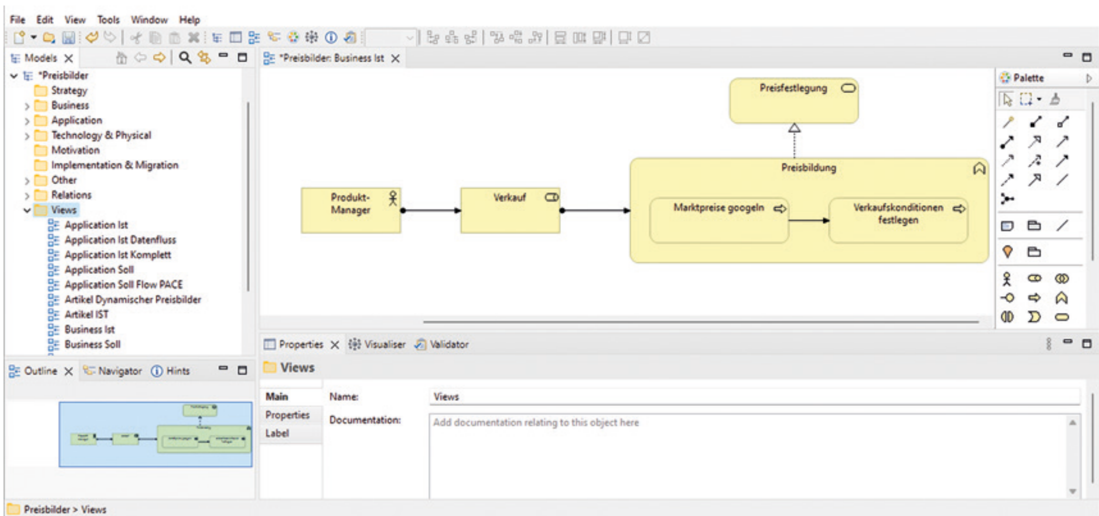
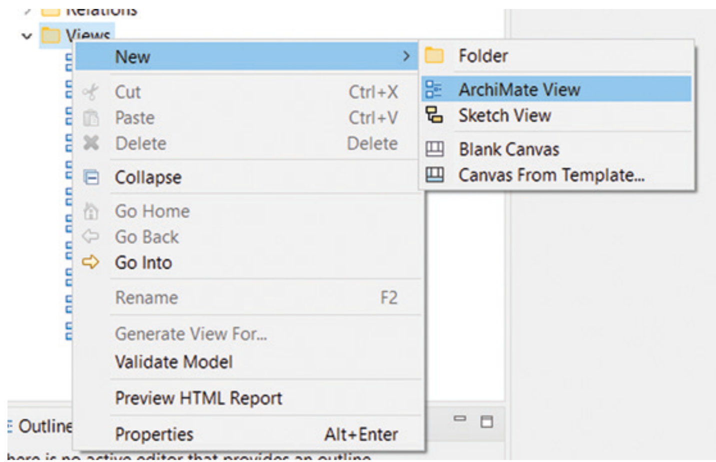
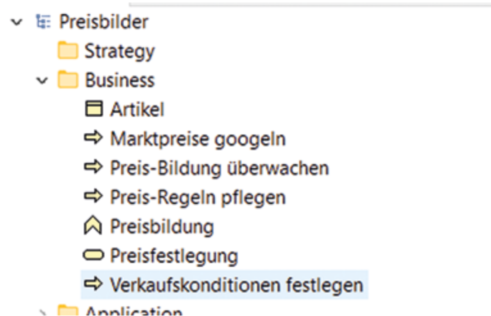


Bild 5.11 Programm Archi

Erstellen Sie mit der rechten Maustaste eine neue View:



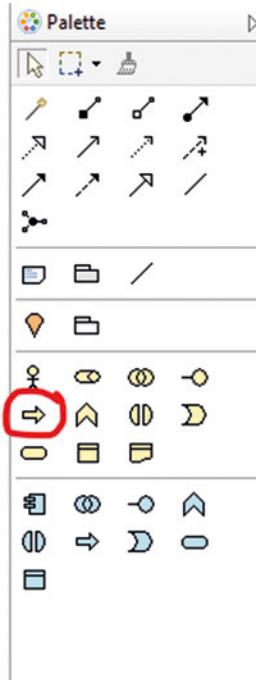
Danach können Sie die bestehenden Geschäftselemente in die View ziehen. In der Baumstruktur unter **Business** finden Sie diese Elemente wieder:



Mittels Drag-and-Drop können Sie alle Elemente aus Bild 5.11 in die Sicht hineinziehen, außer *Marktpreise googeln*, das wird ja unser Crawler übernehmen. Dafür brauchen wir zwei neue Prozessschritte:

1. Die Regeln müssen konfiguriert werden.
2. Die Preisbildung muss überwacht werden.

Ziehen wir also das Element **Business Process** von der **Palette** rechts in die View hinein.



Wir benennen sie **Verkaufskonditionen festlegen** und **Preis-Regeln pflegen**, so dass wir Bild 5.12 erhalten (ohne die Relationen). Im ersten Schritt werden im ERP-System die Konditionen gepflegt und im zweiten Schritt im Regelwerk die Regeln.

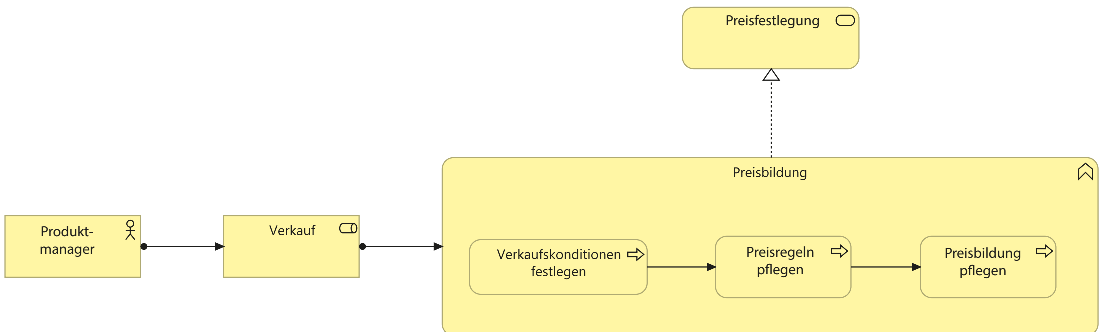
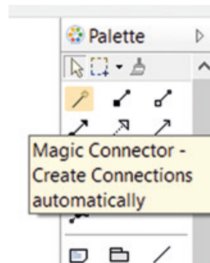


Bild 5.12 Geschäfts-Layer, Soll-Situation Preisfestlegung

Über die Relationen zwischen den Elementen haben wir noch nicht gesprochen. In *Archi* finden Sie diese rechts oben in der Palette:



Mit dem **Magic Connector** können Sie zwei Elemente verbinden und *Archi* schlägt Ihnen erlaubte Relationen vor. Wir brauchen in dieser View die Relationen in Bild 5.13.

Realisation	Die Preisfestlegung wird durch die Preisbildung realisiert.
Komposition	Die Preisbildungsfunktion besteht aus Prozessen.
Trigger	Ein Prozess stößt einen neuen Prozess(-Schritt) an.

Bild 5.13 Benutzte Relationen im Geschäfts-Layer

Wir erhalten das komplette Modell in Bild 5.12. Wir haben die Funktion mit dem Prozess verschachtelt. Dies ist in ArchiMate möglich, jedoch mit Vorsicht zu verwenden, da die Relation zwischen den Elementen nicht mehr sichtbar ist. Für den Leser des Diagramms kann es so aber einfacher zu lesen sein.

Ich möchte noch auf einen interessanten Fall von *Scotland Trading* eingehen. Die neue Funktion *Rabatt-Deal* erarbeitet mit den Lieferanten sogenannte Rabatt-Deals, also zum Beispiel ein Set-Produkt: Kaufe einen Kilt, erhalte einen Flachmann gratis. Ist einer zu Stande gekommen, triggert dieser die Funktion *Preisbildung*. Im Unternehmen gibt es ein *Controlling*, das die Regeln vorgibt, wie solche Deals entstehen dürfen und wie die Preisbildung erfolgen darf, zum Beispiel die gesetzlichen Regeln von Statt-Preisen. Wir haben also die drei Funktionen: *Rabatt-Deal*, *Preisbildung*, *Controlling*.

Zwischen *Rabatt-Deal* und *Preisbildung* wird das Geschäftsobjekt *Artikel* ausgetauscht, hier insbesondere mit den Rabattinformationen. In Bild 5.14 ist dieser Austausch mit einem Fluss dargestellt. In ArchiMate wird dieser mit einer gestrichelten Linie mit Pfeil abgebildet. Der oben erwähnte Trigger ist eine ausgezogene Linie mit Pfeil. Zwischen dem *Controlling* und der *Preisbildung* fließt das Datenobjekt *Regeln für Preisbildung*, das die Regeln für eine Preisbildung festhält. Einen Trigger gibt es nicht, das Controlling ist in diesem Prozess nicht explizit integriert, die Befolgung traut das Unternehmen den Mitarbeitern zu.

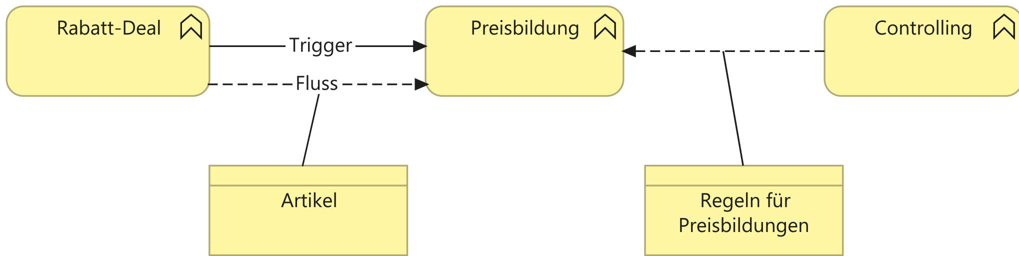


Bild 5.14 Geschäftsfunktionen und Geschäftsobjekt

Die Relation *Fluss* darf nur innerhalb einer Ebene erfolgen. Neu in ArchiMate 3 ist die Möglichkeit, Objekte direkt einem Fluss anzuhängen. In den Versionen zuvor musste über die „Access"-Relation gearbeitet werden, was die Diagramme sehr unübersichtlich machte. Ich finde diese neue Art intuitiv.

Zusätzlich gibt es im Geschäfts-Layer weitere Elemente:

- Wenn mehrere aktive Geschäftselemente zusammenarbeiten, dann kann das Element *Geschäftskollaboration* verwendet werden.
- Die *Geschäftsinteraktion* ist eine Gruppe von Geschäftsverhalten, die durch zwei oder mehr Geschäftsrollen ausgeführt wird.
- Ein *Geschäftsereignis* tritt intern oder extern ein und beeinflusst das Geschäftsverhalten.
- Auf der passiven Seite gibt es zusätzlich den *Vertrag*, die *Repräsentation* und das *Produkt* als Verbundelement.

5.3.4 Ableitungen

Bei größeren Modellen können die Sichten sehr unübersichtlich werden, wenn diese immer nach dem Muster in Bild 5.9 dargestellt werden. Deshalb sollten wir uns überlegen, welche Relationen abgeleitet werden können und so unsere View für den Leser vereinfachen können. Betrachten wir das Grundmuster aus dem Applikations-Layer in Bild 5.15. Einer Komponente ist eine Funktion zugewiesen, die einen Service realisiert. Somit realisiert indirekt eine Komponente einen Service. Als Beziehung verwenden wir immer die schwächste Beziehung. Im Standard von ArchiMate 3.1, im Anhang B[17], wird die Stärke der strukturellen Beziehungen wie folgt definiert:

1. Realisation (schwächste Beziehung)
2. Zuweisung (Assignment)
3. Aggregation
4. Komposition (stärkste Beziehung)

Somit können wir zwischen Komponente und Service die Beziehung *Realisation „a"* (rot) verwenden (Bild 5.15). Wir verlieren damit aber die Information, in welcher Funktion dieser Service abgebildet ist. Vielleicht sagt es uns indirekt der Service-Name ...

[17] *https://pubs.opengroup.org/architecture/archimate3-doc/apdxb.html*

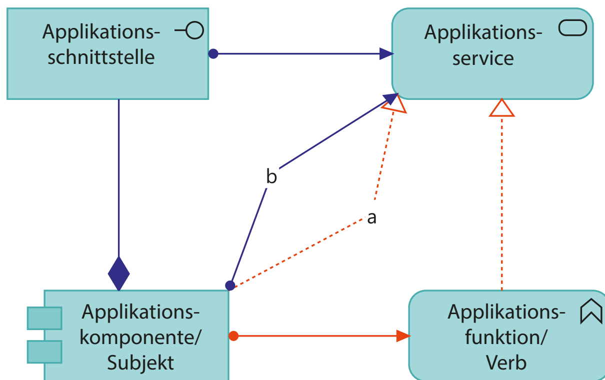


Bild 5.15 Ableitungsbeispiel

Wollen wir den Weg über die Schnittstelle gehen, ergibt sich als schwächste Beziehung die *Zuweisung „b"* (blau).

Bei den Abhängigkeitsbeziehungen ist die Stärke der Beziehung folgendermaßen definiert:

1. Association (schwächste Beziehung)
2. Influence (Einfluss bei dem Motivations-Layer)
3. Zugriff
4. Dient (Serving) (stärkste Beziehung)

Ein Beispiel der Abhängigkeitsbeziehungen zeigt Bild 5.16.

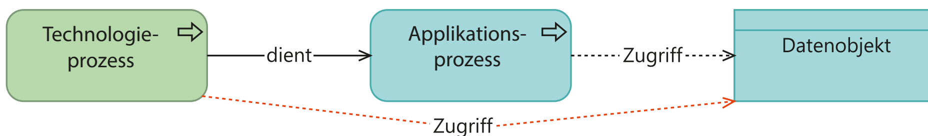


Bild 5.16 Abhängigkeitsbeziehungen

Im Anhang B.5 des Standards oder in *Archi* unter **Help, ArchiMate Relationships** gibt es eine Tabelle, die alle Beziehungen zwischen den Elementen aufzeigt.

Relationen

Die Relationen sind nicht immer so, wie Sie denken. Aus Erfahrung müssen Sie sich mit diesen vertraut machen, sonst sind sie schnell verkehrt herum gezeichnet. Ein Geschäftsprozess dient nicht einer Rolle, sondern eine Rolle führt einen Prozess aus. In Bild 5.17 habe ich die wichtigsten Relationen dokumentiert.

Für „Ableitungen (Abkürzungen)" gilt folgende Reihenfolge, wobei die letzte (schwächste Beziehung) gewinnt:

Komposition – Aggregation – Zuweisung – Realisierung – Dient – Zugriff – Influence – Association

Beachten Sie, dass bei Ableitungen die Richtung nicht geändert werden darf.

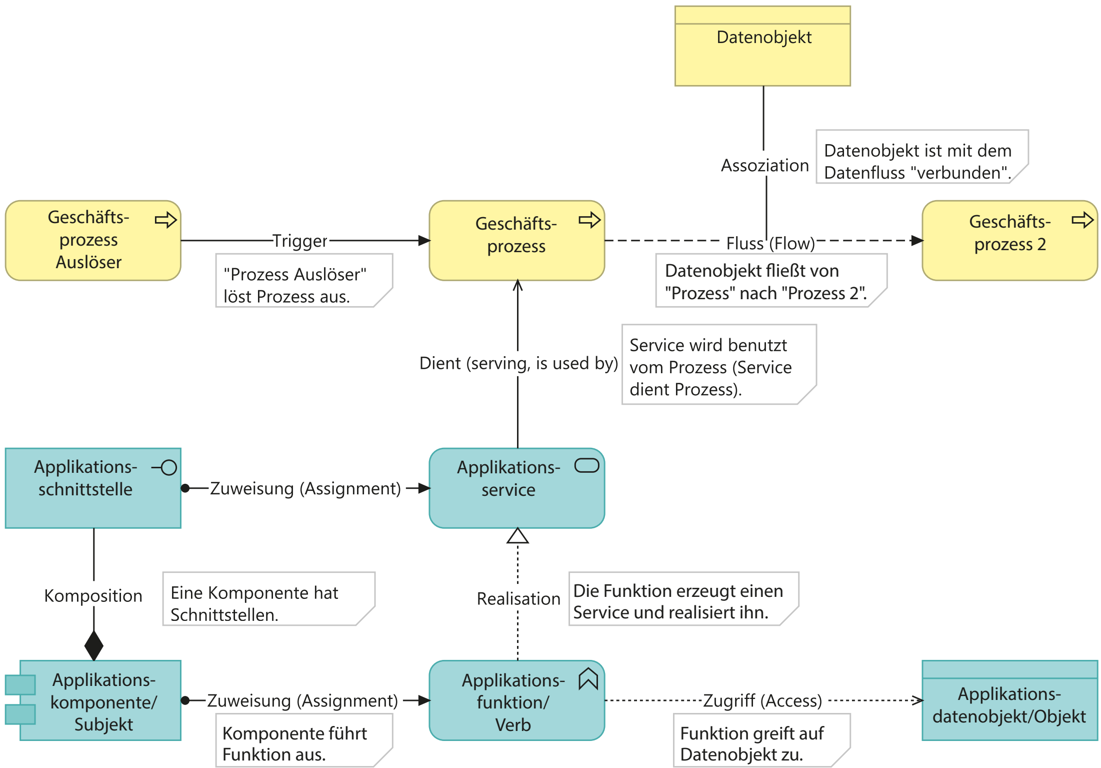


Bild 5.17 ArchiMate Relationen

5.3.5 Layered View

Mit der *Layered View* teilen wir jeden Layer auf. Der Service-Layer stellt die Schnittstelle zum nächsthöheren Layer dar. Daraus ergibt sich Bild 5.18. Ich habe mir aus Gründen der Übersichtlichkeit erlaubt, gleich die Ableitungen anzuwenden, um Ihnen den Service-Gedanken näherzubringen. Jeder Layer bietet dem darüberliegenden Layer einen Service an, über den der Zugriff erfolgt. Das passt ganz gut in unsere Schichtenarchitektur aus dem Kapitel 1. Die Kunst einer Layered-View, die eine Übersicht darstellen soll, ist es, die Details wegzulassen und sich auf die übergeordnete Architektur zu konzentrieren.

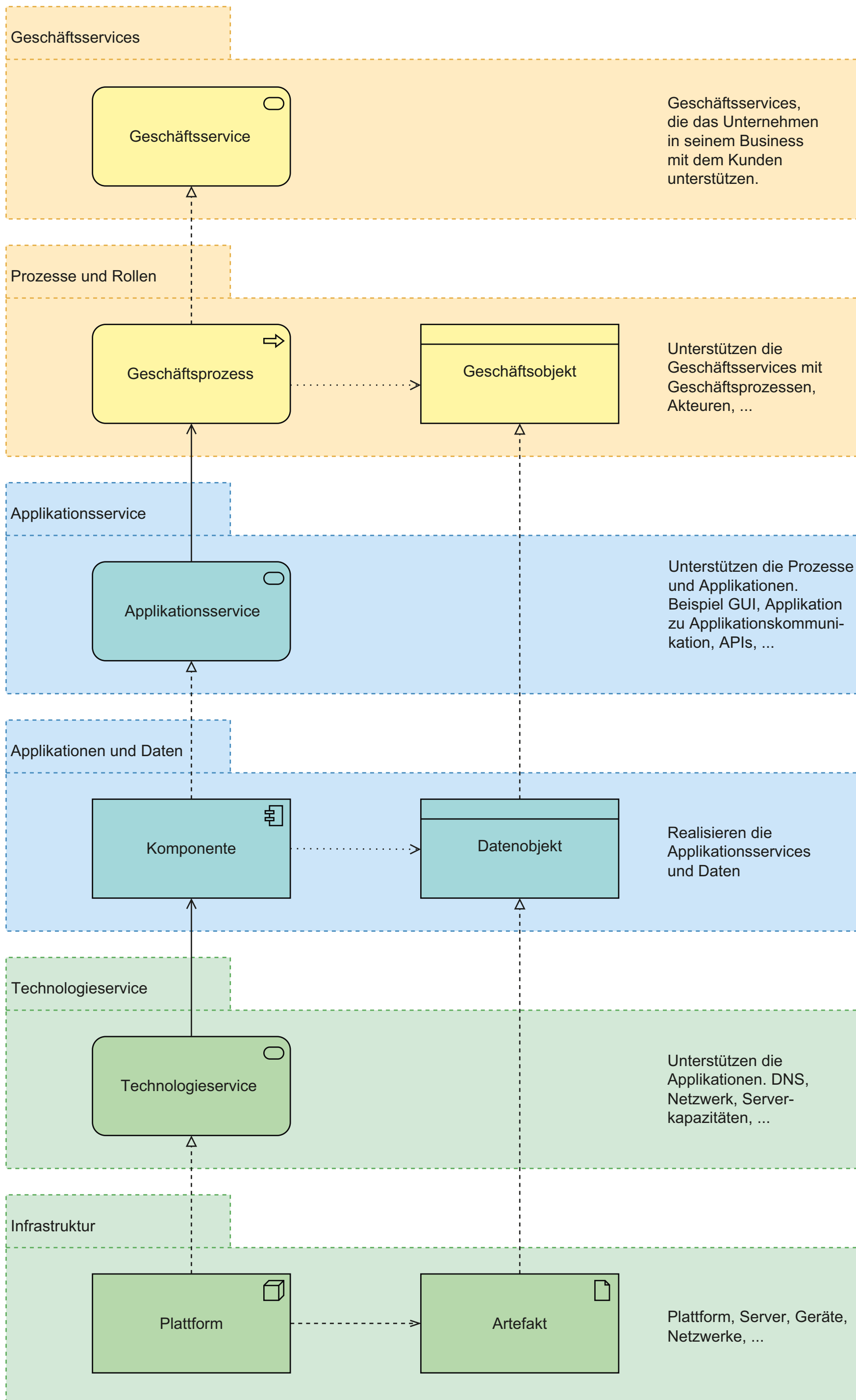


Bild 5.18 Layered-View

5.3.6 Applikations-Layer Scotland Trading

Das Grundmuster des Applikations-Layer haben Sie im Abschnitt 5.3.2 kennengelernt. Jetzt wollen wir diesen für *Scotland Trading* modellieren. Im Geschäfts-Layer haben wir in der Geschäftsfunktion *Preisbildung* den Prozess *Verkaufskonditionen festlegen* modelliert. Nun wollen wir uns fragen, welcher Applikations-Service diesen realisiert. Der Prozess *Marktpreise googeln* ist ein manueller Prozess und wird hier nicht ausmodelliert.

Gemäß dem Grundmuster dient einem Prozess ein Service. Definieren wir also einen Applikationsservice *Verkaufskonditionen pflegen*. Dieser Service stellt ein Applikationsverhalten dar, in unserem Fall die Materialwirtschaft, die durch die Funktion *Materialbewirtschaftung* der Komponente ERP ausgeführt wird (Bild 5.19).

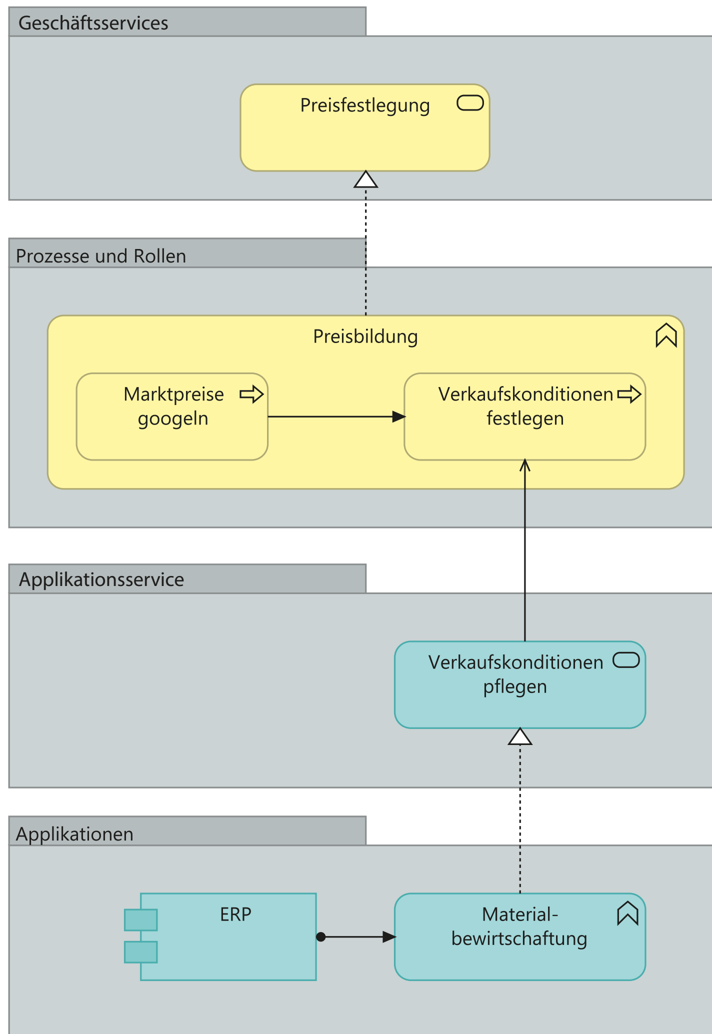


Bild 5.19 Layered-View, Ist-Situation, Preisbildung

Auch der Shop und das Data Warehouse, das das Unternehmen hat, bilden eine Komponente, die wir hier aber nicht dargestellt haben. In der Modellierung im Bild 5.20 haben wir die ERP-Shop-Beziehung modelliert.

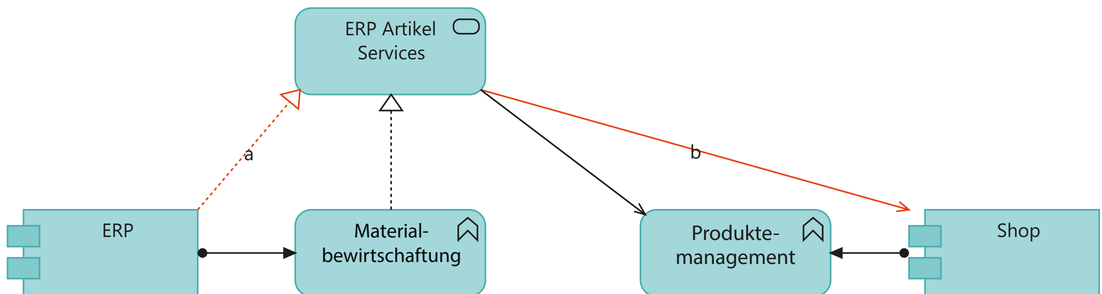


Bild 5.20 Kommunikation ERP – Shop

Die bereits kennengelernte Funktion *Materialbewirtschaftung* hat einen zusätzlichen Service für die Artikelabfrage realisiert: *ERP Artikel Services*. Über diesen können Artikel im ERP abgefragt werden. Durch die Ableitung kann direkt die Realisation *a* zwischen der Komponente und dem Service modelliert werden. Auf der anderen Seite bietet der Shop eine Funktion für das *Produktemanagement* an. Diese Funktion benutzt den *ERP Artikel Service*, ist also von diesem abhängig, um die verkaufbaren Artikel zu erhalten. Diese Realisation kann mittels *b* „dient" abgekürzt werden. Somit können wir auch eine Darstellung ohne die Funktionen machen. Wir haben bereits eine Möglichkeit kennengelernt, wie eine Kommunikation zwischen Komponenten abgebildet werden kann.

Es gibt weitere Möglichkeiten, die Kommunikation zu modellieren, die alle unterschiedliche Aspekte betonen. Starten wir mit der Komponentensicht in Bild 5.21 und danach mit der Funktionssicht in Bild 5.23. Dabei gehen wir vom oben erläuterten Szenario aus.

Die verschiedenen Varianten zeigen unterschiedliche Aspekte, von der Realisationssichtweise zur Datenflusssichtweise. Ich finde die erste Variante mit dem Applikationsservice wichtig, da diese den Zugriffspunkt über das Verhalten deutlich macht. Durch die Realisation ist erkennbar, wer den Service anbietet. Die zweite Variante mit dem Datenfluss finde ich attraktiv, denn dieser verdeutlicht, von wo nach wo Daten gehen. Wird zusätzlich die aktive Komponente *Schnittstelle* verwendet, so ergibt sich das Modell in Bild 5.22. Dies wird aber selten in einer Sicht dargestellt, da es bei mehreren Schnittstellen schnell unübersichtlich werden kann.

Scotland Trading verwendet aktuell für die Kommunikation keine Integrationsplattform. Es werden folglich Punkt-zu-Punkt-Schnittstellen der Hersteller oder Eigenentwicklungen benutzt. Wir werden in Kapitel 7 darauf zurückkommen und dieses Muster wieder hervorholen.

Wie wichtig die richtige Interpretation der Relationen ist, möchte ich anhand von Bild 5.23 erläutern. In der ersten Variante erkennen wir ein typisches Pull-Szenario. Der Shop holt für sich relevante Artikel im ERP ab. Das Push-Szenario ist in der dritten Variante abgebildet, also genau umgekehrt. Der Service wird vom Shop angeboten und das ERP ist vom Shop abhängig. Der Datenfluss ist in beiden Szenarien gleich!

Das ERP bietet einen Service an, den der Shop benutzt. Der Shop ist vom ERP abhängig.
Service-Sicht.

ERP
realisiert
ERP Artikel Services
dient
Shop

Ein Datenobjekt fließt vom ERP zum Shop. Das Verhalten spielt eine geringe Rolle.

ERP
fließt
Shop
Artikel

Das Datenobjekt fließt vom ERP über das API zum Shop.
Wer was anbietet, ist sekundär
Das API steht im Mittelpunkt.

ERP
fließen
Artikel API
fließen
Shop
Artikel

Das ERP hat ein API, das der Shop benutzt.
Aktive Sicht mit API Fokus.

ERP
Artikel API
dient
Shop

Bild 5.21 Kommunikation zwischen Komponenten

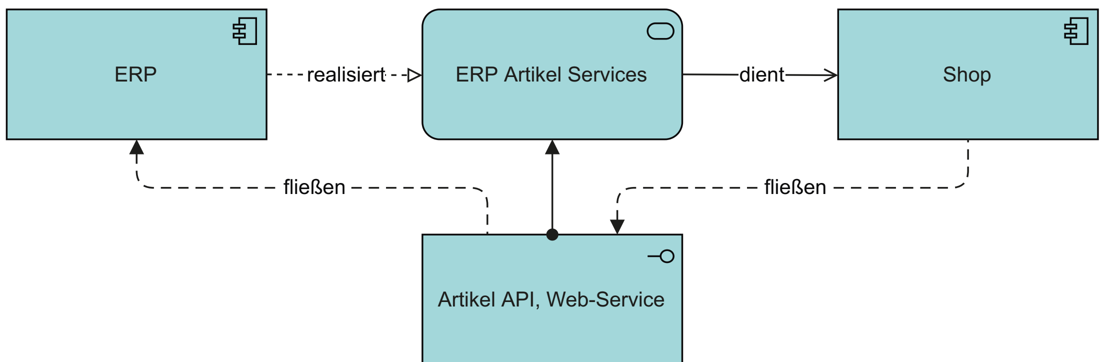


Bild 5.22 Schnittstelle und Service

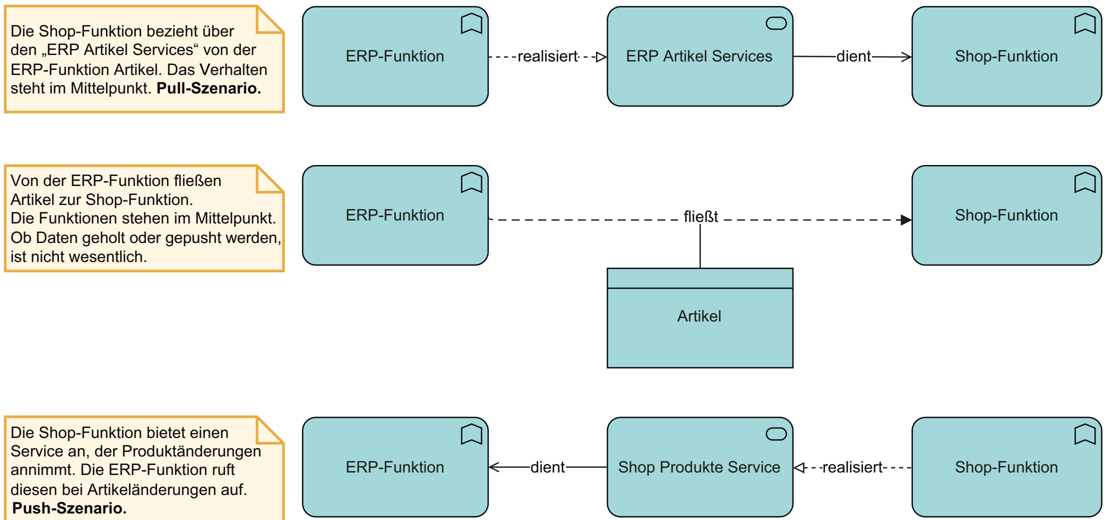


Bild 5.23 Push, Pull und Fluss

In Bild 5.24 finden Sie den Datenfluss der Ist-Situation von *Scotland Trading*. Das Reporting des Data Warehouse findet in der Analytics Cloud statt. Zwischen diesen Komponenten braucht es auch einen Datenfluss. Dieser kann unterschiedlich dargestellt werden, über Services oder Schnittstellen. Generell können Sie folgende Faustregeln anwenden: Steht eine service-orientierte Architektur im Vordergrund, also deren Verhalten, dann verwenden Sie Services. Ist das API wichtig, dann handelt es sich um eine statische Struktur und Sie verwenden Interfaces.

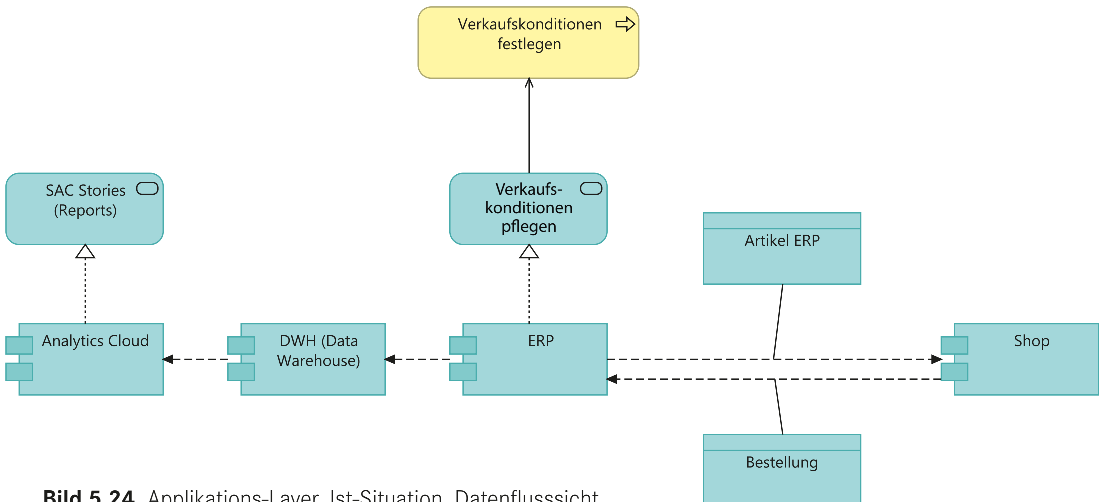


Bild 5.24 Applikations-Layer, Ist-Situation, Datenflusssicht

Im Bild 5.25 ist die service-orientierte Sicht abgebildet. Wir erkennen, dass der Shop die Services vom ERP benutzt. Die Artikel werden somit abgeholt und die Bestellungen werden gepusht.

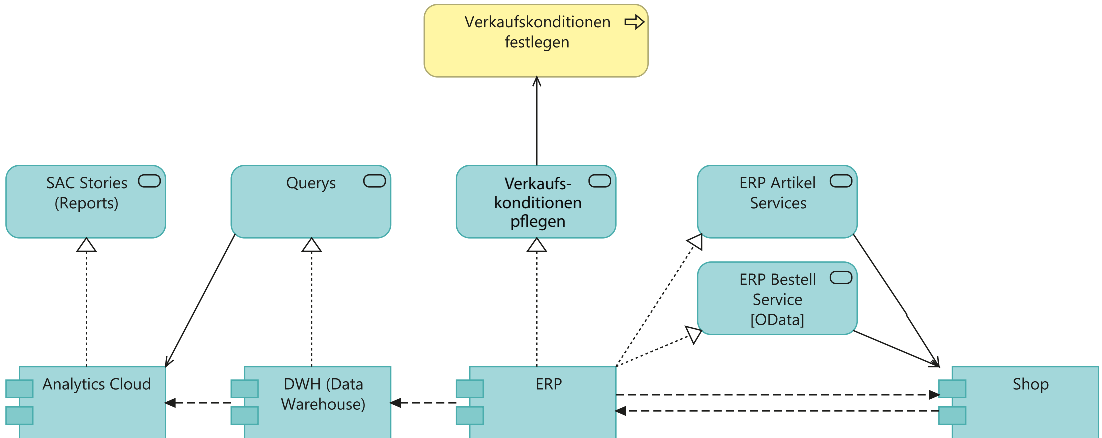


Bild 5.25 Applikations-Layer, Ist-Situation, Service-Sicht

Wenn Sie mit einem Tool wie *Archi* eine kombinierte Sicht mit Services und Schnittstellen erstellen wollen, erzeugen Sie einfach eine neue View. Der Vorteil eines Tools ist, dass alle bereits erstellten Elemente in die Sicht übernommen und platziert werden können. Sollen gewisse Relationen und Elemente nicht dargestellt werden, können diese aus der View gelöscht werden. Sollen sie aber definitiv aus allen Sichten, also aus dem Modell, gelöscht werden, muss dies wie in Bild 5.26 explizit erfolgen.

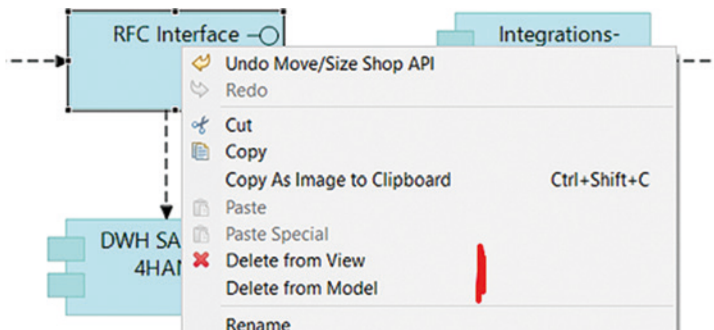
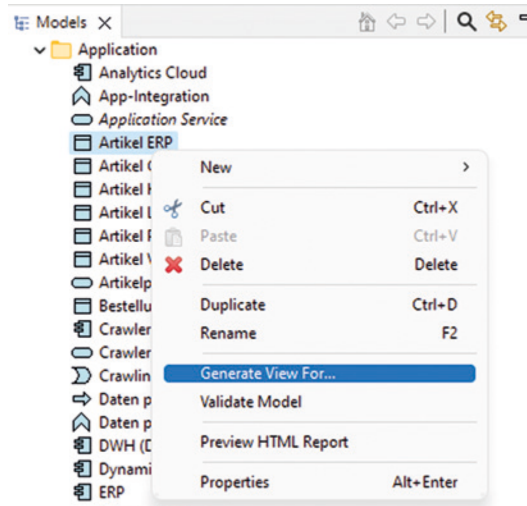


Bild 5.26 Archi: Löschen aus View oder Modell

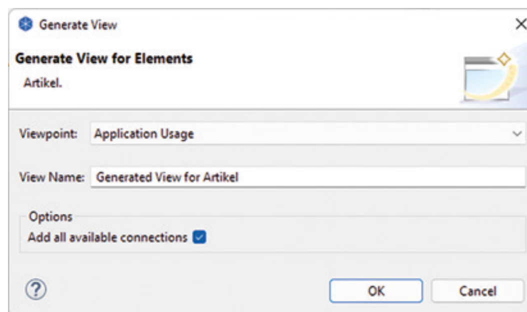
Mit der Modellierungssprache ArchiMate lassen sich Abläufe, Sequenzen und vieles mehr modellieren. Bei Architekturänderungen kann somit schnell festgestellt werden, welche Bereiche von der Änderung betroffen sind. Jedes Tool bietet dazu Analysewerkzeuge an. Wollen wir zum Beispiel sehen, wo der Artikel verwendet wird, können wir das wie folgt machen:

Element *Artikel* wählen und die rechte Maustaste auf diesen drücken.

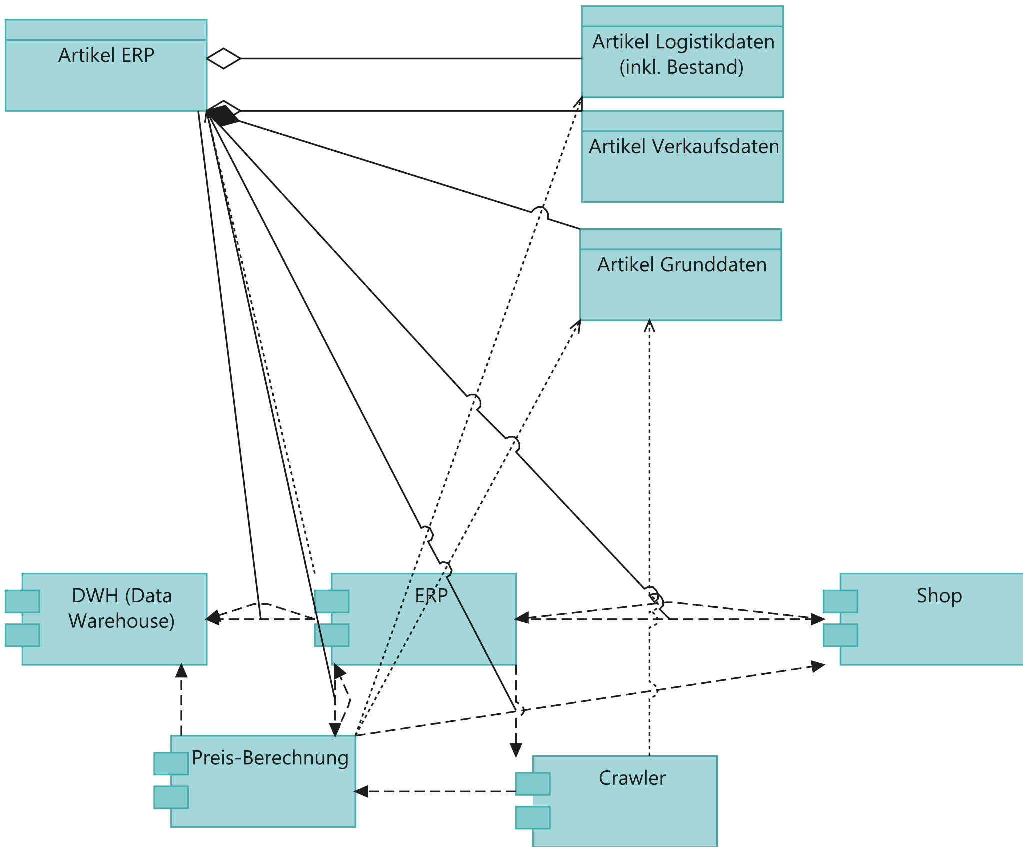
Dann **Generate View For...** wählen.



Jetzt kann der Viewpoint gewählt werden.



Archi generiert eine View und wir erkennen die Abhängigkeiten. Diese generierte View ist das Resultat am Ende des Buchs. Sie muss entsprechend dargestellt und auf die konkrete Problemstellung reduziert werden.



Um beim Modellieren nur die möglichen ArchiMate-Elemente zu sehen, ist es hilfreich, den Viewpoint jeder View manuell festzulegen. Klicken Sie dazu mit der rechten Maustaste in die View und wählen Sie unter **Viewpoint** die gewünschte Sicht. Sie erkennen im Bild 5.27 rechts in der Toolbar, dass nicht relevante Elemente ausgeblendet werden. Verwenden Sie nicht vorgesehene Elemente in der View, werden diese transparent dargestellt.

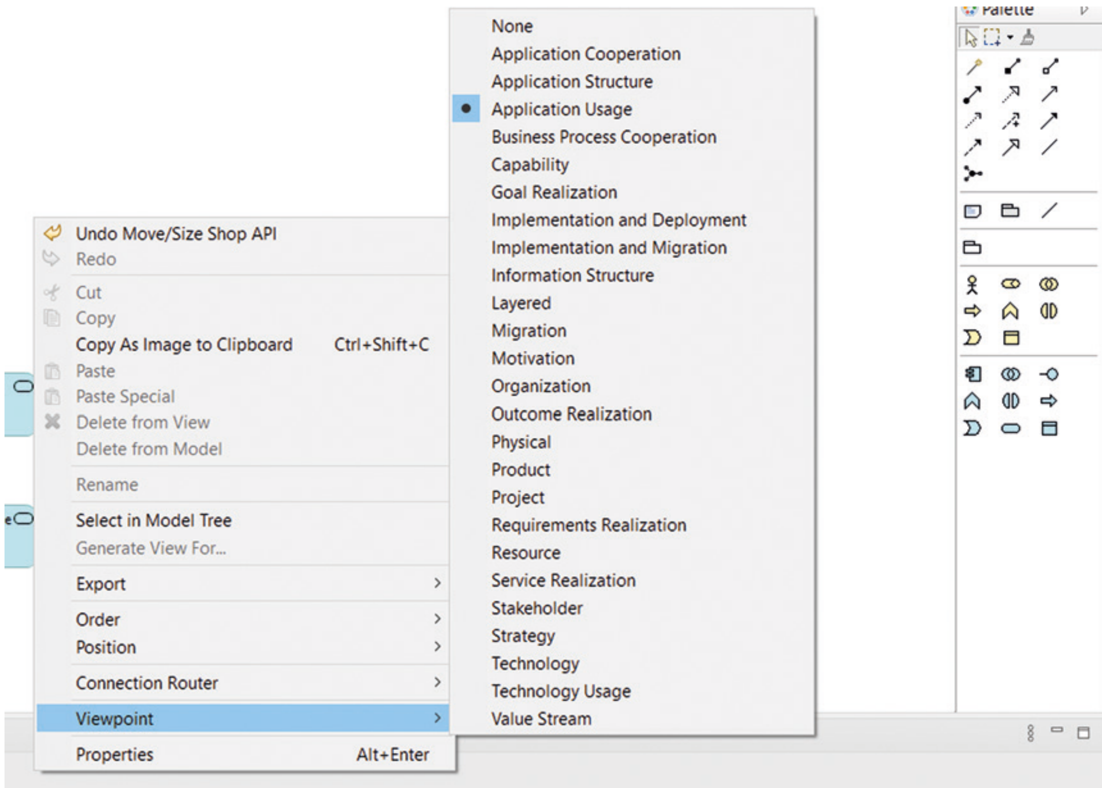


Bild 5.27 Viewpoint auf der View festlegen

5.3.7 Technologie-Layer

Die Applikationssicht ist modelliert, aber worauf laufen diese Komponenten? Dafür gibt es den Technologie-Layer. ArchiMate sieht hier bekannte Elemente wie den Service vor, aber auch Elemente wie ein physisches Gerät. Deshalb wird in der Praxis zwischen Technologie und dem physischen Layer unterschieden. Beide Ebenen werden häufig grün dargestellt. Bild 5.28 zeigt das Grundmuster des Technologie-Layers.

Stichwortverzeichnis

A
Activity (Android) 51
Adapter-Muster 190, 199, 238
Ahead of time Compiler (AOT) 50
ALM 91
Android Runtime 50
Anforderungen 95
Ansatz
– portfolioorientiert 117
– projektorientiert 123
Anspruchsgruppen 85
Antwortzeit (RT) 66
API 18
– Gateway 250
Applikations-Layer 148
arc42 222
ArchiMate 133
– Ableitungen 144
– Anforderung 195
– Applikationsereignis 137
– Applikationsfunktion 136
– Applikationsiteraktion 137
– Applikationskollaboration 137
– Applikationskomponente 136
– Applikationsprozess 137, 189
– Applikationsschnittstelle 136, 240
– Applikationsservice 136, 148
– Artefakt 157
– Datenobjekt 136, 216
– Deliverable 161
– Exchange-Format 139
– Gap 161
– Gerät 157
– Geschäftsereignis 144
– Geschäftsfunktion 139
– Geschäftsinteraktion 144
– Geschäftskollaboration 144
– Geschäftsobjekt 139, 216
– Geschäftsprozess 139
– Geschäftsrolle 138
– Geschäftsschnittstelle 138
– Geschäftsservice 139
– Grammatik 134
– Grundmuster 135, 138
– Knoten 156
– Location 157
– Metamodell 159
– Plateau 161
– Produkt 144
– Relationen 145, 170
– Relationen im Grundmuster 137
– Repräsentation 144
– Systemsoftware 157
– Technologiefunktion 156
– Technologieschnittstelle 157, 242
– Technologie-Service 156, 190, 230
– Vertrag 144
– Work Package 161
Archi (Programm) 139
Architecture-Thinking 127, 173
Architekt
– Berufsbeschreibung 115
– Eigenschaften 84
Architektur
– Anforderungsmanagement 131
– Applikations- 127
– Chancen und Lösungen 130
– Geschäfts- 126
– Informationssystem- 127
– Roadmap 130
– Steuerung und Implementierung 131
– Technologie- 129

– Umsetzungsplanung 130
– Veränderungen 131
– Vision 125
– Ziele 83
Architektur-Board 167
Architektur-Prinzipien 164
Architekturvision 125
Asynchrone Systeme 234
Auftragsbasierte Kommunikation 234

B
Backend for Frontend Muster 199
Baustein
– Architektur-Prinzipien 164
– Bausteinsicht 173
– Datensicht 216
– Geschäftsziele 84
– Kontextsicht 100
– Laufzeitsicht 210
– Pace-layered Application Strategy 205
– Qualitätsszenario 110
– Realisationssicht 183
– Roadmap 130
– Stakeholder-Matrix 86
– Zustandssicht 212
Bausteinsicht 173
Benutzbarkeit 104
Benutzermodus 8
Betriebssystem 6
Bibliothek 12
Big Endian 36
Blackberry 44, 49
Blackbox 4, 100
Blockierendes System 233
Bottom-Up 14
BPEL 250
Building Blocks *siehe* Baustein
Business-Addins 25
Business APIs 19
Business Process Management Systems 250
Business Process Modeling 181
Busy waits 66
Buy-Configure-Build 166, 185

C
Cache 209
Callback-Funktion 234
Chain 58
Choreografie 181, 189
Circuit-Breaker-Muster 201
Conway's Law 89
Core-Layer 134
Correlation IDs 221
CQRS 201, 222
Create 58
Culture Eats Strategy For Breakfast 90

D
Dateiaustausch 61
Dateikonzept 40
Dateizugriff 20
Datenarchitektur 97
Datenformate 235
Datenobjekt 216
Datensegmente 54
Datensicht 216
Datentypkanal 246
Design-Thinking 127
Dient 139
DLL 47
Dokumentation 205, 222
Domain Driven Design 170
Durchlaufzeit (TT) 66
Dynamische Preisbildung 98
Dynamische Priorität 72

E
EAIP 229
Einflussfaktoren 81
Enge Kopplung 33
Enterprise Application Integration Pattern (EAIP) 229
Enterprise Service Bus (ESB) 250
Erweiterbarkeit 24
Event-driven 181, 233
Event Sourcing 221

F
Fachdomänen 96
Fassaden-Muster 26, 199

Fire and Forget 247
First In First Out 67
Fit-Gap-Analysen 202
Fork 58
Framework 12, 187
Funktionale Anforderungen 95
Funktionale Eignung 105

G
Gap 126
Geheimnisprinzip 4
Gerätetreiber 7
Geschäftsarchitektur 101, 116, 126
Geschäfts-Layer 137
Geschäftsobjekt 216
Geschäftsziele 83
Glossar 225
Governance 125, 168
Grafiktreiber 47
Grammatik 134

H
HAL 36
Heap 54
Hilfsmittel
- arc42 222
- Architektur-Board 167
- Glossar 225
- Megatrend-Landkarten 91
- Referenzarchitekturen 202
- Technische Schulden 113
- Werkzeugkoffer 92
Hybridkernel 46

I
ICT-Architekt 115
Implementation und Migration Layer 161
Informationssystemarchitektur 127
Integrationsplattform 207, 209, 242
Intents 51
Interprozess-Kommunikation (IPC) 42
Interrupts 6
Inversion of Control (IoC) 12
IPC 60, 236
IT-Revision 125

J
Java Dalvik 50
Java Nativ Interface 51
Juristische Aspekte 88

K
Kanal 236
Kategorie von Systemen 93, 97
Keep the core clean 25
Kernel-Modus 8
Kohäsion 16, 48, 178
Kommunikation 149
Kommunikationskanal 236
Kompatibilität 106
Komponenten 10
Komposition 137
Kontextsicht 100
Kopplung 14, 185

L
Laufzeitsicht 210
Layer
- Applikation 148
- Geschäft 137
- Implementation und Migration 161
- Motivation 162
- Physisch 155
- Strategie 162
- Technologie 155
Layered View 146
Lebenszyklus 92
Leistungseffizienz 106
Little Endian 36
Logische Gruppierung 183
Logische Kanäle 7
Logs 203
Lose Kopplung 10, 196, 207, 246

M
Meet in the Middle 14
Meldungsorientierte Kommunikation 234
Message Channel 245
Message Queue 247
Metamodell 159
Microservice 48, 179
Middleware 49

Mikrokernel 42, 179
Minimum Viable Product (MVP) 188
Mobile Betriebssysteme 49
Modul 12
Modularisierung 13, 174
Modulkopplung 14
Monitoring 204, 209, 254
Monolith 179
Monolithische Architektur 33
Multiprozessor-System 74
Muster
- Adapter 199
- Backend for Frontend (BFF) 199
- Circuit-Breaker 201
- CQRS 201
- Fassaden 26
- Observer 27
- Pipe 39
- Prozess Oriented 187
- Retry 201
- Rules Oriented 187
- Schichten 26

N
Nachrichtenkanal 245
Nanoservice 180
Native Cloud 94, 129, 187, 201, 222, 246
Nebenläufigkeit 233
Netzwerk-Socket 61
Nicht blockierendes System 234
Non-preemptive 64

O
Observer-Muster 27
Offen-Geschlossen-Prinzip 24, 33, 179, 186, 195
Orchestrierung 181, 189
Ordnungsrahmen 117
Organisation (Randbedingungen) 89
Orthogonal 39

P
Pace-layered Application Strategy 205, 238
Performance 109
Pipe-Muster 39, 181, 189
Portabilität 105
Pragmatisch XIII
Preemptive Scheduling 64
Prinzip
- Correlation IDs 221
- Event Sourcing 221
- Geheimnis 4
- Modularisierung 13
- Offen-Geschlossen- 167
- Separation Of Concerns 177
- Strategie und Mechanismus 29
Prinzip des geringsten Privilegs 42
Priorität 71
Process Control Block (PCB) 64
Process Oriented Pattern 187
Product Owner 86
Programmcode 54
Programm (Definition) 54
Projektvorgehen 123
Projektziele 83
Proof of Conept (PoC) 188
Proxy 51
Prozess 54
Prozesserzeugung 58
Prozesshierarchie 58
Prozesskommunikation 60
Prozessmodell 56
Prozessunterbrechung 64
Prozesszustände 65
Publish-Subscribe-Muster 246
Pull 149, 233
Punkt-zu-Punkt-Kanal 237
Push 149, 233

Q
QNX 44
Qualität (Definition) 103
Qualitative Anforderungen 95, 103
Qualitätskorridor 114
Qualitätskriterien 103
- Betriebssysteme 33
- Wechselwirkungen 107
Qualitätsszenario 110, 188, 196
Quantum 70
Querschnittliche Konzepte 203
Queue 245

R

Randbedingungen 88
- Juristische Aspekte 88
- Organisation 89
- Resssourcen 88
- Standards 89
- Technologien 89
- Trends 90

Raummultiplex 57
Realisation 137
Realisationssicht 183
Referenzarchitektur 89, 202
Referenzprozesse 202
Registermaschine 50
Remote Procedure Calls (RPC) 236
Ressourcen (Randbedingungen) 88
Retry Muster 201
Risiken 116
Risikoanalyse 126
Roadmap 130
Round Robin 69
Rules Oriented Pattern 187

S

SAFe 124
Samsung Knox 49
SAP S/4HANA 3
Scheduler 66
Schichtenarchitektur 26, 35, 48, 146
Schneiden eines Monoliths 35, 182
Schnittstelle 11, 97
Scotland Trading 98, 255
Separation-Of-Concerns-Prinzip 177, 185, 188, 195
Sequenzdiagramme 210
Services (Iaas, PaaS, Saas, BPaas) 170
Serving 139
Shared Memory 60
Shortest Job First 73
Shortest Remaining Time 74
Sicherheit 104
Single-Responsibility-Prinzip 177
Skalierbarkeit 106
SOA 250
Solution Concept Diagram 173
Solution-Context-Diagramm 100
Solution Realization 183
Spannungsfeld 106
Stack 54
Stackmaschine 50
Stakeholder 85
- Matrix 86

Standards (Randbedingungen) 89
Strategie und Mechanismus 29, 77, 188, 195
Strategie- und Motivations-Layer 162
Strategisches Design 170
Struktur XII
Support-Prozesse 204
Synchrone Kommunikation 233
Systemaufruf 7, 20
Systemcall-Aufruf 20
Systemcall-Interface 7
System Call Table 20
Systemkategorie 93
Systems of Differentiation 206
Systems of Innovation 207
Systems of Record 206

T

Taktisches Design 170
Technische Schulden 113
Technologiearchitektur 129
Technologie-Layer 155
Technologien (Randbedingungen) 89
Teile und herrsche 170
Tests 108, 204
Thread 61
TOGAF 119
- ADM 122
- Ebenen 123, 202

Top-Down 13
Topic 246
Transformation 242
Transformationsprojekt 84, 123
Transportsicherheit 234
Traps 37
Treiber 40
Trend 203
Trends (Randbedingungen) 90

U
Übergangstabellen 215
Unix System V 37
Unternehmensstandard 89
Unternehmensstrategie 84
UUID 221

V
Verbesserte monolithische Architektur 35
Verfahrensvorschrift 54
Vermittler 244
View 134
– generieren 153
Viewpoint 134
– festlegen 154
Vogelperspektive 100
Vorbereitungsphase 124
Vorgehensmodelle 117

W
Wartbarkeit 105
Warteschlange 245
Wartezeit (WT) 66
Wechselwirkung 107
Werkzeugkoffer 89, 92, 166, 203
Windows 75
Windows 10 46
Workflow-Systeme 181

Z
Zeitmultiplex 63
Zeitscheibe 69
Zombie-Prozess 59
Zugriff 137
Zustandsdiagramm 212
Zustandsmodell 65
Zustandssicht 212
Zuverlässigkeit 105
Zuweisung 137