

O. Vogel I. Arnold A. Chughtai
E. Ihler T. Kehrer U. Mehlig U. Zdun

Software- Architektur

Grundlagen – Konzepte – Praxis

2. Auflage

Software-Architektur

In dieser Reihe sind bisher erschienen:

Martin Backschat / Bernd Rücker
Enterprise JavaBeans 3.0
Grundlagen – Konzepte – Praxis

Peter Liggesmeyer
Software-Qualität
Testen, Analysieren und Verifizieren von Software

Michael Englbrecht
Entwicklung sicherer Software
Modellierung und Implementierung mit Java

Klaus Zeppenfeld
Objektorientierte Programmiersprachen
Einführung und Vergleich von Java, C++, C#, Ruby

Martin Backschat / Stefan Edlich
J2EE-Entwicklung mit Open-Source-Tools
Coding – Automatisierung – Projektverwaltung – Testen

Marco Kuhrmann / Jens Calamé / Erika Horn
Verteilte Systeme mit .NET Remoting
Grundlagen – Konzepte – Praxis

Peter Liggesmeyer / Dieter Rombach (Hrsg.)
Software Engineering eingebetteter Systeme
Grundlagen – Methodik – Anwendungen

Stefan Conrad / Wilhelm Hasselbring / Arne Koschel / Roland Tritsch
Enterprise Application Integration
Grundlagen – Konzepte – Entwurfsmuster – Praxisbeispiele

Ingo Melzer et al.
Service-orientierte Architekturen mit Web Services, 3. Auflage
Konzepte – Standards – Praxis

Oliver Vogel / Ingo Arnold / Arif Chughtai / Edmund Ihler /
Timo Kehrer / Uwe Mehlig / Uwe Zdun
Software-Architektur, 2. Auflage
Grundlagen – Konzepte – Praxis

Marco Kuhrmann / Gerd Beneken
Windows® Communication Foundation
Konzepte – Programmierung – Konzeption

Andreas Korff
Modellierung von eingebetteten Systemen mit UML und SysML

Oliver Vogel / Ingo Arnold / Arif Chughtai /
Edmund Ihler / Timo Kehrer / Uwe Mehlig /
Uwe Zdun

Software-Architektur

Grundlagen – Konzepte – Praxis

2. Auflage

Unter Mitwirkung von Markus Völter

Spektrum
AKADEMISCHER VERLAG

Autoren:

Oliver Vogel, Ingo Arnold, Arif Chughtai, Edmund Ihler, Timo Kehrer, Uwe Mehlig,
Uwe Zdun
E-Mail: autoren@software-architektur-buch.de

Weiterführende Informationen zum Buch:

www.software-architektur-buch.de

Wichtiger Hinweis für den Benutzer

Der Verlag und die Autoren haben alle Sorgfalt walten lassen, um vollständige und akkurate Informationen in diesem Buch zu publizieren. Der Verlag übernimmt weder Garantie noch die juristische Verantwortung oder irgendeine Haftung für die Nutzung dieser Informationen, für deren Wirtschaftlichkeit oder fehlerfreie Funktion für einen bestimmten Zweck. Ferner kann der Verlag für Schäden, die auf einer Fehlfunktion von Programmen oder ähnliches zurückzuführen sind, nicht haftbar gemacht werden. Auch nicht für die Verletzung von Patent- und anderen Rechten Dritter, die daraus resultieren. Eine telefonische oder schriftliche Beratung durch den Verlag über den Einsatz der Programme ist nicht möglich. Der Verlag übernimmt keine Gewähr dafür, dass die beschriebenen Verfahren, Programme usw. frei von Schutzrechten Dritter sind. Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Buch berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürfen. Der Verlag hat sich bemüht, sämtliche Rechteinhaber von Abbildungen zu ermitteln. Sollte dem Verlag gegenüber dennoch der Nachweis der Rechtsinhaberschaft geführt werden, wird das branchenübliche Honorar gezahlt.

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.

Springer ist ein Unternehmen von Springer Science+Business Media
springer.de

2. Auflage 2009

© Spektrum Akademischer Verlag Heidelberg 2009

Spektrum Akademischer Verlag ist ein Imprint von Springer

09 10 11 12 13

5 4 3 2 1

Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Verlages unzulässig und strafbar. Das gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Planung und Lektorat: Dr. Andreas Rüdinger, Barbara Lühker

Herstellung: Ute Kreutzer

Umschlaggestaltung: SpieszDesign, Neu-Ulm

Satz: Autorensatz

Layout/Gestaltung: Graphik & Text Studio, Barbing

Druck und Bindung: Krips b.v., Meppel

Printed in The Netherlands

ISBN 978-3-8274-1933-0

| Geleitwort

Seit vielen Jahren leite ich die *IT Architect Profession* bei der IBM in Europa. Meine Aufgabe ist es, die Entwicklung von IT-Architekten zu fördern und dafür zu sorgen, dass sie ihr Wissen auf dem neuesten Stand halten. Immer mehr Kunden und Mitbewerber sind daran interessiert, ihre eigenen Architektur-Fähigkeiten auszubauen. Die Open Group, ein Technologie- und Anbieter-unabhängiges Konsortium, bietet seit 2006 das sogenannte *Open Group Information Technology Architect Certification Program* an. Dieses wird inzwischen von vielen unserer Kunden und Mitbewerbern genutzt, um die Qualifikation ihrer Mitarbeiter entsprechend evaluieren zu können.

In diesem Zusammenhang bin ich erfreut über die neue Auflage dieses Buches. Es beschreibt und erklärt sehr anschaulich und gut strukturiert, was Architekten von IT-Systemen tun und womit sich die IT-respektive Software-Architektur überhaupt beschäftigt. Das Buch bietet somit eine gute Grundlage, um sich mit dem Thema vertraut zu machen und um seine Architektur-Fähigkeiten zu verbessern. Es passt genau in den Trend der Zeit, den ich sowohl in der Open Group als auch bei unseren Kunden und Mitbewerbern sehe. Es spiegelt die Denkweise wieder, die wir seit vielen Jahren in der IBM fördern und fordern.

Es ist eine sehr gute Zeit für IT-Architekten. Die IT- und Technologie-trends entwickeln sich immer weiter und schneller. Eine Software-Architektur als Basis bei der Entwicklung von IT-Systemen ist zunehmend wichtiger geworden, um mit diesen rasanten Veränderungen umzugehen. Nicht zuletzt die ganze Diskussion rund um das Thema Service Oriented Architecture (SOA) hat das mehr als deutlich gemacht.

Deshalb finde ich das Buch sehr empfehlenswert für alle, die die Notwendigkeit erkannt haben, sich mit dem Thema Software-Architektur auseinander zu setzen. Es bietet einen sehr umfassenden Einstieg in das bewusste Architektur-Denken.

*Karin Dürmeyer
IBM Distinguished Engineer
IBM IOT Northeast IT Architect Profession Leader*

Architektur-Fähigkeiten werden immer wichtiger

Dieses Buch hilft beim Aufbau und Ausbau dieser Fähigkeiten

Die Zeit ist reif, um in das spannende Thema einzusteigen...

...und ein architektonisches Bewusstsein zu entwickeln

| Vorwort zur 2. Auflage

Seit dem Erscheinen der 1. Auflage dieses Buches vor nunmehr fast drei Jahren ist vieles geschehen. Architektur konnte sich als eigene Disziplin innerhalb der angewandten Informatik fest etablieren und *der Architekt* avancierte zur tragenden Rolle in großen oder komplexen Software-Projekten. Diese Entwicklung kann gut verglichen werden mit derjenigen im Bereich der klassischen Architektur, in der – ausgehend vom zupackenden Maurer, Zimmermann und Dachdecker – nachfolgend erst die Rolle des Architekten entstand, der als Generalist auch ästhetische, zeitliche und bautechnische Aspekte mit ganzheitlichem Blick integrierte. Auch innerhalb der Informatik vollzieht sich ein Wandel weg vom praktischen Handwerker hin zum konzeptuellen Planer. Kurzum: Das Thema *Architektur* hat signifikant an Bedeutung gewonnen.

Konzeptuell planerische, ausbildende oder organisatorische Beiträge haben in dem Maß an Bedeutung gewonnen, in dem technisches Spezialwissen in Länder ausgelagert wird, deren Lohnstrukturen und Expertenbasis diesen Trend fördern. Die Rolle des *Architekten* bildet in unseren Augen mit ihrer ganzheitlichen und integrativen Sicht auf die IT-Herausforderungen die Speerspitze einer neuen Generation von Ausbildungoprofilen innerhalb der Informatik und angrenzender Domänen.

Die Informatik ist eine der am schnellsten wachsenden und sich wandelnden Wissenschaften unserer Zeit. Drei Jahre Entwicklung im Bereich Informationstechnologie entsprechen Dekaden in vielen anderen Technologiebereichen. Einer unserer zentralen Wünsche für die vorliegende 2. Auflage war es daher, die architektonischen Entwicklungen, Veränderungen und Trends angemessen zu berücksichtigen und damit unser Buch auf den *neuesten Stand* zu bringen

Darüber hinaus haben wir in den Jahren seit Erscheinen der 1. Auflage viele Erfahrungen gemacht in der direkten und praktischen Arbeit mit unserem Buch. Wir setzten es z. B. ein, um in Projekten ein einheitliches Vokabular und Verständnis von Architektur zu etablieren; wir verwendeten es, um uns selber in weniger gut bekannten Wissensgebieten zu orientieren; wir stellten es in Vorträgen und an Konferenzen vor und diskutierten viele Aspekte desselben mit einem sehr interessierten Publikum; wir verwendeten es als Basis für Vorlesungen an verschiedenen Hochschulen und erhielten zahlreiche E-Mails von unseren Lesern mit wertvollen Rückmeldungen. Die Verbesserungen, die wir aus

Feste Etablierung von Architektur als eigene Disziplin in der Informatik

Der Architekt als zukunftsorientiertes Rollenprofil

**Motivation 1:
Buch auf den *neuesten Stand* bringen – aktualisieren, korrigieren, entfernen.**

**Motivation 2:
Eigene, praktische Erfahrungen mit dem Buch einbringen**

unseren eigenen Erfahrungen ableiteten sowie die Rückmeldungen und Erweiterungswünsche unserer Leser waren uns ein weiteres starkes Motiv, eine 2. Auflage zu veröffentlichen.

**Motivation 3:
Konsistenz und
didaktische Qualität
verbessern**

Der letzte wichtige Anlass für uns, das Werk noch einmal zu verbessern, war unser eigener Qualitätsanspruch. Wir haben daher für die 2. Auflage sehr großen Wert gelegt auf eine weitere Steigerung der begrifflichen und inhaltlichen Konsistenz sowie die Ergänzung durch verständnisfördernde Mittel wie beispielsweise Concept-Maps oder Checklisten. Zudem haben wir die thematischen Beziehungen zwischen verwandten Buchabschnitten noch deutlicher gemacht.

Obwohl auch die 2. Auflage äußerst arbeitsintensiv war und uns Autoren wieder viele Stunden unserer Freizeit gekostet hat, sind wir alle doch froh, diesen Anlauf genommen zu haben, denn unser Buch hat dadurch noch einmal stark an Relevanz, Vollständigkeit und Qualität gewonnen.

Viel Freude beim Lesen

Wir hoffen, dass Sie das Buch in seiner überarbeiteten Form genauso interessiert, aufmerksam und begeistert lesen werden wie wir es in den vergangenen fast 12 Monaten überarbeitet haben. Wir wünschen Ihnen viel Freude bei der Lektüre.

Unser Dank

Abschließend wollen wir uns auch hier wieder bei all jenen bedanken, die uns für die Arbeit am vorliegenden Werk frei stellten und unterstützten. Dies waren unsere Partnerinnen und Kinder, unsere Eltern und Geschwister, unsere Freunde und Kollegen, unsere Arbeitgeber und Vorgesetzte. All denen möchten wir danken, die uns ihre Zeit opferten und immer wieder neue Kraft gaben.

Für die wertvollen Kommentare, Hilfe und Verbesserungsvorschläge möchten wir Kerstin Gans, Cordula Kleinschmidt, Johannes Blöcker, Martin Gottschalk, Peter Jess, Christoph Mikovic, Bernhard Polarzyk, Bernhard Scheffold, Gilbert Semmer, Martin Sieber sowie Marco Kuhrmann von der Technischen Universität München ganz herzlich danken.

Ein besonderer Dank gilt Karin Dürmeyer für Ihre Worte, die sie uns und der 2. Auflage unseres Buches mit auf den Weg gegeben hat. Sie waren für uns ein wertvoller Antrieb.

Ebenso danken wir Barbara Lühker und Dr. Andreas Rüdinger von Spektrum Akademischer Verlag für ihre Unterstützung.

| Vorwort zur 1. Auflage

Im IT-Arbeitsalltag ist der Begriff „Software-Architektur“ bzw. ganz allgemein „Architektur“ allgegenwärtig. Auf Visitenkarten stehen Rollenbezeichnungen wie Software-, Sicherheits-, Daten- oder Systemarchitekt. Für Kunden werden Dokumente erstellt, die mit „Lösungsarchitektur“ überschrieben sind oder aber man befindet sich selbst in der Kundenrolle und vergibt Architektur-Aufträge an Lieferanten. Obwohl der Begriff „Architektur“ hierbei so häufig verwendet wird, verstehen (wenn man genau hinsieht) Architekten, Projektleiter oder Entwickler diesen Begriff nicht auf die gleiche Art und Weise.

Für manche von uns ist „Architektur“ die Auswahl und der Einsatz einer Technologie, für andere ist „Architektur“ vor allem ein Prozess, für viele ist „die Architektur“ eine Mappe mit Zeichnungen, auf denen miteinander verbundene geometrische Figuren zu sehen sind, für noch andere mag „Architektur“ schlicht all das sein, was „der Architekt“ produziert. Der Begriff „Architektur“ ist in seiner praktischen Verwendung breit ausgelegt – sprich: wenig einheitlich definiert oder verstanden. Dies macht es oft schwer, in der Architektur-Domäne und im Berufsalltag über einzelne Personen hinweg effizient zusammenzuarbeiten und zu kommunizieren.

Als wir uns entschlossen, ein Buch über Software-Architektur zu schreiben und in unser Vorhaben mit einer ersten Bestandsaufnahme starteten, lernten wir schnell, dass sich Software-Architektur selbst in einer scharf begrenzten Gruppe von erfahrenen Software-Architekten nicht so klar definieren ließ, wie wir das erwartet hatten. Wir stellten fest, dass – obwohl wir alle jahrelange Erfahrung mit dem Entwurf, der Beschreibung oder auch Überprüfung von Software-Architekturen hatten – wir doch nicht über ein einheitliches, präzises Ordnungs- und Begriffsverständnis in der Architektur-Domäne verfügten.

Je länger wir uns damit auseinandersetzen, desto mehr wurde uns die Herausforderung bewusst, die der Entwurf und die Beschreibung eines Erklärungsmodells bedeutete, mit dem wir in der Lage sein würden, das Thema Architektur differenziert zu betrachten und zu erklären. Auf der anderen Seite spürten wir, dass ein solches Erklärungsmodell genau das war, was uns fehlte. Ein Modell als fester Architektur-Bezugspunkt also, von dem aus wir das Architektur-Universum konsistent erschließen und auf das wir uns stets beziehen können.

**Architektur als Begriff
ist allgegenwärtig ...**

**... und vielseitig inter-
pretiert ...**

**... zunächst auch in
unserem Autorenteam**

**Unser Wunsch nach
einem Erklärungsmodell ...**

... und nach Orientierung

Wir erinnerten uns an die Zeit zurück, in der wir selber primär Software-Entwickler waren und mit dem Begriff Software-Architektur das erste Mal konfrontiert wurden. Wir erkannten ganz speziell in dieser Zeit unser Streben nach einem inneren und stabilen Erklärungsmodell, nach einem Satz von Mustern, die stabil und unabhängig von einer konkreten Lösung universell einsetzbar waren. Die Suche nach einem solchen Grundmodell, das die wichtigen Dimensionen der Architektur-Domäne aufdeckt, lief in jedem von uns lange Zeit unbewusst bzw. intuitiv ab. Am Anfang unserer „Reise durch die Informatik“ benötigten wir sehr viel Fach- und Detailwissen, konzentrierten wir uns auf den Erwerb von Wissen rund um Techniken und Technologien, Prozessmodelle, Methoden und Organisationen. Ständig und teilweise, ohne uns dies bewusst zu machen, leiteten wir im Laufe unseres Berufs- und damit Lern- und Ausbildungslebens aus dieser Sammlung isolierter Einzelerkenntnisse ein jeder für sich *sein* Erklärungsmodell der Architektur-Domäne ab. Nun waren wir endlich an den Punkt gekommen, an dem wir unsere individuellen Erklärungsmodelle miteinander abstimmen, gemeinsam formulieren und zum Kern unseres Buches machen konnten.

Unser architektonisches Denken entwickelte sich über die Zeit

Wir alle wussten, dass es nicht die eine Architekt-Klausur, das eine Architekt-Zertifikat gibt, das man bestehen oder erwerben kann, um sich anschließend ausgebildeter, diplomierte oder sonstwie zertifizierter Architekt nennen zu dürfen. Im Laufe unseres Informatiker-Lebens hatten wir alle bereits in vielen Rollen gearbeitet. Wir wussten als Entwickler, Tester, Projektleiter oder Designer, dass Architektur viele Gesichter hat, dass der Architektur-Aspekt für viele Rollen – nicht nur für den Architekten selbst – von entscheidender Bedeutung ist. Wir hatten aber auch die Erfahrung gemacht, dass wir neben der fachlichen Aus- und Weiterbildung zunächst einen Reife- und Reflektionsprozess durchleben mussten, bevor wir begannen, uns gegebenen Problem- wie auch Lösungsbetrachtungen aus einem stärker ganzheitlichen Blickwinkel zu nähern – bevor wir begannen, „architektonisch“ zu denken.

Unsere Buchvision

Mit unserem Buch verfolgen wir das primäre Ziel, Lesern Orientierung in der Architektur-Domäne zu ermöglichen. Viele Bücher über Architektur stellen in unseren Augen zu sehr das Thema Technologie in den Vordergrund der Betrachtung. Andere Bücher, die wir kennen, konzentrieren sich auf Architektur-Darstellungen und Nomenklaturen sowie die mit diesen verbundenen Techniken. Wieder andere Bücher betrachten Lösungsmuster für Architektur-Probleme. Und schließlich befassen sich regelmäßig einschlägige Computer-Magazine mit Projekterfahrungsberichten, in denen sehr häufig der Architektur-Aspekt einer

entsprechend vorgestellten Lösung dem Artikel erst Substanz gibt. Es strebte jedoch – in unseren Augen zumindest – keines dieser Werke an, dem Leser eine umfassende Orientierung zum Thema Architektur zu geben. Die uns bekannten Bücher konzentrieren sich jeweils lediglich auf ausgewählte Architektur-Teilgebiete.

Damit stellten wir Autoren uns zwei großen Herausforderungen. Die erste Herausforderung bestand darin, eine Buchstruktur zu entwerfen, welche die uns gleichermaßen wichtigen Aspekte Orientierung, Theorie und Praxis adressierte. Die zweite Herausforderung war für uns, ein Erklärungsmodell der Software-Architektur zu entwerfen und zu beschreiben, das uns im Folgenden erlaubte, die Vieldimensionalität dieses Themas angemessen aufzuarbeiten und als stabilen geistigen Kern für unser Buch zu nutzen. Das Ergebnis dieser ersten und grundlegenden Arbeit an unserem Buch war – wenn man so will – die Architektur des Buches selbst, die ausführlich in Kapitel 1 beschrieben wird und sich ganz grob wie folgt gliedert:

- › Darstellung der Architektur-Dimensionen und eines entsprechenden Ordnungs- und Orientierungsrahmens.
- › Inhaltlich weiterführende Aufarbeitung der Architektur-Dimensionen im Rahmen von Architektur-Theoriekapiteln.
- › Verbindungen zwischen Ordnungsrahmen- wie Theoriekapiteln aus dem Blickwinkel von Projekten in den Fallbeispielkapiteln dieses Buches.

Unsere Herausforderungen

Das vorliegende Buch ist demnach das Resultat unseres Wunsches nach einem Werk, welches den Themenkomplex Architektur sinnvoll strukturiert, an der Praxis orientiert ist sowie entsprechende Praxiserfahrungen vermittelt. Das Buch ist in besonderem Maße technologienutral und zeitlos. Damit zählt dieses Buch für uns zu der Gruppe der Grundlagenwerke, die Ihnen ein stabiles Referenzsystem auch über aktuelle technologische Trends hinaus liefert. Die Aufgabe, die wir uns mit dem Schreiben dieses Buches gestellt hatten, war nicht leicht zu bewältigen und erforderte intensive und tiefe Auseinandersetzung aller Autoren mit dem Thema Architektur – und zwar über das sonst übliche Niveau eher isolierter Betrachtungen hinaus. In der Zeit, in der unser Buch entstand, haben wir sehr viel gelernt. Wir haben miteinander diskutiert und gerungen. Unser Autoren-Team gewann durch die gemeinsame Arbeit am vorliegenden Werk viele wertvolle neue Erkenntnisse und ein gemeinsames Architektur-Verständnis.

Unser Buch

Unser Verständnis von Architektur halten Sie nun in Ihren Händen. Wir hoffen, dass unser Anspruch, das Thema Architektur für Sie zu ordnen, zu erläutern und praktisch zu verankern, Ihnen hilft, sich in Ihrem Berufsleben oder Ihrem Studium mit diesem interessanten und wichtigen Gebiet zu befassen.

Unser Dank

Wir wollen uns an dieser Stelle bei all denen bedanken, die uns für die Arbeit am vorliegenden Werk frei stellten und uns unterstützten. Dies waren unsere Partner und Kinder, unsere Freunde und Kollegen, unsere Arbeitgeber und Vorgesetzte. All denen möchten wir danken, die uns ihre Zeit opferten und immer wieder neue Kraft gaben.

Für die wertvollen Kommentare, Hilfe und Verbesserungsvorschläge möchten wir Kerstin Gans, Dorothee Küpper, Christian Dennler, Martin Fabini, Martin Gottschalk, Peter Jess, Elmar Küpper, Arthur Neudeck, Bernhard Polarzyk, Bernhard Scheffold, Gilbert Semmer, Ralf Steck, Marco Kuhrmann sowie Bernd Oestreich ganz herzlich danken.

Ebenso danken wir Barbara Lühker und Dr. Andreas Rüdinger von Spektrum Akademischer Verlag für ihre Unterstützung.

| Inhaltsverzeichnis

1 Einleitung	1
1.1 Ausgangslage und Zielsetzung des Buches	2
1.2 Was ist Software-Architektur?	8
1.3 Leser-Leitfaden.....	11
1.3.1 Buchaufbau	11
1.3.2 Zielpublikum	15
1.3.3 Kapitelüberblick.....	15
1.3.4 Kapitel im Detail	17
2 Architektonischer Ordnungsrahmen	23
2.1 Motivation.....	24
2.2 Ordnungsrahmen im Überblick	26
2.3 Architekturen und Architektur-Disziplinen (WAS).....	30
2.4 Architektur-Perspektiven (WO)	31
2.5 Architektur-Anforderungen (WARUM)	32
2.6 Architektur-Mittel (WOMIT).....	33
2.7 Organisationen und Individuen (WER).....	36
2.8 Architektur-Vorgehen (WIE)	37
2.9 Zusammenfassung	38
3 Architekturen und Architektur-Disziplinen (WAS)	41
3.1 Klassische Architektur als Ausgangspunkt.....	42
3.2 Von der klassischen Architektur zur Software-Architektur	46
3.3 Architektur und der Systemgedanke	57
3.4 Architektur und die Bausteine eines Systems	62
3.5 Zusammenfassung	68
4 Architektur-Perspektiven (WO).....	71
4.1 Architektur-Ebenen.....	72
4.1.1 Organisationsebene	80
4.1.2 Systemebene.....	81
4.1.3 Bausteinebene	82
4.2 Architektur-Sichten.....	83
4.2.1 Zachman-Framework	94
4.2.2 Reference Model for Open Distributed Processing	97
4.2.3 4+1-Sichtenmodell.....	98
4.3 Zusammenfassung	100

5 Architektur-Anforderungen (WARUM)	103
5.1 Allgemeines	104
5.2 Anforderungen im Überblick	107
5.3 Anforderungen im Detail	111
5.3.1 Organisationsanforderungen	111
5.3.2 Systemanforderungen	113
5.3.3 Bausteinanforderungen	114
5.3.4 Laufzeitanforderungen.....	114
5.3.5 Entwicklungszeitanforderungen	116
5.3.6 Organisatorische Rahmenbedingungen.....	118
5.4 Anforderungen im Architektur-Kontext.....	119
5.5 Zusammenfassung	123
6 Architektur-Mittel (WOMIT).....	125
6.1 Architektur-Prinzipien	128
6.1.1 Prinzip der losen Kopplung.....	130
6.1.2 Prinzip der hohen Kohäsion.....	133
6.1.3 Prinzip des Entwurfs für Veränderung.....	135
6.1.4 Separation-of-Concerns-Prinzip.....	137
6.1.5 Information-Hiding-Prinzip.....	140
6.1.6 Abstraktionsprinzipen.....	142
6.1.7 Modularitätsprinzip.....	145
6.1.8 Rückverfolgbarkeitsprinzip	148
6.1.9 Selbstdokumentationsprinzip.....	148
6.1.10 Inkrementalitätsprinzip	149
6.1.11 Weitere Architektur-Prinzipien.....	150
6.1.12 Zusammenfassung.....	151
6.2 Grundlegende architektonische Konzepte	152
6.2.1 Prozedurale Ansätze.....	153
6.2.2 Objektorientierung	155
6.2.3 Komponentenorientierung	161
6.2.4 Meta-Architekturen und Reflection	164
6.2.5 Generative Erzeugung von Systembausteinen	166
6.2.6 Modellgetriebene Software-Entwicklung.....	170
6.2.7 Aspektorientierung.....	181
6.2.8 Skriptsprachen und dynamische Sprachen	185
6.2.9 Wartung von Software-Architekturen.....	189
6.2.10 Zusammenfassung.....	193
6.3 Architektur-Taktiken, -Stile und -Muster	194
6.3.1 Qualitätsattributzenarien	196
6.3.2 Architektur-Taktiken.....	197
6.3.3 Architektur-Stile.....	199
6.3.4 Architektur-Muster.....	202

6.3.5 Mustersprachen	211
6.3.6 Zusammenfassung.....	215
6.4 Basisarchitekturen	216
6.4.1 Schichtenarchitekturen	217
6.4.2 Datenflussarchitekturen.....	219
6.4.3 Repositories	220
6.4.4 Zentralisierung gegenüber Dezentralisierung.....	221
6.4.5 n-Tier-Architektur.....	224
6.4.6 Rich Client gegenüber Thin Client.....	226
6.4.7 Peer-to-Peer-Architektur.....	228
6.4.8 Publish/Subscribe-Architektur.....	228
6.4.9 Middleware	229
6.4.10 Komponentenplattformen	233
6.4.11 Serviceorientierte Architekturen.....	235
6.4.12 Sicherheitsarchitekturen.....	243
6.4.13 Zusammenfassung	252
6.5 Referenzarchitekturen.....	253
6.5.1 Definition und Bestandteile	254
6.5.2 Einsatz und Vorteile von Referenzarchitekturen	256
6.5.3 Anforderungen an Referenzarchitekturen	257
6.5.4 Arten von Referenzarchitekturen	257
6.5.5 Beispiel für eine Referenzarchitektur	258
6.5.6 Zusammenfassung.....	263
6.6 Architektur-Modellierungsmittel.....	264
6.6.1 Grundlegende Konzepte der Modellierung	265
6.6.2 Unified Modeling Language (UML).....	268
6.6.3 Domain Specific Languages (DSL)	276
6.6.4 Architecture Description Languages (ADL).....	279
6.6.5 Unified Method Architecture (UMA)	283
6.6.6 Zusammenfassung.....	290
6.7 Architekturrelevante Technologien.....	291
6.7.1 Middleware-Systeme	292
6.7.2 Datenbanken und Persistenz von Geschäftsobjekten	297
6.7.3 Datenaustausch und Datentransformation mit XML.....	300
6.7.4 Dynamische Web-Seiten und Web-Anwendungsserver.....	303
6.7.5 Komponentenplattformen	305
6.7.6 Web Services	308
6.7.7 Zusammenfassung.....	310
7 Organisationen und Individuen (WER).....	311
7.1 Allgemeines	312
7.2 Organisationen.....	316
7.3 Individuen.....	321

7.4 Individuen und Gruppen	324
7.5 Architektur und Entscheidungen	328
7.6 Architekt als zentrale Rolle	332
7.7 Zusammenfassung	337
8 Architektur-Vorgehen (WIE).....	341
8.1 Architektur und Entwicklungsprozesse	342
8.2 Architektonisches Vorgehen im Überblick	350
8.3 Erstellen der Systemvision	357
8.4 Verstehen der Anforderungen.....	367
8.5 Entwerfen der Architektur.....	377
8.6 Umsetzen der Architektur	406
8.7 Kommunizieren der Architektur	413
8.8 Anwendungsszenario: Enterprise Application Integration	428
8.8.1 Erstellen der Systemvision.....	430
8.8.2 Verstehen der Anforderungen.....	432
8.8.3 Entwerfen der Architektur	435
8.8.4 Kommunizieren und Umsetzen der Architektur.....	447
9 Risikofallmanagementsystem.....	449
9.1 Überblick.....	450
9.2 Architektur-Anforderungen (WARUM)	451
9.2.1 Systemvision.....	451
9.2.2 Organisationsanforderungen	451
9.2.3 Systemanforderungen.....	452
9.2.4 Bausteinanforderungen	456
9.3 Architekturen und Architektur-Disziplinen (WAS).....	460
9.3.1 Disziplinen	460
9.3.2 Entscheidungen zur Software-Architektur.....	461
9.4 Architektur-Perspektiven (WO)	462
9.4.1 Systemebene.....	462
9.4.2 Bausteinebene	463
9.5 Architektur-Mittel (WOMIT).....	465
9.5.1 Architektur-Prinzipien	465
9.5.2 Grundlegende architektonische Konzepte	467
9.5.3 Generative und generische Verfahren	467
9.6 Organisationen und Individuen (WER).....	470
9.6.1 Organisation	470
9.6.2 Individuen	471
9.7 Architektur-Vorgehen (WIE)	472

10 CRM-Kundendatenbank	473
10.1 Überblick	474
10.2 Architektur-Anforderungen (WARUM)	475
10.2.1 Ausgangssituation.....	476
10.2.2 Anforderungen.....	478
10.2.3 Anwendungsfälle.....	481
10.2.4 Architekturelle Anforderungen	482
10.3 Architekturen und Architektur-Disziplinen (WAS)	484
10.3.1 Disziplinen.....	484
10.3.2 Architektonische Entscheidungen.....	485
10.3.3 Entscheidungen zur Software-Architektur.....	489
10.4 Architektur-Perspektiven (WO)	493
10.5 Architektur-Mittel (WOMIT).....	494
10.6 Organisationen und Individuen (WER)	495
10.7 Architektur-Vorgehen (WIE)	496
10.8 Fazit.....	497
Glossar	499
Abkürzungsverzeichnis	523
Literaturverzeichnis	528
Index	546

| Verzeichnis der Autoren

Oliver Vogel ist zertifizierter IT-Architekt und -Berater bei IBM Global Business Services. Sein Tätigkeitsfeld umfasst die Leitung, Schulung und Beratung von internationalen Projekten und Kunden in diversen Architektur-Themen, wie beispielsweise Architektur-Entwurf, -Umsetzung, -Beurteilung und -Governance. Darüber hinaus beschäftigt er sich intensiv mit modellgetriebener Software-Entwicklung, serviceorientierten Architekturen und Offshoring. Neben Software-Architektur ist Enterprise-Architektur für ihn ein weiterer Interessenschwerpunkt. In seiner Freizeit engagiert er sich als Referent, Dozent und Autor in den genannten Themengebieten.

Ingo Arnold arbeitet als Enterprise-Architekt für die Novartis AG in der Schweiz und ist dort zuständig für die globale Architektur-Planung zentraler Betriebsplattformen. Darüber hinaus gibt er als Dozent Vorlesungen in den Gebieten Software-Architektur, Software Engineering und Software Design Patterns an der Berufsakademie Lörrach sowie der Universität Basel. Auch stellt Ingo Arnold auf internationalen Konferenzen regelmäßig ausgewählte Themen seines Wirkungsbereiches, wie beispielsweise SOA oder Sicherheitsarchitekturen, einem breiten Publikum vor.

Arif Chughtai ist als selbständiger IT-Berater und -Trainer tätig. Sein besonderes Interesse gilt der Verbesserung der technischen Software-Qualität. Software-Engineering gehört deshalb zu den zentralen Gegenständen seiner Arbeit. Er beschäftigt sich dabei insbesondere mit Software-Architektur und -Entwurf, objektorientierter und modellgetriebener Software-Entwicklung sowie serviceorientierten Architekturen. Teile aus den aufgeführten Themenfeldern lässt er als Autor, Dozent und Referent regelmäßig in Fachartikel, Vorlesungen und Vorträge einfließen.

Edmund Ihler war zunächst in der Informatikforschung und später als Architekt und Projektmanager in der Software-Entwicklung für Banken und Versicherungen tätig. Seit 2000 lehrt er als Professor für Informatik an der Hochschule der Medien in Stuttgart mit den Schwerpunkten objektorientierte Software-Modellierung und modellgetriebenes Software Engineering.

Timo Kehrer promoviert an der Universität Siegen und ist Mitarbeiter der Fachgruppe Praktische Informatik an der Fakultät Elektrotechnik und Informatik. Seine Forschungsgebiete sind modellgetriebene Software-Entwicklung, Modell-Versionsmanagement, Modell-Evolution und Analyse von Modell-Repositories. Zeitgleich ist er derzeit noch wissenschaftlicher Mitarbeiter im Studiengang Medieninformatik an der Hochschule der Medien in Stuttgart. Die Schwerpunkte seiner Lehrtätigkeit liegen auf der objektorientierten Programmierung und Modellierung. Während des Studiums, insbesondere im Rahmen seiner Diplomarbeit, spezialisierte er sich auf dem Gebiet der Software-Modellierung.

Uwe Mehlig ist als IT-Architekt bei der IBM Deutschland GmbH im Bereich Global Business Services tätig. Sein aktueller Schwerpunkt liegt auf dem Entwurf von Integrationslösungen basierend auf offenen Standards wie XML, SOAP und Web Services.

Uwe Zdun ist Universitätsassistent in der Distributed Systems Group an der Technischen Universität Wien. Er hat 2001 an der Universität Essen in Informatik promoviert und 2006 an der Wirtschaftsuniversität Wien habilitiert. Seine Forschungsgebiete sind Software Patterns, Software-Architektur, Language Engineering, SOA, verteilte Systeme und Objektorientierung. Er hat viele Software-Systeme entwickelt, darunter Open-Source-Systeme wie die Sprachen Frag und Extended Object Tcl (XOTcl) und kommerzielle Systeme. Darüber hinaus war er in zahlreichen Beratungs- und Forschungsprojekten tätig. Er ist Autor zahlreicher Forschungspublikationen und Koautor des Buches „Remoting Patterns“. Uwe Zdun ist Associate Editor-in-Chief für das Magazin IEEE Software und European Editor für das Journal Transactions on Pattern Languages of Programming (TPLoP).

1 | Einleitung

Dieses Kapitel liefert Motivation und Grundlagen für das Thema Software-Architektur (im weiteren Verlauf auch Architektur). Als Grundstein für die nachfolgenden Kapitel dieses Buches wird zunächst die Bedeutung von Architektur für die Software-Entwicklung erläutert und anschließend aufgezeigt, was sich hinter dem Begriff Architektur im Kontext von IT grundsätzlich verbirgt. Ein Überblick zu Aufbau, anvisiertem Leserkreis und Inhalten des Buches rundet das Kapitel ab. Nach dem Lesen dieses Kapitels kennen Sie die Relevanz von Architektur in der IT und Sie haben eine Vorstellung darüber, was Architektur in diesem Kontext beinhaltet. Des Weiteren kennen Sie unsere Motivation, warum wir dieses Buch geschrieben haben und welche wesentlichen Ziele unser Buch verfolgt. Und Sie kennen die Handhabung dieses Buches.

Übersicht

1.1	Ausgangslage und Zielsetzung des Buches	2
1.2	Was ist Software-Architektur?	8
1.3	Leser-Leitfaden	11

1.1 Ausgangslage und Zielsetzung des Buches

Software wird immer komplexer

Der Wunsch, immer komplexere Anforderungen immer schneller und kostengünstiger bei gleichzeitig hoher Software-Qualität umzusetzen, lässt das Thema Architektur seit einigen Jahren zunehmend ins Blickfeld rücken. Dies gilt nicht nur für kommerzielle Unternehmens-Software, sondern auch für sämtliche anderen IT-Domänen, wie beispielsweise den Embedded-, Mobile- oder Portal-Bereich. Mit der unstrukturierten Art und Weise, wie bis dato häufig Software entwickelt wird, kann sich dieser Wunsch jedoch nicht erfüllen. Nur ein strukturierendes und systematisches Herangehen führt hier zum Erfolg. Architektur ist hierfür ein entscheidender Faktor.

Zunehmende Bedeutung von Software-Architektur

Zukünftig wird der Architektur eine Schlüsselstellung in Entwicklungsprozessen und Technologien zukommen und die Art, wie Software entwickelt wird, wird sich im Vergleich zu heute deutlich verändern. Während heute meist noch die Tätigkeit des manuellen Programmierens das zentrale Element im Selbstverständnis eines Entwicklers ist, wird für den Entwickler der Zukunft die Fähigkeit, mit Architekturen umzugehen und diese zu erstellen, zu einem ganz wesentlichen Berufsaspekt gehören.

Evolution der Software-Entwicklung

Diese sich anbahnenden Veränderungen in der Software-Entwicklung können Sie nachvollziehen, wenn Sie sich die Evolution der Software-Entwicklung bewusst machen. Im Verlauf dieser Evolution arbeitete ein Entwickler zunächst auf der Ebene von Bits und Bytes. Dann verlagerte sich die Entwicklertätigkeit auf immer abstrakteren Ebenen (Assembler, prozedurale Programmiersprachen, objektorientierte Programmiersprachen etc.), welche die Entwickler zunehmend komplexere Dinge tun bzw. immer komplexere Anforderungen umsetzen ließen. Konsequenterweise beinhaltet der bereits begonnene nächste Evolutionsschritt in der Software-Entwicklung modellbasierte und stark architekturzentrierte Konzepte wie MDSD, MDA (siehe Abschnitt 6.2.6) und serviceorientierte Architekturen (SOA) (siehe Abschnitt 6.4.11). Zudem wachsen das Bewusstsein für die technische Qualität von Software und der Wunsch, diese messen zu können. Moderne Software-Entwicklungswerzeuge tragen diesem Wunsch in zunehmendem Maße Rechnung und bieten entsprechende Funktionalität an. Über Metriken (z. B. Anzahl der Abhängigkeiten zwischen Systembausteinen) lässt sich so prüfen, ob Entwickler architekturelle Aspekte ausreichend berücksichtigen.

Unsere Motivation I: Orientierung zu Architektur geben

Die Motivation, ein Buch zum Thema Architektur zu schreiben, entsprang den Herausforderungen und Problemen in der Software-Entwicklung im Zusammenhang mit Architektur, denen wir (das Auto-

ren-Team) in unserer Berufspraxis schon seit Jahren begegnen. Es geht dabei vor allem um zwei Sachverhalte. Zum einen geht es darum, was unter Architektur eigentlich zu verstehen ist. Wir können oft eine Orientierungslosigkeit erkennen, wenn in Projekten das Thema Architektur auf der Tagesordnung steht. Man weiß, Architektur ist eine wichtige Sache und sollte deshalb auch „gemacht“ werden. Was jedoch genau dahinter steckt, ist häufig unbekannt oder nicht im Konsens klar gestellt. Wenn verschiedene Beteiligte von Architektur sprechen, versteht oft jeder etwas anderes darunter. Für den einen repräsentieren schematische Grafiken (Box-and-Lines-Diagramme), dargestellt auf Präsentationsfolien, Architektur. Für den anderen bedeutet Architektur, die Signaturen von Methoden bzw. Funktionen festzulegen. Die Orientierungslosigkeit drückt sich oft in folgenden Fragen aus:

- > Was ist überhaupt Architektur?
- > Was wird von Ihnen als Architekt oder Entwickler erwartet, wenn Sie eine Architektur erstellen sollen?
- > Wie manifestiert sich die Sache, genannt „Architektur“, die Sie ausliefern sollen, eigentlich?
- > Wie können Sie beurteilen, ob es sich tatsächlich um Architektur handelt, wenn Ihnen eine vermeintliche Architektur vorgelegt wird?
- > Wie können Sie die Qualität einer Architektur bestimmen?
- > Wo und wann findet Architektur statt?
- > Warum ist Architektur zu entwickeln?
- > Womit ist Architektur zu entwickeln?
- > Wer ist zuständig für Architektur?
- > Wie ist vorzugehen, um Architektur zu entwickeln?

Zum anderen geht es um die mangelhafte, auf fehlende Berücksichtigung von Architektur zurückzuführende, technische Qualität von Software (z. B. wenn für neue Kundenanforderungen ein Großteil des Quelltextes umgeschrieben werden muss). Wir möchten mit unserem Buch in der IT tätigen Personen Orientierung zum Thema Architektur geben, weil wir beobachten konnten, dass z. B. viele Entwickler und Architekten die eben aufgeführten Fragen beschäftigen und weil wir bisher kein Buch zu Architektur finden konnten, das eine klar strukturierte, umfassende und fokussierte Einführung in das Thema Architektur bietet, zumindest nicht in einer Form und Weise, wie wir uns das oft gewünscht hätten.

Jedes IT-System besitzt eine Architektur. Die Frage ist, ob es sich dabei um eine bewusst geplante Architektur handelt oder diese sich irgendwie unbewusst und zufällig entwickelt hat. Das Ziel muss eine tragfähige

**Unsere Motivation II:
Software-Qualität
verbessern**

**Unser Buch vermittelt
Verständnis für
Architektur-Denken**

Architektur sein. Eine tragfähige Architektur „passiert“ jedoch nicht einfach so, sondern muss bewusst entwickelt werden [Bredemeyer 2002]. Aufgrund der großen Bedeutung von Architektur für die technische Software-Qualität ist es sehr wichtig, Architektur bewusst im Denken zu verankern und dadurch ein Verständnis für Architektur zu entwickeln. Dabei zu helfen, Architektur-Denken zu etablieren und das hierfür nötige Verständnis zu vermitteln, sind die zentralen Ziele unseres Buches.

Am Anfang steht eine „Wunschliste“ ...

Wie erleben Entwickler häufig den Ablauf eines Software-Projekts? Wir sind uns sicher, dass Ihnen die nun folgenden Schilderungen nicht völlig unbekannt sein werden. Es fängt meist damit an, dass die Anforderungen des Kunden rasch in Form einer Art von „Wunschliste“ erfasst werden. Diese „Wunschliste“ ist anschließend ebenso rasch in Quelltext umzusetzen. Zeit, die „Wunschliste“ zu hinterfragen, ist nicht gegeben. Im Fokus steht eine nach außen ansprechend wirkende (nicht notwendigerweise benutzerfreundliche) Benutzerschnittstelle. Damit hält der Kunde schnell etwas Greifbares in der Hand und man kann ihm so zeigen, dass man Herr der Lage ist.

... es folgt ein „Konzept“ ...

Bevor die Punkte der „Wunschliste“ auf die einzelnen Entwickler zum Abarbeiten verteilt werden, wird vom „Chefentwickler“ als Anleitung für die Entwickler ein mehr oder weniger technisches und akzeptiertes „Konzept“ der zu entwickelnden Software auf Basis der „Wunschliste“ verfasst.

... es sind plötzlich Änderungen notwendig ...

Während der Realisierung zeigen sich dann, spätestens wenn sich Anforderungen ändern oder plötzlich neue Anforderungen anstehen, die ersten Unzulänglichkeiten des „Konzepts“.

... es muss vom Konzept abgewichen werden ...

Die Entwickler sind nun gezwungen, im Quelltext vom „Konzept“ abzuweichen und in Eigenregie Maßnahmen zu ergreifen. Diese Behelfs-Maßnahmen sind im „Konzept“ nicht dokumentiert, weil dort selbstverständlich „offiziell“ nichts geändert wird, denn es wurde dem Kunden ja bereits in perfekt inszenierten Präsentationen mit überzeugenden Grafiken verkauft. Zudem ist für Änderungen am „Konzept“ sowieso keine Zeit und auch kein Verständnis des Kunden vorhanden.

... es folgt das Ende, das kommen muss: Big Ball of Mud!

So divergieren das ursprüngliche „Konzept“ und die Quelltext-Realität in zunehmendem Maße. Die Dokumentation des „Konzepts“ ist bald nur noch eine schöne Hülle. Vielleicht einmal vorhandene systematische Strukturen in der Software werden von Flickwerk überdeckt. Im Laufe der Zeit wuchert die Software zu einem undurchschaubaren Gebilde gemäß dem Muster Big Ball of Mud [Foote und Yoder 1999] auch bekannt als „Kludge“ [Bredemeyer 2002] heran:

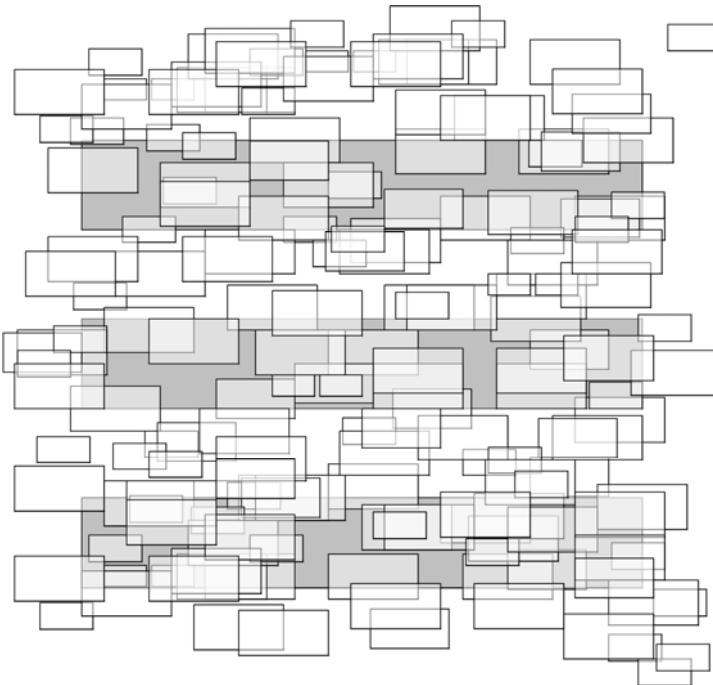


Abb. 1.1-1: Gewucherte Software (Big Ball of Mud).

Es kommt der Moment, ab dem niemand mehr so ganz genau weiß, warum und wie das System überhaupt funktioniert. Man ist einfach nur noch froh, wenn das System läuft. Wartung und Umsetzung neuer Anforderungen werden mit jeder Version der Software zu einem größeren Albtraum, der viel Zeit und Nerven kostet. Wie konnte es so weit kommen? Es war doch ein „Konzept“ vorhanden! Lag es an der „Wunschliste“? Stimmte etwas mit dem „Konzept“ nicht? Wie kann verhindert werden, dass sich eine Software zu einem Big Ball of Mud entwickelt? Diese und andere Fragen haben auch wir uns gestellt und nach Antworten gesucht. Viele der Antworten, die wir in unserem Buch aufführen, ergaben sich im Zusammenhang damit, dass Architektur nicht genügend berücksichtigt wird.

Das eben geschilderte Szenario ist nicht etwa übertrieben, sondern weit verbreitete Realität. Es gibt andere Szenarien, die allesamt im Verlauf in einem Big Ball of Mud münden. Die Mehrzahl der IT-Projekte scheitert mehr oder weniger. Nur ca. 16 % dieser Projekte können für sich in Anspruch nehmen, erfolgreich abgeschlossen worden zu sein [Standish 1994] und das trotz immer fortschrittlicheren Technologien (z. B. schnelle Hardware und mächtige Werkzeuge) und Konzepten (z. B.

Warum musste die Software als Big Ball of Mud enden?

Zahlreiche IT-Projekte scheitern

Objektorientierung und Muster (englisch: *pattern*)). Das Scheitern zeigt sich u. a. in Form von Budget-, Zeitüberschreitungen, Unzufriedenheit des Kunden mit dem ausgelieferten Produkt, bis hin zum Abbruch eines Projekts [Yourdon 2004].

Software-Krise

Seit den 1960er Jahren ist diese Situation bekannt als die sogenannte Software-Krise [Dijkstra 1972], welche erst zu Tage treten konnte durch den immensen Fortschritt der Hardware und den damit verbundenen fast unbegrenzten Möglichkeiten, die sich nun der Software-Entwicklung eröffneten. Die Ursachen für die Software-Krise sind sehr vielfältig. Unzureichende Architekturen gehören dazu. Auf dem Gebiet des Gebäudebaus ist man sich schon lange darüber bewusst, dass es ohne eine vernünftig geplante Architektur früher oder später zu Problemen kommt. Würde man ein Haus errichten, ohne vorher die Architektur festgelegt zu haben, ergäben sich bald Probleme mit Statik, Stabilität, Integration in die kommunale Infrastruktur (z. B. Elektrizität und Wasser) etc. Um bei der Analogie zum Gebäudebau zu bleiben: Häufig werden beim „Bau“ eines IT-Systems zu Anfang ungefähr die Hausmaße festgelegt und, wenn überhaupt, macht man sich rasch noch ein paar Gedanken über Raumaufteilung und Anzahl der Stockwerke. Alles andere (z. B. Statik und Infrastruktur für Strom und Wasser) soll sich dann noch irgendwie im Laufe der „Bauarbeiten“ ergeben. Die „Vorplanung“ wird stichwortartig auf einem „Bierdeckel“ festgehalten und dann wird endlich „losgelegt“: Die Baugrube wird ausgehoben, die Schablonen für die Betonbauteile angefertigt, der Beton gemischt und so fort. Im weiteren Verlauf zeigen sich dann nach und nach fundamentale, nur schwer oder gar nicht korrigierbare Fehler. Unter anderem stellt man fest, dass die Baugrube die falsche Größe für die erstellten Betonbauteile hat. In der Folge bricht eine kontraproduktive operative Hektik aus, in deren Verlauf sich die Situation meist nur noch verschlimmert.

Symptome mangelhafter Architekturen

Fatalerweise zeigen sich die Folgen einer mangelhaften Architektur in der IT nicht selten erst mit erheblicher Verzögerung, das heißt, ernste Probleme treten eventuell erst auf, wenn ein System zum ersten Mal produktiv Eingesetzt wird oder wenn es bereits im Einsatz ist und für neue Anforderungen angepasst werden muss. Eine Architektur, die ungeplant entstanden ist, sich also unbewusst im Laufe der Zeit entwickelt hat, führt zu erheblichen Problemen während der Erstellung, der Auslieferung und dem Betrieb eines Systems. Folgende Symptome können potentiell auf eine mangelhafte Architektur hindeuten:

- > Ergebnisse der Analyse werden nicht bewusst berücksichtigt.
- > Gesamtüberblick fehlt.
- > Komplexität ufert aus und ist nicht mehr beherrschbar.

- > Planbarkeit ist erschwert.
- > Risikofaktoren frühzeitig erkennen, ist kaum möglich.
- > Wiederverwendung von Wissen und Systembausteinen ist erschwert.
- > Flexibilität ist eingeschränkt.
- > Wartbarkeit wird erschwert.
- > Integration verläuft nicht reibungslos.
- > Performanz ist miserabel.
- > Architektur-Dokumentation ist unzureichend.
- > Lernkurve für das Verstehen der Architektur ist sehr hoch.
- > Funktionalität bzw. Quelltext sind redundant.
- > Systembausteine besitzen zahlreiche unnötige Abhängigkeiten untereinander.
- > Entwicklungszyklen (z. B. Übersetzungszeiten) sind sehr lang.

Konkrete Beispiele für die Folgen mangelhafter Architekturen sind:

- > Schnittstellen, die schwer zu verwenden bzw. zu pflegen sind weil sie einen zu großen Umfang besitzen.
- > Quelltext, der an zahlreichen Stellen im System angepasst werden muss, wenn Systembausteine, wie beispielsweise Datenbank oder Betriebssystem, geändert werden.
- > Klassen, die sehr viele ganz unterschiedliche Verantwortlichkeiten abdecken und deshalb nur schwer wiederzuverwenden sind („Monster“-Klassen).
- > Fachklassen, deren Implementierungsdetails im gesamten System bekannt sind.

Folgen mangelhafter Architekturen

Auch wenn eine Architektur gründlich ausgearbeitet wurde, ist das noch keine Garantie dafür, dass keines der oben aufgeführten Probleme auftritt. Dies liegt zum einen daran, dass mangelhafte Architektur nur einer von vielen Faktoren für die Software-Krise ist (andere sind z. B. fehlendes Qualitätsbewusstsein der Benutzer oder eine unzureichende IT-Strategie im Unternehmen) und zum anderen, dass der erfolgreiche Entwurf von Architekturen aufgrund der inhärenten Komplexität von Systemen kein einfaches Unterfangen darstellt, sondern neben einem breiten Fachwissen und fundierter Erfahrung der Verantwortlichen eine Reihe von Aspekten zu beachten ist.

Inhärente Komplexität

Um in einer frühen Phase eines IT-Projekts die Grundzüge einer Architektur einem nicht-technischen Publikum (z. B. Manager und sogar

Pseudo-Architekturen

Chef-Architekten) näher zu bringen und zu „verkaufen“, ist es oft sehr hilfreich, mit sogenannten Pseudo-Architekturen zu arbeiten. Solche Architekturen manifestieren sich meist in Form von Präsentationsfolien mit einer Reihe von Diagrammen und Schlagworten. Es fehlen jedoch alle anderen (technischen) Elemente, die eine echte Architektur ausmachen. Pseudo-Architekturen werden dann zu einem Problem, wenn sie im weiteren Verlauf anstelle einer echten Architektur treten und damit der Begriff Architektur zweckentfremdet wird. Dies liegt darin begründet, dass Pseudo-Architekturen das vorrangige Ziel haben, etwas zu verkaufen aber keinen nennenswerten technischen „Nährwert“ enthalten, also nicht als hinreichendes Erklärungsmodell für ein zu entwickelndes System dienen können und von den Entwicklern nicht wirklich akzeptiert und umgesetzt werden.

1.2 Was ist Software-Architektur?

Architektur ist schwer greifbar

Architektur im Zusammenhang mit Software ist eine relativ junge Disziplin. Bewusstes Architektur-Denken in der Software-Entwicklung ist erst ca. dreißig Jahre alt [Shaw und Garlan 1996]. Aus diesem Grund gibt es widersprüchliche Vorstellungen darüber, was unter diesem Begriff eigentlich zu verstehen ist. Hinzu kommt, dass es auf den ersten Blick, ganz im Unterschied zu physisch greifbaren Dingen, wie Gebäuden, Wohnräumen oder sogar Hardware, nicht unmittelbar ersichtlich wird, dass wirklich (jedes) Software-System eine Architektur benötigt und diese auch in sich trägt. Dies führt dazu, dass Architektur im Zusammenhang mit Software schwer greifbar ist. Trotzdem werden Entwickler, wenn auch oft unbemerkt, in ihrer täglichen Arbeit mit Architektur konfrontiert, weil diese implizit immer ein Aspekt von Software ist und sich nicht eliminieren, allenfalls ignorieren lässt. Was dann jedoch die im vorherigen Abschnitt beschriebenen negativen Konsequenzen nach sich zieht.

Architektur und die Kundenseite

Vor diesem Hintergrund wird verständlich, warum Architektur in einem besonderen Spannungsverhältnis zur Kundenseite respektive dem Management stehen muss. Wenn bereits auf IT-Seite zu Architektur zahlreiche Fragen und Unklarheiten aufgeworfen werden, so ist diese Situation beim Kunden noch viel ausgeprägter. Abgesehen davon, dass es kaum möglich ist, dem Kunden zu vermitteln, dass es so etwas wie Architektur für Software gibt, ist es für diesen nur schwer vorstellbar, welchen unmittelbaren (kommerziellen) Nutzen Architektur für ein Projekt bringen soll. Deshalb ist der Kunde selten dazu bereit, ohne Weiteres Extra-Aufwände im Zusammenhang mit Architektur mitzutragen. Es gibt leider kein Pauschalrezept, wie mit dieser Herausforderung

umgegangen werden kann. Eine Möglichkeit besteht darin, die Kundenseite schon sehr früh auf die mittelfristig eigentlich vermeidbaren höheren (finanziellen) Kosten (beispielsweise aufgrund eines erhöhten Wartungsaufwands) hinzuweisen, die durch eine Vernachlässigung von Architektur verursacht werden.

Architektur ist nicht ausschließlich eine technologische Angelegenheit, sondern beinhaltet zahlreiche soziale und organisatorische Gesichtspunkte (siehe Kapitel 7), die den Erfolg einer Architektur und damit eines gesamten Projektes erheblich beeinflussen können. Aus diesem Grund stehen bei unserer, diesem Buch zugrunde liegenden Vorstellung von Architektur die beteiligten Menschen, insbesondere der Architekt, im Mittelpunkt (siehe Kapitel 2).

Architektur lässt sich nicht so scharf definieren wie beispielsweise Sachverhalte aus Mathematik oder Wirtschaft. Unsere Definition zu Architektur, wie wir sie in Abschnitt 3.2 darlegen werden, ist zu verstehen als intuitive Klarstellung des Architektur-Begriffs auf Grundlage der von uns gemachten Erfahrungen und Eindrücke mit Architektur in unserer täglichen Projektarbeit. Ihre Projektrealität kann sehr wohl eine Definition von Architektur hervorbringen, die in Teilen von unserer abweicht. Zum Begriff Architektur in der IT existieren unzählige Definitionen [SEI 2004]. Daran zeigt sich, dass es eine Herausforderung darstellt, eine Definition zu finden, die allgemein anerkannt wird. Wenn Sie sich vor Augen führen, dass Architektur in verschiedenen Disziplinen (z. B. Software-, Daten-, Sicherheits-Architektur etc.; siehe Kapitel 3) ein Thema ist und ganz unterschiedliche Aspekte (z. B. Anforderungen oder Strukturen; siehe Kapitel 4) bei der Erstellung eines Systems umfasst, wird deutlich, warum eine allgemeingültige Definition, die nicht ausufert, schwer fällt. In den nun folgenden Abschnitten wird der Weg bereitet zu unserer Definition von Architektur.

Unabhängig davon, welche Art von System entwickelt wird, legt eine Architektur ausgehend von den Anforderungen an das System immer die Fundamente und damit die tragenden Säulen, jedoch nicht die Details für das zu entwickelnde System fest [Buschmann et al. 1996]. Architektur handelt also von den Fundamenten, ohne auf deren interne Details einzugehen. Folgende Fragen im Hinblick auf ein System werden durch eine Architektur beantwortet:

- > Auf welche Anforderungen sind Strukturierung und Entscheidungen zurückzuführen?
- > Welches sind die wesentlichen logischen und physikalischen Systembausteine?

Menschen stehen im Mittelpunkt

Zahlreiche Definitionen

Architektur legt keine Details, sondern die tragenden Säulen fest

- > Wie stehen die Systembausteine in Beziehung zueinander?
- > Welche Verantwortlichkeiten haben die Systembausteine?
- > Welche Schnittstellen besitzen die Systembausteine?
- > Wie sind die Systembausteine gruppiert bzw. geschichtet?
- > Was sind die Festlegungen und Kriterien, nach denen das System in Bausteine aufgeteilt wird?

Architektur beinhaltet demnach alle fundamentalen Festlegungen und Vereinbarungen, die zwar durch die fachlichen Anforderungen angestoßen worden sind, sie aber nicht direkt umsetzen.

Wo hört Architektur auf?

Architektur erstreckt sich von der Analyse des Problembereichs eines Systems bis hin zu seiner Realisierung (siehe Kapitel 8). Architektur bewegt sich nicht auf der Abstraktionsebene fein-granularer Strukturen wie Klassen oder Algorithmen, sondern vielmehr auf der Ebene von Systemen, also grob-granularer Strukturen, wie z. B. Komponenten oder Subsystemen (siehe Kapitel 4). Gleichwohl gibt es nicht immer eine scharfe Trennung zwischen den Aspekten fein-granularer und grob-granularer Strukturen, das heißt, die Grenze ist teilweise fließend.

Architektur macht Komplexität überschaubar

Ein wichtiges Charakteristikum von Architektur ist, dass sie Komplexität überschaubar und handhabbar macht, indem sie nur die wesentlichen Aspekte eines Systems zeigt, ohne zu sehr in die Details zu gehen, und es so ermöglicht, in relativ kurzer Zeit einen Überblick über ein System zu erlangen.

Entscheidungen mit systemweiten Auswirkungen

Die Festlegung, was genau die Fundamente und was die Details eines Systems sind, ist subjektiv bzw. kontextabhängig [Fowler 2003]. Gemeint sind in jedem Fall die Dinge, welche sich später nicht ohne Weiteres ändern lassen. Dabei handelt es sich um Strukturen und Entscheidungen, welche für die Entwicklung eines Systems im weiteren Verlauf eine maßgebliche Rolle spielen [Fowler 2003]. Beispiele hierfür sind die Festlegung, wie Systembausteine ihre Daten untereinander austauschen oder die Auswahl der Komponentenplattform (JEE oder .NET). Derartige architekturelle Festlegungen wirken sich, ausgehend von der jeweiligen Architektur-Ebene (siehe Kapitel 4), systemweit aus im Unterschied zu architekturirrelevanten Festlegungen (z. B. bestimmte Implementierung einer Funktion bzw. Methode), die nur lokale Auswirkungen auf ein System haben [Bredemeyer und Malan 2004]. Die architekturellen Strukturen und Entscheidungen sowie Vorgehensweisen, um zu diesen Festlegungen zu kommen, gehören zu den Hauptthemen dieses Buches.

In unserem Buch wird Architektur behandelt, die sich über die Erstellung, Auslieferung und den Betrieb von Software jeglicher Art erstreckt, das heißt, es gibt Berührungs punkte auch zu anderen Architektur-Disziplinen wie z. B. der Daten-Architektur. Architekturen anderer Architektur-Disziplinen werden in unserem Buch nicht im Detail, sondern nur hinsichtlich ihrer Berührungs punkte mit Software-Architektur betrachtet. Wenn im weiteren Verlauf von IT die Rede ist, beschränken wir uns nicht ausschließlich auf Software, sondern wir implizieren damit das ganze Spektrum von IT, in welchem Software nur einen, wenn auch wichtigen Teil darstellt. In Kapitel 3 wird die Diskussion über den Architektur-Begriff vertieft weitergeführt, die soeben aufgestellten Fragen beantwortet und unsere in diesem Buch verwendete Definition bzw. Vorstellung von Architektur entwickelt.

Architektur im Kontext von IT

1.3 Leser-Leitfaden

1.3.1 Buchaufbau

Architektur ist innerhalb der Informatik kein klar abgegrenztes und überschaubares Thema wie beispielsweise formale Sprachen oder Datenstrukturen, sondern ein umfangreicher Themenkomplex, der verschiedene Bereiche der Informatik berührt. Architektur verwendet bekannte Informatikkonzepte (z. B. Schnittstellen) und bringt neue eigene Konzepte mit (z. B. Fassaden-Muster (englisch: *facade pattern*) als Schnittstelle zu Schichten), die bereits bekannte Informatikkonzepte aufgreifen, verwenden und vernetzen.

Architektur ist ein umfangreicher Themenkomplex

Eine der ersten Herausforderungen für uns beim Schreiben des Buches war es, die fundamentale Struktur (also die Architektur) für unser Buch zu entwickeln. Dazu mussten wir Architektur als Themengebiet so strukturieren, dass Sie eine Navigation in der Hand haben, die es Ihnen erlaubt, sich mithilfe unseres Buches das notwendige Wissen effizient anzueignen, ohne dabei im Laufe der Zeit die Orientierung auf diesem großen Themengebiet zu verlieren.

Themengebiet Architektur strukturieren

Die klare und konsequent umgesetzte Strukturierung des Themas Architektur und die Fokussierung auf dieses Thema in seiner ganzen Breite, ohne dabei in Bereiche abzugleiten, die nicht mit Architektur im Zusammenhang stehen, unterscheidet unser Buch von verschiedenen anderen Büchern zu diesem Thema. Diese klare Ausrichtung ist für Sie als Leser ein unschätzbarer Vorteil bei Ihrer Beschäftigung mit diesem umfangreichen Thema.

Unterscheidung zu Konkurrenzwerken

Struktur von Themengebieten

Wir mussten uns zuerst überlegen, wie ein Modell für Themengebiete grundsätzlich aussehen könnte. Wissen liegt nicht isoliert vor, sondern steht in Wechselbeziehung zu seiner Verwendung und den daraus resultierenden Erfahrungen, die ebenfalls Wissen darstellen. Des Weiteren muss Wissen, damit es effizient benutzt werden kann, systematisch geordnet werden. Aus diesen Überlegungen heraus entsprangen die Hauptpfeiler unseres Modells für die Themengebiete:

- > **Ordnung:**
Das System, nach dem Wissen und Erfahrung des Themengebiets geordnet sind.
- > **Wissen:**
Das eigentliche Wissen (Theoriewissen).
- > **Erfahrung:**
Die Anwendung des Wissens (Erfahrungswissen).

Themengebiete Informatik und Architektur

In Abbildung 1.3-1 wird das Themengebiet-Modell auf das Thema Architektur angewendet und unsere Sicht auf Architektur als Themengebiet im Zusammenhang mit Informatik dargestellt:

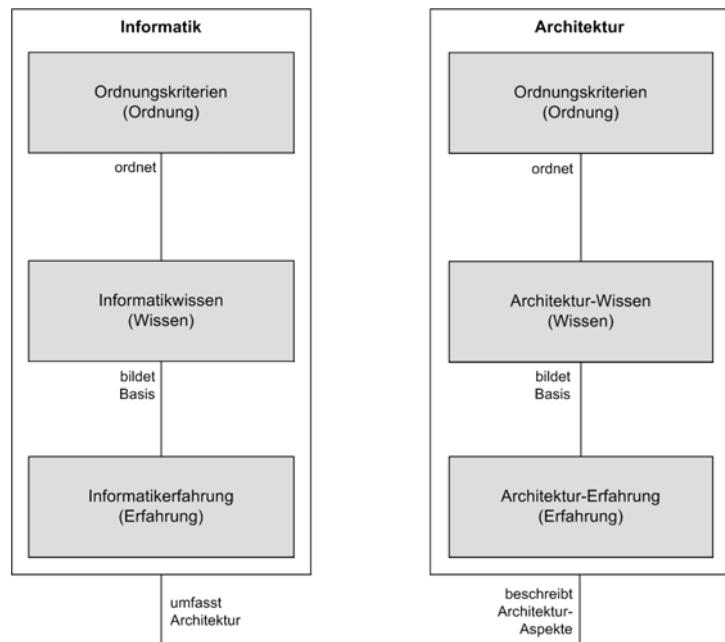


Abb. 1.3-1: Architektur als Themengebiet.

Für das übergeordnete Themengebiet Informatik kommt das Themen-gebiet-Modell wie folgt zum Einsatz:

- > *Ordnungskriterien (Ordnung)*: Verschiedene Möglichkeiten, Informatikwissen systematisch zu ordnen. Je nach betrachtetem Teilgebiet (Algorithmen, Datenstrukturen etc.) der Informatik werden bestimmte Ordnungskriterien, die jedoch der gleichen Kategorie angehören sollten, benutzt.
- > *Informatikwissen (Wissen)*: Sämtliches theoretische Wissen zur Informatik.
- > *Informatikerfahrung (Erfahrung)*: Wendet das theoretische Wissen in einem konkreten Kontext (Projekt) an.

Wird das Themen-gebiet-Modell auf das Themen-gebiet Architektur angewandt, bedeutet dies, dass die für Architektur relevanten Teile bzw. Architektur-Gesichtspunkte des Informatikwissens betrachtet werden:

- > *Ordnungskriterien (Ordnung)*: Ein Ordnungsrahmen basierend auf einfachen Fragerörtern (WAS, WO, WARUM etc.) ordnet Architektur-Wissen. In Kapitel 2 wird der (architektonische) Ordnungsrahmen hergeleitet und beschrieben.
- > *Architektur-Wissen (Wissen)*: Umfasst das theoretische Wissen zu Architektur.
- > *Architektur-Erfahrung (Erfahrung)*: Wendet das theoretische Architektur-Wissen in einem konkreten Kontext (Fallstudien) an.

Bucharchitektur

Unser Modell für Themen-gebiete, angewendet auf Architektur, führt zu folgender Bucharchitektur (siehe Abb. 1.3-2).

Das Buch ist in Anlehnung an das Themen-gebiet-Modell in drei Hauptteile gegliedert und enthält zusätzlich einen Ergänzungsteil mit Anhängen. :

- > *Teil I - Architektur-Überblick und Ordnung*: Gibt einen ersten Überblick zu Architektur und beschreibt den Ordnungsrahmen, der für den zweiten Buchteil die Architektur festlegt.
- > *Teil II - Architektur-Wissen*: Beschreibt detailliert, was Architektur beinhaltet und vermittelt das theoretische Wissen zu Architektur.
- > *Teil III - Architektur-Erfahrung*: Zeigt anhand von Fallstudien die praktische Anwendung des im zweiten Teil vermittelten Architektur-Wissens.
- > *Teil IV - Anhang*: Enthält Glossar, Abkürzungs- und Literaturverzeichnis sowie den Index.

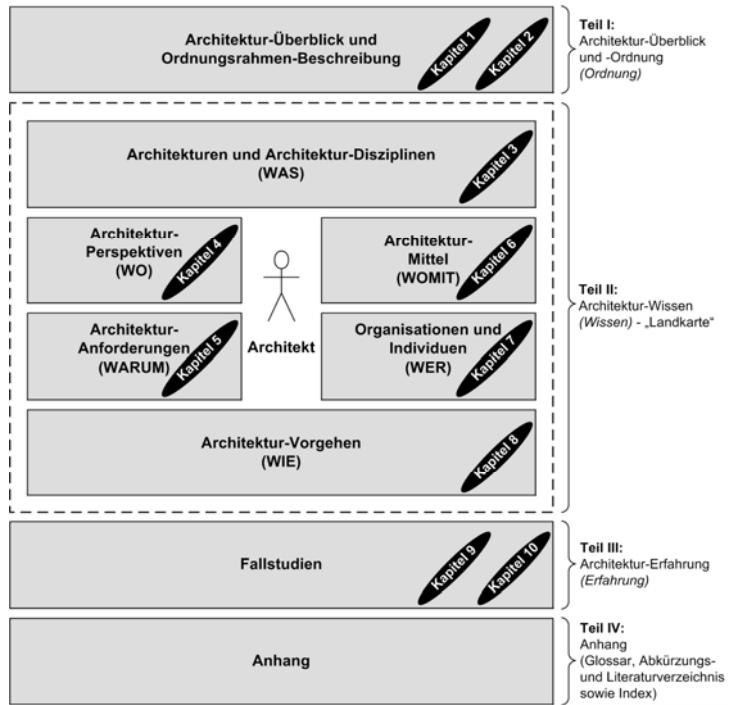


Abb. 1.3-2: Bucharchitektur.

Ihre „Landkarte“ zu Teil II

Der in Abbildung 1.3-2 als „Landkarte“ bezeichnete Teil der Bucharchitektur ist zum einen der architektonische Ordnungsrahmen und zum anderen Ihre Orientierungshilfe für den zweiten Buchteil. Jedes Kapitel aus dem zweiten Buchteil schlägt Ihnen zu Anfang die „Landkarte“ auf und zeigt Ihnen mittels dunkelgrauer Markierung das Gebiet auf der „Landkarte“, in welchem Sie sich mit dem aktuell ausgewählten Kapitel befinden.

Kapitel aus Teil III mit einheitlichen Einleitungsteilen

Die Kapitel des dritten Buchteils beginnen jeweils mit einem einheitlichen Einleitungsteil, der Sie durch das jeweilige Kapitel führt und Beziehe zu den Kapiteln des zweiten Buchteils herstellt und so aufzeigt, wie Architektur-Wissen angewendet wird. Weil die Fallstudien aus ganz unterschiedlichen Kontexten stammen, sind die Kapitel im Anschluss an den jeweiligen Einleitungsteil heterogen aufgebaut.

1.3.2 Zielpublikum

Dieses Buch richtet sich in erster Linie an erfahrenere Software-Entwickler mit praktischer Erfahrung in Entwurf und Implementierung von IT-Systemen für komplexere Problembereiche unter Verwendung etablierter sowie standardisierter Konzepte (z. B. Objektorientierung und Muster) und Technologien (z. B. Java und XML). Informatik-Studenten kann das Buch begleitend zu entsprechenden Studienfächern den Einstieg in das Gebiet Architektur ebnen. Architektur-Experten und IT-Projektleiter haben mit diesem Buch eine Möglichkeit an der Hand, bestimmte Themen gezielt nachzuschlagen und bei Bedarf ihr Architektur-Wissen aufzufrischen bzw. zu ergänzen.

Erfahrenere Software-Entwickler und Studenten

1.3.3 Kapitelüberblick

In Tabelle 1.3-1 werden die einzelnen Buchkapitel in einem Überblick kurz vorgestellt. In Abschnitt 1.3.4 werden sie detaillierter beschrieben.

Tab. 1.3-1: Kapitelübersicht.

Kapitel	Inhalt	Teil	Empfohlen für...
1 Einleitung	Motivation und Einführung	I Architektur-Grundlagen und Ordnung	Einsteiger
2 Architektonischer Ordnungsrahmen	Bucharchitektur		Einsteiger und Fortgeschrittene
3 Architekturen und Architektur-Disziplinen (WAS)	Architektur-Definition		Einsteiger und Fortgeschrittene
4 Architektur-Perspektiven (WO)	Architektur-Modelle		Einsteiger
5 Architektur-Anforderungen (WARUM)	Architektur und Anforderungen		Einsteiger
6 Architekturmittel (WOMIT)	Architekturrelevante Konzepte und Technologien	II Architektur-Wissen	Einsteiger
7 Organisationen und Individuen (WER)	Soziale und organisatorische Aspekte von Architektur resp. Architekten-Rolle		Einsteiger und Fortgeschrittene

Kapitel	Inhalt	Teil	Empfohlen für...
8 Architektur-Vorgehen (WIE)	Architektur im Entwicklungsprozess und Architektur-Wissen angewendet in typischen Anwendungsszenarien	II Architektur-Wissen (Forts.)	Einsteiger und Fortgeschrittene
9 – 10 Fallstudien	Architektur-Wissen angewendet in konkreten Projekten	III Architektur-Erfahrung	Einsteiger und Fortgeschrittene
-	Glossar, Literatur- und Abkürzungerverzeichnis sowie Index	IV Anhang	Einsteiger und Fortgeschrittene

Teil I: Kapitel 2 ist ein Muss

Kapitel 2 ist ein Muss für alle Leser, weil es die Architektur unseres Buches beschreibt und festlegt und damit Voraussetzung für das Grundverständnis unseres Buches ist.

Teil II: Lese-Reihenfolge grundsätzlich beliebig

Die Kapitel aus dem zweiten Teil bauen nicht streng aufeinander auf, sodass sie grundsätzlich in beliebiger Reihenfolge gelesen werden können. Wenn das Thema Architektur für Sie noch mehr oder weniger Neuland darstellt, empfehlen wir Ihnen jedoch neben Kapitel 2 noch folgende Kapitel in dieser Reihenfolge zu lesen: Kapitel 3 bis 5, Abschnitt 6.1 und im Anschluss Kapitel 8.

Teil III: Lese-Reihenfolge beliebig

Die Kapitel aus dem dritten Teil sind unabhängig voneinander und können in beliebiger Reihenfolge gelesen werden.

Buch verwendet Unified Modeling Language (UML)

In zahlreichen Abbildungen dieses Buches wird die Unified Modeling Language (UML) Version 2.0 (UML2) verwendet. Deshalb sollten Leser mit UML vertraut sein. UML wird in diesem Buch nicht grundlegend eingeführt. Der interessierte Leser sei auf [Jeckle et al. 2004 und Oestreich 2006] verwiesen.

Buch vermittelt keine Grundlagen der Software-Entwicklung

Im Zusammenhang mit Architektur erwähnte grundlegende Konzepte der Software-Entwicklung und Technologien werden ebenfalls nicht detailliert betrachtet, sondern ausschließlich hinsichtlich ihrer architektonischen Aspekte behandelt. Im Literaturverzeichnis findet der Leser Angaben zu weiterführenden Informationsquellen.

Lösungen bzw. eine Sammlung von Leitfäden für technologie-spezifische Architektur-Probleme, wie beispielsweise die Trennung von Geschäfts- und Persistenzlogik im Kontext von Java EE, werden Sie in unserem Buch nicht vorfinden. Hierzu existieren bereits eine Reihe empfehlenswerter Werke. Unser Buch hat primär das Ziel, Ihnen eine grundlegende Orientierung zu Architektur zu geben. Diese Orientierung ist unbedingt Voraussetzung dafür, dass Sie in der Lage sind, spezifische Architektur-Probleme zu lösen.

In diesem Buch werden, wenn immer möglich und sinnvoll, deutsche Fachbegriffe und Bezeichner anstelle englischer Begriffe verwendet. Obwohl in der Praxis Englisch die Fachsprache der Informatik ist, haben wir uns für dieses Vorgehen entschieden, weil im Kontext unseres Buches die Vermittlung von Wissen und damit eine gute Verständlichkeit Vorrang hat.

1.3.4 Kapitel im Detail

Der erste Teil des Buches gibt einen ersten Überblick zum Thema Architektur und leitet den architektonischen Ordnungsrahmen her, der für den zweiten Buchteil die Architektur festlegt.

Kapitel 1 liefert Motivation und Grundlagen für das Thema Software-Architektur (im weiteren Verlauf Architektur). Als Ausgangslage für die nachfolgenden Kapitel dieses Buches wird zunächst die Bedeutung von Architektur für die Software-Entwicklung erläutert und anschließend aufgezeigt, was sich hinter dem Begriff Architektur im Kontext von IT grundsätzlich verbirgt. Ein Überblick zu Aufbau, anvisiertem Leserkreis und Inhalten des Buches rundet das Kapitel ab. Nach dem Lesen dieses Kapitels kennen Sie die Relevanz von Architektur in der IT und Sie haben eine Vorstellung darüber, was Architektur in diesem Kontext beinhaltet. Des Weiteren kennen Sie unsere Motivation, warum wir dieses Buch veröffentlicht haben und welche wesentlichen Ziele unser Buch verfolgt. Und Sie kennen die Handhabung dieses Buches.

In Kapitel 2 wird ein Erklärungsmodell zur Beschäftigung mit Architektur vorgestellt. Es bietet Orientierung, indem die wesentlichen Bestandteile von Architektur mittels einfachen Fragewörtern in einem architektonischen Ordnungsrahmen positioniert werden. Der Ordnungsrahmen stellt dabei die Rolle des Architekten in den Mittelpunkt seiner Betrachtung. Ferner dient er als Grundlage für die Vermittlung von Wissen und

Buch gibt grundlegende Orientierung zu Architektur

Deutsche Fachbegriffe und Bezeichner werden verwendet

**Teil I:
Architektur-Überblick
und Ordnung**

**Kapitel 1 –
Einleitung**

**Kapitel 2 –
Architektonischer Ord-nungsrahmen**

Erfahrung im weiteren Verlauf des Buches. Es versetzt Sie in die Lage, über Architektur strukturiert nachzudenken und sich zu orientieren.

Teil II: Architektur-Wissen

Kapitel 3 – Architekturen und Architektur-Disziplinen (WAS)

Der zweite Teil des Buches behandelt essenzielles Architektur-Wissen. Das Wissen wird dabei basierend auf dem zuvor eingeführten architektonischen Ordnungsrahmen strukturiert und vermittelt. Zu Beginn eines jeden Kapitels dieses Buchteils werden zuerst die wesentlichen Konzepte, die in dem Kapitel behandelt und weiter detailliert werden, mittels einer Concept-Map im Überblick und in ihrem Zusammenhang gezeigt.

Kapitel 4 – Architektur- Perspektiven (WO)

Die *WAS-Dimension* des architektonischen Ordnungsrahmens ist Thema des dritten Kapitels. Es vermittelt ein grundlegendes Verständnis von Architektur, indem es aufzeigt, was im Rahmen dieses Buches unter Architektur und damit verbundenen Architektur-Disziplinen zu verstehen ist. Darüber hinaus werden wesentliche Systembausteine und ihre Beziehungen zueinander vorgestellt. Da der Charakter von Systemen und das Denken in Systemen für die Arbeit eines Architekten essenziell sind, wird der Systemgedanke im Kontext von Architektur in diesem Kapitel motiviert. Nach dem Lesen dieses Kapitels sind Sie in der Lage, den allgemeinen Charakter von Architektur zu erklären, einzelne Architektur-Disziplinen zu unterscheiden sowie die wichtigsten Bausteine von Systemen zu differenzieren und ihre Beziehungen darzustellen.

Kapitel 4 befasst sich mit der *WO-Dimension* des architektonischen Ordnungsrahmens. Es erläutert, auf welchen Abstraktionsstufen sich ein Architekt im Rahmen seiner Tätigkeit bewegt und wie sich Architektur auf diesen Abstraktionsstufen manifestiert. Ferner werden architektonische Sichten vorgestellt, die ein Architekt auf den Abstraktionsstufen verwenden kann, um mit den verschiedenen Aspekten und der damit einhergehenden Komplexität einer Architektur besser umgehen zu können. Nach dem Lesen dieses Kapitels sind Sie in der Lage, die relevanten architektonischen Abstraktionsstufen zu unterscheiden und einzusetzen sowie mithilfe von Architektur-Sichten gezielt verschiedene Aspekte einer Architektur zu betrachten und zu bearbeiten.

Kapitel 5 – Architektur- Anforderungen (WARUM)

Die *WARUM-Dimension* des architektonischen Ordnungsrahmens ist Bestandteil von Kapitel 5. Zentrale Elemente der *WARUM-Dimension* sind Anforderungen. Sie umreißen das zu erstellende IT-System und begrenzen den gestalterischen Spielraum des Architekten. Anforderungen treten in unterschiedlichen Arten und auf verschiedenen Architektur-Ebenen auf. Damit ein Architekt seinen gestalterischen Spielraum nutzen kann, muss er die unterschiedlichen Ausprägungen von Anforderungen und ihre Beziehungen zueinander kennen. Dieses Kapitel gibt

einen Überblick über die verschiedenen Anforderungsarten und deren Bezug zu den Architektur-Ebenen. Nach dem Lesen dieses Kapitels können Sie die wichtigsten Anforderungsarten nennen, deren Beziehungen verstehen und sie in den Kontext von Architektur setzen.

Kapitel 6 beschäftigt sich mit der *WOMIT-Dimension* des architektonischen Ordnungsrahmens, indem es grundlegende Konzepte und Techniken aufzeigt, die heutzutage in den „Werkzeugkasten“ eines Software-Architekten gehören. Nach dem Lesen dieses Kapitels haben Sie eine Vorstellung davon erhalten, welche Mittel Sie einsetzen können, um Architekturen zu bewerten, zu beschreiben, zu erstellen und weiterzuentwickeln.

In Kapitel 7 wird die *WER-Dimension* des architektonischen Ordnungsrahmens näher betrachtet und vertieft. Dabei werden organisatorische und soziale Einflussfaktoren aufgezeigt, die die Architektur eines Systems berühren und die Arbeit des Architekten beeinflussen können. Ferner wird grundlegendes Wissen zu Gruppen und ihrer Dynamik vermittelt. Darüber hinaus wird die Rolle des Architekten herausgearbeitet. Durch die Berücksichtigung der in dieser Dimension behandelten Themen sind Sie unter anderem in der Lage, die Relevanz der genannten Einflussfaktoren zu verstehen, die Rolle eines Architekten einzuzuordnen und gruppendifferentielle Prozesse besser zu beachten.

In Kapitel 8 steht die *WIE-Dimension* des Ordnungsrahmens im Mittelpunkt. Zunächst wird für einen Architekten relevantes Wissen zu Entwicklungsprozessen vermittelt. Darauf aufbauend werden die einzelnen Tätigkeiten eines Architekten während der Erarbeitung eines Systems auf einem allgemeingültigen Niveau beschrieben. Abschließend erfolgt eine Konkretisierung dieser Darstellung anhand eines beispielhaften Anwendungsszenarios. Das Anwendungsszenario vernetzt den Ordnungsrahmen sowie den Theorieteil aus dem Kontext eines spezifischen Anwendungsfalls heraus und bietet Ihnen auf diese Weise einen problemorientierten Zugang zu den übrigen Kapiteln.

Während sich die vorhergehenden Kapitel aus Teil II mit dem Wissen um Architektur und einer Verallgemeinerung von Erfahrungen in Form von Anwendungsszenarien beschäftigen, werden in Teil III des Buches zwei Fallstudien aus verschiedenen Bereichen vorgestellt. Die Fallstudien sind zwar jeweils zu einem fiktiven Projekt und nach didaktischen Grundsätzen zusammengestellt, basieren aber auf vielen Erfahrungen aus verschiedenen tatsächlich durchgeföhrten Projekten in verschiedenen Branchen, welche die Autoren in den letzten Jahren sammeln konn-

Kapitel 6 – Architektur-Mittel (WOMIT)

Kapitel 7 – Organisationen und Individuen (WER)

Kapitel 8 – Architektur-Vorgehen (WIE)

Teil III: Architektur-Erfahrung

ten. Allen Fallstudien liegt eine gemeinsame Struktur zugrunde. Diese setzt sich aus einer bei allen Fallstudien identischen Zusammenfassung mit den auf dem Ordnungsrahmen basierenden Dimensionen

- > Architektur-Anforderungen (WARUM),
- > Organisationen und Individuen (WER),
- > Architekturen und Architektur-Disziplinen (WAS),
- > Architektur-Perspektiven (WO),
- > Architektur-Mittel (WOMIT),
- > Architektur-Vorgehen (WIE)

und dem Hauptteil zusammen, wobei jede Fallstudie eine eigene Grundidee für die Strukturierung ihres Hauptteils hat. Das ist so gewollt, um dem unterschiedlichen Charakter der einzelnen Fallstudien Rechnung tragen zu können.

Kapitel 9 – Risikofall- managementsystem

Die Fallstudie Risikofallmanagementsystem in Kapitel 9 bezieht sich schwerpunktmäßig auf die WARUM-Dimension und die WOMIT-Dimension des architektonischen Ordnungsrahmens. Hierzu wird die Entwicklung eines IT-Systems zur Risikoüberwachung für eine Bank vorgestellt. Verschiedene bereits isoliert bestehende IT-Systeme zur Risikoüberwachung sind dabei zu integrieren und die zugehörigen Geschäftsprozesse zu extrahieren. Eine entscheidende Anforderung ist darüber hinaus die leichte Erweiterbarkeit für weitere Geschäftsbereiche der Bank. Der Leser soll durch diese Fallstudie insbesondere den Weg von den Anforderungen zur Strukturierung eines IT-Systems, die Einordnung eines konkreten Projekts in die Dimensionen des Ordnungsrahmens und den Einsatz modellbasierter Verfahren für die Umsetzung von Aspekten eines IT-Systems vertiefen.

Kapitel 10 – CRM-Kundenrepository

Kapitel 10 beschreibt den Entwurf und den Aufbau eines zentralen Kundendatenrepository (KDR) im Rahmen der Einführung eines umfangreichen CRM-Programmes. Voraussetzung für die erfolgreiche Einführung des CRM-Programmes war es, einen konsistenten Bestand an Kundenstammdaten zu haben. Diese Daten sollten allen Systemen des Auftraggebers zentral zur Verfügung gestellt werden. Um Plattformunabhängigkeit zu gewährleisten, sollten alle Daten über Web Services angeboten werden. Eine weitere wichtige Anforderung war die fortlaufende Sicherstellung der Konsistenz der Kundendaten über die angeschlossenen Systeme und die Schaffung einer Möglichkeit, bei Datenverlust in einem System die verlorenen Daten wiederherstellen zu können. Dieses Kapitel betrachtet den Aufbau des zentralen KDR, wobei besonders die Software- und Integrationsarchitekturen näher beleuchtet werden. Die im Anwendungsszenario EAI (siehe Abschnitt 8.8) be-

schriebenen Vorgehensweisen finden hier ihre praktische Anwendung. IT-Architekten, die sich mit der Integration verschiedener Systeme beschäftigen, werden hier interessante Lösungsansätze für immer wieder auftretende Integrationsprobleme finden und erfahrene Architekten werden ihre Erfahrungen bestätigt sehen.

Der Anhang des Buches enthält ergänzende Informationen und Hilfsmittel zur Verwendung des Buches in Form von Glossar, Abkürzungs- und Literaturverzeichnis sowie Index.

Unter www.software-architektur-buch.de finden Sie weiterführende Informationen zum Buch und in Zukunft verschiedene ergänzende Beiträge zum Thema Architektur. Sie sind herzlich dazu eingeladen, eigene Beiträge beizusteuern. Gerne dürfen Sie uns neben Beiträgen Ihre Meinung (Hinweise, Kritik, Lob etc.) zu unserem Buch mit einer E-Mail an autoren@software-architektur-buch.de mitteilen. Wir freuen uns, von Ihnen zu hören!

Teil IV: Anhang

Weitere Informationen unter www.software-architektur-buch.de

2 | Architektonischer Ordnungsrahmen

In diesem Kapitel wird ein Erklärungsmodell zur Beschäftigung mit Architektur vorgestellt. Es bietet Orientierung, indem die wesentlichen Bestandteile von Architektur mittels einfachen Fragewörtern in einem architektonischen Ordnungsrahmen positioniert werden. Der Ordnungsrahmen stellt dabei die Rolle des Architekten in den Mittelpunkt seiner Betrachtung. Ferner dient er als Grundlage für die Vermittlung von Wissen und Erfahrung im weiteren Verlauf des Buches. Es versetzt Sie in die Lage, über Architektur strukturiert nachzudenken und sich zu orientieren.

Übersicht

2.1	Motivation	24
2.2	Ordnungsrahmen im Überblick	26
2.3	Architekturen und Architektur-Disziplinen (WAS)	30
2.4	Architektur-Perspektiven (WO)	31
2.5	Architektur-Anforderungen (WARUM)	32
2.6	Architektur-Mittel (WOMIT)	33
2.7	Organisationen und Individuen (WER)	36
2.8	Architektur-Vorgehen (WIE)	37
2.9	Zusammenfassung	38

Grundlegende Konzepte des architektonischen Ordnungsrahmens

Abbildung 2-1 stellt die grundlegenden Konzepte, welche in diesem Kapitel behandelt werden, vor und visualisiert ihren Zusammenhang.

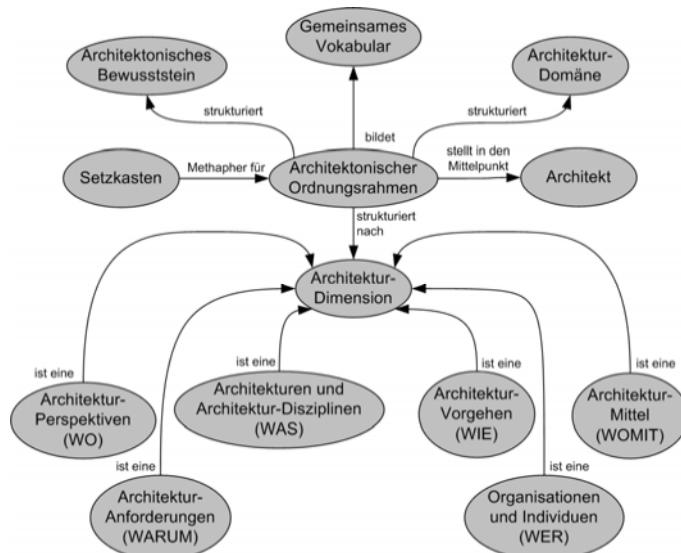


Abb. 2-1: Grundlegende Konzepte des Ordnungsrahmens.

2.1 Motivation

Vielfältiges und dynamisches Umfeld

Architekten arbeiten in einem sehr vielfältigen und dynamischen Umfeld. Neue Technologien drängen auf den Markt, neue Werkzeuge versprechen Effizienz- und Produktivitätssteigerungen, schlanke Methodologien versprechen ein risikoloses Projektmanagement und neue Architektur-Konzepte, wie Serviceorientierung und generative Verfahren, sollen die inhärente Komplexität von IT-Systemen reduzieren. All diese Entwicklungen und Neuerungen muss der Architekt verstehen, einordnen und letztlich beurteilen können, um die Spreu vom Weizen zu trennen und für seine konkrete Problemstellung eine passende Lösung zu wählen. Hierzu müssen Themen entsprechend geordnet, klassifiziert und mit bereits vorhandenem Wissen verglichen werden. Neben der Beherrschung dieser Informationsflut gehören zum Aufgabenfeld eines Architekten auch das Treffen von architektonischen Entscheidungen, das Vorgeben von Richtlinien und das fachliche Führen seines Teams. Darüber hinaus muss er Kundenbedürfnisse aufnehmen, analysieren und tragfähige Architekturen entwerfen. Ferner ist die Auswahl geeigneter Produkte und somit die Kommunikation mit Lieferanten eine wichtige Aufgabe im Rahmen seiner Tätigkeit.

Um in diesem Umfeld erfolgreich zu bestehen, muss man sich dieser vielfältigen Aspekte bewusst sein – sozusagen ein *architektonisches Bewusstsein* entwickeln, das eine Einordnung und Bewertung dieser Aspekte ermöglicht. Ein solches Bewusstsein entwickelt jeder Architekt im Laufe seiner Karriere. Es spiegelt sein Verständnis von Architektur wider und ermöglicht ihm, sich in seinem Arbeitsalltag zu orientieren. Die Güte dieses Bewusstseins ist von strategischer und langfristiger Relevanz, da ein architektonisches Bewusstsein als Grundlage für ein lebenslanges Lernen und somit für ein erfolgreiches Handeln angesehen werden kann. Konkretes Wissen ist zwar wichtig, es ist jedoch erlernbar und kurzlebiger als ein grundlegendes Ordnungsverständnis. Ohne ein Ordnungsverständnis ist Wissen nur schwer positionierbar, anwendbar und beurteilbar. Ebenso wird ein Architekt im Rahmen seines Handelns Erfahrungen machen, die er wie sein konkretes Wissen ordnen muss, um zukünftige Entscheidungen auf Basis seines Erfahrungsschatzes besser und einfacher treffen zu können.

Architektonisches Bewusstsein entwickeln

Architektonisches Bewusstsein sollte wie ein Setzkasten strukturiert sein, in dessen Fächer man zum einen neu Erlerntes und Erfahrenes einordnen und zum anderen bei Bedarf wieder hervorholen kann. Erlerntes bezieht sich dabei auf den Wissensaspekt architektonischen Handelns. Architektur-Prinzipien, -Stile und -Muster, aber auch konkrete Plattformen wie JEE und .NET fallen in diese Kategorie. Erfahrenes umfasst konkrete Erlebnisse aus der Praxis, wie z. B. die Praxistauglichkeit einer zuvor genannten Plattform oder der Umgang mit Spannungen in einem Projektteam. Jedem Fach des architektonischen Setzkastens, um bei unserer vorgestellten Metapher zu bleiben, kommt eine strukturierende beziehungsweise ordnende Aufgabe zu, und alle Elemente eines Faches besitzen gemeinsame Merkmale, die sie von Elementen in anderen Fächern unterscheiden. Dadurch ist man in der Lage, die allgemeinen Merkmale von neu Erlerntem und Erfahrenem aus dem Verständnis der Merkmale des Faches, in das man es eingeordnet hat, abzuleiten.

Architektonisches Bewusstsein strukturieren

Der Aufbau des architektonischen Setzkastens sollte dem vielfältigen Tätigkeitsfeld eines Architekten Rechnung tragen. Daher muss der Setzkasten Architektur ganzheitlich betrachten, sich also z. B. nicht nur auf primär technische Aspekte beschränken. Deshalb ist es wichtig, den Architekten in den Mittelpunkt der Betrachtung zu stellen. Ferner sollte es der Setzkasten ermöglichen, in einem Fach weitere Fächer zu öffnen, um sein Bewusstsein innerhalb eines Faches in weitere strukturierende Bahnen zu lenken und somit über die Zeit zu entwickeln. Daraufhin muss er trotz des Anspruchs nach Ganzheitlichkeit und Erweiter-

Architektonische Strukturierungsmerkmale definieren

Architektonischen Ordnungsrahmen ableiten

barkeit intuitiv und verständlich sein. Nur so wird man effizient mit ihm arbeiten können. Erst wenn man anderen die Gestalt und den Aufbau seines architektonischen Setzkastens, und damit seines Verständnisses von Architektur, in einfachen Worten erklären kann, wird man in der Praxis erfolgreich handeln können.

Der Setzkasten repräsentiert ein prinzipielles Erklärungsmodell der Architektur-Domäne und spannt den ordnenden Rahmen auf, innerhalb dessen sich der Architekt bewegt und handelt. Basierend auf den zuvor definierten Anforderungen nach Ganzheitlichkeit, Erweiterbarkeit, Einfachheit und Verständlichkeit wird in den folgenden Abschnitten ein architektonischer Ordnungsrahmen vorgestellt, der als Setzkasten angesehen werden kann.

2.2 Ordnungsrahmen im Überblick

Basis des architektoni- schen Ordnungs- rahmens

Der im Folgenden vorgestellte Ordnungsrahmen ist durch die Vergegenwärtigung des Alltags eines Architekten und unter Berücksichtigung der im vorherigen Abschnitt formulierten Anforderungen entstanden. Ein Ordnungsrahmen soll einfach sein. Es ist von daher wichtig, sich auf wenige, jedoch wesentliche Hauptdimensionen beziehungsweise Hauptfächer im Sinne der vorgestellten Setzkastenmetapher zu beschränken. Diese Dimensionen sollen jedoch gleichzeitig umfassend genug sein, um die Vielfältigkeit von Architektur beschreiben zu können. Ferner sollen die Dimensionen sinnvoll weiter unterteilt werden können, sodass der Rahmen entsprechend erweitert werden kann. Als weitere Anforderung gilt es noch zu beachten, dass der Ordnungsrahmen leicht verständlich und somit an der Praxis orientiert ist. Was zeichnet nun einen Architekten in der Praxis aus? Im Prinzip gibt er Antworten auf Fragen und Problemstellungen, die ihm Kunden, Team-Mitglieder, Lieferanten oder er sich selbst stellen. Daher ist der Aufbau eines architektonischen Ordnungsrahmens auf Basis von offenen Fragewörtern ein sinnvoller und zielführender Ansatz.

Fragewörter als Haupt- dimensionen

Die Hauptdimensionen des Ordnungsrahmens sind aufgrund dessen:

Tab. 2.2-1: Dimensionen des architektonischen Ordnungsrahmens.

Fragewort	Dimension	Erklärung
WAS	Architekturen und Architektur-Disziplinen	Die WAS-Dimension beinhaltet Architektur-Grundlagen und -Definitionen. Sie bildet hiermit die Basis für die Arbeit als Architekt. Ferner ordnet sie Architektur nach den verschiedenen Tätigkeitsfeldern, in denen Architekten agieren. Die

Fragewort	Dimension	Erklärung
		einzelnen Architektur-Disziplinen tragen in ihrer Gesamtheit zu IT-Systemen bei. In die WAS-Dimension kann der Architekt grundlegendes Wissen und grundlegende Erfahrung einordnen.
WO	Architektur-Perspektiven	Die WO-Dimension umfasst die unterschiedlichen Ebenen, auf denen sich Architektur abspielt, und die Sichten, mit denen Architektur betrachtet werden kann. Die Verwendung verschiedener Perspektiven erlaubt dem Architekten, sich zu einer Zeit auf eine Problemstellung zu konzentrieren. Diese Dimension dient dem Architekten zur Aufnahme verschiedener Betrachtungsweisen.
WARUM	Architektur-Anforderungen	Die WARUM-Dimension widmet sich Anforderungen, die an IT-Systeme im Allgemeinen und Architekturen im Speziellen gestellt werden. Ein Architekt muss in der Lage sein, aus der Fülle von Anforderungen die architektonisch relevanten zu identifizieren und eine Architektur zu entwerfen, die diesen Anforderungen gerecht wird. In die WARUM-Dimension kann ein Architekt die Anforderungen einordnen, die an Architekturen gestellt werden.
WOMIT	Architektur-Mittel	Die WOMIT-Dimension strukturiert die unterschiedlichen architektonischen Mittel, denen sich ein Architekt während seines Handelns bedienen kann. Sie ermöglicht dem Architekten somit die Einordnung verschiedener architektonischer Mittel.
WER	Organisationen und Individuen	Die WER-Dimension behandelt die Rolle des Architekten sowie den Einfluss von Individuen und Organisationen auf Architektur. Dabei werden auch die Wechselwirkungen zwischen Organisationen, Individuen und Architektur näher betrachtet. Die Berücksichtigung dieser Dimension erlaubt dem Architekten, erfolgreich zu handeln. In die WER-Dimension kann der Architekt Wissen und Erfahrung aus seinem sozialen und organisatorischen Umfeld aufnehmen.
WIE	Architektur-Vorgehen	Die WIE-Dimension dient zur Strukturierung von architektonischem Vorgehen. Sie vermittelt die wichtigsten architektonischen Tätigkeiten, die ein Architekt während seiner Arbeit ausübt. Der Architekt kann darin bewährte Vorgehensmodelle ablegen und bei Bedarf wieder hervorholen.

Der Ordnungsrahmen kann wie in Abbildung 2.2-1 visualisiert werden. Diese Abbildung stellt den Architekten in den Mittelpunkt und wird im weiteren Verlauf des Buchs immer wieder verwendet werden, um das aktuelle Thema in den Kontext des Ordnungsrahmens zu stellen und hierdurch dem Leser eine bessere Orientierung zu geben.

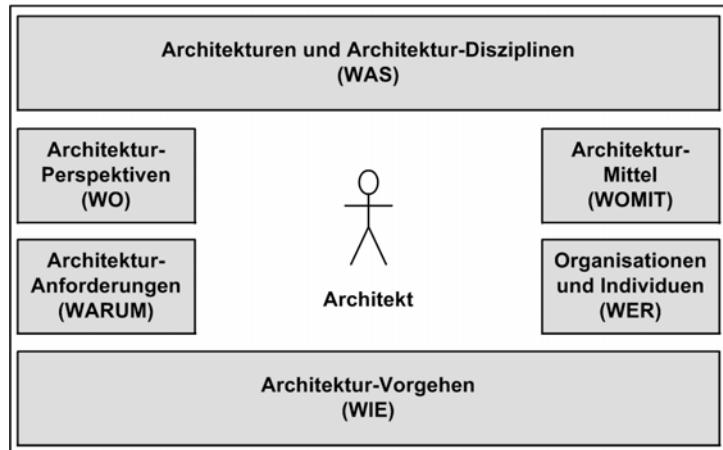


Abb. 2.2-1: Architektonischer Ordnungsrahmen im Überblick.

Ordnungsrahmen in der Praxis

Ein mittels der genannten Fragewörter strukturierter Ordnungsrahmen ermöglicht es, sich grundlegende Fragen zu stellen und sich somit in der Praxis einfach und systematisch zu orientieren. Architektonisches Handeln kann dadurch auf Basis eines Erklärungsmodells erfolgen, indem man sich zu jeder Zeit der unterschiedlichen Dimensionen bewusst ist. So wird man sich im Laufe eines Projekts, also z. B. während der Analyse, dem Entwurf und der Umsetzung, stets fragen, welche Mittel (WOMIT) man auf welche Art und Weise (WIE) einsetzt, um eine bestimmte Anforderung (WARUM) zu realisieren. Der Wunsch nach einer verteilten Architektur wird beispielsweise während der Analyse durch die Durchführung eines Anforderungsanalyse-Workshops (WIE) in einem Anforderungsdokument (WOMIT) festgehalten und im Architektur-Entwurf durch die Verwendung eines entsprechenden Architekturmusters (WOMIT) gewährleistet. Ferner wird man z. B. je nach Architektur-Disziplin (WAS) einen Perspektivenwechsel (WO) vollziehen, um die für die aktuelle Tätigkeit relevanten Aspekte des IT-Systems zu betrachten. Es lassen sich nicht immer alle Aspekte oder Dinge aus der Praxis eindeutig einer Dimension zuordnen, da diese selbst wieder mehrdimensional sind. Vorgehensmodelle, wie der Unified Software Development Process, sind hierfür ein gutes Beispiel. Diese definieren beispielsweise einerseits ein grundsätzliches Vorgehen und

dokumentieren andererseits, mit welchen Mitteln und aus welchen Perspektiven ein System realisiert beziehungsweise betrachtet werden kann. Im Sinne unseres architektonischen Ordnungsrahmens sind solche Vorgehensmodelle prinzipiell der WIE-Dimension zuzuordnen und die übrigen vorgehensneutralen Bestandteile der Vorgehensmodelle den anderen Dimensionen. Letztlich ist es für einen Architekten wichtig, dass er für seine Orientierung Kriterien aufstellt, die ihm eine Zuordnung zu den Dimensionen erlaubt. Dabei sollte er sich immer die grundlegende Frage „Was ist die Essenz des betrachteten Themas?“ beantworten und danach eine Einordnung vornehmen.

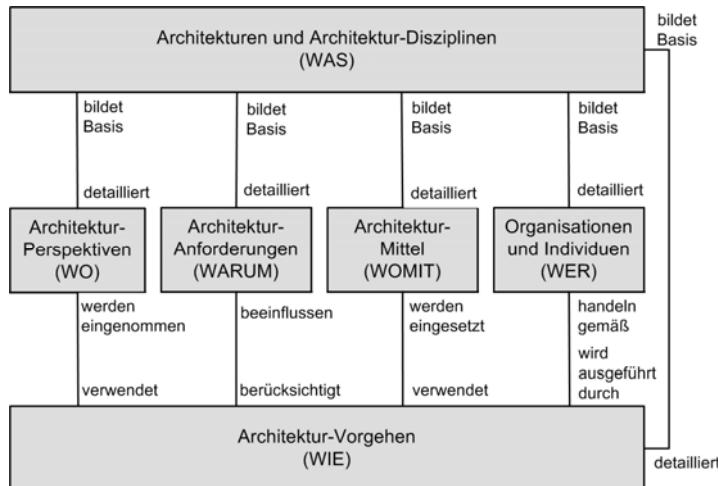


Abb. 2.2-2: Zusammenhänge zwischen den Dimensionen.

Die Dimensionen des Ordnungsrahmens stehen in einem Zusammenhang (siehe Abbildung 2.2-2). Die WAS-Dimension bildet die Basis für alle anderen Dimensionen, da sie grundlegendes Architektur-Wissen und wichtige Architektur-Definitionen beinhaltet. Alle anderen Dimensionen detaillieren die in der WAS-Dimension enthaltenen Grundlagen. Im Rahmen eines Architektur-Vorgehens (WIE-Dimension) werden die Bestandteile der anderen Dimensionen genutzt. Ein Architektur-Vorgehen beschreibt beispielsweise, welche Architektur-Perspektiven (WO-Dimension) einzunehmen und welche Architektur-Mittel (WOMIT-Dimension) einzusetzen sind, um bestimmte Architektur-Anforderungen (WARUM-Dimension) zu erfüllen. Des Weiteren beschreibt das Architektur-Vorgehen, welche Tätigkeiten auszuüben sind. Individuen (WER-Dimension) richten ihr Handeln nach dem Vorgehen aus.

Zusammenhänge zwischen den Dimensionen

Gemeinsames Vokabular und Praxistauglichkeit

Mithilfe dieses Ordnungsrahmens kann ein gemeinsames Vokabular und Verständnis etabliert werden, das die Kommunikation im Team erleichtert. Er ist somit nicht nur einem einzelnen Architekten zur Ordnung seines Bewusstseins von Nutzen, sondern macht auch die Zusammenarbeit mit anderen effizienter, da durch einen gemeinsamen Ordnungsrahmen Missverständnisse verringert werden können. Er kann somit als Katalysator für erfolgreiche Zusammenarbeit im Team dienen. Er repräsentiert selbstverständlich nur ein mögliches Modell, um über Architektur nachzudenken und seine Gedanken zu ordnen. Nach unserer Erfahrung ist das Modell jedoch in der Praxis sehr gut einsetzbar und erleichtert den Arbeitsalltag. In den folgenden Abschnitten werden die einzelnen Dimensionen des architektonischen Ordnungsrahmens im Überblick vorgestellt.

2.3 Architekturen und Architektur-Disziplinen (WAS)

WAS-Dimension repräsentiert Fundament

Die WAS-Dimension widmet sich grundlegendem Architektur-Wissen. Die Bestandteile dieser Dimension ermöglichen es einem Architekten, den Charakter von Architektur zu erklären, Architektur zu definieren und Vergleiche mit anderen Bereichen, wie der Baukunst, herzustellen. Durch ein fundiertes Verständnis dieses Themenkomplexes besitzt ein Architekt die Basis, um sich näher mit den anderen Dimensionen zu beschäftigen.

Vielfältige Aspekte

Der erfolgreiche Entwurf von Architekturen ist aufgrund der inhärenten Komplexität von IT-Systemen kein leichtes Unterfangen. Architekturen müssen heutzutage übliche Anforderungen wie Verteilbarkeit, Verfügbarkeit und hohe Integrierbarkeit würdigen und darüber hinaus eine Basis zur Realisierung funktionaler Anforderungen bieten. Somit stehen Architekten vor der Herausforderung, unterschiedlichste, architektonische Einflussfaktoren, wie funktionale und qualitative Aspekte, zu berücksichtigen und für die konkrete Problemstellung ausreichend zu balancieren. Hierfür ist neben einem fundierten architektonischen Basiswissen auch immer mehr ein tief reichendes Wissen in einem Spezialgebiet notwendig. Für die Integration von IT-Systemen ist beispielsweise ein sehr gutes Verständnis der einzusetzenden Integrationsplattform und möglicher Integrationsansätze, wie nachrichten- oder prozessbasierte Integration, relevant. Aufgrund dessen haben sich unterschiedliche *Architektur-Disziplinen* herausgebildet. Im Laufe seiner Karriere wird man sich häufig für eine dieser Disziplinen als Vertiefungsgebiet entscheiden. Eine Architektur wird dadurch oft als Teamleistung durch die Zusammenarbeit von Architekten der einzelnen Disziplinen entstehen.

Deshalb werden in der *WAS-Dimension* neben einer detaillierten Behandlung der Grundlagen von Software-Architektur auch weitere Architektur-Disziplinen kurz vorgestellt. Dabei werden die einzelnen Disziplinen im Überblick dargestellt, um sie im architektonischen Ordnungsrahmen positionieren und voneinander abgrenzen zu können. Folgende Disziplinen aus der Praxis werden behandelt:

- > Software-Architektur
- > Datenarchitektur
- > Integrationsarchitektur
- > Netzwerkarchitektur
- > Sicherheitsarchitektur
- > System-Management-Architektur
- > Enterprise-Architektur

Architektur-Disziplinen

In Kapitel 3 werden die Inhalte dieser Dimension näher besprochen.

2.4 Architektur-Perspektiven (WO)

Architektonisches Denken und Handeln ist komplex. Psychologische Untersuchungen belegen, dass Menschen gerade einmal 7 ± 2 Informationseinheiten gleichzeitig verarbeiten können [Miller 1956]. Alle Aspekte einer Architektur übersteigen diese Kennzahl um ein Vielfaches. Aus diesem Grund ist es äußerst schwierig, die Bausteine eines Systems, ihre Gruppierung, ihr Zusammenspiel, ihre Verteilung sowie ihr Verhalten zur Laufzeit auf einmal zu erfassen. Um trotz der Beschränkungen des menschlichen Verstandes erfolgreich zu agieren, ist es notwendig, die Komplexität zu reduzieren, indem zu einem Zeitpunkt immer nur ein überschaubarer Teil einer Architektur genauer untersucht wird.

Konzentration auf Perspektiven

Architektur kann sich dabei auf unterschiedlichen Ebenen abspielen. Dabei ist es wichtig, sich stets vor Augen zu halten, auf welcher Ebene man sich gerade befindet. Erst dadurch ist es möglich, die für die betrachtete *Architektur-Ebene* sinnvollen Mittel und Disziplinen einzusetzen. Die möglichen Ebenen reichen von Organisationen über Systeme bis hin zu Bausteinen.

Architektur-Ebenen

Auf jeder Ebene kann man verschiedene *Architektur-Sichten* auf ein System einnehmen. In ihrer Gesamtheit ergeben die Sichten ein komplementäres Bild der umzusetzenden Architektur. *Architektur-Sichten-*

Architektur-Sichten und -Modelle

modelle ermöglichen zu diesem Zweck, Architekturen systematisch und komplexitätsreduzierend zu betrachten. Sie fassen relevante Sichten, aus denen Architekturen betrachtet werden sollten, in einem Modell zusammen und erlauben so ihre ganzheitliche Darstellung. Die *4+1-Sicht* von Kruchten [Kruchten 2000] ist ein Beispiel für ein Architektur-Sichtenmodell. Architektur-Rahmenwerke, wie das *Zachman-Framework* [Zachman 1987] und das *Reference Model for Open Distributed Processing (RM-ODP)* [ISO10746 1998] beinhalten ebenfalls Architektur-Sichtenmodelle.

Kapitel 4 diskutiert die einzelnen Architektur-Ebenen und -Sichten. Ferner werden die verschiedenen Sichten der genannten Architektur-Modelle näher betrachtet.

2.5 Architektur-Anforderungen (WARUM)

Architektur ist kein Selbstzweck

Informationstechnologie (IT) ist für Firmen ein wesentliches Mittel, um ihre Geschäftsstrategien zu realisieren und ihr operationales Geschäft zu unterstützen. IT-Systeme und somit auch Architekturen werden also nicht zu ihrem Selbstzweck, sondern stets vor dem Hintergrund eines konkreten Geschäftsnutzens entworfen. Primäre Motivation für Architektur ist also nicht technologische Eleganz, sondern der konkrete und langfristige Mehrwert für den Kunden. Dieser Mehrwert kann selbstverständlich nur erreicht werden, wenn die an das IT-System gestellten funktionalen Anforderungen erfüllt werden. Allerdings wird ein IT-System, das zwar die funktionalen Anforderungen befriedigt, aber nicht-funktionale Anforderungen nicht entsprechend würdigt, keinen wirklichen Nutzen für den Kunden haben. Ein E-Commerce Shop, der eigentlich allen funktionalen Anforderungen gerecht wird, jedoch beim gleichzeitigen Zugriff von einer hohen Benutzerzahl zusammenbricht, wird die eigentliche Geschäftsstrategie nicht unterstützen und letztlich keinen Mehrwert liefern können.

Arten von Anforderungen

Ein Architekt muss demnach sicherstellen, dass die an ein IT-System gestellten Anforderungen durch die dem IT-System zugrunde liegende Architektur unterstützt werden. Daher ist es für einen Architekten unerlässlich, unterschiedliche Anforderungsarten und ihre Implikationen auf Architektur zu kennen. Prinzipiell können funktionale und nicht-funktionale Anforderungen unterschieden werden. Darauf aufbauend können folgende Anforderungsarten genannt werden:

- > Organisationsanforderungen
- > Systemanforderungen
- > Bausteinanforderungen
- > Entwicklungszeitanforderungen
- > Laufzeitanforderungen
- > Organisatorische Rahmenbedingungen

Aufgabe der *WARUM-Dimension* des Ordnungsrahmens ist es, die verschiedenen Anforderungsarten zu identifizieren und zu erklären. Erst wenn man sich den unterschiedlichen *Anforderungen* bewusst ist und diese in sein architektonisches Handeln einbezieht, kann man zielgerichtete und lösungsorientierte IT-Systeme entwerfen. Dies ist von großer Bedeutung, da der Erfolg einer Architektur und somit eines Projekts hiervon abhängt. Untersuchungen belegen, dass 37 % aller gescheiterten Projekte auf den falschen Umgang mit Anforderungen zurückzuführen sind [Larman 2002 und Davis 1993]. In Kapitel 5 werden diese unterschiedlichen Anforderungsarten detailliert besprochen.

Relevanz von Architektur-Anforderungen

2.6 Architektur-Mittel (WOMIT)

Diese Dimension widmet sich der Frage, womit ein Architekt seine Lösungen entwirft und umsetzt. Im Sinne der Setzkastenmetapher enthält dieses Fach sehr viele kleinere Unterfächer, um die große Anzahl *architektonischer Mittel* zu strukturieren und die Orientierung zu erleichtern. Ein Architekt wird im Laufe seiner Karriere immer wieder neue Mittel in diesen Fächern ablegen und veraltete entfernen. Die Lebensdauer ergibt sich dabei aus der Relevanz der architektonischen Mittel. Das Spektrum der möglichen Architektur-Mittel reicht dabei von elementaren Prinzipien bis hin zu konkreten Technologien.

Bestandteile der WOMIT-Dimension

Es existieren elementare Mittel, deren Einsatz und Berücksichtigung von großer Relevanz zur Etablierung erfolgreicher Architekturen sind. Diese Mittel gehören zu der Kategorie der *Architektur-Prinzipien*. Ein Mittel dieser Kategorie ist das *Separation-of-Concerns-Prinzip*, das darauf abzielt, Verantwortlichkeiten von Bausteinen klar zu trennen. So sollte z. B. ein Baustein zur Visualisierung von Daten nicht auch für deren Speicherung in einer Datenbank zuständig sein. *Architektur-Prinzipien* sind von langfristiger Bedeutung und sollten jegliches architektonisches Handeln begleiten. Sie verkörpern grundlegende Architektur-Erfahrungen.

Prinzipien

Grundlegende Konzepte

Um sicherzustellen, dass architektonische Prinzipien auch in eine Architektur einfließen, kann der Architekt auf grundlegende *Konzepte* zurückgreifen, die diese Prinzipien entsprechend unterstützen. Dabei ist es wichtig, sich die verschiedenen Konzepte vor Augen zu führen und je nach Problemstellung das passende auszuwählen. Zu den architektonischen Konzepten gehören grundlegende Konzeptions- und Realisierungsparadigmen, wie Objekt- und Komponentenorientierung. Des Weiteren sind ganzheitliche auf Modellierung und Generierung beruhende Mittel wie die modellgetriebene Software-Entwicklung respektive die Model Driven Architecture [OMG 2007a] Bestandteile dieser Subdimension.

Taktiken, Stile und Muster

Neben der Berücksichtigung elementarer Prinzipien und Konzepte ist es empfehlenswert, bewährte Architektur-Lösungen in seinem Werkzeugkasten zu haben, um diese für ähnliche Problemstellungen wieder verwenden zu können. Diese auf den *Architektur-Prinzipien* aufbauenden Lösungen gehören zur Familie der *Architektur-Taktiken*, -*Stile* und -*Muster*. Eine *Architektur-Taktik* ist eine Hilfe für den Architekten, eine erste Idee zu einem Entwurfsproblem zu erhalten. Diese Idee kann er weiter ausarbeiten. Dazu kann er z. B. *Stile* und *Muster* als weitergehende Mittel verwenden. Ein *Architektur-Stil* dokumentiert einen erprobten und erfolgreichen Weg, eine Architektur zu strukturieren. Jeder Stil besitzt bestimmte Charakteristika und dient als Vorlage für den Entwurf der eigentlichen Architektur. Ein *Architektur-Stil* ist auch ein effizientes Dokumentations- und Kommunikationswerkzeug, da die Eigenschaften des eingesetzten Stils unabhängig vom eigentlichen Zweck des Systems verstanden werden können. Es gibt verschiedene Möglichkeiten, *Architektur-Stile* zu dokumentieren. Eine bewährte und empfehlenswerte Form ist die Dokumentation als *Architektur-Muster*. Ein *Architektur-Muster* beschreibt *Architektur-Stile* anhand einer allgemeinen Struktur. Einen wesentlichen Beitrag zu diesem Bereich haben die Autoren von POSA1 und POSA2 geleistet [Buschmann et al. 1996, Schmidt et al. 2000]. Ein in Musterform beschriebener *Architektur-Stil* ist z. B. das Schichten-Architektur-Muster (*Layers*). Dieses dokumentiert die Anordnung von Systembausteinen auf unterschiedlichen Ebenen, sodass eine klare Trennung der Verantwortlichkeiten erreicht und eine monolithische Architektur vermieden wird [Buschmann et al. 1996]. Die klassische Anordnung von Präsentations-, Geschäfts- und Persistenz-Logik auf unterschiedlichen Schichten ist eine bekannte Anwendung dieses Musters. Architektur-Stile und -Muster sind ähnliche Mittel, die sich in ihrer Beschreibungsform unterscheiden.

Als weitere Mittelart können *Basisarchitekturen* genannt werden. Basisarchitekturen setzen die bisher genannten Architektur-Mittel in einem größeren Kontext ein. Beispiele für solche Basisarchitekturen sind:

- > Schichtenarchitekturen
- > Datenflussarchitekturen
- > n-Tier-Architekturen
- > Middleware-Architektur
- > Rich-Client-Architektur
- > Thin-Client-Architektur

Basisarchitekturen

Durch die Kenntnis dieser Basisarchitekturen kann ein Architekt sein Architektur-Wissen erweitern und schneller zu einer probaten Software-Architektur gelangen.

Architekturen komplexer Systeme müssen mehrere unterschiedliche Architektur-Probleme lösen beziehungsweise entsprechend ausbalancieren. Aus diesem Grund werden mehrere *Architektur-Muster* eingesetzt. Darüber hinaus sind *Architektur-Muster* problembereichsneutrale Architektur-Mittel, dass heißt, sie adressieren z. B. nicht die spezifischen Charakteristika einer Call-Center-Architektur. Um eine Lösung für solch eine Architektur zu entwerfen, reicht es also nicht aus, sich nur auf *Architektur-Muster* zu verlassen. Es ist vielmehr für einen Architekten wichtig, komplett Architektur-Lösungen als Referenzen in seinen Werkzeugkasten aufzunehmen. Solche *Referenzarchitekturen* beschreiben Lösungen, die für einen bestimmten Problembereich unter Verwendung unterschiedlicher *Architektur-Stile* beziehungsweise *Architektur-Muster* entworfen wurden. *Referenzarchitekturen* spiegeln somit den größten Wiederverwendungsgrad architektonischen Wissens und architektonischer Erfahrung wider.

Referenzarchitekturen

Für den Erfolg und die Akzeptanz einer Architektur ist es von großer Bedeutung, dass sie von allen Beteiligten (Kunde, Projektleiter, Software-Entwickler etc.) verstanden und getragen wird. Deshalb ist eine wichtige Aufgabe eines Architekten, seine Ideen und Ansätze zu kommunizieren und entsprechend zu modellieren. Zu diesem Zweck muss er die Architektur mit adäquaten Mitteln ausdrücken. Diese Mittel können je nach Zielgruppe variieren. Für eine Angebotspräsentation mag es beispielsweise ausreichend sein, die wesentlichen Bausteine einer Architektur mit grafischen Elementen zu visualisieren. Im Rahmen des Architektur-Entwurfs sind jedoch ausdrucksstärkere Mittel notwendig, um Missverständnisse auszuschließen und alle wesentlichen

Modellierungsmittel

Architektur-Aspekte, wie z. B. die Struktur und die Dynamik einer Architektur, zu würdigen. Die in diesem Zusammenhang eingesetzten Mittel dienen der Modellierung der Architektur und gehören zur Familie der *Architektur-Modellierungsmittel*. Ein weitverbreitetes, standardisiertes Modellierungsmittel ist beispielsweise die Unified Modeling Language (UML) der Object Management Group [OMG 2005c].

Technologien

Abgesehen von Architektur-Strukturen ist die Wahl von Technologien, die den Architektur-Entwurf in der eigentlichen Umsetzung tragen und unterstützen, ein wichtiger Garant für eine erfolgreiche Architektur. Deshalb sollte sich ein Fach des Werkzeugkastens diesen *Basistechnologien* widmen. Ein Architekt wird insbesondere in diesem Bereich sehr häufig neue Technologien in seinen Werkzeugkasten aufnehmen und überholte daraus entfernen. In dieser Subdimension lassen sich beispielsweise Datenbanken, Transaktionsmonitore oder Middleware ansiedeln. Des Weiteren ist es für die erfolgreichere Umsetzung einer Architektur von großer Bedeutung, mögliche Zielplattformen zu kennen und ihre Stärken und Schwächen bei der eigentlichen Architektur-Projektion zu berücksichtigen. Zielplattformen, wie z. B. Suns Java Enterprise Edition oder Microsofts .NET, gehören zur Kategorie der *Komponentenplattformen* und sind wesentliche Gestaltungsmittel zur Umsetzung von architektonischen Anforderungen, wie z. B. Skalierbarkeit, Verfügbarkeit und Zuverlässigkeit, indem sie elementare Basisfunktionalität bereitstellen.

Kapitel 6 widmet sich diesem Themen- komplex

Die Vergegenwärtigung und der bewusste Einsatz dieser Mittel erleichtern das architektonische Handeln und tragen erheblich zum Architektur-Erfolg bei. Kapitel 6 stellt die WOMIT-Dimension genauer vor und geht auf einzelne Architektur-Mittel näher ein. In diesem Buch beschränken wir uns momentan auf primär IT-bezogene Mittel. Es ist jedoch auch denkbar, in dieser Dimension andere Mittel, wie Präsentations- und Gesprächstechniken, die dem Architekten bei der Kommunikation mit Interessenvertretern nützlich sind, anzusiedeln.

2.7 Organisationen und Individuen (WER)

Interaktion und Kom- munikation als Archi- tekture-Leistung

Architekturen entstehen durch Menschen. Ein Architekt interagiert und kommuniziert mit vielen unterschiedlichen Personengruppen, um eine Architektur zu entwerfen. So wird er z. B. eng mit dem Kunden und den Endnutzern des zu entwickelnden Systems zusammenarbeiten, um aus den an das System gestellten Anforderungen die architektonisch relevanten zu extrahieren. Des Weiteren ist er erster Ansprechpartner für

Projektleiter, um diese bei der Erstellung von Projektplänen und Aufwandsschätzungen zu unterstützen. Darüber hinaus führt er Projektteams fachlich und agiert als Kommunikator und Motivator der umzusetzenden Architektur.

Um diese Aufgaben erfolgreich zu bewältigen, benötigt ein Architekt mehr als nur fundierte Kompetenzen in fachlichen und methodischen Themen. Er muss vielmehr auch über ausgeprägte soziale Fähigkeiten verfügen. Eine technisch noch so gute Architektur-Idee wird nicht realisierbar sein, wenn ein Architekt sein Team und seinen Kunden nicht für die Idee gewinnen kann. Den sozialen Kompetenzen kommt heutzutage bei der Betrachtung der Architektenrolle leider noch zu wenig Beachtung zu, obwohl Melvin Conway bereits 1968 die These aufgestellt hat, dass eine Architektur wesentlich durch organisatorische Einflüsse geprägt wird [Conway 1968]. Es ist von großer Bedeutung, sich dieser organisatorischen Einflüsse und der benötigten sozialen Kompetenz bewusst zu sein. Erst hierdurch wird aus einem technischen Spezialisten ein Architekt.

Die *WER-Dimension* adressiert diese sozialen Kompetenzen und zeichnet somit die Rolle des Architekten in Organisationen und Teams. Dabei werden zum einen allgemeine Themen wie gruppendiffusiv-dynamische Prozesse, Faktoren für gut funktionierende Teams und die Interdependenzen von Organisationen und Teams behandelt. Zum anderen werden aber auch Themen vorgestellt, die aus konkreten Projekterfahrungen hervorgegangen sind. Hierzu gehören zum Beispiel Organisationsmuster (englisch: *organizational patterns*). Organisationsmuster beschreiben erfolgreiche Möglichkeiten der Zusammenarbeit von Rollen in Projekten [Coplien und Harrison 2004]. Eine ausführliche Behandlung dieser Themen findet sich in Kapitel 7.

2.8 Architektur-Vorgehen (WIE)

Architekten agieren mit dem Ziel, eine Architektur zu entwerfen, die als Fundament zur Realisierung eines Systems dienen kann. Um dieses Ziel zu erreichen, können sie auf verschiedene architektonische Mittel zurückgreifen, ihr Abstraktionsniveau variieren und mit unterschiedlichen Partnern wie Projektleitern, Entwicklern und Analysten kommunizieren. Die Berücksichtigung dieser Möglichkeiten garantiert jedoch noch nicht, dass das Ziel auch erreicht wird. Selbst wenn man ein System einmal erfolgreich realisiert hat, bedeutet dies noch nicht, dass man beim nächsten Mal den gleichen Erfolg haben wird. Erst wenn man in

Relevanz sozialer Kompetenzen

Bestandteile der WER-Dimension

Systematisches und wiederholbares Handeln

der Lage ist, sein architektonisches Handeln zu systematisieren und zu wiederholen, wird man langfristig erfolgreich sein. Aus diesem Grund ist es von großer Bedeutung, sich bewährten Lösungswegen bewusst zu sein und diese wiederholt anwenden zu können. Diesen erfolgreichen Lösungswegen beziehungsweise der Frage „*Wie gehe ich vor, um eine Architektur zu entwerfen und umzusetzen?*“ widmet sich die *WIE-Dimension*.

Allgemeines Vorgehen und Anwendungsszenarien

Zu diesem Zweck wird zum einen ein allgemeines architektonisches Vorgehensmodell vorgestellt, zum anderen gehören Leitfäden für konkrete Problemstellungen in Form von Anwendungsszenarien zu dieser Dimension.

Architektur-Tätigkeiten

Das architektonische Vorgehensmodell beinhaltet dabei folgende Tätigkeiten, die bei der Gestaltung einer Architektur durchgeführt werden:

- > Erstellen der Systemvision.
- > Verstehen der Anforderungen.
- > Entwerfen der Architektur.
- > Umsetzen der Architektur.
- > Kommunizieren der Architektur.

Die Tätigkeiten können dabei im Sinne eines iterativen Vorgehens mehrfach ausgeübt werden.

Tätigkeitsabhängige Relevanz anderer Dimensionen

Je nachdem, welche Tätigkeit gerade durchgeführt wird, wirken Bestandteile der anderen *Dimensionen* unterschiedlich stark auf die Architektur ein. So sind z. B. je nach Tätigkeit verschiedene Mittel und Perspektiven anzuwenden. Zum *Verstehen der Anforderungen* ist es beispielsweise besonders wichtig, aus den an das System gestellten Anforderungen die architektonisch relevanten auszuwählen.

Dieses Themengebiet wird in Kapitel 8 näher besprochen.

2.9 Zusammenfassung

Zusammenfassung: architektonischer Ordnungsrahmen

- > Der architektonische Ordnungsrahmen strukturiert Architektur entlang den Dimensionen WAS, WO, WARUM, WER und WIE.
- > Die WAS-Dimension (Architekturen und Architektur-Disziplinen) beinhaltet Architektur-Grundlagen und -Definitionen. Sie bildet hiermit die Basis für die Arbeit als Architekt.

- > Die WO-Dimension (Architektur-Perspektiven) umfasst die unterschiedlichen Ebenen, auf denen sich Architektur abspielt, und die Sichten, mit denen Architektur betrachtet werden kann.
- > Die WARUM-Dimension (Architektur-Anforderungen) widmet sich Anforderungen, die an IT-Systeme im Allgemeinen und Architekturen im Speziellen gestellt werden.
- > Die WOMIT-Dimension (Architektur-Mittel) strukturiert die unterschiedlichen architektonischen Mittel, deren sich ein Architekt während seines Handelns bedienen kann.
- > Die WER-Dimension (Organisationen und Individuen) behandelt die Rolle des Architekten sowie den Einfluss von Individuen und Organisationen auf Architektur.
- > Die WIE-Dimension (Architektur-Vorgehen) dient zur Strukturierung von architektonischem Vorgehen. Sie vermittelt die wichtigsten architektonischen Tätigkeiten, die ein Architekt während seiner Arbeit ausübt.

3 | Architekturen und Architektur-Disziplinen (WAS)

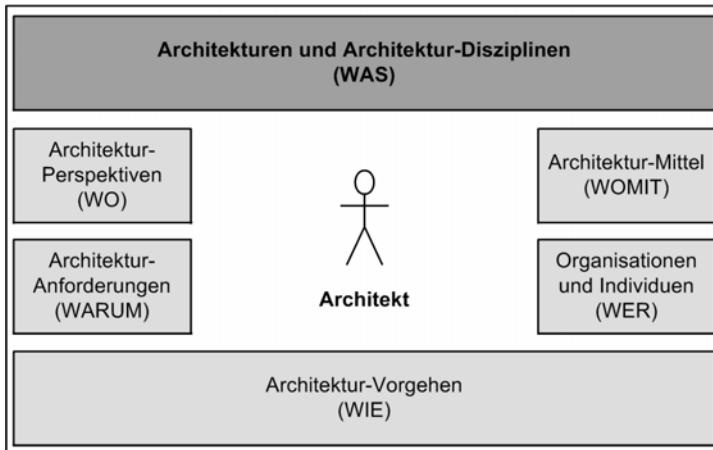


Abb. 3-1: Positionierung des Kapitels im Ordnungsrahmen.

Dieses Kapitel befasst sich mit der *WAS-Dimension* des architektonischen Ordnungsrahmens. Es vermittelt ein grundlegendes Verständnis von Architektur, indem es aufzeigt, was im Rahmen dieses Buches unter Architektur und damit verbundenen Architektur-Disziplinen zu verstehen ist. Darüber hinaus werden wesentliche Systembausteine und ihre Beziehungen zueinander vorgestellt. Da der Charakter von Systemen und das Denken in Systemen für die Arbeit eines Architekten essenziell sind, wird der Systemgedanke im Kontext von Architektur in diesem Kapitel motiviert. Nach dem Lesen dieses Kapitels sind Sie in der Lage, den allgemeinen Charakter von Architektur zu erklären, einzelne Architektur-Disziplinen zu unterscheiden sowie die wichtigsten Bausteine von Systemen zu differenzieren und ihre Beziehungen darzustellen.

Übersicht

3.1	Klassische Architektur als Ausgangspunkt	42
3.2	Von der klassischen Architektur zur Software-Architektur	46
3.3	Architektur und der Systemgedanke	57
3.4	Architektur und die Bausteine eines Systems	62
3.5	Zusammenfassung	68

Grundlegende Konzepte der WAS-Dimension

Abbildung 3-2 stellt die grundlegenden Konzepte, welche in diesem Kapitel behandelt werden, vor und visualisiert ihren Zusammenhang.

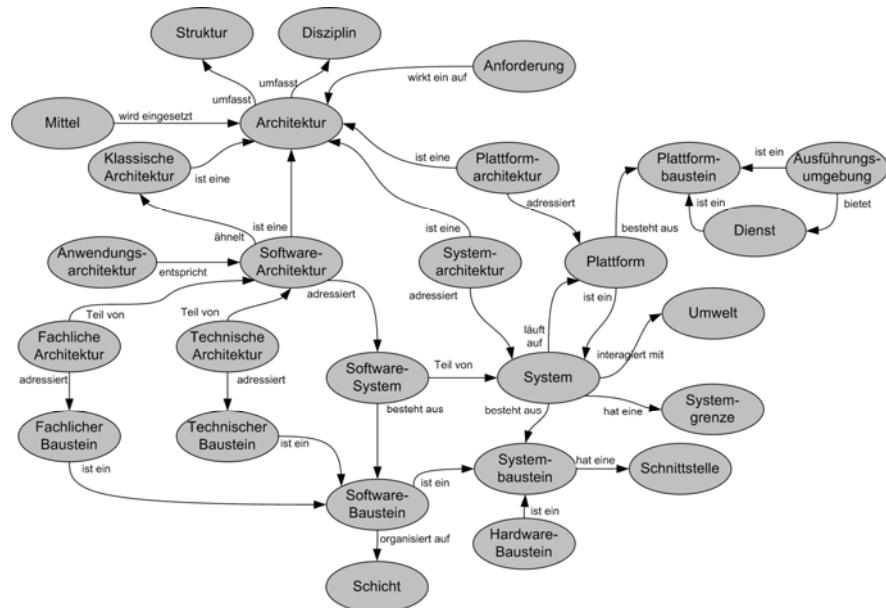


Abb. 3-2: Grundlegende Konzepte der WAS-Dimension.

3.1 Klassische Architektur als Ausgangspunkt

Allgemeine Architektur-Betrachtung

Dieser Abschnitt betrachtet Architektur aus einem allgemeinen Blickwinkel. Er zeigt auf, was ganz allgemein unter Architektur zu verstehen ist. Auf der Grundlage dieses Verständnisses wird im weiteren Verlauf dieses Kapitels Software-Architektur vorgestellt. Als Ausgangspunkt für diese Betrachtung dient die klassische Architektur von Gebäuden und Bauwerken. Eine mögliche Definition der klassischen Architektur bietet das American Heritage Dictionary:

Klassische Architektur-Definition

The art and science of designing and erecting buildings
A style and method of design and construction
Orderly arrangement of parts

Wenn man diese Definition zugrunde legt, ist Architektur sowohl eine Kunst (englisch: *art*) als auch eine Wissenschaft (englisch: *science*), die sich sowohl mit dem Entwerfen (englisch: *designing*) als auch mit dem

Bauen (englisch: *erecting*) von Bauwerken beschäftigt. Sie konzentriert sich also nicht nur auf die Planung des Bauwerks, sondern erstreckt sich hinein bis in dessen Realisierung. Ferner ist ein Schlüsselergebnis der Architektur-Tätigkeit das Arrangieren beziehungsweise das systematisch Anordnen von Teilen des Bauwerks (englisch: *orderly arrangement of parts*). Somit trifft Architektur wichtige Aussagen über die Struktur des Bauwerks. Die Definition besagt weiter, dass Architekturstile (englisch: *styles*) und -Methoden (englisch: *methods*) Bestandteile von Architektur sind. Diese repräsentieren architektonische Erfahrung, die der Architekt im Rahmen seiner Tätigkeit einsetzt. Architektur ist hiermit nicht nur die Struktur eines Bauwerks, sondern auch die Art und Weise, an etwas heranzugehen.

Der Begriff Architektur ist nicht eindeutig belegt. Stattdessen versteht man unter Architektur zum einen die Struktur eines Bauwerks oder eines IT-beziehungsweise Software-Systems und zum anderen die von Menschen ausgeübten Tätigkeiten zum Entwurf der Struktur. Um die eigentliche Tätigkeit beziehungsweise das architektonische Handeln besser von den strukturellen Aspekten von Architektur zu unterscheiden, wird das Handeln im weiteren Verlauf als Architektur-Disziplin verstanden.

Architektur umfasst Struktur und Tätigkeit

Architekturen entstehen ganz generell aufgrund von Anforderungen (z. B. dem Wunsch nach einfachen Behausungen) und unter Verwendung von vorhandenen Mitteln (z. B. Baumaterialien und Werkzeugen). Historisch gesehen basierte der eigentliche Entwurf in der klassischen Architektur zunächst auf dem Prinzip von Versuch und Irrtum (englisch: *trial and error*) und erfolgte in aller Regel ad hoc. Dadurch besaß auch jedes Bauwerk seine individuellen Strukturen. Eine geordnete Anordnung von Teilen durch eine geplante Architektur war meist nicht gegeben. Erst als die gewonnenen Architektur-Erfahrungen mündlich oder schriftlich weitergegeben wurden, entwickelten sich Architekturstile. Architektur basiert aufgrund dessen immer auch auf Heuristiken, respektive architektonischen Mitteln und Vorgehensweisen, die sich in der Vergangenheit bewährt haben. Architekturstile sind somit Mittel, um erprobte Lösungen für architektonische Problemstellungen zu dokumentieren. Den Einfluss von Anforderungen und Mitteln auf Architektur illustriert Abbildung 3.1-1.

Entwicklung von Architektur

Ein wichtiger Vordenker der klassischen Architektur war Marcus Vitruvius Pollio, ein römischer Architekt aus dem ersten Jahrhundert vor Christus. Er ist Autor des Werks „*De architectura libri decem*“, welches heute unter dem Titel „*Zehn Bücher über Architektur*“ bekannt ist [Morgan 1960]. Vitruvius vertrat die These, dass gute Architektur folgende Anforderungen erfüllen muss:

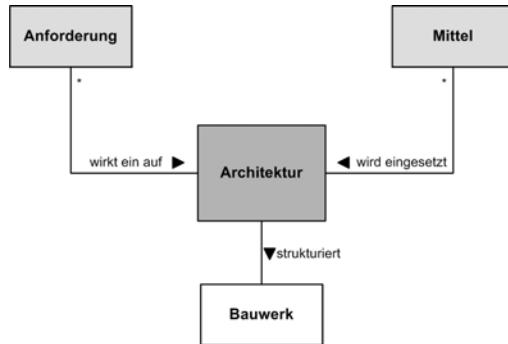


Abb. 3.1-1: Anforderungen und Mittel als Einflussfaktoren auf Architektur.

- > Beständigkeit (*firmitas*)
- > Zweckmäßigkeit (*utilitas*)
- > Eleganz (*venustas*)

Anforderungen an Architektur

Diese Anforderungen sind noch immer gültig. Die *Beständigkeit* einer Architektur besagt, ob zukünftige Bedürfnisse durch sie befriedigt werden können und ob sie Weiterentwicklungen tragen kann. So kann ein Haus beispielsweise nur um ein Stockwerk erweitert werden, wenn diese Erweiterung grundsätzlich in der Architektur vorgesehen wurde. Die *Zweckmäßigkeit* ist ein Gütesiegel dafür, ob die Architektur die konkreten Bedürfnisse erfüllt. Die Anordnung von Türen eines Hauses sollte z. B. durch die Architektur so vorgesehen sein, dass die Erreichbarkeit der Räume gewährleistet ist. Die *Eleganz* drückt letztlich aus, wie die Architektur strukturiert wurde. Die Art und Weise, wie die einzelnen Bestandteile der Architektur angeordnet wurden, spiegelt sich in der *Eleganz* wider. Verschiedene Architektur-Alternativen können die gleichen Anforderungen unterschiedlich elegant umsetzen.

Architektur als Kompromiss

Eine Architektur entsteht somit aufgrund von verschiedenen Anforderungen und wird durch diese beeinflusst (siehe Abbildung 3.1-2).

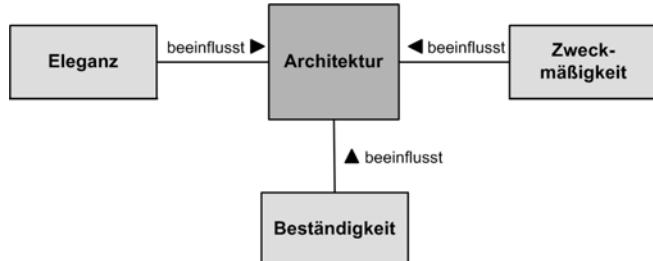


Abb. 3.1-2: Architektur im Spannungsfeld klassischer Anforderungen.

Architekturen erfüllen dabei Anforderungen unterschiedlich gut. Sie sind somit immer ein Kompromiss und das Ergebnis von Abwägungen und Entscheidungen des Architekten (siehe Kapitel 5).

Ein Architekt kommuniziert mit verschiedenen Interessenvertretern (englisch: *stakeholder*). So bespricht er mit dem Kunden beispielsweise die Außenansicht eines Gebäudes oder die geplante Raumaufteilung. Mit anderen am Bau des Gebäudes beteiligten Personen klärt er technische Themen, beispielsweise den Elektroplan des Gebäudes. Ein Architekt besitzt bei einem Bauvorhaben somit eine zentrale Rolle.

Ein Architekt diskutiert mit Interessenvertretern (z. B. Kunde oder Vorarbeiter) auf unterschiedlichen Ebenen und aus unterschiedlichen Sichten auf ein und dasselbe Gebäude. Ebenso wendet er bei der Ausarbeitung der Architektur diese verschiedenen Betrachtungsweisen an.

Zwischen der klassischen Architektur und der Software-Architektur existieren Gemeinsamkeiten. Diese Gemeinsamkeiten sollen im Folgenden durch die kurze Zusammenfassung der bislang gewonnenen Erkenntnisse veranschaulicht werden:

- Eine Architektur legt die Anordnung von Teilen eines Bauwerks beziehungsweise eines IT-Systems fest. Im Sinne der klassischen Architektur sind dies die tragenden Teile einer Architektur [Perry und Wolf 1992]. Die Architektur definiert folglich die Fundamente, aber nicht die Details für das zu entwickelnde System [Buschmann et al. 1996].
- Ein Architekt verwendet unterschiedliche Sichten zur Darstellung einer Architektur und bewegt sich im Rahmen seiner Tätigkeit auf unterschiedlichen Ebenen. Die für einen Software-Architekten relevanten Ebenen und Sichten werden in Kapitel 4 dargestellt.
- Jede Architektur-Disziplin muss das Gleichgewicht zwischen den an die Architektur gestellten Anforderungen sicherstellen. Wie ein klassischer Architekt muss infolgedessen auch ein Software-Architekt die für Software-Architekturen relevanten Anforderungen kennen und beim Architektur-Entwurf berücksichtigen. Aus diesem Grund werden Architektur-Anforderungen in Kapitel 5 eingehend besprochen.
- Die Art und Weise der Anordnung basiert auf Erfahrung und verkörpert einen architektonischen Stil. Beim Entwurf der Architektur werden somit Architektur-Mittel eingesetzt. In der Software-Architektur werden hierzu verschiedene Mittel verwendet. Diese werden in Kapitel 6 vorgestellt.
- Ein Architekt kommuniziert und agiert mit unterschiedlichen Personen. Er hat eine zentrale Rolle bei der Realisierung eines Bauvor-

Architekt kommuniziert mit Interessenvertretern

Architekt verwendet unterschiedliche Perspektiven

Gemeinsamkeiten zwischen klassischer Architektur und Software-Architektur

habens. Dies gilt ebenso für den Software-Architekten. Deshalb widmet sich Kapitel 7 den sozialen respektive zwischenmenschlichen Aspekten von Software-Architektur.

- > Das konkrete Handeln zur Anordnung der Teile erfolgt basierend auf einer Methode, und die architektonische Tätigkeit erstreckt sich vom Entwurf bis hin zur Umsetzung. Das architektonische Vorgehen bei der Gestaltung einer Software-Architektur wird in Kapitel 8 behandelt.

Im nachfolgenden Abschnitt werden die bislang gewonnenen Erkenntnisse konkretisiert und auf die Software-Architektur angewendet.

3.2 Von der klassischen Architektur zur Software-Architektur

Software-Architektur und die klassische Architektur-Definition

Wie im vorherigen Abschnitt verdeutlicht wurde, kann die allgemeine Definition von Architektur auch auf Software-Architektur übertragen werden. Software-Architektur beschäftigt sich mit dem Entwurf und der Umsetzung von IT-Systemen, wie z. B. Web-2.0-Anwendungen oder eingebetteten Systemen. Unter dem Blickwinkel der architektonischen Tätigkeit umfasst Software-Architektur die Art und Weise der Architektur-Gestaltung. Hinsichtlich des strukturellen Aspekts von Architektur beschreibt Software-Architektur die Strukturen von IT-Systemen.

IT-System und System als Synonyme

Im weiteren Verlauf werden die Begriffe **IT-System** und **System** synonym verwendet, soweit keine explizite Unterscheidung notwendig ist.

Systeme sind facettenreich

Systeme sind in der heutigen Zeit sehr facettenreich und bedingen Entscheidungen in architektonischen Bereichen, die über den reinen Software-Aspekt hinausgehen. So kann, je nach Art von System, bereits beim Entwurf der Software-Architektur die Kenntnis der eingesetzten Hardware notwendig sein. Ein System als Ganzes besteht folglich aus mehr als nur Software-Bausteinen. Aus diesem Grund wird an dieser Stelle eine Systemdefinition eingeführt, die den allgemeinen Aspekt von Systemen berücksichtigt:

Definition: System

Ein System ist eine Einheit, die aus miteinander interagierenden Software- und Hardware-Bausteinen besteht sowie zur Erfüllung eines fachlichen Ziels existiert. Es kommuniziert zur Erreichung seines Ziels mit seiner Umwelt und muss den durch die Umwelt vorgegebenen Rahmenbedingungen Rechnung tragen.

Die grafische Darstellung der Bausteine eines Systems können Abbildung 3.2-1 entnommen werden.

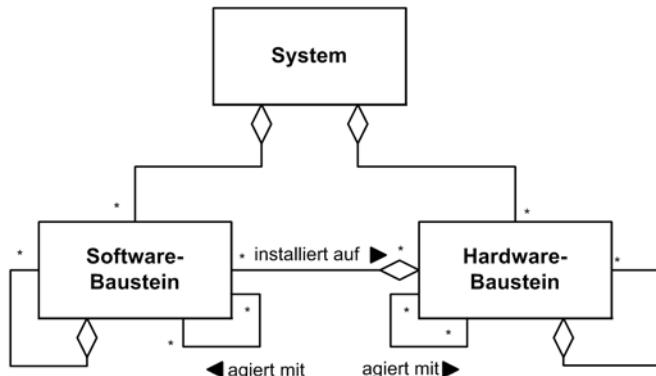


Abb.3.2-1: Bausteine eines Systems.

Das fachliche Ziel ist dabei durch die an das System gestellten funktionalen Anforderungen beschrieben (siehe Kapitel 5). Die Systemdefinition betont bewusst Software- als auch Hardware-Bausteine, um festzuhalten, dass ein System mehr ist als nur Software. Die Umwelt eines Systems ist die Organisation, in die es eingebettet ist. Es kommuniziert mit Teilen der Organisation. Dies können zum einen Menschen als Benutzer des Systems und zum anderen auch andere Systeme sein, mit denen das System verbunden ist. Je nach Art des Systems existieren menschliche Benutzer oder nicht. Ein System zur Steuerung eines Motors hat z. B. keinen direkten menschlichen Benutzer. Es wird jedoch sehr wohl mit anderen Systemen seiner Umwelt kommunizieren. Die Organisation setzt den Rahmen, in dem sich das System bewegen kann. Festgeschriebene Standards und Entwicklungsrichtlinien können z. B. Rahmenbedingungen sein, die das System im übertragenen Sinn berücksichtigen muss. Letztlich sind dies natürlich Punkte, die während des Entwurfs und der Umsetzung des Systems beachtet werden müssen.

In Kapitel 1 wurde illustriert, dass eine Vielzahl von Definitionen für Software-Architektur existiert. Daran wird deutlich, wie groß der Interpretationsspielraum in diesem Bereich ist. Aufgrund dieses breiten Spektrums ist es in unseren Augen auch nicht möglich, die eine richtige Definition zu geben. Vielmehr haben wir uns für dieses Buch entschlossen, eine Definition zu finden, die auf unserem gemeinsamen Verständnis beruht und welche die zuvor eingeführte, klassische Architektur-Definition berücksichtigt. Daher muss eine entsprechende Definition sowohl die Struktur als auch die Tätigkeit umfassen.

Versuch einer Definition von Software-Architektur

Software-Architektur beschreibt Software-Bausteine

Software-Architektur beschreibt die Software-Bausteine eines Systems. Um dies genauer zu fassen, kann auf eine Definition zurückgegriffen werden, die den strukturellen Charakter von Software-Architektur gut wiedergibt und in der Literatur sowie der Praxis weit verbreitet ist. Es handelt sich dabei um die Definition von Software-Architektur nach Bass et al. [Bass et al. 2003]:

Definition: Software-Architektur eines Systems nach Bass et al.

Die Software-Architektur eines Systems beschreibt dessen Software-Struktur respektive dessen -Strukturen, dessen Software-Bausteine sowie deren sichtbare Eigenschaften und Beziehungen zueinander.

Diese Definition ist sehr allgemein gehalten. Sie beinhaltet jedoch die wichtigsten Aspekte einer Software-Architektur:

- > die Software-Struktur beziehungsweise die Software-Strukturen eines Systems.
- > die Software-Bausteine eines Systems.
- > die Eigenschaften der Software-Bausteine eines Systems.
- > die Beziehungen zwischen den Software-Bausteinen eines Systems.

Software-Bausteine eines Systems und ihre sichtbaren Eigenschaften

Die Definition besagt, dass eine Software-Architektur Software-Bausteine eines Systems definiert. Hierbei stellt sich die Frage, welche Software-Bausteine durch eine Architektur definiert werden. Im Sinne von Perry und Wolf sind dies die tragenden Bausteine eines Systems [Perry und Wolf 1992]. Mit anderen Worten sind dies die Software-Bausteine, die von wesentlicher Bedeutung für das zu realisierende System sind. Dies können Schlüsselklassen, Schnittstellen, Komponenten, Frameworks, Subsysteme und Module sein (Kapitel 6 beschäftigt sich näher mit diesen Begriffen). Eine genaue Festlegung erfolgt hierbei bewusst nicht, da Systeme sehr vielfältig sind und damit auch die konkreten Ausprägungen ihrer Software-Bausteine sehr unterschiedlich sein können. Dies veranschaulichen z. B. die in diesem Buch enthaltenen Fallstudien. Eine klare Abgrenzung zwischen Software-Architektur und -Design ist dabei nur schwer möglich. Die richtige Grenze zu ziehen, hängt dabei oft von der entsprechenden Erfahrung des Architekten und dem eingenommenen Blickwinkel ab (siehe Kapitel 4). Die nach außen sichtbaren Eigenschaften von Software-Bausteinen sind die Eigenschaften, die von anderen Software-Bausteinen wahrgenommen werden können. Hierzu gehören beispielsweise die angebotene Funktionalität, die Schnittstellen und das Leistungsverhalten der Software-Bausteine. Die inneren Strukturen und Charakteristika von Software-Bausteinen werden bei einer architektonischen Betrachtung im Allge-

meinen nicht berücksichtigt. Es erfolgt im Wesentlichen eine holistische Betrachtung eines Systems (siehe Abschnitt 3.3).

Neben der reinen Benennung der Software-Bausteine beschreibt eine Software-Architektur auch die Strukturen zwischen den Software-Bausteinen sowie die damit implizierten Beziehungen. Dabei ist es wichtig, festzuhalten, dass es nicht die eine Struktur gibt, sondern dass je nach Perspektive unterschiedliche Strukturen eines Systems wichtig sind und durch eine Architektur festgelegt werden müssen. So existiert beispielsweise stets eine statische und eine dynamische Struktur eines Systems. Aus diesem Grund nimmt der Architekt beim Entwurf der Architektur auch unterschiedliche Perspektiven ein (siehe Kapitel 4).

Eine weitere wichtige Definition von Architektur stammt von der IEEE:

Architecture is the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution.

Diese Definition wurde von der IEEE in ihrem Standard 1471 [IEEE 2007] eingeführt, welcher sich mit der Beschreibung softwareintensiver Systeme beschäftigt. Softwareintensive Systeme sind die Systeme, deren Charakter größtenteils von Software geprägt wird, jedoch nicht nur aus Software bestehen.

Ein wichtiger Aspekt dieser Definition ist die explizite Berücksichtigung der Umwelt (englisch: *environment*) des Systems. Dies ist ein wesentlicher Punkt, der in Abschnitt 3.3 sowie in Kapitel 8 vertieft wird.

Aus der Definition nach Bass und der Definition nach IEEE lässt sich folgende Definition von Software-Architektur ableiten, die den strukturellen Charakter von Software-Architektur würdigt:

Die Software-Architektur eines Systems beschreibt dessen Software-Struktur respektive dessen -Strukturen, dessen Software-Bausteine sowie deren sichtbaren Eigenschaften und Beziehungen zueinander und zu ihrer Umwelt.

Im Hinblick auf die in Abschnitt 3.1 vorgestellte Definition der klassischen Architektur adressiert die bisherige Definition von Software-Architektur nur das systematische Anordnen der Teile (englisch: *orderly arrangement of parts*). Gemäß der klassischen Definition ist Architektur

Software-Struktur(en) eines Systems

Definition: Architektur softwareintensiver Systeme nach IEEE

Software-Architektur und die Systemumwelt

Definition: Software-Architektur_{Struktur}

Software-Architektur als Disziplin

**Definition:
Software-
Architektur**_{Disziplin}

aber weit mehr als nur eine architektonische Beschreibung eines Systems. Sie beinhaltet auch die eigentliche Architektur-Tätigkeit (in der klassischen Architektur-Definition in Englisch: *art and science*), die letztlich zu der Architektur eines Systems führt. Diesem Gesichtspunkt widmet sich Software-Architektur als Disziplin:

Software-Architektur als Disziplin befasst sich mit den architektonischen Tätigkeiten und den hiermit verbundenen Entscheidungen zum Entwurf und zur Umsetzung einer Software-Architektur.

Mit anderen Worten befasst sich Software-Architektur im Sinne einer Architektur-Disziplin primär mit architektonischen Tätigkeiten im Rahmen der Analyse, dem Entwurf und der Implementierung von einzelnen Systemen. Wichtige Tätigkeiten sind hierbei die Identifikation und der Entwurf von Software-Bausteinen, ihrer Schnittstellen und ihrer Interaktionen. Dabei erfolgt in der Regel eine rekursive Dekomposition eines Systems. Dies geschieht ausgehend von den aus der Analyse gewonnenen Anforderungen an das System. Ebenso ist die Auswahl von Prinzipien, welche dem Entwurf und der Evolution eines Systems zugrunde liegen, ein wichtiger Gesichtspunkt dieser Disziplin. Dies wird gerade durch die Definition der IEEE betont. Darüber hinaus sind beispielsweise die Berücksichtigung der zugrunde liegenden Plattform und die Wahl der Verteilung der Software-Bausteine Aufgaben eines Software-Architekten. Die Wahl von entsprechenden Entwicklungsmethoden und Werkzeugen gehören auch zu seinem Aufgabenfeld (siehe Kapitel 8). Maier und Rechtin verwenden in diesem Zusammenhang auch den Begriff *Architecting*, um dem handelnden Aspekt von Architektur entsprechend Ausdruck zu verleihen [Maier und Rechtin 2000]. Im Deutschen könnte man hierbei vom *Architekturieren* von Software-Architekturen sprechen.

**Ganzheitliche Betrach-
tung von Software-
Architektur**

Für dieses Buch haben wir uns bewusst entschlossen, die Gesichtspunkte Struktur und Tätigkeit zunächst in Definitionen zu trennen. Die Software-Architektur als Struktur kann als Ergebnis der Software-Architektur als Disziplin angesehen werden. Software-Architektur in ihrer Gesamtheit umfasst letztlich die Software-Architektur als Struktur und die Software-Architektur als Tätigkeit beziehungsweise Disziplin:

Software-Architektur_{Gesamt} = Software-Architektur_{Struktur} +
Software-Architektur_{Disziplin}

Wenn im weiteren Verlauf des Buches von Software-Architektur gesprochen wird, kann darunter zum einen das Ergebnis respektive die Software-Struktur(en) eines IT-Systems und zum anderen die Tätigkeit des Software-Architekten verstanden werden.

Die gesamte Architektur eines Systems berücksichtigt neben Software-Bausteinen auch Hardware-Bausteine. Aus Gründen der Vollständigkeit wird deshalb eine breiter gefasste Definition von Systemarchitektur eingeführt:

Systemarchitektur_{Struktur}: Die Systemarchitektur eines Systems beschreibt dessen Struktur respektive dessen Strukturen, dessen Bausteine (Software- und Hardware-Bausteine) sowie deren sichtbare Eigenschaften und Beziehungen sowohl zueinander als auch zu ihrer Umwelt.

Systemarchitektur_{Disziplin}: Systemarchitektur als Disziplin befasst sich mit den architektonischen Tätigkeiten und den hiermit verbundenen Entscheidungen zum Entwurf und zur Umsetzung einer Systemarchitektur.

$$\text{Systemarchitektur}_{\text{Gesamt}} = \text{Systemarchitektur}_{\text{Struktur}} + \text{Systemarchitektur}_{\text{Disziplin}}$$

Neben den Begriffen Software-Architektur und Systemarchitektur kursieren in der Praxis und in der Literatur noch viele weitere Architekturbegriffe. Hierzu gehören Begriffe wie technische Architektur, fachliche Architektur und Plattformarchitektur. Die Begriffe werden dabei meist nicht klar definiert und mitunter willkürlich verwendet. Daher wird auf diese Begriffe im Anschluss näher eingegangen und versucht, Definitionen zu finden, die der Praxis gerecht werden. Abbildung 3.2-2 veranschaulicht die verschiedenen Architektur-Begriffe und setzt sie mit den Bausteinen eines Systems in Beziehung.

Die Software-Architektur ist ein Teil einer Systemarchitektur. Sie strukturiert die Software-Bausteine eines Systems. Wenn man sich die Software-Architektur näher betrachtet, stellt man fest, dass diese sowohl fachliche als auch technische Aspekte berücksichtigt. Oftmals wird aus diesem Grund von einer fachlichen und einer technischen Architektur gesprochen.

**Definition:
Systemarchitektur**

Es existieren viele weitere Architektur-Begriffe

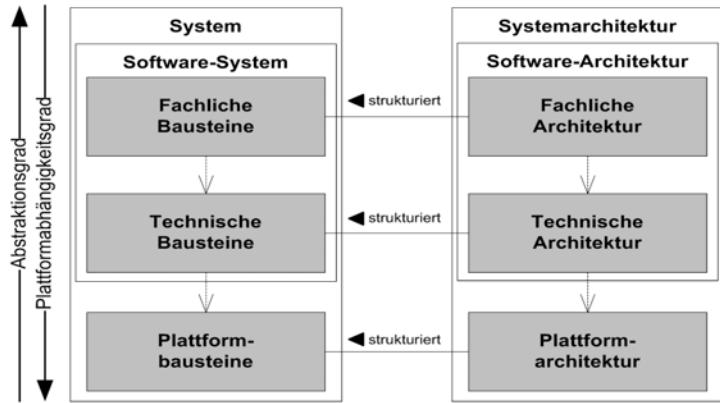


Abb. 3.2-2: Architektur-Begriffe und ihre Beziehungen.

Fachliche Architektur

Die fachliche Architektur entspringt der Domäne respektive dem Problembereich für die das System entwickelt wird. Sie unterteilt das System in fachliche Bausteine. Getrieben wird die fachliche Architektur somit durch den Charakter der Domäne und die an das System gestellten funktionalen Anforderungen. Im Rahmen einer Auftragsabwicklungslösung könnte man beispielsweise die fachlichen Bausteine Auftragserfassung, Auftragsverwaltung und Kundenverwaltung identifizieren (siehe Abschnitt 8.3.2). Der Abstraktionsgrad der fachlichen Architektur ist hoch und deren Plattformabhängigkeit ist niedrig. Die fachliche Architektur setzt auf der technischen Architektur auf.

Technische Architektur

Die technische Architektur ist im ursprünglichen Sinn domänenneutral und widmet sich primär der Realisierung von nicht-funktionalen Anforderungen respektive Qualitäten. Sie definiert technische Bausteine für nicht-funktionale Aspekte, wie z. B. Logging, Auditing, Sicherheit, Referenzdaten, Persistenz und Transaktionsmanagement. Die technischen Bausteine nutzen Dienste der Plattform und abstrahieren sie so, dass sie von fachlichen Bausteinen plattformneutral genutzt werden können. Der Abstraktionsgrad einer technischen Architektur ist niedriger als der einer fachlichen Architektur. Umgekehrt ist die Plattformabhängigkeit einer technischen Architektur höher. Technische Architekturen können in der Regel für Software-Systeme unterschiedlicher Domänen genutzt werden, da sie domänenneutral sind. Ferner legt die technische Architektur fest, wie fachliche Bausteine auf die technische Architektur abgebildet werden. Ein grobgranularer fachlicher Baustein kann sich somit durch feingranularere fachliche Bausteine manifestieren. Die so entstandenen fachlichen Bausteine besitzen ein niedrigeres Abstraktionsniveau. Beispielsweise kann die technische Architektur den Einsatz

des MVC-Musters (siehe Abschnitt 6.4) vorschreiben. Dadurch wird der fachliche Baustein Auftragsverwaltung in einen Modell-Baustein, einen View-Baustein sowie einen Controller-Baustein aufgeteilt.

Um den Begriff der Plattformarchitektur zu verstehen, muss zunächst der Begriff der Plattform näher beleuchtet werden. Eine ausführliche Behandlung erfolgt in Abschnitt 3.3. Für den Moment soll es genügen, dass eine Plattform selbst wieder ein System ist, welches aus Software- und ggf. Hardware-Bausteinen bestehen kann. Es dient zur Ausführung von Software-Bausteinen eines Systems und bietet diesen Dienste an. Die Plattformarchitektur legt somit die Plattformbausteine und deren Strukturierung fest. Die Plattformbausteine werden von den technischen Bausteinen genutzt. Beispiele für reine Software-Plattformen finden sich in Abschnitt 6.7.5. So ist z. B. JEE eine Komponentenplattform, die wiederum auf unterschiedlichen Betriebssystem- und Hardware-Plattformen existieren kann.

Für ein erstes Verständnis der Begriffe fachliche, technische und Plattformarchitektur sind die vorherigen Ausführungen ausreichend. Allerdings ist es in der Praxis nicht immer leicht, eine Architektur der einen oder anderen Kategorie zuzuordnen. So werden Plattformen immer mächtiger und bieten nicht mehr nur reine infrastrukturahe Dienste wie Transaktionsmanagement oder Persistenz an. Vielmehr entstehen auch Plattformen, die fachliche Basisfunktionalität anbieten. Eine Portal-Plattform wird z. B. auch Personalisierung, Syndizierung oder Kampagnenmanagement anbieten. Somit deckt die Plattform bereits auch fachliche Anforderungen ab. Sie besitzt damit sowohl einen fachlichen als auch einen technischen Charakter. Man kann sich auch vorstellen, dass eine fachliche Plattform auf einer technischen Plattform aufsetzt und das eigentliche Software-System wiederum auf der fachlichen Plattform aufbaut. Je zweckmäßiger die verwendete Plattform ist, d. h. je mehr Anforderungen durch die Plattform bereits erfüllt werden, desto schlanker können die technische und die fachliche Architektur ausfallen. Modellgetriebene Software-Entwicklung (siehe Abschnitt 6.2.6) und domänenpezifische Modellierung tragen mit zu diesem Trend bei.

Ein weiterer Begriff, der sowohl in der Praxis als auch in der Theorie vielfältig verwendet wird, ist Anwendungsarchitektur (Synonym: Applikationsarchitektur). Für manche ist die Anwendungsarchitektur die Software-Architektur und für andere die fachliche Architektur. In diesem Buch setzen wir Anwendungsarchitektur mit Software-Architektur gleich.

Plattformarchitektur

Der Übergang zwischen fachlicher, technischer und Plattformarchitektur ist fließend

Und was war nun schon wieder die Anwendungsarchitektur?

Weitere Architektur-Disziplinen in der IT

Neben der Software-Architektur existieren in der IT weitere Architektur-Disziplinen. Dies liegt unter anderem darin begründet, dass IT-Systeme immer komplexer werden und daher eine Spezialisierung in einem Gebiet notwendig machen. Folgende beispielhafte Ausführungen sollen dies verdeutlichen. Der Fokus liegt hierbei auf einem unternehmensbezogenen IT-System. Die Problematik lässt sich jedoch auch auf andere IT-Systeme, wie eingebettete Systeme, übertragen. In Klammern wird hierbei jeweils die entsprechende Architektur-Disziplin benannt.

- > Verschiedene Software-Bausteine eines Systems können auf unterschiedliche Hardware verteilt sein, die über ein Netzwerk miteinander verbunden sind. Somit sind beim Architektur-Entwurf auch Netzwerkaspekte zu berücksichtigen (Architektur-Disziplin: *Netzwerkarchitektur*).
- > Darüber hinaus müssen Systeme zum Austausch von Daten oder zur Unterstützung systemübergreifender Geschäftsprozesse miteinander kommunizieren. Dies macht deren Integration innerhalb von Unternehmen und über Unternehmensgrenzen hinweg erforderlich. Die Architektur eines Systems muss aufgrund dessen auch integrative Aspekte würdigen (Architektur-Disziplin: *Integrationsarchitektur*).
- > Die Qualität der ausgetauschten Daten hat hierbei eine wesentliche Bedeutung für den Erfolg eines Unternehmens. Systeme haben aus diesem Grund oftmals die Aufgabe, in einem Unternehmen verstreut existierende Daten zu sammeln und zur Verfügung zu stellen. Die Repräsentation dieser Daten muss auch beim Architektur-Entwurf berücksichtigt werden (Architektur-Disziplin: *Datenarchitektur*).
- > Die ausgetauschten und von einem System verarbeiteten Daten können hoch sensible Informationen enthalten und müssen vor dem Zugriff durch unberechtigte Dritte geschützt werden. Die Gewährleistung dieser Sicherheit ist deshalb durch ein System zu garantieren und in dessen Architektur entsprechend vorzusehen (Architektur-Disziplin: *Sicherheitsarchitektur*).
- > Nicht zuletzt ist es auch erforderlich, dass Systeme in einer adäquaten Art und Weise betrieben werden können, um die benötigte Verfügbarkeit und Zuverlässigkeit zu gewährleisten. Deshalb sind Gesichtspunkte hinsichtlich des Betriebs des Systems in die Architektur einzuplanen (Architektur-Disziplin: *System-Management-Architektur*).
- > Darüber hinaus müssen Systeme in der Regel gemäß vorgegebener Standards und Richtlinien entwickelt werden, die in einem Unternehmen im Rahmen einer umfassenden IT-Strategie festgelegt wurden (Architektur-Disziplin: *Enterprise-Architektur*).

Die genannten Architektur-Disziplinen werden im Folgenden kurz charakterisiert, um diese voneinander unterscheiden zu können.

Netzwerkarchitektur beleuchtet die Infrastruktur von Systemen beziehungsweise gesamter Unternehmungen. Die Planung und der Entwurf der Funktionen, Dienste, Bausteine und Protokolle eines Netzwerks sind die Hauptaufgaben dieser Disziplin.

Netzwerkarchitektur

Integrationsarchitektur beschäftigt sich mit der Planung und Realisierung von integrativen Lösungen mit dem Ziel, mehrere Anwendungen oder Systeme eines oder mehrerer Unternehmen miteinander zu verbinden. Dabei müssen meist heterogene Plattformen, Technologien, Organisationen und Daten in Einklang gebracht werden. Eine wichtige Aufgabe ist hierbei oftmals auch die Berücksichtigung von Legacy-Systemen und deren Anschluss an E-Business-Lösungen.

Integrationsarchitektur

Datenarchitektur umfasst die datenorientierten Aspekte eines Systems. Der Entwurf logischer und physischer Datenmodelle, die Auswahl von Persistenzmechanismen (z. B. Datenbank oder Dateisystem), die Konfiguration einer Datenbank und der Entwurf eines Data Warehouse sind Tätigkeiten dieser Disziplin.

Datenarchitektur

Eine Sicherheitsarchitektur fokussiert auf die Gewährleistung von Vertraulichkeit, Integrität, Verfügbarkeit von Systemen beziehungsweise Systemlandschaften, Identitäts- und Berechtigungsüberprüfung sowie Nachweisbarkeit und Unleugbarkeit sicherheitsrelevanter Vorgänge. Der Entwurf und die Umsetzung von PKI-Infrastrukturen, die Implementierung einer unternehmensweiten Single-Sign-on-Lösung und die Etablierung eines Identitätsmanagements sind z. B. Aufgabenfelder dieser Architektur-Disziplin. Ferner ist jedoch auch die Authentifizierung und Autorisierung von Benutzern innerhalb einer Anwendung ein Aspekt dieser Disziplin. Des Weiteren ist die Durchführung von Tests zur Identifikation von Sicherheitslücken dieser Disziplin zuzuordnen.

Sicherheitsarchitektur

System-Management-Architektur beinhaltet hauptsächlich den operationalen Aspekt von Systemen. Der Entwurf von Betriebsstrategien zentraler und dezentraler Systemlandschaften und die Definition von Service Level Agreements sind Aufgaben, denen ein Architekt in dieser Disziplin gegenübersteht. Ferner beschreibt eine System-Management-Architektur z. B., wie ein System an eine System-Management-Umgebung angeschlossen wird.

System-Management-Architektur

Enterprise-Architektur

Enterprise-Architektur (deutsch: Unternehmensarchitektur) ist eine Disziplin, die unter Berücksichtigung von Geschäftsstrategien, -prozessen und -daten eine unternehmensweite IT-Architektur entwirft. Die Enterprise-Architektur umfasst dabei Prozesse, Anwendung, Daten und Technologien zur Realisierung der Geschäftsstrategie. Des Weiteren hat die Enterprise-Architektur die Aufgabe, den Übergangsprozess von der Ist-Architektur zur geplanten Ziel-Architektur aufzuzeigen und zu überwachen. Enterprise-Architektur adressiert somit die Zielarchitektur, den Übergangsprozess (englisch: *transition*) und die Steuerung (englisch: *governance*) desselben. Im Hinblick auf das zuvor eingeführte Beispiel legt Enterprise-Architektur die Standards und Richtlinien fest, an denen sich Architekten beim Entwurf von Systemen orientieren müssen. Im Sinne der klassischen Architektur würde die Enterprise-Architektur den Bebauungsplan festlegen.

Im Fokus der Betrachtung liegt bei Enterprise-Architektur nicht das einzelne IT-System, sondern die Gesamtheit aller IT-Systeme des betrachteten Unternehmensausschnitts (Architektur von Architekturen). Die Fragen, die eine Enterprise-Architektur damit beantwortet, beziehen sich entsprechend auf diese Systemgesamtheit. Auch existieren spezifische Architektur-Modelle für Enterprise-Architektur-Betrachtungen. Diese Architektur-Modelle setzen sich mit den Aspekten Form, Prozess, Werkzeug sowie Inhalt beziehungsweise Inventar auseinander. Als Beispiele können das Zachman-Framework (siehe Abschnitt 4.2.1) sowie TOGAF [Opengroup 2008a] genannt werden.

Herausforderungen von Enterprise-Architektur

Einige Herausforderungen, denen eine Enterprise-Architektur begegnen muss, sind:

- > Unternehmensweite Motivation findet keine direkte Abbildung in der Aufbauorganisation des Unternehmens.
- > Identifikation der Vorgaben und Richtlinien, die als EA-relevant gesehen werden (Organisationsebene).
- > Durchsetzen und Operationalisieren dieser Vorgaben und Richtlinien im Unternehmen.
- > Planung und Management des Systemportfolios eines Unternehmens.

Kein Anspruch auf Vollständigkeit

Die Aufzählung erhebt keinen Anspruch auf Vollständigkeit. Die in der *WAS-Dimension* genannten Disziplinen haben jedoch eine wesentliche Bedeutung in der Informationstechnologie – insbesondere im Bereich unternehmensbezogener IT-Systeme.

In ihrer Gesamtheit können die genannten Architektur-Disziplinen zu einem System beitragen. Mit zunehmender Komplexität und Größe des Systems werden fundierte Kenntnisse in anderen Architektur-Disziplinen als der Software-Architektur immer wichtiger. Oftmals sind die Aufgaben der Disziplinen auf mehrere Schultern verteilt, sodass die Architektur eines Systems als Teamleistung anzusehen ist. In der Regel kommt dem Software-Architekten hierbei die Hauptverantwortung zu. Der Software-Architekt zieht zur Klärung von Fragen und zur Abstimmung Architekten anderer Disziplinen hinzu. Das Zusammenspiel der Architektur-Disziplinen kann Abbildung 3.2-3 entnommen werden.

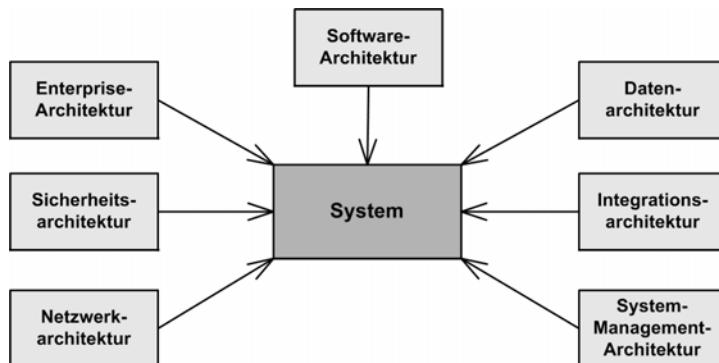


Abb. 3.2-3: Architektur-Disziplinen im Zusammenspiel.

Eine Differenzierung zwischen den Architektur-Disziplinen ist nicht immer einfach, und als Software-Architekt wird man sich mit allen Themen befassen müssen, da ein zu realisierendes System Aspekte der unterschiedlichen Disziplinen aufweisen kann.

3.3 Architektur und der Systemgedanke

Der Begriff des Systems wurde in den vorangegangenen Kapiteln bereits mehrfach benutzt, da sich Architektur generell mit Systemen beschäftigt, seien es nun Bauwerke, Städte, Landschaften oder IT-Systeme. Aus diesem Grund ist es wichtig, die allgemeinen Eigenschaften von Systemen zu verstehen. Ein grundlegendes Verständnis von Systemen und des Denkens in Systemen ist folglich eine Voraussetzung für das erfolgreiche Handeln als Software-Architekt. Aus diesem Grund werden in diesem Abschnitt wesentliche Aspekte der Systemtheorie kurz behandelt.

Architektur-Disziplinen im Zusammenspiel

Architektur und Systeme

Definition: System

Als Ausgangspunkt für die Ausführungen dient die im Folgenden vorgestellte Systemdefinition [Wikipedia 2008a]:

Ein System ist eine Gesamtheit von Elementen, die so aufeinander bezogen sind und in einer Weise wechselwirken, dass sie als eine aufgaben-, sinn- oder zweckgebundene Einheit angesehen werden können und sich in dieser Hinsicht gegenüber der sie umgebenden Umwelt abgrenzen.

Systeme bestehen aus Teilen

Nach dieser allgemeinen Definition ist ein System eine Einheit, die aus wechselseitig interagierenden Teilen respektive Systembausteinen besteht. Ein System kann aus Subsystemen oder feingranulareren Systembausteinen bestehen und hierarchisch aufgebaut sein. Mit anderen Worten kann ein System als Systembaustein anderer Systeme angesehen werden.

Systeme besitzen eine Systemgrenze

Ein System besitzt eine Systemgrenze, die es von seiner Umwelt trennt. Dieser Sachverhalt wird in Abbildung 3.3-1 illustriert.

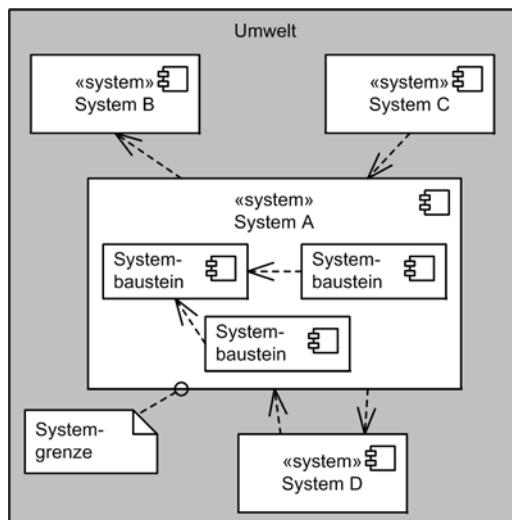


Abb. 3.3-1: System im Kontext seiner Umwelt.

Systeme haben Ziele

Ein System existiert immer zur Erreichung eines Ziels. Eine Fußballmannschaft, als ein Beispiel für ein organisatorisches System, existiert beispielsweise mit dem Ziel, die Meisterschaft zu gewinnen. Genauso bildet sich eine Projektgruppe mit dem Ziel, ein Projekt erfolgreich abzuschließen und das damit verbundene Ziel zu erreichen. Ein IT-System existiert wiederum zur Erreichung fachlicher Ziele (siehe Abschnitt 3.2).

Systeme können mit ihrer Umwelt interagieren und Informationen austauschen, um ihr Ziel zu erreichen. Das Beispiel aus Abbildung 3.3-1 zeigt, dass das betrachtete System A auf Informationen von System B angewiesen ist und System C mit Informationen versorgt. Des Weiteren findet eine beidseitige Kommunikation zwischen System D und dem Beispielsystem A statt. Je nachdem ob ein System mit seiner Umwelt interagiert, unterscheidet die Systemtheorie zwischen folgenden Systemarten:

- > *Offene Systeme* stehen in Kontakt mit ihrer Umwelt und tauschen mit ihr Informationen aus. Darüber hinaus kann ein Energieaustausch stattfinden. Sie müssen mit ihrer Umwelt interagieren, um existieren zu können.
- > *Geschlossene Systeme* tauschen mit ihrer Umwelt zwar keine Informationen aus, jedoch stehen sie mit ihr in einer energetischen Beziehung.

Systemarten

Geschlossene Systeme sind in der Praxis so gut wie nie vorzufinden, da sie immer in Wechselwirkung mit ihrer Umwelt stehen. Ein Haus als System der klassischen Architektur ist in aller Regel an die Strom- und Wasserversorgung, die von seiner Umwelt bereitgestellt werden, angegeschlossen. Ferner ist der Bau einer Straße im Bereich der Raumplanung immer eingebettet in die übergreifende Straßenplanung seiner Umwelt respektive des Straßennetzes.

Existieren geschlossene Systeme?

Eine wichtige Erkenntnis der Systemtheorie ist die Emergenz von Systemen. Emergenz besagt, dass ein System Eigenschaften besitzt, die es von seinen Systembausteinen unterscheidet. Demnach weist kein Systembaustein diese Eigenschaften auf, sondern diese entstehen erst durch das Zusammenspiel der einzelnen Systembausteine. Mit anderen Worten ist das Ganze (das System) mehr als die Summe seiner Einzelteile (Systembausteine) [Rechtin 1991]. Die emergenten Eigenschaften von Systemen existieren somit nur auf der Ebene des Systems und nicht auf der untergeordneten Ebene seiner Systembausteine. Im Hinblick auf Architektur bedeutet dies, dass jede *Architektur-Ebene* unterschiedliche emergente Eigenschaften aufweist (siehe Kapitel 4).

Emergenz von Systemen

Die Entstehung eines Tornados ist ein Beispiel für die Emergenz von Systemen. Als Systembausteine sind in diesem Zusammenhang sowohl feuchte, warme als auch trockene, kalte Luftmassen zu verstehen. Erst durch das Zusammentreffen und Zusammenspiel dieser Systembausteine kann sich ein Tornado bilden. Das Gesamtsystem, der Tornado, verhält sich dabei vollkommen anders als seine Bausteine. Er besitzt also

Beispiele für emergente Systeme

Merkmale, die ihn deutlich von denen seiner Systembausteine unterscheiden. Ähnlich verhält es sich mit dem menschlichen Gehirn, das aus vielen Neuronen besteht und erst durch deren Zusammenspiel imstande ist, zu denken. Das Verhalten von Systemen lässt sich also nicht allein durch das Verhalten ihrer einzelnen Systembausteine erklären. In der IT zeigt sich die Emergenz von Systemen sehr oft in Großprojekten, wie beispielsweise einer automatischen, satellitengestützten Mauterfassung oder dem vollautomatischen Transport von Gepäckstücken in einem Flughafen.

Holismus

Die ganzheitliche Betrachtung von Systemen untersucht ein System in seiner Gesamtheit. Es erfolgt eine Konzentration auf die emergenten Systemeigenschaften, die durch die Interaktion der Systembausteine entstehen. Erst hierdurch lässt sich entscheiden, ob eine Architektur tragfähig ist, da Aussagen über das Gesamtverhalten nur durch eine ganzheitliche Sicht gemacht werden können. Diesen als Holismus bezeichneten Ansatz verdeutlicht Abbildung 3.3-2. Die Subsysteme werden dabei als Black Box angesehen.

Reduktionismus

Im Gegensatz zum Holismus werden die einzelnen Systembausteine im Sinne des Reduktionismus getrennt voneinander analysiert. Bezogen auf Abbildung 3.3-2 bedeutet dies, dass beispielsweise nur Subsystem A1 inklusive seiner Systembausteine näher beleuchtet wird (siehe Abbildung 3.3-3). Diese Sichtweise hilft, konkrete Aussagen über das Verhalten und die Funktionsweise einzelner Systembausteine zu treffen. Ein Subsystem wird somit als White Box wahrgenommen. Aufgrund der inhärenten Emergenz von Systemen ist es jedoch so nicht möglich, das Verhalten des Gesamtsystems zu bestimmen.

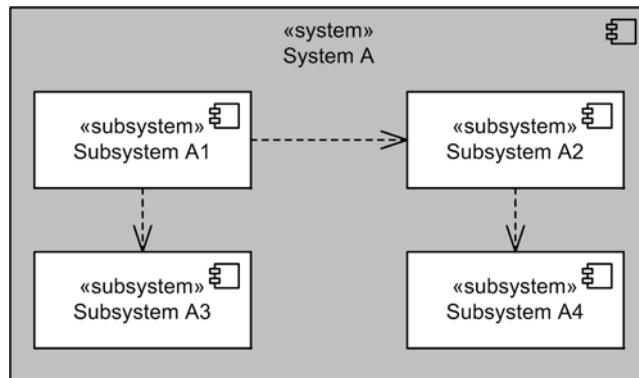


Abb. 3.3-2: Systembetrachtung im Sinne des Holismus.

Deshalb sind der Holismus und der Reduktionismus als komplementäre Ansätze zu verstehen. Nur wenn man weiß, aus welchen Systembausteinen ein System besteht (*Reduktionismus*), lassen sich auch Aussagen über das holistische Verhalten des Systems treffen (*Holismus*).

Holismus und Reduktionismus sind komplementär

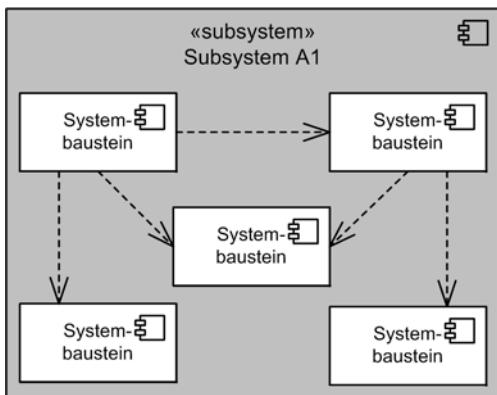


Abb. 3.3-3: Systembetrachtung im Sinne des Reduktionismus.

Das Denken in Systemen lässt sich als *architektonisches Denken* verstehen, das die wesentlichen Bausteine einer Architektur sowie deren Zusammenspiel behandelt und Aussagen darüber trifft, wie die Anforderungen an das System durch die Architektur getragen werden. Ferner beinhaltet dieser Ansatz die Besinnung auf Architektur-Erfahrung in Form von Architekturstilen. Letztlich ist architektonisches Denken eine Methode, um sich zielführend mit Architektur zu beschäftigen. Im architektonischen Sinn steht hierbei die holistische Betrachtungsweise im Vordergrund.

Denken in Systemen

Um ein System beziehungsweise dessen Architektur im Sinne der Systemtheorie zu untersuchen, ist es wichtig, die für die jeweilige Betrachtung relevante Systemgrenze zu wählen. Falls die Systemgrenze nicht passend gezogen wird, leidet das Ergebnis der Betrachtung, da unter Umständen zu viel respektive zu wenig Aspekte des Systems dargestellt werden. Das folgende Beispiel skizziert die Wahl der Systemgrenze anhand der Planung eines Heizungssystems.

Wahl der Systemgrenze

Eine Heizungsanlage ist ein System mit dem Ziel, Behaglichkeit in einem Gebäude, z. B. einem Haus, zu erreichen. Das zu beheizende Gebäude stellt die Systemgrenze dar. Eine Heizungsanlage besteht aus den Systembausteinen *Energiezufuhr*, *Wärmeerzeuger*, *Wärmeverteilung*, *Heizflächen* und *Regelung*. Ein einzelner Raum des Gebäudes kann als ein Subsystem betrachtet werden, das Systembausteine des Gesamtsystems enthält.

Beispiel zur Wahl der Systemgrenze

Bei der Planung, Konzeption und Realisierung eines Heizungssystems wird in einem ersten Schritt im Sinne des Reduktionismus vorgegangen, indem zunächst jeder einzelne Raum als System betrachtet wird, um im Anschluss auf Basis der einzelnen Räume das Gesamtsystem zu erstellen. Die Systemgrenze wird also zu Beginn bewusst eng gefasst, um ein Heizungssystem zu erhalten, das die Heizanforderungen jedes einzelnen Raumes befriedigt. Bei der Betrachtung der Räume müssen z. B. folgende Aspekte geklärt werden:

- > Raumnutzung zur Bestimmung der benötigten Innentemperatur (Anforderung).
- > Bauausführung mit Wärmedämmung der Raumumfassungsflächen (Grenzen des Subsystems).
- > Anzunehmende Raumtemperatur der Nachbarräume (weitere Subsysteme), um Wärmeverluste beziehungsweise -gewinne festzustellen (Kommunikation und Energieaustausch mit anderen Subsystemen).

Die Ergebnisse dieser Untersuchungen dienen als Basis zur Dimensionierung der für die Räume benötigten Heizkörper und der erforderlichen Heizleistung.

Es reicht jedoch nicht aus, nur die Bedürfnisse der einzelnen Räume zu betrachten und zu summieren, um die an die Heizungsanlage gestellten Anforderungen zu bestimmen. Vielmehr muss das gesamte Gebäude als System angesehen und die Systemgrenze entsprechend verschoben werden. Es erfolgt also eine holistische Betrachtung. Auf der Architektur-Ebene *Gebäude* ist es wichtig, den Standort des Gebäudes und die damit verbunden klimatischen Bedingungen wie tiefste Außentemperatur und Windstärke zu kennen.

Aus den durch die Veränderung der Systemgrenze erhaltenen Erkenntnissen kann ein Heizungssystem konzipiert werden, das den Anforderungen des gesamten Gebäudes gerecht wird. Vollzieht man diesen Systemgrenzenwechsel nicht, erhält man beispielsweise ein Heizungssystem, das das Gebäude nicht mit genügend Wärme versorgen kann oder überdimensioniert und dadurch im Energieverbrauch nicht wirtschaftlich ist.

3.4 Architektur und die Bausteine eines Systems

Grundlegendes Verständnis von Systembausteinen

Architektur beschäftigt sich ganz allgemein mit der Strukturierung von Systembausteinen eines Systems. Software-Architektur legt dabei den Fokus auf die Software-Bausteine eines Systems. Unabhängig davon, mit welcher Architektur-Disziplin man sich schwerpunktmäßig in der Informationstechnologie beschäftigt, ist es wichtig, die grundlegenden Arten von Systembausteinen zu kennen. Dies unterstützt das architektonische Denken, da es auf einem höheren Abstraktionsniveau illustriert, aus welchen Bausteinen ein System besteht, wie diese Bausteine miteinander in Beziehung stehen und welche Bedeutung den einzelnen

Bausteinen zukommt. Dadurch werden Systeme greifbar und Entscheidungen auf verschiedene Bausteine fokussierbar.

Des Weiteren entsteht hierdurch ein gemeinsames Vokabular beziehungsweise eine gemeinsame Taxonomie, die in allen Architektur-Disziplinen der Informationstechnologie Anwendung finden kann. Aufgrund dessen kann damit auch die Zusammenarbeit von Architekten der einzelnen Disziplinen verbessert werden, da sie ein gemeinsames Verständnis von Systemen besitzen.

Die Architektur-Disziplinen befassen sich mit unterschiedlichen Gesichtspunkten eines Systems. Aus diesem Grund stehen für Architekten der einzelnen Disziplinen verschiedene Bausteine beziehungsweise Aspekte von Bausteinen bei ihrer Tätigkeit im Vordergrund. Architekten betrachten dabei Bausteine unter verschiedenen Blickwinkeln. Ein Software-Architekt wird z. B. einen Software-Baustein primär hinsichtlich seiner Funktionalität, Verantwortung und Schnittstellen betrachten. Im Gegensatz hierzu analysiert ein Sicherheitsarchitekt, ob der Software-Baustein den Sicherheitsansprüchen genügt und beispielsweise keine in Klartext gespeicherten Passwörter verwendet.

In dem in Abbildung 3.4-1 vorgestellten Modell werden die wichtigsten Systembausteine und ihre Beziehungen zueinander vorgestellt. Der Schwerpunkt liegt hierbei auf der einfachen Darstellung der für Architektur relevanten Bausteine eines Systems.

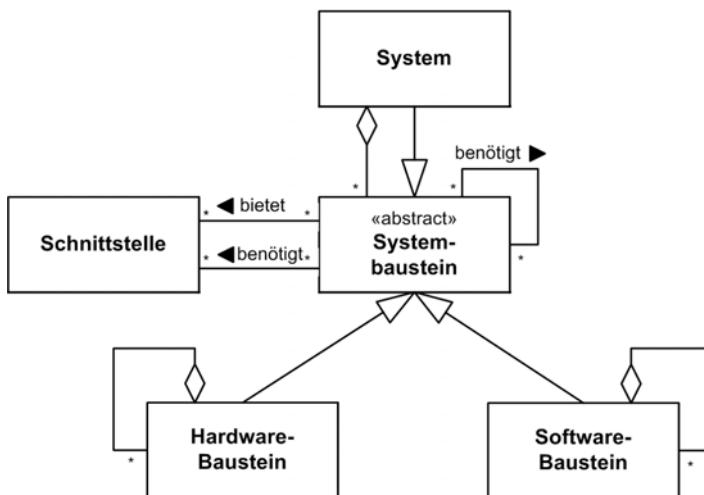


Abb. 3.4-1: Systembausteine und ihre Beziehungen.

Gemeinsames Vokabular

Unterschiedliche Be- trachtungsweise je nach Architektur- Disziplin

Systembausteine und ihre Beziehungen

Systembaustein	Das zentrale Konzept in Abbildung 3.4-1 ist der <i>Systembaustein</i> . Er repräsentiert den abstrakten Typ aller konkreten Bausteine eines Systems. Ein Systembaustein kann andere Systembausteine benötigen und kann über eine oder mehrere Schnittstellen verfügen respektive eine oder mehrere Schnittstellen anderer Systembausteine erfordern.
System	Ein <i>System</i> besteht aus Systembausteinen und ist selbst von diesem Typ. Dadurch kommt zum Ausdruck, dass ein System auch Subsysteme beinhalten kann.
Subsystem	<i>Subsysteme</i> vereinen kohärente Funktionalität und sind in sich selbst abgeschlossen. Somit bietet ein Subsystem zusammengehörende Funktionalität, die einen Teil der an das System gestellten Anforderungen befriedigt.
Software- und Hardware-Bausteine	Als Spezialisierungen von Systembaustein können <i>Software- und Hardware-Bausteine</i> angesehen werden. Ein Hardware-Baustein kann einen anderen Hardware-Baustein benötigen und aus Hardware-Bausteinen bestehen. Ein Personal-Computer als Hardware-Baustein besteht beispielsweise aus einem Motherboard, einer Grafikkarte, einer Netzwerk-karte und vielen anderen Hardware-Bausteinen. Hardware-Bausteine besitzen auch Schnittstellen. So wird eine Grafikkarte beispielsweise in eine Schnittstelle des Motherboards eingesetzt.
MIS-Beispiel	Die Beziehungen zwischen den verschiedenen Systembausteinen werden im Folgenden an einem Beispiel näher betrachtet. Das Hauptaugenmerk liegt dabei auf den Software-Aspekten von Systemen. Als Beispiel dient ein Management-Informations-System (MIS) zur Erhebung und Auswertung von Geschäftskennzahlen. Dieses Beispiel wird in Kapitel 8 wieder aufgegriffen und verfeinert.
Systembausteine benötigten Systembausteine	In dem MIS ist es beispielsweise notwendig, Daten, die als Basis zur Berechnung der Kennzahlen dienen, aus anderen Systemen zu importieren. Die hierfür benötigte Funktionalität kann in einem <i>Import-Subsystem</i> vereint und von anderen Subsystemen des MIS genutzt werden. Dieses Beispiel verdeutlicht bereits, dass Subsysteme miteinander kommunizieren, um die Anforderungen des Systems zu erfüllen. Im Fall des <i>Import-Subsystems</i> könnte ein <i>Benutzerschnittstellen-Subsystem</i> , das den Wunsch des Benutzers entgegennimmt, einen Import durchzuführen, mit dem <i>Import-Subsystem</i> kommunizieren und einen Import initialisieren (siehe Abbildung 3.4-2).

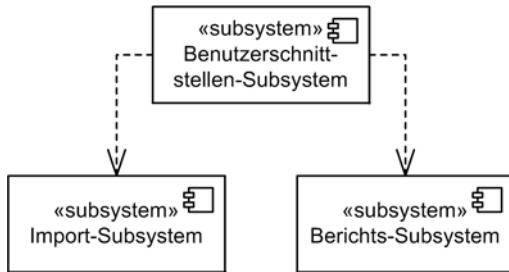


Abb. 3.4-2: Subsysteme benötigen andere Subsysteme.

Darüber hinaus können Systembausteine aus weiteren Systembausteinen bestehen, um diese weiter zu strukturieren. Dies kann wieder anhand der Subsysteme des MIS-Beispiels illustriert werden. Eine wichtige Funktionalität des MIS ist die Generierung von Geschäftskennzahlen-Berichten. Dieser Aufgabe könnte sich ein *Berichts-Subsystem* widmen. Es kann angebracht sein, dass Berichte über unterschiedliche Publikationskanäle wie E-Mail, HTML oder das Adobe Portable Document Format (PDF) verteilt werden. Die Logik zur Generierung kanalspezifischer Berichte kann in dedizierte Subsysteme des Berichts-Subsystems eingebettet werden (siehe Abbildung 3.4-3).

Systembausteine bestehen aus Systembausteinen

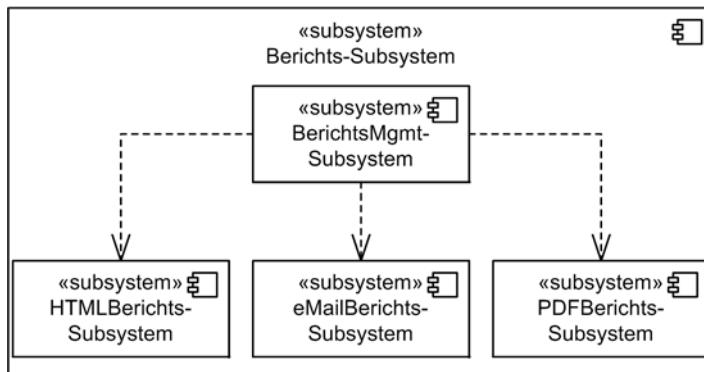


Abb. 3.4-3: Systembausteine bestehen aus Systembausteinen.

Systeme bestehen aus Hardware- und Software-Bausteinen. Diese Bausteine verfügen über *implizite* oder *explizite Schnittstellen*. Eine Schnittstelle definiert einen Vertrag zwischen dem Systembaustein, der die Schnittstelle anbietet, und den Systembausteinen, die sie nutzen. Des Weiteren legt eine Schnittstelle die Operationen fest, die von dem Systembaustein angeboten werden. Eine *explizite* Schnittstelle ist losgelöst vom eigentlichen Systembaustein. Das Konzept der expliziten Schnittstelle wird beispielsweise durch Technologien wie Enterprise Java Beans

Zugriff auf Systembausteine über Schnittstellen

oder Web Services umgesetzt. *Implizite* Schnittstellen sind dahingegen direkte Bestandteile des Software-Bausteins. Ein C-Modul ist ein Beispiel für einen Software-Baustein, der über eine implizite Schnittstelle verfügt. Das *Berichts-Subsystem* unseres Beispielsystems enthält z. B. einen Software-Baustein *BerichtsMgr*, der eine Schnittstelle zur Generierung von E-Mail-, HTML- oder PDF-Berichten für Geschäftskennzahlen bietet. Über diese Schnittstelle kann die Erzeugung von Berichten von Seiten des Benutzerschnittstellen-Subsystems angestoßen werden.

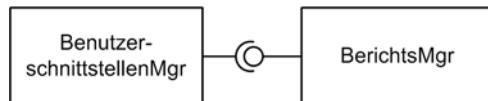


Abb. 3.4-4: Systembausteine nutzen Systembausteine über Schnittstellen.

Die in Abbildung 3.4-4 dargestellte Beziehung verdeutlicht, dass der Software-Baustein *BenutzerschnittstellenMgr* über die Schnittstelle des *BerichtsMgr*-Bausteins die Erzeugung von Berichten initiiert.

Software-Bausteine bestehen aus Software-Bausteinen

Zur Erzeugung von konkreten PDF-Berichten wird im MIS ein *PDFBerichtsMgr* eingesetzt. Dieser besteht aus weiteren Software-Bausteinen, wie z. B. einem *PDFBerichtsabsatzErzeugerROI* zur Generierung eines Absatzes innerhalb des Berichts, der Informationen zum Return On Investment enthält (siehe Abbildung 3.4-5).

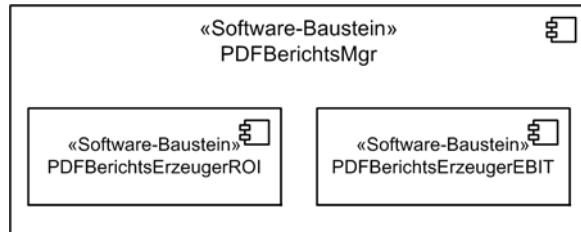


Abb. 3.4-5: Software-Bausteine bestehen aus Software-Bausteinen.

Dieser Sachverhalt gilt selbstverständlich auch für Hardware-Bausteine. Da der Fokus in diesem Zusammenhang jedoch auf Software-Bausteinen liegt, wird auf ein konkretes Beispiel verzichtet.

Schichten

Ein weiteres, wichtiges Konzept, welches im eingeführten Modell nicht enthalten ist, ist die Anordnung eines Systems in *Schichten* (englisch: *layers*). Ein System kann in Schichten organisiert sein, die Subsysteme beinhalten. Eine *Schicht* dient zur logischen Strukturierung eines Systems in Hierarchieebenen. Subsysteme einer Schicht besitzen gemein-

same Merkmale und Aufgaben. Sie können nur auf Subsysteme darunter liegender Schichten zugreifen. Je nachdem wie streng diese Konvention ausgelegt wird, sind sogar nur Zugriffe auf die nächsttiefere Schicht erlaubt. Für eine genaue Beschreibung dieses Prinzips sei an dieser Stelle auf das Layers-Architekturmuster verwiesen [Buschmann et al. 1996].

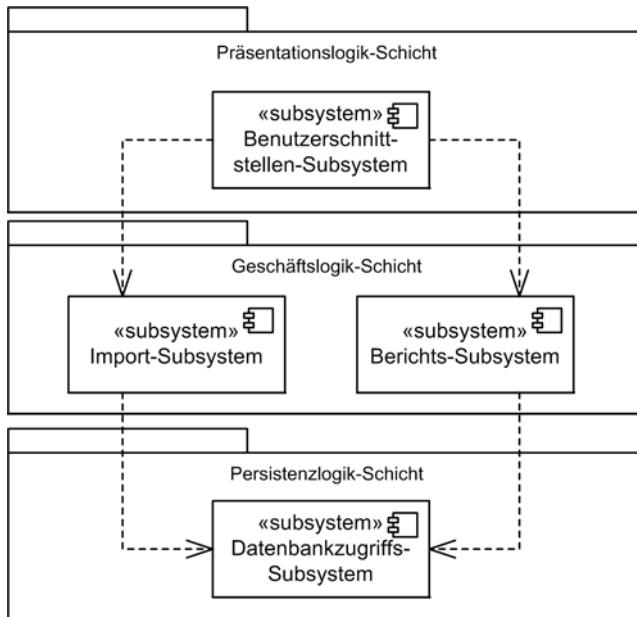


Abb. 3.4-6: Ein System ist organisiert auf Schichten.

Abbildung 3.4-6 illustriert die *Schichten* des MIS-Beispiels und die Positionierung der *Subsysteme* je nach Aufgabenbereich. Es wird deutlich, dass das MIS in eine Präsentationslogik-, Geschäftslogik- und Persistenzlogik-Schicht unterteilt ist. Das *Import*- und das *Berichts-Subsystem* sind auf der Geschäftslogik-Schicht angesiedelt, und beide können von dem *Benutzerschnittstellen-Subsystem* der Präsentationslogik-Schicht genutzt werden.

Bislang wurden die grundlegenden Bausteine eines Systems vorgestellt. Abschließend wird das eingeführte Modell um den Plattform-Aspekt erweitert. Dies wird in Abbildung 3.4-7 dargestellt.

Eine *Plattform* ist ein System, welches aus Software- und ggf. Hardware-Bausteinen bestehen kann. Sie dient zur Ausführung von Software-Bausteinen eines Systems und ist Teil der ganzheitlichen Betrachtung von Software-Architektur.

Plattform

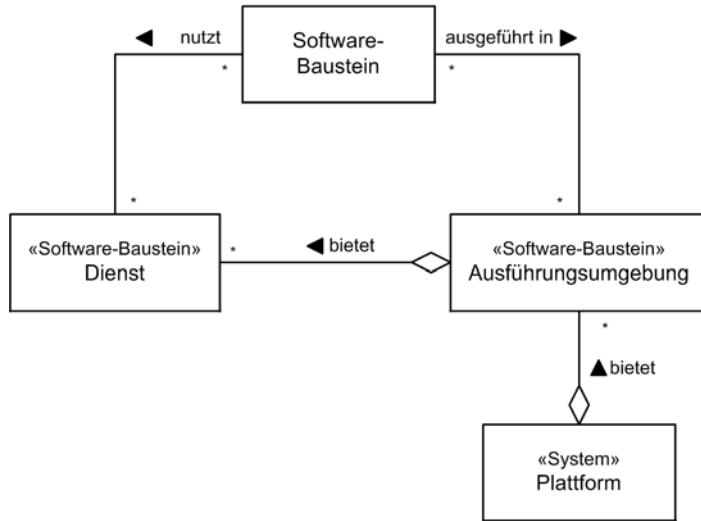


Abb. 3.4-7: Plattform, Ausführungsumgebung und Dienst.

Ausführungsumgebung

Eine Plattform bietet *Ausführungsumgebungen*, in denen Software-Bausteine ausgeführt werden. Eine Ausführungsumgebung ist selbst wieder ein Software-Baustein, der Dienste bereitstellt. Ein JEE-Anwendungsserver bietet beispielsweise Ausführungsumgebungen für JEE-Bausteine, wie z. B. Java Servlets oder Enterprise Java Beans (siehe Abschnitt 6.7.5).

Dienste

Dienste sind ebenfalls Software-Bausteine. Sie bieten Basisfunktionalität, die in der Regel unabhängig von jeglicher durch das System realisierten Geschäftsfunktionalität ist. Mit anderen Worten bietet ein Dienst Funktionalität zur Befriedigung nicht-funktionaler Anforderungen. Die Bausteine des MIS-Beispiels benötigen z. B. einen JEE-Plattform. Diese bietet Dienste, wie Ressourcenmanagement, Sicherheit, Transaktionskontrolle oder Persistenz.

3.5 Zusammenfassung

Zusammenfassung: Architekturen und Architektur-Disziplinen

- > Anforderungen und Architektur-Mittel bestimmen die Gestalt einer Architektur.
- > Eine gute Architektur muss elegant, beständig und zweckmäßig sein.
- > Ein System ist eine Einheit, die aus miteinander interagierenden Software- und Hardware-Bausteinen besteht und zur Erfüllung eines fachlichen Ziels existiert. Es kommuniziert zur Erreichung seines Ziels

mit seiner Umwelt und muss den durch die Umwelt vorgegebenen Rahmenbedingungen Rechnung tragen.

- > Die Software-Architektur eines Systems beschreibt dessen Software-Struktur respektive dessen -Strukturen, dessen Software-Bausteine sowie deren sichtbare Eigenschaften und Beziehungen zueinander.
- > Software-Architektur als Disziplin befasst sich mit den architektonischen Tätigkeiten und den hiermit verbundenen Entscheidungen zum Entwurf und zur Umsetzung einer Software-Architektur.
- > Software-Architektur_{Gesamt} = Software-Architektur_{Struktur} + Software-Architektur_{Disziplin}
- > Software-Architektur umfasst die fachliche und die technische Architektur.
- > Die fachliche Architektur entspringt der fachlichen Domäne respektive des Problembereichs, für den das System entwickelt wird. Sie unterteilt das System in fachliche Bausteine, welche für die Realisierung von Fachlichkeit verantwortlich sind.
- > Die technische Architektur im klassischen Sinn ist domänenneutral und widmet sich primär der Realisierung von nicht-funktionalen Anforderungen respektive Qualitäten. Sie definiert technische Bausteine für nicht-funktionale Aspekte, wie z. B. Logging, Auditing, Sicherheit, Referenzdaten, Persistenz und Transaktionsmanagement.
- > Eine Plattform ist ein System, welches aus Software- und ggf. Hardware-Bausteinen bestehen kann. Sie dient zur Ausführung von Software-Bausteinen eines Systems und ist Teil der ganzheitlichen Betrachtung von Software-Architektur.

4 | Architektur-Perspektiven (WO)

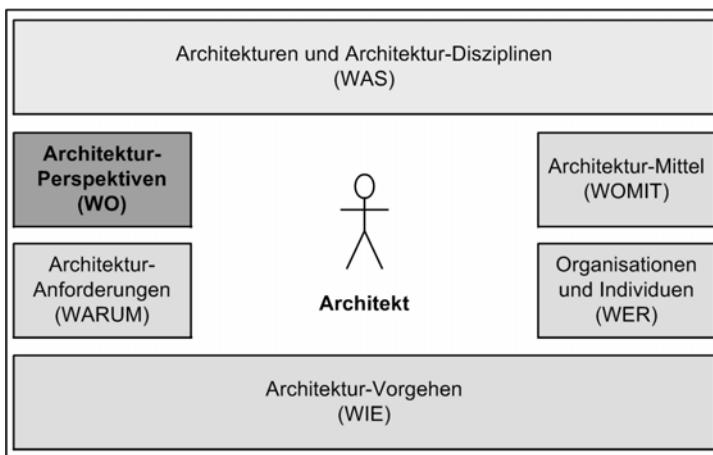


Abb. 4-1: Positionierung des Kapitels im Ordnungsrahmen.

Dieses Kapitel befasst sich mit der *WO-Dimension* des architektonischen Ordnungsrahmens. Es erläutert, auf welchen Abstraktionsstufen sich ein Architekt im Rahmen seiner Tätigkeit bewegt und wie sich Architektur auf diesen Abstraktionsstufen manifestiert. Ferner werden architektonische Sichten vorgestellt, die ein Architekt auf den Abstraktionsstufen verwenden kann, um mit den verschiedenen Aspekten und der damit einhergehenden Komplexität einer Architektur besser umgehen zu können. Nach dem Lesen dieses Kapitels sind Sie in der Lage, die relevanten architektonischen Abstraktionsstufen zu unterscheiden und einzusetzen sowie mithilfe von Architektur-Sichten gezielt verschiedene Aspekte einer Architektur zu betrachten und zu bearbeiten.

Übersicht

4.1	Architektur-Ebenen	72
4.2	Architektur-Sichten	83
4.3	Zusammenfassung	100

Grundlegende Konzepte der WO-Dimension

Abbildung 4-2 stellt die grundlegenden Konzepte, welche in diesem Kapitel behandelt werden, vor und visualisiert ihren Zusammenhang.

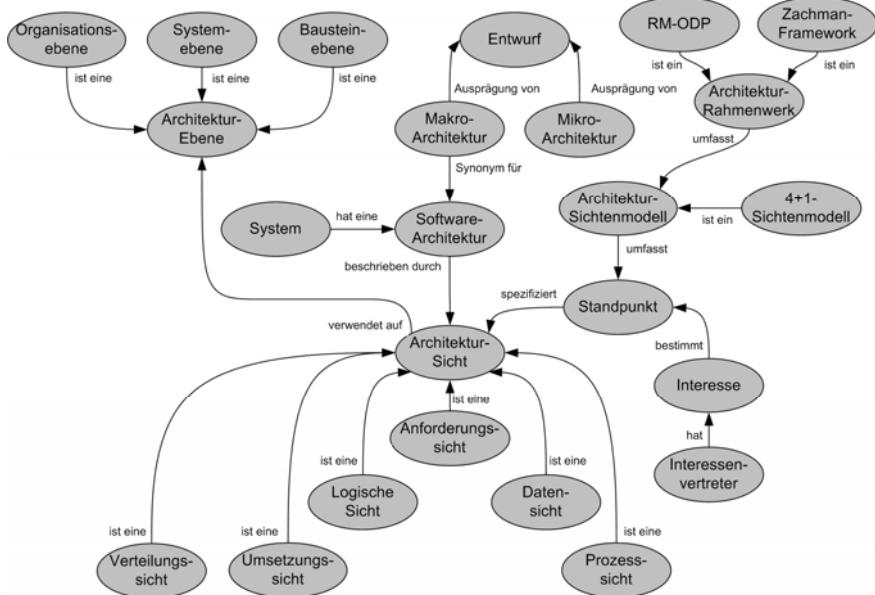


Abb. 4-2: Grundlegende Konzepte der WO-Dimension.

4.1 Architektur-Ebenen

Betrachtung von Systemen durch ein Teleskop

Stellen Sie sich vor, Sie betrachten ein System durch ein Teleskop. Je nach eingestelltem Zoomfaktor wird die Betrachtungsebene verändert, indem bestimmte Details des Systems sichtbar und andere ausgeblendet werden. Plastisch darstellen lässt sich dies am Beispiel des Städte- und Gebäudebaus sowie der Verkehrsinfrastruktur. Wenn Sie die Erde als System mit einem Teleskop von der Raumstation ISS beobachten und dabei langsam heranzoomen würden, könnten Sie verschiedene Architektur-Ebenen in Bezug auf Städte- und Gebäudebau betrachten. Dabei würden Sie sich von der Ebene der Kontinente, über die Ebene der Länder auf die Ebene der Städte und Stadtteile bis hin zu der Ebene einzelner Gebäude und ihrer Stockwerke bewegen. Ein solches Ebenenschema lässt sich auf Software-Architektur übertragen. Welche Architektur-Ebenen können grundsätzlich identifiziert werden? Diese Frage kann beantwortet werden, indem man sich vergegenwärtigt, welche äußeren und inneren Kontexte grundsätzlich bei einem IT-System vorliegen und welchen Abstraktionsgrad diese Kontexte jeweils besitzen.

Organisationen, Systeme und Systembausteine

Für die Erarbeitung der Architektur-Ebenen kommt wieder die Teleskop-Metapher zum Einsatz. Stellen Sie sich dazu bitte vor, Sie möchten ein IT-System durch ein Teleskop betrachten. Bevor Sie jedoch auf ein bestimmtes IT-System fokussieren können, zeigt Ihnen das Teleskop zunächst die äußeren Kontexte des IT-Systems. Dabei sehen Sie zuerst verschiedene Organisationen und wie diese im Rahmen ihrer verschiedenen Rollen (Auftraggeber, Lieferanten, Partner etc.) zusammenarbeiten. In der nächsten Teleskop-Einstellung schauen Sie sich mit dem Teleskop eine bestimmte Organisation genauer an. Sie sehen die Systeme (Mitarbeiter, Abteilungen, IT-Systeme etc.) innerhalb der Organisation und wie diese im Rahmen der Geschäftsprozesse der Organisation zum Einsatz kommen und interagieren. Sie erhöhen den Zoomfaktor des Teleskops weiter und können jetzt ein bestimmtes IT-System der Organisation näher betrachten. Sie sehen seine Schnittstellen, seine Funktionalität und seine Benutzer. Danach zoomen Sie das Innere des Systems heran. Das Teleskop zeigt Ihnen nun im weiteren Verlauf die inneren Kontexte des Systems. Sie erkennen die Bausteine, aus denen das System besteht (siehe Abschnitte 3.3 und 3.4). Sie sehen deren Schnittstellen, Verantwortlichkeiten und Interaktionen. Anschließend erhöhen Sie den Zoomfaktor noch weiter und nun zeigt sich Ihnen das Innenleben der Systembausteine im Detail. Und Sie stellen fest, dass die Systembausteine wiederum aus Systembausteinen bestehen. Weiter unten erfahren Sie beim Thema Mikro-Architektur noch weitere wichtige Sachverhalte zum Innenleben von Systembausteinen. Folgende drei grundsätzliche Architektur-Ebenen haben Sie bis hierhin mit dem Teleskop betrachten können:

- > **Organisationsebene:**
Hier betrachtet man die Organisationen.
- > **Systemebene:**
Hier betrachtet man die Systeme der Organisationen.
- > **Bausteinsebene:**
Hier betrachtet man die Bausteine der Systeme.

Abbildung 4.1-1 zeigt das zugehörige Architektur-Ebenen-Modell, das die Beziehungen zwischen den oben genannten Architektur-Ebenen und ihren Kontext illustriert.

Jeder Architektur-Ebene sind architektonische Anforderungen (siehe Kapitel 5) und Entscheidungen (siehe Kapitel 3.2) zugeordnet, die sich aus Sicht eines IT-Systems jeweils auf einem bestimmten Abstraktionsniveau befinden. Die Architektur-Ebenen sind entlang dieses Abstraktions-

Das Architektur-Ebenen-Modell in Kürze

onsniveaus in einer hierarchischen Reihenfolge angeordnet. Dabei nimmt das Abstraktionsniveau ausgehend von der Bausteinebene zu. Beispielsweise ist die Entscheidung auf der Organisationsebene, Systeme der IT-Landschaft einer Organisation zukünftig mittels einer Standard-Middleware zu integrieren, auf einem höheren Abstraktionsniveau angesiedelt als die Entscheidung auf Systemebene, wie (z. B. mittels XML) ein konkretes System an ein bestimmtes Middleware-Produkt (z. B. IBM WebSphere MQ) anzubinden ist. Die architektonischen Anforderungen und Entscheidungen auf einer untergeordneten, weniger abstrakten Architektur-Ebene berücksichtigen und detaillieren die architektonischen Vorgaben (Anforderungen und Entscheidungen) auf der jeweils übergeordneten Architektur-Ebene. Auf Organisationsebene sind organisationsweite Anforderungen und Entscheidungen angesiedelt. Auf Systemebene fließen die Vorgaben der Organisationsebene in die Anforderungen und Entscheidungen in Hinblick auf die IT-Systeme einer Organisation ein. Auf der untersten Architektur-Ebene schließlich, der Bausteinebene, dienen die Vorgaben der Systemebene als Grundlage für Anforderungen und Entscheidungen, die die Bausteine (siehe Abschnitt 3.4) eines Systems betreffen.

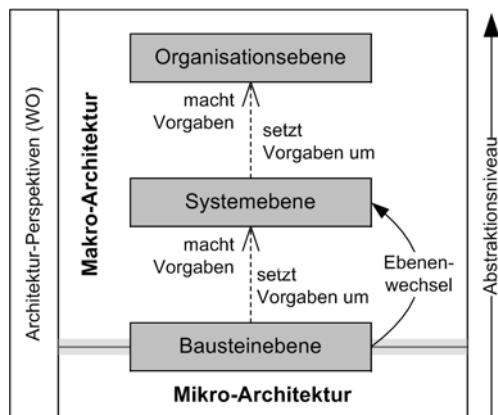


Abb. 4.1-1: Modell der grundsätzlichen Architektur-Ebenen.

Die Berücksichtigung von Architektur-Ebenen führt zu qualitativ hochwertigeren Architekturen

Mithilfe von Architektur-Ebenen ist man sich als Architekt den Kräften und ihren Ursprüngen bewusster, die auf eine Architektur grundsätzlich immer einwirken. Auf Bausteinebene wirken Kräfte von der Systemebene und auf Systemebene Kräfte von der Organisationsebene auf eine Architektur. Durch das Bewusstsein über diese Sachverhalte werden bei der Erstellung einer Architektur die auftretenden Problem- und Fragestellungen auf einheitlichere Weise behandelt und die Vermischung von unterschiedlichen Aspekten eher vermieden. Beispielsweise

sollte die Frage, wie Systeme ihre Daten austauschen sollten, nicht auf der Baustein-Ebene beim Entwurf eines fachlichen Bausteins, wie z. B. Kunde, beantwortet werden, sondern auf der Organisations- bzw. Systemebene einheitlich für alle fachlichen Bausteine. Wenn man also Architektur-Ebenen beachtet, führt dies zu einer Architektur, die in sich einheitlicher und konsistenter ist. Die Berücksichtigung von Architektur-Ebenen führt zusammengefasst aus folgenden Gründen zu qualitativ hochwertigeren Architekturen:

- > Architektur-Probleme/-Aspekte werden passenden Ebenen zugeordnet und sind damit einfacher und einheitlicher zu handhaben [Brown et al. 1998].
- > Unterschiedliche Architektur-Probleme/-Aspekte werden nicht vermischt, sondern getrennt mit den jeweils probaten Mitteln behandelt.
- > Einflüsse auf eine Architektur liegen explizit vor und können deshalb besser verstanden und berücksichtigt werden.

Ein Architekt muss sich auf jeder Architektur-Ebene mit spezifischen Problemstellungen befassen und diese lösen. Beispielsweise beschäftigt er sich auf der Organisationsebene mit Geschäftsprozessen, auf der Systemebene mit Systemanwendungsfällen und auf der Baustein-Ebene mit der Erstellung von Systembausteinen, welche die Bausteinanwendungsfälle umsetzen. Dies sollte jedoch nicht nur auf der Baustein-Ebene, sondern auf jeder der genannten Architektur-Ebenen immer unter Berücksichtigung der architektonischen Prinzipien aus Kapitel 6 geschehen. Als Beispiel sei das Prinzip der hohen Kohäsion genannt (siehe Abschnitt 6.1.2). Auf Organisationsebene sollten die Schritte innerhalb von Geschäftsprozessen zueinander in hoher Kohäsion stehen. Zum Beispiel sollten in einem Geschäftsprozess „Auftrag erfassen“ keine Schritte aus dem Geschäftsprozess „Rechnung stellen“ enthalten sein. Auf Systemebene gilt dies für die Schritte innerhalb der Systemanwendungsfälle. Und auf Baustein-Ebene wird hohe Kohäsion z. B. bei Systembausteinen wie Komponenten oder Klassen angewandt. Nichtbeachtung dieser Prinzipien führt im Wesentlichen zu zahlreichen unnötigen Abhängigkeiten, zu falschen Zuordnungen von Verantwortlichkeiten und zu Redundanzen bei den Artefakten (z. B. Klassendiagramme oder Quelltext), die den Architektur-Ebenen zugeordnet sind. Dies wirkt sich vor allem negativ auf die Umsetzung nicht-funktionaler Anforderungen (siehe Kapitel 5) und damit auf die Software-Qualität aus.

Auf allen Architektur-Ebenen müssen Architektur-Prinzipien beachtet werden

Architektur-Ebenen lassen sich weiter unterteilen

Bei den vorgestellten Ebenen handelt es sich um die grundlegenden Architektur-Ebenen. Für die Bedürfnisse dieses Buches ist diese Einteilung ausreichend detailliert. Jede der genannten Architektur-Ebenen lässt sich jedoch bei Bedarf in weitere Architektur-Ebenen aufteilen. Das Software Design Level Model (SDLM) [Brown et al. 1998] ist ähnlich aufgebaut, zerlegt jedoch beispielsweise die Systemebene in die drei Architektur-Ebenen Systeme, Anwendungen und Rahmenwerke (englisch: *framework*).

Architektur-Ebenen und der architektonische Ordnungsrahmen

Neben den architektonischen Anforderungen und Entscheidungen sind ebenfalls alle anderen Elemente des architektonischen Ordnungsrahmens (siehe Kapitel 2) entsprechend des Abstraktionsniveaus einer der Architektur-Ebenen zugeordnet. Kommen neue Elemente in den Ordnungsrahmen hinzu, sollte ein Architekt diese den passenden Architektur-Ebenen zuordnen. Auf diesem Weg wird ein systematischer und bewusster Umgang mit Architektur-Wissen etabliert. In Tabelle 4.1-1 wird für einige beispielhafte Inhalte der Dimensionen WAS, WO, WARUM, WOMIT, WER und WIE des Ordnungsrahmens die Zuordnung zu den Architektur-Ebenen aufgezeigt.

Tab. 4.1-1: Architektur-Ebenen und architektonischer Ordnungsrahmen.

Ebene Dimension	Organisationen	Systeme	Bausteine
WAS	Enterprise-Architektur, Geschäftsprozesse, ...	Software-Architektur, ...	Software-Architektur, ...
WO	Anforderungs-sicht, ...	Anforderungs-sicht, logische Sicht, ...	Anforderungs-sicht, logische Sicht, ...
WARUM	Organisationsanforderungen, IT-Standards, -Richtlinien, ...	Systemanforde-rungen, ...	Bausteinanforderungen, ...
WOMIT	Geschäftsprozessbeschreibun-gen, Business Use Case, ...	Systemkon-textdiagramm, Architektur-Muster, Frameworks, ...	Architektur-Muster, Ent-wurfsmuster, Frameworks, ...
WER	Enterprise-Architekt, ...	Software-Architekt, ...	Software-Architekt, ...
WIE	Erstellen der Sys-temvision, ...	Erstellen der Systemvision, Entwerfen der Architektur, ...	Entwerfen der Architektur, ...

Bevor die Architektur-Ebenen nun im Detail erläutert werden können, ist zum Gesamtverständnis die Bedeutung der zwei bis hierhin noch nicht betrachteten Begriffe Makro- und Mikro-Architektur aus der Abbildung 4.1-1 zu erörtern. Beide Begriffe stehen im Zusammenhang mit dem Begriff Entwurf. Dieser Begriff ist zentral, weil ein Architekt seine verschiedenen Tätigkeiten (siehe Kapitel 8) im Kontext der verschiedenen Architektur-Ebenen mit dem Ziel ausführt, einen Entwurf für die Architektur eines Systems zu erarbeiten. Was genau ist jedoch mit Entwurf gemeint? Zum Begriff (Software-)Entwurf findet sich bei [IEEE 2004] folgende Erläuterung:

Design is defined in [IEEE610.12-90] as both “the process of defining the architecture, components, interfaces, and other characteristics of a system or component” and “the result of [that] process”.

Gemäß dieser Erläuterung umfasst Entwurf allgemein betrachtet:

- Den Prozess (siehe Abschnitt 8.1), um Architektur, Bausteine, Schnittstellen und andere Eigenschaften eines Systems oder eines Systembausteins festzulegen.
- Das Ergebnis dieses Prozesses selber.

Die Eigenschaften eines Systems oder eines Systembausteins sind sehr zahlreich und auf unterschiedlichen Abstraktionsstufen (Detaillierungsstufen) angesiedelt. Aufgrund dessen muss Entwurf auf unterschiedlichen Abstraktionsstufen stattfinden. Deshalb genügt es nicht, einfach nur allgemein von „Entwurf“ zu sprechen, wenn ein System entworfen wird, sondern man muss die jeweilige Abstraktionsstufe berücksichtigen. Die folgenden zwei weiteren Erläuterungen aus [IEEE 2004] liefern hierzu eine differenziertere Betrachtung von Entwurf:

Software architectural design (sometimes called top-level design): describing software's top-level structure and organization and identifying the various components.

Was ist noch mal (Software-) „Entwurf“ genau?

Software detailed design: describing each component sufficiently to allow for its construction.

Definition:
software architectural design

Definition:
software detailed design

An diesen beiden Erläuterungen wird deutlich, dass Entwurf auf zwei verschiedenen Abstraktionsstufen stattfindet. Das Architektur-Ebenen-Modell unterscheidet deshalb in Anlehnung an [Brown et al. 1998] zwischen:

- > Makro-Architektur
(Software-Architektur, Grob-Entwurf, englisch: *architectural design, top-level/high-level design*) und
- > Mikro-Architektur
(Detail- oder Fein-Entwurf, englisch: *detailed design*).

Um Unklarheiten zu vermeiden, sollte deshalb immer unterschieden werden zwischen „Entwurf“ im allgemeinen Sinne (als Oberbegriff) und „Entwurf“ in Bezug auf Abstraktionsstufen (Makro- oder Mikro-Architektur). Makro-Architektur entspricht Software-Architektur (siehe Abschnitt 3.2) und findet deshalb auf den zuvor besprochenen Architektur-Ebenen (Organisations-, System- und Bausteinebene) statt. Mikro-Architektur hingegen findet nur auf einem Teil der Bausteinebene statt (dazu später mehr).

Wo ist die Grenze zwischen Makro- und Mikro-Architektur?

Wie verhält es sich nun, wenn das Abstraktionsniveau immer weiter abnimmt und die Details zunehmen? Wann hört Architektur auf und wann fängt Detail-Entwurf an? Die Grenze zwischen Makro- und Mikro-Architektur kann nicht immer klar gezogen werden. Diese ist auch abhängig von der Sicht der jeweiligen Interessenvertreter und deshalb ein fließender Übergang.

Makro-Architektur

Makro-Architektur umfasst das Spektrum der Architektur-Ebenen, denen architektonisch relevante Elemente zugeordnet sind (Organisations- und Systemebene sowie den Teilbereich der Bausteinebene, auf dem sich tragende Systembausteine (siehe Abschnitt 3.3) befinden). Damit ist Architektur im „Großen“ gemeint. Diese befasst sich mit Aspekten wie Anforderungen, Entscheidungen und Strukturen auf einem hohen Abstraktionsniveau. Beispielsweise handelt es sich hier um Entscheidungen hinsichtlich wichtiger Systemschnittstellen. Ein Systembaustein, der gemäß dem Fassade-Entwurfsmusters [Gamma et al. 1995] als Fassade für eine Gruppe zusammengehörender Systembausteine fungiert, ist ein konkretes Beispiel hierfür. Die Organisations- und Systemebene sowie der Teil der Bausteinebene, dem die tragenden Bausteine (siehe Abschnitt 3.2) eines Systems zugeordnet sind, gehören zum Bereich der Makro-Architektur.

Mikro-Architektur

Mikro-Architektur dagegen befasst sich mit Aspekten auf einem niedrigen Abstraktionsniveau. Dabei handelt es sich dann um Detail-Entwurf (Architektur im „Kleinen“) mit großer Nähe zum Quelltext ohne fundamentalen Einfluss auf eine Architektur. Zum Bereich Mikro-Architektur gehört der Teilbereich der Bausteinebene, auf dem sich die nicht-tragenden Systembausteine befinden. In Abbildung 4.1-1 wird dieser Aspekt dadurch verdeutlicht, dass die Bausteinebene auf der Grenze

zwischen Makro- und Mikro-Architektur liegt. Es stellt sich nun die Frage, warum der Bausteinebene überhaupt nicht-tragende Systembausteine zugeordnet sind? Tragende Systembausteine können wiederum aus tragenden Systembausteinen bestehen (siehe Abschnitt 3.3). Eine rekursive Dekomposition der Systembausteine entlang dieser Baustein-hierarchie führt zu Systembausteinen, deren Abstraktionsniveaus tendenziell abnehmen. Irgendwann stößt man auf nicht-tragende Systembausteine mit niedrigem Abstraktionsniveau. Dabei findet dann der Wechsel von Makro- zu Mikro-Architektur statt. Das Beispiel hierzu aus Abbildung 4.1-2 zeigt, wie auf der Bausteinebene (Makro-Architektur) zunächst für den tragenden Systembaustein B eine Dekomposition in die Systembausteine B1, B2 usw. erfolgt. Da diese Bausteine ebenfalls tragend sind, bleibt man im Bereich der Makro-Architektur. Die anschließende Dekomposition des Systembausteins B2 führt zu den Systembausteinen B2', B2'' usw. Diese Bausteine sind nun nicht-tragend und man wechselt für die Betrachtung dieser Bausteine in den Bereich der Mikro-Architektur. Eine Hilfsklasse für Zeichenkettenoperationen ist ein konkretes Beispiel für einen nicht-tragenden Systembaustein. Entscheidungen (z. B. zu Signaturen, Gültigkeitsbereichen von Variablen, Entwurfsmustern etc.) in Bezug auf nicht-tragende Systembausteine sind ein anderes konkretes Beispiel für Elemente, die nicht von architektonischer Relevanz sind und deshalb wie die nicht-tragenden Systembausteine zum Bereich der Mikro-Architektur gehören.

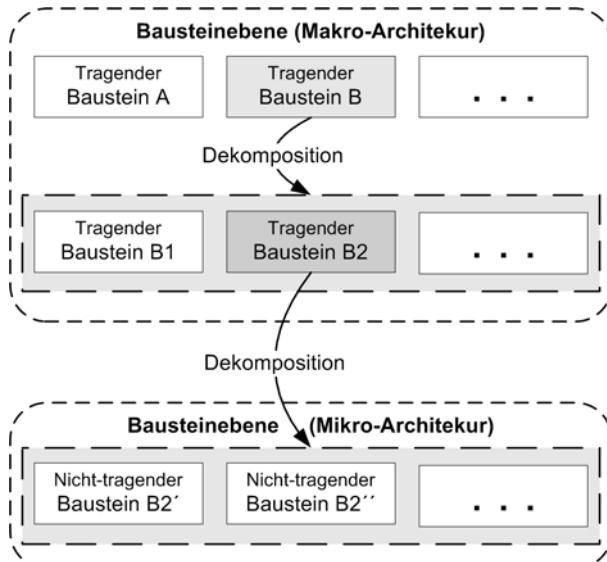


Abb. 4.1-2: Makro- und Mikro-Architektur im Zusammenhang.

4.1.1 Organisationsebene

Bebauungspläne, Geschäftsprozesse und IT-Landschaften

Auf dieser Architektur-Ebene werden Organisationen (z. B. Unternehmen oder Institutionen), deren Bebauungspläne, Geschäftsprozesse und IT-Landschaften sowie deren Interaktionen mit anderen Organisationen (Organisationskontext) betrachtet. IT-Systeme gehören hier zu den an den Geschäftsprozessen einer Organisation beteiligten Akteuren (Abteilungen, Mitarbeiter, Systeme etc.). Der innere Aufbau der IT-Systeme interessiert dabei nicht. Einzelne IT-Systeme werden als Black Boxes behandelt. Beispielsweise könnte der Organisationsebene die Beschreibung eines Geschäftsprozesses für die Auftragsabwicklung einer Organisation mit all seinen beteiligten Systemen (z. B. Kundenabteilung, Buchhaltung, Material- oder Warenwirtschaft) zugeordnet werden.

IT-Standards und -Richtlinien

Des Weiteren liegen im Kontext der Organisationsebene Anforderungen an eine Organisation sowie organisationsweit zu verwendende IT-Standards und -Richtlinien (siehe Kapitel 5). Beispielsweise kann es sich hier um die IT-Richtlinie handeln, XML für den Datenaustausch zwischen verschiedenen Systemen zu verwenden, ohne dabei auf konkrete XML-Technologien, wie XML-Parser etc., einzugehen. Diese Festlegungen ermöglichen es, dass Systeme verschiedener Organisationen bzw. verschiedene Systeme innerhalb einer Organisation zusammenarbeiten können, um integrierte Dienstleistungen anbieten zu können.

Problemfelder mit organisationsweitem Bezug

Insbesondere bei folgenden Problemfeldern wird sich ein Architekt auch auf der Organisationsebene bewegen müssen:

- > IT-gestützte Umsetzung organisationsübergreifender Prozesse (z. B. Supply Chain Management).
- > Enterprise-Architektur (siehe Abschnitt 3.2).
- > Enterprise Application Integration (EAI) (siehe Abschnitt 8.8).

Architektonische Entscheidungen müssen bei diesen Problemfeldern über die Sicht eines einzelnen Systems hinausgehen und deshalb eine organisationsweite Ausrichtung haben. Diese Entscheidungen umfassen z. B. die Vorgabe zu verwendeter IT-Standards. Dabei können beispielsweise Technologien wie JEE oder Architekturmuster wie Model View Controller (MVC) für Architekturen von organisationsweiten Systemen [Fowler 2003] vorgegeben werden. Als Beispiele sind hier zu nennen: Kliniken mit Patientendatenverwaltung, Krankenkassenverrechnung und medizinischem Auskunftssystem oder das Internet mit all seinen IT-Standards und IT-Richtlinien.

4.1.2 Systemebene

Auf der Systemebene werden die IT-Systeme von Organisationen betrachtet (herangezoomt). Der innere Aufbau der Systeme spielt hier nur in Bezug auf deren Subsysteme eine Rolle. Die einzelnen Systeme respektive deren Subsysteme werden jeweils als Black Box behandelt. Die Verantwortlichkeiten, Schnittstellen und Interaktionen der Systeme mit ihrem Kontext stehen hier im Fokus. Die wesentlichen Elemente auf der Systemebene sind:

- > Anforderungen an die Systeme (siehe Abschnitt 5.3).
- > Systemkontakte der Systeme (siehe Abschnitt 8.3).
- > Subsysteme der Systeme (siehe Abschnitt 3.4).

IT-Systeme von Organisationen

Eines der Artefakte, die ein Architekt erstellt und das der Systemebene zugeordnet ist, ist die Architektur-Vision bzw. Architektur-Übersicht (siehe Abschnitt 8.5). Im Beispiel aus Abbildung 4.1-3 betrachtet man dazu zunächst das zu realisierende System (englisch: *system under construction*) A im Kontext seiner Umsysteme U1, U2 und U3. Anschließend führt man eine erste Dekomposition für das System A durch. Es ergeben sich dabei eine Reihe von Subsystemen ($A1'$, $A2'$, $A1''$ usw.). Weil es sich jedoch bei diesen Subsystemen auch wieder um Systeme handelt (siehe Abschnitt 3.3), bewegt man sich weiterhin auf der Systemebene und weist nun die Subsysteme den logischen Schichten X und Y zu (siehe Abschnitt 6.3).

Systemkontext und Subsysteme

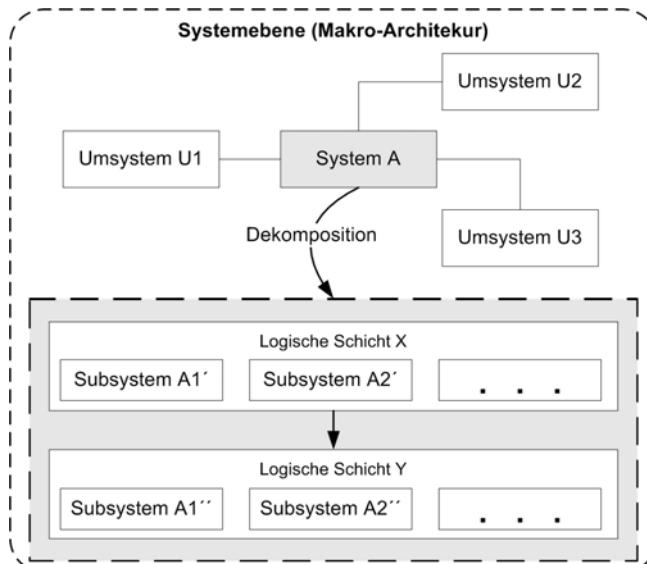


Abb. 4.1-3: Zusammenhang zwischen Systemebene, Systemkontext und Subsystemen.

4.1.3 Bausteinebene

Bausteine der Subsysteme

Auf der Bausteinebene im Bereich Makro-Architektur wird die innere Struktur der einzelnen Subsysteme betrachtet. Es wird in die Subsysteme hineingezoomt (White Box), indem eine Dekomposition der Subsysteme in Software-Bausteine (siehe Abschnitt 3.4) stattfindet. Dabei wechselt man von der System- auf die Bausteinebene. Wesentliche Aspekte, mit denen man sich auf der Bausteinebene beschäftigt, sind die Verantwortlichkeiten der Systembausteine, deren Schnittstellen sowie deren Interaktionen untereinander. Das Beispiel aus Abbildung 4.1-4 zeigt die Dekomposition für das Subsystem B in die tragenden Bausteine B1, B2 usw. Da man nun Software-Bausteine betrachtet, wechselt man von der Systemebene hinunter auf die Bausteinebene.

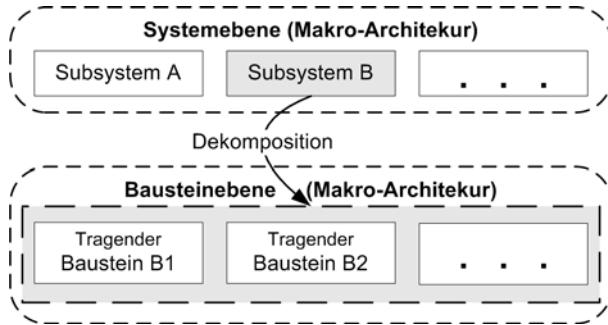


Abb. 4.1-4: Zusammenhang zwischen System- und Bausteinebene.

Ebenenwechsel zwischen Baustein- und Systemebene

Bei sehr großen Systemen kann es sich bei den Systembausteinen, die sich aus der Dekomposition eines Subsystems ergeben haben, wiederum um Subsysteme anstelle von Software-Bausteinen handeln. In diesem Fall findet ein Ebenenwechsel von der Baustein- zurück auf die Systemebene statt. Dort findet die weitere Behandlung des Systembausteins als System ausgehend vom Systemkontext usw. statt (wie oben beschrieben). Dadurch erhält das Architektur-Ebenen-Modell an dieser Stelle einen rekursiven Charakter. Dieser Aspekt wird in Abbildung 4.1-1 durch den Ebenenwechsel-Pfeil von der Baustein- zurück auf die Systemebene angedeutet. Das Beispiel aus Abbildung 4.1-5 betrachtet das Subsystem B. Die Dekomposition dieses Subsystems führt zunächst (wie üblich bei Subsystemen; siehe auch Abbildung 4.1-4) auf die Bausteinebene. Dort betrachtet man die vermeintlichen Software-Bausteine B2, B2 usw. und stellt fest, dass es sich tatsächlich um Subsysteme handelt. Aus diesem Grund wechselt man bei der weiteren Betrachtung des Systembausteins B2 wieder zurück auf die Systemebene.

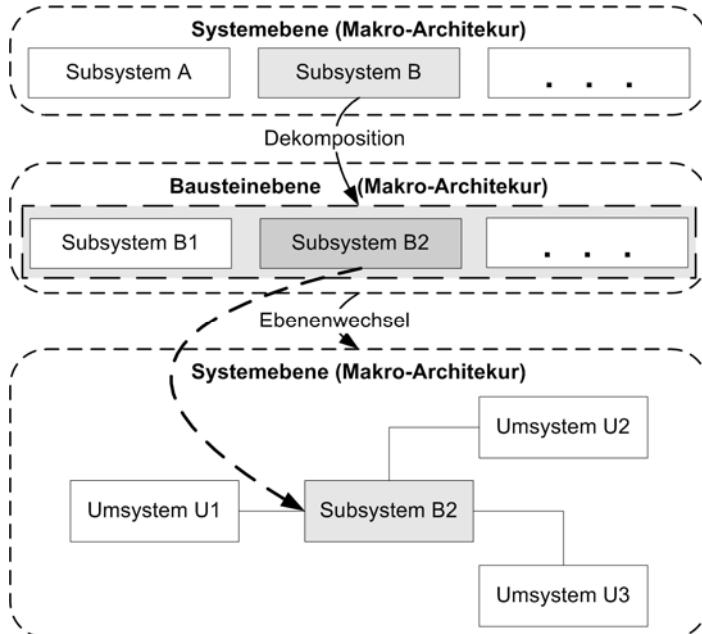


Abb. 4.1-5: Ebenenwechsel von Baustein- auf Systemebene.

4.2 Architektur-Sichten

Alle Aspekte komplexer Systeme (Menschen, Bauwerke, IT-Systeme etc.) zu jedem Zeitpunkt komplett zu erfassen, ist zumindest für die menschliche Wahrnehmung nicht möglich. Dies zu versuchen, wäre darüber hinaus auch nicht zielführend, weil nicht zu jedem Zeitpunkt gleichzeitig alle Aspekte eines Systems relevant sind. Daher ist es sinnvoll, nur bestimmte, für den Moment interessante Aspekte eines Systems betrachten zu können. Für IT-Systeme existiert zu diesem Zweck das Konzept der Architektur-Sichten:

Architektur-Sichten machen komplexe Systeme überschaubar

- > Eine Sicht stellt ein vollständiges System aus dem Blickwinkel einer Menge von zusammenhängenden Interessen heraus dar [IEEE 2000].
- > Eine Sicht ist eine Menge von zusammengehörenden architekturelevanten Elementen, die erstellt und genutzt wird von den Interessenvertretern eines Systems [Bass et al. 2003].

Architektur-Sichten werden durch Interessenvertreter motiviert

Beide Definitionen machen die wichtigste Eigenschaft von Architektur-Sichten deutlich: Architektur-Sichten werden durch die Interessenvertreter eines Systems motiviert („...einer Menge von zusammenhängenden Interessen...“ und „...genutzt wird von den Interessenvertretern eines Systems“). Zusammenhängende Interessen meint hier verschiedene Fragestellungen zu einem bestimmten Aspekt. Ein solcher Aspekt sind z. B. die Strukturen eines Systems. An diesen Aspekt lassen sich verschiedene Interessen anknüpfen, z. B., welche Bausteine liegen vor oder wie sehen die Schnittstellen aus. Eine Architektur-Sicht wird demnach von bestimmten Interessenvertretern (siehe Abschnitt 8.3) genutzt. Diese zeigt deshalb sinnvoller Weise auch nur die, für bestimmte Interessenvertreter wichtigen Aspekte eines Systems. Ein solcher Aspekt sind z. B. Anforderungen. Eine Architektur-Sicht, die diesen Aspekt abdeckt, würde alle Artefakte mit Bezug zu Anforderungen umfassen. In Abschnitt 8.4 finden Sie Beispiele hierzu. Auftraggeber oder Domänenexperte sind Beispiele für Interessenvertreter, die eine solche Anforderungssicht nutzen würden.

Architektur-Sichten nach IEEE

Von der IEEE werden in ihrem Standard 1471 [IEEE 2000] neben der oben bereits genannten Definition weitere wichtige Grundlagen zu Architektur-Sichten gelegt. Dieser Standard beschäftigt sich mit der Beschreibung softwareintensiver Systeme und war anfangs ein Standard für Architektur-Dokumentation (siehe Abschnitt 8.7). Der Überblick (in Anlehnung an [IEEE 2000]) zum Kontext von Architektur-Sichten aus Abbildung 4.2-1 zeigt wesentliche Begriffe und Zusammenhänge in Hinblick auf Architektur-Sichten:

- > Interessenvertreter haben verschiedene Interessen (englisch: *concern*) in Bezug auf ein System respektive auf dessen Architektur. Ausgehend von bestimmten Interessen nehmen Interessenvertreter jeweils einen bestimmten Standpunkt (englisch: *viewpoint*) ein.
- > Auf einem Standpunkt steht Interessenvertretern jeweils eine bestimmte Architektur-Sicht (z. B. Sicht auf Daten) zur Verfügung (Blickwinkelanalogie). Diese wird vom Standpunkt definiert und nimmt Bezug auf eine Menge von bestimmten Interessen.
- > Jede Architektur-Sicht hat Bezug zu genau einem Standpunkt.
- > Architektur-Sichten bestimmen ganz wesentlich Inhalt und Struktur von Architektur-Dokumentation (siehe Abschnitt 8.7).

Definition: Interesse

Ein Interesse ist allgemein wie folgt definiert [Wikipedia 2008b]:

Interessen allgemein

Unter Interesse (von lateinisch *interesse*: dabei sein, teilnehmen an, "dazwischen-stecken/-sein") versteht man die kognitive Anteilnahme respektive die Aufmerksamkeit, die eine Person an einer Sache oder einer anderen Person nimmt.

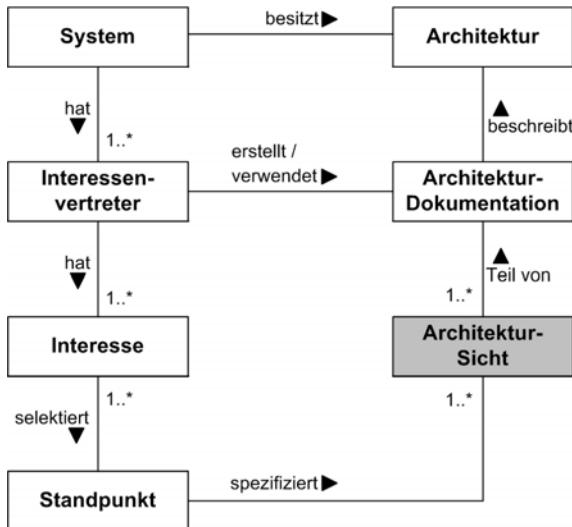


Abb. 4.2-1: Architektur-Sichten im Kontext [IEEE 2000].

Nach dieser Definition steht hinter Interesse Anteilnahme und Aufmerksamkeit, die man einer Sache oder Person entgegenbringt. Bei Anteilnahme und Aufmerksamkeit auf ein IT-System bezogen kann es sich unter anderem um

- > Anforderungen
(z. B. das System muss in bestimmten Intervallen automatisch bestimmte Daten sichern),
- > Sorgen
(z. B. die Sorge, dass die Entscheidung, für ein System eine bestimmte Technologie-Plattform zu verwenden, in einem wichtigen Zukunftsmarkt in eine Sackgasse führen kann) oder
- > Ziele
(z. B. das System soll die Produktivität um 15 % steigern)

handeln. Interessen sind demnach im weitesten Sinne auch Anforderungen. Jedoch genügen Interessen oft nicht den Kriterien, die an Anforderungen gestellt werden (siehe Kapitel 5).

Im Zusammenhang mit Architektur-Sichten werden solche Interessen betrachtet, die allgemeinen generischen Fragestellungen zu einem System respektive seiner Architektur entsprechen. Beispiele hierfür sind:

- > Welches sind die Anforderungen an das System?
- > Aus welchen logischen Bausteinen besteht das System?
- > Welche Schnittstellen haben die Systembausteine?
- > Wie interagieren die Systembausteine?

Interessen und Architektur-Sichten

Standpunkt (englisch: *viewpoint*)

Bei Architektur-Sichten ist zwischen deren Spezifikation und deren konkreter Verwendung als Ausprägung ihrer Spezifikation zu unterscheiden. Beispielsweise kann für eine Architektur-Sicht auf logische Systembausteine die entsprechende Spezifikation festlegen, dass bestimmte UML-Diagramme (z. B. Klassen- und Paketdiagramm) und bestimmte Textvorlagen für die Beschreibung der Systembausteine zu verwenden sind. Eine Ausprägung dieser Spezifikation als Architektur-Sicht auf ein bestimmtes System würde dann eine Reihe von UML-Diagrammen und textuellen Beschreibungen der logischen Systembausteine zeigen. Spezifikationen von Architektur-Sichten können mehr oder weniger umfassend sein (vom Namen bis hin zur methodischen Erstellung einer Sicht). Bei [IEEE 2000] wurde für die Spezifikation von Architektur-Sichten das wichtige Konzept des Standpunkts eingeführt:

- > Bei [IEEE 2000] wird zwischen einer Architektur-Sicht und ihrer Spezifikation begrifflich und konzeptionell unterschieden.
- > Ein Standpunkt entspricht bei [IEEE 2000] der Spezifikation für eine bestimmte Architektur-Sicht (z. B. Verteilungssicht) unabhängig von einem bestimmten System.
- > Eine Architektur-Sicht entspricht bei [IEEE 2000] der Ausprägung ihrer entsprechenden Spezifikation abhängig von einem bestimmten System.

Standpunkte spezifizieren, zu gegebenen Interessen, die jeweiligen Architektur-Sichten respektive wie diese zu erstellen sind. Beispielsweise kann ein Standpunkt für die Fragestellung, welches die logischen Strukturen eines Systems sind, für die entsprechende Architektur-Sichten festlegen, welches Architektur-Modellierungsmittel (siehe Abschnitt 6.6) nach welchen Regeln verwendet werden soll.

Standpunkte spezifizieren für Architektur-Sichten:

- > Name.
- > Interessenvertreter und deren Interessen.
- > Was zu dokumentieren ist (Dokumentationsgegenstände).

Standpunkte spezifizieren für die Erstellung von Architektur-Sichten:

- > Zu verwendende Architektur-Modellierungsmittel.
- > Vorlagen/Muster.
- > Best-Practices.
- > Methoden.
- > Richtlinien.

Durch den Einsatz von Standpunkten wird der Umgang mit Architektur-Sichten erleichtert, weil generische Aspekte bei der Erstellung von Architektur-Sichten besser wiederverwendet werden können und nicht redundant für jedes System neu festgelegt werden müssen. Standpunkte geben einem Architekten ein Rahmenwerk (englisch: *framework*) bzw. eine Vorlage (englisch: *template*) für die Erstellung von Architektur-Sichten an die Hand. Jedoch ist die Erstellung von Standpunkten nicht trivial. Weiterführende Informationen zum Thema Standpunkte finden sich bei [Rozanski und Woods 2005].

Standpunkte erleichtern den Umgang mit Sichten

Warum kommt Architektur-Sichten eine große Bedeutung zu und wie wird mit ihnen gearbeitet? Sichten auf eine Architektur sind notwendig, weil sich ganz unterschiedliche Interessengruppen mit einer Architektur befassen. Jedoch immer nur mit den für sie interessanten Teilen (Sichten) einer Architektur. Ist dies nicht möglich, wird eine Architektur nicht oder falsch verstanden. Deshalb muss eine bestimmte Architektur-Sicht eine bestimmte Abstraktion der Architektur eines Systems liefern, die von der betroffenen Interessengruppe auch verstanden werden kann. Beispielsweise sind für Business-Analysten die Anwendungsfälle von Interesse, nicht jedoch technische Details der Bausteine eines Systems. Deshalb werden sie mit der entsprechenden Architektur-Sicht arbeiten, die Anwendungsfälle umfasst. Dies würde so einfach jedoch nur unter der Voraussetzung funktionieren, dass bei der Erstellung respektive der Dokumentation (siehe Abschnitt 8.7) einer Architektur Architektur-Sichten von Anfang an explizit berücksichtigt wurden. Andernfalls müssten die Business-Analysten die Anwendungsfälle mühsam im „Dickicht“ der Architektur-Dokumentation suchen.

Relevanz von Architektur-Sichten

Architektur umfasst ganz unterschiedliche Aspekte eines Systems, wie z. B. Schnittstellen von Systembausteinen oder die Interaktionen von Systembausteinen. Im Rahmen der architektonischen Tätigkeiten (siehe Abschnitt 8.2) sind jedoch jeweils immer nur bestimmte dieser Aspekte relevant, und es wäre unhandlich, immer sämtliche dieser Aspekte gleichzeitig vor Augen haben zu müssen. Aus diesem Umstand ergibt sich eine weitere Notwendigkeit für Architektur-Sichten. Mit Architektur-Sichten bekommt man ein Mittel an die Hand, um sich zum geeigneten Zeitpunkt auf eine bestimmte Problemstellung zu fokussieren bzw. verschiedene Aspekte der Architektur eines Systems voneinander zu trennen. Daran zeigt es sich auch, wie Architektur-Sichten einem Architekten dabei helfen, mit der Komplexität im Zusammenhang mit Architektur umgehen zu können. Beispielsweise kann man sich im Rahmen der architektonischen Tätigkeit „Verstehen der Anforderungen“ (siehe Abschnitt 8.4) auf die architektonisch relevanten Anwendungsfälle

Architektur-Sichten und der Entwicklungsprozess

Architektur-Sichten werden nach einer bestimmten Reihenfolge entwickelt

fokussieren. Jedoch ohne dabei noch zusätzlich mit Sachverhalten der technischen Realisierung der Anwendungsfälle durch bestimmte Systembausteine konfrontiert zu werden. Der eben beschriebene Sachverhalt geht analog zur Architektur von Gebäuden. Auch dort sind verschiedene Architektur-Sichten in Gestalt von Plänen für Raumaufteilung, elektrische Verkabelung, Wasserleitungen etc. notwendig, um überhaupt eine überschaubare und verwendbare Architektur zu erhalten. Je nach Tätigkeit werden gezielt nur bestimmte, für die jeweils zuständigen Fachleute notwendige Pläne verwendet.

Architektur-Sichtenmodelle

Um mit Architektur-Sichten sinnvoll arbeiten zu können, sollten diese disjunkt sein. Die Reihenfolge, in der Architektur-Sichten erarbeitet werden, wird durch die Abhängigkeiten zwischen den Architektur-Sichten bestimmt. Idealerweise werden die Architektur-Sichten iterativ erarbeitet, das heißt, die Architektur-Sichten können evolutionär entstehen, es muss nicht zuerst eine Architektur-Sicht vollständig erarbeitet werden, bevor andere abhängige Architektur-Sichten erarbeitet werden können. Darüber hinaus ist es teilweise (wenn die Abhängigkeiten dies zulassen) möglich, sich einer Architektur-Sicht zu widmen, während an anderen Architektur-Sichten bereits weitergearbeitet wird. So lassen sich Architektur-Sichten zumindest teilweise parallel erstellen.

Wiederverwendung bewährter Standpunkte

Fasst man verschiedene Standpunkte zusammen, spricht man von einem Architektur-Sichtenmodell. In der Praxis existieren bereits verschiedene Architektur-Sichtenmodelle (z. B. das Architektur-Sichtenmodell aus dem 4+1-Sichtenmodell), die teilweise domänen spezifisch sind und jeweils verschiedene, für die Praxis sinnvoll aufeinander abgestimmte Architektur-Sichten spezifizieren. Darüber hinaus gibt es Architektur-Rahmenwerke (z. B. Reference Model for Open Distributed Processing (RM-ODP), das Zachman-Framework oder The Open Group Architecture Framework (TOGAF)), die Architektur-Sichtenmodelle umfassen und zusätzlich noch Standards, Best-Practices, Werkzeugunterstützung oder sogar Vorgehensmodelle für den Architekten. Häufig sind Architektur-Rahmenwerke für Enterprise-Architektur ausgelegt. Dieser Abschnitt fokussiert sich jedoch auf das Thema Architektur-Sichtenmodelle.

Wesentlicher Vorteil der Verwendung existierender standardisierter Architektur-Sichtenmodelle ist zum einen, dass man Standpunkte wieder verwendet, die sich vielfach in der Praxis bewährt haben. Und zum anderen, dass man als Architekt so von dem Aufwand befreit wird, ein eigenes Architektur-Sichtenmodell entwickeln zu müssen, nebst Spezifikation der Architektur-Sichten, Dokumentation, Richtlinien, Regeln, Werkzeugen etc.

Architektur-Sichtenmodellen umfassen alle relevanten Architektur-Sichten und ermöglichen es damit, Architektur überhaupt erst greifbar bzw. sichtbar zu machen. Eine Architektur muss ganz unterschiedliche Aspekte eines Systems abdecken. Beispielsweise auf der einen Seite die logischen Bausteine des Systems und auf der anderen Seite die physikalische Verteilung dieser Bausteine. Die eben genannten Aspekte sind Beispiele für Sichten auf eine Architektur. Die Komplexität der Architektur eines Systems kann erst durch Architektur-Sichtenmodelle bewältigt werden. Man erstellt und dokumentiert eine Architektur auf Basis der Architektur-Sichten eines Architektur-Sichtenmodells mit den in Abschnitt 6.6 vorgestellten Architektur-Modellierungsmitteln. Um für eine Architektur geplant verschiedene Sichten vorzusehen, ist es erforderlich, dass man einer Architektur (von Beginn an) ein Architektur-Sichtenmodell zugrunde legt.

Wesentlich für Architektur-Sichtenmodelle ist die Qualität der Spezifikationen für jede ihrer Architektur-Sichten. Deshalb sollten diese Spezifikationen folgende Punkte umfassen [Kruchten 2000]:

- > Kontext der Architektur-Sicht.
- > Im Fokus der Architektur-Sicht stehende Systembausteine und ihre Beziehungen.
- > Prinzipien zur Strukturierung einer Architektur-Sicht.
- > Beziehungen einer Architektur-Sicht zu anderen Architektur-Sichten.
- > Vorgehen zur Erzeugung einer Architektur-Sicht.

Eine gute Architektur zeichnet sich unter anderem dadurch aus, dass keine architekturellen Aspekte fehlen und trotzdem Komplexität reduziert wurde. Mithilfe eines Architektur-Sichtenmodells lässt sich nachprüfen, ob eine Architektur wie gewünscht alle relevanten Aspekte eines Systems abdeckt. Eine vollständige Architektur enthält demnach alle relevanten Architektur-Sichten und ist deshalb mehrdimensional. Wird eine Architektur gleich zu Beginn auf Basis eines Architektur-Sichtenmodells erstellt, verringert sich die Wahrscheinlichkeit, dass wichtige architekturelle Punkte vergessen und deshalb von der Architektur nicht gewürdigt werden.

Welche relevanten Architektur-Sichten sollte ein Architektur-Sichtenmodell spezifizieren? Es ist in Anlehnung an [Rozanski und Woods 2005], [Bredemeyer und Malan 2004] und [Larman 2002] zu empfehlen, zumindest folgende grundlegende Architektur-Sichten immer vorzusehen:

Architektur-Sichtenmodelle machen Architektur greifbar

Qualität der Spezifikationen von Architektur-Sichten ist wichtig

Architektur-Sichtenmodelle verbessern Architektur-Qualität

Grundlegenden Architektur-Sichten

- > *Konzeptionelle Sicht (Geschäftssicht)*: Diese Architektur-Sicht beschreibt die Systembausteine und ihre Beziehungen untereinander, ohne auf Details wie z. B. Schnittstellen einzugehen. Sie ist dazu geeignet, eine Architektur nicht-technischen Interessenvertretern zu vermitteln.
- > *Logische Sicht*: Diese Architektur-Sicht beschreibt die Systembausteine und ihre Beziehungen untereinander im Detail. Dabei werden die Systembausteine und ihre Beziehungen respektive die Kommunikationsmechanismen genau spezifiziert. Dies geschieht im Hinblick auf die technische Realisierung. Damit richtet sich diese Architektur-Sicht an technische Interessenvertreter.
- > *Ausführungssicht (Verteilungssicht)*: Diese Architektur-Sicht beschreibt im Detail die physikalische Verteilung der Systembausteine zur Laufzeit. Sie richtet sich ebenfalls an technische Interessenvertreter.

Grundlegende Architektur-Sichten sind nicht ausreichend

Die vorgestellten Architektur-Sichten zeigen jeweils statische und dynamische Strukturen eines Systems. Die grundlegenden Architektur-Sichten sind jedoch noch nicht ausreichend für ein „richtiges“ Architektur-Sichtenmodell, weil weitere Aspekte wie z. B. Anforderungen, Daten oder Entwicklungsumgebung nicht speziell berücksichtigt werden. Dies würde dazu führen, dass man weitere Aspekte jeweils einer der drei grundlegenden Architektur-Sichten zuordnen müsste mit negativen Folgen für die Kohärenz der Sichten (z. B. müsste man der logischen oder der Ausführungssicht den Aspekt Entwicklungsumgebung zuordnen.). Deshalb ergänzen und verfeinern die im Folgenden vorgestellten Architektur-Sichtenmodelle die grundlegenden Sichten.

Abstraktes Architektur-Sichtenmodell

Bevor auf die drei wichtigsten Vertreter in der Praxis genutzter Architektur-Sichtenmodelle näher eingegangen wird, wird zunächst ein abstraktes Architektur-Sichtenmodell vorgestellt, um den Umgang mit Sichtenmodellen zu erleichtern. Dieses Architektur-Sichtenmodell abstrahiert von den Sichten, der anschließend behandelten Architektur-Sichtenmodelle und umfasst Standpunkte, die den Namen, die Interessenvertreter und deren Interessen sowie die wesentlichen Artefakte für die verwendeten Architektur-Sichten spezifizieren. In den konkreten Architektur-Sichtenmodellen können die Architektur-Sichten teilweise anders heißen oder nochmals aufgeteilt bzw. zusammengefasst sein. Für die Tätigkeiten eines Architekten ist jedoch von zentraler Bedeutung, dass überhaupt Architektur-Sichten bzw. Architektur-Sichtenmodelle zum Einsatz kommen. Welches der Architektur-Sichtenmodelle letztlich verwendet wird, ist zweitrangig. Abbildung 4.2-2 zeigt das abstrakte Architektur-Sichtenmodell, welches in Anlehnung an die Architektur-Sichten aus [IEEE 2000], [Rozanski und Woods 2005] sowie [Kruchten 2000] entstanden ist.



Abb. 4.2-2: Abstraktes Architektur-Sichtenmodell.

In Tabelle 4.2-1 ist beispielhaft der Standpunkt der Anforderungssicht aufgeführt.

Anforderungssicht

Tab. 4.2-1: Standpunkt der Anforderungssicht.

Anforderungssicht	
Zweck	Dokumentation der Architektur-Anforderungen
Interessenvertreter	Architekten, Entwickler, Kunden, Management, Domänen-Experten, Tester, Projektleitung
Interesse(n)	Wie sieht der Geschäftskontext des Systems aus? Welches sind die essentiellen Anforderungen an das System?
Artefakte	Geschäftschancen und Problembeschreibung Interessenvertreter Geschäftsprozesse Anforderungen Richtlinien

Beispiele für Artefakte der Anforderungssicht sind die Tabellen 8.3-1, 8.3-2, 8.4-2 und 8.4-3.

In Tabelle 4.2-2 ist beispielhaft der Standpunkt der logischen Sicht aufgeführt.

Logische Sicht

Tab. 4.2-2: Standpunkt der logischen Sicht.

Logische Sicht	
Zweck	Dokumentation des Architektur-Entwurfs
Interessenvertreter	Architekten, Entwickler, Domänen-Experten
Interesse(n)	Welches sind die logischen Strukturen des Systems?
Artefakte	Architektur-Übersicht/-Vision Systemkontext Schlüsselabstraktionen (mit Verhalten) Fachliche Systembausteine Technische Systembausteine Richtlinien

Beispiele für Artefakte der logischen Sicht zeigen die Abbildungen 8.3-4, 8.5-4, 8.5-5 und 8.5-11 sowie Tabelle 8.5-2.

Datensicht

In Tabelle 4.2-3 ist beispielhaft der Standpunkt der Datensicht aufgeführt.

Tab. 4.2-3: Standpunkt der Datensicht.

Datensicht	
Zweck	Dokumentation von Aspekten bezüglich Speicherung, Manipulation, Verwaltung und Verteilung von Daten
Interessenvertreter	(Daten-)Architekten, Entwickler
Interesse(n)	Welches sind die Datenstrukturen und -flüsse des Systems?
Artefakte	Schlüsselabstraktionen (ohne Verhalten) Datenmodelle Datenflüsse Richtlinien

Ein Beispiel für ein Artefakt der Datensicht ist die Tabelle 8.5-6.

Umsetzungssicht

In Tabelle 4.2-4 ist beispielhaft der Standpunkt der Umsetzungssicht aufgeführt.

Tab. 4.2-4: Standpunkt der Umsetzungssicht.

Umsetzungssicht	
Zweck	Dokumentation der Umsetzungsstruktur und der Umsetzungsinfrastruktur
Interessenvertreter	Architekten, Entwickler, Konfigurationsmanager, Testmanager, Tester
Interesse(n)	Wie sehen Umsetzungsstruktur und Umsetzungsinfrastruktur des Systems aus?
Artefakte	Umsetzungsstruktur Richtlinien

Prozesssicht

In Tabelle 4.2-5 ist beispielhaft der Standpunkt der Prozesssicht aufgeführt.

Tab. 4.2-5: Standpunkt der Prozesssicht.

Prozesssicht	
Zweck	Dokumentation der Steuerung und Koordination nebenläufiger Bausteine
Interessenvertreter	Architekten, Entwickler

Prozesssicht	
Interesse(n)	Welches sind die nebenläufigen Bausteine eines Systems?
Artefakte	Prozesse und Threads Interprozess-Kommunikation Zustandsmodelle Richtlinien

In Tabelle 4.2-6 ist beispielhaft der Standpunkt der Verteilungssicht aufgeführt.

Verteilungssicht

Verteilungssicht	
Zweck	Dokumentation der physikalischen Verteilung von Software-Bausteinen
Interessenvertreter	Architekten, Entwickler, Betrieb
Interesse(n)	Wie sind die Software-Bausteine eines Systems auf Hardware-Bausteine verteilt und wie werden sie betrieben?
Artefakte	Installation und Konfiguration Netztopologie Netzprotokolle Betriebsumgebung Richtlinien

Ein Beispiel für ein Artefakt der Verteilungssicht zeigt die Abbildung 8.5-13.

Architektur-Sichten bauen nicht in einer bestimmten Reihenfolge sequentiell aufeinander auf. Vielmehr sind die Abhängigkeiten zwischen den Architektur-Sichten vielschichtig. Informationen einer Architektur-Sicht werden in anderen Architektur-Sichten genutzt. Zentral ist die Anforderungssicht weil Anforderungen Ausgangspunkt aller Tätigkeiten sind. Deshalb sind alle Architektur-Sichten unidirektional von der Anforderungssicht abhängig. Weitere nachvollziehbare unidirektionale Abhängigkeiten bestehen zwischen der Realisierungs- und der logischen bzw. der Datensicht sowie zwischen Verteilungs- und Umsetzungssicht.

Abhängigkeiten zwischen Architektur-Sichten sind nicht trivial

Im konkreten Fall können zusätzliche Architektur-Sichten notwendig und/oder bestimmt von einem Architektur-Sichtenmodell vorgegebene Architektur-Sichten nicht relevant sein. Das abstrakte Architektur-Sichtenmodell wie auch die im Anschluss vorgestellten konkreten Architektur-Sichtenmodelle können und sollten deshalb durch einen

Architektur-Sichtenmodelle können angepasst werden

Architekten bei Bedarf adaptiert werden. Beispielsweise um eine Sicherheits- oder Testsicht.

Wichtige Architektur-Sichtenmodelle im Überblick

Drei wichtige, in der Praxis häufig genutzte Architektur-Sichtenmodelle finden sich bei folgenden Architektur-Rahmenwerken:

- > Zachman-Framework.
- > Reference Model for Open Distributed Processing (RM-ODP).
- > 4+1-Sichtenmodell.

Tabelle 4.2-7 bildet die Architektur-Sichten des abstrakten Sichtenmodells auf die Architektur-Sichten dieser Architektur-Sichtenmodelle ab. Analoge Architektur-Sichten unterscheiden sich zwischen den Architektur-Sichtenmodellen teilweise etwas hinsichtlich ihrer Inhalte.

Tab. 4.2-7: Architektur-Sichten der wichtigsten Architektur-Sichtenmodelle.

Sichtenmodelle Architektur-Sicht (abstrakt)	Zachman-Framework	RM-ODP	4+1-Sichtenmodell
Anforderungssicht	Geschäfts- und Kontextsicht	Unternehmenssicht	Anwendungsfallsicht
Logische Sicht	Systemsicht	Systemsicht	Logische Sicht
Datensicht	Systemsicht	Informationssicht	Datensicht
Umsetzungssicht	Technologie-sicht	Technologie-sicht	Implementationssicht
Prozesssicht	Integrations- und Laufzeit-sicht	Konstruktions-sicht	Prozesssicht
Verteilungssicht	Integrations- und Laufzeit-sicht	Konstruktions-sicht	Verteilungs-sicht

4.2.1 Zachman-Framework

Vater der Architektur-Sichtenmodelle

Das Zachman-Framework [Zachman 1987] ist ein Architektur-Rahmenwerk, dessen Architektur-Sichtenmodell als einer der Väter der heute gängigen Architektur-Sichtenmodelle angesehen werden kann. Es wurde 1987 von John Zachman bei IBM entwickelt und war im Kern dazu gedacht, die Architektur von Organisationen (z. B. Unternehmen) zu beschreiben, ohne dabei zwingend auf IT einzugehen. Hierzu wird im

Zachman-Framework eine Organisation zunächst abstrakt beschrieben, um dann im weiteren Verlauf schrittweise die „Implementierung“ der Organisation zu zeigen. Aufgrund seines generischen Aufbaus hat sich das Zachman-Framework auch als geeignet erwiesen, um organisationsweite IT-Architekturen zu beschreiben.

In seiner aktuellen Ausbaustufe kennt das Zachman-Framework sechs allgemeine Architektur-Sichten und sechs zu den Architektur-Sichten orthogonal liegende Sichtenaspekte. Architektur-Sichten und Sichtenaspekte bilden in Form einer Matrix den Kern des Architektur-Sichtenmodells. Mit dieser Matrix sind dann bis zu 36 spezifische Architektur-Sichten möglich. Für eine konkrete Architektur sollte das Zachman-Framework, aufgrund seiner Eigenschaft als Referenzmodell, zunächst konfiguriert werden, indem eine Auswahl nach Relevanz unter den möglichen Architektur-Sichten stattfindet.

Als domänen- und technologieneutrales Architektur-Rahmenwerk kann das Zachman-Framework einer Architektur von grundsätzlich jeder Art von System zugrunde gelegt werden. Aufgrund seiner Ausrichtung auf organisationsweite Aspekte wird dieses Rahmenwerk idealerweise für Enterprise-Architekturen (siehe Abschnitt 3.2) verwendet. Für einfachere Architekturen muss Aufwand betrieben werden, um die Komplexität seines Architektur-Sichtenmodells zu reduzieren.

Auf folgenden Grundprinzipien baut das Architektur-Sichtenmodell des Zachman-Frameworks auf:

- > Systeme können komplett modelliert werden, indem die Antworten auf die Fragen warum? wer? was? wie? wo? wann? (Sichtenaspekte) beschrieben werden.
- > Sechs Architektur-Sichten umfassen sämtliche essenziellen Modelle für die Entwicklung eines Systems.
- > Übergeordnete Architektur-Sichten übernehmen Einschränkungen ihrer nachgeordneten Architektur-Sichten.
- > Spalten in der Matrix repräsentieren verschiedene Abstraktionen, um die Komplexität eines Modells zu verringern.
- > Spalten haben keine Reihenfolge.
- > Zeilen, Spalten und Zellen sind eindeutig.
- > Verschiedene Instanzen des Frameworks können sich rekursiv verwenden.

Abbildung 4.2-3 zeigt die sechs Architektur-Sichten und die zugehörigen sechs Sichtenaspekte des Zachman-Frameworks im Überblick.

Matrix aus Architektur-Sichten und Sichtenaspekten

Geeignet für Enterprise-Architekturen

Grundprinzipien



Abb. 4.2-3: Architektur-Sichten im Zachman-Framework.

Sichtenaspekte

Bevor die einzelnen Architektur-Sichten des Zachman-Frameworks näher erläutert werden, sollen zunächst die auf den Architektur-Sichten orthogonal liegenden Sichtenaspekte erläutert werden:

- > *Was (Daten)*: Beschreibt die Daten. Beispiele sind Geschäftsobjekte, Datenbanktabellen oder Felddefinitionen.
- > *Wie (Funktionen)*: Beschreibt die Funktionalität. Beispiele sind Geschäftsprozesse, Anwendungs- oder Rechnerfunktionalität.
- > *Wo (Netzwerk)*: Zeigt Knoten und ihre Beziehungen in einem Organisationsnetz. Beispiele sind verteilte Objekte, Speicheradressen oder Nachrichtenaustausch.
- > *Wer (Personen)*: Beschreibt die Personen in Bezug auf eine Organisation. Beispiele sind Interessensvertreter für funktionale Anforderungen, Rollen und Verantwortlichkeiten in Geschäftsprozessen oder Zugriffsrechte auf Systemfunktionalitäten.
- > *Wann (Zeit)*: Beschreibt performanz-relevante Zeit- oder Ereignisabhängigkeiten zwischen den Ressourcen einer Organisation. Beispiel ist die Zuteilung von Zeitfenstern für Geschäftsprozesse.
- > *Warum (Motivation)*: Beschreibt die Organisationsziele und ihre Gegenstände. Beispiel sind Geschäftspläne, Standards für Geschäftsprozesse oder Technologiestandards für Geschäftsregeln.

Architektur-Sichten

Es folgen Erläuterungen zu den einzelnen Architektur-Sichten im Zachman-Framework. Kontext- und Systemsicht haben einen geschäftlichen, die anderen Architektur-Sichten einen technischen Fokus:

- > *Kontextsicht (Planung, englisch: scope)*: Diese Architektur-Sicht beschäftigt sich mit den grundlegenden Anforderungen und dient als Basis für Abschätzungen bezüglich Kosten, Umfang und Funktionalität eines Systems.

- > *Geschäftssicht* (*Geschäftsmodellierung*, englisch: *business model*): Diese Architektur-Sicht zeigt alle geschäftlichen Entitäten und Prozesse auf.
- > *Systemsicht* (*Entwurf*, englisch: *system model*): Diese Schicht bestimmt die Daten und Funktionen, die das Geschäftsmodell realisieren. Dabei werden die Anforderungen im Detail definiert und es entstehen logische Modelle.
- > *Technologiesicht* (*Realisierung*, englisch: *technology model*): Diese Architektur-Sicht beschäftigt sich mit der technologischen Umsetzung eines Systems. Dabei geht es um Technologieauswahl und -Management, physische Modelle sowie die Realisierung mit konkreten Technologien.
- > *Integrationssicht* (*Verteilung*, englisch: *detailed representations*): In dieser Architektur-Sicht werden Verteilungsaspekte und das Konfigurationsmanagement eines Systems betrachtet.
- > *Laufzeitsicht* (*Verwendung*, englisch: *functioning enterprise*): Diese Architektur-Sicht beschäftigt sich mit dem Betrieb eines Systems innerhalb einer Organisation.

4.2.2 Reference Model for Open Distributed Processing

Das Reference Model for Open Distributed Processing (RM-ODP) ist ein Architektur-Rahmenwerk, ausgearbeitet von ISO (International Standards Organization,) und ITU (International Telecommunication Union). RM-ODP bringt ein Architektur-Sichtenmodell mit, das seit 1996 ein internationaler Standard für Architektur-Sichtenmodelle in Form eines generischen Referenzmodells [ISO10746 1998] ist. RM-ODP ist auf der einen Seite zugeschnitten auf verteilte objektbasierte Systeme, liefert jedoch auf der anderen Seite ein allgemeines auch für andere Systemarten nutzbares Architektur-Sichtenmodell. Dieses Modell entstand in einem lange Jahre andauernden Standardisierungsverfahren.

**Generisches
Referenzmodell**

Von RM-ODP können aufgrund seines generischen Sichtenmodells spezifische Architektur-Sichtenmodelle instanziert werden. RM-ODP wird dabei als Meta-Meta-Architektur-Sichtenmodell genutzt. Es findet z. B. im 4+1-Sichtenmodell des Unified Software Development Process (USDP) oder in der Object Management Architecture (OMA) der OMG seine Anwendung [Malveau und Mowbray 2001]. In Abbildung 4.2-4 werden die Architektur-Sichten gezeigt, die im RM-ODP definiert werden.

**Vorlage für spezifische
Architektur-
Sichtenmodelle**



Abb. 4.2-4: Architektur-Sichten im RM-ODP.

Architektur-Sichten

Die Architektur-Sichten des RM-ODP liefern jeweils für ihren Kontext das vollständige objektorientierte Modell eines Systems. Hauptziel von RM-ODP ist es, dass Architekturen weitgehend unabhängig sind von Verteilungs- und Implementierungsaspekten, um Systeme zu erreichen, welche die nicht-funktionalen Anforderungen (siehe Kapitel 5) optimal umsetzen. Im Folgenden werden die Architektur-Sichten des RM-ODP beschrieben:

- > *Unternehmenssicht* (*englisch: enterprise viewpoint*): Diese Architektur-Sicht geht auf Architektur aus dem Blickwinkel des Problembereichs ein. Kern ist hier das Geschäftsmodell aus Management- und Endbenutzersicht. Mit dieser Architektur-Sicht wird sichergestellt, dass die Anforderungen durch eine Architektur berücksichtigt werden.
- > *Informationssicht* (*englisch: information viewpoint*): Mit dieser Architektur-Sicht werden Struktur und Bedeutung der zu verarbeitenden Informationen sowie deren Verarbeitung beschrieben.
- > *Systemsicht* (*englisch: computational viewpoint*): In dieser Architektur-Sicht steht die Definition der Schnittstellen von verteilbaren Systembausteinen (Komponenten und Subsystemen) eines Systems im Mittelpunkt.
- > *Konstruktionssicht* (*englisch: engineering viewpoint*): Diese Architektur-Sicht geht auf die verteilten Interaktionen zwischen Systembausteinen zum Zwecke der Verarbeitung von Informationen und der Bereitstellung von Funktionalität ein.
- > *Technologiesicht* (*englisch: technology viewpoint*): Im Fokus dieser Architektur-Sicht steht die technologische Umsetzung der, durch die anderen Architektur-Sichten beschriebenen, Architektur.

4.2.3 4+1-Sichtenmodell

Das sogenannte 4+1-Sichtenmodell [Kruchten 2000] ist Ende der 90er-Jahre im Umfeld des USDP (Unified Software Development Process) entstanden. Abbildung 4.2-5 illustriert die in diesem Architektur-Sichtenmodell vorgesehenen Architektur-Sichten.



Abb. 4.2-5: Architektur-Sichten im 4+1-Sichtenmodell.

Anfänglich sah dieses Architektur-Sichtenmodell insgesamt 5 Architektur-Sichten vor. Später kam noch die Datensicht hinzu [Larman 2002]. Es blieb jedoch die Bezeichnung 4+1-Sichtenmodell. Im Folgenden werden die einzelnen Sichten des 4+1-Sichtenmodells beschrieben:

- > **Anwendungsfallsicht** (*englisch: use-case view*): Diese Architektur-Sicht ist zentral im 4+1-Sichtenmodell. Das 4+1-Sichtenmodell besagt, dass sämtliche architektonischen Entscheidungen auf den Anwendungsfällen des betroffenen Systems beruhen müssen. Diese Architektur-Sicht umfasst die wichtigsten Anwendungsfälle und dient als Basis und zur Validierung der anderen Architektur-Sichten.
- > **Logische Sicht** (*englisch: logical view*): In dieser Architektur-Sicht wird die Umsetzung der funktionalen Anforderungen betrachtet. Hier werden die wichtigsten Systembausteine (Subsysteme, Komponenten, Klassen etc.) und ihre Interaktionen behandelt.
- > **Implementierungssicht** (*englisch: implementation view*): Diese Architektur-Sicht behandelt die Organisation und Verwaltung der statischen Artefakte (Quelltext, Grafiken etc.) in Pakete, Schichten etc.
- > **Datensicht** (*englisch: data view*): In dieser Architektur-Sicht werden Datenmodelle beschrieben und die Abbildung (*englisch: mapping*) zwischen Systembausteinen und persistenten Daten betrachtet.
- > **Prozesssicht** (*englisch: process view*): Verhalten und Verteilung des Systems zur Laufzeit sind die Themen dieser Architektur-Sicht. Parallelverarbeitung und konkurrierende Zugriffe stehen dabei im Mittelpunkt.
- > **Verteilungssicht** (*englisch: deployment view*): Diese Architektur-Sicht beschreibt, wie die statischen Artefakte aus der Implementierungssicht physikalisch verteilt werden.

Architektur-Sichten

4.3 Zusammenfassung

Zusammenfassung: Architektur-Ebenen

- > Architektonisches Handeln findet auf verschiedenen Abstraktionsebenen statt. Als Architekt sollte man diese Abstraktionsebenen explizit berücksichtigen.
- > Ein Architekt wird sich dadurch der Kräfte bewusster, die auf eine Architektur einwirken und wird Problem- und Fragestellungen auf einheitlichere Weise behandeln und die Vermischung von unterschiedlichen Aspekten eher vermeiden.
- > Die Berücksichtigung von Abstraktionsebenen führt deshalb zu qualitativ hochwertigeren Architekturen.
- > Es werden die drei grundlegenden Abstraktionsebenen Organisations-, System- und Bausteinebene unterschieden.
- > Die genannten Abstraktionsebenen stehen in einer hierarchischen Reihenfolge zueinander. Aus Sicht eines Systems befindet sich die Organisationsebene auf der höchsten Abstraktionsstufe. Gefolgt von der System- und dann der Bausteinebene.
- > Vorgaben (Anforderungen und Entscheidungen) auf einer übergeordneten Architektur-Ebene werden auf der jeweils untergeordneten Architektur-Ebene berücksichtigt.
- > Die Elemente des architektonischen Ordnungsrahmens sind entsprechend des Abstraktionsniveaus einer der Architektur-Ebenen zugeordnet. Dies trägt zum systematischen und bewussten Umgang mit Architektur-Wissen bei.
- > Auf der Organisationsebene werden Organisationen deren Geschäftsprozesse und IT-Landschaften sowie deren Interaktionen mit anderen Organisationen betrachtet.
- > Auf der Systemebene werden die IT-Systeme von Organisationen betrachtet. Die einzelnen Systeme und deren Subsysteme werden jeweils als Black Box behandelt. Die Schnittstellen und die Interaktionen der Systeme mit ihrem Kontext stehen im Fokus
- > Auf der Bausteinebene werden die Bausteine der einzelnen Subsysteme, deren Verantwortlichkeiten, deren Schnittstellen sowie deren Interaktionen betrachtet.
- > Handelt es sich bei einem Systembaustein um ein Subsystem, findet ein Ebenenwechsel von der Bausteinebene zurück zur Systemebene statt.
- > Handelt es sich bei einem Systembaustein um einen nicht-tragenden Baustein, findet auf der Bausteinebene ein Wechsel von der Makro- zur Mikro-Architektur statt.

Zusammenfassung: Architektur-Sichten

- > Mittels Architektur-Sichten lassen sich gezielt bestimmte Aspekte eines IT-Systems betrachten. Damit verhelfen Architektur-Sichten dazu, mit der Komplexität aller Aspekte eines IT-Systems einfacher umzugehen.
- > Welche Aspekte betrachtet werden ist abhängig von dem jeweiligen Interessenvertreter und seiner aktuellen Tätigkeit bei der Erstellung eines IT-Systems.
- > Architektur-Sichten werden mittels Standpunkten (englisch: *view-point*) spezifiziert.
- > Die Spezifikation einer Architektur-Sicht umfasst Interessenvertreter und deren Interessen sowie Artefakte. Darüber hinaus kann auch die Erstellung einer Architektur-Sicht spezifiziert werden.
- > Als grundlegende Architektur-Sichten werden die Anforderungssicht, die logische Sicht, die Datensicht, die Umsetzungssicht, die Prozesssicht und die Verteilungssicht unterschieden.
- > Für die Architektur-Qualität ist es von Bedeutung, dass eine Architektur von Anfang an auf Basis von Architektur-Sichten erstellt respektive beschrieben wird.
- > Architektur-Sichten müssen nicht zwingend neu erstellt werden, sondern es können bereits existierende Architektur-Sichtenmodelle benutzt werden, die bereits alle, für die Praxis relevanten Sichten umfassen.
- > Zu den in der Praxis relevanten Architektur-Sichtenmodelle zählen das 4+1-Sichtenmodell sowie die Architektur-Sichtenmodelle der Architektur-Rahmenwerken Zachman-Framework und Reference Model for Open Distributed Processing (RM-ODP).

5 | Architektur-Anforderungen (WARUM)

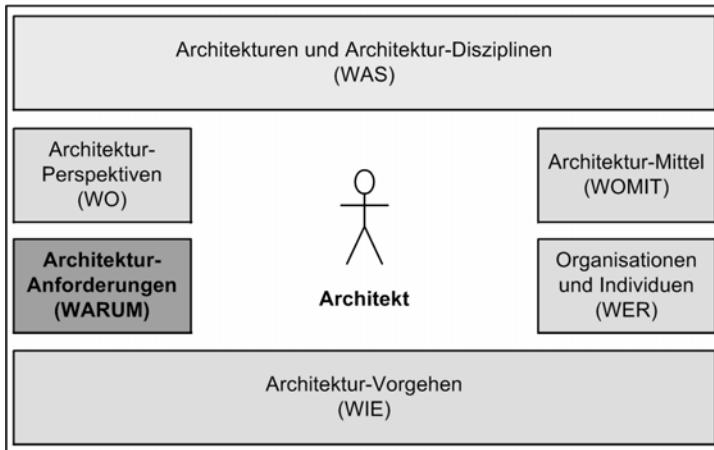


Abb. 5-1: Positionierung des Kapitels im Ordnungsrahmen.

Dieses Kapitel befasst sich mit der *WARUM-Dimension* des architektonischen Ordnungsrahmens. Zentrale Elemente der *WARUM-Dimension* sind Anforderungen. Sie umreißen das zu erstellende IT-System und begrenzen den gestalterischen Spielraum des Architekten. Anforderungen treten in unterschiedlichen Arten und auf verschiedenen Architektur-Ebenen auf. Damit ein Architekt seinen gestalterischen Spielraum nutzen kann, muss er die unterschiedlichen Ausprägungen von Anforderungen und ihre Beziehungen zueinander kennen. Dieses Kapitel gibt einen Überblick über die verschiedenen Anforderungsarten und deren Bezug zu den Architektur-Ebenen. Nach dem Lesen dieses Kapitels können Sie die wichtigsten Anforderungsarten nennen, deren Beziehungen verstehen und sie in den Kontext von Architektur setzen.

Übersicht

5.1	Allgemeines	104
5.2	Anforderungen im Überblick	107
5.3	Anforderungen im Detail	111
5.4	Anforderungen im Architektur-Kontext	119
5.5	Zusammenfassung	123

Grundlegende Konzepte der WARUM-Dimension

Abbildung 5-2 stellt die grundlegenden Konzepte, welche in diesem Kapitel behandelt werden, vor und visualisiert ihren Zusammenhang.

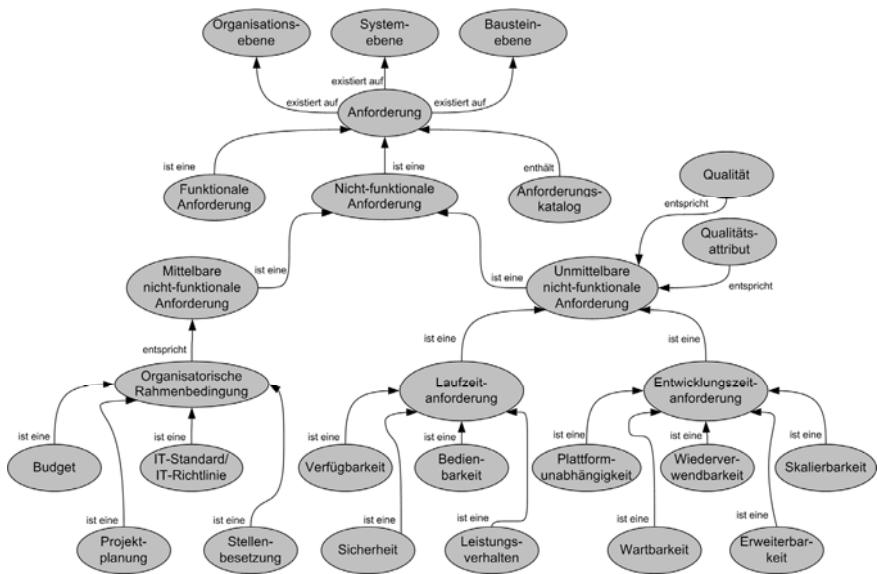


Abb. 5-2: Grundlegende Konzepte der WARUM-Dimension.

5.1 Allgemeines

Anforderungen als Motivatoren

Unterhält man sich mit einem Architekten über die Architektur und fragt ihn, warum er zu dieser architektonischen Lösung gekommen ist, erhält man in den meisten Fällen die Antwort: „Es bestand die Anforderung, dass ...“. Das heißt, die Architektur resultiert aus Anforderungen, die bereits zu Beginn bekannt sind oder die sich im Laufe der Zeit ergeben (siehe Abschnitt 3.1). Die Architektur entsteht also nicht willkürlich. Vielmehr stellen die verschiedenen Anforderungen ein Spannungsfeld dar, in dem das System und die zugehörige Architektur entstehen (siehe Abbildung 5.1-1).

Anforderungen sind Kräfte

Bildlich gesprochen sind Anforderungen Kräfte, die auf das System wirken und es formen. Diese Kräfte wirken in unterschiedliche Richtungen, zu unterschiedlichen Zeiten und in unterschiedlicher Stärke. Das System muss nun so gestaltet sein, dass es die gestellten Anforderungen erfüllen kann. Ähnlich wie bei einem Knochen. Bei einem Knochen bilden sich die feinen Knochenstrukturen entlang der Wirkungslinien der von außen wirkenden Kräfte aus und gleichen diese aus. Entsprechend legt die Architektur die Grundstruktur des Systems fest, um

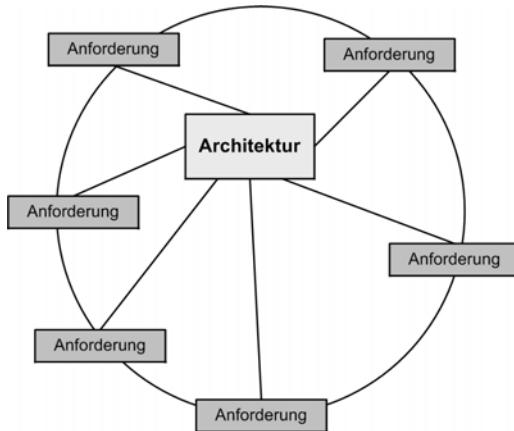


Abb. 5.1-1: Architektur im Spannungsfeld von Anforderungen.

die Kräfte der auf das System wirkenden Anforderungen auszugleichen. Weitere Informationen zu Kräften (englisch: *forces*) im Zusammenhang mit Architektur-Mustern finden sich in Abschnitt 6.3.

Eine Definition von Anforderungen stammt von Dorfmann und Thayer [Dorfmann und Thayer 1990]:

**Definition:
Anforderung**

Eine Anforderung ist

- > eine vom Anwender benötigte Fähigkeit (englisch: *capability*) des Systems, um ein Problem zu lösen oder ein Ziel zu erreichen, oder
- > eine Fähigkeit, die das System besitzen muss, damit es einen Vertrag, einen Standard, eine Spezifikation oder ein anderes formelles Dokument erfüllt.

Dies ist eine recht weit gefasste Definition einer Anforderung. So entspricht die Anforderung „Das System muss schnell sein“ durchaus der Definition, ist aber noch nicht präzise genug, um daraus eine Architektur abzuleiten. Aus diesem Grund müssen Anforderungen zusätzlich die nachfolgenden Eigenschaften erfüllen [Wiegert 2003]. Damit erhalten Anforderungen die notwendige Präzision und können als Grundlage für eine Architektur dienen.

Jede Anforderung muss korrekt sein. Die Korrektheit einer Anforderung kann jedoch nur ein Interessenvertreter beurteilen. Ein Interessenvertreter kann z. B. ein Benutzer, Sponsor oder Auftraggeber sein. Aus diesem Grund ist es wichtig, von Anfang an die diversen Interessenvertreter in die Ermittlung der Anforderungen einzubeziehen.

Korrekt

Machbar	Es muss möglich sein, die Anforderung unter den gegebenen Randbedingungen und mit den zur Verfügung stehenden Mitteln zu realisieren. Um dies sicherzustellen, sollte jemand mit technischem Verständnis (beispielsweise der Architekt) bei der Definition der Anforderungen mitwirken.
Eindeutig	Eine Anforderung muss so formuliert sein, dass der Leser nur einen Schluss aus dieser Anforderung ziehen kann. Eine einfache und klare Sprache ist somit sehr hilfreich bei der Formulierung der Anforderungen. Die Aussage, dass alle relevanten Kundendaten im Kundenverwaltungssystem verwaltet werden können, ist z. B. keine eindeutige Anforderung. Um sie zu präzisieren, müssten die relevanten Kundendaten klar spezifiziert werden.
Nachprüfbar	Die beste Anforderung nützt nichts, wenn sie nicht eindeutig überprüft werden kann. Es ist somit wichtig, schon bei der Formulierung der Anforderung zu überlegen, anhand welcher Tests eine Anforderung überprüft werden kann. Die Anforderung, dass in 90 % der Fälle die Einstiegsbildschirmmaske innerhalb von 5 Sekunden nach dem Aufruf einer Web Site erscheinen muss, ist eine nachprüfbare Anforderung. Diese kann z. B. mit einem entsprechenden Testwerkzeug (z. B. Apache JMeter), welches die Web Site aufruft und eine entsprechende Last auf dem System verursacht, überprüft werden.
Anforderungskatalog	Ein System definiert sich aber nicht nur über eine einzige Anforderung. Vielmehr ist es immer ein ganzer Katalog von Anforderungen. Ein Anforderungskatalog muss auch als Ganzes folgende Eigenschaften aufweisen:
Vollständig	Natürlich sollte der Anforderungskatalog möglichst vollständig sein, damit er ein abgerundetes Bild vom Gesamtsystem abgibt. Aber was sind die Kriterien für Vollständigkeit? Eine gute Möglichkeit ist, die Anforderungen durch einen Dritten auf Vollständigkeit überprüfen zu lassen. Oftmals sind nur die Standardabläufe bei der Anforderungsdefinition betrachtet worden. Fehlersituationen müssen die Anforderungen jedoch auch abdecken. Was passiert beispielsweise, wenn Informationen falsch angeliefert oder ein Auftrag aus Versehen freigegeben wurde? Diese möglichen Szenarien sollten auch durch Anforderungen abgedeckt werden [Cockburn 2000].
Konsistent	Die Menge aller Anforderungen müssen in sich stimmig sein und einzelne Anforderungen dürfen sich nicht gegenseitig widersprechen. Auch hier können nur die Interessenvertreter weiterhelfen, falls Widersprü-

che auftauchen und diese beseitigen. Es ist jedoch Aufgabe des Architekten, die an die Architektur gestellten Anforderungen kritisch zu hinterfragen und gegebenenfalls auf Inkonsistenzen hinzuweisen. So können sich beispielsweise ein gewünschtes hohes Leistungsverhalten eines Client-/Server-Systems (siehe Abschnitt 6.4.4) und eine geforderte, geringe Netzwerkbandbreite zwischen dem Klienten und dem Server widersprechen. Dies ist immer dann der Fall, wenn aufgrund der physikalischen Eigenschaften des Netzwerkes das geforderte Leistungsverhalten nicht erreichbar ist.

5.2 Anforderungen im Überblick

Um sich als Architekt mit Anforderungen zielgerichtet beschäftigen zu können, ist es wichtig, unterschiedliche Arten von Anforderungen unterscheiden zu können.

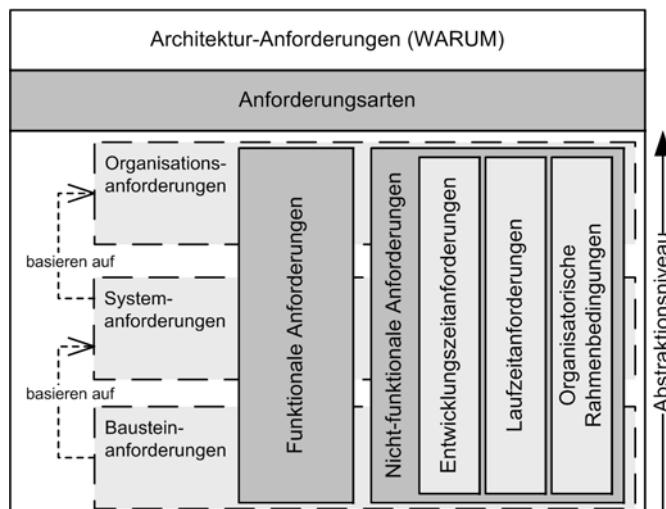


Abb. 5.2-1: Arten von Anforderungen.

Wie aus Abbildung 5.2-1 ersichtlich wird, können Anforderungen generell in funktionale und nicht-funktionale Anforderungen unterteilt werden. Diese können an Organisationen (*Organisationsanforderungen*), Systeme (*Systemanforderungen*) und Bausteine (*Bausteinanforderungen*) gestellt werden.

Funktionale Anforderungen definieren benötigte Funktionalitäten. Dabei können Organisationen, Systeme und Bausteine funktionale Anfor-

**Funktionale
Anforderungen**

derungen erfüllen. Tabelle 5.2-1 gibt einen Überblick über die verschiedenen funktionalen Anforderungsarten.

Tab. 5.2-1: Funktionale Anforderungsarten im Überblick.

Anforderungsart	Beschreibung
Funktionale Organisationsanforderungen	Funktionale Organisationsanforderungen verkörpern die funktionalen Anforderungen, die an Organisationen z. B. von deren Kunden, Mitarbeitern, Geschäftspartnern oder von Behörden gestellt werden. Der Wunsch von Kunden, Bestellungen bei der Organisation in Auftrag geben zu können, ist ein Beispiel für diese Anforderungsart. Ein weiteres Beispiel ist der Wunsch von Mitarbeitern, von der Organisation ihr Entgelt zu erhalten. Organisationen können auch Anforderungen an sich selbst stellen. Beispielsweise kann eine Organisation fordern, dass ein IT-System zur Erfassung von Aufträgen existieren muss, um die Auftragserfassung mittels IT zu unterstützen.
Funktionale Systemanforderungen	Funktionale Systemanforderungen drücken die konkreten funktionalen Bedürfnisse von Interessenvertretern bzw. von Systemen aus, die mit dem betrachteten System interagieren. Der Wunsch des Benutzers eines Systems, einen Auftrag in dem System erfassen zu können, ist ein Beispiel für diese Anforderungsart.
Funktionale Bausteinanforderungen	Funktionale Bausteinanforderungen verkörpern funktionale Eigenschaften, die ein Baustein eines Systems erfüllen muss, damit das System in der Lage ist, seine Anforderungen zu erfüllen. Der Wunsch eines Bausteins durch den Aufruf eines Dienstes eines anderen Bausteins, PDF-Dokumente erzeugen zu können, ist ein Beispiel für diese Anforderungsart (siehe Abschnitt 3.4).

Nicht-funktionale Anforderungen

Nicht-funktionale Anforderungen verkörpern Erwartungen und Notwendigkeiten, die von Interessenvertretern (Auftraggeber, Benutzer, Architekt, Entwickler etc.) neben den funktionalen Anforderungen als wichtig erachtet werden und über die reine gewünschte Funktionalität hinausgehen. Dabei können *unmittelbare* und *mittelbare* nicht-funktionale Anforderungen unterschieden werden.

Unmittelbare nicht-funktionale Anforderungen (Qualitäten)

Unmittelbare nicht-funktionale Anforderungen werden häufig auch als *Qualitäten* bzw. als *Qualitätsattribute* bezeichnet, da sie den qualitativen Charakter der durch Organisationen, IT-Systeme oder Bausteine erfüllten funktionalen Anforderungen widerspiegeln. Der Wunsch von Kunden, eine Bestellung innerhalb von 24 Stunden zu erhalten, kann z. B. als nicht-funktionale Anforderung an eine Organisation angesehen werden. Diese Anforderung entspricht einer gewünschten Qualität der von einer Organisation angebotenen Funktionalität *Bestell- bzw. Auftragsverarbeitung*. Im Hinblick auf IT-Systeme drücken nicht-funktionale

Anforderungen wie Leistungsverhalten, Erweiterbarkeit und Wiederverwendbarkeit Qualitäten aus.

Mittelbare nicht-funktionale Anforderungen wirken auf die Art und Weise der Realisierung der gewünschten Funktionalitäten und Qualitäten. Sie repräsentieren Vorgaben oder Gegebenheiten, die eingehalten respektive berücksichtigt werden müssen und somit den Realisierungsrahmen vorgeben. Daher spricht man in diesem Zusammenhang auch oftmals von Rahmenbedingungen. Das zur Realisierung eines IT-Systems zur Verfügung stehende Budget oder gesetzliche Bestimmungen sind Beispiele für Rahmenbedingungen, da das Budget den finanziellen Rahmen vorgibt, in dem das IT-System realisiert werden kann, und gesetzliche Bestimmungen nicht verletzt werden dürfen.

Die Erfüllung nicht-funktionaler Anforderungen ist essenziell für die Akzeptanz der Funktionalitäten einer Organisation, eines Systems oder eines Bausteins. Trotzdem werden nicht-funktionale Anforderungen oftmals unzureichend berücksichtigt, da der Fokus klar auf die funktionalen Anforderungen gelegt wird. Es ist die Aufgabe des Architekten, Interessenvertreter für dieses Gebiet zu sensibilisieren, da gerade die Architektur wesentlich für die Erfüllung der nicht-funktionalen Anforderungen ist. Durch dieses Bewusstmachen kann der Architekt erreichen, dass bereits in frühen Phasen Zeit für die Berücksichtigung nicht-funktionaler Anforderungen vorgesehen wird.

Neben der Unterscheidung zwischen Qualitäten und Rahmenbedingungen kann die Klasse der nicht-funktionalen Anforderungen weiter unterteilt werden. Tabelle 5.2-2 stellt die verschiedenen nicht-funktionalen Anforderungsarten im Überblick dar.

Tab. 5.2-2: Nicht-funktionale Anforderungsarten im Überblick.

Anforderungsart	Beschreibung
Entwicklungszeit-anforderungen	Entwicklungszeitanforderungen drücken Qualitäten und Rahmenbedingungen aus, die hauptsächlich bei der Entwicklung eines Systems berücksichtigt werden müssen. Hierzu gehören die klassischen Qualitätsattribute wie Erweiterbarkeit, Wiederverwendbarkeit oder Plattformunabhängigkeit. Darüber hinaus gehören Vorgaben hinsichtlich einzusetzender Technologien zu dieser Anforderungsart.
Laufzeit-anforderungen	Laufzeitanforderungen beinhalten Erwartungen hinsichtlich des Verhaltens eines Systems zur Laufzeit. Hierzu gehören Anforderungen wie Verfügbarkeit, Stabilität und Leistungsverhalten, die primär zur Laufzeit sichtbar werden.

Mittelbare nicht-funktionale Anforderungen (Rahmenbedingungen)

Relevanz nicht-funktionaler Anforderungen

Anforderungsart	Beschreibung
Organisatorische Rahmenbedingungen	Organisatorische Rahmenbedingungen beinhalten Vorgaben wie Budget und Time-to-Market. Ferner gehören hierzu auch Restriktionen hinsichtlich der Architektur-Gestaltung aufgrund von vorhandenem Wissen und Erfahrung innerhalb des Teams, welches mit der Realisierung eines Systems betraut ist.

Beziehungen zwischen Anforderungsarten

Baustein-, System- und Organisationsanforderungen basieren aufeinander (siehe Abbildung 5.2-1). Aus Organisationsanforderungen können Systemanforderungen und aus diesen wiederum Bausteinanforderungen abgeleitet werden. Allgemein kann man festhalten, dass Organisationsanforderungen, die mittels IT unterstützt werden sollen, eine vollständige Abdeckung durch Systemanforderungen finden sollten. Ebenso sollten Systemanforderungen entsprechend durch Bausteinanforderungen abgedeckt werden. Durch eine Verknüpfung der Anforderungen können somit zum einen die Nachvollziehbarkeit der Anforderungen und zum anderen die Vollständigkeit der Anforderungen geprüft werden.

Unterscheidung zwischen Entwicklungszeit- und Laufzeitanforderungen

Nicht-funktionale Anforderungen lassen sich nicht immer eindeutig der Entwicklungszeit oder der Laufzeit zu ordnen. Es ist wichtig, sich zu vergegenwärtigen, wann die konkrete nicht-funktionale Anforderung schwerpunktmäßig wirkt und berücksichtigt werden muss. Wenn beispielsweise beim Entwurf einer Architektur zur Entwicklungszeit die nicht-funktionale Anforderung nach Erweiterbarkeit berücksichtigt wurde, kann das System zur Laufzeit um neue Funktionalität erweitert werden. Somit ist Erweiterbarkeit sowohl zur Entwicklungszeit als auch zur Laufzeit sichtbar. Sie muss jedoch primär zu der Entwicklungszeit berücksichtigt werden.

Detaillierungsgrad von Anforderungen

Anforderungen können unterschiedlich detailliert beschrieben werden. So kann z. B. der undetaillierte Wunsch nach einem System zur Erfassung von Aufträgen durch eine Organisationsanforderung „Es muss ein System existieren, mit dem Mitarbeiter Aufträge erfassen können“ formuliert werden. Darauf aufbauend kann eine Systemanforderung in Form eines Systemanwendungsfalles beschrieben werden, der detailliert dokumentiert, wie Benutzer in einem Auftragserfassungssystem Aufträge erfassen möchten. Die Detaillierung im Hinblick auf ein zu realisierendes System nimmt daher von den Organisationsanforderungen hin zu den Bausteinanforderungen zu.

Anforderungen und ihre Wechselwirkungen

Anforderungen haben nicht nur Auswirkungen auf die Architektur, sondern treten auch untereinander in Wechselwirkung. Aufgabe der

Architektur ist es, diese Wechselwirkungen so gut wie möglich auszugleichen. So können sich das vorhandene Wissen eines Projektteams und das geforderte Leistungsverhalten (siehe Abschnitt 5.3.4) des Systems gegenseitig beeinflussen. Ein hohes Leistungsverhalten legt zum Beispiel eine nebenläufige Architektur nahe. Nun kann es vorkommen, dass die an dem Entwicklungsprojekt beteiligten Personen jedoch keine Erfahrung im Umgang mit nebenläufiger Programmierung besitzen. Der Architekt hat nun verschiedene Möglichkeiten, wie er mit der Wechselwirkung dieser beiden Anforderungen umgeht. Eine Möglichkeit wäre die Integration zusätzlicher Projektmitarbeiter mit dem geforderten Profil. Eine zweite Alternative ist der Einsatz von Architektur-Mitteln, wie Anwendungsserver, die den Einsatz einer nebenläufigen Architektur erleichtern. Es muss entschieden werden, welche der möglichen Alternativen die Wechselwirkungen zwischen den einzelnen Anforderungen am besten ausgleicht.

5.3 Anforderungen im Detail

Das in Abschnitt 5.2 vorgestellte Klassifizierungsschema erlaubt es, einzelne Anforderungen einzuordnen und zueinander in Beziehung zu setzen. Es erleichtert dem Architekten am Anfang seiner Arbeit, den Überblick zu erhalten. In diesem Abschnitt werden zunächst Organisationsanforderungen, Systemanforderungen und Bausteinanforderungen weiter beleuchtet. Im Anschluss wird auf die nicht-funktionalen Anforderungstypen nochmals näher eingegangen. Hierbei wird das besondere Augenmerk auf Systeme gelegt. In diesem Zusammenhang werden einige Anforderungen detaillierter besprochen, die in der Praxis häufig anzutreffen sind. Dabei wird jedoch kein Anspruch auf Vollständigkeit gelegt.

5.3.1 Organisationsanforderungen

Organisationsanforderungen verkörpern Anforderungen, die an Organisationen gestellt werden. Sie lassen sich auf die Umwelt der Organisation zurückführen (siehe Abschnitt 7.2). Die Umwelt stellt funktionale und nicht-funktionale Anforderungen, die von der Organisation erfüllt werden müssen.

Funktionale Organisationsanforderungen beziehen sich auf Dienstleistungen, die die Organisation anbietet. Der Wunsch von Kunden, Bestel-

Anforderungen an die Organisation

Funktionale Organisationsanforderungen

lungen bei der Organisation in Auftrag geben zu können, ist ein Beispiel für eine funktionale Organisationsanforderung.

Nicht-funktionale Organisations-anforderungen

Nicht-funktionale Organisationsanforderungen formulieren den Qualitätsanspruch, den die Umwelt an die Erbringung der funktionalen Organisationsanforderungen stellt. Ein Lieferung innerhalb von 24 Stunden oder eine zweijährige Garantiezeit sind Beispiele für nicht-funktionale Organisationsanforderungen.

Organisation definiert Organisations-anforderungen

Zur Erfüllung der an sie gestellten Anforderungen kann sich eine Organisation entschließen, IT-Systeme einzusetzen. Sie kann somit eigene Organisationsanforderungen formulieren, die den Bedarf an zu entwickelnde IT-Systeme widerspiegeln. Eine Organisation stellt beispielsweise PCs her. Die einzelnen PCs werden immer genau auf die Wünsche des einzelnen Kunden abgestimmt. Die Mitarbeiter in der Abteilung Auftragseingang werden ständig von einer Flut neuer Aufträge überschüttet. Diese Flut ist nur mit einem extrem hohen Arbeitseinsatz zu bändigen. Der hohe Arbeitseinsatz und die damit verbundene Belastung der Mitarbeiter ist als Problem identifiziert worden. Im Rahmen der Problemanalyse wird deutlich, dass der hohe Arbeitseinsatz auf eine Vielzahl immer wiederkehrender, manueller Arbeitsschritte zurückzuführen ist. Um die Mitarbeiter des Auftragseingangs bei der Erfüllung ihrer Aufgaben zu unterstützen, entschließt sich die Organisation, ein IT-System zur Lösung dieses Problems einzusetzen. Das System soll die Mitarbeiter entlasten, indem es möglichst viele der manuellen Arbeitsschritte automatisiert. Die Organisation formuliert somit eine Organisationsanforderung nach einem IT-System zur Auftragserfassung.

IT-Standards und -Richtlinien

IT-Standards und -Richtlinien sind organisationsweite Vorgaben, denen zu entwickelnde IT-Systeme innerhalb der Organisation genügen müssen. Diese Vorgaben können sowohl funktionalen als auch nicht-funktionalen Charakter besitzen. Der zwingende Einsatz von JEE als Komponentenplattform ist ein Beispiel für eine nicht-funktionale Organisationsanforderung, die als IT-Standard respektive -Richtlinie festgeschrieben werden kann. Die Grundsätze ordnungsgemäßer Buchführung sehen beispielsweise vor, dass Buchauszüge erstellt werden können. Die Möglichkeit der Erstellung von Buchauszügen ist ein Beispiel für eine funktionale Organisationsanforderung, die ebenfalls als organisationsweite Richtlinie angesehen werden kann. In IT-Systemen muss hierzu entsprechende Funktionalität vorgesehen werden.

5.3.2 Systemanforderungen

Systemanforderungen beschreiben die Anforderungen, die an Systeme gestellt werden. Die innere Struktur des neuen Systems bleibt bei der Betrachtung der Systemanforderungen außen vor.

Funktionale Systemanforderungen basieren auf funktionalen Organisationsanforderungen. Im Fokus stehen die Funktionalitäten, die ein Benutzer bzw. ein anderes System von dem System erwarten. Auf das Beispiel des neuen Systems zur Auftragserfassung angewandt könnte eine funktionale Systemanforderung, die mittels eines Systemanwendungsfalles dokumentiert wurde, z. B. lauten „Erfasse einen neuen Auftrag“. Diese Systemanforderung beschreibt die einzelnen Tätigkeiten, die der Benutzer mit dem zu entwickelnden System ausübt. Funktionale Systemanforderungen manifestieren sich in konkreten Bausteinen eines Systems. Für das Auftragserfassungsbeispiel bedeutet dies, dass in dem Auftragserfassungssystem Bausteine, wie Benutzerdialogsteuerung, Auftragsverarbeitung und Persistenz, existieren, die die Erfassung eines Auftrags ermöglichen. Eine funktionale Anforderung schlägt sich also immer im konkreten funktionalen Verhalten des Systems nieder. Ebenso kann von dem Auftragserfassungssystem gefordert werden,

- > dass neue Aufträge automatisch an das Montageplanungssystem weitergegeben werden
- > und dass aus dem Buchhaltungssystem die neusten Informationen zu den momentan offenen Forderungen bezogen werden.

Funktionale Systemanforderungen

Typische nicht-funktionale Systemanforderungen sind Leistungsverhalten, Verfügbarkeit, Erweiterbarkeit und Plattformunabhängigkeit. Die Erfüllung nicht-funktionaler Anforderungen ist ein weiteres wichtiges Kriterium für die Akzeptanz eines zu entwickelnden Systems (siehe Abschnitt 2.5). Nicht-funktionale Anforderungen lassen sich schlecht oder gar nicht in einem System lokalisieren. So gibt es beispielsweise keinen Systembaustein, der für Erweiterbarkeit verantwortlich ist. Erweiterbarkeit ergibt sich viel mehr aus einer Reihe von Prinzipien, wie beispielsweise Kapselung, die an einer Vielzahl von Stellen ins System einfließen (siehe Abschnitt 6.1). Um der Anforderung nach Erweiterbarkeit Rechnung zu tragen, können z. B. dedizierte Erweiterungspunkte im System vorhanden sein, die in der Architektur vorgesehen werden müssen. In unserem Auftragserfassungsszenario könnte vom Auftragserfassungssystem beispielsweise gefordert sein, dass in einem nächsten Release Kunden über das Internet den Status ihrer

Nicht-funktionale Systemanforderungen

Bestellung abrufen können. Diese mögliche Erweiterung muss bereits in der Software-Architektur des Systems vorgesehen werden. Gerade die nicht-funktionalen Systemanforderungen werden oft vergessen respektive vernachlässigt, da für den Auftraggeber und die Benutzer die funktionalen Systemanforderungen im Fokus stehen. Der Auftraggeber ist daher oft nicht gewillt, einen höheren Aufwand für den notwendigen Architektur-Entwurf zu tragen. Der Architekt ist an dieser Stelle gefordert. Er muss erkennen, welche nicht-funktionalen Anforderungen explizit berücksichtigt werden sollten. Die Architektur muss die nicht-funktionalen Anforderungen umsetzen. Dies geschieht nicht nebenbei, sondern muss von Anfang an in den Architektur-Entwurf einfließen.

5.3.3 Bausteinanforderungen

Bausteinanforderungen definieren die funktionalen und nicht-funktionalen Anforderungen an die Bausteine eines Systems.

Funktionale Baustein-anforderungen

Funktionale Bausteinanforderungen definieren dabei die von einem Baustein erwarteten Funktionalitäten. Eine mögliche funktionale Bausteinanforderung an einen fiktiven Datenzugriffsbaustein des Auftragserfassungssystems könnte beispielsweise lauten: Der Datenzugriffsbaustein erlaubt das Suchen nach Kundenobjekten mittels Kundennummer und Kundenname.

Nicht-funktionale Bau-steinanforderungen

Ebenso können generelle nicht-funktionale Anforderungen an Bausteine formuliert werden. An den Datenzugriffsbaustein könnte z. B. folgende Entwicklungszeitanforderung gestellt werden: Es muss Bausteinen aus der Geschäftslogikschicht eine Schnittstelle gemäß dem Data-Access-Object-Muster [Alur et al. 2003] angeboten werden. Deshalb hat der Datenzugriffsbaustein das Data-Access-Object-Muster umzusetzen. Diese Bausteinanforderung kann auf eine Systemanforderung nach einer Unabhängigkeit von dem eingesetzten Datenbankmanagementsystem zurückgeführt werden.

5.3.4 Laufzeitanforderungen

Qualitäten zur Laufzeit

Laufzeitanforderungen sind nicht-funktionale Anforderungen, die an das zu erstellende System zur Laufzeit gestellt werden. Damit haben sie eine besondere Bedeutung für das System. Sie beschreiben Qualitäten, die die Akzeptanz des Systems beim Auftraggeber oder Benutzer beeinflus-

sen. Typische Laufzeitanforderungen sind Verfügbarkeit, Leistungsverhalten, Bedienbarkeit und Sicherheit.

Verfügbarkeit drückt sich in dem Verhältnis der Fehlzeiten zu den Produktivzeiten aus. Je kleiner die Fehlzeiten gegenüber den Produktivzeiten, desto höher die Verfügbarkeit des Systems. Die Architektur hat zwei Möglichkeiten, eine möglichst gute Verfügbarkeit zu erreichen. Sie kann versuchen, die Fehlzeiten an sich zu minimieren, sodass möglichst wenige auftreten. Die zweite Möglichkeit betrifft die Fehlzeiten an sich. Steht ein System aufgrund eines Fehlers nicht zur Verfügung, so sollte die Architektur es ermöglichen, dass die Fehlerursache möglichst schnell lokalisiert und behoben werden kann. Auf diese Weise trägt die Architektur zu einer Verkürzung der Fehlzeit bei.

Verfügbarkeit

Ein System reagiert immer auf äußere Ereignisse. Das Leistungsverhalten beschreibt das Leistungsvermögen respektive die Performanz des Systems bei der Reaktion auf diese äußeren Ereignisse. Es gibt zwei Möglichkeiten, um das Leistungsverhalten eines Systems auszudrücken. Zum einen kann das Leistungsverhalten anhand der Anzahl der Ereignisse, die ein System in einem bestimmten Zeitraum bearbeiten kann, gemessen werden. Die zweite Möglichkeit misst die durchschnittliche Zeitspanne, die das System für die Bearbeitung eines Ereignisses braucht. Das Leistungsverhalten eines Systems wird wesentlich durch die Kommunikation an seinen internen und externen Schnittstellen bestimmt. Damit trägt die Architektur durch das Festlegen der Schnittstellen zwischen den einzelnen funktionalen Bausteinen wesentlich zum Leistungsverhalten eines Systems bei.

Leistungsverhalten (Performanz)

Die Bedienbarkeit (englisch: *usability*) eines Systems drückt sich in erster Linie in der Gestaltung der Benutzeroberfläche aus und steht nicht im unmittelbaren Zusammenhang mit der Software-Architektur eines Systems. Um eine adäquate Bedienbarkeit eines Systems zu erreichen, muss eine passende Architektur ausgewählt werden (siehe Abschnitt 6.4). Eine wesentliche Architektur-Entscheidung ist hierbei die Wahl zwischen einer Rich- oder Thin-Client-Architektur (siehe Abschnitt 6.4.6). Um darüber hinaus ein barrierefreies Arbeiten zu ermöglichen, kann von der Architektur gefordert sein, unterschiedliche Benutzerschnittstellen (z. B. eine sprachgesteuerte Benutzerschnittstelle) zu unterstützen. Des Weiteren kann die Anforderung existieren, dass Benutzer mit dem System in einem Offline-Modus arbeiten können. Dies trifft sehr oft für Außendienstmitarbeiter zu, die Aufträge auf ihrem Laptop erfassen können und später mit einem Auftragserfassungsserver synchronisieren können. Diese Bedienbarkeitsanforderung muss in der

Bedienbarkeit

Architektur explizit vorgesehen werden. Ebenso kann eine adäquate Architektur für die Präsentationslogik eines Systems die Entwicklung einer Benutzerschnittstelle hinsichtlich Erweiterbarkeit, Wiederverwendbarkeit und Konsistenz erleichtern. Ebenso kann die Architektur die grundlegenden Mechanismen der Fehlerbehandlung eines Systems bereits vorsehen. Dies kommt ebenfalls dem visuellen Teil der Fehlerbehandlung zugute.

Sicherheit

Sicherheit ist eine nicht-funktionale Anforderung mit durchdringendem Charakter. Sie erwartet beispielsweise, dass ein System unautorisierte Zugriffe verweigert und authentifizierten Benutzern Zugriff auf Systemressourcen gewährt, für die sie eine entsprechende Berechtigung besitzen. Vertraulichkeit (englisch: *confidentiality*), Authentifizierung (englisch: *authentication*), Integrität (englisch: *integrity*), Privatsphäre (englisch: *privacy*), Unleugbarkeit (englisch: *non-repudiation*) sowie Schutz vor Zerstörung und Schutz des Betriebs (englisch: *intrusion protection*) sind Aspekte, die in diesem Zusammenhang zu nennen sind. Sicherheit ist ein sehr wichtiges Themengebiet. Sicherheitsarchitektur ist eine eigene Architektur-Disziplin (siehe Abschnitt 3.2), für die verschiedene Basisarchitekturen (siehe Abschnitt 6.4.12) existieren.

5.3.5 Entwicklungszeitanforderungen

Zur Entwicklungszeit relevante Anforderungen

Entwicklungszeitanforderungen beziehen sich auf die einzusetzenden Architektur-Mittel. Zum einen gehören hierzu die in dem IT-System eingesetzten Mittel. Zum anderen bestimmen sie aber auch die Mittel, die zur Herstellung des IT-Systems verwendet werden. Damit wirken sich diese Anforderungen vor allen Dingen während der Entwicklung des Systems aus. Beispiele für Entwicklungszeitanforderungen sind Plattformunabhängigkeit, Wiederverwendbarkeit, Skalierbarkeit und Wartbarkeit. Ebenso zählt die Vorgabe bestimmter Technologien (z. B. JEE oder .NET) zu dieser Anforderungsart.

Plattform-unabhängigkeit

Es wird häufig gefordert, dass ein System auf unterschiedlichen Plattformen (siehe Abschnitt 3.4) betrieben werden kann. So kann es z. B. möglich sein, ein System auf unterschiedlichen JEE-Komponentenplattformen (z. B. IBM WebSphere Application Server und JBoss Application Server) zu installieren. Aufgabe der Architektur ist es, durch den Einsatz entsprechender Architektur-Mittel (siehe Kapitel 6) diese unterschiedlichen Kombinationen zu ermöglichen. Durch den Einsatz des Prinzips der Modularisierung können beispielsweise die plattformspezifischen von den nicht plattformspezifischen Systembausteinen getrennt

werden. Plattformunabhängigkeit wird auch häufig als Portierbarkeit bezeichnet.

Der sinnvolle Einsatz von Architektur-Mitteln kann nicht nur zur Erreichung einer Plattformunabhängigkeit eingesetzt werden. Vielmehr können Architektur-Mittel auch dazu beitragen, bereits entwickelte Bausteine in späteren Entwicklungen oder in anderen Systemen erneut einzusetzen, um so den Entwicklungsaufwand zu reduzieren. Mit diesem Sachverhalt befasst sich die nicht-funktionale Anforderung der Wiederverwendbarkeit, indem sie zum einen festlegen kann, dass Bausteine wiederverwendbar zu entwerfen sind und zum anderen, dass existierende Bausteine einzusetzen sind. Der Themenkomplex Software-Wiederverwendung wird ausführlich in [Chugtai und Vogel 2001] behandelt.

Wiederverwendbarkeit

Systeme müssen mit steigenden Lasten umgehen können. Mit anderen Worten müssen sie bei zunehmender Last adäquat reagieren, um ihre Dienste in einer definierten Güte anbieten zu können. Beispielsweise darf ein Nachrichtendienstes nicht zusammenbrechen, wenn die Anzahl der Anfragen aufgrund einer wichtigen Neuigkeit stark zunimmt. Man unterscheidet in der Regel *vertikale* und *horizontale* Skalierbarkeit. Bei ersterer wird z. B. ein Server durch einen leistungsfähigeren Server ausgetauscht. Bei letzterer wird die Last auf mehrere Server verteilt.

Skalierbarkeit

Der Lebenszyklus eines Systems geht über die anfängliche Entwicklung hinaus. Ist ein System in Betrieb genommen worden, werden einerseits zu korrigierende Fehler identifiziert und zum anderen werden sich neue Anforderungen ergeben, die mit dem momentanen Stand des Systems nicht abgedeckt werden können. Entsprechend müssen Systemfehler behoben und neue Anforderungen umgesetzt werden. Die nicht-funktionale Anforderung der *Wartbarkeit* befasst sich primär mit dem Beheben von Fehlern während sich die *Erweiterbarkeit* hauptsächlich mit der Umsetzung neuer Anforderungen und dem Austausch von Systembausteinen beschäftigt. Je einfacher Fehler behoben werden können, umso leichter kann das System gewartet werden. Ein leicht wartbares System zeichnet sich durch Systembausteine mit hoher Kohäsion aus. Die Erweiterbarkeit eines Systems ist umso besser, je geringer die Kopplung zwischen Systembausteinen ist. Um eine gute Wartbarkeit und eine gute Erweiterbarkeit zu erreichen, sollten also die Prinzipien der hohen Kohäsion (siehe Abschnitt 6.1.2) und der losen Kopplung (siehe Abschnitt 6.1.1) berücksichtigt werden.

Wartbarkeit und Erweiterbarkeit

5.3.6 Organisatorische Rahmenbedingungen

Organisatorische Rahmenbedingungen beeinflussen die Architektur

Organisatorische Rahmenbedingungen fallen auf den ersten Blick meist in den Verantwortungsbereich des Projektleiters, da es sich hierbei um Themen wie Budget, Zeitpläne und organisatorische Strukturen handelt. Sie haben jedoch auch Einfluss auf die Architektur. Beachtet die Architektur nicht die Einflüsse der organisatorischen Anforderungen, können diese bewirken, dass eine Architektur gar nicht umgesetzt wird (siehe Kapitel 7).

Stellenbesetzung

Die Architektur steht immer unter dem Einfluss der Fähigkeiten und Kenntnisse der Projektmitglieder. Sie haben Erfahrungen und Kenntnisse mit bestimmten Technologien und Vorgehensweisen. Trägt eine Architektur diesem Erfahrungsschatz nicht Rechnung, kann dies nur mit zusätzlichem Aufwand ausgeglichen werden. Der zusätzliche Aufwand entsteht entweder, indem neue Entwickler in das Projektteam geholt werden oder aber die bestehenden Entwickler in der neuen Technologie geschult werden. Aus diesem Grund ist es wichtig für den Architekten, zu wissen, welche Technologien im Projektteam bisher eingesetzt wurden und wie der Entwicklungsprozess aussieht. Mit diesem Wissen kann die Architektur an diesen organisatorischen Rahmenbedingungen ausgerichtet werden.

Projektplanung

Oft sind Entwicklungsprojekte mit engen Zeitvorgaben konfrontiert. In solchen Fällen lassen sich die Zeitvorgaben nur durch die Verwendung von Halb- oder Fertigprodukten halten. In diesem Fall ist es Aufgabe der Architektur, diese Halb- oder Fertigprodukte in das Gesamtsystem zu integrieren. Damit beeinflussen diese vorgefertigten Bausteine durch ihre Funktionalitäten und Schnittstellen die Aufteilung des Gesamtsystems in einzelne Bausteine.

Budget

Jedes Entwicklungsprojekt hat ein Budget. Dieses Budget beeinflusst natürlich auch die Architektur. Jede Technologie ist mit verschiedenen Kosten verbunden. Seien es Anschaffungskosten oder aber auch Aufwendungen für Ausbildung im Umgang mit einer neuen Technologie. Die Architektur muss dem vorhandenen Budget Rechnung tragen, indem sich die Technologieauswahl und die Umsetzung der Anforderungen an dem zur Verfügung stehenden Budget ausrichten.

5.4 Anforderungen im Architektur-Kontext

In diesem Abschnitt werden die zuvor eingeführten Anforderungsarten im Kontext von Architektur betrachtet. Hierzu werden die Anforderungsarten mit den anderen Dimensionen des architektonischen Ordnungsrahmens in Beziehung gesetzt. Abbildung 5.4-1 visualisiert diesen Architektur-Kontext. Als zentrales Strukturierungsmerkmal dienen die verschiedenen Architektur-Ebenen aus Kapitel 4, indem die wichtigen Elemente der Architektur-Dimensionen auf den verschiedenen Architektur-Ebenen positioniert werden. Hierdurch können die Elemente auf einem einheitlichen Abstraktionsniveau (Abstraktionsebene) betrachtet werden. Das Abstraktionsniveau sinkt dabei von der Organisationsebene hin zur Bausteinebene. Mit anderen Worten nimmt der Detaillierungsgrad im Hinblick auf ein zu realisierendes System respektive dessen Architektur von oben nach unten zu. Elemente auf der Organisationsebene besitzen organisationsweiten Charakter. Elemente auf der Systemebene beziehen sich dagegen direkt auf ein System und Elemente auf der Bausteinebene entsprechend auf Bausteine eines Systems (siehe Kapitel 4). Abbildung 5.4-1 illustriert die Anforderungsarten im Architektur-Kontext.

Architekturrelevante Anforderungen sind ganz allgemein all die Anforderungen, die einen wesentlichen Einfluss auf den Entwurf der Architektur besitzen. Unglücklicherweise lässt sich jedoch nicht pauschal festlegen, welche dies nun konkret sind. Dies hängt immer vom jeweiligen Einzelfall ab. Der Nutzen, das Risiko und die Wirkung einer Anforderung können als Identifizierungsmerkmale für architekturrelevante Anforderungen dienen. Näheres findet sich zu diesem Thema in Abschnitt 8.4.

Anforderungen können auf unterschiedlichen Architektur-Ebenen angesiedelt werden. Anforderungen auf einer Architektur-Ebene basieren dabei auf Anforderungen auf der darüber liegenden Architektur-Ebene.

Organisationsanforderungen auf der Organisationsebene drücken Anforderungen aus, die von Kunden, Geschäftspartnern, Mitarbeitern oder von Behörden an die Organisation gestellt werden. Aus diesen Anforderungen kann sich die Notwendigkeit nach Systemen zur Unterstützung bzw. zur Erfüllung der Anforderungen auf der Organisationsebene ergeben. Ferner sind auf dieser Ebene auch organisationsweite IT-Standards und -Richtlinien angesiedelt.

Anforderungen und Architektur-Dimensionen

Architekturrelevante Anforderungen

Anforderungen und Architektur-Perspektiven

Anforderungen auf der Organisationsebene

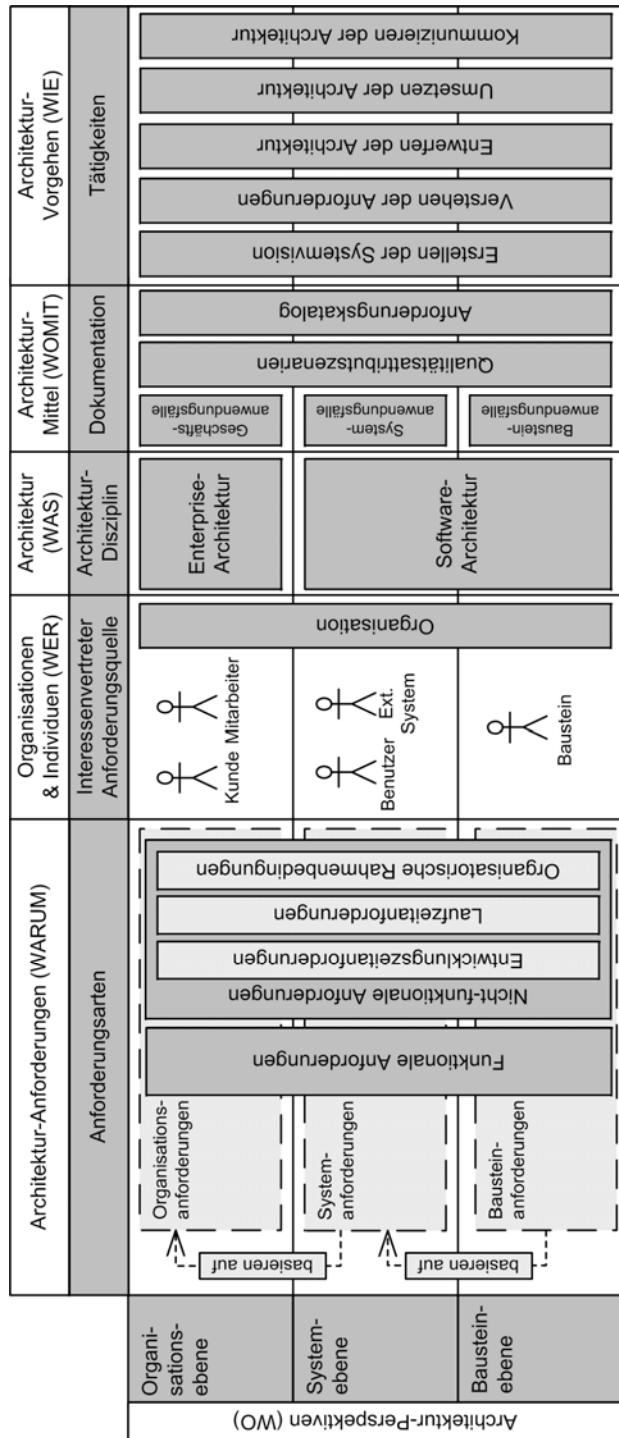


Abb. 5.4-1: Anforderungen im Kontext von Architektur.

Auf der Systemebene liegt das Augenmerk auf Systemen und auf den von ihnen zur Verfügung gestellten Funktionen und Qualitäten. Die Systemanforderungen basieren auf Organisationsanforderungen. Aus der funktionalen Organisationsanforderung, dass ein System zur Erfassung von Aufträgen existieren soll, können auf der Systemebene die konkreten funktionalen Anforderungen an das System abgeleitet werden. Ferner kann z. B. auf der Organisationsebene die Entwicklungszeitanforderung formuliert werden, dass Systeme innerhalb der Organisation auf Basis von JEE zu realisieren sind. Für das System bedeutet dies nun beispielsweise, dass das System die JSP, Java Servlets und EJB APIs berücksichtigen muss.

Anforderungen auf der Systemebene

Auf der letzten Ebene, der Bausteinebene, tauchen die Bausteinanforderungen in den inneren Aufbau des Systems ein, indem sie die Anforderungen nach bestimmten Bausteinen, deren Funktionalitäten sowie nicht-funktionalen Eigenschaften beschreiben. Für das Beispiel der Auftragsabwicklung heißt dies, dass auf der Bausteinebene eine Bausteinanforderung etwa die Anforderungen an einen Baustein zum Schreiben und Lesen von Aufträgen in eine Datenbank (*funktionale Anforderung*) beschreibt, der mittels JDBC realisiert werden muss (*nicht-funktionale Anforderung*). Basierend auf dem eingeführten JEE-Beispiel würden nicht-funktionale Bausteinanforderungen vorschreiben, dass Bausteine mittels JEE-Bausteinen (z. B. JSPs, Servlets, EJBs) realisiert werden müssen.

Anforderungen auf der Bausteinebene

Anforderungen werden von unterschiedlichen Interessenvertretern formuliert. Organisationsanforderungen ergeben sich beispielsweise aus den Wünschen von Kunden, Mitarbeitern, Geschäftspartnern oder Behörden. Darüber hinaus formuliert die Organisation selbst auf der Organisationsebene IT-Standards und Richtlinien. Systemanforderungen können wiederum den tatsächlichen Benutzern des zu entwickelnden Systems beziehungsweise den Systemen, mit denen das System interagiert, zugeordnet werden. Darüber hinaus wirken sich auch Vorgaben der Organisation auf der Systemebene aus. Bausteinanforderungen basieren auf den Anforderungen der Systembausteine, die mit dem betrachteten Systembaustein kollaborieren.

Anforderungen und Organisationen / Individuen

Je nachdem in welcher Architektur-Disziplin (siehe Kapitel 3) man als Architekt tätig ist, beschäftigt man sich mit unterschiedlichen Anforderungsarten. Agiert man beispielsweise als Enterprise-Architekt, wird man sich primär mit Anforderungen beschäftigen, die auf der Organisationsebene angesiedelt sind. Hierzu gehören funktionale und nicht-funktionale Anforderungen, die im Rahmen einer Enterprise-Architek-

Anforderungen und Architektur-Disziplinen

tur festgelegt werden. Diese Anforderungen drücken sich durch von der Organisation vorgegebene IT-Standards und -Richtlinien aus. Dagegen beschäftigt sich Software-Architektur mit den funktionalen und nicht-funktionalen Anforderungen der System- und Bausteinebene. Als Software-Architekt ist man dafür verantwortlich, eine Architektur zu entwerfen, die es erlaubt, ein System zu bauen, welches die an das System gestellten Anforderungen erfüllt. Hierzu gehört auch, dass man vorgegebene IT-Standards und -Richtlinien berücksichtigt. Der Fokus der Tätigkeit eines Software-Architekten liegt auf dem zu entwerfenden System.

Anforderungen und Architektur-Mittel

Anforderungen können mit verschiedenen Mitteln dokumentiert werden (siehe Abschnitt 6.6). Anwendungsfälle (englisch: *use cases*) sind ein adäquates Mittel, um funktionale Anforderungen auf allen Architektur-Ebenen zu beschreiben. Organisationsanforderungen können hierbei als Geschäftsanwendungsfälle (englisch: *business use cases*), Systemanforderungen als Systemanwendungsfälle (englisch: *system use cases*) und Bausteinanforderungen als Bausteinanwendungsfälle (englisch: *component use cases*) formuliert werden. Diese Klassifizierung von Anwendungsfällen stammt von Alistair Cockburn [Cockburn 2000]. Eine Komponente kann in diesem Zusammenhang als Systembaustein verstanden werden. Auf der System- und Bausteinebene findet man häufig auch Qualitätsattributszenarien zur Beschreibung nicht-funktionaler Anforderungen (siehe Abschnitt 6.3.1). Prinzipiell können Qualitätsattributszenarien natürlich auch auf der Organisationsebene genutzt werden, um nicht-funktionale Anforderungen an Organisationen zu dokumentieren. Anforderungskataloge kommen auf allen Ebenen zum Einsatz.

Anforderungen und Architektur-Vorgehen

Anforderungen spielen im Rahmen eines Architektur-Vorgehens eine wichtige Rolle. So ist ein Architekt bei der Formulierung der Systemvision und der darin enthaltenen Anforderungen beteiligt (Tätigkeit: *Erstellen der Systemvision*). Ferner muss er die an das System gestellten architekturelevanten Anforderungen verstehen (Tätigkeit: *Verstehen der Anforderungen*), um eine passende Architektur zu entwerfen (Tätigkeit: *Entwerfen der Architektur*). Die Software-Architektur wird beispielsweise dedizierte Subsysteme respektive Software-Bausteine vorsehen, die zur Erfüllung der definierten funktionalen Anforderungen verantwortlich sind. Ferner wird durch den Einsatz entsprechender Architekturmittel in einer Software-Architektur sichergestellt, dass nicht-funktionale Anforderungen wie Erweiterbarkeit und Plattform-unabhängigkeit gewährleistet werden. Darüber hinaus muss er sich mit der Realisierung der Anforderungen beschäftigen (Tätigkeit: *Umsetzen der Architektur*). Des Weiteren wird er den Interessenvertretern aufzei-

gen, wie die Architektur die architekturellen Anforderungen befriedigt (Tätigkeit: *Kommunizieren der Architektur*) Die architektonischen Tätigkeiten werden in Kapitel 8 näher behandelt.

5.5 Zusammenfassung

- > Anforderungen sind Kräfte, die das zu erstellende IT-System umreißen und den gestalterischen Spielraum des Architekten beschränken.
- > Eine Anforderung ist eine vom Anwender benötigte Fähigkeit (englisch: *capability*) des Systems, um ein Problem zu lösen oder ein Ziel zu erreichen bzw. eine Fähigkeit, die das System besitzen muss, damit es einen Vertrag, einen Standard, eine Spezifikation oder ein anderes formelles Dokument erfüllt.
- > Anforderungen werden von unterschiedlichen Interessenvertretern formuliert.
- > Anforderungen müssen korrekt, machbar, eindeutig und nachprüfbar sein.
- > Anforderungen können in funktionale und nicht-funktionale Anforderungen unterteilt werden.
- > Funktionale Anforderungen definieren benötigte Funktionalitäten.
- > Funktionale Organisationsanforderungen verkörpern die funktionalen Anforderungen, die an Organisationen z. B. von deren Kunden, Mitarbeitern, Geschäftspartnern oder von Behörden gestellt werden.
- > Funktionale Systemanforderungen drücken die konkreten funktionalen Bedürfnisse von Interessenvertretern bzw. von Systemen aus, die mit dem betrachteten System interagieren.
- > Funktionale Bausteinanforderungen verkörpern funktionale Eigenschaften, die ein Baustein eines Systems erfüllen muss, damit das System in der Lage ist, seine Anforderungen zu erfüllen.
- > Nicht-funktionale Anforderungen verkörpern Erwartungen und Notwendigkeiten, die von Interessenvertretern als wichtig erachtet werden und über die reine, gewünschte Funktionalität hinausgehen.
- > Es können unmittelbare und mittelbare nicht-funktionale Anforderungen unterschieden werden.
- > Unmittelbare nicht-funktionale Anforderungen bezeichnet man auch als Qualitäten bzw. als Qualitätsattribute.
- > Mittelbare nicht-funktionale Anforderungen repräsentieren Vorgaben oder Gegebenheiten, die eingehalten respektive berücksichtigt werden müssen.

**Zusammenfassung:
Architektur-
Anforderungen**

- > Entwicklungszeitanforderungen drücken Qualitäten und Rahmenbedingungen aus, die bei der Entwicklung eines Systems schwerpunkt-mäßig berücksichtigt werden müssen.
- > Laufzeitanforderungen beinhalten Erwartungen hinsichtlich des Ver-haltens eines Systems zur Laufzeit.
- > Organisatorische Rahmenbedingungen beinhalten Vorgaben wie Budget und Time-to-Market.
- > Architekturrelevante Anforderungen sind all die Anforderungen, die einen wesentlichen Einfluss auf den Entwurf der Architektur besitzen.
- > Anforderungen können auf unterschiedlichen Architektur-Ebenen (Organisations-, System- und Bausteinebene) angesiedelt werden. An-forderungen auf einer Architektur-Ebene basieren dabei auf Anfor-de-rungen auf der darüber liegenden Architektur-Ebene.
- > Je nachdem in welcher Architektur-Disziplin man als Architekt tätig ist, beschäftigt man sich mit unterschiedlichen Anforderungsarten.
- > Anforderungen können mit verschiedenen Mitteln dokumentiert werden. Anwendungsfälle (englisch: *use cases*) sind ein adäquates Mittel, um funktionale Anforderungen auf allen Architektur-Ebenen zu beschreiben.
- > Anforderungen spielen im Rahmen eines Architektur-Vorgehens bei allen Tätigkeiten eines Architekten eine wichtige Rolle.

6 | Architektur-Mittel (WOMIT)

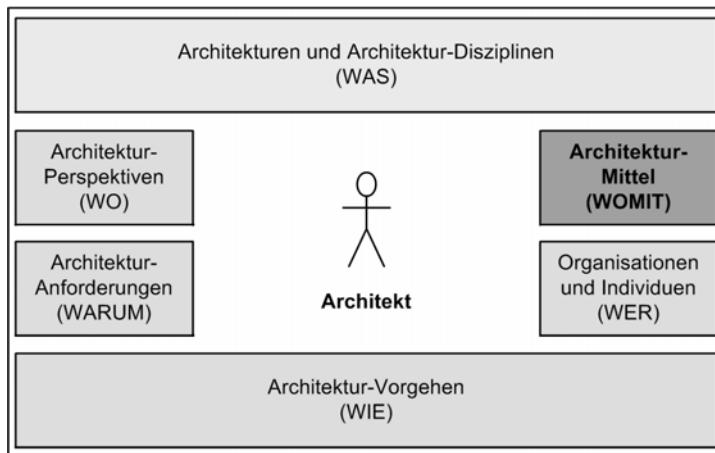


Abb. 6-1: Positionierung des Kapitels im Ordnungsrahmen.

Dieses Kapitel befasst sich mit der *WOMIT-Dimension* des architektonischen Ordnungsrahmens, indem es grundlegende Konzepte und Techniken aufzeigt, die heutzutage in den „Werkzeugkasten“ eines Software-Architekten gehören. Nach dem Lesen dieses Kapitels haben Sie eine Vorstellung davon erhalten, welche Mittel Sie einsetzen können, um Architekturen zu bewerten, zu beschreiben, zu erstellen und weiterzuentwickeln.

Übersicht

6.1	Architektur-Prinzipien	128
6.2	Grundlegende architektonische Konzepte	152
6.3	Architektur-Taktiken, -Stile und -Muster	194
6.4	Basisarchitekturen	216
6.5	Referenzarchitekturen	253
6.6	Architektur-Modellierungsmittel	264
6.7	Architekturrelevante Technologien	291

Grundlegende Konzepte der WOMIT-Dimension

Ein Überblick über den Zusammenhang der Themen, die in diesem Kapitel vorgestellt werden, gibt die Abbildung 6-2. Wie man in der Abbildung sehen kann, werden zunächst grundlegende Architekturmittel vorgestellt: Prinzipien, Konzepte, Muster, Stile und Taktiken. Danach werden weiterführende Mittel vorgestellt, die zur Umsetzung der grundlegenden Mittel dienen.

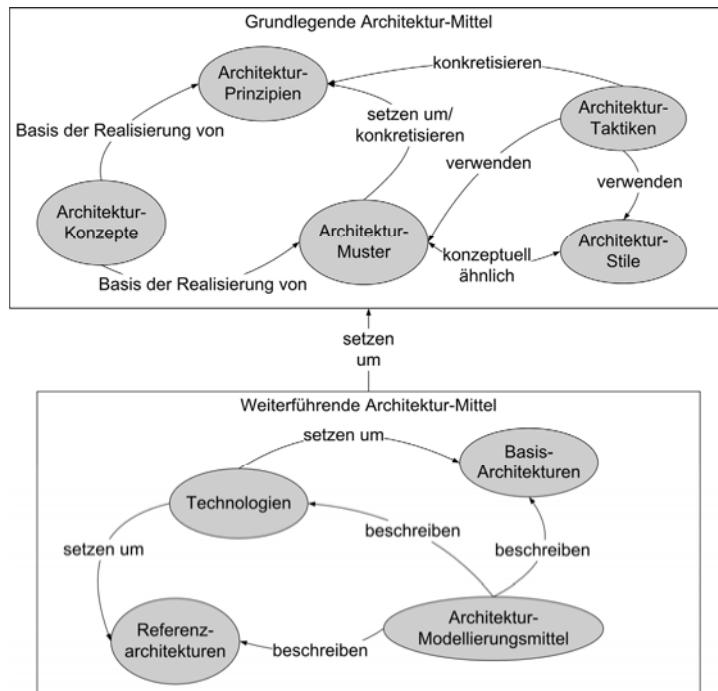


Abb. 6-2: Grundlegende Konzepte der WOMIT-Dimension.

Prinzipien, wie lose Kopplung oder hohe Kohäsion, sind sehr allgemeine Richtlinien. Grundlegende Konzepte, wie Objektorientierung oder Aspektorientierung, konkretisieren diese, indem sie die Basis zur Realisierung der Prinzipien darstellen. Taktiken konkretisieren allgemeine Prinzipien indem sie Handlungsanweisungen auf Basis von Qualitätsattributen anbieten. Noch konkreter und detaillierter in der Handlungsanweisung sind die sich ähnlichen Konzepte Muster und Stile. Architekturmuster und Architekturstile bieten detaillierte Lösungsansätze für konkrete Entwurfsentscheidungen.

Die bisher genannten Architekturmittel abstrahieren von konkreten Domänen, Technologien und Technologieansätzen. Die weiterführenden Architekturmittel stellen eine Konkretisierung in diesen Bereichen dar.

Basis-Architekturen, wie Schichtenarchitekturen, n-Tier-Architekturen oder Architekturen basierend auf Komponenten-Containern, sind konkrete Richtlinien, mit denen man Systeme ganzheitlich strukturieren kann. Referenzarchitekturen stellen allgemeine Lösungsansätze für konkrete Domänen, oft kombiniert mit bestimmten Technologieansätzen, dar. Architektur-Modellierungsmittel, wie beispielsweise die UML, Domain Specific Languages oder Architecture Description Languages, sind Ansätze zur Modellierung und Dokumentation von Architekturen. Zu guter Letzt werden die wesentlichen aktuellen Architektur-Technologien diskutiert – also Technologien, wie Plattformen und Infrastrukturen, mit einer grundlegenden Bedeutung für die Architektur des Systems.

Verschiedene Kategorien von Architektur-Mitteln haben einen unterschiedlichen Einfluss auf die zu erarbeitende Architektur. Abbildung 6-3 verdeutlicht diesen Sachverhalt. Der Einfluss steigt von den Architektur-Prinzipien (siehe Abschnitt 6.1) hin zu den Referenzarchitekturen (siehe Abschnitt 6.5).

Der Einfluss von Architektur-Mitteln sollte beachtet werden

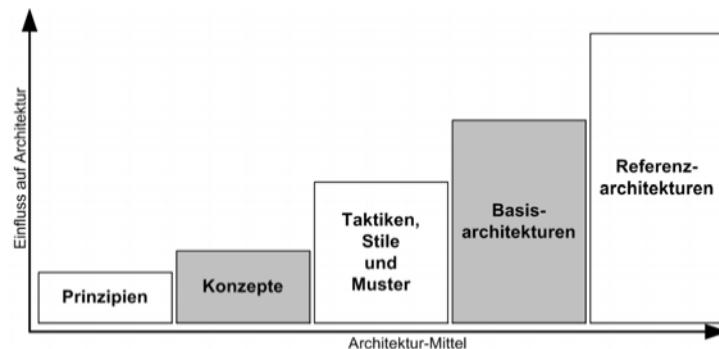


Abb. 6-3: Einfluss von Architektur-Mitteln.

Architektur-Prinzipien repräsentieren bewährte Grundsätze, die bei der Gestaltung einer Architektur zur Anwendung kommen sollten. Sie sagen noch nichts darüber aus, wie diese Prinzipien in konkreten Fällen angewendet werden. Hingegen zeigen Referenzarchitekturen klar auf, wie eine konkrete Architektur aussehen sollte. Der Grad der Strukturierung steigt somit an. Ebenso erhöht sich der Wiederverwendungsgrad, da auf bewährtes, konkretes Architektur-Wissen zurückgegriffen werden kann. Dies ist vorteilhaft, da der Architekt das Rad nicht neu erfinden muss. Allerdings nehmen Referenzarchitekturen auch viele Entscheidungen vorweg. Damit sinkt der Freiheitsgrad des Architekten. Solange die Referenzarchitektur für die konkrete Problemstellung unmittelbare Anwendung finden kann, ist dies unproblematisch. Bei-

Wie beeinflussen Mittel die Architektur?

spielsweise kann eine Referenzarchitektur je nach Umfang die fachliche als auch die technische Architektur umfassen. Die Verwendung der technischen Aspekte einer Referenzarchitektur setzt jedoch voraus, dass die Rahmenbedingungen auch die Verwendung der durch die Referenzarchitektur festgelegten Technologien erlauben. Sollte dies nicht zutreffen, kann bereits ein Teil der Referenzarchitektur nicht mehr verwendet werden. Ein Architekt muss daher eine Balance zwischen den durch Architektur-Mittel gemachten Vorgaben und der konkreten Problemstellung finden. Es empfiehlt sich, bei der Identifikation der Architektur-Mittel zunächst solche zu begutachten, die einen hohen Wiederverwendungsgrad besitzen. Erst wenn keine höherwertigen Mittel (z. B. Basis- und Referenzarchitekturen) zur Verfügung stehen, sollte auf grundlegendere Mittel (Architektur-Prinzipien) zurückgegriffen werden.

6.1 Architektur-Prinzipien

Einflussfaktoren auf eine Architektur

Wie in den vorhergehenden Kapiteln erläutert, beschäftigt sich die Software-Architektur in erster Linie mit den Bausteinen eines Software-Systems und deren Interaktion. Diese übernehmen die Umsetzung der funktionalen Anforderungen an ein Software-System. Zusätzlich dazu spielen eine große Anzahl von nicht-funktionalen Anforderungen, wie beispielsweise Performanz, Produkteinführungszeit, Kosten, Wartbarkeit, Wiederverwendbarkeit, Änderbarkeit, Verfügbarkeit und Einfachheit, eine zentrale Rolle (siehe auch Kapitel 5).

Diese Einflussfaktoren bestimmen wesentlich den Aufbau einer Software-Architektur. Dies heißt aber, dass zwei Software-Systeme mit gleichen technischen Anforderungen, die von zwei unterschiedlichen Architekten in unterschiedlichen Organisationen erstellt werden, unweigerlich voneinander abweichende Software-Architekturen haben. Es stellt sich die Frage: Wie erkennt man eine „gute“ Software-Architektur?

Gute und schlechte Architekturen

In der Tat kann man schwerlich sagen, dass eine Architektur an sich „gut“ oder „schlecht“ ist – sie erfüllt nur die gesetzten funktionalen und nicht-funktionalen Anforderungen, die an das Software-System gestellt werden, mehr oder weniger gut. Das heißt in anderen Worten: Die Architekturen haben unterschiedliche Ausprägungen der Qualitätsattributte. Beispielsweise mag eine hochflexible und konfigurierbare Server-Architektur für einen Anwendungsserver sehr geeignet sein. Hingegen ist dieselbe Architektur – im Vergleich zu einer weit inflexibleren Architektur – für einen Server im Bereich von eingebetteten Systemen

unter Umständen hochproblematisch: Zum einen wird die hohe Flexibilität im eingebetteten System nicht wirklich benötigt und zum anderen ist die flexiblere Architektur auch deutlich komplexer und benötigt mehr Speicher, Rechenleistung und andere Ressourcen.

Trotzdem gibt es allgemeine Prinzipien, die man beim Entwurf einer Software-Architektur beachten sollte. In diesem Abschnitt werden einige zentrale Architektur-Prinzipien näher erläutert. Diese setzen sich mit verschiedenen architektonischen Fragestellungen und Problemen auseinander. Es gibt zwei Hauptprobleme, die für so gut wie alle in der Folge behandelten Prinzipien eine Rolle spielen: die Reduktion der Komplexität einer Architektur und die Erhöhung der Flexibilität (oder Änderbarkeit) einer Architektur.

Architektur-Prinzipien

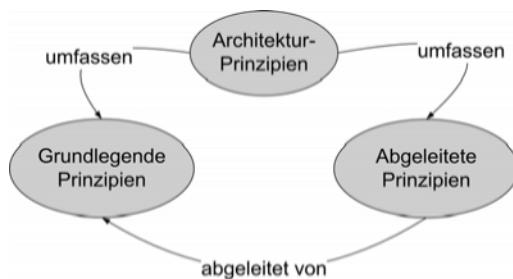


Abb. 6.1-1: Übersicht zur Kategorisierung der Architektur-Prinzipien.

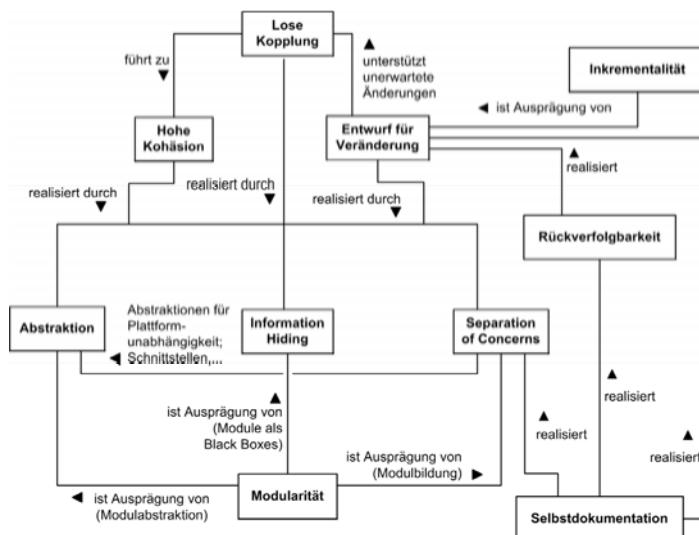


Abb. 6.1-2: Übersicht zu den grundlegenden Architektur-Prinzipien.

Abbildung 6.1-1 gibt eine grobe Übersicht über die Architektur-Prinzipien: Es werden grundlegende Prinzipien und davon abgeleitete Prinzipien behandelt. In Abbildung 6.1-2 werden die grundlegenden Prinzipien samt ihren Beziehungen untereinander im Detail dargestellt. Abbildung 6.1-3 stellt die abgeleiteten Prinzipien in einem Überblick dar. Die einzelnen Prinzipien und ihre Beziehungen werden in der Folge näher erläutert.

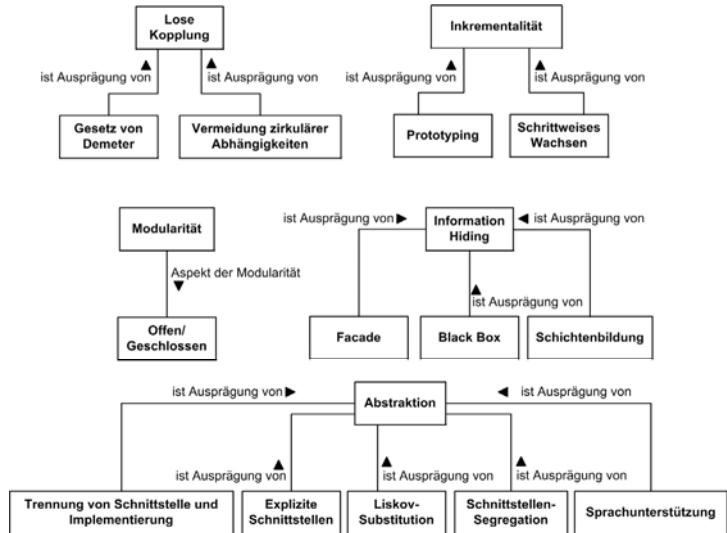


Abb. 6.1-3: Übersicht zu den abgeleiteten Architektur-Prinzipien.

6.1.1 Prinzip der losen Kopplung

Kopplung

Ein wichtiger Kern der Definition von Software-Architektur ist, dass Software-Architekturen sich in erster Linie mit den Bausteinen eines Software-Systems und deren Interaktion beschäftigen. Die Bausteine können wiederum durch viele Konstrukte realisiert werden, beispielsweise durch Module, Komponenten, Klassen, Prozeduren etc. Einerseits kann man die Bausteine der Software-Architektur unabhängig voneinander betrachten, andererseits stehen sie aber auch in einer Beziehung zueinander. Solche Beziehungen unter den Bausteinen einer Software-Architektur werden als *Kopplung* bezeichnet. Die Kopplung ist besonders wichtig, da sie gerade die Interaktionen der Bausteine charakterisiert, welche ja zentral für die Betrachtung aus architektonischer Sicht sind.

Den Begriff der Kopplung kann man sich an einigen Beispielen klar machen. Eine Klasse ist ein zentraler Baustein eines objektorientierten Systems. Daher macht es Sinn, die Kopplung der Klassen zu betrachten. Allgemein kann die Kopplung eines Bausteins der Architektur durch einfaches Zählen der Beziehungen zu einem anderen betrachteten Baustein gemessen werden. Also, auf Klassen bezogen, kann man z. B. messen, wie stark Klassen mit anderen Klassen gekoppelt sind. Dies geschieht, indem man die Anzahl der anderen Klassen ermittelt, die mit einer gegebenen Klasse in Beziehung stehen.

Eine solche einfache Metrik der Kopplung liefert einen ersten Eindruck über die Kopplung in einer Architektur. Es ist aber auch wichtig, zu betrachten, auf welche Weise die Bausteine einer Architektur gekoppelt sind. Verschiedene Realisierungen einer Beziehung zwischen den Bausteinen bedeuten als Resultat oft verschieden starke Abhängigkeiten. Angenommen zwei Klassen benötigen gemeinsame Daten. Als Beispiele sollen die folgenden drei Arten der Kopplung betrachtet werden (es gibt noch viele andere Arten der Kopplung):

- › Die Klassen können gegenseitig auf ihre (privaten) Daten zugreifen, was eine sehr starke Form der Kopplung darstellt, denn man kann keine der beiden Klassen mehr ändern, ohne die andere zu betrachten.
- › Eine weniger starke Kopplung liegt vor, wenn die Klassen über eine globale Datenstruktur kommunizieren, denn die direkten Abhängigkeiten der Klassen werden aufgelöst und in die globale Datenstruktur ausgelagert. Trotzdem ist die Kopplung noch recht stark, denn alle Änderungen, die die globalen Daten betreffen, betreffen auch alle Klassen, die mit den Daten arbeiten.
- › Wenn die Klassen nur über Methodenparameter kommunizieren, ist die Kopplung deutlich geringer: Die beteiligten Methoden enthalten nur die wirklich notwendigen Daten. Somit ziehen Änderungen an dieser Datenkopplung meist auch nur lokale Änderungen an den beteiligten Methoden der beteiligten Klassen nach sich.

Kopplungsmaßzahlen können auch für andere Architektur-Sichten und Arten von Bausteinen betrachtet werden. Eine andere sinnvolle Sicht ist die Betrachtung der Objekt- und Aufrufstrukturen zur Laufzeit. Hier kann man sagen, ein Objekt x hat dann eine hohe Kopplung zu einem anderen Objekt y , wenn es y oft aufruft.

Das Prinzip der losen Kopplung besagt, dass die Kopplung zwischen Systembausteinen möglichst niedrig gehalten werden soll. Das Prinzip

Prinzip der losen Kopplung

beschäftigt sich mit dem Problem, dass es für das Verstehen und Ändern eines Bausteins oft auch notwendig ist, weitere Bausteine zu verstehen oder zu ändern [Yourdon und Constantine 1978]. Es wird angenommen, dass diese Qualitätsattribute positiv durch lose Kopplung beeinflusst werden.

Ein Zweck der losen Kopplung ist also, die Komplexität von Strukturen gering zu halten: Je weniger stark ein Baustein mit anderen Bausteinen gekoppelt ist, umso einfacher ist es, den Baustein zu verstehen, ohne viele andere Bausteine gleichzeitig verstehen zu müssen. Ein zweiter Zweck ist, die Änderbarkeit der Architektur zu erhöhen: Je weniger Bausteine durch starke Kopplung von einer Änderung in einem anderen Baustein betroffen sind und je loser die existierenden Beziehungen sind, umso einfacher ist es, Änderungen lokal an einzelnen Bausteinen – ohne Betrachtung ihrer „Umwelt“ – durchzuführen.

Zusammenhang zu anderen Prinzipien

Wie man sieht: Lose Kopplung ermöglicht den Entwurf für Veränderung, ein weiteres wichtiges Prinzip der Software-Architektur. Überdies führt lose Kopplung zum Prinzip der hohen Kohäsion, denn wenn man die externen Beziehungen „lose“ hält, ist häufig eine direkte Konsequenz, dass die Bausteine intern stärker zusammenhängend entworfen werden.

Erreicht werden kann lose Kopplung insbesondere durch Umsetzung der Prinzipien Abstraktion, Separation of Concerns und Information Hiding. Die Einführung von Schnittstellenabstraktionen ist hier ein wichtiger Aspekt. Das heißt, die Belange „Schnittstelle“ und „Implementierung“ werden separiert und die Implementierungsinformationen werden hinter den Schnittstellen verborgen. Um dann lose Kopplung zu erreichen, sollte man versuchen, die Anzahl der Schnittstellenelemente gering zu halten und auch die Häufigkeit des Austausches von Informationen über Schnittstellen zu begrenzen. Grundsätzlich sollten Bausteine einer Architektur nur über wohl definierte Schnittstellen kommunizieren. Dies dient der Abstraktion und zusätzlich macht es auch die Kopplung von Systembausteinen kontrollierbar.

Gesetz von Demeter

Ein verwandtes Prinzip zu loser Kopplung ist das Gesetz von Demeter (Law of Demeter) [Lieberherr und Holland 1989], das besagt: Ein Systembaustein sollte nur eng verwandte Bausteine benutzen („Sprich nicht mit Fremden“). Dies ist insbesondere wichtig, da Menschen nur eine begrenzte Anzahl an Informationen im Kurzzeitgedächtnis halten können. Somit macht es Sinn, Systembausteine nicht mit zu viel externer Information zu überladen, um ihre Verstehbarkeit zu erhöhen.

Ein wichtiges Teilprinzip der losen Kopplung ist die Vermeidung zirkulärer Abhängigkeiten zwischen den Bausteinen eines Systems, weil zirkuläre Abhängigkeiten eine besonders hohe Kopplung der Bausteine nach sich ziehen. Zirkuläre Abhängigkeiten sind die Ursache für viele Probleme in der Software-Entwicklung, wie z. B. Deadlocks und erschwere Änderbarkeit. Ein wichtiges architektonisches Problem ist hier, dass keiner der zirkulär abhängigen Bausteine verstanden oder getestet werden kann, ohne den gesamten Zyklus zu verstehen oder zu testen. Das heißt, die arbeitsteilige Entwicklung solcher Bausteine gestaltet sich schwierig.

Vermeidung zirkulärer Abhängigkeiten

Statt zirkulärer Abhängigkeiten sollten die Beziehungen der Bausteine dem sogenannten Hollywood-Prinzip „Don't call us, we call you!“ folgen – also lose gekoppelt sein. Dies bezeichnet man auch als Inversion of Control.

Hollywood-Prinzip und Inversion of Control

Dependency Inversion ist eine Anwendung des Hollywood-Prinzips bzw. der losen Kopplung: Ein Baustein definiert eine Schnittstelle, mit der er arbeitet und andere Bausteine realisieren die Schnittstelle.

Dependency Inversion

Dependency Injection ist eine weitere Anwendung des Hollywood-Prinzips. Sie überträgt die Verantwortung für das Erzeugen und Verknüpfen von Bausteinen an ein extern konfigurierbares Rahmenwerk (englisch: *framework*), um die Kopplung zu der Umgebung des Bausteins zu reduzieren. Dadurch werden auch Abhängigkeiten leichter verwaltbar.

Dependency Injection

6.1.2 Prinzip der hohen Kohäsion

Kopplung beschäftigt sich mit den Abhängigkeiten von verschiedenen Bausteinen einer Architektur. Ein Baustein besteht aber häufig selbst aus vielen Teilen. Beispielsweise besteht eine Klasse aus Variablen und Methoden. Die Abhängigkeiten innerhalb eines Systembausteins werden als *Kohäsion* bezeichnet.

Kohäsion

Auch die Kohäsion kann man sich an Beispielen klarmachen. Bezogen auf die Methoden einer Klasse kann man die Kohäsion sinnvoll durch die Aufrufbeziehungen der Methoden dieser Klasse untereinander messen. In der Laufzeitsicht hat ein Objekt *x* dann eine hohe Kohäsion, wenn es sich oft selbst aufruft.

Prinzip der hohen Kohäsion

Die Kohäsion innerhalb eines Systembausteins soll möglichst hoch sein. Wie bei der losen Kopplung geht es auch hier um die lokale Änderbarkeit und Verstehbarkeit von Systembausteinen [Yourdon und Constantine 1978]: Wenn ein Systembaustein alle die zum Verstehen und Ändern relevanten Eigenschaften in seiner Beschreibung vereint, kann man ihn folglich ändern, ohne andere Systembausteine verstehen oder ändern zu müssen.

Zusammenhang zur losen Kopplung

Kopplung und Kohäsion stehen normalerweise miteinander in einer Wechselbeziehung. Zumeist gilt: Je höher die Kohäsion individueller Bausteine einer Architektur ist, desto geringer ist die Kopplung zwischen den Bausteinen. Dieser Zusammenhang wird schematisch in Abbildung 6.1-4 dargestellt.

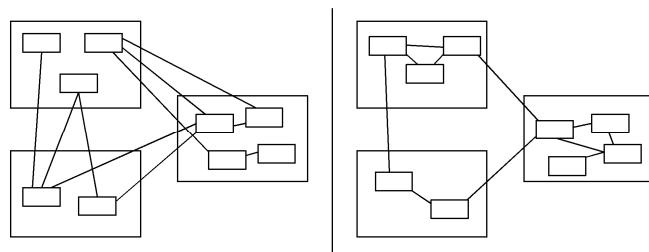


Abb. 6.1-4: Links sieht man ein Beispiel mit starker Kopplung und geringer Kohäsion; rechts sind lose Kopplung und hohe Kohäsion umgesetzt.

Eine Architektur mit loser Kopplung und hoher Kohäsion ist gut geeignet, wenn man die Gesamtstruktur des Software-Systems schnell verstehen.

Zusammenhang zu anderen Prinzipien

Eine hohe Kohäsion führt oft zu einer losen Kopplung und umgekehrt – also haben diese beiden Prinzipien in vielen Fällen eine Wechselwirkung untereinander. Erreicht werden kann hohe Kohäsion wiederum insbesondere durch Umsetzung der Prinzipien Abstraktion, Separation of Concerns und Information Hiding. Eine hohe Kohäsion lässt sich durch Kapselung verwandter Anforderungen in einem Systembaustein erreichen. Das heißt insbesondere, man wendet Separation of Concerns und Information Hiding beim Entwurf an: Verwandte Anforderungen tendieren zu hohem Kommunikationsbedarf, daher sollten sie Teil des selben Systembausteins sein, um die Kohäsion des Bausteins zu erhöhen. Ein solcher Baustein sollte alle Interna vor der Außenwelt verbergen, um die Kopplung lose zu halten. Architekturen mit hoher Kohäsion ermöglichen es, die einzelnen Systembausteine als *Black Boxes* zu betrachten, die unabhängig voneinander geändert und ausgetauscht werden können.

6.1.3 Prinzip des Entwurfs für Veränderung

Das Prinzip des Entwurfs für Veränderung (englisch: *Design for Change*) [Parnas 1994] ist ein sehr allgemeingültiges Prinzip. Dahinter steht das Problem, dass sich Software ständig ändert und Änderungen häufig schwer vorhersehbar sind. Manche Software-Architekturen gehen jedoch tendenziell mit Änderungen leichter um als andere. Die Idee hinter dem Entwurf für Veränderung ist, dass man vorhersehbare Änderungen architektonisch vorausplant.

Um diesem Prinzip gerecht zu werden, sollte man zunächst versuchen, die Architektur so zu entwerfen, dass man leicht mit den wahrscheinlichen Änderungen eines Software-Systems umgehen kann. Beispielsweise kann man weiter gehende Anforderungen bereits vorab erheben und beachten. Unklarheiten in Anforderungsspezifikationen können beispielsweise auf zu erwartende weiter gehende Funktionalitätsanforderungen hindeuten. Oft hat man es auch einfach mit erwartbaren Weiterentwicklungen zu tun. Beispielsweise, wenn eine Funktionalität aufgrund von Kostenzwängen nicht implementiert wurde, kann man unter Umständen damit rechnen, dass diese Funktionalität in einer der nächsten Versionen des Systems implementiert werden könnte.

Alternativ kann man Erfahrungen aus dem Entwurf ähnlicher Architekturen in einen neuen Architektur-Entwurf einfließen lassen. Änderungen, die oft für einen System-Typ benötigt werden, kann man somit schon bei der Entwicklung einer neuen, ähnlichen Architektur voraussehen. Eine Architektur sollte daher so entworfen werden, dass man mit erwarteten Änderungen leicht umgehen kann.

Die bisher besprochenen Änderungen sind erwartbar – zumindest mit entsprechender Erfahrung des Architekten. Hingegen gibt es auch solche Änderungen, die nicht erwartbar sind. Grundsätzlich ist es problematisch, generell Änderungen vorauszuplanen, wenn keine Erwartungen dafür vorliegen, denn der Entwurf für Veränderung bringt auch Nachteile mit sich. Z. B. kostet ein weiter gehender Entwurf Zeit und verursacht einen höheren Implementierungsaufwand. Oft haben hochflexible Architekturen auch Nachteile gegenüber einfacheren Architekturen. Beispielsweise kann der Ressourcenverbrauch (z. B. Performanz und Speicher) höher sein als bei inflexibleren Architekturen. Daher sollte man sehr vorsichtig sein, einen Entwurf für Veränderung an Stellen einzusetzen, wo man nicht sicher ist, dass diese Veränderung auch wirklich irgendwann einmal benötigt wird.

**Zur Idee des Entwurfs
für Veränderung**

**Erwartbare
Änderungen**

**Nicht erwartbare
Änderungen**

Zusammenhang zur losen Kopplung

Generell kann man einen Entwurf für Veränderung durch den Einsatz des Prinzips der losen Kopplung im Architektur-Entwurf erreichen. Beispiele von bekannten Ansätzen zum Architektur-Entwurf in diesem Bereich sind der Einsatz von serviceorientierten Architekturen (siehe Kapitel 6.7) oder Aspektorientierung (AOP) [Kiczales et al. 1997] (siehe Kapitel 6.2). Generell dienen lose gekoppelte Strukturen hier der raschen Änderbarkeit an bestimmten Stellen einer Architektur, ohne dabei die Änderung an einer Vielzahl anderer Stellen nachvollziehen zu müssen.

Lose gekoppelte Architekturen können Nachteile bergen, wie höhere Komplexität oder gesteigerten Ressourcenverbrauch, wenn sie falsch eingesetzt werden. Daher sollte man sie insbesondere dort einsetzen, wo häufige Änderungen vermutet werden. Dies sind insbesondere Stellen, wo schon oft Änderungen aufgetreten sind oder wo viele verschiedene Aspekte einer Architektur zusammenkommen (sogenannte „Hot Spots“ einer Architektur [Pree 1995]).

Beispiele für die Nutzung loser Kopplung zum Umgang mit unerwarteten Änderungen

Beispielsweise in verteilten Objekt-Middleware-Systemen, wie CORBA- oder Web-Service-Implementierungen, ist die Auswertung des verteilten Aufrufs ein solcher Hot Spot. Die meisten Middleware-Systeme, wie CORBA- oder Web-Service-Implementierungen, haben eine sogenannte Broker-Architektur [Buschmann et al. 1996], wie in Abbildung 6.1-5 dargestellt (nach [Völter et al. 2004]). Hier sieht man mehrere Aufruf-schnittstellen, an denen häufig neue Änderungsanforderungen wie Sicherheitsbelange, Logging, Transaktionen u. v. a. eingebracht werden müssen. Daher sind diese Punkte die Hot Spots der Broker-Architektur.

Wenn die Middleware eine Abstraktion zur Verfügung stellt, die eine einfache Änderung der Hot Spots ermöglicht, so kann man wahrscheinlich dort auch domänen-spezifische Änderungen, die schwer vorhersehbar sind, umsetzen

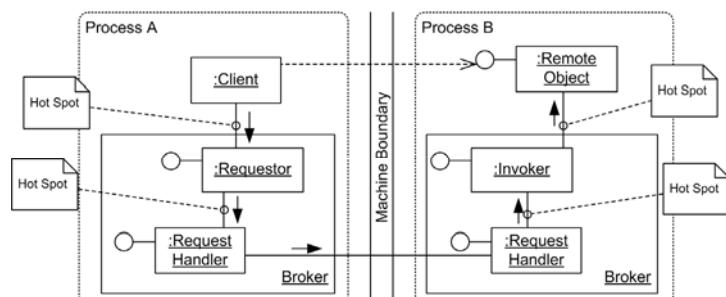


Abb. 6.1-5: Hot Spots einer Broker-Architektur.

Als ein zweites Beispiel für den Umgang mit unerwarteten Änderungen soll eine beliebige Komponenten-Architektur betrachtet werden: Hier kann man – ganz ungeachtet der tatsächlichen Funktionalität – damit rechnen, dass die Konfiguration der Komponenten sich öfters ändert. Daher ist es eine gute Idee, den Aspekt „Konfiguration von Komponenten“ nicht „fest verdrahtet“ zu kodieren, sondern eine leicht änderbare Abstraktion für Konfigurationsoptionen einzusetzen, wie beispielsweise eine Skriptsprache oder XML-Konfigurationsoptionen.

In beiden Beispielen wurde lose Kopplung eingesetzt, um die Änderbarkeit der Architektur zu erhöhen. Der Entwurf für Veränderung kann aber auch durch andere Prinzipien umgesetzt werden, wie Abstraktion, Modularität, Separation of Concerns und Information Hiding – welche ja auch schon bei der Umsetzung der losen Kopplung eine Rolle spielen.

Viele andere Prinzipien dienen ebenso dem Entwurf für Veränderung. Z. B. kann man den Aspekt „Dokumentation der Architektur“ vom System separieren und automatisieren. So kann man die Entwickler „zwingen“, architektonische Strukturen im Code zu beschreiben. Dies führt zur Umsetzung des Prinzips der Selbstdokumentation und erhöht die Änderbarkeit des Systems, da die architektonische Rolle eines zu ändernden Bausteins der Architektur so für den Entwickler während der Änderung klar ist [Parnas 1994].

Zusammenhang zu anderen Prinzipien

6.1.4 Separation-of-Concerns-Prinzip

Das Prinzip Separation of Concerns (deutsch in etwa: *Trennung von Aufgabenbereichen oder Belangen*) sagt allgemein aus, dass man verschiedene Aspekte eines Problems voneinander trennen und jedes dieser Teilprobleme für sich behandeln soll. Separation of Concerns ist ein allgemeines Software-Engineering-Prinzip, das nicht nur in der Software-Architektur, sondern auch in vielen anderen Software-Engineering-Bereichen zum Einsatz kommt. Es ist zurückzuführen auf das römische Prinzip „Teile und herrsche“ – ein generelles Prinzip, das man auch in vielen Situationen des täglichen Lebens einsetzt, um die Schwierigkeiten, auf die man stößt, zu lösen. Generell reduziert Separation of Concerns die Komplexität eines Problems und erlaubt die arbeitsteilige Bearbeitung.

Zur Idee von Separation of Concerns

Im Bereich der Software-Architektur wird Separation of Concerns für die Zerlegung eines Software-Systems in eine Struktur von Systembausteinen eingesetzt. Überdies gibt es eine Vielzahl von anderen Software-

Architektur-Bereichen, in denen Separation of Concerns angewendet werden kann. Dazu gehören unter anderem:

- > die Zerlegung der Anforderungen an eine Software-Architektur,
- > die Zerlegung einer komplexen Architektur-Beschreibung in Sichten auf die Architektur,
- > die Zerlegung der organisatorischen Verantwortlichkeiten für die Software-Architektur und
- > die Zerlegung der Prozesse der Erstellung einer Software-Architektur in Teilprozesse.

Einsatz von Separation of Concerns für die Modularisierung

Zunächst soll als Beispiel auf den vielleicht wichtigsten Einsatz von Separation of Concerns im Bereich Software-Architektur eingegangen werden: die Unterstützung der Modularisierung. Dies heißt in erster Linie, man soll Teile eines Software-Systems, die für bestimmte Angelegenheiten, Aspekte oder Aufgaben verantwortlich sind, identifizieren und als eigene Systembausteine kapseln. Das hat den Sinn, dass man das komplexe Gesamtsystem in verständliche und handhabbare Einzelteile zerlegt. Überdies ermöglicht die sinnvolle Zerteilung des Gesamtsystems in relativ unabhängige Einzelteile das Verteilen von Verantwortlichkeiten für verschiedene Systemteile und somit das parallele Arbeiten an dem Software-System durch mehrere Entwickler. Auch hier ist also ein zentraler Hintergedanke eine möglichst lose Kopplung der Bausteine einer Architektur durch ihre Modularisierung zu erreichen.

Dekompositions-kriterien

Als Kriterium für die Dekomposition eines Software-Systems kommen oft die Funktionalitätsanforderungen infrage. Das heißt, jeder identifizierte Baustein erfüllt eine bestimmte Funktionalität. Es gibt aber auch andere Unterteilungskriterien. Beispielsweise kann man Bausteine identifizieren, die möglichst wiederverwendbar sind, und somit die Wiederverwendung als zentrales Dekompositionskriterium einsetzen.

Beispiel einer Dekomposition

Als einfaches Beispiel kann man sich ein Software-System vorstellen, das Bestellungen in einem Unternehmen abwickelt. Dieses System muss die Bestellung mittels einer Benutzerschnittstelle vom Bearbeiter entgegennehmen und dabei die Datenbank der verfügbaren Artikel abfragen. Dann muss es die Eingaben des Bearbeiters prüfen und nach erfolgreicher Prüfung an den Lieferanten weiterleiten. Während dieser Schritte muss auf die Lieferantendatenbank zugegriffen werden und die Bestellung muss in der Bestellungsdatenbank archiviert werden. Angenommen dieses System wäre mit einer monolithischen Architektur implementiert worden, dann würde dies eine ganze Reihe von Problemen aufwerfen, wie die folgenden:

- > Das System wäre sehr komplex und schwer zu verstehen.
- > Es wäre nicht einfach möglich, arbeitsteilig verschiedene Teile des Systems zu bearbeiten, da Änderungen an einer Stelle nicht unabhängig von Änderungen an anderen Stellen wären.
- > Es wäre problematisch, Änderungen am System durchzuführen. Beispielsweise müsste für die Einführung einer weiteren Benutzerschnittstelle das komplette System nach Benutzerschnittstellencode durchsucht werden.
- > Einzelteile des Systems könnten nicht wiederverwendet werden.

Diese Probleme würden nicht auftreten, wenn man das System nach dem Prinzip Separation of Concerns entwirft. Beispielsweise könnte man das System in die Bausteine Benutzerschnittstelle, Bestellabwicklung, Datenbankzugriff allgemein, Datenbankzugriff für die Artikeldatenbank, Datenbankzugriff für die Lieferantendatenbank und Datenbankzugriff für die Bestellungsdatenbank unterteilen.

Separation of Concerns ist im Bereich von Software-Architekturen schwerlich eindimensional zu betrachten. Beispielsweise erfolgte am obigen Beispiel die Dekomposition entlang der Funktionalität, also der fachlichen Teile. Andere wichtige Aspekte, wie Performanz, Nutzbarkeit, Ressourcenverbrauch, Zusatzdienste wie Logging und Transaktionen etc., wurden nicht explizit berücksichtigt.

Grundsätzlich wird angestrebt, eine *Trennung von fachlichen und technischen Teilen* zu erreichen. Damit wird letztlich erreicht, dass fachliche Abstraktionen von ihrer konkreten technischen Umsetzung getrennt werden, das heißt, die evolutionäre Weiterentwicklung von verschiedenen Bausteinen wird gefördert.

Man kann allerdings noch einen Schritt weiter gehen und noch weitere Dimensionen als nur die fachlichen und technischen Aspekte betrachten. Beispielsweise kann man in der Modularisierung der Architektur die Hauptfunktionalität von *Aspekten* wie Transaktionsmanagement, Sicherheit, Logging etc. unterscheiden. Auch kann man verschiedene Qualitätsattribute wie Performanz, Nutzbarkeit, Ressourcenverbrauch oder Flexibilität von den fachlichen Belangen des Systems separieren.

Die explizite Betrachtung aller dieser Dimensionen bezeichnet man als mehrdimensionales Separation of Concerns [Tarr 2004]. Mehrdimensionales Separation of Concerns wird von einigen Ansätzen unterstützt. Der momentan vielleicht bekannteste ist die Aspektorientie-

Trennung von fachlichen und technischen Teilen

Mehrdimensionales Separation of Concerns und AOP

rung [Kiczales et al. 1997], welche die verschiedenen Aspekte in eigene Bausteine zerlegt und automatisiert wieder zu einem lauffähigen System komponieren kann (siehe auch Abschnitt 6.2).

6.1.5 Information-Hiding-Prinzip

Zur Idee des Information Hiding

Information Hiding (deutsch in etwa: *geheim halten von Information*) ist ein fundamentales Prinzip zur Gliederung und zum Verstehen komplexer Systeme. Das Prinzip sagt allgemein, dass man einem Klienten nur den wirklich notwendigen Teilausschnitt der gesamten Information zeigt, der für die Aufgabe des Klienten gebraucht wird und alle restliche Information verbirgt. Da Software-Architekturen inhärent komplex sind, ist dieses Prinzip von enormer Bedeutung für die Verständlichkeit von Architekturen.

Information Hiding und Modularisierung

In einer Software-Architektur findet Information Hiding zum Beispiel Anwendung in der Modularisierung eines Systems: Entwurfsentscheidungen werden in einem Systembaustein gekapselt und nach außen durch wohl definierte Schnittstellen bekannt gegeben. Die Realisierung des Systembausteins ist aber dem verwendenden Systembaustein unbekannt. In Objekt- und Komponentenkonzepten gibt es beispielsweise das Konzept der Sichtbarkeit von Information, das Information Hiding unterstützen soll. Hier findet man die Unterscheidung in öffentliche („public“) und private Variablen und Operationen: Klienten können nur auf öffentliche Elemente zugreifen, private Elemente sind für Klienten verborgen.

Data Hiding

Ein Teilaspekt von Information Hiding ist Data Hiding, also das Verbergen von Daten. Dieser Aspekt wird zum Beispiel durch die Objektorientierung oft umgesetzt, wenn Objekte mit ihren Methoden die Daten verbergen (also: alle Daten sind privat und, wo notwendig, gibt es öffentliche Methoden für den Zugriff). Data Hiding wird aber auch durch Datenbankschnittstellen oder Abfragesprachen realisiert.

Information Hiding mittels einer Facade

Information Hiding ist nicht beschränkt auf einzelne Bausteine einer Software-Architektur. Es ist auch ein wichtiges Strukturierungsprinzip für größere Strukturen der Architektur. Beispielsweise kommt das Facade-Entwurfsmuster [Gamma et al. 1995] in vielen Architekturen zum Einsatz. Eine Facade ist ein Objekt, das ein ganzes Subsystem vor dem direkten Zugriff schützt. Die Facade liefert eine gemeinsame Schnittstelle für die Bausteine eines Subsystems und verbirgt somit das

dahinter liegende Subsystem. Abbildung 6.1-6 zeigt ein Beispiel für eine Facade.

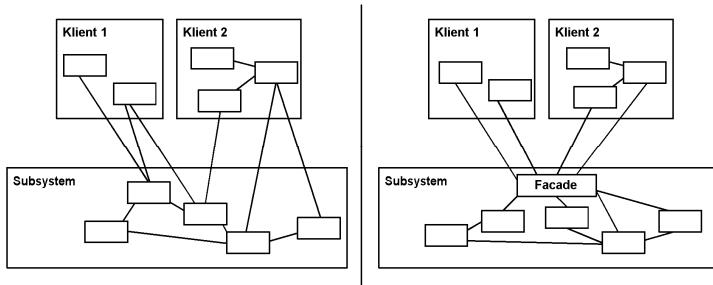


Abb. 6.1-6: Links sieht man ein Subsystem mit direkten Beziehungen; rechts ist bei dem gleichen Subsystem eine Facade eingezogen.

Ein typisches Beispiel für ein Facade-Objekt ist ein Interpreter: Der Interpreter besteht üblicherweise aus einer Vielzahl von Bausteinen, die die Klienten nicht zu Gesicht bekommen, wie Parser, Implementierung von Sprachelementen, Byte-Code Compiler etc.

Man sieht an diesem Beispiel: Auch Information Hiding dient der Umsetzung der losen Kopplung. Im vorangegangenen Beispiel diente die Einführung des Facade-Entwurfsmusters dazu, die Klienten von den internen Bausteinen des Subsystems zu entkoppeln. Dies wurde durch das Verbergen von internen Details dieser Bausteine erreicht.

Auch eine Schichten-Architektur ist gewöhnlich so aufgebaut, dass jede Schicht nur die direkt darunterliegende Schicht sieht. Das heißt, aus der Sicht von Schicht X verbirgt die Schicht $X-1$ alle darunterliegenden Schichten. Auf eine Schicht sollte möglichst nur über klare Schnittstellen zugegriffen werden. Die nutzende Schicht sollte keine schichtspezifischen Objekte oder andere Implementierungsdetails sehen. Das führt dann z. B. dazu, dass Datenaustausch zwischen verschiedenen Schichten über „neutrale“ Datentransferobjekte geschieht.

Information Hiding durch Schichtenbildung

Ein typisches Beispiel ist ein verteiltes Objektsystem wie CORBA. Eine CORBA-Implementierung verbirgt die darunterliegenden Protokollsichten, die Betriebssystem-APIs, das Netzwerk etc. so weit wie möglich. Sie ist selbst in Schichten, wie Anwendungsschicht, Aufrufschicht und Request-Handling-Schicht, unterteilt. Eine solche Schichtenbildung für eine CORBA-Middleware ist in Abbildung 6.1-7 dargestellt.

Beispiel: Schichtenbildung in CORBA

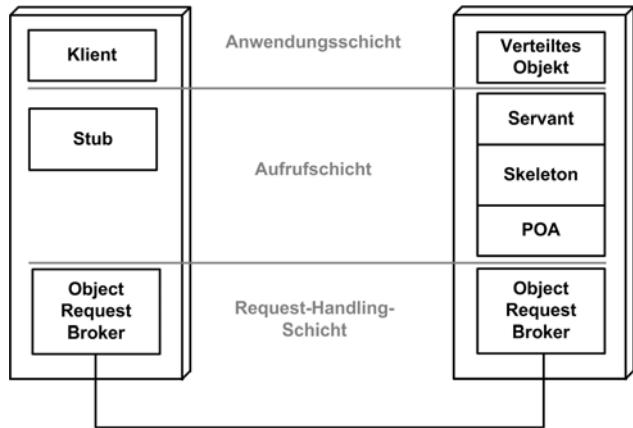


Abb. 6.1-7: Beispiel zur Schichtenbildung: CORBA-Middleware.

Black-Box-Prinzip

Eine weitere Ausprägung des Information Hiding ist das Black-Box-Prinzip, welches besagt, dass die Interna eines Systembausteins nicht für den Klienten sichtbar sein sollen, sondern nur die Schnittstelle des Bausteins. Dadurch können die Interna verändert werden, ohne dass Klienten von den Änderungen betroffen sind. Das Black-Box-Prinzip wird häufig durch die Schnittstellenabstraktionsprinzipien, die in dem nächsten Abschnitt besprochen werden, umgesetzt.

6.1.6 Abstraktionsprinzipien

Zum Konzept der Abstraktion

Abstraktion bedeutet, ein komplexes Problem dadurch verständlicher zu machen, dass man wichtige Aspekte identifiziert und die unwichtigen Details vernachlässigt. Daraus folgt, dass Abstraktion ein Spezialfall des Separation of Concerns ist: Man separiert die wichtigen Details von den unwichtigeren. Abstraktion ist ein mächtiges Konzept, das in allen möglichen Ingenieursdisziplinen angewendet wird, um komplexe Probleme zu verstehen und zu bewältigen. Die Abstraktion durchdringt auch die Erstellung von Software: Abstraktionen werden in Programmiersprachen, Software-Prozessen, Entwurfsmethoden, Architektur-Beschreibungs-sprachen etc. benutzt.

Teilprinzipien mit Fokus auf Schnittstellen

Im Bereich der Software-Architektur gibt es einige spezielle Teilprinzipien der Abstraktion, die sich auf Schnittstellenabstraktionen beziehen. Das Resultat der Anwendung dieser Abstraktionsprinzipien soll ein *Fokus auf die Schnittstellen* sein: Erst durch die Beziehungen der Bausteine eines Systems untereinander kommt eine Architektur wirklich zum Tragen. Wesentlich für das Zustandekommen und die Qualität

dieser Beziehungen sind die Schnittstellen der Bausteine. Die Prinzipien für Schnittstellenabstraktionen sind im Einzelnen:

- > *Explizite Schnittstellen*: Dieses Prinzip besagt, dass man anhand eines Systembausteins direkt erkennen kann, mit welchen anderen Bausteinen er kommuniziert. Überdies soll jeder Baustein explizit bekannt geben, welche Schnittstellen er an Klienten herausgibt. Typische Beispiele für explizite Schnittstellen sind Header-Files (wie z. B. in C++) und Schnittstellenbeschreibungssprachen (englisch: *interface description language, IDL*). Der Entwurf einer Software-Architektur soll auf Basis dieser Schnittstellen erfolgen und diese sollen nicht umgangen werden.
- > *Trennung von Schnittstelle und Implementierung*: Schnittstellen sollen separat von den Implementierungen beschrieben werden, damit der Klient sich auf die Schnittstelle verlassen kann, ohne die Implementierungsdetails zu kennen. Beispielsweise kann man so verschiedene Versionen eines Systembausteins parallel verwenden oder Implementierungen verschiedener Hersteller verwenden, ohne den Klienten ändern zu müssen. Dies macht insbesondere Sinn, wenn die Schnittstelle standardisiert ist. Viele Entwurfsmuster erreichen dadurch Flexibilität, dass sie eine abstrakte Schnittstelle von konkreten Implementierungen trennen (vgl. z. B. [Gamma et al. 1995]). Dieses Prinzip wird auch als Dependency-Inversion-Prinzip (deutsch: *Prinzip der Umkehrung von Beziehungen*) [Martin 2000] bezeichnet.
- > *Liskov-Substitutions-Prinzip* [Liskov 1988]: Bei Vererbungsabstraktionen sollen erbende Klassen von Klienten durch die Schnittstelle der vererbenden Klasse aufrufbar sein. Dieses Prinzip sorgt insbesondere dafür, dass ein Klient sich darauf verlassen kann, dass alle Objekte eines Typs (also: auch ererbte) die gleiche Schnittstelle unterstützen. Bei diesem Prinzip ist es wichtig zu beachten, dass viele Programmiersprachen (inklusive objektorientierter Sprachen) es erlauben, das Prinzip zu verletzen, indem erbende Klassen Schnittstellen von vererbenden Klassen überschreiben. Dies wird problematisch, wenn erbende Klassen Teile der Schnittstelle „verstecken“ (aus öffentlich wird privat) und somit die Signatur einer Schnittstelle verändern; z. B. eine Methode wirft plötzlich keine Ausnahmen (englisch: *exception*) mehr.
- > *Schnittstellen-Segregations-Prinzip* [Martin 2000]: Ein Klient sollte nie auf einer Schnittstelle basieren, die er nicht benutzt. Das heißt insbesondere auch, man sollte komplexe Schnittstellen, auf denen mehrere Kliententypen basieren, in mehrere einzelne Schnittstelle auftrennen (segregieren).

- > *Sprachunterstützung für Abstraktionen* [Meyer 1997]: Architektonische Abstraktionen, wie Komponenten oder Schnittstellen, sollten sowohl in der Entwurfssprache wie auch in der Programmiersprache sprachlich unterstützt werden. Ist dies nicht möglich, kann man die Sprache eventuell mit geeigneten Abstraktionen erweitern. Dies hat den Zweck, dass der Architekt oder Entwickler eine Abbildung der Abstraktionen nicht wieder und wieder von Hand durchführen muss. Ein weiterer Vorteil ist, dass man Systembausteine und Schnittstellen syntaktisch sofort im Programmtext erkennen.
- > *Design-by-Contract* [Meyer 1997]: Ein wichtiger Aspekt zu Schnittstellenabstraktionen ist, dass gängige Schnittstellenabstraktionen nur die Syntax standardisieren aber nichts über die Bedeutung der Beziehung aussagen. Dabei bleiben z. B. das verwendete Protokoll oder die Semantik der Operationen und Daten offen. Somit obliegt es dem Architekten, für die Dokumentation der Bedeutung einer Beziehung zu sorgen. Selbstdokumentation ist eine Möglichkeit, dieses Problem anzugehen. Eine andere Möglichkeit ist der Entwurf mit einem Vertrag (englisch: *design by contract*). Hier gibt man zu den Beziehungen geeignete Vor- und Nachbedingungen an, sowie Invarianten, die die Beziehung näher charakterisieren.

Zusammenhänge zu anderen Prinzipien

Wie oben bereits erklärt, dienen Schnittstellenabstraktionen oft der Realisierung von loser Kopplung. Diesem Ziel dienen direkt oder indirekt auch andere Abstraktionen, wie Aspekte, Komponenten, Klassen etc.

Auch das Modularitätsprinzip hängt eng mit Abstraktionen zusammen, da eine sinnvolle Modularisierung üblicherweise eine implizite oder explizite Modulabstraktion erfordert.

Ein Aspekt, bei dem Abstraktion eng mit Information Hiding zusammenhängt, ist die Portabilität. Eine Architektur oder ihre Systembausteine sollten auch in anderen Umgebungen verwendbar sein, als in denen, in denen sie erstellt wurden. Ein wichtiger Aspekt dabei ist die Plattformunabhängigkeit. Typischerweise werden hier Abstraktionen verwendet, die ein Information Hiding der Plattform-Details leisten. Beispiele sind virtuelle Maschinen, die einen Byte-Code einer Programmiersprache auf mehreren Betriebssystemen ausführen können, und Datenbankzugriffsschichten, die Datenbankoperationen auf verschiedenen Datenbankprodukten mit einer einheitlichen Schnittstelle unterstützen.

6.1.7 Modularitätsprinzip

Die Architektur sollte aus wohl definierten Systembausteinen bestehen, deren funktionale Verantwortlichkeiten klar abgegrenzt sind. Das heißt, dass die Systembausteine leicht austauschbar und in sich abgeschlossen sind. Insbesondere dient die Modularität der Änderbarkeit, Erweiterbarkeit und Wiederverwendbarkeit von Bausteinen einer Architektur.

Ursprünglich bezog sich das Modularitätsprinzip in erster Linie auf das Komponieren von einzelnen Operationen. Dies reicht jedoch nicht aus, um änderbare, erweiterbare und wiederverwendbare Strukturen in Software-Architekturen zu ermöglichen, da einzelne Operationen keine autonomen, in sich selbst abgeschlossenen Systembausteine sind. Einzelne Operationen haben gewöhnlich komplexe Abhängigkeiten zu anderen Operationen und zu Daten. Im Gegensatz dazu besagt das Modularitätsprinzip jedoch, dass man in sich selbst abgeschlossene Systembausteine mit einfachen und stabilen architektonischen Beziehungen anstreben sollte.

Es mag auffallen, dass Modularität bereits in den Beschreibungen einiger anderer Prinzipien erwähnt wurde. In der Tat ist das Modularitätsprinzip, wie es hier im Kontext von Software-Architektur betrachtet wird, eine – besonders wichtige – Kombination der bereits besprochenen Prinzipien Abstraktion, Separation of Concerns und Information Hiding, die gerade dann zum Tragen kommt, wenn diese drei Prinzipien zur Umsetzung der Prinzipien der losen Kopplung und der hohen Kohäsion kombiniert werden.

Es gibt eine Vielzahl von Ansätzen, die die Modularität einer Software-Architektur unterstützen, wie beispielsweise die Objektorientierung, Komponentenansätze, Schichten-Architekturen, n-Tier-Architekturen und viele andere. Diese werden im Zuge dieses Buches noch näher erläutert.

Grundsätzlich sollte jeder architektonische Modularitätsansatz eine Anzahl an Kriterien erfüllen, die die Modularität auszeichnen. So gesehen ist die Modularität nicht nur von einem speziellen Ansatz abhängig, sondern in erster Linie von dem Entwurf durch den Architekten. Beispielsweise kann ein prozedurales C-System eine hoch modulare Architektur haben, wenn die Entwickler diszipliniert vorgehen. Hingegen kann ein System, das mit einem Komponentenansatz wie EJB entwickelt wurde, trotzdem völlig unmodular sein, wenn die Entwickler wichtige Grundsätze der Modularität verletzen.

Zur Idee der Modularität

Modularitätsansätze

Beispiel für die Modularisierung

Als Beispiel für die Modularisierung eines Systems soll ein Heizungssteuerungssystem dienen. Dieses reguliert eine zentrale Ofensteuerung und muss für jeden beheizten Raum einen Temperatursensor, eine Zieltemperatur, die aktuelle Raumbelegung und die Heizzeiten verwalten. Außerdem ist ein Außentemperatursensor für das gesamte Gebäude vorhanden.

Ein völlig unmodularer Entwurf würde all diese Elemente in einem Algorithmus, der auf einer Datenstruktur operiert, zusammenfassen. Dies wäre jedoch schwer verständlich und Änderungen an einzelnen Systembausteinen würden jeweils die Betrachtung des gesamten Algorithmus und der gesamten Datenstruktur nach sich ziehen.

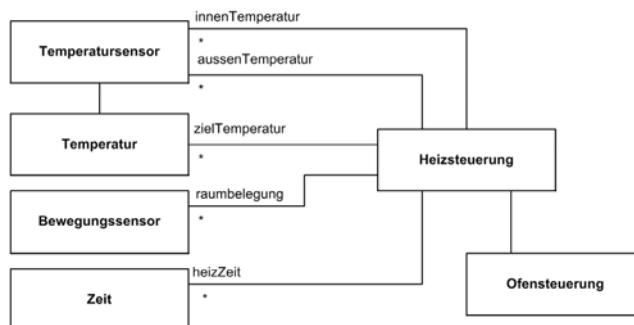


Abb. 6.1-8: Beispielentwurf mit modularen Systembausteinen, aber noch ist eine Gott-Klasse vorhanden.

Ein besserer Entwurf wäre, das System in einzelne modulare Systembausteine zu zerlegen, wie in Abbildung 6.1-8, da jeder Einzelbaustein nun unabhängig voneinander betrachtet werden kann. Hier wurden Klassen als modulare Systembausteine verwendet. In diesem Entwurf gibt es aber noch ein großes Problem hinsichtlich der Modularität: Die Klasse „Heizsteuerung“ ist eine sogenannte „Gott-Klasse“ – eine Klasse, die alle Verantwortlichkeiten des Systems (oder Subsystems) in sich vereint. Auch wenn die Belange in modulare Systembausteine zerlegt sind, so sind trotzdem ihre Verantwortlichkeiten in der Heizungssteuerung vereint. So gut wie jede Änderung am System zieht eine Änderung an der Gott-Klasse nach sich und betrifft somit gleich mehrere Verantwortlichkeiten. Außerdem ist es schwer, die verschiedenen Abstraktionen, die in einer Gott-Klasse vermischt sind, zu verstehen.

Gott-Klassen sollte man durch geeignete Dekomposition auftrennen. Wie Abbildung 6.1-9 zeigt, kann man dies in dem Beispiel durch Einführung einer Klasse „Raum“ erreichen: Die Heizsteuerung kann nun die

Entscheidung, ob zu heizen ist, an die einzelnen Räume delegieren, die wiederum ihre Teilbausteine benutzen, um die entsprechenden Informationen zu erhalten.

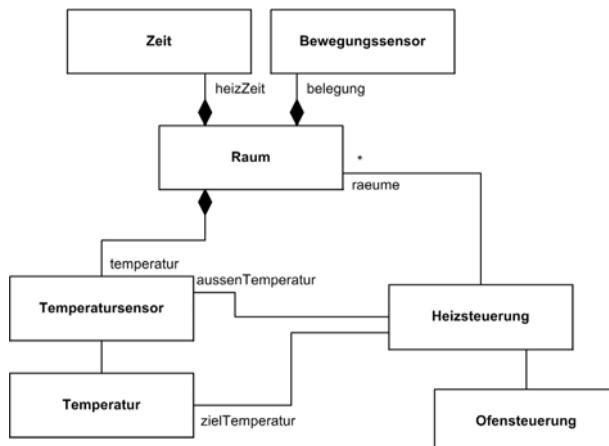


Abb. 6.1-9: Beispielentwurf mit modularen Systembausteinen, ohne eine Gott-Klasse.

Wie oben bereits erläutert, hängt Modularität eng mit einer Reihe weiterer Architektur-Prinzipien zusammen. Die beiden Prinzipien Separation of Concerns und Information Hiding können genutzt werden, um das Prinzip Modularität umzusetzen. Modularität ist auch eng mit Abstraktion verbunden: Durch geeignete Modul- und Schnittstellenabstraktionen, wie beispielsweise Objektmodelle oder Komponentenmodelle, ist es möglich, die Modularität eines Systems zu unterstützen. Jedoch bleibt das Finden der richtigen Dekomposition noch immer Aufgabe des Architekten – die Ansätze helfen nur bei der Umsetzung eines Entwurfs.

Zusammenhang zu anderen Prinzipien

Modularität dient der losen Kopplung und hohen Kohäsion, denn sie ermöglicht es, zusammengehörige Belange in einem modularen Systembaustein zu kapseln. Die Kopplung zwischen den Systembausteinen wird durch die Nutzung expliziter Schnittstellen zwischen den Bausteinen reduziert.

Ein wichtiger Aspekt der Modularität ist das Open-/Closed- (deutsch: Offen-/Geschlossen-) Prinzip [Meyer 1997]. Dieses Prinzip besagt, dass Systembausteine offen für Änderungen sein sollen, aber geschlossen für die Nutzung ihrer internen Details durch andere Systembausteine. Die Offenheit wird im Wesentlichen durch das Prinzip „Entwurf für Verän-

Offen-/Geschlossen-Prinzip

derung“ erreicht. Die Geschlossenheit wird durch Abstraktionen für stabile Schnittstellen und das Information-Hiding-Prinzip erreicht.

6.1.8 Rückverfolgbarkeitsprinzip

Zur Idee der Rückverfolgbarkeit

Die Rückverfolgbarkeit (englisch: *traceability*) architektonischer Strukturen und Entscheidungen zu gewährleisten, ist wichtig für die Verstehbarkeit einer Architektur. Daher sollte es möglich sein, die tatsächlichen architektonischen Strukturen, wie sie im Code umgesetzt werden, auch in anderen Beschreibungen einer Architektur wiederzufinden, wie beispielsweise einem Entwurfsmodell oder der Anforderungsspezifikation. Umgekehrt sollte es genauso möglich sein, eine Anforderung den Systembausteinen zuzuordnen, die sie umsetzen. Auch unter verschiedenen Beschreibungen auf derselben Ebene sollte eine solche Rückverfolgbarkeit existieren. Beispielsweise sollten verschiedene Entwurfssichten auf die Architektur aufeinander abbildbar sein.

Ansätze für die Rückverfolgbarkeit

Einfache Regeln, wie dass man architektonische Strukturen im Quelltext und in Entwurfsdokumenten als solche annotiert (z. B. durch Kommentare) und dass man konsistent die gleichen Namen für die gleichen Bausteine verwendet, können die Rückverfolgbarkeit schon entscheidend verbessern. Es gibt viele Ansätze, die eine weiter gehende Unterstützung für die Rückverfolgbarkeit bieten. Beispielsweise kann man im Code Metadaten einbetten, die die Anforderungen an einen Systembaustein referenzieren und so leicht auffindbar machen.

Zusammenhang zu anderen Prinzipien

Rückverfolgbarkeit dient der losen Kopplung und dem Entwurf für Veränderung, weil dadurch die Strukturen leichter verstehbar werden und somit auch unabhängiger und änderbarer.

6.1.9 Selbstdokumentationsprinzip

Zur Idee der Selbstdokumentation

Selbstdokumentation bedeutet, dass der Architekt oder Entwickler eines Systembausteins sich bemühen sollte, jede Information über den Systembaustein zum Teil des Systembausteins selbst zu machen [Meyer 1997]. Dies hat zunächst den Sinn, den Entwurf für Veränderung hinsichtlich der Änderung von Dokumentationen und anderen Zusatzinformationen zu unterstützen.

Im täglichen Geschäft werden Dokumentationen tendenziell oft vergessen, wenn kleinere Änderungen an der Software stattfinden. Dadurch

werden der Code, die Dokumentationen, Architektur-Beschreibungen, Entwürfe und andere Beschreibungen des Software-Systems schnell inkonsistent untereinander.

Das Selbstdokumentationsprinzip hängt aber auch mit der Rückverfolgbarkeit zusammen: Information, die direkt im Systembaustein vorhanden ist, kann auch leicht rückverfolgt werden.

Ein wichtiger praktischer Aspekt der Selbstdokumentation ist, dass man mit solchen Informationen auch zusammenhängende Dokumente, die auf Basis von Code und weiterer Information erstellt werden müssen, generieren kann. Beispielsweise die HTML-Beschreibung einer API kann durch eine Reihe von Werkzeugen, wie beispielsweise JavaDoc, automatisch generiert werden.

6.1.10 Inkrementalitätsprinzip

Ein erster Architektur-Entwurf wie auch Veränderungen einer bestehenden Architektur sollten möglichst inkrementell umgesetzt werden. Dies hängt damit zusammen, dass Software-Architekturen oft hochkomplex sind. Daher scheitert ein Versuch oft, ein ganzes System direkt komplett zu entwerfen, beispielsweise weil auf nebensächliche Aspekte zu viel Wert gelegt wurde oder weil wichtige Aspekte übersehen wurden.

Zur Idee der Inkrementalität

Solche Situationen entstehen zum Beispiel, weil software-technisch ausgebildete Architekten und Entwickler oft nicht dieselbe Sprache sprechen wie Domänenexperten und weil beide Seiten oft manche Dinge für selbstverständlich erachten, die einem Nicht-Experten in dem jeweiligen Bereich nicht unmittelbar klar sind.

Um solche Missverständnisse zu vermeiden, sollte man inkrementell vorgehen und oft ein Feedback einholen. Resultierende Regeln für das Vorgehen sind z. B.:

- > erste Versionen eines Systems frühzeitig auszuliefern,
- > früh die Meinung von realen Nutzern des Systems einzuholen und
- > neue Funktionalität nur schrittweise einzuführen.

Inkrementalität bedeutet also, das Separation-of-Concerns-Prinzip auf die Entwicklungsschritte bei der Systementwicklung anzuwenden.

Schrittweises Wachsen

Schrittweises Wachsen (englisch: *piecemeal growth*) ist eine weiter gehende Variante der Inkrementalität, die von dem Architekten Christopher Alexander beschrieben wird [Alexander 1977] und auch auf Software-Architekturen angewendet werden kann. Die Idee ist, eine Architektur Schritt für Schritt wachsen zu lassen. Nach jedem Schritt folgt eine Beurteilung, die eine Entscheidung, was als Nächstes unternommen werden soll, nach sich zieht. Das heißt, es gibt keine oder nur eine sehr geringe Vorausplanung. Schrittweises Wachsen findet man im Software-Architektur-Zusammenhang z. B. in den Konzepten Refactoring und eXtreme Programming wieder.

Prototyping

Eine weitere Variante der Inkrementalität ist das sogenannte Prototyping. Oft macht es Sinn, zunächst einfache Prototypen zu entwickeln, bevor man an die Entwicklung eines Produktes herangeht, um die Problemstellung näher kennenzulernen. Manchmal kann man solche Prototypen in Produkte verwandeln; manchmal ist es sinnvoller, den Prototyp wegzwerfen und von neuem zu beginnen. Trotzdem kann so ein Prototyp von hohem Wert sein, da der Architekt und die Entwickler dadurch ein Verständnis für die wirklichen Probleme der Domäne entwickeln. Ein möglicher Mittelweg ist ein evolutionärer Prototyp, also ein Prototyp, der inkrementell zu einem Produkt ausgebaut wird.

6.1.11 Weitere Architektur-Prinzipien

Es gibt einige weitere Architektur-Prinzipien allgemeiner Natur, die in diesem Abschnitt knapp zusammengefasst werden sollen:

- > *Bezug zu Anwendungsfällen:* Eine Architektur soll nicht „auf der grünen Wiese“ entstehen, sondern sich im Entwurf an den relevanten Anwendungsfällen orientieren. Damit wird erreicht, dass eine Architektur nicht über das angestrebte und gewünschte Ziel des Systems hinausschießt.
- > *Vermeidung überflüssiger Komplexität:* Auch bei der Architektur gilt: „Weniger ist mehr.“ Unnötig komplexe Architekturen sind meist fehleranfällig und werden nur unzureichend verstanden.
- > *Konsistenz:* Eine Architektur sollte durchgängig einem einheitlichen Satz von Regeln folgen: Namensgebung, Kommunikation der System-Bausteine, Struktur der Schnittstellen, Aufbau der Dokumentation etc. Dieses Prinzip erleichtert die Entwicklung, das Verständnis und die Umsetzung einer Architektur.
- > *Konventionen statt Konfigurationen:* Es werden eher sinnvolle Standardannahmen gemacht und nur notwendige Anpassungen müssen

konfiguriert werden. Somit kommen Entwickler in der Regel schnell zu einem ersten Ergebnis, das man dann schrittweise an die eigenen Bedürfnisse anpassen kann.

6.1.12 Zusammenfassung

- > Es gibt allgemeine Prinzipien, die man beim Entwurf einer Software-Architektur beachten sollte.
- > Es gibt zwei Hauptprobleme, die für so gut wie alle Architektur-Prinzipien eine Rolle spielen: die Reduktion der Komplexität einer Architektur und die Erhöhung der Flexibilität (oder Änderbarkeit) einer Architektur.
- > Man kann die Prinzipien in einem System miteinander in Verbindung setzen.
- > Das Prinzip der losen Kopplung ist ein zentrales Prinzip und besagt, dass die Kopplung zwischen Systembausteinen möglichst niedrig gehalten werden soll.
- > Kohäsion ist ein Maß für die Abhängigkeiten innerhalb eines Systembausteins. Das Prinzip der hohen Kohäsion besagt, dass diese Kohäsion innerhalb eines Systembausteins möglichst hoch sein soll.
- > Das Prinzip des Entwurfs für Veränderung besagt, dass man versuchen soll, die Architektur so zu entwerfen, dass man leicht mit den wahrscheinlichen Änderungen eines Software-Systems umgehen kann.
- > Das Separation-of-Concerns-Prinzip besagt, dass man verschiedene Aspekte eines Problems voneinander trennen soll und jedes dieser Teilprobleme für sich behandeln soll.
- > Das Information-Hiding-Prinzip besagt, dass man einem Klienten nur den wirklich notwendigen Teilausschnitt der gesamten Information zeigt, der für die Aufgabe des Klienten gebraucht wird und alle restliche Information verbirgt.
- > Abstraktionsprinzipien wenden Abstraktionen an. Abstraktion bedeutet, ein komplexes Problem dadurch verständlicher zu machen, indem man wichtige Aspekte identifiziert und die unwichtigen Details vernachlässigt.
- > Das Modularitätsprinzip besagt, dass die Architektur aus wohl definierten Systembausteinen bestehen sollte, deren funktionalen Verantwortlichkeiten klar abgegrenzt sind.
- > Neben diesen zentralen Prinzipien gibt es eine Reihe speziellerer Prinzipien, wie z. B. das Rückverfolgbarkeitsprinzip, das Selbstdokumentationsprinzip, das Inkrementalitätsprinzip etc.

Zusammenfassung: Architektur-Prinzipien

6.2 Grundlegende architektonische Konzepte

In diesem Abschnitt werden wesentliche Konzepte besprochen, die ein Architekt heute einsetzt, um Architekturen umzusetzen. Abbildung 6.2-1 gibt einen Überblick. Der Abschnitt startet mit einfachen prozeduralen Ansätzen und kommt dann Schritt für Schritt zu weiter gehenden Ansätzen, wie Aspektorientierung, Komponentenorientierung und modellgetriebener Entwicklung. Ein Leser, der einige dieser Gebiete schon kennt, mag diese Teile überspringen. Hier wird lediglich ein knapper Überblick der Themen aus der Architektur-Sicht vermittelt, um die Begriffe zu erklären, die in der Folge verwendet werden. Für eine vollständige Einführung in die Gebiete sei auf die weiterführende Literatur verwiesen.

Die in Abbildung 6.2-1 dargestellten Architektur-Mittel sollen zunächst – zur Vereinfachung – aus der Perspektive, in der ein System neu erstellt wird, präsentiert werden. Abschnitt 6.2.9 ergänzt diese Perspektive dann um die Wartungsperspektive. Der Abschnitt 6.2.9 setzt sich also mit architekturverbessernden Maßnahmen an existierenden Systemen auseinander.

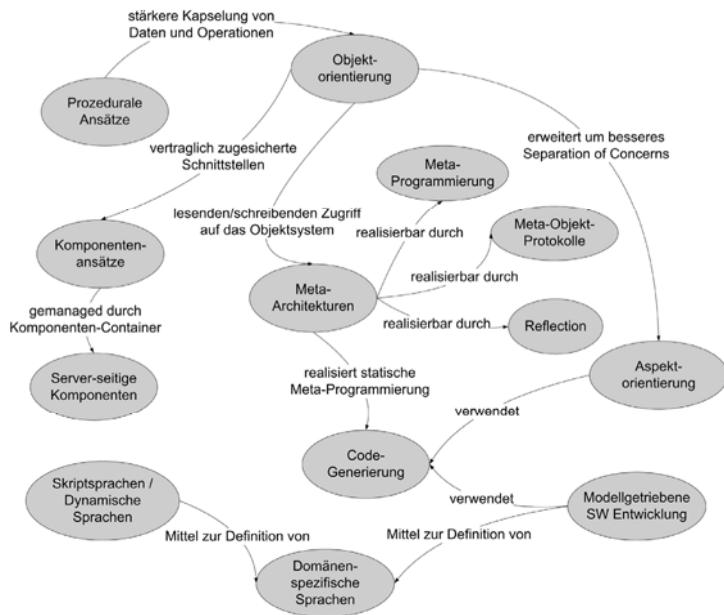


Abb. 6.2-1: Übersicht über die grundlegenden architektonischen Konzepte.

6.2.1 Prozedurale Ansätze

Prozeduren (englisch: *procedure*) sind ein althergebrachtes und noch immer weitverbreitetes Mittel der Strukturierung von Architekturen. Sie erlauben die Zerlegung eines komplexen Algorithmus in wiederverwendbare Teilarithmen. Durch Prozeduren wird das Separation-of-Concerns-Prinzip umgesetzt, denn Prozeduren erlauben die Zerteilung eines komplexen, algorithmischen Problems in einfache Teilprobleme.

Überblick Prozeduren

Prozeduren werden heute noch in Systemen, die auf prozeduralen Programmiersprachen wie C und Cobol basieren, sowie in prozeduralen verteilten Systemen eingesetzt. Viele objektorientierte Systeme erlauben auch prozedurale Abstraktionen (wie statische Methoden in Java), denn manchmal sind diese besser geeignet, ein System zu strukturieren als Objekte und Klassen.

Es sei darauf hingewiesen, dass viele synonyme Begriffe für den Begriff Prozedur gebräuchlich sind, wie Unterprogramm (englisch: *sub program*), Funktion (englisch: *function*), Routine (englisch: *routine*) oder Operation (englisch: *operation*).

Eine Prozedurdefinition besteht aus mehreren Teilen:

- Der *Prozedurname* ist ein Bezeichner, mit dem die Prozedur von außen aufgerufen werden kann.
- Die *Prozedurparameter* sind eine Anzahl von Werten oder Referenzen, die der Prozedur übergeben werden können. Die Prozedurdefinition umfasst formale Parameter mit jeweils einem Parameterotyp und einem Parameternamen. Beim Aufruf werden aktuelle Werte für diese Parameter übergeben, die den Typen der Parameterdefinition entsprechen müssen. Man unterscheidet Call-by-Reference-Parameter und Call-by-Value-Parameter. Call-by-Reference-Parameter bekommen eine Referenz auf die übergebenen Daten, was bedeutet, dass eine Änderung der Daten in der Prozedur implizit eine Änderung im aufrufenden Gültigkeitsbereich bewirkt. Call-by-Value-Parameter hingegen arbeiten mit einer Kopie der Daten, was bedeutet, dass Änderungen in der Prozedur keinen Einfluss auf die Daten im aufrufenden Gültigkeitsbereich haben.
- Der *Prozedurrückgabetyp* dient der Rückgabe von Ergebnisdaten von der Prozedur an ihre Klienten.
- Der *Prozedurkörper* spezifiziert den Algorithmus, der beim Aufruf der Prozedur durchgeführt werden soll. Hier werden die beim Aufruf angegeben aktuellen Parameter verwendet (das heißt, die formalen

Aufbau einer Prozedur

Parameter werden durch die aktuellen Werte ersetzt). Das Ergebnis der Prozedur muss dem Rückgabetypr entsprechen.

Abbildung 6.2-2 veranschaulicht diese Begrifflichkeiten am Beispiel einer Prozedurdefinition und Abbildung 6.2-3 zeigt einen Aufruf dieser Prozedur.

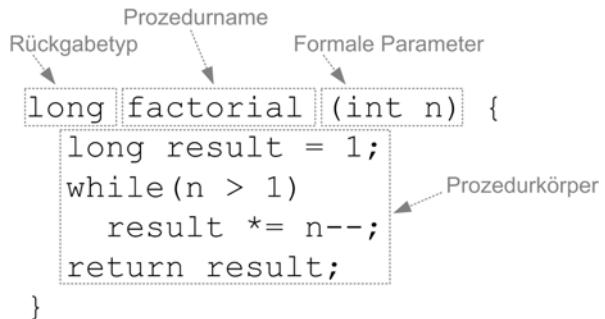


Abb. 6.2-2: Beispiel einer Prozedurdefinition.

Die ersten drei Elemente der Prozedurdefinition, also Name, Rückgabewert und Parameter, bilden die *Schnittstelle* der Prozedur, auch *Prozedur-Signatur* genannt. Auf Basis der Schnittstelle ist es möglich, Prozeduren mit anderen Daten oder in anderem Kontext wiederzuverwenden.



Abb. 6.2-3: Beispiel eines Prozederaufrufs.

Prozeduren und Architektur-Prinzipien

Meist teilen sich die Prozeduren einen gemeinsamen (oft globalen) Datenbereich mit anderen Prozeduren. Diese Datenbereiche werden oft als Zwischenspeicher und zum Zweck der Kommunikation mit anderen Prozeduren genutzt. Falls diese Datenbereiche als globale Strukturen adressiert werden, werden Prozeduren schnell voneinander abhängig und es wird schwierig, sie in anderen Zusammenhängen wiederzuverwenden. Hier werden also schnell die Prinzipien der Modularisierung und des Information Hidings verletzt, was dazu führt, dass man, um eine Prozedur zu ändern, auch andere Prozeduren ändern muss.

Prozedurale Abstraktionen tendieren dazu, Architekturen mit großen Ansammlungen von Prozeduren und dazugehörigen Datenstrukturen in Bibliotheken (englisch: *library*) hervorzubringen. Die Schnittstellen sind durch die Signaturen der in der Bibliothek enthaltenen Prozeduren gegeben. Viele prozedurale Programmiersprachen erlauben es, auch auf der Quelltext-Ebene die prozeduralen Schnittstellen durch sogenannte Header Files anzugeben und gegen diese Schnittstellendefinitionen zu programmieren. Mit diesem Mittel kann man zwar die Schnittstellenabstraktionsprinzipien aus Abschnitt 6.1 umsetzen, doch diese Art der Strukturierung einer Architektur und ihrer Schnittstellen erschwert es oft, änderbare, erweiterbare und wiederverwendbare Strukturen herzustellen. Dies hat den Grund, dass einzelne Prozeduren, die zu einem komplexen System verknüpft werden, häufig komplexe Abhängigkeiten zu anderen Prozeduren und zu den Daten haben.

Jedoch ist es durchaus möglich, die Architektur-Prinzipien aus Abschnitt 6.1 gut in einer prozeduralen Sprache umzusetzen. Es gibt eine Vielzahl erfolgreicher Systeme mit sehr guten prozeduralen Architekturen. Allerdings gibt es auch viele gegenteilige Beispiele. Als Fazit kann gelten: Es macht Mühe und bedarf eines wohl durchdachten Entwurfs, um alle Architektur-Prinzipien gut in einer prozeduralen Programmiersprache umzusetzen.

Man kann beobachten, dass die notwendigen Maßnahmen dazu wiederkehrend sind – dies wird in dem Architekturmuster Object System Layer beschrieben [Goedicke et al. 2000], das zeigt wie man ein Objekt-System innerhalb einer prozeduralen Architektur nachbildet.

Die Beobachtung, dass wiederkehrende Abstraktionen zu guten prozeduralen Systemen führen, haben zur Objektorientierung geführt, die die Prinzipien durch entsprechende sprachunterstützte Abstraktionen einfacher umsetzen lässt („sprachunterstützt“ meint hier sowohl Entwurfs- als auch Programmiersprachen).

6.2.2 Objektorientierung

Die Objektorientierung basiert auf der Idee, die Daten, die eine Reihe von zusammenhängenden Prozeduren gemeinsam bearbeiten, mit diesen Prozeduren zu bündeln. Die Prozeduren, in der Objektorientierung Operationen oder Methoden genannt, können ihre Daten exklusiv bearbeiten – somit versucht die Objektorientierung, das Information-Hiding-Prinzip und die Modularisierung unmittelbar umzusetzen.

Objektorientierte Entwürfe in einer prozeduralen Architektur

Idee der Objektorientierung

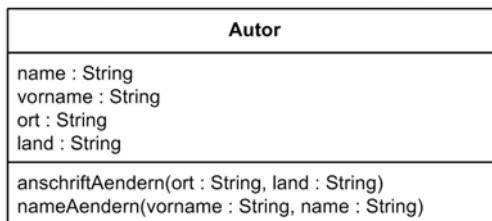
Von der Idee her sollten Objekte in erster Linie Realweltkonzepte abbilden. Beispielsweise kann ein Objekt für einen Autor stehen und dessen Daten wie Name, Anschrift etc. speichern. Es stellt dann auch die Operationen für deren Änderung und Abfrage bereit.

Klassifizierung und Identität

Ein wichtiger Aspekt der Objektorientierung ist die Klassifizierung. Als einfaches Beispiel soll die Modellierung eines Autors dienen, der in einem Verlag Bücher publiziert. Verlage und manche Bücher verfügen nicht nur über einen Autor, sondern über viele. Daher ist hier eine Klassifizierung sinnvoll: Die *Klasse „Autor“* kann somit als Abstraktion für den wiederkehrenden Belang „Autor“ angesehen werden. Eine Klasse wird in der Objektorientierung einmal definiert und kann dann vielfach instanziert werden. Jedes Objekt hat die durch die Klasse spezifizierten Daten exklusiv für sich in einem eigenen Daten- und Namensraum. Das Objekt kann auf die durch seine Klasse definierten Operationen zugreifen.

Ein weiteres, wichtiges Konzept in diesem Zusammenhang ist die Objektidentität. Wenn von einer Klasse eine ganze Reihe von Objekten instanziert werden können, so muss man diese zur Laufzeit unterscheiden können. Deshalb hat jedes Objekt eine eindeutige Objekt-ID, mit der es identifiziert werden kann. Mittels der Objekt-ID kann man dem Objekt Nachrichten schicken, also Operationen des Objekts aufrufen.

Klassendiagramm



Objektdiagramm

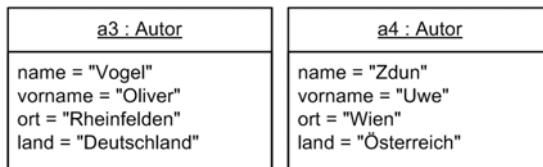


Abb. 6.2-4: Beispiel eines Klassendiagramms (oben) und eines dazugehörigen Objektdiagramms (unten).

Abbildung 6.2-4 zeigt ein UML-Klassendiagramm mit einer Klasse „Autor“ und zwei von dieser Klasse instanzierte Objekte in einem Objektdiagramm. Man sieht, jedes der abgeleiteten Objekte hat seine eigene Identität und seine eigenen Ausprägungen der in der Klasse definierten Daten.

Objektkonzepte bieten in der Regel auch eine Reihe von Beziehungsabstraktionen, mit denen man die möglichen Interaktionen zwischen Objekten näher spezifizieren kann:

- > **Assoziation:** Ein Objekt „kennt“ ein Objekt eines bestimmten anderen Typs; das heißt, in seinen Daten speichert es die Objekt-ID eines anderen Objekts und kann dieses Objekt in der Folge aufrufen. Beispielsweise kann ein Autorobjekt ein oder mehrere Bücher assoziieren, an denen der Autor mitgewirkt hat.
- > **Aggregation:** Ein Objekt ist Teil eines anderen Objekts. Es gibt viele verschiedene Arten von „Ist-Teil-von-Beziehungen“. Z. B. unterscheidet UML zwei Arten von Aggregationsbeziehungen: solche, die eine Lebenszeitverantwortlichkeit des aggregierenden Objekts für die aggregierten Objekte einschließen (UML-Komposition), und solche, die dies nicht tun (UML-Aggregation). Ein Beispiel einer Aggregationsbeziehung ist eine Klasse „Verlag“, die die verlegten Bücher und Autoren aggregiert.
- > **Vererbung:** Eine Klasse *X* ist eine Spezialisierung einer anderen Klasse *Y*. Das heißt, alle Instanzen von *X* können die Eigenschaften von *Y* auch nutzen. Am Beispiel könnte etwa eine weitere Klasse „Lektor“ benötigt werden, die wie „Autor“ auch einen Namen und eine Anschrift verwalten muss. Dann kann man gemeinsame Eigenschaften und Operationen von Lektor und Autor in eine Superklasse „Person“ auslagern, von der beide anderen Klassen erben.
- > **Schnittstellen und abstrakte Klasse:** Eine Klasse kann eine Schnittstelle implementieren oder von einer abstrakten Klasse erben. Eine solche Beziehung dient in der Objektorientierung dazu, Schnittstellen explizit zu machen und die Abstraktionsprinzipien, die in Abschnitt 6.1 eingeführt wurden, zu unterstützen, denn Klientenklassen können sich auf die Schnittstelle verlassen, während die Implementierung wechseln oder sich verändern kann.

Beziehungen

Die genannten Erweiterungen des Beispiels sind in Abbildung 6.2-5 als UML-Klassendiagramm dargestellt.

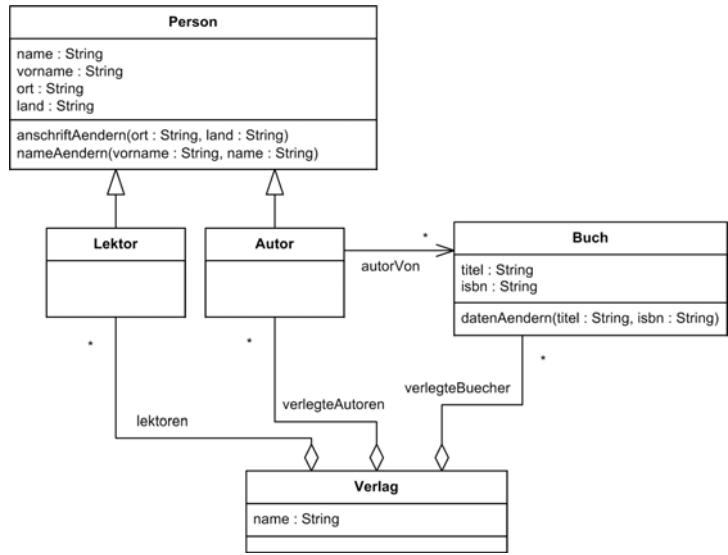


Abb. 6.2-5: Beispiel eines Klassendiagramms mit einigen Beziehungen.

Polymorphie

Das Konzept der Polymorphie (Vielgestaltigkeit) in der Objektorientierung erlaubt es, dass man Objekte und Operationen trotz der Typisierung durch Klassen flexibel verwenden kann.

Operationen sind immer einem bestimmten Typ bzw. einer bestimmten Klasse zugeordnet. Die Polymorphie sorgt dafür, dass je nach Klasse eines bestimmten Objekts die zugehörige Operation ausgeführt wird. Dies kann man mit dem Schaubild in Abbildung 6.2-6 veranschaulichen: Polymorphie erlaubt es, eine „Steckerkompatibilität“ auf Basis von Schnittstellen zu erreichen. Die Realisierung des Steckers ist für die Bausteine unwichtig.

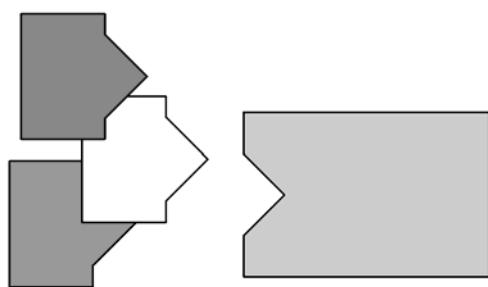


Abb. 6.2-6: „Steckerkompatibilität“ durch Polymorphie.

Man unterscheidet zwischen:

- > *Kompilationszeit-Polymorphie („statisches“ Binden):* Bei Kompilationszeit-Polymorphie bestimmt der statische Typ des Objekts die aufgerufene Operation. Dies funktioniert nur bei Objektkonzepten mit statischer Typisierung.
- > *Laufzeit-Polymorphie („dynamisches“ oder „spätes“ Binden):* Laufzeit-Polymorphie hingegen bestimmt die Klasse des Objekts zur Laufzeit und ruft dann die Operation auf dieser Klasse auf. Die gefundene Klasse muss nicht unbedingt identisch mit dem statischen Typ sein. Das heißt, es hängt vom eingesteckten Objekt ab, welche Operation ausgeführt wird. Diese Operation wird dynamisch auf Basis des Objekttyps zur Laufzeit bestimmt. Dies wird in Abbildung 6.2-7 veranschaulicht.

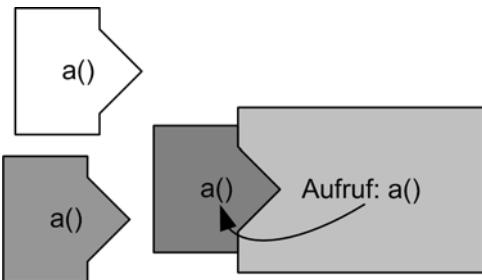


Abb. 6.2-7: Laufzeit-Polymorphie.

Laufzeit-Polymorphie ist einer der wichtigsten Bestandteile der Objektorientierung. Am Beispiel von oben kann man dies leicht verdeutlichen. Angenommen man will einfach auf alle im Verlag beschäftigten Personen zugreifen. Dies könnte ohne Polymorphie nicht in allgemeiner Form definiert werden, sondern man muss zunächst auf die Autoren zugreifen, dann auf die Lektoren etc. Nicht nur, dass bei einer solchen Lösung keine Wiederverwendung möglich wäre, überdies könnte man den Code ohne Änderungen nicht für zukünftige Erweiterungen, wie der Ergänzung anderer Personentypen, nutzen.

Somit wäre es schwer, die Prinzipien lose Kopplung und Entwurf für Veränderung mit objektorientierten Klassenkonzepten umzusetzen. Laufzeit-Polymorphie löst dieses Problem, indem man auf Basis von Superklassen und Schnittstellen die Architektur entwerfen kann und das Laufzeitsystem die konkreten Objekttypen spät bindet.

Objektorientierte Architekturen sind aus architektonischer Hinsicht prozeduralen Architekturen nicht unähnlich, doch durch die zusätzli-

Objektorientierung und Architektur-Prinzipien

chen Abstraktionen bietet die Objektorientierung eine bessere Unterstützung der Modularisierung. Wichtige Abstraktionsprinzipien werden sprachunterstützt (wiederum sowohl in Programmier- als auch Entwurfssprachen). Besseres Information Hiding wird durch Kapselung von Daten und zugehörigen Operationen erreicht.

Dies alles ermöglicht es, die Prinzipien aus Abschnitt 6.1 einfacher umzusetzen. Aber trotzdem ist der Architekt auch bei der Objektorientierung gefragt, um ein geeignetes objektorientiertes Modell zu entwerfen, denn die einfache Gleichung „objektorientierte Architektur = gute Architektur“ geht nicht auf. Z. B. das wichtige Ziel „Wiederverwendung“ in objektorientierten Ansätzen benötigt weit mehr als Klassen und Objekte. Die Wiederverwendung wird durch Objektorientierung in der Tat nur ansatzweise unterstützt. Daher muss auch die Objektorientierung richtig verstanden werden und ihre Techniken müssen richtig angewendet werden.

Framework

Insbesondere Ansätze zur Beherrschung komplexerer objektorientierter Architekturen sind wichtig. Ein zentraler Ansatz in diesem Zusammenhang sind objektorientierte *Frameworks* (deutsch: *Rahmenwerk*) [Johnson und Foote 1988].

Ein Framework ist ein teilweise komplettes Software-System (oder Subsystem), das instanziert werden soll. Somit definiert ein Framework eine Architektur für eine Familie von (Sub-)Systemen und stellt die elementaren Bausteine dieser Architektur zur Verfügung. Das Framework definiert auch die Stellen, wo Adaptationen des Frameworks vorkommen können (sogenannte „Hot Spots“ [Pree 1995]). Frameworks setzen dabei stark auf Inversion of Control (beziehungsweise das in Abschnitt 6.1 beschriebene Hollywood-Prinzip): Anstatt die Anwendung den Kontrollfluss steuern zu lassen, wird die Steuerung von Teilen des Systems dem Framework überlassen und nur konfigurierbare Teile werden über Hot Spots adaptiert.

In objektorientierten Systemen besteht das Framework aus Klassen, die instanziert werden sollen, und abstrakten Klassen/Schnittstellen, die konkretisiert werden sollen. Aus den gegebenen Klassen kann man durch Vererbung speziellere Klassen abbilden. Die späte Bindung ist insbesondere für Frameworks von großer Bedeutung, denn dadurch kann man das Framework auf allgemeinen Typen basierend definieren und trotzdem z. B. durch Vererbung abgeleitete Klassen verwenden.

Noch ein Hinweis: Frameworks sind nicht immer objektorientiert und viele objektorientierte Frameworks haben auch nicht objektorientierte Teile. Trotzdem sollen Frameworks hier im Kontext der Objektorientierung Erwähnung finden, da der Begriff „Framework“ hauptsächlich in diesem Bereich geprägt wurde.

Ein Nachteil der vielen Vorgaben, die Frameworks machen, ist, dass sie den Architekten und Entwickler mehr oder weniger in seinen (architektonischen) Freiheiten einschränken. In letzter Zeit versuchen einige Frameworks, wie beispielsweise Spring, minimal-invasive Lösungen anzubieten, wie Dependency Injection (siehe auch Abschnitt 6.1), um diesen Nachteil zum Teil aufzuheben.

Minimal-invasive Frameworks

Neben Frameworks gibt es eine Reihe anderer Ansätze, die komplexere objektorientierte Architekturen beherrschbar machen sollen. Einige davon werden noch im Laufe dieses Buches angesprochen, wie z. B. objektorientierte Entwurfsmuster in Abschnitt 6.3. Die Entwurfsmuster sind bei der Entwicklung von Frameworks wichtig und einige Muster kommen häufig in Frameworks zum Einsatz, wie z. B. Template Method [Gamma et al. 1995].

Weitere Ansätze

Es gibt eine Vielzahl von objektorientierten Abstraktionen oder Abstraktionen, die aus der Objektorientierung entstanden sind. Weiter unten werden noch Komponenten, Meta-Objekte und Aspekte als Beispiele von Weiterentwicklungen der Objektorientierung besprochen.

6.2.3 Komponentenorientierung

Komponenten sollen wiederverwendbare, in sich geschlossene Bausteine einer Architektur sein. Komponentenorientierung entstand aus der Problematik, dass Objekte zwar das Modularitätsprinzip umsetzen, aber als wiederverwendbare Einheiten oft zu klein sind.

Zum Konzept „Komponente“

Überdies wünscht man sich oft weiter gehende Eigenschaften von einem wiederverwendbaren Systembaustein. Z. B. kann man sich zusätzliche nicht-fachliche Anforderungen vorstellen, wie verschiedene Aktivierungs- oder Passivierungsstrategien oder die konkurrente Bereitstellung mehrerer gleicher Komponenteninstanzen zur Erhöhung der Skalierbarkeit. Solche nicht-fachlichen Anforderungen sind jedoch nicht Teil des objektorientierten Ansatzes.

Zu guter Letzt sind in der Praxis oft Wiederverwendungseinheiten zu finden, die nicht den Ideen der Objektorientierung entsprechen, zum Beispiel durch das Vorhandensein von Alt-Systemen. Hier findet man beispielsweise große, prozedurale Bibliotheken, die als Wiederverwendungseinheit gesehen werden und somit in praktischen Architekturen konzeptuell eingebettet werden müssen.

Definition von Komponenten

Komponenten sind ein recht allgemeines Konzept, das versucht, diese und ähnliche Probleme zu lösen. Es gibt viele Definitionen von Komponenten. Hier soll die Definition von Clemens Szyperski verwendet werden [Szyperski 1998]:

Eine Komponente ist eine Kompositionseinheit mit vertraglich spezifizierten Schnittstellen, die nur explizite Abhängigkeiten zu ihrem Kontext hat. Eine Software-Komponente kann unabhängig eingesetzt werden und sie kann durch Dritte komponiert werden.

Schaut man etwas genauer auf diese Definition, so sieht man, dass die Definition recht breit ist und viele Dinge meint. Der Begriff Kompositionseinheit besagt schon, dass der Hauptzweck einer Komponente das Zusammenfügen und die Interaktion mit anderen Komponenten ist. Um dies durchzuführen, verfügt eine Komponente über eine oder mehrere Schnittstellen, die als Vertrag zwischen der Komponente und ihrer Umgebung fungieren. Die Schnittstellen der Komponente definieren klar, welche Leistungen die Komponente erbringt. Die Komponente hat keine impliziten Abhängigkeiten, sondern jedes Element der Architektur, das die Komponente benötigt, ist auch durch sie spezifiziert – insbesondere sind dies die anderen Komponenten, die von einer Komponente benötigt werden.

Eine Komponente ist in sich selbst geschlossen, daher kann sie unabhängig von einer speziellen Umgebung eingesetzt werden. Insbesondere heißt dies, dass die Komponente nicht verändert werden muss, um sie einzusetzen, und dass ihr Einsatz auch keine Änderung an anderen Komponenten nach sich zieht.

Ein wichtiger Punkt, der durch diese Eigenschaften von Komponenten ermöglicht wird, ist, dass eine Komponente normalerweise nicht nur durch diejenigen eingesetzt wird, die sie erstellt haben, sondern auch von Dritten.

Die Definition umfasst bewusst viele verschiedene Konzepte, wie beispielsweise Subsysteme, DLLs, JavaBeans, ActiveX Controls, JEE-Komponenten, .NET-Komponenten, CORBA-Komponenten (CCM), Komponentenansätze von Skriptsprachen (z. B. Tcl, Python, Perl) und viele andere. Natürlich setzen diese Ansätze das Komponentenkonzept unterschiedlich stark um. In Abschnitt 6.4.10 soll insbesondere ein genauerer Blick auf Komponentenplattformen geworfen werden, wie sie durch EJB, .NET und CCM umgesetzt werden – da diese Basisarchitektur eine große Bedeutung in der Praxis hat.

Die folgende Abbildung 6.2-8 zeigt ein Beispiel für die Komponentenmodellierung mit UML. Eine Komponente „Kurs“ bietet eine Schnittstelle für eine Komponente „Student“ und eine Schnittstelle für eine Komponente „Manager“ an. Diese beiden Komponenten benötigen die jeweilige Schnittstelle. Dies wird durch die sogenannte Ball-and-Socket-Notation dargestellt. „Manager“ hat noch eine weitere Beziehung zu der Komponente „Büro“ mit dem Stereotyp „uses“. Dies bedeutet, dass die Bürokomponente zur vollständigen Implementierung der Managerkomponente benötigt wird.

Beispiel zur Komponentenmodellierung

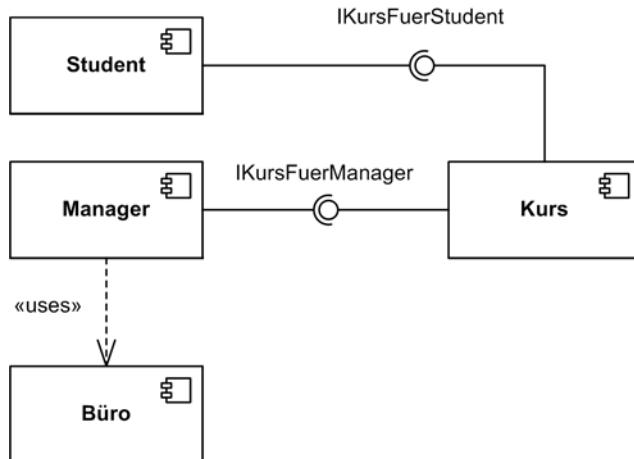


Abb. 6.2-8: Beispiel für ein UML-Komponentendiagramm.

In erster Linie setzen Komponentenarchitekturen das Prinzip der Modularisierung um. Komponenten sind oft größer granular und/oder von ihrer Umgebung unabhängiger (das heißt loser gekoppelt) als Objekte. Ein weiteres Prinzip, das viele Komponentenarchitekturen stärker als Objekte umsetzen, ist Separation of Concerns: Durch die Laufzeitumge-

Umsetzung von Architektur-Prinzipien

Idee der Meta-Programmierung

bung werden technische und funktionale Belange getrennt und in verschiedenen Bausteinen gekapselt. Die Trennung dieser Belange ermöglicht, dass sie unabhängig voneinander weiterentwickelt werden können und dass die technischen Belange in verschiedenen Systemen wieder verwendet werden können.

6.2.4 Meta-Architekturen und Reflection

Vom Meta-Programm zum Meta-Objekt-Protokoll

Die Idee der Meta-Programmierung ist es, durch eine zusätzliche Abstraktionsebene in Software-Systemen eine höhere Flexibilität und Kontrolle zu erreichen. In vielen Programmiersprachen wird zwischen dem Programm als ausführbare Anweisungen und den Daten, mit denen das Programm operiert, unterschieden. In der Tat sind aber die Programme selbst auch Daten. Jedoch kennt das Programm zwar seine Daten, aber sich selbst nicht. Meta-Programmierung ändert dieses Paradigma ab und erlaubt dem Programm den Zugriff auf sich selbst. Meta-Programmierung heißt also, dass Programme auch als Daten behandelt werden.

Lesender und schreibender Zugriff

Programme, die andere Programme als Daten verwenden, werden an sich sehr häufig eingesetzt. Z. B. behandeln eine virtuelle Maschine und ein Interpreter das ausgeführte Programm als Daten. Zur Meta-Programmierung fehlt nur noch ein Schritt: Das Programm selbst muss auf das Meta-Programm zugreifen können. Die entsprechende Schnittstelle wird als Meta-Objekt-Protokoll (MOP) bezeichnet [Kiczales et al. 1991].

Man kann unterscheiden zwischen lesendem und schreibendem Zugriff eines Programms auf sich selbst bzw. seine Repräsentation im Meta-Programm.

Beim lesenden Zugriff, auch Introspektion oder Reflection genannt, kann das Programm beispielsweise Typinformationen, Informationen über Klassen (Attribute und Operationen) und Vererbungshierarchien abfragen. Oft ist es auch möglich, dynamisch Methoden aufzurufen und Klassen zu instanziieren. Java Reflection ist ein bekanntes Beispiel lesender Meta-Programmierung.

Wenn ein Programm schreibenden Zugriff auf sich selbst hat, kann es zum Beispiel Klassendefinitionen ändern, Klassen hinzufügen und entfernen oder Klassenhierarchien ändern. Beispiele für Sprachen, die schreibenden Zugriff zur Laufzeit erlauben, sind CLOS, Smalltalk und Tcl.

Einige Sprache verfügen über Makro-Sprachen, wie Lisp Macros, die statische Meta-Programmierung unterstützen – also ein Ändern des Programms vor der Laufzeit.

Bei einigen dieser Sprachen ist ferner zu beachten, dass sie es auch erlauben, die Definition von Sprachelementen zu modifizieren.

Dies alles sind sehr mächtige Sprachinstrumente. Von manchen Entwicklern werden sie aber auch als schwer verständlich oder komplex angesehen, insbesondere weil man die Veränderungen durch die Meta-Programmierung immer gut verstanden haben muss, um das eigentliche Programm zu verstehen. Mit anderen Worten, Meta-Programmierung kann sehr viel bringen, erfordert aber auch eine hohe Disziplin vom Entwickler.

Lesende Reflection-Techniken haben sich bereits im Mainstream durchgesetzt, wie Java Reflection zeigt. Programmverändernde Techniken sind nicht so weit verbreitet.

Oft findet man auch „selbst gebaute“ Meta-Architekturen – also Meta-Architekturen, die nicht von einer Sprache oder Laufzeitumgebung unterstützt werden. Zum Beispiel setzen dies viele Analyse-Muster ein (vgl. z. B. [Fowler 1996]), insbesondere um dynamische Typisierung zu simulieren. Das Reflection-Muster [Buschmann et al. 1996] zeigt allgemein, wie man eine Meta-Architektur selbst bauen kann. Eine solche Architektur ist beispielhaft in Abbildung 6.2-9 dargestellt.

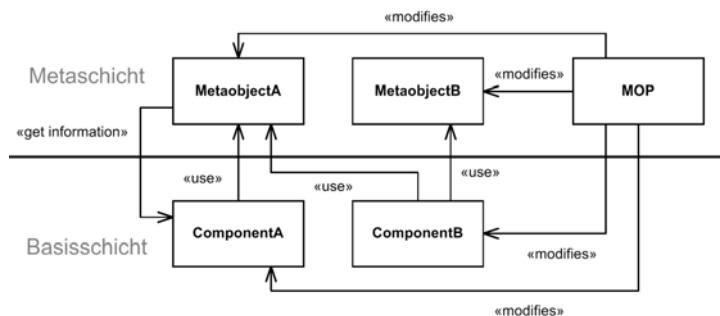


Abb. 6.2-9: Beispiel einer „selbst gebauten“ Meta-Architektur nach dem Reflection-Muster.

Praktischer Einsatz von Meta-Architekturen und Reflection

6.2.5 Generative Erzeugung von Systembausteinen

Bei einem Blick über den Tellerrand der Software-Entwicklung hinaus lässt sich erkennen, dass in anderen Ingenieursdisziplinen eine Automatisierung der Abläufe immer dann eingesetzt wird, wenn bestimmte Arbeiten wiederkehrend auf ähnliche Art und Weise zu erbringen sind. Der Einsatz generativer Techniken in der Software-Entwicklung verfolgt das Ziel, den Automatisierungsgrad bei der Erstellung von Software anderen Ingenieurdisziplinen anzupassen. Im Folgenden wird der Einsatz generativer Techniken anhand eines Beispiels motiviert, bevor im weiteren Verlauf dieses Abschnitts die Funktionsweise eines Generators auf einfachste Art und Weise vorgestellt wird. Weiterhin werden verschiedene Generierungstechniken skizziert, bevor die praktischen Einsatzmöglichkeiten von Generatoren kurz diskutiert werden.

Motivation

In der Software-Entwicklung ist es oftmals notwendig, Varianten eines Software-Systems zu realisieren, welche sich in nur wenigen Details unterscheiden. Die Spezifika der einzelnen Varianten können dabei sowohl fachlich als auch technisch motiviert sein. Ein Beispieldaten um einer fachlich motivierten Variantenbildung sind spezielle Kundenanforderungen an ein Produkt, das für mehrere Kunden gleichzeitig erstellt wird. Ein typisches Beispiel einer technisch motivierten Variantenbildung stellt die Realisierung einer Architektur auf verschiedenen Komponentenplattformen (siehe Abschnitt 6.7.5) dar. Ein Hauptziel der Software-Entwicklung ist die Erzielung eines möglichst hohen Anteils an wiederverwendbaren Systembausteinen. Im Zentrum der hier kurz charakterisierten Variantenbildung steht daher stets die Anpassung dedizierter Teile einer Software an spezielle Anforderungen oder spezifische fachliche bzw. technische Rahmenbedingungen. Eine solche Anpassung kann dynamisch zur Laufzeit oder statisch zur Übersetzungszeit erfolgen. Möglichkeiten einer dynamischen Anpassung sind beispielsweise die deskriptive Anpassung des Systems über Konfigurationsdateien oder das in Abschnitt 6.2.4 vorgestellte Konzept der Metaprogrammierung. Eine statische Variante stellt das im Folgenden kurz skizzierte Konzept der generativen Erzeugung von Systembausteinen dar.

Funktionsweise eines Generators

Die generative Erzeugung von Systembausteinen und dasitive Programmieren [Czarnecki und Eisenecker 2000] verfolgen das Ziel, einen zu anderen Ingenieurdisziplinen ähnlichen Automatisierungsgrad auch in der Software-Entwicklung zu erreichen. Im Kern des generativen Ansatzes stehen zwei wesentliche Schritte. Zunächst muss der Fokus vom Entwurf eines einzelnen Systems hin zum Entwurf einer ganzen Familie

von Systemen beziehungsweise vom Entwurf eines einzelnen Anwendungsbausteins hin zu einer Menge ähnlich aufgebauter Bausteine verschoben worden. Entscheidendes Kriterium ist dabei die Herausfaktorisierung der gemeinsamen, schematischen Anteile der Anwendungen einer solchen Systemfamilie beziehungsweise der ähnlich strukturierten Systembausteine. In einem zweiten Schritt müssen geeignete Techniken eingesetzt werden, um diese schematischen Anteile automatisiert zu erzeugen. Darunter fallen die Auswahl von Mitteln zur präzisen maschinenlesbaren Spezifikation der zu generierenden Systemanteile auf einem möglichst hohen Abstraktionsniveau (wie beispielsweise die Bildung von Modellen, siehe Abschnitt 6.6) sowie der Einsatz von Generatoren, welche die abstrakten Spezifikationen (*Input*) einlesen und daraus letztlich die zu generierenden Systembausteine (*Output*) automatisiert erzeugen (siehe Abbildung 6.2-10).

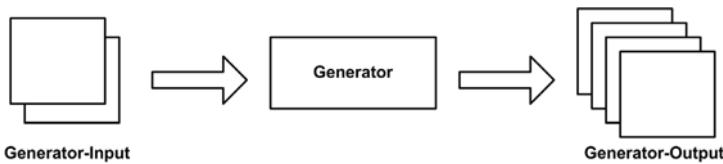


Abb. 6.2-10: Funktionsweise eines Generators: Generator-Input, Generator und Generator-Output.

Um die oben beschriebene Generierung von Systembausteinen aus abstrakten Spezifikationen zu realisieren, existieren verschiedene Generierungstechniken, welche hier nicht im Detail behandelt werden sollen. Der interessierte Leser sei an dieser Stelle auf [Czarnecki und Eisenecker 2000] verwiesen. Im Folgenden werden die wichtigsten Generierungstechniken kurz skizziert.

Generierungstechniken

Die wohl am weitesten verbreitete Generierungstechnik manifestiert sich in den *template-basierten Generatoren*. Unter einem Template versteht man in diesem Zusammenhang eine (meist textbasierte) Vorlage, die im Wesentlichen aus zwei Teilen besteht: Ein Teil ermöglicht es, auf den (ebenfalls meist textbasierten) Generator-Input zuzugreifen. Über Muster (englisch: *patterns*) kann definiert werden, wann das Template angewendet werden soll. Der zweite Teil besteht aus einer Reihe von Regeln, welche die Generator-Ausgabe, das heißt die Manipulation der Vorlage, in Abhängigkeit vom Generator-Input steuern. Prominente Vertreter stellen beispielsweise die Java Emitter Templates (JET) [Popma 2004a, Popma 2004b], das Velocity-Projekt [Apache 2008a] oder die Transformationssprache XSLT (XSL Transformations) [W3C 2006] dar. XSLT ist Teil der Extensible Stylesheet Language (XSL) und dient zur

Template-basierte Generatoren

Transformation von XML-Dokumenten. Das resultierende Dokument entspricht meist der XML-Syntax, es können aber auch andere Textdateien und sogar Binärdateien erstellt werden. Ein XSLT-Template besitzt ein auf XPath (XML Path Language) [W3C 1999] basierendes Pattern, das beschreibt, für welche Knoten des Syntaxbaums des XML-Quelldokuments es gilt, und einen Inhalt, der bestimmt, wie das Template seinen Teil des Syntaxbaumes des XML-Zieldokuments erzeugen soll. Eine Anwendung der oben beschriebenen template-basierten Generierung findet sich in der Erzeugung von PDF-Dokumenten mittels XSL Formatting Objects (XSL-FO) [W3C 2006] und einem geeigneten Konverter, beispielsweise dem Apache Formatting Objects Processor (FOP) [Apache 2008b].

API-basierte Generatoren

Eine Alternative zur oben beschriebenen Generierung von PDF-Dokumenten stellt das iText-Projekt [iText 2008] dar. Im Rahmen des Projekts wird eine Java-Klassenbibliothek zur Erzeugung von PDF-Dokumenten zur Verfügung gestellt. Der gesamte Aufbau des zu generierenden Dokuments kann somit über ein API (englisch: Application Programming Interface) beschrieben werden. Solche Generatoren werden daher als *API-basierte Generatoren* bezeichnet. Selbstverständlich beschränkt sich der Einsatz API-basierter Generatoren nicht nur auf die Generierung von PDF-Dokumenten, sondern findet in vielen Szenarien seine Anwendung.

Frame-Technologie und Frame-Prozessoren

Wird das aus der objektorientierten Programmierung bekannte Konzept der Instanziierung von Klassen (siehe Abschnitt 6.2.2) von der Laufzeit auf die Übersetzungszeit vorgezogen, so gelangt man zur sogenannten *Frame-Technologie*. Frames bilden das Pendant zum objektorientierten Konzept der Klasse und fungieren als Templates für zu generierende Quelltextfragmente. Frames können dabei durch den *Frame-Prozessor* (beliebig oft) instanziert werden. Variable Anteile der Frames (Slots) werden bei der Instanziierung an konkrete Werte gebunden. Durch entsprechende Anweisungen an den Frame-Prozessor kann in einem weiteren Schritt aus den Frame-Instanzen konkreter Quelltext generiert werden. Die Werte, welche die Slots bei der Instanziierung annehmen können, sind dabei im Gegensatz zur einfachen template-basierten Generierungstechnik nicht nur Zeichenketten (Strings), sondern können wiederum Frame-Instanzen darstellen, was die Bildung ganzer Hierarchien von Frame-Instanzen ermöglicht. Ein Repräsentant dieser Generierungstechnik ist der Frame-Prozessor ANGIE [DSTG 2008].

Sind im regulären Quelltext Konstrukte enthalten, die während des Übersetzungsvorgangs (Komplilierung) weiteren Quell-, Zwischen- oder Maschinencode erzeugen, so spricht man von *Inline-Generierung*. Als Beispiel dieser Generierungstechnik sind Anweisungen an den Präprozessor einer Sprache zu nennen. Programmiersprachen mit Präprozessor sind beispielsweise C oder C++.

Inline-Generierung

Sind im Quelltext über die reinen Sprachelemente der Programmiersprache hinausgehende Angaben und Informationen (Attribute) enthalten, welche von einem Generator ausgewertet werden können, so spricht man von einem auf *Code-Attributen basierenden Generierungsansatz*. Prominentes Beispiel ist die automatische Erzeugung von HTML-Dokumenten anhand von Javadoc-Annotationen. Eine Abgrenzung zur Inline-Generierung besteht insofern, als Code-Attribute ausschließlich zur Generierung von zusätzlichen Artefakten wie Dokumentationen, Deployment-Deskriptoren, Datenbank-Schnittstellen etc. genutzt werden. Der Quelltext selbst wird dabei im Gegensatz zur Inline-Generierung nicht verändert.

Code-Attribute

Werden in sich vollständige, voneinander unabhängige Quelltext-Fragmente zusammengefügt, so spricht man auch von *Code-Weaving*. Dazu muss definiert werden, wie diese verschiedenen Bestandteile zusammengewoben werden können. Ein Beispiel für diese Generierungstechnik ist die aspektorientierte Programmierung (siehe Abschnitt 6.2.7).

Code-Weaving

Die oben beschriebene Spezifikation abstrakter Modelle als Eingaben für Generatoren, welche daraus weniger abstrakte Bausteine eines Systems erzeugen, ist ein in der Informatik wohlbekanntes Konzept. Letzten Endes kann ein Compiler ebenfalls als Generator betrachtet werden. Eine Unterscheidung der beiden Begriffe ist sehr subtil und in der Literatur oftmals nicht konsistent. In diesem Buch soll ein Compiler als Spezialfall eines Generators verstanden werden, welcher in einer Hochsprache formulierte Programme in weniger abstrakten, maschinennahen Code für eine bestimmte Laufzeitumgebung übersetzt, während der Output eines Generators sich nicht auf in einer definierten Laufzeitumgebung ausführbaren Code beschränkt. Meist erzeugt ein Generator Systembausteine, welche wiederum durch einen Compiler zu ausführbarem Code übersetzt werden. Zudem kann ein Generator auch Artefakte ohne die Intention der Ausführbarkeit erzeugen, beispielsweise Konfigurationsdateien oder Dokumentationsbestandteile.

Compiler als spezielle Generatoren

Einsatzgebiete generativer Techniken

Der Einsatz generativer Techniken verfolgt prinzipiell das Ziel, für einen Anwendungsfall oder ein Problem optimierte Ausgaben zu erzeugen. Es gibt selbstverständlich auch andere Möglichkeiten, solche problem- oder anwendungfallspezifischen Ausgaben zu erzeugen. Der Einsatz generativer Techniken ist dann vorzuziehen, wenn andere Mittel, wie beispielsweise die Meta-Programmierung oder die deskriptive Anpassung über Konfigurationsdateien, aus diversen Gründen nicht gut einsetzbar sind (beispielsweise aus Performanzgründen oder der erschwerten Wartbarkeit großer Konfigurationsdateien). Ein häufiges Einsatzgebiet für den Einsatz von Generatoren ist die im folgenden Abschnitt 6.2.6 beschriebene modellgetriebene Software-Entwicklung.

Praktische Abwägungen

Bei der Verwendung von Codegenerierung ist es wichtig abzuschätzen, wie groß der Aufwand für die Erstellung eines Generators ist. Zusätzliche Abstraktionen wie Templates, Aspekte, Metadaten etc. erhöhen potenziell auch die Komplexität der Architektur eines Systems: Die Architektur kann nur verstanden werden, wenn auch diese Abstraktionen verstanden werden. Auch dies ist zu beachten, um den Nutzen eines generativen Ansatzes abzuwegen. Üblicherweise lohnt es sich dann, einen Generator einzusetzen, wenn der Generator an verschiedenen Stellen verwendet werden kann.

6.2.6 Modellgetriebene Software-Entwicklung

Grundidee modellgetriebener Software-Entwicklung

Der Einsatz von Modellen in der Software-Entwicklung ist vielfältig. Ein Überblick über mögliche Einsatzszenarien sowie eine Einführung in grundlegende Konzepte der Modellierung liefert Abschnitt 6.6.1. Von modellgetriebener Software-Entwicklung (englisch: *Model Driven Software Development*, kurz MDSD) spricht man, wenn Modelle nicht lediglich zu Dokumentationszwecken eingesetzt werden, sondern zentrale Artefakte eines lauffähigen Systems bilden. Dabei ist Anwendungslogik nicht, wie bei traditioneller Entwicklung üblich, in einer 3GL-Programmiersprache (wie beispielsweise Java, C# oder C++) ausformuliert, sondern in Modellen spezifiziert. Solche Modelle müssen, so exakt und so ausdrucksstark wie möglich die durch die Software zu erbringende Funktionalität beschreiben. Das ist nur dann möglich, wenn die Elemente eines solchen Modells mit Semantik hinterlegt sind, welche eindeutig ein bestimmtes Verhalten zur Laufzeit bzw. eine bestimmte Software-Struktur definiert. Zu Beginn dieses Abschnitts wird, ausgehend von dem zentralen Konzept der Domäne und der domänenspezifischen Sprache (englisch: *Domain Specific Language*, kurz DSL, siehe Abschnitt 6.6.3), ein Überblick über die Kernkonzepte und grundlegen-

den Begrifflichkeiten des MDSD-Ansatzes gegeben. Weiterhin wird eine Abgrenzung der vorgestellten Konzepte zur *Model Driven Architecture* (MDA) der OMG vorgenommen sowie die Begriffe der *fachlich zentrierten MDSD* und *architekturzentrierten MDSD* erläutert, bevor am Ende dieses Abschnitts sowohl Chancen und Ziele, als auch Herausforderungen des MDSD-Ansatzes kurz diskutiert werden.

Ausgangspunkt eines Vorgehens nach MDSD ist stets ein begrenztes Wissens- oder Interessengebiet, das in der Regel als *Domäne* oder *Anwendungsdomäne* bezeichnet wird. Die Elemente einer Modellierungssprache, die in beliebigen Kontexten (Domänen) verwendet werden kann, sozusagen „general purpose“ ist, wären so wenig abstrakt und problemspezifisch, dass sie einer herkömmlichen 3GL-Sprache gliche. Die Definition einer Modellierungssprache rechnet sich nur dann, wenn Modellelemente den Problemraum prägnanter repräsentieren können als 3GL-Programmiersprachen. Das ist dann möglich, wenn die Sprache für eine spezielle Domäne entwickelt wird. Solche Modellierungssprachen nennt man domänenspezifische Sprachen oder DSLs (siehe Abschnitt 6.6.3).

Im Zentrum von MDSD steht das Modell. Dieses befindet sich auf dem Abstraktionsniveau der Anwendungsdomäne und wird typischerweise mittels einer domänenspezifischen Sprache formuliert. Diese definiert die Bedeutung des Modells. Die DSL, genauer: deren konkrete Syntax, kann entweder textuell oder grafisch sein. Auch tabellarische oder andere Notationen können verwendet werden. Ein Beispiel eines in einer domänenspezifischen Sprache formulierten Modells ist in Abbildung 6.2-11 zu sehen. Die verwendete DSL dient zur Modellierung von Java-Mobile-Information-Device-Profile- (MIDP-) Anwendungen, wie sie beispielsweise auf Handys zum Einsatz kommen. Das dargestellte Modell beschreibt, wie zwei Zahlen mittels eines Formulars eingegeben und anschließend addiert werden. Das Ergebnis der durchgeführten Addition wird wieder ausgegeben. Das zugehörige Metamodell ist in Abbildung 6.2-12 dargestellt. Domänenspezifische Sprachen sowie grundlegende Konzepte und Begrifflichkeiten der Modellierung, wie etwa jene des Meta- oder Metametamodells werden in Abschnitt 6.6.1 ausführlich behandelt. In diesem Zusammenhang wird auch der Begriff der Modellierungssprache von einem theoretischen Standpunkt aus beleuchtet.

Domänen und domänenspezifische Sprachen

Modelle auf dem Abstraktionsniveau der Anwendungsdomäne

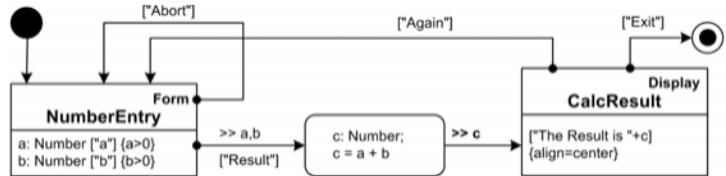


Abb. 6.2-11: Beispiel eines in einer DSL (MIDP) formulierten Modells.

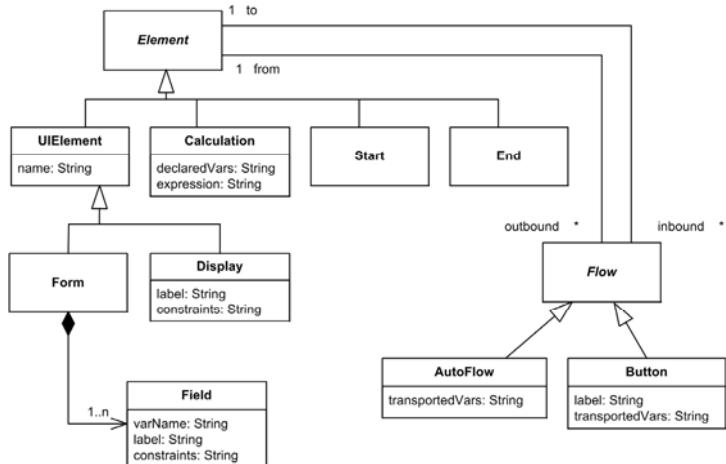


Abb. 6.2-12: Vereinfachtes Metamodell für die Mobile-Information-Device-Profile-Entwicklung.

Die Plattform

Neben den Modellen und der DSL gibt es einen weiteren zentralen Bestandteil modellgetriebener Entwicklung: die Plattform. Eine MDSD-Plattform besteht aus wiederverwendbaren, domänenspezifischen Bausteinen und Frameworks. Eine gute Plattform für MDSD im Bereich von Enterprise-Systemen besteht zunächst aus technischer Middleware wie CORBA [OMG 2004], JEE [Sun 2005] oder .NET [Microsoft 2004a]. Darauf aufbauend enthält die Plattform aber auch eine Reihe spezifischer Frameworks, die fachliche Basisdienste im Rahmen einer bestimmten Domäne erbringen. DSL und Plattform stellen praktisch zwei Seiten derselben Medaille dar: Die Plattform stellt Dienste zur Verfügung. Die DSL ermöglicht die einfache, effiziente und richtige Verwendung dieser Dienste. Die Regeln zur Transformation der Modelle in weniger abstrakte Modelle oder ausführbare Zielsprachen enthalten das Wissen über die Verwendung der Plattform. Abbildung 6.2-13 zeigt die typische Struktur einer Plattform, wie sie im Zusammenhang mit MDSD oft Verwendung findet. Die Inhalte der einzelnen Schichten variieren von Domäne zu Domäne; die Schichtung ist aber praktisch überall identisch.

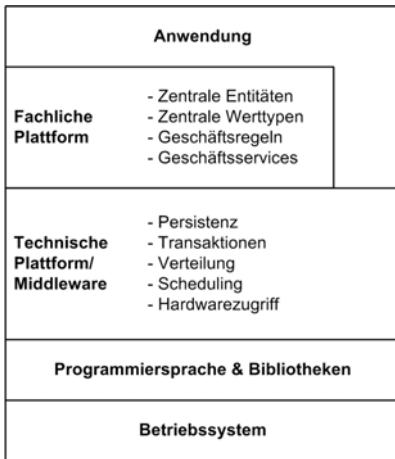


Abb. 6.2-13: Typische MDSD-Plattform im Bereich von Enterprise-Systemen.

Um letztendlich eine ausführbare Anwendung zu erhalten, bestehen grundsätzlich zwei verschiedene Möglichkeiten: Die direkte Interpretation der Modelle oder deren Transformation [Czarnecki und Helsen 2006] in weniger abstrakte, ausführbare Zielsprachen. Beide Möglichkeiten werden im Folgenden kurz beleuchtet und diskutiert.

Im Falle der *Interpretation* werden ausführbare Modelle durch eine virtuelle Maschine ohne den Zwischenschritt der Übersetzung direkt interpretiert. Prominentester Vertreter dieses Ansatzes ist die durch die OMG vorangetriebene Initiative einer ausführbaren UML (englisch: *Executable UML*) [Raistrick et al. 2004]. Des Weiteren existieren aber auch zahlreiche interessante Forschungsansätze. Beispielhaft genannt sei an dieser Stelle das Projekt *Active Charts* [ActiveCharts 2007]. Hier wird durch zur Laufzeit interpretierte Aktivitätsdiagramme das Verhalten aktiver Klassen modelliert und so der Programmfluss gesteuert.

In Falle des in der Praxis gängigen *generativen Ansatzes* wird die Übersetzung des Modells in eine ausführbare Anwendung üblicherweise mittels einer oder mehreren Transformationen erreicht. Die Codegenerierung, das heißt also die direkte Übersetzung des Modells in eine ausführbare Programmiersprache, stellt in diesem Zusammenhang einen Spezialfall dar. Technisch werden in diesem Fall meist Generierungstemplates verwendet, während im Falle eines mehrstufigen Transformationsprozesses geeignete Modelltransformationssprachen benötigt werden. Derartige Sprachen, insbesondere aber geeignete Werkzeuge, stecken meist noch in den Kinderschuhen, weshalb in der Praxis meist der templatebasierte Ansatz zum Einsatz kommt.

Vom Modell zur lauffähigen Anwendung

Interpretativer Ansatz

Generativer Ansatz

Interpretativer und generativer Ansatz im Vergleich

Die Interpretation ausführbarer Modelle wird oftmals mit dem Argument kritisiert, dass die Ausdrucksmächtigkeit einer solchen Modellierungssprache wiederum der einer generischen General Purpose Language entsprechen muss, sich also nicht auf eine bestimmte Anwendungsdomäne fokussiert und somit keinen echten Mehrwert gegenüber existierenden 3GL-Sprachen mit sich bringt. Wäre eine Anwendung vollständig mittels eines zur Laufzeit interpretierten Modells spezifiziert, so müsste dieser Argumentation beigeplichtet werden. Insbesondere als Ergänzung zu handgeschriebenen und generierten Quelltext-Anteilen und System-Bausteinen kann die Interpretation von Modellen jedoch sehr gewinnbringend eingesetzt werden. Zum Teil kann man eine Diskussion der Vor- und Nachteile beider Ansätze mit jenen einer Gegenüberstellung von statischen, kompilierten Programmiersprachen und dynamischen, interpretierten Skriptsprachen vergleichen (siehe Abschnitt 6.2.8). Zu nennen ist hier beispielsweise die größere Flexibilität aber schlechtere Laufzeit-Performanz interpretierter Sprachen.

Roundtrip Engineering

Der oben vorgestellte generative Ansatz wurde bisher unter dem Blickwinkel einer Vorgehensweise „vom Abstrakten zum Konkreten“ (*Forward Engineering*) betrachtet. Transformationsregeln dienen dabei der Übersetzung der Modelle in weniger abstrakte, meist ausführbare Zielsprachen. Eine vollständige Generierung einer lauffähigen Anwendung auf Basis domänenpezifischer Modelle ist in der Praxis derzeit jedoch noch auf wenige Anwendungsdomänen beschränkt. In einem MDSD-Projekt sollten daher von Anfang an geeignete Strategien zum Umgang mit generierten und nicht generierten Systemanteilen festgelegt werden. Eine mögliche Strategie ist die strikte Trennung generierter und nicht generierter Systemanteile. Eine weitere Lösungsstrategie besteht in der (automatisierten) Rückführung von Änderungen auf einer niedrigen Abstraktionsebene (beispielsweise dem Quellcode) in die Modelle der höheren Abstraktionsebenen. Unter *Roundtrip Engineering* wird gemeinhin die Möglichkeit verstanden, an zwei Artefakten auf verschiedenen Abstraktionsebenen Änderungen durchzuführen, wobei die Änderungen in beide Richtungen propagiert werden. Die beteiligten Artefakte werden dabei stets synchronisiert und konsistent gehalten. Inwiefern Roundtrip Engineering im Kontext von MDSD sinnvoll praktiziert werden kann, ist unter Entwicklern zum Teil heftig umstritten. Eine generelle Aussage lässt sich hier nicht treffen. Während im Falle von template-basierten Generierungsansätzen eine automatisierte Rückführung des generierten Quellcodes in das Modell aufgrund der im Quellcode nicht mehr ersichtlichen Domänensemantik nicht möglich ist (es sei denn, man arbeitet mit Quelltext-Annotationen), ist eine Synchronisation von Modellen auf verschiedenen Abstraktionsebenen (bei-

spielsweise dem PIM und dem PSM) durchaus wünschenswert. Die technische Umsetzbarkeit ist dabei abhängig von der Eigenschaft der Bidirektonalität der Transformationsregeln, welche die Modelle ineinander überführen [Czarnecki und Helsen 2006].

Oft lässt sich eine komplexe Anwendung nicht mit *einer* DSL alleine sinnvoll abbilden. Man wird für verschiedene Aspekte des Systems verschiedene DSLs verwenden. Für bestimmte Teile des Systems, wo sich die Erstellung einer DSL nicht lohnt und wo sich die Anwendungslogik gut mittels einer 3GL-Sprache ausdrücken lässt, kann natürlich auch in dieser programmiert werden. Da die Transformationen aber meist alle Aspekte gemeinsam berücksichtigen müssen, müssen die verschiedenen Modelle (und damit die DSLs) kombinierbar sein. Damit wird ein gemeinsames Metametamodell, also ein Mittel zur Konstruktion der verschiedenen DSLs, benötigt. Praktisch ist das Metametamodell meist durch die Einbettung der DSL in eine bestehende Modellierungsarchitektur implizit gegeben (siehe Abschnitt 6.6).

Die Anwendungsdomäne sowie die damit korrelierte domänen spezifische Sprache nehmen eine Schlüsselstellung im Rahmen von MDSD ein. Eine Domäne kann dabei fachlich oder technisch motiviert sein. Im Falle von fachlich motivierten Domänen spricht man daher auch von *fachlich zentrierter MDSD*. Im Kontext der modellgetriebenen Software-Entwicklung ist der entscheidende Aspekt einer technisch motivierten Domäne meist die Architektur der zu erstellenden Anwendung. Im Falle einer architektonisch motivierten Domäne spricht man daher auch von *architektonisch zentrierter MDSD*, welcher im Rahmen dieses Abschnitts besonderer Aufmerksamkeit gewidmet werden soll.

Den obigen Ausführungen zu Folge ist architektonisch zentrierte MDSD also eine Spezialisierung von MDSD mit den folgenden Eckpfeilern:

- Die Domäne ist architektonisch motiviert – z. B. „Architektur für Business-Software“ oder „Komponenteninfrastrukturen für Eingebettete Systeme“.
- Das Metamodell der DSL enthält dementsprechend die Architektur-Abstraktionen.
- Die Transformation hat nicht den Anspruch, die gesamte Anwendung zu erzeugen, sondern nur einen Implementierungsrahmen, der den architektonischen Infrastrukturcode enthält (Skelett). Der nicht generierte fachliche Implementierungscode wird manuell in der Zielsprache implementiert.

Die Rolle eines Metametamodells

Fachlich zentrierte vs. architektonisch zentrierte MDSD

Eckpfeiler architektonisch zentrierter MDSD

Architektur-Meta-modellierung

Es sei darauf hingewiesen, dass die Erstellung eines Architektur-Metamodells auch ohne MDSD sinnvoll ist, weil es den Architekten dazu zwingt, sich systematisch über die Software-Architektur Gedanken zu machen, was insbesondere die Aufdeckung von Inkonsistenzen erleichtert. Dieser Aspekt wird im Rahmen von *Architecture Description Languages* (ADL) in Abschnitt 6.6.4 besprochen.

Begriffsvielfalt in der modellgetriebenen Software-Entwicklung

Im Kontext modellgetriebener Software-Entwicklung kursieren unter Software-Entwicklern einige Begrifflichkeiten und Akronyme. Der in diesem Buch verwendete Begriff *Model Driven Software Development* (MDSD), sowie die beiden Begriffe *Model Driven Development* (MDD) und *Model Driven Engineering* (MDE) sind die wohl am häufigsten verwendeten Begriffe, bezeichnen aber alle die in diesem Abschnitt vorgestellten Konzepte.

Der MDA-Standard als Spezialisierung von MDSD

Die *Model Driven Architecture* (MDA) [OMG 2007a] der OMG ist letztendlich nichts anderes als eine Spezialisierung von modellgetriebener Software-Entwicklung, wie sie in diesem Abschnitt eingeführt wurde. Während ein genereller MDSD-Ansatz die Wahl der verwendeten Modellierungssprachen offen gestaltet und keine Einschränkungen hinsichtlich der Transformationen in lauffähige Anwendungen macht, äußert sich die Spezialisierung in Gestalt der MDA in der Standardisierung

- > der zu verwendenden Modellierungssprachen und Modellierungsarchitektur, also der zu verwendenden Modellierungsmittel zur Definition der domänen spezifischen Sprachen,
- > eines mehrstufigen Prozesses und der darin involvierten Artefakte, um über eine Reihe von Transformationen vom an der Anwendungsdomäne orientierten Modell zu einer lauffähigen Anwendung zu gelangen und
- > der zu verwendenden Mittel zur Beschreibung der benötigten Transformationsregeln.

Die primär damit verfolgten Ziele sind in erster Linie die Interoperabilität zwischen den verwendeten Werkzeugen und, auf längere Sicht, die Standardisierung von Modellen für populäre Anwendungsdomänen. Die folgende Auflistung liefert einen Überblick über diejenigen der in diesem Abschnitt eingeführten Konzepte, für welche die MDA konkretere Vorstellungen hat als der allgemeine MDSD-Ansatz [Stahl und Völter 2005].

- > *DSL*: MDA-konforme DSLs sind Sprachen, die mittels der MOF (Meta-Object Facility, siehe Abschnitt 6.6) definiert werden. In der Praxis werden meist UML-Profile verwendet, also Anpassungen der

UML und Profile

UML mittels Stereotypen, Tagged Values und Constraints. Die Erweiterungsmöglichkeiten der UML werden in Abschnitt 6.6.2 näher beleuchtet.

- > **Modellierungsarchitektur:** Domänenpezifische Sprachen in Form von UML-Profilen sind in die vierschichtige Modellierungsarchitektur der OMG eingebettet. Die Meta-Object Facility bildet das Metamodell, also die oberste Instanz, dieser Modellierungsarchitektur. Modellierungsarchitekturen werden in Abschnitt 6.6 im Detail diskutiert.
- > **Präzisierung von Modellen:** Um Modelle zu präzisieren und semantisch anzureichern bzw. um Verhalten besser spezialisieren zu können, können die OCL (*Object Constraint Language*) sowie seit UML 2.0 die *Action Semantics* verwendet werden. Der Aspekt der statischen Semantik wird in Abschnitt 6.6.1 von einem theoretischen Standpunkt aus betrachtet.
- > **Transformationen:** MDA-konforme Transformationen sollten auf Modelltransformationssprachen aufbauen, die im Rahmen der *Query/Views/Transformations-* (QVT) [OMG 2007b] Spezifikation standardisiert werden. Eine vollständige Implementierung dieser Spezifikation ist derzeit noch nicht verfügbar.
- > **PIM und PSM:** Kernbestandteil der MDA ist das Konzept der vertikalen Separation of Concerns (siehe auch Abschnitt 6.2.7), das heißt, die Spezifikation von Aspekten unterschiedlicher Abstraktionsstufen durch verschiedene Modelle. Plattformunabhängige Aspekte werden im Rahmen des PIM (englisch: *Platform Independent Model*) spezifiziert. Das PIM wird auf ein oder mehrere plattformspezifische Modelle (englisch: *Platform Specific Model*, kurz PSM) abgebildet. Das PSM stellt somit den Bezug zu einer konkreten Plattform her.

MOF

OCL

QVT

PIM und PSM

Die Gründe, modellgetriebene Software-Entwicklung zu verwenden, können vielfältig sein. Einige davon werden im Folgenden beleuchtet. Die MDA verfolgt vor allem das Ziel, die gleiche Anwendungslogik mittels verschiedener Transformationen auf verschiedenen Plattformen zur Ausführung zu bringen. Es gibt aber noch eine ganze Reihe weiterer Gründe für MDSD. Im JEE-Umfeld beispielsweise birgt die Entwicklung von Software sehr viele, sich oft wiederholende, fehleranfällige Schritte. Diese können mittels MDSD sehr gut automatisiert werden.

**Chancen und Ziele
modellgetriebener
Software-Entwicklung**

Modellgetriebene Software-Entwicklung erfordert den Aufbau einer Infrastruktur bestehend aus DSL, Modellierungswerzeugen, Generatoren, Plattformen etc. Auch muss ein erheblicher Aufwand in die Domänenanalyse gesteckt werden, sodass man überhaupt zu einer sinnvollen

**Software-
Produktlinien, -Fabri-
ken und -System-
familien**

Infrastruktur kommt. Dieser Aufwand rechnet sich in der Regel nicht für eine „einmalige Anwendung“, sondern nur, wenn man die Infrastruktur mehrmals anwendet. Dies führt zum Entwurf von *Software-Produktlinien* (englisch: *Software Product Lines*) [Pohl et al. 2005] und *Software-Fabriken* (englisch: *Software Factories*) [Greenfield und Short 2004] durch die Identifikation von *Software-Systemfamilien* (englisch: *Program Families*) [Parnas 1976]. Die Familienmitglieder einer solchen Systemfamilie zeichnen sich dadurch aus, dass sie eine Menge von fachlichen oder technischen Merkmalen gemeinsam haben. Oft setzen sie auf denselben technischen Infrastruktur auf. Damit eröffnet sich die Möglichkeit zur Wiederverwendung, und zwar eben nicht nur von Bausteinen und Frameworks, sondern auch von Metamodellen, Generatoren und Transformationen. Genau diese Wiederverwendungsmöglichkeiten sind dafür verantwortlich, dass sich MDSD trotz des zusätzlichen Aufwands der Erstellung einer MDSD-Infrastruktur rechnet.

Integration von Domänenexperten

Durch die Nähe der DSL zur Domäne wird es erheblich einfacher, Fachexperten direkt in die Entwicklung einzubinden. Voraussetzung ist dabei, dass die DSL die Domäne gut repräsentiert. Zu einer solchen DSL kommt man nicht über Nacht. Fundiertes Domänenwissen und Erfahrung in der Definition von DSLs sind nötig. Ein iterativer Ansatz liegt hier nahe (siehe Abschnitt 8.1).

Definierte Zielarchitektur

Im Hinblick auf die Software-Architektur hat MDSD einige nützliche „Nebeneffekte“. Modelltransformationen bilden Konstrukte des Quellmetamodells auf Elemente des Zielmetamodells ab [Czarnecki und Helsen 2006]. Damit man diese Abbildung prägnant formulieren kann, müssen diese beiden Metamodelle eine begrenzte Menge definierter Konzepte enthalten. Es muss klar in Regeln fassbar sein, was worauf abgebildet werden soll. Im Falle der Transformation auf die Zielplattform bedeutet dies, dass die Plattform (bzw. deren Architektur) eine begrenzte Menge definierter Konzepte beinhalten muss. Genau dies ist eines der wesentlichen Merkmale guter Architektur. Insofern „erzwingt“ MDSD eine wohl definierte Architektur und unterstützt die Entwickler, konform zu dieser Architektur zu entwickeln. Regeln im Umgang mit der Architektur sind in den Transformationen codiert.

Die wesentlichen Vorteile eines MDSD-basierten Vorgehens lassen sich unter den folgenden Stichpunkten prägnant zusammenfassen:

- > Größere Entwicklungseffizienz.
- > Bessere Integration der Fachexperten.
- > Leichtere Änderbarkeit von Software.

- > Verbesserte (Umsetzung der) Software-Architektur.
- > Möglichkeit zur verhältnismäßig einfachen Portierung der Fachlogik auf andere Plattformen.

Das Paradigma der modellgetriebenen Software-Entwicklung wird von dessen Befürworten oftmals hochgradig idealisiert dargestellt. Es soll daher zum Abschluss dieses Abschnitts auch kurz auf existierende Probleme im Kontext modellgetriebener Software-Entwicklung eingegangen werden.

Software-Konfigurations-Management (SKM) ist ein unverzichtbarer Bestandteil qualitativ hochwertiger Software-Entwicklung. Dies gilt selbstverständlich auch im Kontext von MDSD. Der bei traditionellen Vorgehensweisen im Fokus stehende Quelltext einer Anwendung wird im Rahmen von MDSD zumindest zu großen Anteilen durch Modelle ersetzt. Daraus generierter Quelltext ist jederzeit reproduzierbar und im Kontext des SKM von untergeordneter Bedeutung. Essentiell ist dagegen die Versionskontrolle über die verschiedenen Modelle. Der Einsatz bestehender Werkzeuge wie *CVS* oder *Subversion* zur Versionierung, parallelen Bearbeitung von Modellen sowie zur Nutzung typischer Repository-Funktionen- wie beispielsweise der Differenzanalyse und dem Mischen (Merge) konkurrierender Versionen, ist nur sehr eingeschränkt möglich, da existierende Versions-Management-Systeme (VMSysteme) meist nur Textdateien verwalten. Auch wenn der Quelltext eines Programms eine konzeptionell über eine reine Sequenz von Textzeilen hinausgehende Struktur besitzt, nämlich die des abstrakten Syntaxbaumes (englisch: Abstract Syntax Tree, kurz AST), so sind diese beiden Repräsentationsformen nahezu konsistent. Die Differenzen zwischen zwei textuellen Repräsentationen einer Programmstruktur sind zufriedenstellend (wenn auch nicht perfekt) auf die konzeptuelle Differenz beider Versionen des Programms abbildbar.

Im Gegensatz hierzu handelt es sich bei der dem Entwickler vertrauten konkreten Syntax der meisten Modellierungssprachen um Diagramme und nicht um Text (textuelle DSLs sind von der im Folgenden geführten Diskussion und Argumentation nicht betroffen). Die persistente Speicherung des Modells in Form von Dateien (beispielsweise im XMI-Format) ist im Falle graphbasierter Diagramme nicht von Interesse. Eine Analyse auf Basis der verwalteten Dateien ist nur von geringem Nutzen. Layout und Lokalität im Rahmen eines Diagramms spielen für den Betrachter eine nicht unwesentliche Rolle, da dieser meist eine Art mentale Landkarte mit einem Diagramm assoziiert. Das Layout sollte

Probleme im Kontext von MDSD

Probleme mit existierenden SKM-Werkzeugen

Differenzanzeige für graphbasierte Dokumente

Differenzberechnung für graphbasierte Dokumente

daher zumindest bei der Differenzanzeige beachtet werden. Die Konzeption und Entwicklung verbesserter Anzeigeformen und Interaktionsmöglichkeiten ist zwingend notwendig, derzeit aber noch Gegenstand der aktuellen Forschung auf diesem Gebiet. Qualitativ hochwertige Werkzeuge wie im Falle von textbasierten Dokumenten sind derzeit noch nicht verfügbar.

Auch im Hinblick auf die Berechnung einer Differenz gibt es entscheidende Aspekte zu beachten. Im Rahmen der Berechnung einer Differenz erhält die Festlegung korrespondierender Modellelemente eine zentrale Bedeutung [Kelter et al. 2008]. Schwierigkeiten ergeben sich bei der Bildung der Korrespondenzen, welche sich für graphartig strukturierte Dokumente (wie dies typischerweise für Modelle in MDSD der Fall ist) weitaus schwieriger darstellt als für textuelle Dokumente. Existierende Lösungen zur Berechnung von Modelldifferenzen basieren oftmals auf persistenten Identifizierern (IDs). Sofern das Modellierungswerkzeug persistente IDs unterstützt und zu vergleichende Modelle Revisionen voneinander sind, können die persistenten IDs zur Bildung der Korrespondenzen herangezogen werden. Dieser Ansatz bringt jedoch eine Reihe von Problemen mit sich und ist zudem von einer homogenen Entwicklungsumgebung hinsichtlich des verwendeten Modellierungswerkzeugs abhängig. Eine mögliche Alternative stellen ähnlichkeitsbasierte Ansätze zur Berechnung der Korrespondenzen dar. Ein Beispiel eines solchen ähnlichkeitsbasierten Verfahrens ist der SiDiff-Ansatz [Kelter et al. 2005], dessen Kern ein generischer Algorithmus bildet, welcher Korrespondenzen auf Basis externer, auf Heuristiken basierender Steuerinformationen berechnet.

Mischen von Modellen

Ebenso wie die Anzeige und die Berechnung von Differenzen stellt auch das Mischen verschiedener Varianten eines Modells eine Herausforderung dar. Das Mischen von Modellen basiert auf einer in der Regel durch Vergleich von zwei (oder drei) Modellen gewonnene Differenz. Ziel ist es, ein weiteres Modell zu erzeugen, das die „Besonderheiten“ der Ausgangsmodelle enthält. Hinsichtlich des automatisierten Treffens von Mischentscheidungen sowie des manuellen Eingreifens im Falle von Konflikten stellt das Mischen höchste Anforderungen an die zu konzipierenden Werkzeuge [Kelter 2007]. Qualitativ hochwertige Mischwerkzeuge für Modelle sind derzeit noch Mangelware.

Vielfalt der Modellierungssprachen

Abgesehen von den bisher diskutierten Problemen ist ebenfalls zu beachten, dass sich Modellierungssprachen beziehungsweise Diagrammarten hinsichtlich ihres strukturellen Aufbaus, ihrer Semantik und/oder ihrer Notation teilweise erheblich unterscheiden. Während

textbasierte Versions-Management-Systeme für das Management der Artefakte verschiedener Programmiersprachen gleichermaßen eingesetzt werden können, werden im Falle der Verwaltung von Modellen im Prinzip für jeden Modelltyp, evtl. sogar für jeden Diagrammtyp, dedizierte Werkzeuge benötigt. An den Modelltyp bzw. den Diagrammtyp angepasst werden müssen hierbei insbesondere die Anzeigeformen, die Differenzberechnung, die Konfliktbehandlung beim Mischen sowie die Integration mit anderen Werkzeugen. Ein darüber hinaus oftmals bestehendes Problem im Zusammenhang mit Sprachen, die nicht stark verbreitet sind, ist das Tooling (siehe Abschnitt 6.2.8).

Über die typischen Dienste eines Konfigurations-Management-Systems hinausgehend werden geeignete Werkzeuge und Methoden zur Bewältigung und Kontrolle der Evolution verschiedener Modelle benötigt. Hervorzuheben ist hierbei die Evolution der domänen spezifischen Sprache. Wie bereits angesprochen, entstehen DSLs nicht über Nacht, sondern sind meist das Resultat eines hochgradig iterativen Prozesses. Die durch die schrittweise Verfeinerung des formal erfassten Domänenwissens oder durch externe Anforderungen einer instabilen und sich ständig ändernden Umwelt getriebene Evolution der DSL zieht zu beachtende Konsequenzen wie z.B. die Anpassung der Generatoren und Transformationsregeln nach sich. Geeignete Lösungsansätze sind ebenfalls noch Gegenstand der Forschung [Van Deursen et al. 2007].

Modellevolution

6.2.7 Aspektorientierung

Die Aspektorientierung [Kiczales et al. 1997] vermeidet über den Code oder den Entwurf verstreute Lösungen für sogenannte Crosscutting Concerns. Dies sind Belange, die übergreifend oder quer liegend zur Anwendungslogik sind. Stattdessen werden solche Lösungen in einem Aspekt gekapselt und somit von dem durch den Aspekt betroffenen System separiert. Ein Aspekt stellt einen Belang dar, der aus der eigentlichen Anwendungslogik herausgelöst betrachtet werden kann. In der Diskussion der Komponentenorientierung wurden bereits einige solche Belange als technische Belange diskutiert, wie beispielsweise wie Protokollierung, Sicherheit, Aktivierung oder Lebenszeitmanagement von Komponenten. Solche in naiven Implementierungen durch den Code verteilten Aspekte zu separieren, ist die Hauptaufgabe der Aspektorientierung. Das heißt, Aspektorientierung realisiert das Prinzip Separation of Concerns für diese Crosscutting Concerns.

Idee der Aspektorientierung

Abbildung 6.2-14 zeigt als Beispiel drei Komponenten, in denen die Aspekte Persistenz, Logging und Synchronisation fix codiert sind. Wie durch die gestrichelte Umrandung angedeutet, sind dies also Crosscutting Concerns der drei Komponenten.

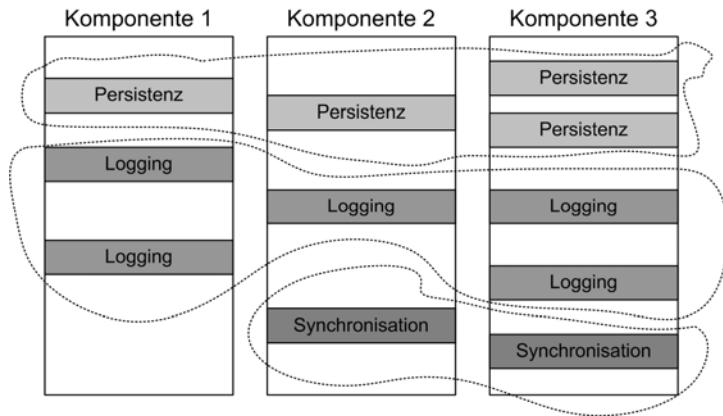


Abb. 6.2-14: Beispiel Crosscutting.

Aspektorientierung vermeidet das Problem Crosscutting Concerns über den Code verstreut zu implementieren, indem es den Belang als Aspekt kapselt. Der Aspekt wird automatisch in das System eingewoben, sodass in die eigentlichen Programme nicht direkt eingegriffen werden muss – die Entwickler der Systembausteine müssen den Aspekt also gar nicht beachten. Das heißt, Aspekte sind „nicht-invasiv“ [Filman und Friedman 2000] aus der Sicht des Programms, das um den Aspekt erweitert wird.

Aspektorientierung und Meta-Programmierung

„Nicht-invasiv“ zu sein, stellt eine wichtige Eigenschaft der aspektorientierten Programmierung dar. Aspektorientierung ist aus der Meta-Programmierung entstanden. Hier gibt es die Beobachtung, dass einige Meta-Programmierungskonstrukte recht komplex werden können, da sie es schwer machen, das System zu verstehen, ohne den „Meta-Kontext“ in dem es sich gerade befindet, zu verstehen. Ein Beispiel sind Lisp-Makros: Man kann ein gegebenes Lisp-System nur verstehen, wenn man vorher auch die Makros für dieses System betrachtet hat, denn die Makros können die Bedeutung der Sprachelemente verändern. Aspektorientierung vermeidet solche Konstrukte, denn die Aspekte und das System können relativ unabhängig voneinander betrachtet werden.

Aspektsysteme und ihre Basiskonzepte

Aspektorientierung wird durch Systeme zur aspektorientierten Programmierung (AOP) realisiert. Populäre AOP-Implementierungen, wie AspectJ [Kiczales et al. 2001], Hyper/J [Tarr 2004], JBoss AOP [Burke

2004] oder AspectWerkz [Bonér und Vasseur 2004], realisieren dieses Konzept auf recht unterschiedliche Weise und insgesamt befinden sich viele AOP-Konzepte noch stark in der Entwicklung. Intern können AOP-Konzepte durch Meta-Programmierung, Bytecode-Manipulation oder generative Programmierung realisiert werden. Diese Realisierungstechniken werden näher in [Zdun 2004] erklärt.

Ein bekanntes und inzwischen weit verbreitetes Beispiel eines Tools ist AspectJ. AspectJ behandelt Aspekte auf Sprachebene von Java. Es definiert dazu eine Reihe von Spracherweiterungen, die die oben genannten Konzepte umsetzen. Rein technisch gesehen, funktioniert die Umsetzung mittels Quell- oder Bytecode-Manipulierung, was bedeutet, dass der Aspektcode statisch mit dem Kernprogramm „verwoben“ wird (mit einem sogenannten „Aspect Weaver“; in anderen Ansätzen wird ein „Aspect Composition Framework“ zur Komponierung der Aspekte eingesetzt). Damit erlaubt es AspectJ, Java-Bytecode zu erzeugen, der die Aspekte enthält, aber trotzdem bleiben Aspekte und Kernprogramm im Quelltext voneinander separiert. Dieses Vorgehen zeigt die Abbildung 6.2-15.

Aspect

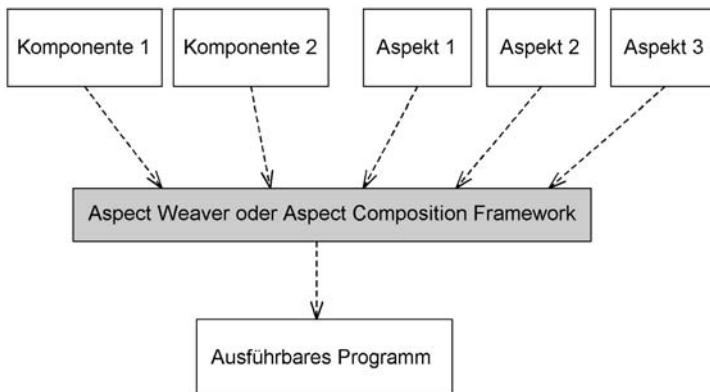


Abb. 6.2-15: Funktionsweise eines Aspect Weavers bzw. Aspect Composition Frameworks.

Alle oben genannten Ansätze haben einige Konzepte gemeinsam, die allerdings unterschiedlich realisiert werden. Zum Teil wird auch eine andere Terminologie verwendet. Hier soll die AspectJ-Terminologie benutzt werden:

- > *Join Points*
- > *oinpoints* definieren die Stellen eines Programms, an denen der Aspekt zur Laufzeit eingreifen kann.

Konzepte der Aspektorientierung

- > *Advices* definieren ein Verhalten, das der Aspekt dem Programm vor, nach oder anstatt der Ausführung eines Joinpoints hinzufügen kann.
- > *Pointcuts* stellen eine Menge von Joinpoints dar, an denen ein bestimmter Advice tatsächlich eingreift. Mit anderen Worten erlaubt der Pointcut dem Entwickler, den Zusammenhang zwischen Joinpoints und Pointcuts für eine bestimmte Anwendung zu spezifizieren.
- > *Introductions* stellen Strukturveränderungen an einem Programm dar. Beispielsweise kann man einer Klasse eine neue Schnittstelle oder eine neue Methode hinzufügen.

Beispiel mit AspectJ

Folgende Konto-Klasse sei als – einfaches – Beispiel gegeben:

```
public class Konto {
    String kontonummer;
    double guthaben;
    public void abheben(double betrag) {
        guthaben = guthaben - betrag;
    }
    public void einzahlen(double betrag) {
        guthaben = guthaben + betrag;
    }
    public void ueberweisen(double betrag,
                           Konto zielKonto) {
        guthaben = guthaben - betrag;
        zielKonto.einzahlen(betrag);
    }
}
```

Angenommen man will diese Klassen um eine Logging-Funktionalität bei jedem Methodeneintritt in Methoden dieser Klasse erweitern. Dazu müsste man eine Code-Zeile für das Logging in jede Methode einfügen. Das verletzt das Architektur-Prinzip Separation of Concerns, denn der Aspekt Logging ist nicht gekapselt. Mit einem Aspekt kann man dieses Problem beheben.

Der Beispielaspekt unten besteht aus einem Advice, der mit `before` anfängt, was bedeutet, dass der Advice vor dem Joinpoint ausgeführt wird. Alle betrachteten Joinpoints werden durch den Pointcut `execution(* Konto.*(..))` bestimmt: Das sind alle Methodenausführungen auf einem Konto mit beliebigen Argumenten. Immer wenn ein solcher Joinpoint zur Laufzeit erreicht wird, wird der Advice (also der folgende Block in geschweiften Klammern) ausgeführt. Also ist das Resultat, dass jeder Methodeneintritt eines Kontos automatisch mitgeloggt wird.

```
public aspect KontoLogger {  
    before(): execution(* Konto.*(..)) {  
        Logger.log(thisJoinPoint,  
                  thisJoinPoint.getArgs());  
    }  
}
```

6.2.8 Skriptsprachen und dynamische Sprachen

Skriptsprachen (englisch: *scripting language*) sind ursprünglich Programmiersprachen, die Software-Systeme kontrollieren oder steuern sollen. Typischerweise benutzen Skriptsprachen einen Zwei-Sprachen-Ansatz: das Kernsystem wird in einer anderen Programmiersprache als der Skriptsprache implementiert und die Skriptsprache übernimmt nur das Zusammenfügen der Systembausteine in ein lauffähiges System. Dies hat den Vorteil, dass der Systemnutzer die Systembausteine flexibel komponieren kann und somit das System an seine Bedürfnisse anpassen kann, ohne in den Kernsystemcode eingreifen zu müssen.

Skriptsprachen

Daher sind Skriptsprachen häufig Sprachen auf hoher Abstraktionsebene, die interpretiert oder zur Laufzeit kompiliert werden. Die Sprachen, die von den Skriptsprachen komponiert werden, werden hingegen typischerweise zu nativen Maschinencode kompiliert. Ein typisches Beispiel sind moderne Skriptsprachen, wie Perl, Python, Ruby oder Tcl, die selbst in C implementiert sind und typischerweise C- oder C++-Bausteine komponieren. Wie man an diesem Beispiel sieht, werden Skriptsprachen üblicherweise in der Sprache, die sie komponieren, eingebettet. Neben Sprachen, wie C oder C++, sind viele Skriptsprachen heute auch auf Basis von Sprachen, die mit virtuellen Maschinen ausgeführt werden, wie C# oder Java, realisiert.

Der Name Skriptsprache kommt daher, dass Skripte ursprünglich im Bereich von Batch-Jobs oder Shell-Skripten des Betriebssystems eingesetzt wurden. Moderne Skriptsprachen sind jedoch vollständige Programmiersprachen, und nicht selten werden auch ganze Systeme ausschließlich in diesen Sprachen implementiert. In vielen Fällen werden Skriptsprachen auch eingesetzt, um zunächst einen Prototyp schnell zu erstellen und dann wird dieser schrittweise in die Sprache migriert, in der die Skriptsprache eingebettet ist.

Durch ihre Historie werden Skriptsprachen in manchen Entwicklergemeinden als „Hackersprachen“ angesehen und haben dementsprechend

Dynamische Sprachen

ein schlechtes Image. Auch die häufige Nutzung solcher Sprachen als eigenständige Programmiersprachen lässt die Bezeichnung „Skriptsprache“ obsolet erscheinen. Aus diesen Gründen positionieren sich viele dieser Sprachen heute als *dynamische Sprachen*. Zum Teil werden die Sprachen auch als *agile Sprache* positioniert, um ihre häufige Nutzung im Kontext agiler Entwicklungsprozesse herauszustreichen.

Dynamische Sprachen bezeichnen allgemein Sprachen auf hohem Abstraktionsniveau, die während der Laufzeit viele Aufgaben ausführen, die andere Sprachen zu Kompilierzeit durchführen. Beispiele für diese Aufgaben sind das Parsen der Sprache, die Erweiterung der Sprache um neuen Code, die Erweiterung existierender Klassen-, Prozedur- oder Datendefinitionen, die Änderung des Typsystems etc. Viele dynamische Sprachen sind dynamisch getypt, aber dies ist keine Voraussetzung für dynamische Sprachen. Neben den genannten Sprachen aus dem Skriptsprachenumfeld, zählen insbesondere auch Sprachen wie Lisp oder Smalltalk (und ihre Derivate) zu den dynamischen Sprachen.

Ein wichtiges Beispiel für ein Sprachmittel dynamischer Sprachen ist, dass Variablentypen nicht deklariert werden müssen, sondern automatisch ermittelt werden. Beispielsweise in dem folgenden Tcl-Code, wird die Variable `a` zunächst mit einem Integer-Wert belegt und an diesen Datentyp intern (das heißt im Interpreter) gebunden. Bei der darauffolgenden erneuten Zuweisung bekommt `a` einen String-Wert und wird daher automatisch an diesen Datentyp gebunden.

```
set a 1          ;# a wird an Integer gebunden
set a "a b c"  ;# a wird an String gebunden
```

Ein weiteres typisches Beispiel für ein dynamisches Sprachmittel ist die Möglichkeit, Daten, die in der Sprache angegeben werden, als Code zu verwenden. Dieses Sprachmittel kann man mit dem folgenden einfachen Lisp-Beispiel verdeutlichen. In dem Beispiel wird einer Variable `a` ein Programmfragment übergeben, das der Variable `b` den Wert `1` zuweist. Später wird dieses Programmfragment mit dem Befehl `eval` ausgewertet. Die Konsequenz ist, dass der Code in der Variable `a` dynamisch ausgeführt wird und `b` den Wert `1` erhält.

```
(setf a '(setf b 1))
; ...
; einige Zeit spaeter
```

```
; ...
(eval a)
```

Closures sind ein weiteres dynamisches Sprachmittel, das aus dem Bereich der funktionalen Programmiersprachen stammt. Closures sind Funktionen, die beim Aufruf den Kontext, in dem sie definiert wurden, konservieren. Ein Beispiel für dieses Sprachmittel sind Ruby-Blöcke, die den Kontext ihrer Definition konservieren. In dem folgenden Beispiel wird `nMal` an `b` mit dem Parameter `20` übergeben. In dem in `nMal` enthaltenen Block wird der dynamisch übergebene Wert für `m` konserviert. Die Variable `n` hingegen verändert sich von Aufruf zu Aufruf.

```
def nMal(m)
  return proc{ |n| m * n }
end

b=nMal(20)
b.call(1)  # liefert 20
b.call(3)  # liefert 60
```

Dynamische Sprachen und Skriptsprachen sind durch ihre mächtigen Mittel zur Spracherweiterung insbesondere gut geeignet, um domänen-spezifische Sprachen zu implementieren. DSLs wurden bereits in Abschnitt 6.2.6 im Kontext der modellgetriebenen Entwicklung diskutiert. Eine Abhandlung zu DSLs aus Sicht der Modellierung findet sich in Abschnitt 6.6.3. In dynamischen Sprachen werden DSLs nicht als *externe* Sprachen für die Eingabe konkreter Modelle erstellt, sondern eher als *interne* Spracherweiterungen der existierenden Sprache.

Beispielsweise wird kein neuer Parser für die DSL implementiert, sondern der existierende Parser der dynamischen Sprache (erweitert und) verwendet. Somit sind DSLs mit dynamischen Sprachen recht einfach zu realisieren und man kann in ihnen Abstraktionen der dynamischen Sprache verwenden. Beispielsweise macht es oft Sinn, in der DSL die Schleifen der dynamischen Sprache anzubieten, um Fälle abzudecken, in denen ein Code-Fragment wiederholt ausgeführt werden soll. Wenn eine neue Sprache implementiert wird (wie es oft im Rahmen der modellgetriebenen Entwicklung der Fall ist), ist eine solche Wiederverwendung nicht möglich. Allerdings hat die interne DSL den Nachteil, dass auch Sprachkonstrukte, die man dem Domänenexperten nicht zur Verfügung stellen möchte, verfügbar sind. Dies vollständig zu verhindern, ist in vielen dynamischen Sprachen aufwendig.

DSLs in dynamischen Sprachen

Beide Ansätze können natürlich kombiniert werden. Das heißt, eine DSL basierend auf einer dynamischen Sprache kann als DSL für modellgetriebene Entwicklung verwendet werden: Bei ihrer Ausführung befüllt so eine DSL die Modelle für den Generator.

Einen weiter gehenden Überblick zu diesen Themen bietet der Artikel [Fowler 2005].

Dynamische Sprachen für Web-Anwendungen

Im Moment sind einige der dynamischen Sprachen, wie Ruby, Groovy und Smalltalk, im Zusammenhang mit sogenannten agilen Rahmenwerken für Web-Anwendungen recht populär. Beispiele sind Ruby on Rails, Grails und Seaside. Diese Rahmenwerke nutzen die dynamischen Sprachmittel, unter anderem um die Entwicklung von Web-Anwendungen zu beschleunigen und Rapid Prototyping zu unterstützen. Ruby on Rails ist beispielsweise im Wesentlichen eine Reihe von DSLs für Web-Anwendungen. Darunter liegt ein Rahmenwerk, das dem Model/View/Controller-Muster folgt. Wie viele andere Anwendungen von dynamischen Sprachen, folgt auch Ruby on Rails dem „Konventionen-statt-Konfigurationen-Prinzip“ (siehe Abschnitt 6.1). Das heißt, es werden eher sinnvolle Standardannahmen gemacht und nur notwendige Anpassungen müssen konfiguriert werden. Somit kommen Entwickler in der Regel schnell zu einem ersten Ergebnis, dass man dann schrittweise an die eigenen Bedürfnisse anpassen kann.

Vorteile Skriptsprachen und dynamischer Sprachen

Die Möglichkeit auf hohem Abstraktionsniveau in der Skriptsprache, DSL oder dynamischen Sprache zu entwickeln, ist der zentrale Vorteil dieser Ansätze. Ein Teilaспект dieses Vorteils ist, dass man die Sprache an das gegebene Problem anpassen und somit die für das Problem passendste Sprache verwenden kann. Wenn ein Zwei-Sprachen-Ansatz verfolgt wird, kann man die Skriptsprache oder dynamische Sprache zum Rapid Prototyping einsetzen.

Nachteile Skriptsprachen und dynamischer Sprachen

Demgegenüber stehen aber auch einige Nachteile. Zunächst haben viele Skriptsprachen oder dynamische Sprachen eine schlechtere Performanz als Sprachen, wie C, C++, C# oder Java. Ein Mehrsprachenansatz kann sich negativ auswirken, da alle verwendeten Sprachen gewartet werden müssen und man auf lange Sicht Experten für alle verwendeten Sprachen haben muss. Ein generelles Problem mit Sprachen, die nicht stark verbreitet sind, ist das Tooling. Beispielsweise ist es für viele Unternehmen nicht sinnvoll, eine Sprache zu verwenden, die in der Theorie leistungsfähiger ist, wenn es dazu keine ausreichende Unterstützung durch IDEs gibt. Im Fall von DSLs muss man die Entwicklungskosten für ein geeignetes Tooling einkalkulieren.

6.2.9 Wartung von Software-Architekturen

In den vorhergehenden Teilabschnitten von Abschnitt 6.2 wurden die Mittel – der Einfachheit halber – aus Sicht des sogenannten Forward Engineering dargestellt. Forward Engineering hat die Perspektive, dass man ein System neu erstellt. In diesem Abschnitt wird diese Perspektive um die Sicht der Software-Wartung ergänzt, wo man es mit existierenden Systemen zu tun hat. Alle Mittel, die in diesem Abschnitt vorgestellt wurden, sind sowohl im Rahmen des Forward Engineering als auch in der Wartung einsetzbar.

Software-Wartung und der wichtige Teilbereich Reengineering sollen hier allgemein angerissen werden, aber es soll daneben speziell auch auf architekturverbessernde Maßnahmen im Rahmen der Wartung eingegangen werden.

Wartung von Software-Systemen (englisch: *software maintenance*) beschäftigt sich mit Änderungen am System nach dessen Auslieferung. Dies umfasst beispielsweise das Korrigieren von Fehlern, die Verbesserung von Qualitätsattributen wie Performanz und die Evolution bzw. Weiterentwicklung des Systems. Laut einer Studie von Nosek und Palvia [Nosek und Palvia 1990] beschäftigen sich nur 20 % des Zeitaufwandes, der in ein Software-System investiert wird, mit der eigentlichen Entwicklung; hingegen entfallen ca. 40 % auf das Verstehen und nochmals ca. 40 % auf das Ändern des Software-Systems – also auf typische Aktivitäten der Wartung von Software-Systemen.

In der realen Software-Entwicklung lassen sich zwei wichtige Gesetzmäßigkeiten beobachten [Lehmann und Belady 1985]: das Gesetz des ständigen Wandels und das Gesetz der wachsenden Komplexität. Das heißt, bezogen auf die Wartung von Software-Architekturen muss der Architekt ständig mit Änderungen rechnen und trotzdem bemüht sein, die Komplexität im Griff zu halten. In der Realität gestaltet sich das aus folgenden Gründen oft schwierig:

- Es ist schwer, sich auf nicht erwartbare Änderungen vorzubereiten.
- Wissen wird nicht hinreichend dokumentiert oder Dokumentationen gehen verloren – bei manchem Altsystem ist nicht einmal mehr der Quelltext vorhanden. Wenn die ursprünglichen Architekten und Entwickler das Unternehmen verlassen haben, ist es schwer, das Wissen über das System wiederzugewinnen.
- Bei raschen Änderungen wird – im hektischen täglichen Geschäft – leicht das Nachziehen der Dokumentation und der Entwurfs-, Archi-

Forward Engineering

Motivation

Ständiger Wandel und wachsende Komplexität

tekur- und Anforderungsdokumenten vergessen. Das heißt, das Prinzip der Nachvollziehbarkeit wird verletzt.

- > Schnelle Änderungen und Korrekturen, insbesondere wenn sie von anderen Personen durchgeführt werden, als denjenigen, die die Architektur ursprünglich entworfen haben, neigen dazu, architektonische Konventionen zu verletzen.

Techniken der Software-Wartung beschäftigen sich damit, diese Probleme zu behandeln. In der Folge sollen die wichtigsten Disziplinen der Software-Wartung kurz angerissen werden.

Reengineering

Aus Architektur-Sicht ist ein wichtiger Teil der Software-Wartung das Software Reengineering. Es beschäftigt sich mit den folgenden wesentlichen Aufgaben [Chikofsky und Cross 1990]:

- > *Reverse Engineering* bezeichnet Aktivitäten der Wiedergewinnung verlorener Informationen über existierende Software-Systeme. Hier müssen in erster Linie die Systembausteine identifiziert werden. Ferner müssen deren Interaktionen und Beziehungen rekonstruiert werden. Ziel ist die Beschreibung einer Sicht auf ein System, die einen höheren Grad an Abstraktion hat. Beispielsweise kann dies die Rekonstruktion einer Architektur auf Basis des Codes sein. Viele Reverse Engineering Tools setzen Visualisierungen ein, um eine Architektur zu veranschaulichen.
- > *Restrukturierung* bezeichnet alle Aktivitäten der Änderung der Struktur eines Systems. Dies kann sich sowohl auf den Code wie auch auf andere mit dem System zusammenhängende Dokumente beziehen. So können auch Architektur-Dokumente restrukturiert werden. Restrukturierung ist also in erster Linie die Änderung einer Repräsentation in eine andere Repräsentation auf derselben Abstraktionsebene.
- > *Software-Evolution* bezeichnet die Implementierung von Änderungen am Software-System. Man unterscheidet hier insbesondere Techniken für die Einbringung erwartbarer und unerwarteter Änderungen.
- > *Wrapping* verleiht einem gegebenen System bzw. einem Baustein eine neue Schnittstelle, ändert aber das System selbst nicht. Wrapping wird u. a. häufig für Versionsanpassungen oder andere leichte Schnittstellenänderungen an Software-Bausteinen verwendet. Es wird auch für die Evolution von Software eingesetzt: Wenn man ein großes System komplett ändern will, ist es oft nicht klug, es komplett neu zu schreiben. Stattdessen ist eine schrittweise Änderung besser. Dann müssen aber, beispielsweise für das Testen des Sys-

tems, die neuen und alten Teile miteinander kommunizieren können. Hier kann ein Wrapper ein altes Subsystem als Baustein im neuen System zur Verfügung stellen. Zum Teil werden Wrapper auch für Interimslösungen eingesetzt, wenn ein altes System bald ausgewechselt werden soll. In der Realität überleben solche Interimslösungen aber oft länger als geplant.

Der Begriff *Reengineering* umfasst also alle Aktivitäten zum Verstehen und Ändern eines Software-Systems, um es in neuer Form zu implementieren.

Beim Software Reengineering hat man es oft mit Altsystemen (englisch: *legacy system*) zu tun, also Systemen, die schon eine lange Zeit vorhanden sind, aber trotzdem noch einen Wert für das Unternehmen darstellen. Das Neuentwickeln des Systems wird als problematischer eingeschätzt, als das Altsystem an neue Gegebenheiten anzupassen – zum Beispiel aus Kostengründen.

Reengineering hat aber nicht nur mit Altsystemen zu tun. Gerade moderne objektorientierte oder komponentenorientierte Systeme erlauben durch ihre zusätzlichen Abstraktionen, wie Klassenhierarchien oder explizite Schnittstellen, relativ einfach Restrukturierungen, das Nachvollziehen der Architektur oder Refactoring. Das heißt, durch die zusätzliche Abstraktion und Modularität ist es in diesen Systemen einfacher möglich, die Architektur inkrementell zu verbessern und zu erweitern – was einen stärkeren Fokus auf Reengineering-Aktivitäten bedeutet.

Beispielsweise können hier die anderen, in diesem Abschnitt eingeführten Mittel wie Aspektorientierung, generatives Programmieren und modellgetriebene Entwicklung zum Einsatz kommen.

Leider haben in der Praxis architekturverbessernde Maßnahmen oft nur einen geringen Stellenwert. Dies ist dadurch begründet, dass diese Maßnahmen relativ viel Aufwand bedeuten, aber der kurzfristige Geschäftsnutzen nicht absehbar ist. Beispielsweise kann die Verbesserung der Modularität sehr wichtig sein für die Verständlichkeit, Änderbarkeit und Wiederverwendbarkeit der Architektur – aber eine neue Funktionalität ist trotzdem meist besser geeignet, einen Kunden von einer Investition in ein Produkt zu überzeugen.

Einsatzgebiete des Reengineering

Architekturverbessern-de Maßnahmen

Reengineering durch Umgebungseinflüsse

Aus diesem Grund sind wichtige Aufgaben des Reengineering in der Praxis eher durch Änderungen der „Umgebung“ beeinflusst; einige Beispiele sind:

- > Oft werden Systeme mit neuen Programmiersprachen, neuen Standards, neuen Plattformen, neuen Middleware-Produkten, Datenbanken und anderen neuen Umgebungselementen konfrontiert, die integriert werden müssen oder zu denen die Systeme migriert werden müssen.
- > Oft kommt es auch vor, dass ein System eine neue Benutzerschnittstelle benötigt. Beispielsweise sind Mainframe-Systeme oft zeichenorientiert und müssen zu grafischen Benutzerschnittstellen migriert werden. Heutzutage bekommen viele Systeme eine (zusätzliche) Web-Schnittstelle.
- > Mainframe-Anwendungen werden auch oft zu Client-Server-Architekturen verändert oder in diese integriert.

Architekt muss auf architekturverbessernde Maßnahmen hinwirken

Der Architekt hat die Aufgabe, darauf zu achten und Interessenvertreter zu überzeugen, dass architekturverbessernde Maßnahmen einen ausreichenden Stellenwert im Rahmen der Wartungsarbeiten bekommen. Insbesondere muss er auf die Langzeitfolgen mangelnder Architekturverbesserung hinweisen. Kapitel 8 beschäftigt sich ausführlich mit diesen und ähnlichen Kommunikationsaufgaben des Architekten.

Unterscheidung zum Forward Engineering

Im Zusammenhang mit Reengineering unterscheidet man häufig den Begriff „Forward Engineering“. Hierunter versteht man in erster Linie Situationen, in denen man ein System neu entwickelt. Im extremen Fall sind beim Forward Engineering das Problem und die möglichen Lösungsalternativen noch völlig unklar und der Aufwand ist nur schwer abschätzbar. Man kann die Architektur mit vielen Freiheitsgraden entwerfen, ohne Abhängigkeiten von Altsystemen beachten zu müssen.

Im Reengineering hat man es gewöhnlich mit der gegenteiligen Situation zu tun: Es gibt viele Abhängigkeiten, aber dafür hat man bereits einiges an Erfahrung und kann oft recht gut abschätzen, was der Reengineering-Aufwand bedeutet. Gerade die Abschätzbarkeit bedeutet in der Praxis oft, dass das Reengineering dem Forward Engineering – wenn möglich – vorgezogen wird, um Risiken kontrollierbar zu halten.

Reengineering-Werkzeuge

Für die verschiedenen Reengineering-Techniken gibt es eine Vielzahl an Werkzeugen.

- > Für die *Programmanalyse* stehen klassische Werkzeuge, wie grep, diff und Debugger, zur Verfügung, aber auch weiter gehende Werkzeuge z. B. zur Architektur-Visualisierungen, Analysewerkzeuge auf Basis von Syntaxbäumen oder Kontrollflüssen sowie Werkzeuge zur statischen und dynamischen Merkmalsanalyse.
- > Auf Basis der Programmanalysewerkzeuge kann man automatisch *Metriken* berechnen, also Maßzahlen über die Software-Architektur oder das Software-System.
- > Mittels Lastgeneratoren und Profilern kann man *Performanz-Analysen* durchführen.
- > Das *Wrapping* kann mit Wrapper-Generatoren automatisiert werden.

Werkzeuge für das *Refactoring* erlauben es, eine Architektur schrittweise zu verfeinern. Einfache Refactorings, wie „Verschiebe Methode“ oder „Benenne Klasse um“ werden von modernen Entwicklungsumgebungen, wie Eclipse, bereits zur Verfügung gestellt. Weiter gehende Tools erlauben die *Remodularisierung*, das *Clustering* von Strukturen, um Modul-Abhängigkeiten aufzufinden, sowie die *Analyse und Restrukturierung von Vererbungshierarchien*.

6.2.10 Zusammenfassung

- > Es gibt eine Reihe grundlegender architektonischer Konzepte, die in vielen Architekturen und anderen Architektur-Mitteln Verwendung finden.
- > Prozedurale Ansätze sind ein klassischer Ansatz der Strukturierung von Architekturen und werden insbesondere zur Zerlegung eines komplexen Algorithmus in wiederverwendbare Teilalgorithmen benutzt.
- > Die Objektorientierung ist eine Weiterentwicklung und basiert auf der Idee, die Daten, die eine Reihe von zusammenhängenden Methoden gemeinsam bearbeiten, mit diesen Methoden zu bündeln. Objektorientierung ist ein heute vorherrschendes architektonisches Konzept.
- > Die Komponentenorientierung ist eine Weiterentwicklung und bietet Komponenten als wiederverwendbare, in sich geschlossene Bausteine einer Architektur an.
- > Die Idee der Meta-Programmierung ist es, durch eine zusätzliche Abstraktionsebene in Software-Systemen eine höhere Flexibilität und Kontrolle zu erreichen.
- > Reflection erlaubt Programmen beispielsweise Typinformationen, Informationen über Klassen (Attribute und Operationen) und Vererbungshierarchien abzufragen.

**Zusammenfassung:
Grundlegende
Architektonische
Konzepte**

- > Die generative Erzeugung von Systembausteinen verfolgt das Ziel, den Automatisierungsgrad bei der Erstellung von Software zu erhöhen.
- > Von modellgetriebener Software-Entwicklung spricht man, wenn Modelle nicht lediglich zu Dokumentationszwecken eingesetzt werden, sondern zentrale Artefakte eines lauffähigen Systems bilden.
- > Die Aspektorientierung vermeidet über den Code oder den Entwurf verstreute Lösungen für sogenannte Crosscutting Concerns.
- > Eine Skriptsprache ist ursprünglich eine Programmiersprache, die Software-Systeme kontrolliert oder steuern sollen. Heute werden Skriptsprachen aber auch für alle möglichen anderen Zwecke als Sprachen auf hoher Abstraktionsebene eingesetzt.
- > Dynamische Sprachen bezeichnen allgemein Sprachen auf hohem Abstraktionsniveau, die während der Laufzeit viele Aufgaben ausführen, die andere Sprachen zu Kompilierzeit durchführen.
- > Wartung von Software-Systemen beschäftigt sich mit Änderungen am System nach dessen Auslieferung. Dies umfasst beispielsweise das Korrigieren von Fehlern, die Verbesserung von Qualitätsattributen wie Performanz und die Evolution bzw. Weiterentwicklung des Systems.
- > Software Reengineering beschäftigt sich mit den folgenden wesentlichen Aufgaben: Reverse Engineering, Restrukturierung, Software-Evolution und Wrapping.
- > Leider haben in der Praxis architekturverbessernde Maßnahmen nur einen geringen Stellenwert.
- > Reengineering findet oft eher durch Umgebungseinflüsse statt.
- > Der Architekt muss auf ausreichende architekturverbessernde Maßnahmen hinwirken.

6.3 Architektur-Taktiken, -Stile und -Muster

Taktiken, Stile und Muster

Die im Abschnitt 6.1 besprochenen Prinzipien erklären recht allgemein, wie man eine „gute“ Software-Architektur entwirft und (weiter-)entwickelt. Die im Abschnitt 6.2 besprochenen Konzepte sind konkreter in der Handlungsanweisung, beziehen sich aber, wie auch die Prinzipien, nicht direkt auf bestimmte Problemstellungen, mit denen man es beim Entwerfen einer Architektur zu tun hat, sondern auf das allgemeine Problem des Software-Entwurfs. Dieser Abschnitt beschäftigt sich mit architektonischen Taktiken, Stilen und Mustern. Diese drei Mittel haben gemeinsam, dass sie prinzipielle Lösungen zu bestimmten wiederkeh-

renden Problemstellungen im Architektur-Entwurf beschreiben. Dies geschieht in einer Form, die auf eine Vielzahl von Fällen anwendbar ist. Das heißt, Taktiken, Stile und Muster abstrahieren von schon einmal getroffenen Entwurfsentscheidungen, die in ähnlichen Kontexten in der Vergangenheit zu erfolgreichen Software-Architekturen geführt haben. Taktiken, Stile und Muster dienen also der Wiederverwendung von Entwurfsentscheidungen.

Abbildung 6.3-1 veranschaulicht die Beziehungen von Architektur-Taktiken, Architektur-Stilen und Architektur-Mustern zu Prinzipien und Basiskonzepten. Alle drei Mittel helfen bei der Konkretisierung und Umsetzung von Prinzipien und nutzen die Basiskonzepte in der Umsetzung. Muster sind das allgemeinste der drei Mittel, denn sie werden nicht nur in der Architektur, sondern auch in vielen anderen Domänen eingesetzt. Überdies bieten Muster das Konzept der Mustersprache an, das eine Anzahl von zusammenhängenden Mustern kombiniert. Muster und Stile sind konzeptuell gesehen sehr ähnliche Mittel. Taktiken sind etwas allgemeiner als Stile und Muster. Wenn man Taktiken anwendet, kann man oft Stile und Muster verwenden, um die Taktiken umzusetzen. Bevor Taktiken diskutiert werden, sollen aber zunächst Qualitätsattributszenarien als ein Mittel zur Anforderungsanalyse eingeführt werden, die als Voraussetzung der Anwendung von Taktiken dienen.

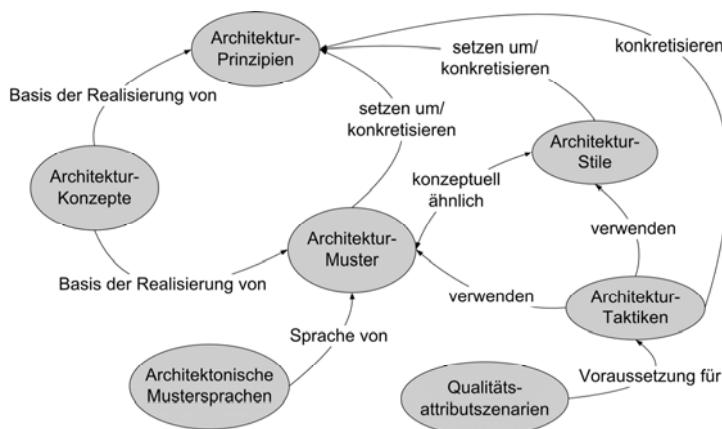


Abb. 6.3-1: Architektur-Taktiken, -Stile und -Muster im Überblick.

6.3.1 Qualitätsattributszenarien

Qualitätsattribut-szenarien

Taktiken benötigen eine Analyse der Qualitätsattribute als Voraussetzung. Ein geeignetes Mittel dazu sind Qualitätsattributszenarien [Bass et al. 2003], die in diesem Abschnitt kurz eingeführt werden sollen.

Bei Mustern und Stilen muss man auch die wesentlichen Anforderungen hinsichtlich der Qualitätsattribute analysieren. Auch dazu können Qualitätsattributszenarien benutzt werden – z. B. in Kombination mit Taktiken, die dann durch Muster und Stile verfeinert werden.

Dokumentation von Qualitäten mit Qualitätsattributszenarien

In der Praxis hat es sich etabliert, funktionale Anforderungen mittels Anwendungsfällen zu beschreiben. Die Dokumentation von Qualitäten, die man z. B. mit Qualitätsattributszenarien umsetzen kann, ist noch nicht so etabliert. In Tabelle 6.3-1 wird das Dokumentationsschema von Qualitätsattributszenarien eingeführt und erklärt. Manche Qualitäten gelten für das System als Ganzes. Andere Qualitäten gelten jedoch nur für einen Teil der funktionalen Anforderungen. Qualitätsattributszenarien können sowohl für systemweite als auch für anwendungsfallspezifische Qualitäten genutzt werden.

Tab. 6.3-1: Dokumentationsschema von Qualitätsattributszenarien.

Kriterium	Bedeutung
Quelle	System oder Benutzer, der ein Ereignis bzw. Stimulus generiert.
Stimulus	Ereignis, auf welches das System reagieren muss.
Artefakt	Der Teil des Systems, der von dem eintreffenden Ereignis betroffen ist.
Kontext	Beschreibt den Kontext, in dem das Ereignis eintrifft. Z. B.: „System befindet sich im Normalzustand“.
Reaktion	Beschreibt die Aktivität, die der Stimulus im System auslöst. Eine mögliche Reaktion wäre das Verwerfen der Vorfallsart und das Schreiben eines Eintrags in das Betriebslog.
Reaktionsmessgröße	Beschreibt die Messgröße sowie die Art und Weise, wie der Erfolg bzw. Misserfolg der erfolgten Reaktion zu messen ist.

Man kann Qualitätsattributszenarien nach Qualitätsattributen in Szenariotypen einteilen. Z. B. gibt es die folgenden allgemeinen Szenariotypen [Bass et al. 2003]:

- > Verfügbarkeitsszenarien
- > Änderbarkeitsszenarien
- > Performanzszenarien

- > Sicherheitsszenarien
- > Testbarkeitsszenarien
- > Benutzbarkeitsszenarien

Es liegt nahe, architekturelle Anwendungsfälle mit architekturellen Qualitäten in Verbindung zu bringen. Hierzu betrachtet man jeden architekturellen Anwendungsfall hinsichtlich seiner qualitativen Realisierungsgüte. Je mehr verschiedene Qualitätsmerkmale (z. B. Sicherheit, Nachvollziehbarkeit, Performanz) ein Anwendungsfall aufweist, umso größer ist sein architektonischer Charakter. Für jede Kombination können Qualitätsattributszenarien erstellt werden.

Abschließend sollte untersucht werden, wie die Qualitätsattributszenarien innerhalb der definierten Rahmenbedingungen realisiert werden können. Dabei wird manchmal deutlich werden, dass die Erreichung eines gewünschten Qualitätsattributszenarios innerhalb der gesetzten Rahmenbedingungen nicht möglich ist. In diesem Fall ist man als Architekt gefordert, auf diese Widersprüche hinzuweisen und Interessenvertretern Alternativen aufzuzeigen.

6.3.2 Architektur-Taktiken

Architektur-Taktiken gehen von einer Analyse der nichtfunktionalen Anforderungen aus und die Taktiken geben – anhand von Qualitätsattributen – Richtlinien für die mögliche Verbesserung der Architektur. Eine Taktik ist also im Prinzip eine Hilfe für den Architekten, eine erste Idee zu einem Entwurfsproblem zu erhalten. Diese Idee wird er dann weiter ausarbeiten. Dazu kann er z. B. Stile und Muster als weitergehende Mittel verwenden.

Qualitätsattributszenarien sind eine Möglichkeit der Analyse der Anforderungen als Basis für Taktiken. Andere Techniken könnten natürlich auch als Basis für eine Taktik zum Einsatz kommen. Neben der Erhöhung des Formalisierungsgrades von Anforderungsbeschreibungen, bereiten Qualitätsattributszenarien durch ihre Reaktionsmessgrößen unmittelbar die Überprüfung der Realisierungsgüte der Taktik vor. Dadurch erlauben sie schnelles und direktes Operationalisieren entsprechender Messmetriken.

Eine Taktik ist eine Entwurfsentscheidung, die Einfluss nimmt auf die Realisierung der Reaktion eines Qualitätsattributszenarios.

Verbinden von funktionalen Anforderungen und Qualitäten

Architekturelle Anwendungsfälle und Qualitäten innerhalb der gesetzten Rahmenbedingungen betrachten

Architektur-Taktiken

Rolle von Qualitätsattributszenarien

**Definition:
Taktik**

In Ergänzung zu den Qualitätsattributzenariotypen, die im vorherigen Abschnitt kurz eingeführt wurden, bieten Bass et al. [Bass et al. 2003] überdies eine Sammlung genereller Taktiken im Umgang mit wiederkehrenden Problemen je Szenariotyp an. Man kann also pro Szenariotyp eine generelle Taktik ableiten, die dann von einigen konkreteren Taktiken, die Alternativen zueinander sind, verfeinert wird. Abbildung 6.3-2 gibt als Beispiel einen Überblick über Änderbarkeitstaktiken, die eine Ableitung der Änderbarkeitsszenarien sind.

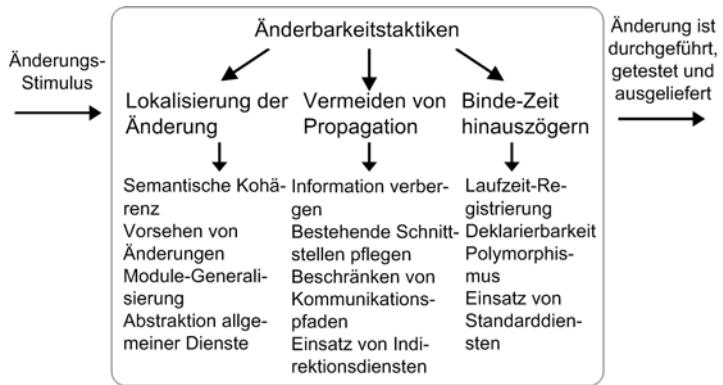


Abb. 6.3-2: Beispiel: Änderbarkeitstaktiken nach [Bass et al. 2003].

Ein Architekt nutzt Qualitätsattributzenarien sowie entsprechende Taktiken, um die Vollständigkeit der ermittelten nicht-funktionalen Anforderungen zu gewährleisten.

Taktiken versus Muster und Stile

In Bezug auf Taktiken ähnelt ein Qualitätsattributzenario der Problemstellung und dem Kontext eines Musters. Jedoch verfügen Muster über keine Analysetechnik, wie die Qualitätsattributzenarien, um konkrete Systeme zu analysieren. Taktiken ähneln der Lösung eines Musters, aber sie sind in der Regel weit weniger detailliert beschrieben. Wie Stile decken Taktiken aber auch nur einen ganz speziellen Anwendungsbereich ab und sind somit ein spezialisierteres Architekturmittel als Muster. Taktiken nehmen speziell auf Qualitätsattribute Bezug. Muster nehmen auch auf Qualitätsattribute der Architektur Bezug, verfügen aber auch über viele andere Arten von Kräften. Z. B. findet man in einer Musterbeschreibung auch Kräfte, die keinen direkten Bezug zur Architektur haben, wie die Umsetzung einer Entwurfsentscheidung im Entwurfsprozess oder strategische Managementüberlegungen, die eine Entwurfsentscheidung beeinflussen können.

Aus diesen Gründen macht es Sinn, Taktiken einzusetzen, um sich über ein generelles Vorgehen klar zu werden und dabei Qualitätsattribute

allgemein zu analysieren. Für die detaillierte Ausgestaltung der Entwurfsentscheidung bieten dann Muster und Stile konkrete Handlungsanweisungen. Die spezifischen Kräfte, die z. B. in einem Muster vorkommen und auch Qualitätsattribute umfassen, helfen bei der Entwurfsentscheidung. Z. B. könnte die Entscheidung für die Änderungstaktik „Lokalisierung der Änderung“ nach weiterer Analyse der Anforderungen dazu führen, dass man eine Schichtenarchitektur einführt, wie sie durch das Layers-Muster bzw. den Layers-Stil beschrieben wird (siehe Abschnitt 6.4.1 für Details).

6.3.3 Architektur-Stile

Als Nächstes sollen in diesem Abschnitt architektonische Stile behandelt werden. Shaw und Garlan [Shaw und Garlan 1996] definieren einen Architektur-Stil als ein Muster der strukturellen Organisation einer Familie von Systemen. Ein Architektur-Stil besteht bei Shaw und Garlan aus den folgenden Elementen:

- > Eine Menge von *Komponententypen*, die bestimmte Funktionen zur Laufzeit erfüllen.
- > Eine *topologische Anordnung* dieser Komponenten.
- > Eine Menge von *Konnektoren*, die die Kommunikation und Koordination zwischen den Komponenten regeln:
- > Eine Menge von *semantischen Einschränkungen*, die bestimmen, wie Komponenten und Konnektoren miteinander verbunden werden können.

Architektur-Stile

Es sei darauf hingewiesen, dass sowohl die Komponenten als auch die Konnektoren eines Stils meistens als eigenständige Bausteine einer Architektur bzw. eines Systems realisiert werden.

Ein Architektur-Stil gibt in erster Linie die fundamentale Struktur eines Software-Systems und dessen Eigenschaften wieder. Ein Stil kann also genutzt werden, um Architekturen zu kategorisieren. Ferner kann man Stile dazu verwenden, um die Konsequenzen einer fundamentalen Architektur und ihrer Varianten zu verstehen.

Es ist recht schwer, Architektur-Stile und Architektur-Muster voneinander zu unterscheiden – bis auf den Punkt, dass verschiedene Beschreibungsformen gewählt werden. So deckt die Beschreibungsform von Mustern viele Aspekte ab, wie z. B. die Gründe hinter einer Entwurfsentscheidung, die in der Beschreibungsform von Stilen nicht vorkom-

Stile versus Muster

men. Muster kommen nicht nur in der Architektur, sondern auch in anderen Bereichen zum Einsatz, wohingegen das Mittel der Stile bislang nur in diesem Bereich verwendet wird. Viele der Architektur-Stile, die von Shaw und Garlan dokumentiert wurden, wurden auch in Form von Architektur-Mustern dokumentiert. Aus diesem Grund und da Architektur-Stile ein sehr musterähnliches Konzept sind, kann man in der Anwendung die Begriffe Stile und Muster im Architektur-Bereich synonym verwenden.

Shaw und Garlan haben einige häufig verwendete Architektur-Stile katalogisiert. Diese werden in Abbildung 6.3-3 zusammengefasst und in Kategorien unterteilt.

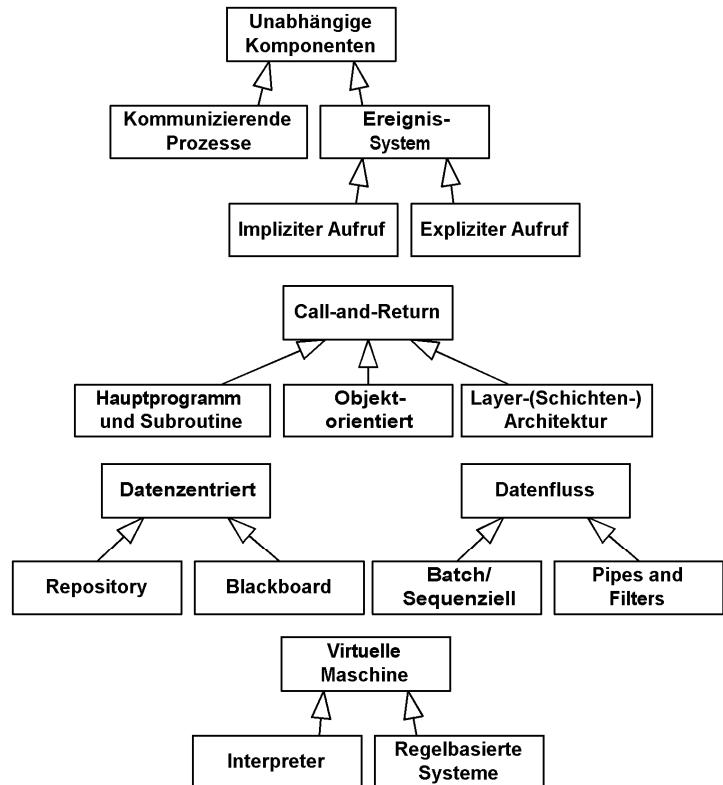


Abb. 6.3-3: Überblick und Kategorisierung der Architektur-Stile.

Beispiel eines Stils: **Pipes and Filters**

Als Beispiel soll nun der Architektur-Stil *Pipes and Filters* betrachtet werden. Dieser hat das Ziel, eine flexible Architektur für das sequenzi-

le Abarbeiten von Datenströmen zu beschreiben. Pipes and Filters hat einen Komponententyp:

- > *Filter* transformieren inkrementell Ströme von Eingabedaten in Ströme von Ausgabedaten.

Der Stil hat auch einen Konnektortyp:

- > *Pipes* bewegen Daten von einer Filterausgabe zu einer Filtereingabe.

Die topologische Anordnung dieser Komponenten und Konnektoren wird in der folgenden Abbildung 6.3-4 mit einem Beispiel veranschaulicht. Ein Filter kann über mehrere Pipes mehrere andere Filter mit Input-Daten versorgen. Die Verarbeitung läuft (nicht deterministisch) so lange, bis keine Pipe mehr an dem letzten Filter hängt und somit die Verarbeitung terminiert.

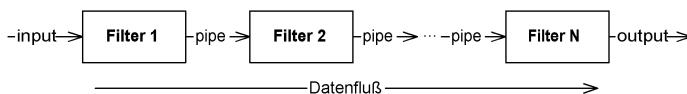


Abb. 6.3-4: Pipes-and-Filters-Architektur.

Der Stil hat eine Reihe von Invarianten:

- > Filter sind unabhängige Verarbeitungskomponenten. Externe Daten werden nur durch Inputs und Outputs in das System gegeben.
- > Die Identität anderer Filter ist einem Filter in einer Pipes-and-Filters-Architektur nicht bekannt.
- > Filter können mittels Pipes in beliebiger Reihenfolge kombiniert werden.

Typische Beispiele von Pipes-and-Filters-Architekturen sind:

- > *UNIX Pipes*, die UNIX-Programme miteinander über Interprozess-kommunikation verbinden.
- > *Compiler*, die eine Verarbeitung schrittweise durchführen und das Ergebnis jeweils nach jedem Verarbeitungsschritt weiterreichen. Typische Schritte sind lexikalische Analyse, Parsen etc.

Eine Pipes-and-Filters-Architektur bietet den Vorteil, dass sie sehr flexibel ist, was die Kombination von Pipes und Filtern angeht. Filter-Komponenten können leicht wiederverwendet werden. Es ist einfach, Pipes und Filters parallel arbeiten zu lassen, beispielsweise in getrenn-

ten Prozessen oder Threads, da sie recht unabhängig voneinander sind, was die Effizienz des Gesamtsystems erhöhen kann.

Allerdings gibt es auch einige mögliche Nachteile. Das Weiterreichen des Zustandes zwischen Filtern kann einen hohen Aufwand und Ressourcenverbrauch nach sich ziehen. Wenn Daten transformiert werden müssen, um in die Pipe gelegt zu werden, kann es zu unnötigen Hin- und Zurück-Transformationen kommen. Das Debugging oder Verhalten bei Fehlern kann problematischer sein als bei anderen Architekturen, da auch Fehler bzw. Debugging-Information durch die Pipes geschickt werden müssen.

Im POSA-Buch [Buschmann et al. 1996] wird der Stil Pipes and Filters auch als Architektur-Muster dokumentiert.

6.3.4 Architektur-Muster

Muster im Software Engineering

Im Laufe der letzten Jahre sind Software-Muster (englisch: *software patterns*) zu einem wichtigen Instrument des Software-Entwicklers und -Architekten geworden. Insbesondere im Bereich der Objektorientierung haben Muster eine wichtige Bedeutung bekommen. Hier sind insbesondere das Gang-of-Four-Buch (GoF) [Gamma et al. 1995], das sich mit Entwurfsmustern (englisch: *design patterns*) beschäftigt, sowie die POSA-Bücher [Buschmann et al. 1996, Schmidt et al. 2000], die sich mit Software-Architekturmustern beschäftigen, als wichtige Beiträge zu nennen. Es gibt aber auch Muster in vielen anderen Bereichen der Software-Entwicklung, wie beispielsweise Muster für die Analyse von Domänen [Fowler 1996], Muster für den domänengebundenen Entwurf [Evans 2004], Muster für die Software-Organisation [Coplien und Harrison 2004] oder pädagogische Muster [Fricke und Völter 2000].

Kontext, Problem, Lösung und Kräfte

Die ursprüngliche Musterdefinition von Christopher Alexander – der das Musterkonzept im Bereich der klassischen Architektur ursprünglich eingeführt hat – besagt [Alexander 1977]:

Ein Muster ist eine dreiteilige Regel, die die Beziehung zwischen einem bestimmten Kontext, einem Problem und einer Lösung ausdrückt.

Alexander geht jedoch in vielen Punkten über diese einfache Definition hinaus. Diese Punkte sollen in den nächsten Paragraphen angesprochen werden. Eine eingängige Zusammenfassung dieser Punkte – bezogen auf ein Software-System – bietet die etwas längere Definition von Coplien [Coplien 2004]:

Jedes Muster ist eine dreiteilige Regel, die die Beziehung zwischen einem bestimmten Kontext, einem bestimmten System an Kräften (englisch: *forces*), die in diesem Kontext wiederkehrend auftreten, und einer bestimmten Software-Konfiguration, die diesen Kräften erlaubt, sich gegenseitig aufzulösen, ausdrückt.

Ein sehr wichtiger Punkt ist, dass Muster prinzipielle Lösungen für wiederkehrende Probleme sind. Das heißt, dass sie so allgemein formuliert sein müssen, dass ein Muster nicht nur für ein bestimmtes Problem anwendbar ist, sondern für eine ganze Reihe von konkreten Problemen. Auf der anderen Seite sind Muster aber auch ein praktischer Ansatz: das heißt, nach dem Lesen des Musters sollte der Leser eine klare Lösungsanweisung an der Hand haben, wie er ein konkret gegebenes Problem lösen kann, das auf das Problem in der Musterbeschreibung passt. Allerdings muss noch eine Anpassung der allgemeinen Lösung, die in dem Muster beschrieben ist, auf die konkrete Entwurfssituation stattfinden.

Der Architekt sollte die zentralen Entwurfs- und Architektur-Muster gut zu kennen, da diese die typischen wiederkehrenden Lösungen zu den sich wiederholenden Problemen in der Software-Architektur im Allgemeinen darstellen. Überdies sollte der Architekt auch die Entwurfsmuster und Architektur-Muster der konkreten technischen und nicht-technischen Domänen, in denen er tätig ist, gut beherrschen. Dies ist wichtig, um neue Probleme derselben Art lösen zu können, ohne das „Rad neu erfinden“ zu müssen. Muster sind also ein Mittel, etabliertes Wissen weiterzugeben.

Zu diesem Zweck geben Muster den Architekten und Entwicklern ein gemeinsames Vokabular an die Hand, um wiederkehrende Architektur-Strukturen benennen zu können. Wenn man die zentralen Muster in einem Bereich einmal gut beherrscht, wird man schnell sehen, dass man diese Muster in gegebenen Architekturen immer wieder vorfindet. Deshalb sind Muster auch ein wichtiges Instrument für die Dokumentation und die Diskussion von Software-Architekturen. Sie versetzen den Architekten in die Lage, die Gemeinsamkeiten in wiederkehrenden architektonischen Strukturen zu erkennen und zu benennen.

Wiederkehrende Probleme und Lösungen

Known Uses

Ein wichtiger Punkt ist, dass Muster keine neuen Ideen beschreiben, sondern bewährte Lösungen darstellen. Für Muster gibt es deshalb allgemein die Forderung, dass ein Software-Muster immer mindestens drei bekannte Anwendungen (englisch: *known uses*) in realen, praktischen Systemen haben muss. Solche Known Uses sind oft als Teil des Musters dargestellt, um dem Leser die praktische Nutzung des Musters zu veranschaulichen.

Kräfte

Ein wichtiger Bestandteil jedes Musters ist ein System von Kräften (englisch: *forces*). Diese Kräfte sind Teil des Problems, das durch das Muster gelöst wird. In erster Linie bauen sie Spannung auf, die durch die Lösung wieder aufgelöst wird. Die Lösung muss also eine Balance zwischen den Kräften herstellen. Die folgende Abbildung 6.3-5 veranschaulicht dies. Natürlich kann das Muster diese Balance nur allgemein beschreiben. Der Architekt muss die Ausbalancierung der Kräfte für eine konkrete Lösung in einer Entwurfssituation dem Muster folgend erarbeiten. Im Bereich der Software-Architektur sind oft die Qualitätsattribute (siehe Kapitel 5) wichtige Kräfte für eine Lösung. In der Abbildung wird gezeigt, wie verschiedene Kräfte, die allesamt typische architektonische Qualitätsattribute sind, eine Lösung beeinflussen.

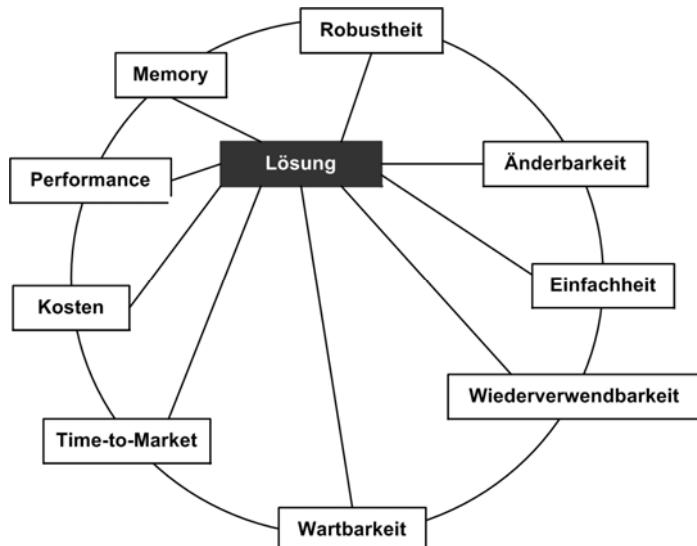


Abb. 6.3-5: Qualitätsattribute als Kräfte, die eine Lösung beeinflussen.

Konsequenzen

Nicht immer bietet jede Lösung eine gute Balance zwischen allen gegebenen Kräften in einer Problemsituation. Beispielsweise kann eine sehr effiziente Lösung (durch lange Entwicklungszeiten für diese spezielle

Lösung) die Kosten in die Höhe treiben oder den Ressourcenverbrauch negativ beeinflussen. In solchen Fällen können auch mehrere Muster, die ein ähnlich gelagertes Problem unterschiedlich lösen, existieren. Beispielsweise können die Muster Layers [Buschmann et al. 1996] (siehe Abschnitt 6.4.1) und Pipes and Filters [Buschmann et al. 1996] (siehe Abschnitt 6.4.2) unter Umständen Alternativen für einen Entwurf sein, der Verantwortlichkeiten aufteilt. Pipes and Filters ist positiv für die Flexibilität, wenn relativ lineare Aufrufsequenzen abgebildet werden müssen. Layer eignet sich zum Beispiel besser, wenn komplexe Strukturen von Bausteine in einer Architektur verstehbar komponiert werden müssen.

In einem Muster werden die *Konsequenzen* des Mustereinsatzes explizit angegeben, damit der Leser abwägen kann, ob die Anwendung des Musters infrage kommt und welche positiven wie auch negativen Auswirkungen zu erwarten sind.

In einem Muster sollte explizit beschrieben werden, wie die Lösung die Kräfte auflöst und warum die Lösung die Kräfte gerade in dieser Art und Weise auflöst. Verschiedene Musterbeschreibungen machen dies in verschiedener Art und Weise. Insbesondere werden aber häufig eine detaillierte Lösung mit Beispielen, Beschreibungen der Lösungselemente des Musters (englisch: *participants*), Beschreibungen der Interaktion der Teile des Musters, Varianten des Muster und dem Verhältnis des Muster zu anderen Mustern angegeben. Oft findet man Teile der detaillierten Lösung, die sehr konkret sind, zum Beispiel in Form von UML-Diagrammen oder Quelltext-Fragmenten. Diese dienen der Veranschaulichung des Musters. Man sollte sie nicht mit dem *Muster an sich* verwechseln, das allgemeiner ist als alle diese Lösungsvarianten.

Detaillierte Lösung

Im Kontext der Software-Architektur spielen sowohl Entwurfsmuster als auch Architektur-Muster eine große Rolle. Sie haben gemeinsam, dass sie in der Regel strukturelle, technische Lösungen präsentieren. Somit sind sie von Mustern zu unterscheiden, die fachliche Aspekte behandeln (siehe z. B. [Fowler 1996] und [Evans 2004]). Muster für fachliche Aspekte dienen oft als Grundlage für den domänengebundenen Entwurf eines Analysemodells. In der Umsetzung des Analysemodells kommt es dann häufig zum Einsatz von Entwurfs- und Architektur-Mustern. Generell beschreiben Entwurfsmuster eher spezifische, lokal wirkende Entwurfslösungen, wohingegen Architektur-Muster eher Systemstrukturen, die systemweit wirken, beschreiben. Hierbei ist der Unterschied zwischen diesen beiden Kategorien von Mustern jedoch fließend.

Entwurfs- und Architektur-Muster

Unterscheidung ist abhängig vom Betrachtungswinkel

Beispiel: Entwurfsmuster Proxy

Beispielsweise ist das Muster Interpreter [Gamma et al. 1995] ursprünglich als Entwurfsmuster präsentiert worden und kann mittels einiger weniger Klassen implementiert werden. Dasselbe Muster kann aber auch die Grundlage komplexer Architekturen sein, wie z. B. bei der Verwendung von Interpreter als Architektur einer interpretierten Programmiersprache (siehe auch: Interpreter-Architektur-Stil in [Shaw und Garlan 1996]).

Typische Entwurfsmuster, wie die in [Gamma et al. 1995] beschriebenen Muster, werden oft als Teile der Lösung eines Architektur-Musters eingesetzt. Auch dies ist aber keine allgemeingültige Regel und hängt von der konkreten Architektur und der Betrachtungsweise der Muster ab.

Man sieht, dass die Unterscheidung zwischen Entwurfsmuster und Architektur-Muster von der Sicht des Betrachters und dem Zweck der Betrachtung abhängig ist.

Nun sollen zwei Muster als Beispiele betrachtet werden: das Entwurfsmuster Proxy und das Architektur-Muster Broker. Zunächst folgt eine Kurzfassung des Proxy-Musters [Gamma et al. 1995, Buschmann et al. 1996].

Name:	Proxy
Kontext:	Ein Klient muss auf die Operationen einer Instanz einer bestimmten Klasse zugreifen.
Problem:	Der direkte Zugriff auf die Operationen der Klasse ist nicht möglich, schwierig oder unangebracht. Beispielsweise kann der direkte Zugriff unsicher oder uneffizient sein oder man befindet sich in einem verteilten Umfeld. Hier mag es nicht gewollt sein, dass die physikalische Netzadresse für den direkten Zugriff auf ein verteiltes Objekt im Klienten hart-kodiert ist. Aber ohne diese Adresse ist ein direkter Zugriff über das Netzwerk nicht möglich.
Kräfte:	Der Zugriff auf eine Instanz einer bestimmten Klasse soll laufzeiteffizient und sicher sein. Dies kann mit einem direkten Aufruf nicht erreicht werden.
	Der Zugriff auf eine Instanz einer bestimmten Klasse soll transparent aus Sicht des Klienten sein. Insbesondere soll es nicht notwendig sein, dass

der Client in seinem üblichen Aufrufverhalten oder seiner üblichen Aufrufsyntax verändert werden muss.

Klientenentwickler sollen mögliche Auswirkungen eines Aufrufs kennen und einschätzen können. Vollständige Transparenz des Aufrufverhaltens aus Sicht des Klienten kann dies erschweren.

Lösung: Der Client kommuniziert mit einem Platzhalter, dem Proxy, anstatt mit einer Instanz der eigentlichen Klasse. Der Proxy bietet dieselbe Schnittstelle an wie die Instanzen der Klasse, die aufgerufen werden sollen. Intern leitet der Proxy den Aufruf an eine Instanz dieser Klasse weiter. Er kann aber auch zusätzliche Funktionalitäten, wie beispielsweise Authentifizierung, oder das Auslösen eines verteilten Aufrufs, implementieren.

Konsequenzen: Ein Proxy bietet den Vorteil, dass er den Klienten von der implementierenden Instanz entkoppelt. Beispielsweise heißt dies im verteilten Umfeld, dass der Client die Server-Netzadresse nicht hart-kodiert vorhalten muss. Somit erhöht der Proxy die Flexibilität der Anwendung, denn er erlaubt es einem Klienten, durch das Wechseln des Proxy-Objektes auch das Verhalten des aufgerufenen „Services“ zu beeinflussen. Ein Proxy kann die Laufzeiteffizienz steigern, beispielsweise dadurch, dass er Ergebnisse in einem Cache vorhält und diese dann ausliefert anstatt einer erneuten Berechnung.

Ein Proxy ist aber immer eine zusätzliche Indirektion, also mindestens ein zusätzlicher Aufruf. Das heißt, der Proxy reduziert die Laufzeiteffizienz leicht. Man sollte komplexe Proxy-Varianten vorsichtig einsetzen, denn eine komplexe Logik im Proxy kann einen erheblichen Aufwand, beispielsweise in Bezug auf den Ressourcenverbrauch, produzieren.

In der Regel ist die Basisstruktur eines Proxies mit einigen wenigen Klassen implementiert, wie in Abbildung 6.3-6 dargestellt. Die Abbildung zeigt kein Beispiel für eine Anwendung des Proxy-Musters für ein konkretes Problem, sondern ein Beispiel eines Lösungsschemas – ein Teil der Musterbeschreibung.

Proxy-Beispiel

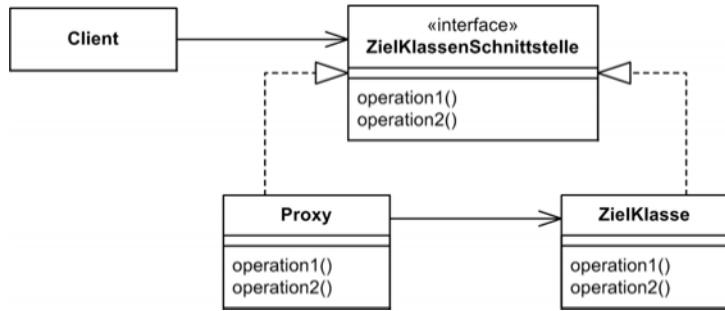


Abb. 6.3-6: Ein Beispiel eines Lösungsschemas für einen Proxy.

Hier sieht man einen Proxy und eine Zielklasse. Beide implementieren die gleiche Schnittstelle. Der Proxy delegiert Aufrufe an die Zielklasse über eine Assoziationsbeziehung. Somit kann der Proxy als Platzhalter für Instanzen der Zielklasse fungieren. Der Client hat somit nur eine Referenz auf die Schnittstelle, weil er nicht wissen soll, ob er es mit dem Proxy als Platzhalter oder einer Instanz der Zielklasse zu tun hat.

Ein Proxy ist ein typisches Entwurfsmuster. Proxies können aber auch wichtiger Teil komplexerer Architekturen sein. Beispielsweise sind Proxies oft ein Teil des Musters Broker [Buschmann et al. 1996, Völter et al. 2004]. Dieses Muster soll im Folgenden als Beispiel eines Architekturmusters beschrieben werden.

Zu Beispieldiagrammen in Mustern

Ein wichtiger Hinweis zu den Beispielen in Musterbeschreibungen: Die Klassendiagramme (und anderen Beispiele) in der Musterbeschreibung sind reine Konzeptdiagramme mit Konzeptklassen, die in der Lösung eines konkreten Problems nicht eins zu eins in „echte“ Klassen umgesetzt werden müssen bzw. können. Die Beispiele zeigen also das Lösungsprinzip, aber eine konkrete Umsetzung des Proxy-Musters kann mehr als nur einige wenige Klassen haben oder sogar noch weniger Klassen als im Muster beschrieben.

Beispiel: Architekturmuster Broker

Name:	Broker
Kontext:	Ein verteiltes Objektsystem soll entworfen werden. Das heißt, Objekte sollen in einem Server-Prozess zur Verfügung gestellt werden und auf sie soll von verteilten Klienten über das Netzwerk zugegriffen werden.
Problem:	In einem verteilten System hat man es mit vielen Herausforderungen zu tun, mit denen man es in

einem lokalen System, das in einem einzigen Prozess läuft, nicht zu tun hat. Eine wichtige Herausforderung in diesem Bereich ist die Kommunikation über nicht zuverlässige Netzwerke – im Gegensatz zu lokalen Aufrufen, kann ein Netzwerk ausfallen, ohne dass Klient oder Server ausfallen. Überdies müssen heterogene Komponenten in einer kohärente Architektur gebracht werden und die verteilten Ressourcen müssen effizient genutzt werden. Wenn Entwickler verteilter Anwendungen alle diese Herausforderungen meistern müssten, ist es wahrscheinlich, dass sie ihre eigentliche Aufgabe vergessen: eine verteilte Anwendung zu entwickeln, die die Probleme der Domäne gut löst.

Kräfte: Die Kommunikation über ein Netzwerk ist komplexer als lokale Aufrufe: Verbindungen müssen aufgebaut werden, Aufrufparameter müssen über das Netzwerk verschickt werden und es müssen netzwerkspezifische Fehler, wie der Ausfall des Netzwerkes, behandelt werden.

Es soll vermieden werden, dass Aspekte verteilter Programmierung über den Code einer verteilten Anwendung verstreut sind.

Die Netzwerkadresse und andere Parameter des Servers sollen nicht in der Klientenanwendung hart-kodiert werden. Dies ist wichtig, um zu erlauben, dass ein verteilter Service durch andere Server realisiert wird, ohne dass man den Klienten ändern muss.

Lösung: Durch die Verlagerung aller Kommunikationsaufgaben in einen Broker werden die Kommunikationsaufgaben eines verteilten Systems von dessen Anwendungslogik getrennt. Der Broker verbirgt und steuert die Kommunikation zwischen den Objekten oder Komponenten des verteilten Systems. Auf der Klientenseite baut der Broker die verteilten Aufrufe zusammen und leitet sie dann an den Server weiter. Auf der Server-Seite nimmt der Broker die Anfrage entgegen und baut daraus einen Aufruf zusammen, den er dann auf einem Server-Objekt durchführt. Auf dieselbe Weise leitet der Broker die Antwort an den Klienten zurück. Der Broker übernimmt alle Details der verteilten Kommunikation, wie Verbindungsaubau, Marshalling der Nachricht (das heißt Umwandlung der zu versendenden Daten in das Nachrichtenformat) etc., und verbirgt diese Details – so weit wie möglich – vor dem Klienten und dem verteilten Objekt.

Konsequenzen: Ein Broker hat den Vorteil, dass er verteilte Kommunikation abstrahiert und vereinfacht. Die Broker-Infrastruktur kann von verschiedenen verteilten Anwendungen wiederverwendet werden. Da der Broker dafür verantwortlich ist, den Server bzw. das verteilte Objekt über einen symbolischen Namen oder eine ID aufzufinden, erlaubt der Broker die Transparenz des Ortes, an dem sich das verteilte Objekt wirklich im Netzwerk befindet.

Eine Broker-Architektur ist typischerweise leicht weniger performant und verbraucht mehr Ressourcen als eine gut entworfene verteilte Architektur, in der statische, verteilte Objekte direkt ans Netz gebunden werden. Ein Broker hat eine gewisse Komplexität, die verstanden werden muss. Für sehr einfache Anwendungen, beispielsweise im Bereich von eingebetteten Systemen, mögen einfachere Architekturen denselben Nutzen bringen wie eine Broker-Architektur, sind aber leichter zu warten und zu verstehen. Für die meisten anderen verteilten Systeme, wie z. B. im Enterprise-Bereich, ist eher der Einsatz eines Brokers zu empfehlen.

Broker: Beispiel-Architektur

Die folgende Abbildung 6.3-7 gibt einen Grobüberblick über die Broker-Architektur (für weitere Details siehe [Völter et al. 2004]). Man sieht, wie ein Klient „virtuell“ mit einem verteilten Objekt kommuniziert, aber anstatt das Objekt direkt zu adressieren, richtet der Klient die Anfrage an den Broker.

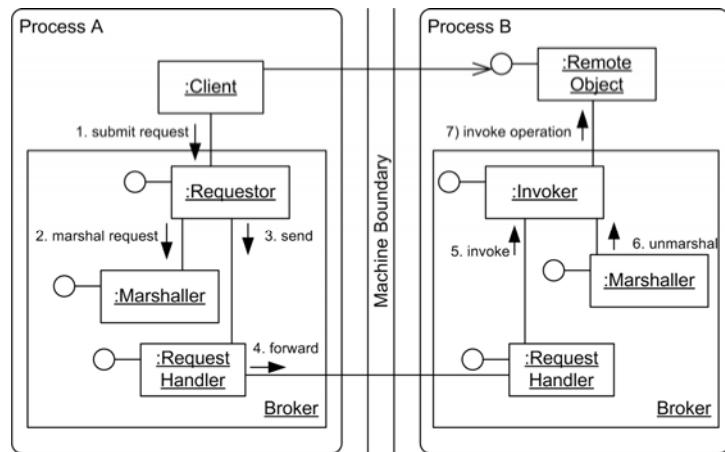


Abb. 6.3-7: Beispiel-Architektur eines Brokers für verteilte Objekte.

Dieser enthält einen Requestor, der die Anfrage mittels eines Marshallers in eine über das Netzwerk übertragbare Form bringt. Auf der Server-Seite wird die Nachricht wieder mittels eines Marshallers in einen Aufruf übersetzt und ein Invoker ruft damit das verteilte Objekt auf. Was man hier schon sieht, ist, dass das Broker-Muster sich aus vielen anderen Mustern zusammensetzt, die die verschiedenen Einzelaufgaben des Brokers lösen.

Das oben beschriebene Proxy-Muster findet auch oft im Broker Einsatz: Damit der Client auf die Schnittstelle des verteilten Objektes zugreifen kann, benötigt er einen lokalen Platzhalter, der dieselbe Schnittstelle implementiert und den Aufruf an den Requestor weitergibt. Dies übernimmt der Client Proxy [Völter et al. 2004], eine Variante des Proxy-Musters für verteilte Objektsysteme.

Proxy-Einsatz im Broker

6.3.5 Mustersprachen

Ein einzelnes Muster beschreibt eine Lösung zu einem einzelnen wiederkehrenden Problem. In der Mehrzahl der Anwendungsfälle ist die Situation jedoch komplexer. Dies gilt sowohl in der klassischen Architektur wie auch in der Software-Architektur und anderen Anwendungsfeldern von Mustern. Typischerweise gibt es bei der Konzeption einer Software-Architektur mehrere Entwurfsprobleme, die häufig zusammen auftreten und starke Beziehungen zueinander haben. Beispielsweise kann ein Muster Teil der Lösung eines anderen Musters sein oder der Kontext eines Musters ist eine Situation, in der ein anderes Muster angewendet wurde. Daher werden Muster oft nicht isoliert voneinander beschrieben, sondern gemeinsam. Folgende Arten, zusammenhängende Muster zu beschreiben, können unterschieden werden:

- > *Verwandte Muster*: In der einfachsten Form kommen solche Beziehungen in einzelnen Mustern vor, wenn diese Beziehungen zu einem oder mehreren anderen Muster haben. Z. B. kann ein Muster beschreiben, welche anderen Muster Alternativen sind oder welche anderen Muster oft im gleichen Zusammenhang zur Anwendung kommen.
- > *Zusammengesetzte Muster*: Eine stärkere Art der Musterbeziehung sind zusammengesetzte Muster. Diese bestehen aus einem oder mehreren anderen Mustern und fügen zu diesen Einzellösungen selbst noch ein Inkrement hinzu. Ein Beispiel für ein zusammengesetztes Muster ist das oben beschriebene Broker-Muster, das sich aus einigen Einzelmustern im Bereich des Entwurfs verteilter Objektsysteme zusammensetzt.

Komplexe Musterbeziehungen

- > *Mustersysteme*: Einige Autoren beschreiben Mustersysteme, die z. B. in der gleichen Domäne Anwendung finden oder die ein anderes Ordnungskriterium erfüllen. Ein Beispiel ist das POSA-Buch [Buschmann et al. 1996], in dem ein System von Mustern im Bereich der Software-Architektur beschrieben wird.
- > *Mustersprachen*: Alexander [Alexander 1977] strebt in seiner ursprünglichen Musterdefinition tiefere Beziehungen zwischen den Mustern als einfache Mustersysteme an. Diese tieferen Musterbeziehungen werden in Mustersprachen (englisch: *pattern languages*) beschrieben, welche in dem restlichen Teil dieses Abschnittes genauer betrachtet werden sollen.

Mustersprachen

Eine Mustersprache ist eine Sammlung von semantisch zusammengehörenden Mustern, welche Lösungsszenarien für Probleme in einem bestimmten Kontext bietet. Mustersprachen sind insbesondere auf die Beziehungen der Muster in der Sprache fokussiert. Das heißt konkret, dass die Musterbeschreibungen stark integriert sind – z. B. dadurch, dass der Kontext des einen Musters ein anderes Muster aufgreift und dass besonderer Wert auf die Beschreibung der Musterinteraktionen gelegt wird.

Alexander [Alexander 1977] fordert überdies eine „generative“ Natur einer Mustersprache: Die Idee ist, dass, wenn man ein Muster anwendet, „automatisch“ ein neuer Kontext entsteht, in dem andere Muster der Mustersprache angewendet werden können. So entsteht schrittweise eine bessere Architektur. Jeder inkrementelle Schritt der Weiterentwicklung führt zur Verbesserung der Qualität der Gesamtarchitektur. Ein Beispiel für die „generative“ Natur einer Mustersprache wird weiter unten gegeben.

Mustersequenzen

Die domänen spezifische Natur einer Mustersprache ist sehr wichtig für deren Anwendung. Bei Sammlungen einzelner Muster ist es oft sehr schwierig zu bestimmen, wann welches Muster eingesetzt werden soll. Diese Abwägung erleichtert eine Mustersprache, da sie die Muster in einer kohärenten Form präsentiert. Nach jeder Musteranwendung ist unmittelbar aus der Musterbeschreibung heraus klar, welche anderen Muster als Nächstes zur Anwendung kommen können, welche Muster Alternativen zu einem gegebenen Muster bilden etc. Diese Beziehungen werden auch als *Mustersequenzen* bezeichnet. Die Hauptidee hinter dieser Beschreibungsform ist, dass die Anzahl der möglichen Kombinationen von Mustern in einer Mustersprache riesig ist, aber die Anzahl der Kombinationen, die funktionieren, eher gering ist.

Beispiel einer Mustersprache

In der folgenden Abbildung 6.3-8 sieht man als Beispiel einen Ausschnitt aus einer Mustersprache, die im Bereich der Domäne „Entwicklung verteilter Objektsysteme“ operiert [Völter et al. 2004]. Das heißt, diese Mustersprache beschreibt in erster Linie den Aufbau von OO-RPC-Middleware, wie CORBA, Web Service Frameworks, .NET Remoting, Java RMI und vielen anderen. Die Abbildung zeigt einen Ausschnitt aus dieser Mustersprache, der sich mit den grundlegenden Mustern für die Realisierung einer Broker-Architektur beschäftigt.

An diesem Beispiel kann man auch die „generative“ Natur der Mustersprache näher erläutern. Wenn man einen Architektur-Entwurf beispielsweise mit der Bereitstellung eines Remote Objects startet, so entsteht dadurch ein Kontext, in dem man einen Invoker benötigt, um Remote Objects aufzurufen. Der Invoker muss vom Klienten aus adressiert werden und Nachrichten empfangen, die beide Kommunikationspartner verstehen. Dies ist der Kontext, in dem man Marshaller und Requestor anwenden kann. Auf der anderen Seite müssen sowohl Klient, als auch Server, an das Netzwerk gebunden werden und Betriebssystemressourcen müssen effizient genutzt werden, was zum Einsatz von Client und Server Request Handler führt. Oft soll der Klient sich auf die Schnittstelle der Remote Objects beziehen können. Diese wird durch Interface Description beschrieben und mit einem Client Proxy klientenseitig unterstützt. Zu guter Letzt müssen zwischen Klient und Server die verteilten Fehler behandelt und weitergemeldet werden, was zum Einsatz des Remoting Error Musters führt.

Man sieht an diesem Beispiel, die Muster in dieser Mustersprache hängen eng zusammen und ihre schrittweise Anwendung führt zu der Realisierung einer komplexen Broker-Architektur.

Es gibt also eine ganze Reihe von Mustern, aus denen sich die Basisarchitektur eines typischen Brokers zusammensetzt. Zunächst stellt der Client Proxy, wie bereits im Beispiel oben erwähnt, die Schnittstelle des Remote Objects zur Verfügung. Dieser Client Proxy kann von den Server-Entwicklern an die Klienten ausgeliefert werden.

Als Alternative, die häufig im Bereich verteilter Systeme realisiert ist, kann der Client Proxy aber auch auf der Klientenseite erzeugt werden. Um diese Alternative zu realisieren, muss der Klient wissen, wie die Schnittstelle des Remote Objects ist – sonst ist ein typkorrektes Erzeugen des Client Proxies nicht möglich. Dies leistet das Muster Interface Description, welches die öffentliche Schnittstelle eines Remote Objects beschreibt.

Beschreibung des Mustersprachenbeispiels

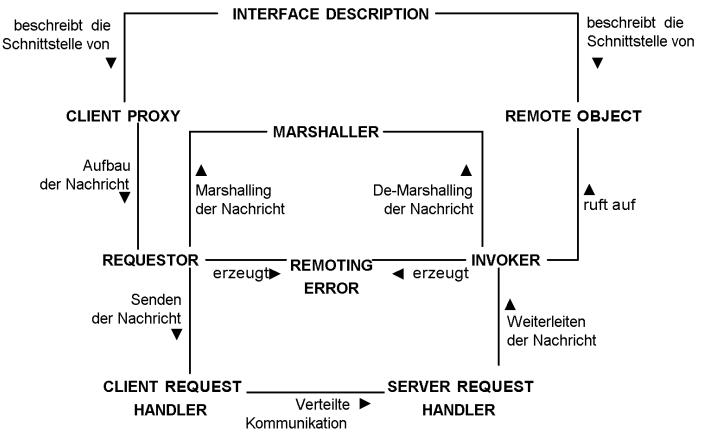


Abb. 6.3-8: Übersicht eines Ausschnitts einer Mustersprache für verteilte Objektsysteme.

Der Requestor ist dafür verantwortlich, die Nachricht aufzubauen und das Senden der Nachricht anzustoßen. Er muss auch für den Klienten auf das Resultat warten. Der Klient kann den Requestor auch direkt verwenden, verliert aber dann die Aufrufrücksicht, die das Client-Proxy-Muster ihm bietet. Falls Aufrufrücksicht gewünscht wird, sollte also ein Client Proxy eingesetzt werden. Dann benutzt der Client Proxy den Requestor intern. Das effiziente Übertragen der Nachricht über das Netzwerk übernimmt der Client Request Handler.

Auf der Server-Seite nimmt der Server Request Handler die Nachricht entgegen: Das heißt, er horcht am Netzwerk-Port und wartet auf eingehende Nachrichten. Wenn Nachrichten hereinkommen, leitet der Server Request Handler die Nachricht an einen Invoker weiter, der aus der Nachricht wieder einen Aufruf macht und damit das Remote Object aufruft.

Requester und Invoker benutzen einen Marshaller für das automatische Marshallen und Unmarshallen der Nachricht.

Im Netzwerk und während des verteilten Aufrufs können Fehler entstehen, die in lokalen Aufrufen nicht entstehen können, wie der Ausfall des Netzwerks. Diese speziellen Fehler werden durch Remoting Errors weitergeleitet, die überall in der Aufrufkette erzeugt werden können.

Zur Rolle der Muster in Mustersprachen

Dieser kurze Abriss sollte ausreichen, um einen Ausschnitt aus der architektonischen Mustersprache für verteilte Objektsysteme darzustellen.

len. In der Tat beschreibt die Mustersprache noch eine ganze Menge an weiteren Mustern, die in jeder Broker-Architektur vorkommen.

Was aus dem kurzen Beispiel klar geworden sein sollte, ist, dass die Muster in einer Mustersprache normalerweise eher *Rollen* als Bausteine eines Systems beschreiben. Bei einigen der oben genannten Muster wird es sicher in einigen Systemen gerade einen Baustein geben, der dieses Muster implementiert. Oft sind die Muster aber auch über mehrere Klassen hin verteilt implementiert oder eine Implementierungs-klaasse implementiert mehrere Muster.

Trotzdem ist es ein Leichtes, die oben genannten Muster in so gut wie jeder OO-RPC-Middleware, wie CORBA, Web Service Frameworks, .NET Remoting, Java RMI etc. wieder zu finden. Auch kann man schnell die Sequenzen und Alternativen erkennen, das heißt, wie diese Muster zusammenhängen.

Dies alles sind entscheidende Vorteile der Anwendung von Mustersprachen gegenüber der Nutzung einzelner, isolierter Muster. Aus diesem Grund setzen sich Mustersprachen in der Architektur-Muster-Literatur zunehmend durch.

6.3.6 Zusammenfassung

- > Taktiken benötigen eine Analyse der Qualitätsattribute als Voraussetzung. Ein geeignetes Mittel dazu sind Qualitätsattributszenarien.
- > Man kann Qualitätsattributszenarien nach Qualitätsattributen in Szenariotypen einteilen. Beispiele sind: Verfügbarkeitsszenarien, Änderbarkeitsszenarien, Performanzszenerien etc.
- > Eine Taktik ist eine Entwurfsentscheidung, die Einfluss nimmt auf die Realisierung der Reaktion eines Qualitätsattributszenarios.
- > Ein Architektur-Stil ist ein Muster der strukturellen Organisation einer Familie von Systemen.
- > Architektur-Stile und Architektur-Muster sind sehr ähnliche Konzepte.
- > Im Laufe der letzten Jahre sind Software-Muster zu einem wichtigen Instrument des Software-Entwicklers und -Architekten geworden.
- > Ein Muster ist eine dreiteilige Regel, die die Beziehung zwischen einem bestimmten Kontext, einem bestimmten System an Kräften, die in diesem Kontext wiederkehrend auftreten, und einer bestimmten Software-Konfiguration, die diesen Kräften erlaubt, sich gegenseitig aufzulösen, ausdrückt.

**Zusammenfassung:
Taktiken, Stile und
Muster**

- > Muster drücken auch die Überlegungen hinter einer Entwurfsentscheidung sowie ihre Konsequenzen aus.
- > Im Kontext der Software-Architektur spielen sowohl Entwurfsmuster als auch Architektur-Muster eine große Rolle.
- > Eine Mustersprache ist eine Ansammlung von Mustern, welche die Probleme in einer bestimmten Domäne und/oder in einem bestimmten Kontext löst.
- > Mustersprachen sind insbesondere auf die Beziehungen der Muster in der Sprache fokussiert.

6.4 Basisarchitekturen

Überblick

In diesem Abschnitt werden einige grundlegende Basisarchitekturen behandelt, die in vielen Systemen zum Einsatz kommen. Diese Basisarchitekturen setzen die verschiedenen Architektur-Mittel, die bereits in den vorhergegangenen Abschnitten diskutiert wurden, zur Architektur-Strukturierung ein. Sie selbst stellen also konkretere Architektur-Mittel dar, mit denen man Systeme ganzheitlich strukturieren kann. Zunächst sollen einfache Basisarchitekturen diskutiert werden, die auf einzelnen Mustern und Stilen basieren, wie beispielsweise Schichtenarchitekturen. Mehr Details zu den grundlegenden Mustern und Stilen, die in diesem Abschnitt behandelt werden, kann man in [Avgeriou and Zdun 2005] finden. Anschließend werden „größere“ Basisarchitekturen diskutiert, wie beispielsweise serviceorientierte Architekturen oder Sicherheitsarchitekturen.

Monolith

Die oftmals „schlimmste“ Form der Architektur-Strukturierung ist ein *Monolith*. In einem Monolithen ist die komplette Architektur, die während des Entwurfsprozesses spezifiziert wurde, in einem einzelnen Systembaustein zusammengefasst. Eine solche Architektur kann nur in seltenen Fällen die Architektur-Prinzipien aus Abschnitt 6.1 gut umsetzen. Da zum Beispiel in einer monolithischen Architektur-Struktur keine verschiedenen Bausteine separat betrachtet werden, ist Separation of Concerns nicht zu erreichen. Ebenso ergeht es der losen Kopplung: Bei einem Monolithen sind keine Bausteine gekoppelt, also kann es auch zu keiner geeigneten Umsetzung der losen Kopplung kommen. Im Folgenden werden einige prototypische Lösungen erklärt, in denen eine andere Architektur-Struktur als der Monolith zum Einsatz kommt. Für geeignete Anwendungsfälle setzen die im Folgenden behandelten Basisarchitekturen die Architektur-Prinzipien besser als ein Monolith um.

Ein typisches Beispiel für Monolithen sind Altsysteme, die oft über Jahrzehnte gewachsen sind. In ihnen ist die Geschäftslogik oft hartcodiert und schwer aufzufinden, weil sie über den Code verstreut und nicht dokumentiert ist. Typische Symptome für Altsysteme als Monolithen sind, dass

- > die Anpassung der Geschäftslogik an neue Anforderungen schwierig ist,
- > Tests neuer Releases aufwendig und langwierig sind,
- > kleine Änderungen sehr aufwendig sind und die Stabilität des Gesamtsystems gefährden,
- > Mitarbeiter ihre Arbeitsabläufe dem Verhalten der Software anpassen und nicht umgekehrt,
- > nur wenige altgediente Mitarbeiter sich mit der Software wirklich auskennen, weil keine ausreichende Dokumentation vorhanden ist.

Natürlich kann die folgende, kurze Abhandlung über Basisarchitekturen nur einen Überblick über wichtige Vertreter in der heutigen Praxis – ohne Anspruch auf Vollständigkeit – geben. Viele andere geeignete Basisarchitekturen existieren. Abbildung 6.4-1 gibt einen Überblick der behandelten Basisarchitekturen.

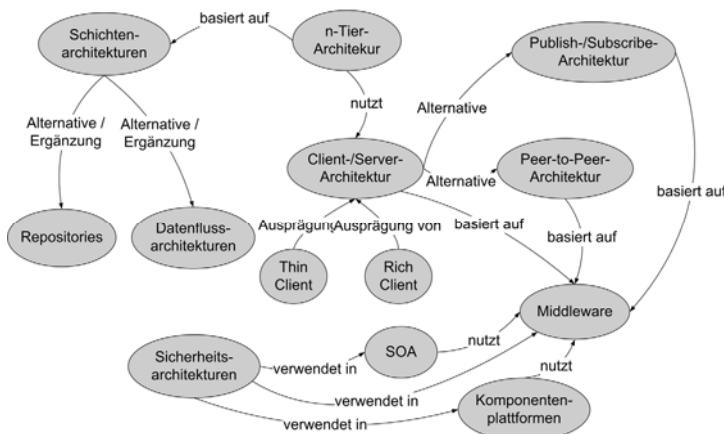


Abb. 6.4-1: Überblick Basisarchitekturen.

6.4.1 Schichtenarchitekturen

Eine grundlegende Fragestellung der Strukturierung von Architekturen, die sowohl die funktionalen, als auch die nicht-funktionalen Anforderungen betrifft, ist, wie man ein System in Gruppen von ähnlichen

Beispiel: Altsysteme als Monolithen

Überblick Basisarchitekturen

Layers-Muster

Funktionen bzw. Verantwortlichkeiten unterteilt. Eine typische Lösung wird durch das Layers-Muster [Buschmann et al. 1996] bzw. den Layers-Stil [Shaw und Garlan 1996] beschrieben.

Layers wird in Situationen angewendet, wo eine Gruppe von Bausteinen von einer anderen Gruppe von Bausteinen abhängt, um ihre Funktion erbringen zu können, die wiederum von einer Gruppe von Bausteinen abhängt etc. Layers besagt, dass jede Schicht eine Anzahl von Bausteinen gruppieren und der darüber liegenden Schicht Dienste durch Schnittstellen bereitstellt. In jeder Schicht können die Bausteine untereinander frei interagieren. Zwischen zwei Schichten kann nur durch die vorgegebenen Schnittstellen kommuniziert werden. Generell sollten Schichten nicht überbrückt werden. Das heißt Schicht „n“ ruft nur die Dienste von Schicht „n-1“ auf, aber nicht die von „n-2“.

Die Hauptziele der Schichtenbildung sind die Veränderbarkeit des Systems zu erhöhen, das System portierbar zu gestalten und/oder die Wiederverwendbarkeit der Schichten zu erhöhen.

Abbildung 6.4-2 zeigt ein schematisches Beispiel einer Schichtenarchitektur.

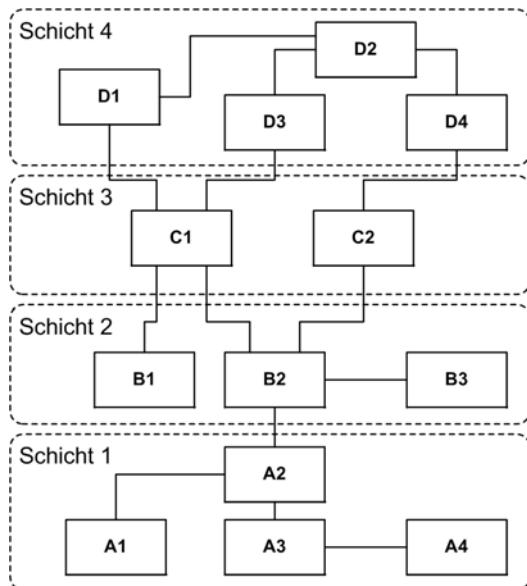


Abb. 6.4-2: Beispiel einer Schichtenarchitektur [Avgeriou and Zdun 2005].

6.4.2 Datenflussarchitekturen

Eine weitere grundlegende Art der Strukturierung ist die Strukturierung entlang der Datenflüsse einer Architektur. Solche Architekturen machen insbesondere dann Sinn, wenn eine komplexe Aufgabe in eine Reihe einfacher Aufgaben zerteilt und dann als Kombination unabhängiger Aufrufe dargestellt werden kann. Dies kann mittels eines monolithischen Bausteins nicht gut erreicht werden, weil dieser schnell überkomplex würde und somit Änderbarkeit und Wiederverwendbarkeit leiden würden.

Eine Lösung ist der Batch-Sequential-Stil [Shaw und Garlan 1996], in welchem eine Gesamtaufgabe in Teilschritte zerlegt wird, die dann als separate und unabhängige Bausteine implementiert werden. Jeder Teilschritt wird bis zur Vollendung durchlaufen und ruft dann den nächsten Schritt in einer Sequenz von Schritten auf. In jedem Schritt werden die Daten zu Berechnungen herangezogen und die Ergebnisdaten werden als Ganzes an den nächsten Schritt weiter gegeben.

Batch Sequential ist die Grundlage für einfache Datenflussarchitekturen. Manchmal kommt als zusätzliche Anforderung hinzu, dass Ströme von Daten verarbeitet werden müssen. Verschiedene Klienten benötigen zudem verschiedene Kombinationen der Verarbeitungsschritte.

Dann macht es Sinn, das Pipes-and-Filters-Muster [Buschmann et al. 1996] bzw. den Pipes-and-Filters-Stil [Shaw und Garlan 1996] einzusetzen. Auch hier wird eine Gesamtaufgabe in sequentielle Teilschritte zerlegt, die in separaten Bausteinen (den Filtern) implementiert werden. Filter haben eine Anzahl an Eingabe- und Ausgabeschmittstellen durch die sie flexibel mittels sogenannter Pipes kombiniert werden können. Jede Pipe realisiert einen Datenstrom zwischen zwei Filtern. Filter konsumieren und liefern Daten inkrementell. Das heißt, verschiedene Filter können potentiell parallel an Aufgaben arbeiten. Pipes agieren als Datenpuffer zwischen Filtern. Mehr Details zu Pipes and Filters findet man in Abschnitt 6.3.

Ein Beispiel für eine Pipes-and-Filters-Architektur sind Java-Servlet-Filter. Sie erlauben es, auf die Anfragen und Antworten bei dem Zugriff auf eine Web-Ressource zuzugreifen. Dabei können mehrere Filter in einer Kette flexibel jeweils für eine bestimmte Web-Ressource konfiguriert werden. Beispiele für die Anwendung von Servlet-Filtern sind das Logging, die Verschlüsselung und Entschlüsselung, die Komprimierung und Dekomprimierung, sowie die Transformation (z. B. von XML mittels XSLT).

Batch-Sequential-Stil

Pipes-and-Filters-Muster/-Stil

Beispiel: Servlet-Filter

6.4.3 Repositories

Shared-Repository-Stil

Die Hauptaufgabe von Repositories ist, verschiedenen Bausteinen den gleichzeitigen Zugriff auf Daten zu ermöglichen. Der Shared-Repository-Stil [Shaw und Garlan 1996] stellt hier eine grundlegende Strukturierung dar. Diese wird auch in dem Repository-Muster [Evans 2004] beschrieben.

In einer Shared-Repository-Architektur stellt ein Baustein des Systems einen zentralen Datenspeicher zur Verfügung. Das Shared Repository stellt Mittel zur Verfügung, um auf die Daten zuzugreifen und zu ändern, z. B. ein API oder eine Query-Sprache. Das Shared Repository muss den effizienten Zugriff auf Daten für Klienten sichern, skalierbar sein und die Konsistenz der Daten gewährleisten. Beispielsweise kann das Repository geeignete Locking-Funktionen oder Transaktionsmechanismen anbieten, um die gleichzeitige Veränderung der Daten durch zwei Klienten auszuschließen. Es können auch weitere Funktionen, wie Sicherheitsfunktionen angeboten werden.

Abbildung 6.4-3 zeigt ein schematisches Beispiel einer Repository-Architektur.

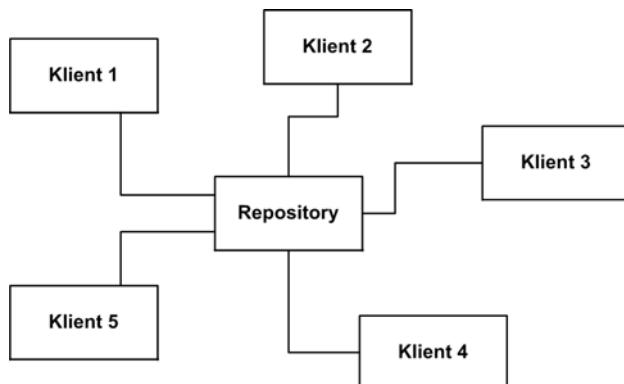


Abb. 6.4-3: Beispiel einer Repository-Architektur [Avgeriou and Zdun 2005].

Beispiel: EAI-Metadaten-Repository

Ein einfaches Beispiel für eine Repository-Architektur ist ein Metadaten-Repository für eine Enterprise-Application-Integration-(EAI)-Lösung. EAI-Lösungen benötigen oft Transformationen von Daten, die mit Transformationswerkzeugen durchgeführt werden (dies wird auch als Mapping bezeichnet). Die verschiedenen verteilten Bausteine der EAI-

Architektur benötigen einheitliche und konsistente Informationen über die Datenquellen und Datenziele einer Transformation, die Transformationsregeln und die Datenflussregeln. Überdies müssen konkurrente Änderungen an diesen Informationen und Regeln ermöglicht werden. Dies kann durch ein zentrales Metadaten-Repository gelöst werden, auf das von allen Bausteinen der EAI-Architektur zugegriffen werden kann und Dienste, wie Locking bei konkurrenten Änderungen, bereitstellt.

6.4.4 Zentralisierung gegenüber Dezentralisierung

Eine grundlegende Frage, wenn man sich gegen eine monolithische Architektur entscheidet, ist die der Zentralisierung gegenüber der Dezentralisierung. Auf vielen Ebenen des Architektur-Entwurfs muss man sich fragen: Ist es besser, einen Belang in einem Systembaustein zu bündeln (Zentralisierung) oder ihn auf mehrere Systembausteine zu verteilen (Dezentralisierung)? Der vorher angesprochene Monolith stellt eine extreme Form der Zentralisierung dar: Alles ist in einem einzigen Systembaustein gebündelt. Meist muss man allerdings eher eine Abwägung zwischen der Zentralisierung und der Dezentralisierung treffen und kommt so zu einem Kompromiss.

Die anderen zuvor diskutierten Architektur-Strukturierungen stellen solche Kompromisse dar: Schichten- und Datenflussarchitekturen stellen zwei spezielle Zerteilungen einer Aufgabenstellung – also eine dezentrale Behandlung dieser Aufgabenstellung – dar. Repositories bilden eine Zentralisierung der Datenhaltung und eine Dezentralisierung der Klienten der Daten ab.

Es sei darauf hingewiesen, dass Dezentralisierung im Sinne der Architektur meist bedeutet, dass man es mit einem verteilten System zu tun hat. Das heißt, die Systembausteine werden z. B. auf verschiedene Rechner, Prozessoren oder Prozesse verteilt.

Ein wichtiger Einflussfaktor bei der Abwägung zwischen Zentralisierung und Dezentralisierung ist der Anwendungsfall. Wird z. B. darüber nachgedacht, ein unternehmensweites Anwendungssystem zu zentralisieren, sind in der Regel mehr Teilgebiete, Systeme und Interessenvertreter in die Entscheidung einzubeziehen, als wenn sich die Zentralisierung oder Dezentralisierung z. B. nur auf ein Projekt oder eine Abteilung bezieht.

Kompromiss Zentralisierung versus Dezentralisierung

Anwendungsfallbezug

Abwägungskriterien

Allgemein kann man einige Vorteile für Dezentralisierung wie auch für Zentralisierung benennen, die für die meisten Architekturen gelten; also beispielsweise sowohl für die Hardware- als auch die Software-Architektur.

Vorteile der Dezentralisierung

Ein zentraler Vorteil von Dezentralisierung sind die oft niedrigeren Hardwarekosten. Wenn man auf viele Rechner dezentralisiert, dann kann man meist auf günstigere Hardware zugreifen, als wenn man die Software-Bausteine auf einem leistungsfähigen und deshalb teuren Rechner bündelt (aber auch das Gegenteil kann der Fall sein). Dezentrale Architekturen sind auch flexibler gegenüber Veränderungen und können leichter mit Ausfällen einzelner Bausteine umgehen, da meist dieselben Bausteine redundant vorhanden sind. Es ist auch zum Teil einfacher, die Architektur aufgabenorientiert zu strukturieren, da die Systemlandschaft strukturell den Verantwortlichkeiten im Unternehmen (oder der Systemumgebung) folgend modelliert werden kann.

Vorteile der Zentralisierung

Zentralisierung hingegen hat ihre Vorteile, wo zentrale Aufgaben im Vordergrund stehen. Beispielsweise ist es deutlich einfacher, in einem zentralen System hohe Daten- und IT-Sicherheit, Logging, Kontrolle, Monitoring, einfache Auslieferung neuer Software-Bausteine etc. sicherzustellen. Als Resultat ergeben sich Vorteile im Bereich der Kosten für den personellen Betreuungsaufwand. Daher sind typische zentrale Aufgaben die Verwaltung großer Datenbestände, die Netzwerksteuerung, die Steuerung der Transaktionsverarbeitung, die laufende Prüfung der Hard- und Software im gesamten Netz und die Auslieferung und Bereitstellung von Software-Bausteinen. Weitere Vorteile der Zentralisierung sind in manchen Fällen niedrigere Hardware-Kosten (z. B. wenn Anwendungen auf Hardware gepoolt werden können) und niedrigere Kosten für den Energieverbrauch der Hardware.

Mainframe-Architektur

In den Anfangszeiten der Computer waren diese sehr teuer und sehr groß. Es war nicht möglich, jedem Mitarbeiter einen Computer zur Verfügung zu stellen. Stattdessen stellte man einen teuren und leistungsfähigen Großrechner (englisch: *mainframe*) bereit. Dieser stellte gemeinsame Anwendungen zur Verfügung und auf ihn wurde über sogenannte Terminals zugegriffen. Die Terminals wurden über eine serielle Leitung mit dem Mainframe verbunden. Sie waren also Eingabe-Ausgabe-Schnittstelle zwischen Benutzern und dem Mainframe. Auch heute noch existieren Mainframes, aber moderne Formen des Terminals sind mit Arbeitsspeicher, Prozessor und Schnittstellen ausgestattet.

Mit dem Aufkommen kostengünstiger Personal Computer (PCs) fiel der zentrale Vorteil des Mainframes, die hohen Kosten eines Arbeitsplatzrechners, weg. Daher kam es zu einer stärkeren Dezentralisierung. Die ersten PC-Rechnernetze basierten auf File Sharing. Der Server stellt Dateien in einem gemeinsamen Speicher zur Verfügung, auf die von Arbeitsplatzrechnern zugegriffen werden kann. File Sharing hat jedoch seine Grenzen. Es funktioniert nur dann gut, wenn die gemeinsame Nutzung gering ist und es nicht oft zu gleichzeitigen Zugriffen kommt. Da die gesamten Daten einer Datei übertragen werden, ist auch die Netzbelastung beim File Sharing relativ hoch.

File Sharing

In neuerer Zeit hat sich daher das Client-/Server-Modell durchgesetzt, welches auch als Architekturstil dokumentiert wurde (siehe [Shaw und Garlan 1996]). Hier betreibt der Anwender auf seinem Rechner Anwendungsprogramme (Klienten, englisch: *clients*), welche auf die Ressourcen des Servers zugreifen. Die Ressourcen werden zentral verwaltet, aufgeteilt und zur Verfügung gestellt. Wie man in Abbildung 6.4-4 sieht, basiert das Client-/Server-Modell auf einem einfachen Anfrage-Antwort-Schema. Dadurch müssen Dateien nicht mehr als Ganzes übertragen werden, sondern Anfragen können gezielt gestellt werden. Z. B. greift der Server oft auf eine (relationale) Datenbank zu und führt eine Abfrage durch, die in der Anfrage des Klienten spezifiziert wurde.

Client-/Server-Modell

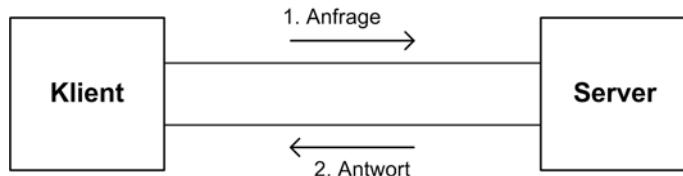


Abb. 6.4-4: Client-/Server-Modell.

Es sei darauf hingewiesen, dass ähnliche Entscheidungen für Zentralisierung gegenüber Dezentralisierung auch in „größerem“ oder „kleinem“ Kontext getroffen werden müssen. Z. B. wenn die Systembausteine nicht einzelne Rechner sind, sondern ganze Netzwerke, kommt es zu ähnlichen Abwägungen und meist zu einer Mischform aus zentralen und dezentralen Elementen. Abbildung 6.4-5 zeigt beispielhaft ein zentrales Rechenzentrum (mit einigen zentralen Servern Z1...Z3), das mit mehreren dezentralen Niederlassungen verbunden ist und diesen zentralisierte Server-Dienste anbietet. Jede Niederlassung selbst hat einen oder mehrere zentrale Server. Die einzelnen Rechnerverbindungen realisieren wiederum das Client-/Server-Modell.

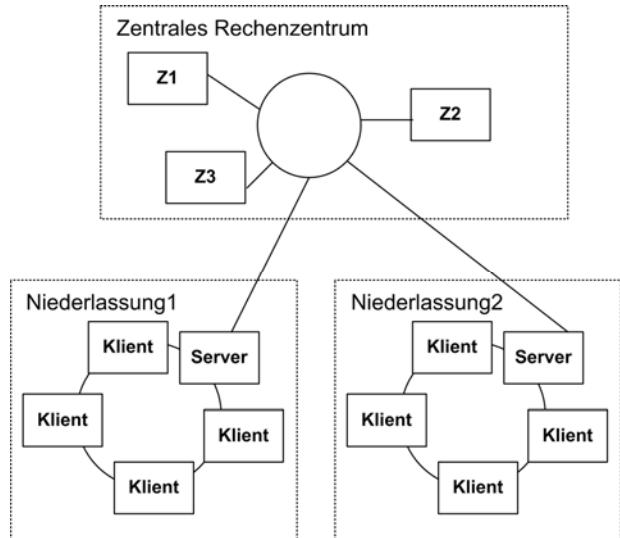


Abb. 6.4-5: Beispiel eines Rechnernetzes.

6.4.5 n-Tier-Architektur

2-Tier-Architektur

Das klassische Client/Server-Modell basiert auf einer sogenannten 2-Tier-Architektur: Die Benutzerschnittstelle ist für gewöhnlich auf dem PC des Anwenders angesiedelt und das Datenbankmanagement ist auf einem leistungsfähigerem Rechner, der mehrere Klienten bedient, angeordnet. Die Berechnungen und die Ablaufsteuerung sind somit aufgeteilt zwischen dem Klienten und dem Server. Der Datenbank-Server stellt z. B. Stored Procedures und Datenbank-Trigger zur Verfügung, um auf der Datenbank Berechnungen durchzuführen. Natürlich gibt es auch 2-Tier-Architekturen (und allgemein n-Tier-Architekturen), die keine Datenbank nutzen, aber dieses typische Beispiel soll hier zur Veranschaulichung dienen.

2-Tier-Architekturen funktionieren bis zu einer bestimmten Anzahl von Klienten, die gleichzeitig auf der Datenbank arbeiten, gut. Bei sehr hohen Nutzerzahlen nimmt die Performanz jedoch rapide ab. Überdies ist eine Abhängigkeit vom Hersteller der Datenbank gegeben: Da die Datenbankprozeduren proprietär und vom Hersteller abhängig sind, ist ein Wechsel der Datenbank gewöhnlich mit erheblichem Aufwand verbunden.

3-Tier-Architekturen lösen diese Probleme, indem sie eine Zwischenschicht zwischen Client und den Datenbankserver einführen. Die Zwischenschicht übernimmt Aufgaben, wie das Queuing (Aufreihen) von Anfragen, Durchsetzen von Ablaufplänen für Anfragen (englisch: *scheduling*), Behandeln von Anfragen gemäß von Prioritäten und so weiter. Überdies werden zentrale Aufgaben der Anwendungslogik hier implementiert. Es gibt eine Reihe von Standardlösungen, die diese Schicht realisieren, wie Transaktionsmonitore, Messaging Server, Anwendungsserver und andere. Die zusätzliche Schicht führt zu einer Verbesserung der Performance bei großer Anzahl von Klienten und zu einer Steigerung der Flexibilität. Eine typische 3-Tier-Architektur ist in Abbildung 6.4-6 schematisch dargestellt.

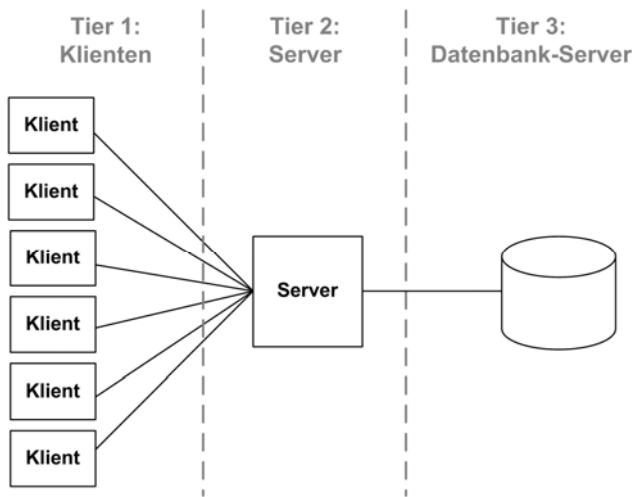


Abb. 6.4-6: 3-Tier-Architektur.

2-Tier-Architekturen und 3-Tier-Architekturen sind Spezialfälle von n-Tier-Architekturen. Mitunter kommt es zu Architekturen, die mehr Tier haben als zwei oder drei. Dies hat häufig mit hohen Lastzahlen, Sicherheitsanforderungen, Ausfallsicherheitsanforderungen oder ähnlichen Forderungen an bestimmte Qualitätsattribute der Architektur zu tun. Z. B. kann man die Server in der Zwischenschicht redundant anlegen, damit man eine höhere Performance, Ausfallsicherheit oder Verteilung der Last (englisch: *load balancing*) erreicht. Dazu muss man eine zusätzliche Instanz einfügen, die die Verteilung von Anfragen auf die redundanten Server übernimmt, also kommt es zu einer 4-Tier-Architektur. Genauso kann man auch die Datenbank auf der Datenschicht redundant gestalten. Solche Architekturen kommen insbesondere dann zum Einsatz, wenn eine verteilte Architektur besonders hohe Anforderungen an

Lastzahlen, Verfügbarkeit oder Zuverlässigkeit hat. Ein weiterer möglicher Anwendungsfall für n-Tier-Architekturen ist, dass eine Gesamtanwendung auf verschiedenen, dedizierten Hardware-Bausteinen laufen soll. Für eine weiter gehende Behandlung sei hier auf [Dyson und Longshaw 2004] verwiesen, wo solche Architekturen im Detail behandelt werden.

Auch bei der Verbindung von mehreren 3-Tier-Architekturen kommt es zu n-Tier-Architekturen mit höherer Ordnungszahl. Oft ist ein Server in einer 3-Tier-Architektur Klient eines anderen Servers in einer anderen 3-Tier-Architektur. Aus der Sicht des eigenen Klienten ist dann die Gesamtarchitektur eine 4-Tier-Architektur.

6.4.6 Rich Client gegenüber Thin Client

Rich Client und Thin Client

Eine zentrale Frage beim Entwurf einer Client-/Server-Architektur ist die Frage, wie man die Funktionalität zwischen dem Klienten und dem Server aufteilt. Diese Frage bezeichnet man allgemein als Entscheidung zwischen einem Rich Client und einem Thin Client.

Das zuvor skizzierte Mainframe-Modell stellt eine extreme Form des Thin Clients dar: Der Terminal als Klient hat so gut wie gar keine Funktionalität. Alle Berechnungen werden auf dem Mainframe durchgeführt.

Mit der Einführung der PCs und des File-Sharing-Konzepts kam es zu einer sehr starken Betonung des Rich Clients: Fast alle Berechnungen wurden beim Klienten durchgeführt.

Beim Client-/Server-Modell und bei n-Tier-Architekturen kommt es zu Kompromissen zwischen diesen beiden Extremen. Die ersten Client-/Server-Architekturen hatten gewöhnlich einen Rich Client, der speziell für die jeweilige Anwendung geschrieben wurde. Der Rich Client übernimmt einige Funktionalität, aber immer ist auch ein Teil der Funktionalität beim Server angeordnet.

Abwägung: Rich Client oder Thin Client

Es gibt eine Anzahl von Kriterien, um Rich Client und Thin Client zu vergleichen:

- > Ein Thin Client belastet die *Server-Ressourcen* generell stärker, denn es müssen mehr Berechnungen am Server durchgeführt werden.
- > Die *Netzbelastung* ist ein weiteres wichtiges Kriterium, wie auch die damit zusammenhängende *Performanz des Klienten*, denn auf jede

Netzanfrage muss man erheblich länger warten als auf eine lokale Anfrage. Daher spielen die Art und Menge der zu übertragenden Daten eine gewichtige Rolle für die Frage, welches Modell hinsichtlich Netzbelastrung und Performanz besser geeignet ist.

- › Bei Thin Clients ist man generell stärker von der *Funktionsfähigkeit des Netzwerkes* abhängig. Bei Rich Clients ist es gewöhnlich einfacher möglich, eine lokale Arbeitsumgebung zur Verfügung zu stellen. Daher sind Rich Clients z. B. für Laptops von Außendienstmitarbeitern oft besser geeignet. Allerdings besteht hier der Nachteil, dass unter Umständen die Daten repliziert und wieder synchronisiert werden müssen.
- › Zu guter Letzt ist noch die *Wartung und Auslieferung der Klienten* ein wichtiges Thema: Zentrale Software-Bausteine können am Server, der unter direktem Zugriff der Entwickler steht, ausgetauscht werden. Hingegen müssen für ein Update der Klienten unter Umständen auf eine Vielzahl von Arbeitsplatzrechnern zugegriffen werden, was einen erheblichen Mehraufwand bedeutet.

Insbesondere aufgrund des letzten Punktes kam es mit dem Erfolg des World Wide Webs dazu, dass der Web Browser in vielen Fällen als Thin Client Lösung zum Einsatz kam. Ein *Anwendungsserver* beinhaltet die komplette Anwendungslogik und die Benutzer nutzen nur einen Standardbrowser, um auf die Anwendungen zuzugreifen. Dies hat den Vorteil, dass man kaum noch Klienten warten muss. Die Nachteile dieser Lösung sind, dass man sich auf die Funktionalität des Browsers – insbesondere die eingeschränkte Benutzeroberfläche – beschränken muss und dass man jede Anfrage über das Netz verschicken muss. Das heißt, ohne Netzzugang ist die Arbeit nicht (ohne Weiteres) möglich. Eine Folge ist auch, dass die Performanz der Anwendung eher gering ist.

Web Browser als Thin Client und Anwendungsserver

Ein jüngerer Trend ist die Rückkehr zu Rich Clients, weil die auf Browsern beruhenden Thin-Client-Techniken nicht immer allen Ansprüchen genügen. Ein wichtiger Antriebsfaktor dafür sind sogenannte Rich-Client-Plattformen, wie Eclipse, die automatisches Ausliefern und Updaten von Klienten in einer Standardumgebung bieten, aber in der Benutzерumgebung einer klassischen Desktopanwendung in nichts nachstehen.

Rich-Client-Plattformen

Einen Mittelweg zwischen beiden Alternativen stellt Web 2.0 bzw. Ajax dar. Ajax bezeichnet die Möglichkeit mit Javascript asynchrone Abfragen durchzuführen und somit Inhalte nachzuladen und zu verändern, ohne dass der Nutzer die Web-Seite wechseln muss. Hier werden also

Alternative: Web 2.0 / Ajax

Teile der Aufgaben im Browser durchgeführt, aber die Skripts, die diese Aufgaben implementieren, werden vom Server ausgeliefert. Der Begriff Web 2.0 bezieht sich weniger auf spezifische Technologien als auf bestimmte Arten von Web-Anwendungen, in denen Nutzer Inhalte selbst erstellen und bearbeiten, z. B. mit Ajax oder ähnlichen Technologien. Oft sind diese Web-Anwendungen soziale Software, die es den Nutzern erlaubt, sich untereinander zu vernetzen.

6.4.7 Peer-to-Peer-Architektur

Peer-to-Peer-Stil

Client/Server ist ein Architekturstil, der eine Spezialisierung des allgemeinen Explicit-Invocation-Stils [Shaw und Garlan 1996] ist. Das heißt der Klient kommuniziert mit dem Server über direkte, explizite Aufrufe und bekommt Antworten auf diese Aufrufe vom Server. Eine Alternative stellt der Peer-to-Peer-Stil dar [Shaw und Garlan 1996], der auch Explicit Invocation spezialisiert, aber auf die direkte Kommunikation von Klienten setzt, statt mit einem zentralen Service zu kommunizieren.

P2P-Modell

Peer-to-Peer (P2P) nutzt also eine Reihe von gleichwertigen Peers (deutsch in etwa: *Teilnehmer am P2P-System*). In der reinen P2P-Architektur-Struktur gibt es keine zentralen Server. Jeder Peer kann im Netzwerk Services anbieten und konsumieren. Der Gesamtzustand des Systems ist über die Peers verteilt. Zum Teil werden P2P-Systeme zwar mit gewöhnlicher Middleware intern realisiert, aber der Nutzer eines P2P-Systems bekommt dies nicht mit.

Ein Dienst in der P2P-Architektur kann zu jeder Zeit hinzugefügt und wieder entfernt werden. Daher müssen Klienten herausfinden, welche Dienste gerade zur Verfügung stehen, bevor sie einen Dienst nutzen.

Nicht alle P2P-Systeme sind „reine“ P2P-Systeme. Die meisten Systeme sind hybrid und benutzen zentrale Server für bestimmte Dienste, z. B. als Einstiegspunkte in das Netzwerk.

6.4.8 Publish/Subscribe-Architektur

Publish/Subscribe-Stil und -Muster

Publish/Subscribe ist ein Muster [Buschmann et al. 1996] bzw. Stil [Shaw und Garlan 1996] und stellt eine Alternative zu Client/Server und Peer-to-Peer dar. Im Gegensatz zu Client/Server und Peer-to-Peer basiert Publish/Subscribe nicht auf dem Explicit-Invocation-Stil [Shaw

und Garlan 1996], sondern stellt eine Form des Implicit-Invocation-Stils [Shaw und Garlan 1996] dar. Das heißt, Aufrufe werden nicht direkt unter den Kommunikationsteilnehmern versendet, sondern Ereignisse werden durch einen Vermittler weitergeleitet.

Publish/Subscribe widmet sich dem Problem, dass eine Reihe von Klienten über Laufzeitereignisse informiert werden müssen. Diese Ereignisse haben eine andere Natur, als die direkten, expliziten Invokationen, die man in Client/Server und Peer-to-Peer vorfindet: Manchmal sollen eine Reihe von Klienten aktiv über ein Ereignis informiert werden, in anderen Fällen ist nur ein spezifischer Klient am Ereignis interessiert. Im Gegensatz zum Explicit-Invocation-Stil müssen der Produzent und Konsument eines Ereignisses voneinander entkoppelt sein, z. B. um getrennte Veränderbarkeit zu gewährleisten oder damit eine beliebige Zeit zwischen dem Auftreten und der Behandlung des Ereignisses verstreichen kann.

Publish/Subscribe ermöglicht es dem Ereigniskonsumenten, sich für bestimmte Ereignistypen zu registrieren. Wenn ein solches Ereignis stattfindet, informiert der Ereignisproduzent alle registrierten Konsumenten – z. B. mithilfe eines Publish/Subscribe-Systems. Das Publish/Subscribe-System entkoppelt also die Produzenten und Konsumenten von Ereignissen.

6.4.9 Middleware

Hat man sich einmal für eine der oben genannten Architektur-Alternativen Client/Server, Peer-to-Peer oder Publish/Subscribe entschieden, so muss man sich darauf folgend für die Art der Verbindung der Systembausteine entscheiden. Die wichtigsten Architekturstrukturen in diesem Bereich sind verschiedene middleware-basierte Architekturen. Im Folgenden soll Kommunikations-Middleware, hier kurz als „Middleware“ bezeichnet, behandelt werden. (Es gibt noch andere Formen von Middleware, aber diese werden hier nicht näher behandelt.)

Die Middleware ist eine Plattform, die Anwendungen Dienste für alle Aspekte der Verteilung anbietet, wie verteilte Aufrufe, effizienten Zugriff auf das Netzwerk, Transaktionen und viele andere. Eine generelle Einführung in das Thema „verteilte Systeme“ bietet das Buch *Distributed Systems* von Tanenbaum und van Steen [Tanenbaum und van Steen 2003]. Die folgende Darstellung orientiert sich an der Einführung in

Architektur für verteilte Systeme

dem Buch *Remoting Patterns* [Völter et al. 2004], in dem eine Mustersprache für OO-RPC-Middleware-Systeme eingeführt wird, die auf dem zuvor schon als Beispiel verwendeten Broker-Muster [Buschmann et al. 1996] aufbaut (siehe Abschnitt 6.3). Broker stellt generell als Muster die architektonische Basis für die meisten gängigen Middleware-Systeme dar.

Die Anwendungsgebiete für verteilte Systeme sind recht divers: Viele der größten und komplexesten Systeme, die heutzutage im Einsatz sind, sind verteilte Systeme. Beispiele sind Internet-Systeme, Telekommunikationsnetzwerke, Business-to-Business-Anwendungen (B2B), internationale Finanztransaktionen, eingebettete Systeme und viele andere.

Neben verteilten Problemstellungen, wie der Kollaboration von räumlich verteilten Partnern über das Netzwerk, gibt es viele andere Gründe, warum verteilte Architekturen gewählt werden. Zum Beispiel können die Performanz und Skalierbarkeit eines verteilten Systems deutlich besser sein als bei einem nicht verteilten System. Somit kann ein verteiltes System Szenarien bewältigen, in denen derartig hohe Systemlasten vorkommen, dass sie von einem einzelnen Rechner nicht mehr kosteneffektiv bewältigt werden können. Oder die Fehlertoleranz des Systems kann durch Verteilung erhöht werden: Viele Fehlertoleranzverfahren basieren auf der physikalischen Redundanz von Hardwareeinheiten, wie Rechnern oder Prozessoren – was wiederum eine verteilte Architektur nach sich zieht.

Herausforderungen in verteilten Systemen

Verglichen mit nicht verteilten Systemen muss allerdings eine Reihe von „neuen“ Herausforderungen gemeistert werden, wenn man einem System eine verteilte Architektur gibt. Wichtige Herausforderungen sind:

- > *Latenz des Netzwerkes*: Ein verteilter Aufruf benötigt deutlich mehr Zeit als ein lokaler Aufruf.
- > *Vorhersagbarkeit*: Durch die Latenz und den möglichen Ausfall des Netzwerkes ist es auch viel schwieriger, Aufrufzeiten vorherzusagen. Daher ist es in verteilten Architekturen, die Realzeitverhalten benötigen, eine wesentliche Herausforderung, dies zu garantieren.
- > *Nebenläufigkeit*: Im Gegensatz zu Ein-Prozessor-Systemen herrscht in verteilten Systemen wirkliche Nebenläufigkeit. Daher muss man sich – schon bei der Planung einer verteilten Architektur – mit resultierenden Problemen wie Nicht-Determinismus und Deadlocks beschäftigen.

- > *Skalierbarkeit*: Durch die Verteilung ist es in vielen Systemen schwerer vorherzusagen, wann wie viele Klienten auf das System zugreifen. Daher muss man sich in verteilten Systemen tendenziell mehr mit potenziellen Hochlastsituationen beschäftigen und stärker dafür Sorge tragen, dass das Gesamtsystem skaliert.
- > *Teilweiser Systemausfall*: Da in einem verteilten System mehrere Hardware-Elemente und Software-Elemente gemeinsam verwendet werden, kann es dazu kommen, dass Teile des Systems ausfallen, andere aber weiterfunktionieren. In solchen Fällen können die noch funktionierenden Systemteile versuchen, das System so zu rekonfigurieren, dass es trotz des teilweisen Systemausfalls weiter funktioniert – was allerdings meist nicht trivial ist (siehe [Tanenbaum und van Steen 2003]).

Man könnte verteilte Systeme direkt auf Basis der Netzwerk-APIs des Betriebssystems entwickeln, z. B. mit den TCP/IP-Protokollen. Aber dann müsste der Entwickler sich mit all den Herausforderungen, die oben angeführt wurden, selbst beschäftigen. Dadurch würde er aber sehr wahrscheinlich seine eigentliche Aufgabe – ein fachliches, verteiltes System zu entwickeln – schnell aus den Augen verlieren. Ferner würden gute Lösungen für die oben genannten Herausforderungen sehr wahrscheinlich nicht wiederverwendet werden.

Diesen Problemen widmet sich eine Kommunikations-Middleware. Sie hat die Aufgabe, die Kommunikationsaufgaben transparent für den Entwickler zu übernehmen und die Komplexität und Heterogenität der darunter liegenden Plattformen zu verbergen.

Abbildung 6.4-7 zeigt den Aufbau einer Middleware schematisch. Die Middleware ist eine zusätzliche Software-Schicht, die zwischen der verteilten Anwendung und den APIs des Betriebssystems sitzt. Klienten und Server-Anwendungen dürfen diese Schicht normalerweise nicht umgehen, um Services niedrigerer Schichten direkt auszuführen – somit erreicht die Middleware die Transparenz der Verteilungsaufgaben aus Anwendungssicht.

Die Middleware macht die Verteilung allerdings nur so weit wie möglich transparent. Entwickler und Architekten müssen den Verteilungsaspekt immer im Hinterkopf behalten, denn die Anwendung muss zum Beispiel Fehler, die durch einen teilweisen Systemausfall des Netzwerkes oder eines Servers verursacht werden, oft direkt behandeln. Beim Beispiel

Kommunikations-Middleware

„Ausfall einer Servers“ könnte der Client einen anderen Server kontaktieren oder einfach eine Fehlermeldung weiterreichen – was hier konkret zu tun ist, hängt von der Anwendungslogik des Klienten ab und kann daher nicht transparent von der Middleware übernommen werden. Es ist die Aufgabe der Middleware, qualifizierte Fehlermeldungen an den Klienten zurückzusenden bzw. dem Server dies zu ermöglichen.

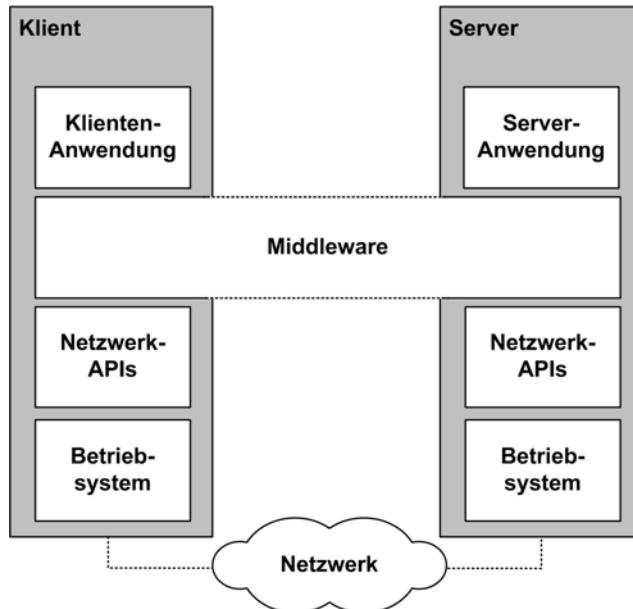


Abb. 6.4-7: Schematische Darstellung einer Middleware-Architektur.

Verteilungsstile

Es gibt einige Verteilungsstile, die in heutigen Middleware-Systemen verwendet werden. Historisch gesehen basiert die verteilte Berechnung auf einfacher *Dateiübertragung* (siehe File-Sharing-Diskussion oben). Diese ist für viele heutige Systeme mit hohen Systemlasten nicht mehr zeitgemäß, da sie zu hoher Latenz und hoher Ressourcenbelastung führt. Die meisten modernen Middleware-Systeme basieren auf einem oder mehreren der folgenden Verteilungsstilen [Völter et al. 2004]:

- > *Remote-Procedure-Call-Systeme (RPC)* nutzen die bekannte und für viele Entwickler gewohnte Prozedurabstraktion im verteilten Umfeld. Ein verteilter Prozedurauftrag kann sehr ähnlich wie ein lokaler Prozedurauftrag durchgeführt werden und wird durch die RPC-Middleware transparent vom Klienten zum Server weitergeleitet. Eine ganze Reihe von RPC-Systemen unterstützen objektorientierte Abstraktionen – sogenannte OO-RPC-Systeme. Das OO-RPC-System wird ausführlich in [Völter et al. 2004] behandelt.

- > *Messaging-Systeme* versenden sogenannte Nachrichten, welche asynchron (oder optional auch synchron) von einem Sender an ein oder mehrere Empfängersysteme versendet werden. Es gibt verschiedene Nachrichtentypen, wie Anfragen, Antworten, Fehlermeldungen etc. Diese werden in Nachrichtenwarteschlangen so lange zwischengespeichert, bis sie versendet bzw. konsumiert werden können. So können Messaging-Systeme die Auslieferung einer Nachricht garantieren, auch wenn es zeitweise zu Systemausfällen kommt. Das Thema Messaging-Systeme wird ausführlich in [Hohpe und Woolf 2003] behandelt.
- > *Gemeinsame Speicher* (englisch: *shared repository*) stellen verschiedenen Klienten einen gemeinsamen Datenraum verteilt zur Verfügung, auf den diese lesend und schreibend zugreifen können.
- > *Streaming-Systeme* erlauben – im Gegensatz zu dem diskreten Austausch von Daten in den drei zuvor genannten Stilen – den kontinuierlichen Datenaustausch mittels eines Datenstroms.

6.4.10 Komponentenplattformen

Komponentenarchitekturen wurden – als Konzept – bereits in Abschnitt 6.2.3 eingeführt. Im Folgenden sollen exemplarisch Komponentenplattformen im Enterprise-Umfeld als wichtige Basisarchitektur diskutiert werden. Eine Mustersprache zu diesem Thema, an der sich die folgende Darstellung grob orientiert, findet sich in [Völter et al. 2002]. Es sei jedoch darauf hingewiesen, dass alle anderen Komponentenansätze, die in Abschnitt 6.2.3 genannt werden, auch ein großes Gewicht in der Praxis haben und heutzutage in den „Werkzeugkasten“ des Software-Architekten gehören.

Komponenten-plattformen im Enter-prise-Umfeld

Typische Beispiele für Komponentenplattformen im Enterprise-Umfeld sind JEE, CCM und .NET. Diese Komponentenplattformen basieren auf der Trennung von technischen Belangen und den fachlichen Belangen an ein Informationssystem. Beispiele für technische Belange im Enterprise-Umfeld sind Verteilung, Sicherheit, Persistenz, Transaktionen, Nebenläufigkeit und Ressourcenmanagement. In anderen Umgebungen, wie eingebetteten Systemen, mögen andere technische Belange eine Rolle spielen. Die technischen Belange werden vom Komponenten-Container – dem zentralen Baustein der genannten Komponentenplattformen – automatisiert übernommen. Diese Architektur wird in Abbildung 6.4-8 dargestellt.

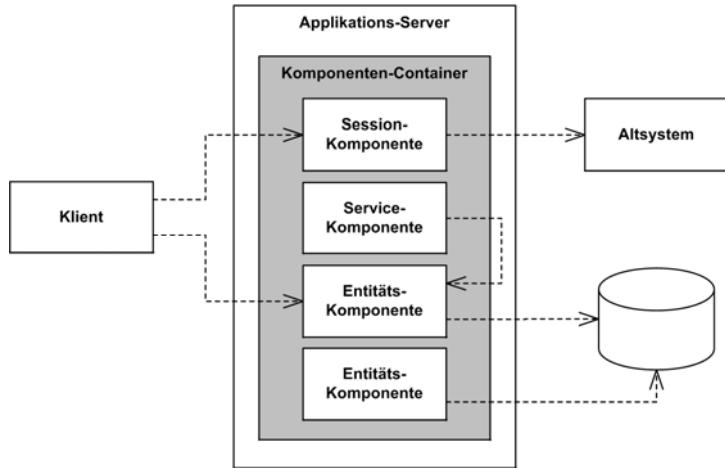


Abb. 6.4-8: Komponenten-Container-Architektur.

Arten von Komponenten

Die fachlichen Anforderungen an ein komponentenbasiertes System werden durch die Komponenten realisiert. In den genannten Komponentenansätzen werden verschiedene Arten von Komponenten unterschieden, die diverse Aufgaben erfüllen und unterschiedliche Lebenszeiten haben (auch diese Lebenszeiten werden vom Container gemanagt):

- > *Entitäts-Komponenten* repräsentieren persistente Daten. Die Persistenz wird automatisch vom Container gemanagt. Entitäts-Komponenten existieren gewöhnlich vom Start der Anwendung bis zu ihrer Terminierung.
- > *Session-Komponenten* können einen Zustand während einer Benutzersitzung vorhalten. Das heißt, ihre Lebenszeit entspricht gewöhnlich einer Benutzersitzung.
- > *Service-Komponenten* stellen Dienste zur Verfügung, die innerhalb eines einzigen Aufrufs abgearbeitet werden. Somit entspricht ihre Lebenszeit genau einem Aufruf.

Black-Box-Wieder-verwendung

Im Gegensatz zu objektorientierten Ansätzen verwenden komponentenbasierte Systeme ausschließlich Black-Box-Wiederverwendung auf Basis der Komponentenschnittstelle. Das heißt, die Interaktion zwischen den Komponenten wird durch wohldefinierte Komponentenschnittstellen und Delegation an andere Komponenten realisiert, ohne auf konkrete Implementierungen dieser Komponenten angewiesen zu sein. Die Implementierungen der Komponenten können dadurch unabhängig voneinander weiterentwickelt werden und verschiedene Versionen einer Komponente können parallel unterstützt werden.

Gewöhnlich unterstützten Komponentenplattformen eine Reihe von Middleware-Systemen zur verteilten Kommunikation, wie bspw. CORBA, RMI oder .NET. Viele Komponentenplattformen sind in existierenden Anwendungsservern integriert.

Komponenten und Middleware

In vielen Situationen können nicht ständig alle Instanzen einer Komponente aktiv im Speicher des Servers vorgehalten werden, da dies zu Ressourcenproblemen führen würde. Der Container kann einer physikalischen Instanz nacheinander beliebig viele logische Instanzen zuweisen (sogenanntes Pooling von Ressourcen). Ferner kann der Container temporär nicht gebrauchte Komponenteninstanzen aus dem Speicher entfernen und in einer Datenbank zwischenspeichern. Dies bezeichnet man als Passivierung. Die Komponenteninstanzen werden automatisch reaktiviert, wenn sie wieder gebraucht werden. Damit all dies funktioniert, muss der Container den Lebenszyklus der Komponenteninstanzen kontrollieren können. Dazu stellen die Komponenten sogenannte Lebenszyklusoperationen wie *aktivieren*, *zerstören*, *passivieren* etc. zur Verfügung.

Pooling und Passivierung

Für ein tieferes Verständnis von Komponentenplattformen sei das Buch *Server Component Patterns* [Völter et al. 2002] empfohlen.

6.4.11 Serviceorientierte Architekturen

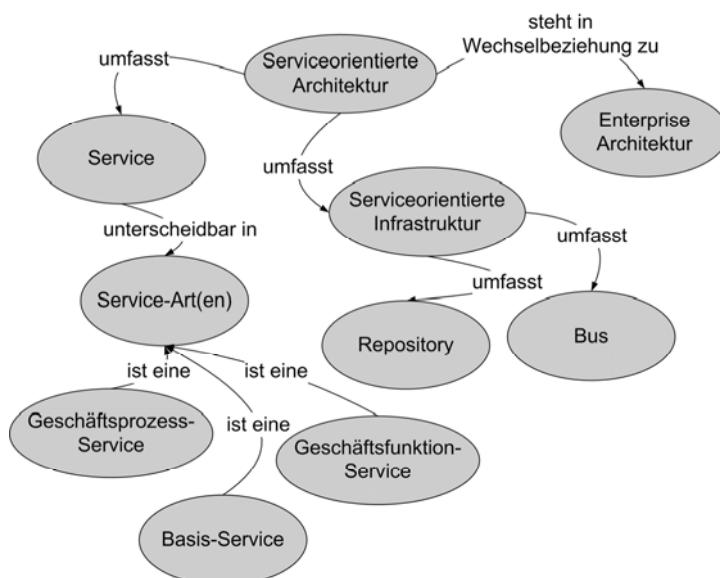


Abb. 6.4-9: Konzept-Überblick zu serviceorientierten Architekturen.

Serviceorientierte Architekturen (SOA) und Services

Serviceorientierte Architekturen sind eine Basisarchitektur, welche die fachlich funktionalen Schnittstellen von Software-Bausteinen als wiederverwendbare, verteilte, lose gekoppelte und standardisiert zugreifbare Dienste (englisch: Services) repräsentiert (siehe [Zdun et al. 2006] und [Zdun und Henrich 2006]).

SOA-Schlüsselkonzepte

Einen Überblick über Schlüsselkonzepte im Bereich serviceorientierter Architekturen gibt die Abbildung 6.4-9.

Eigenschaften von Services

Services, die im Folgenden synonym auch mit *Dienst* bezeichnet werden, zeichnen sich in einer SOA allgemein durch folgende Eigenschaften aus:

- > Services sind generell grober granular als Komponentenschnittstellen und hinsichtlich ihrer Geschäftsrelevanz stärker strukturiert als Komponenten (siehe auch Komponentenarchitekturen in Abschnitt 6.4.10).
- > Services kommunizieren technologienutral und standardisiert mit synchronen oder asynchronen Nachrichten.
- > Services erlauben oft die anonyme Nutzung. Das heißt, man weiß nicht, wer den Service verwendet. Der Service funktioniert so, als ob er seinen Klienten nicht kennen würde. Mit anderen Worten: Der Klient und der Service sind lose gekoppelt.
- > Services sind in gewissen Grenzen selbstbeschreibend. Die Selbstbeschreibung von Services ist häufig realisiert auf der Basis von Metadaten, deren Eigenschaften durch einen Lookup-Service ermittelt werden.
- > Services sind im Idealfall idempotent, zustandsfrei und transaktional abgeschlossen. Als idempotent bezeichnet man generell Arbeitsgänge, die immer zu den gleichen Ergebnissen führen, unabhängig davon, wie oft sie mit den gleichen Daten wiederholt werden. Mathematisch knapper könnte man auch sagen: $a^n = a$, $n > 0$.
- > Services bestehen aus der Service-Schnittstelle und der Service-Implementierung. Die Service-Schnittstelle besitzt Vertragscharakter (siehe hierzu auch *Design-by-Contract* in Abschnitt 6.1.6) und bindet Service-Konsumenten an Service-Anbieter (siehe Architekturprinzip Lose Kopplung in Abschnitt 6.1.1). Die Service-Implementierung ist nicht Teil des Vertrags und unter Einhaltung der Schnittstellenzusagen austauschbar.

Vier Schlüssel-abstraktionen einer SOA

Die strukturellen Schlüsselabstraktionen einer SOA sind in Abbildung 6.4-10 dargestellt. Die dort verwendete ablauftopologische Sicht unterstreicht den vermittelnden (englisch: *intermediate*) Charakter, der SOA-

Systemen zu loser Kopplung verhilft. Der Service-Konsument fragt eine Dienstleistung beim Service-Anbieter nicht direkt nach. Für die Kommunikationsleistung beim Aufruf wie bei der Ergebnisübermittlung sind die Bausteine Bus und Repository zuständig, wenngleich sich eine einfache SOA auch bereits ohne diese realisieren liesse. Der Repository-Baustein dient der Registrierung des Service-Anbieters und unterstützt komplexe Suchen nach diesem. Der Baustein Bus unterstützt die Übermittlung von Nachrichten und schaltet sich somit zwischen alle anderen Bausteine einer SOA. Der SOA Ansatz erlaubt das Herauslösen von nicht-funktionalen Aspekten aus Service-Konsument wie -Anbieter. Beispiele solcher nicht-funktionaler Aspekte sind Sicherheit, Logging, Transformation von Nachrichten und inhaltsbasierte Nachrichtenweitergabe. So bieten Bus und Repository einen konfigurierbaren Zugang zu den oben genannten Aspekten an. Auf diesem Weg können sehr elegant fundamentale Architektur-Prinzipien umgesetzt werden, wie zum Beispiel Entwurf für Veränderung in Abschnitt 6.1.3 oder Separation-of-Concerns in Abschnitt 6.1.4. So kann beispielsweise der Sicherheitsbelang aus den fachlichen Bausteinen herausgelöst und auf der Basis von Bus und Repository konfigurativ zugänglich gemacht werden.

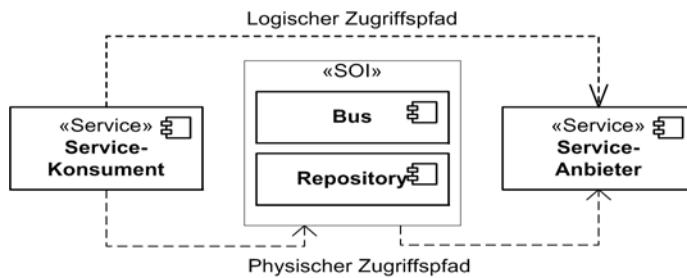


Abb. 6.4-10: Schlüsselabstraktionen einer serviceorientierten Architektur.

Das Hauptaugenmerk einer SOA liegt auf der Ebene der fachlichen Architektur einer Anwendung (siehe hierzu auch Abschnitt 3.2). Der SOA-Ansatz legt jedoch für die Bausteine Bus und Repository den Einsatz vermittelnder Middleware nahe. Für Middleware, die speziell auf die Bedürfnisse von SOA-Lösungen zugeschnitten ist, werden heute verschiedene Begriffe verwendet. Ein eher produktneutraler Begriff ist serviceorientierte Infrastruktur (SOI). Ein weiterer, heute sehr häufig verwendeter Begriff, mit nahezu gleicher Bedeutung, ist Enterprise Service Bus (ESB).

SOA gegenüber SOI

Bezogen auf die in Abbildung 6.4-10 dargestellten Schlüsselabstraktionen einer SOA ist eine SOI auf die Bausteine Bus und Repository beschränkt. Jedoch werden mittlerweile weitere Bausteine zu einer

entsprechend erweiterten SOI gezählt, wie beispielsweise Prozess-Engines. Eine elementare SOA kann technisch bereits auf der Basis von Komponentenplattformen wie CORBA, RMI oder .NET realisiert werden, die alle rudimentäre Bus- und Repository-Bausteine besitzen. Spezielle serviceorientierte Infrastrukturen bieten jedoch Funktionen an, die über die Fähigkeiten der zuvor erklärten Komponentenplattformen hinausgehen:

- > Entstehung neuer, komplexer Dienste durch Orchestrierung (englisch: *orchestration*). Komplexe Dienste entstehen dabei auf der Basis einfacherer Dienste, wobei die Ausführung des komplexen Dienstes durch einen Interpreter oder Dirigenten erfolgt. Ein wichtiger Standard für die Orchestrierung von Web Services ist die Business Process Execution Language (BPEL). Diese Fähigkeit wird heute durch Prozess-Engines realisiert.
- > Unterstützung von Internet-Standards wie Web Services, SOAP und XML. Diese Standards dienen der Umsetzung von entfernten Dienstaufrufen bezüglich Schnittstellenabstraktion, Aufrufprotokoll und Formatbeschreibung.
- > Einbettung und Unterstützung von Enterprise Application Integration (EAI). Siehe hierzu auch das Enterprise-Application-Integration-Anwendungsszenario in Abschnitt 8.3.
- > Injektionspunkte für Aspekte, wie Sicherheit, inhaltsbasierte Nachrichtenweitergabe oder Nachrichtenfilterung. Siehe hierzu auch Abschnitt 6.2.7.
- > Bausteine zur Implementierung von SOA Governance, Service-Bewirtschaftung und -Versionierung.
- > Business Activity Monitoring (BAM) Bausteine zur Unterstützung kontinuierlicher Prozessanalyse und -optimierung.

Enterprise Service Bus (ESB)

Eine Produktkategorie, die sich in der Industrie für weite Bereiche serviceorientierter Infrastrukturen herausgebildet hat, wird mit Enterprise Service Bus (ESB) bezeichnet. Obwohl sich auch andere Produktkategorien im Umfeld serviceorientierter Infrastrukturen einordnen lassen, wird ein wesentlicher Teil der Integrationsanforderungen großer Systeme mithilfe von ESBs umgesetzt. Das Fähigkeitsspektrum von ESBs umfasst:

- > Ein ereignisgetriebenes und nachrichtenorientiertes Verarbeitungsmodell, das auf Dokumentenstandards wie XML basiert und synchrone wie asynchrone Kommunikation unterstützt. Damit ist ein ESB auch Basis einer EDA (englisch: Event-driven Architecture).
- > Inhaltsbasierte Nachrichtenweitergabe sowie Filterfunktionen, die dafür sorgen, dass Nachrichten immer zum richtigen Empfänger gelangen. Siehe hierzu auch message-oriented Middleware in Abschnitt 6.7.1.3.

- > Transformationsfunktionalitäten, die Format- und inhaltliche Veränderungen von Nachrichten unterstützen.
- > Eine Vielzahl von Schnittstellen für gängige Middleware-Systeme, Datenbanken sowie Standardanwendungen. ESBs bieten häufig Konverter bzw. Adapter an, die Abbildungen zwischen Standardschnittstellen unterstützen.

Das Management eines ESB ist verteilt und basiert nicht auf einer zentralen Steuerung. Die ESBs verschiedener Hersteller sind jedoch technisch oft sehr unterschiedlich realisiert. Die Basis der heute gängigen ESBs ist meist gekoppelt an die Produktvergangenheit der entsprechenden Hersteller. Die heute bekannten Produktansätze lassen sich jedoch jeweils einer der hier aufgeführten Kategorien zuordnen:

- > Anwendungsserver-basierte ESBs.
- > ESBs auf der Basis von EAI-Frameworks, bzw. nachrichtenorientierter Middleware (siehe message-oriented Middleware in Abschnitt 6.7.1.3).
- > ESBs, die auf XML Appliances basieren. Eine XML Appliance ist ein eigenes Computersystem, das in der Lage ist, XML-basierte Nachrichten mit anderen Systemen auszutauschen, wobei diese Nachrichten insbesondere inhaltsbasiert, sicher und effizient weiter geleitet werden.

Konkrete ESBs unterscheiden sich über ihre Implementierung hinaus zum Teil sehr deutlich voneinander hinsichtlich ihrer Funktionalität, sowie ihrer Betriebsmerkmale (z. B. Bewirtschaftbarkeit oder Betriebssicherheit). Für Hersteller, die ohnehin bereits ein umfangreicheres Middleware-Angebot haben, komplettieren ESB-artige Produkte die entsprechenden Middleware-Pakete.

Die horizontale Gliederung einer SOA, die bis hier besprochen wurde, ist in erster Linie eine Laufzeitbetrachtung des SOA-Architekturansatzes. Bei dieser Betrachtung spielen serviceorientierte Infrastrukturen eine wichtige Rolle. Darüber hinaus werden SOAs jedoch auch vertikal tiefer gegliedert. Eine SOA umfasst nämlich häufig mehrere Schichten. Wie in Abbildung 6.4-11 beispielhaft dargestellt, können sich diese Ebenen bezüglich Granularität, Freiheit von fachlichen Kontexten, Anzahl der Beziehung zu anderen Services oder auch bezüglich der mittleren Änderungshäufigkeit der jeweiligen Services unterscheiden.

ESB-Implementierungen

Vertikale Ebenen einer SOA

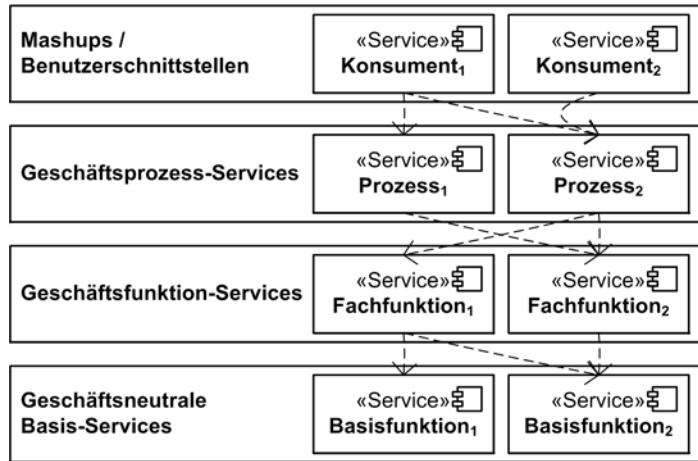


Abb. 6.4-11: Vertikale Gliederung einer SOA.

Nicht-funktionale Anforderungen an eine SOA, bzw. einzelne Services, wie zum Beispiel Wiederverwendbarkeit sind demnach ebenenabhängig zu bewerten. So können geschäftsneutrale Basis-Services in unterschiedlichen Geschäftskontexten leichter wiederverwendet werden. Für einen Geschäftsprozess-Service dagegen wird Wiederverwendbarkeit nur sehr viel schwerer zu erreichen sein.

SOA und Enterprise-Architektur

Der SOA-Ansatz verdrängt zunehmend den Begriff der Anwendung (englisch: *application*) zu Gunsten des Service-Begriffs. Das liegt in erster Linie daran, dass Services weitgehend feiner granular sein sollen sowie in der Lage, unabhängig vom Geschäftskontext existieren zu können. Dies ist gut und gewünscht und erhöht die Möglichkeit, einen entsprechenden Service leichter wiederzuverwenden. Gleichzeitig erhöht dieser Anspruch jedoch auch die Komplexität einer SOA. Ein Service, der vorher lediglich im Rahmen einer einzelnen Anwendung verwendet wurde und existierte, soll jetzt ausserhalb der Grenzen dieser Anwendung verwendet werden können. Anwendungen, welche solche Services gestern noch beherbergten, waren jedoch nie einfach nur seelelose Funktions-Container, sondern begrenzten immer auch den Deutungsraum, auf den sich eingebettete Services bezogen. Einfache Fragen wie „Was ist ein *Produkt*?“ oder „Was bedeutet für mich das Konzept *Kunde*?“ konnte eine Geschäftsfunktion auf den begrenzten Deutungsraum beziehen, in dem sie existierte. An dieser Stelle ist grundsätzlich zu bemerken, dass eine Integration auf hohem, semantischen Niveau häufig nur schwer gelingt, was dazu führt, dass später heterogene „Kunden-Konzepte“ integriert werden müssen. Beginnt man nun im SOA-Sinn eine solche Geschäftsfunktion allein zu stellen und aus dem

Kontext geschlossener Anwendungen zu lösen, erweitert sich der Deutungsraum signifikant – in vielen Fällen um Größenordnungen. Er dehnt sich beispielsweise aus auf die gesamte Abteilung, den Bereich, das Unternehmen oder ein betrachtetes Partnernetz. Mussten zuvor nur im Rahmen einzelner Anwendungen Begriffe wie Kunde oder Produkt geklärt werden, ist dies nun für den erweiterten Raum zu leisten. Da ein systemübergreifendes Verständnis von Grundkonzepten in die Verantwortung von Enterprise-Architektur fällt, ist der Erfolg einer umfassender angelegten SOA-Initiative direkt mit der Thematik Enterprise-Architektur verknüpft (siehe hierzu auch Enterprise-Architektur in Abschnitt 3.2).

Bis hierher wurden Topologie und nicht-funktionale Aspekte von SOA untersucht. Eine der grössten Herausforderung für den Architekten ist jedoch die problemangemessene fachliche bzw. funktionale Zerlegung (englisch: *decomposition*). So hat der Architekt z. B. die Frage danach zu beantworten, wie der gesamte Lösungsraum von Prozessen und Anwendungen in vernünftige Teillösungen bzw. Services zerlegt werden soll. Das Optimierungsziel, das der Architekt bei seinem Zerlegungsansatz häufig verfolgen wird, ist, auf zukünftige Änderungen in der Spezifikation einfach und mit geringem Aufwand reagieren zu können (siehe dazu auch *Entwurf für Veränderung* in Abschnitt 6.1.3). Ein Architekt könnte sich aber auch fragen, wie eine bestimmte Service-Schnittstelle zu entwerfen ist, um für diesen Service hohe Wiederverwendbarkeit zu erzielen. Die Verantwortung dafür, solche Fragen problemangemessen zu beantworten, teilt sich der Architekt mit den Experten der entsprechenden Geschäftsdomäne. Die Güte eines gegebenen SOA-Entwurfs ist aber, mehr als bei anderen Basisarchitekturen, abhängen vom fachlichen Zugang, Verständnis sowie der Erfahrung der beteiligten Experten.

Grundsätzlich unterscheidet man die folgenden Herangehensweisen zur Modellierung des Lösungsraumes:

- > *Top-down-Ansatz*: Beim Top-down-Ansatz geht der Architekt von gegebenen Prozessen aus und identifiziert die funktional notwendigen Software-Bausteine zu deren Realisierung.
- > *Bottom-up-Ansatz*: Beim Bottom-up-Ansatz geht der Architekt von der Menge schon verfügbarer Software-Bausteine aus und versucht, diese in neue Anwendungskontexte hinein abzubilden.
- > *Der Meet-in-the-Middle-Ansatz* schließlich verbindet den Top-down mit dem Bottom-up-Ansatz. Top down werden funktionale Bedürfnisse ermittelt. Bottom up wird eine Bestandsaufnahme schon existierender Services durchgeführt. Auf der Basis dieser Informationen

SOA-Modellentwicklung

können jetzt noch fehlende Funktionen ermittelt und deren Realisierung geplant werden. Diese Vorgehensweise wird im Detail in dem Business-Driven-Service-Muster [Zdun und Henrich 2006] erklärt.

Einsatzszenarien einer SOA

Eine typische Situation, in der eine SOA sinnvoll zum Einsatz kommen kann, ist eine Fusion zweier Unternehmen. Häufig existieren in solchen Situationen bereits unterschiedliche Infrastrukturen, Programmiersprachen, Komponentenplattformen, Middleware-Systeme etc. In solchen Unternehmen ist auch eine Vielzahl von Geschäftsprozessen häufig schon definiert und implementiert. Obwohl auf Infrastrukturebene die Heterogenität eher zunehmen wird, überschneiden sich die Geschäftsprozesse der beteiligten Unternehmen meistens beträchtlich. All diese Software-Bausteine müssen nun irgendwie integriert, aussortiert oder ersetzt werden. Derart heterogene Situationen auf technischer wie fachlicher Ebene sind durchaus nicht allein auf Unternehmensfusionen beschränkt: Es gibt viele Beispiele, in denen diese Form funktionaler Überlappung, bzw. infrastruktureller Heterogenität bereits zwischen einzelnen Abteilungen ein und desselben Unternehmens zu beobachten sind. Solange solche Abteilungsbereiche nichts miteinander zu tun haben, stellt dies noch kein gravierendes Problem dar. Ändern sich aber plötzlich Geschäftsfelder, geschäftliche oder gesetzliche Rahmenbedingungen und wird eine enge Vernetzung der genannten Abteilungen als geschäftskritisch beurteilt, dann unterscheidet sich diese Situation kaum noch von einer Situation bei Fusionen.

Weitere Beispiele, in denen SOAs sinnvoll zum Einsatz kommen können:

- > Entwicklung eines Mashups basierend auf Web Services zur schnellen Realisierung einer Anforderung im Unternehmen.
- > Wiederverwendung eines im Unternehmen implementierten Währungsumrechnungsdienstes. Der Effekt dieses Szenarios liegt in einer Reduktion von Redundanzen sowie Senkung von Implementierungs-, Test- und Wartungskosten.
- > Zugänglichmachen der Funktionalität eines Altsystems (z. B. AS 400) via Web Services, um getätigte Investitionen zu schützen.
- > Realisierung der funktionalen Verbindung zu einem Geschäftspartner über eine abgesprochene Service-Schnittstelle. Dieser Ansatz erlaubt die Durchsetzung von Regeln direkt auf dem Aufrufpfad und damit ausserhalb der beteiligten Geschäftslösungen im Unternehmen.

6.4.12 Sicherheitsarchitekturen

Vor der vertiefenden Betrachtung des Themas Sicherheitsarchitekturen ist es wichtig zu verstehen, dass Sicherheit ein sogenannter durchdringender Belang (englisch: *crosscutting concern*) ist. Das heißt aus Sicht des Software-Architekten [Schumacher et al. 2005], dass die Sicherheitsarchitektur einer Anwendung nicht vollständig in der Software-Architektur derselben aufgeht. Der Sicherheitsaspekt ist sogar ein hochgradig verteilter Aspekt. Er wird über viele Systembausteine hinweg realisiert, die nicht unmittelbar der Software-Architektur betrachteter Anwendungen zugerechnet werden können. Einige Beispiele solcher Systembausteine sind Firewalls, Public-Key-Infrastrukturen (PKI), Reverse Proxies, Web-Access-Management-Lösungen (WAM) und Verzeichnisdienste.

Sicherheit als verteilter Aspekt

Der Begriff Sicherheitsarchitektur bezieht sich damit auf:

- > Eine zu schützende bzw. sichernde Anwendung.
- > Weitere Systembausteine, die nicht direkt zur betrachteten Anwendung gehören und damit unterliegender Sicherheitsinfrastruktur zugerechnet werden.

Einen Überblick über Schlüsselkonzepte im Bereich von Sicherheitsarchitekturen gibt die Abbildung 6.4-12.

Schlüsselkonzepte von Sicherheitsarchitekturen

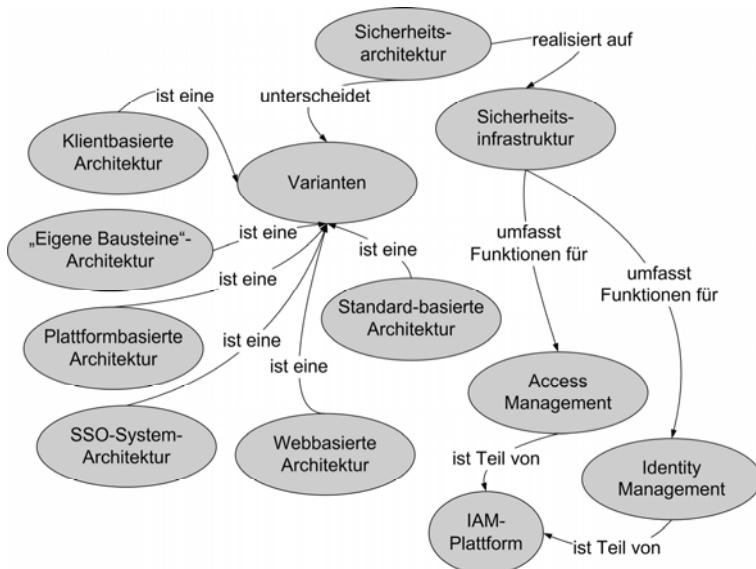


Abb. 6.4-12: Konzept-Überblick zu Sicherheitsarchitekturen.

Voraussetzungen für Sicherheitsarchitekturen

Bevor eine umfassende Sicherheitsarchitektur für eine Anwendung entworfen werden kann, müssen einige Voraussetzungen erfüllt sein. Die Güte sowie Vollständigkeit, mit der eine Sicherheitsarchitektur auf Anwendungsebene entworfen werden kann, hängt sehr stark vom Umfang und Maß ab, in dem folgende Voraussetzungen erfüllt sind:

- > Existenz von Sicherheitssystemen auf allen Netzwerkebenen, einschließlich Betriebssysteme.
- > Systeme zur Benutzer- und Identitätsverwaltung.
- > Systeme zur Anmeldungsüberprüfung (Authentifizierung).
- > Systeme zur Rechteverwaltung und -durchsetzung (Autorisierung).
- > Systeme zur geschützten Informationsübermittlung (englisch: *privacy*).
- > Systeme zur Gewährleistung von Unbestreitbarkeit (englisch: *non repudiation*).
- > Systeme zur Betriebsüberwachung und Angriffserkennung (englisch: *threat detection*).

Neben den primär technischen Voraussetzungen existieren weitere organisatorische Rahmenbedingungen, auf die ein Sicherheitsarchitekt Bezug nehmen muss:

- > Informations- und Schutzklasseneinteilung im Unternehmen. Solche Einteilungen werden vorgenommen, um das Schutzbedürfnis für Informationen, Daten und Dokumente kategorisch formulieren und in den IT-Systemen durchsetzen zu können.
- > Richtlinien zum Umgang mit schützenswerten Gütern und Informationen auf den Ebenen Abteilung, Unternehmen, Partner, Staat etc.
- > Organisationen und Institutionen im Unternehmen, die für die Planung, Umsetzung sowie Durchsetzung der hier betrachteten Sicherheitsarchitekturen verantwortlich sind.

Identity und Access Management (IAM)

Identity and Access Management (IAM) ist ein häufig verwendeter Dachbegriff für die Gesamtheit der oben genannten Fähigkeiten. Gebräuchliche Synonyme sind IAM-System oder IAM-Plattform. Der Zusammenhang zwischen IAM-System und Sicherheitsarchitektur ist dabei wie folgt: Die Sicherheitsarchitektur einer Anwendung wird auf der Basis der Sicherheitssysteme einer IAM-Plattform entworfen und implementiert.

IAM-Plattformen werden üblicherweise zentral und unternehmensweit implementiert, um allen Anwendungen eines Unternehmens sowohl Sicherheitsfunktionen (englisch: *access management*) als auch Funktionen zur Bewirtschaftung von Identitätsinformationen (englisch: *identity management*) zur Verfügung zu stellen. Unter einem Identity-Management-System (IM-System) versteht man also ein System, das Organisation, Prozesse sowie IT-Infrastruktur umfasst, welche das Anlegen, die Bewirtschaftung sowie die Verwendung von digitalen Identitäten unterstützen. IM-Systeme sind damit keine reinen Software-Systeme.

IAM-Plattformen

Als digitale Identität bezeichnet man die Repräsentation eines Subjekts, also einer Person, eines Prozesses, Dienstes oder Anwendung, mittels

- > eindeutiger Kennung (z. B. künstlicher oder abgeleiteter Schlüssel),
- > eines oder mehrerer Berechtigungsnachweise (z. B. Benutzername und Passwort) und
- > weiterer Attribute (z. B. E-Mail-Adresse, Alter, Position).

Digitale Identität

Unter Access-Management-Systemen (AM-System) versteht man Systeme, welche den unterstützten Anwendungen Funktionen zur Identitätsüberprüfung (Authentifizierung) sowie Zugriffs- und Berechtigungskontrolle (Autorisierung) zur Verfügung stellen.

Access Management (AM)

Personalbewirtschaftungssysteme und -prozesse werden meist direkt auf der Basis von IAM-Plattformen realisiert. Damit gibt es (zumindest für den IM-Bereich) eine sehr enge Verknüpfung zwischen Personalbewirtschaftungs- und Sicherheitssystemen.

Verknüpfung zwischen Personalsystemen und IAM-Plattformen

Die Zielsetzung von IAM-Systemen ist es, den richtigen Personen, Gruppen und Anwendungen zeitgerecht Zugriff auf die richtigen Unternehmensfunktionen zu geben, und zwar auf eine benutzerfreundliche, effiziente, sichere und nachvollziehbare Art und Weise. Es geht IAM-Systemen also nicht ausschließlich um die Erhöhung von Sicherheit für Anwendungen, sondern auch darum, grundsätzliche Zugriffsregelungen effizient zu gestalten. Somit fallen Funktionen wie Single-Sign-On (SSO) sowie Rollen- und Gruppen-Management ebenfalls in den Bereich von IAM-Systemen. Motiviert werden IAM-Systeme in Unternehmen neben Sicherheitsargumenten vor allem durch Kosten- und Redundanzreduktion. So ist es besser, einzelne Anwendungen mit einer unternehmensweiten und zentralen Benutzerverwaltung zu integrieren, anstatt für eine eigene Benutzerverwaltungsfunktion jedes Unternehmenssystems zu sorgen. Ein Beispiel für die Reduktion von Kosten ist die Einführung

IAM-Ziele

Anwendungsintegration mit IAM-Plattform

Leistungsmerkmale von Sicherheitsarchitekturen

einer Single-Sign-On-Lösung, welche dafür sorgt, dass ein großer Teil der Unterstützungsanfragen bei Helpdesks abgesenkt und damit Kosten reduziert werden können.

Die Beispiele zeigen bereits, dass Anwendungen mit einer unternehmensweiten IAM-Plattform integriert werden müssen, bevor die dargestellten Verbesserungen erreicht werden können. Die Umsetzung dieser Integration ist die Aufgabe von Sicherheitsarchitekten.

Die wichtigsten Leistungsmerkmale, bzw. Funktionen von Sicherheitsarchitekturen sind:

- > *Privatsphäre* (englisch: *privacy*) bedeutet, dass Nachrichten, die zwischen zwei Systembausteinen ausgetauscht werden, auf dem Kommunikationspfad selber nicht gelesen, bzw. verstanden werden können. Privatsphäre wird erreicht durch Verschlüsselung der Nachricht vor dem Versenden und Entschlüsselung der Nachricht direkt vor der Weiterverarbeitung auf der Seite des empfangenden Systembausteins.
- > *Integrität* verlangt, dass Nachrichten auf dem Kommunikationspfad nicht änderbar sind. Der Empfänger hat die Gewähr, dass kein unerlaubter Dritter die erhaltene Nachricht unterwegs geändert und sie dann erst weiter geleitet hat. Integrität wird üblicherweise realisiert über Hashing-Verfahren. Bei Hashing-Verfahren wird ein Hashwert über der Originalnachricht berechnet und mit der Nachricht selber versandt. Der Empfänger berechnet einen Vergleichswert auf der Basis der erhaltenen Nachricht und vergleicht anschließend beide Hashwerte miteinander. Stimmen diese überein, ist alles in Ordnung. Wertedifferenz bedeutet Verletzung von Integrität.
- > *Authentifizierung* (englisch: *authentication*) ist der Vorgang der Identitätsüberprüfung. Ein Benutzer beweist mithilfe von Berechtigungsnachweisen gegenüber der Authentifizierungsfunktion, dass er der ist, der er vorgibt zu sein. Ein sehr verbreiteter Typ von Berechtigungsnachweisen ist die Kombination von Benutzernamen und Passwort.
- > *Autorisierung* (englisch: *authorization*) ist der Vorgang der Zugriffskontrolle durch einen entsprechenden Systembaustein. Die Identifizierung eines Benutzers oder Systems ist notwendige Voraussetzung für den Autorisierungsschritt. So kann einem identifizierten Benutzer durch das Zugriffskontrollsystem zum Beispiel die Ausführung einer Funktion versagt, das Lesen einer Wertetabelle aber gleichzeitig gestattet werden.

- > *Unleugbarkeit*, bzw. Unbestreitbarkeit (englisch: *non-repudiation*) ist die Fähigkeit, sicherheitsbezogene Ereignisse zweifelsfrei zu belegen; das heißt also bei Bedarf auch vor Gericht. Häufig wird dieses Leistungsmerkmal realisiert mithilfe von speziell zugriffs geschützten Journals; im Zusammenhang mit entsprechenden Transaktionen bisweilen auch durch digitale Signaturen.
- > *Schutz vor Zerstörung und des Betriebs* (englisch: *intrusion protection*) ist die Menge aller Maßnahmen, die betriebliche Integrität und Vitalität gewährleisten. Hier geht es unter anderem darum, Angriffen wie Denial-of-Service-(DoS)-Attacken eine wirksame Verteidigung entgegenzusetzen.

Weitere Leistungsmerkmale, die sich zum Teil aus den oben genannten Merkmalen ableiten, sind:

- > Entdecken von Sicherheitslücken und -vorfällen.
- > Erarbeitung und Durchsetzung von Sicherheitsrichtlinien.
- > Single Sign-On.
- > Rollen- und gruppenbasierte Zugriffsmodelle.

Bereits die Umsetzung eines vergleichsweise einfachen Leistungsmerkmals wie Authentifizierung kann auf sehr viele unterschiedliche Arten erfolgen. Zum Beispiel könnte eine Anwendung für alle Benutzer deren Kennung sowie Passwörter in einer eigenen Datenbank speichern. Anschließend könnte die Anwendung über einen Dialog diese zwei Werte bei einem anmeldenden Benutzer erfragen, um sie mit den Einträgen der Datenbank zu vergleichen. Existierte ein entsprechender Eintrag, hieße dies, dass sich der Benutzer ausweisen konnte. Im anderen Fall würde die Anwendung dem Benutzer den Zutritt verweigern.

In einem völlig anderen Beispiel hätte eine Anwendung den Authentifizierungsschritt jedoch auch an einen zentralen Verzeichnisdienst delegieren können statt diesen selber zu implementieren.

Im Folgenden werden die drei heute am weitesten verbreiteten Gattungen von Sicherheitsarchitektur erläutert. Jede beschriebene Gattung befasst sich, wenngleich in unterschiedlichem Ausmaß, mit allen Sicherheitsleistungsmerkmalen; also mit Privatsphäre, Integrität, Authentifizierung und Autorisierung.

Abbildung 6.4-13 zeigt Sicherheitsarchitekturen, die auf eigenen Systembausteinen basieren. Bei diesem Ansatz werden alle wesentlichen Sicherheitsfunktionen selbst implementiert. So wird die Authentifizie-

Architektur-Ansätze für Sicherheitsarchitekturen

Sicherheitsarchitektur auf der Basis eigener Systembausteine

rungsfunktion in einer Methode programmiert, die eine Werteüberprüfung auf der eigenen Benutzerdatenbank durchführt. Die Autorisierungsfunktion wird implementiert, indem für den identifizierten Benutzer Rolleninformationen aus einer eigenen Privilegdatenbank gelesen werden. Privatsphäre wird erreicht, indem der Zugriff zu den Datensystemen kontrolliert und Datentransportwege selber verschlüsselt werden. Diese Sicherheitsarchitekturgattung ist nach wie vor sehr verbreitet. Selbst viele Standardsysteme implementieren ihre Sicherheitsarchitektur auf der Basis eigener Systembausteine, da sie nicht davon ausgehen können, in allen Zielumgebungen die notwendigen Systembausteine standardisiert und zentral angeboten vorzufinden. Sicherheitsarchitekturen dieser Art besitzen jedoch viele Nachteile. Sie lassen sich oft nur schlecht mit bestehenden IAM-Systemen integrieren. Als einzige Integrationsmöglichkeit bleibt häufig, Daten aus IAM-Systemen in entsprechende Anwendungen hinein zu provisionieren, was allerdings zu enger Kopplung führt (siehe hierzu auch Lose Kopplung in Abschnitt 6.1.1). Auch genügen selbst implementierte Verschlüsselungsverfahren nur selten modernen Anforderungen. Die Provisionierung bezieht sich hierbei auf die Automatisierung aller Prozesse bezüglich der Erstellung, Verwaltung, Deaktivierung und Löschung von digitalen Identitäten sowie deren Attribute und Berechtigungen.

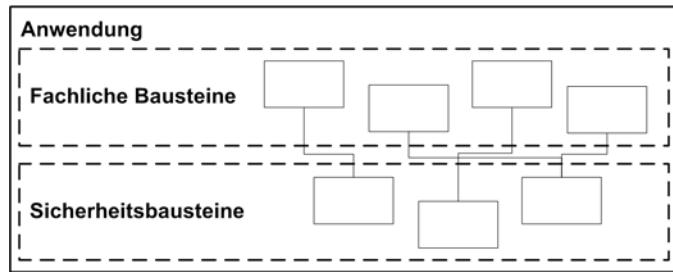


Abb. 6.4-13: Sicherheitsarchitekturen auf Basis eigener Sicherheitsbausteine.

Sicherheitsarchitekturen, die auf Standarddiensten basieren

Abbildung 6.4-14 zeigt eine Architektur, die auf Standarddiensten basiert. Solche Sicherheitsarchitekturen stellen eine Verbesserung dar gegenüber den völlig proprietären Sicherheitsarchitekturen oben. Beispiele für solche Standarddienste sind LDAP-Verzeichnisdienste oder Public-Key-Infrastrukturen (PKI). Der Vorteil von Sicherheitsarchitekturen, die auf Standarddiensten basieren, ist der, dass sicherheitsbezogene Systembausteine ausgewechselt werden können, ohne die Anwendung selbst anpassen zu müssen. So kann die Implementierung eines LDAP-Verzeichnisdienstes, der die Authentifizierungsfunktion implementiert, jederzeit ausgewechselt werden, ohne dass deshalb die An-

wendung angepasst werden muss. Ein Nachteil ist der, dass die Vernetzung von sicherheitsrelevanten Informationen nach wie vor auf der Ebene der Anwendung passieren muss. Werden Benutzerinformationen z. B. aus einem LDAP-Verzeichnis gelesen, Daten, welche die Zugriffsrechtsituation beschreiben, jedoch aus einer Privilegeddatenbank, dann muss die Verknüpfung dieser beiden Informationen nach wie vor auf der Ebene der Anwendung passieren.

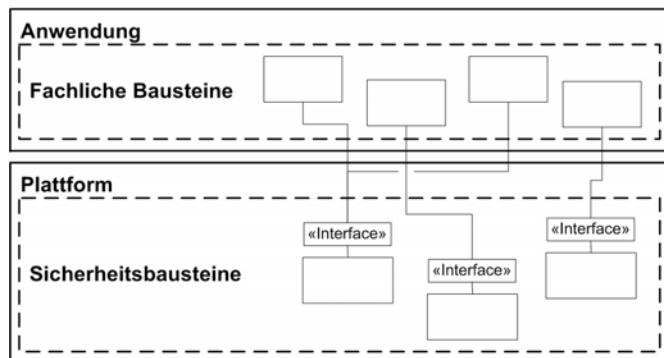


Abb. 6.4-14: Sicherheitsarchitekturen auf der Basis von Standarddiensten.

Grundsätzlich beseitigen Sicherheitsarchitekturen, die auf Komponentenplattformen basieren, die Nachteile der beiden zuvor aufgeführten Kategorien. Wie in Abbildung 6.4-15 dargestellt, definieren Komponentenplattformen den unterstützten Anwendungen gegenüber eigene Standards, welche Sicherheitsfunktionalität auf ein höheres Abstraktions- und Integrationsniveau bündeln. Sie vernetzen beispielsweise Authentifizierungs- mit Autorisierungsfunktionalität. Dabei basieren sie selber wieder auf Standarddiensten; können bezüglich ihrer Implementierung also angepasst werden, ohne dass dies Auswirkung auf betriebene Anwendungen hätte. Ein Beispiel für eine entsprechende Komponentenplattform ist JEE. Der JEE-Server stellt seinen Anwendungen integrierte Sicherheitsfunktionen (z. B. Authentifizierung und Autorisierung) standardisiert (z. B. JAAS-API) zur Verfügung und erlaubt die Einbindung von Sicherheitsbausteinen, die außerhalb des JEE-Servers liegen. Ein Beispiel hierfür ist die Anbindung eines externen Verzeichnisdienstes auf der Basis der JAAS-SPI. Komponentenplattformen unterstützen außerdem die ganze Palette von Sicherheitsfunktionen von eingebauter Benutzer- und Rollenverwaltung über die Unterstützung diverser Authentifizierungsmethoden bis hin zu Verschlüsselungs- und Signaturfunktionen. CORBA oder die .NET-Plattform von Microsoft sind weitere Beispiele für Komponentenplattformen, die eigene Sicherheitsstandards definieren.

Sicherheitsarchitekturen, die auf Komponentenplattformen basieren

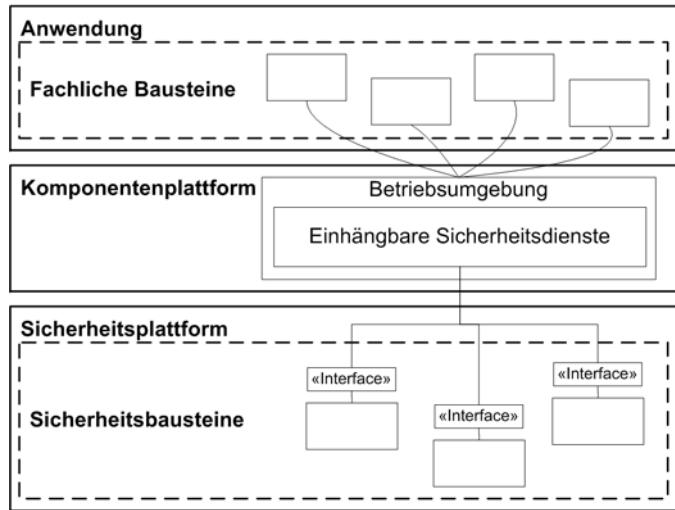


Abb. 6.4-15: Sicherheitsarchitekturen auf der Basis von Komponentenplattformen.

Weitere Sicherheitsarchitekturansätze

Neben den oben dargestellten Ansätzen, existieren heute eine Vielzahl weiterer Sicherheitsarchitekturansätze, von denen lediglich klientenseitige, web-zentrische sowie Single-Sign-On-Architekturen noch näher erläutert werden sollen.

Klientenseitige Sicherheitsarchitekturen

Klientenseitige Sicherheitsarchitekturen. Das zentrale Element klientenseitiger Sicherheitsarchitekturen (siehe Abbildung 6.4-16) ist ein Software-Baustein, der auf dem Endgerät des Benutzers (z. B. Laptop) installiert ist und die Zugangsdaten zu allen registrierten Anwendungen verwaltet. Dieser Baustein schaltet sich in den Zugriff des Benutzers auf eine erkannte Anwendung ein und übernimmt (quasi „von außen“) in erster Linie die Authentifizierung dieses Benutzers. Klientenseitige Sicherheitsarchitekturen sind einfach zu implementieren und vollständig non-invasiv. Sie simulieren den Benutzer gegenüber allen registrierten Anwendungen. Ihr größter Nachteil besteht darin, dass ihre Leistungsfähigkeit auf Anmeldeüberprüfungen begrenzt ist, in denen ein echter Dialog mit einem Endbenutzer stattfindet. Situationen, in denen ein Software-Baustein sich gegenüber einem anderen Software-Baustein anmelden muss, sind durch diese Architektur nicht abgedeckt. Klientenseitige Sicherheitsarchitekturen eignen sich zwar für viele Software-Architekturen von Thin Clients bis Rich Clients, jedoch stoßen sie bei komplexen Authentifizierungsprozessen an ihre Grenzen. Auch helfen sie nicht dabei, Anwendungen von der Last der Benutzerverwaltung zu befreien, indem sie diese zum Beispiel zentralisierten.

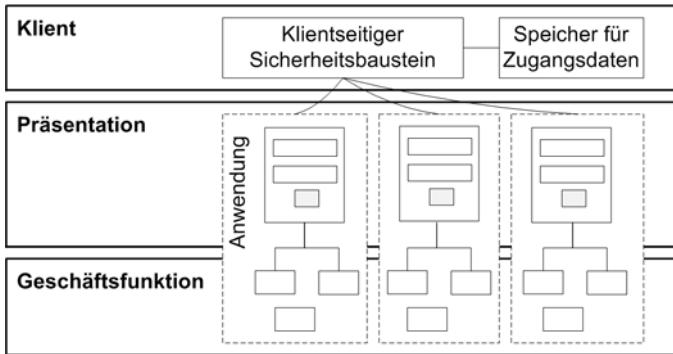


Abb. 6.4-16: Klientenseitige Sicherheitsarchitekturen.

Web-zentrische Sicherheitsarchitekturen, die auch als Reverse-Proxy-Architekturen bekannt und in Abbildung 6.4-17 dargestellt sind, sind beschränkt auf Web-Anwendungen. Ein Web-Baustein (Reverse Proxy) schützt statische und dynamische Web-Anwendungen. Aus Sicht des Software-Architekten liegt ihr Vorteil darin, dass Authentifizierung, Autorisierung sowie die Sicherheitssitzung auf die Systemgrenzen (Perimeter) wandern und nicht mehr in der Anwendung selbst programmiert werden müssen. Web-zentrische Sicherheitsarchitekturen können offene Token-Standards unterstützen sowie selbst Token-Formate einführen. Anwendungen sind entweder direkt mit dem web-zentrischen Sicherheitsbaustein integriert oder werden auf einer Komponentenplattform betrieben, die ihrerseits integriert wurde.

Web-zentrische Sicherheitsarchitekturen

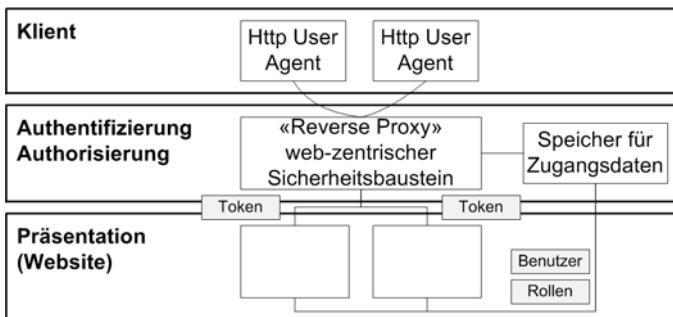


Abb. 6.4-17: Web-zentrische Sicherheitsarchitekturen.

Single-Sign-On-Bausteine, wie in Abbildung 6.4-18 dargestellt, sind Systembausteine auf der Sicherheitsinfrastrukturebene. Sie übernehmen Authentifizierung, Sitzungsverwaltung sowie Tokenverwaltung und -verifizierung. Idealerweise ist der Authentifizierungsschritt ein Teil der ersten Systemanmeldung (z. B. Anmeldung am Betriebssystem). Sobald der Benutzer erfolgreich angemeldet werden konnte, legt der

Single-Sign-On-Architekturen

Single-Sign-On-Baustein eine Sicherheitssitzung an und stellt dem Benutzer einen Schlüssel (Token) aus, der diese Sitzung repräsentiert. Von jetzt an muss sich der Benutzer bei allen Anwendungen, die diesen Schlüssel akzeptieren, nicht mehr einzeln anmelden. Vorteil wie Nachteil von Single-Sign-On-Architekturen ist ihre tiefen Integration mit entsprechenden Anwendungen.

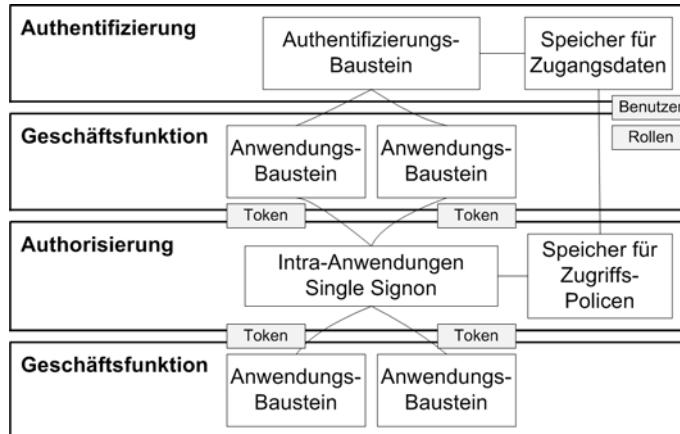


Abb. 6.4-18: Single-Sign-On-Architekturen.

6.4.13 Zusammenfassung

Zusammenfassung: Basisarchitekturen

- > Es gibt grundlegende Basisarchitekturen, die in vielen Systemen zum Einsatz kommen.
- > Schichtenarchitekturen dienen der Strukturierung von Architekturen, indem sie Bausteine in Schichten anordnen und jede Schicht der darüber liegenden Schicht Dienste durch Schnittstellen bereitstellt.
- > Datenflussarchitekturen strukturieren eine Architektur entlang der Datenflüsse und machen insbesondere dann Sinn, wenn eine komplexe Aufgabe in eine Reihe einfacher Aufgaben zerteilt und dann als Kombination unabhängiger Aufrufe dargestellt werden kann.
- > In einer Shared-Repository-Architektur stellt ein Baustein des Systems einen zentralen Datenspeicher zur Verfügung.
- > Eine grundlegende Frage ist die der Zentralisierung gegenüber der Dezentralisierung. Hier muss oft ein Kompromiss gefunden werden.
- > Das klassische Client-/Server-Modell basiert auf einer 2-Tier-Architektur.
- > 3-Tier-Architekturen erweitern die 2-Tier-Architektur, indem sie eine Zwischenschicht zwischen Klient und Datenbankserver einführen.

- > 2-Tier-Architekturen und 3-Tier-Architekturen sind Spezialfälle von n-Tier-Architekturen.
- > Die Entscheidung zwischen einem Rich Client und einem Thin Client basiert auf der Frage, wie man die Funktionalität zwischen dem Klienten und dem Server aufteilt.
- > Peer-to-Peer ist eine Basisarchitektur, die eine Reihe von gleichwertigen Peers zur (verteilten) Kommunikation nutzt.
- > Eine Publish-/Subscribe-Architektur ist eine Basisarchitektur bei der Aufrufe nicht direkt unter den Kommunikationsteilnehmern versendet, sondern durch einen Vermittler weitergeleitet werden. Typischerweise wird in einer Publish-/Subscribe-Architektur über asynchrone Ereignisse kommuniziert.
- > Die Middleware ist eine zentrale Technologie für viele verteilte Systeme. Sie ist eine Plattform, die Anwendungen Dienste für alle Aspekte der Verteilung anbietet, wie verteilte Aufrufe, effizienten Zugriff auf das Netzwerk, Transaktionen und viele andere.
- > Komponentenplattformen basieren auf der Trennung von technischen Belangen und den fachlichen Anforderungen. Die technischen Belange werden automatisiert von einem Container übernommen. Beispiele für technische Belange im Enterprise-Umfeld sind Verteilung, Sicherheit, Persistenz, Transaktionen, Nebenläufigkeit und Resourcenmanagement.
- > Serviceorientierte Architekturen (SOA) sind eine Basisarchitektur, welche die fachlich funktionalen Schnittstellen von Software-Bausteinen als wiederverwendbare, verteilte, lose gekoppelte und standardisiert zugreifbare Services repräsentiert.
- > Sicherheitsarchitekturen beziehen sich auf eine zu schützende Anwendung und eine darunter liegende Sicherheitsinfrastruktur.

6.5 Referenzarchitekturen

In den vorangegangenen Abschnitten wurden wichtige, architektonische Gestaltungsmittel wie Architektur-Prinzipien, -Taktiken, -Stile und -Muster vorgestellt, die die Grundlage für erfolgreiche Architekturen bilden. Überdies wurde die Anwendung dieser Mittel in gängigen Basisarchitekturen diskutiert. Diese Mittel repräsentieren Lösungen für allgemeine, architektonische Anforderungen respektive Qualitäten. IT-Systeme werden jedoch nicht primär aufgrund ihrer architektonischen Eleganz beurteilt. Vielmehr müssen sie fachliche Anforderungen befriedigen und einen konkreten Nutzen für ihren Auftraggeber bieten. Dies

Unterschiedliche Aspekte und Anforderungen

stellt Architekten vor eine große Herausforderung, da sie zwar zum einen Experten in grundlegenden Architektur-Fragen sein müssen, zum anderen aber auch die besonderen Charakteristika und Bedürfnisse verschiedener Industrien kennen und während des Architektur-Entwurfs würdigen müssen. Erst durch das Zusammenführen allgemeiner Architektur-Expertisen und industriespezifischer Kenntnisse können IT-Lösungen entstehen, die Geschäftsstrategien unterstützen und zur Differenzierung im Wettbewerb beitragen. Die industriespezifischen Kenntnisse erstrecken sich dabei nicht nur auf Geschäftsmodelle und -prozesse sowie deren IT-basierte Unterstützung, sondern vielmehr auch auf die in einer Industrie anzutreffenden IT-Systemlandschaften und ihre Anforderungen. Um in verschiedenen Industrien als Architekt erfolgreich zu agieren, benötigt ein Architekt Mittel, die das Wissen und die Erfahrung der allgemeinen Architektur-Disziplinen mit denen der konkreten Industrien vereinen.

Abbildung 6.5-1 veranschaulicht, dass Referenzarchitekturen sowohl allgemeines Architektur-Wissen, wie Muster und Konzepte, als auch Konzepte und Wissen der Domäne kombinieren.

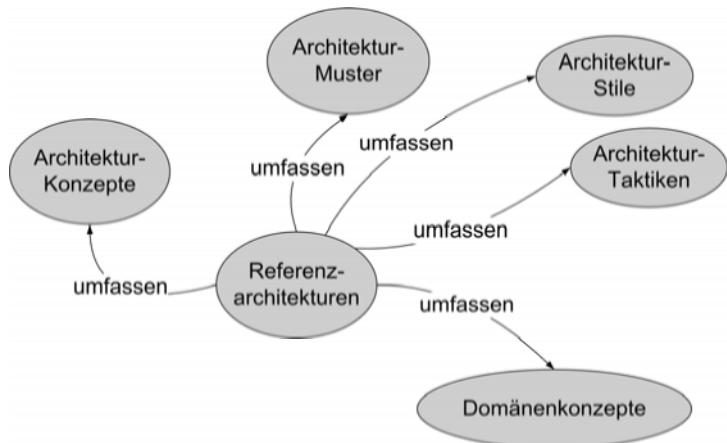


Abb. 6.5-1: Übersicht Referenzarchitekturen.

6.5.1 Definition und Bestandteile

Definition:

Referenzarchitekturen

Referenzarchitekturen kombinieren allgemeines Architektur-Wissen und allgemeine -Erfahrung mit spezifischen Anforderungen zu einer architektonischen Gesamtlösung für einen bestimmten Problembereich. Sie dokumentieren die Strukturen des Systems, die wesentlichen Systembausteine, deren Verantwortlichkeiten und deren Zusammenspiel.

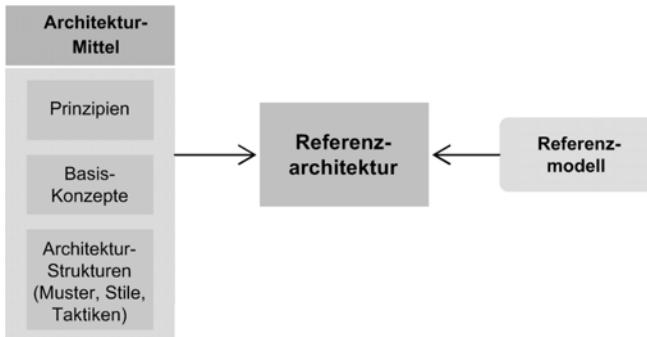


Abb. 6.5-2: Bestandteile einer Referenzarchitektur.

Wie aus Abbildung 6.5-2 ersichtlich, entstehen Referenzarchitekturen also zum einen auf Basis bewährter Architekturmittel und zum anderen auf der Grundlage von spezifischen Anforderungen in Form von gewünschter Funktionalität, die in einem Referenzmodell zum Ausdruck kommt.

Bestandteile

Ein Referenzmodell enthält die spezifischen Charakteristika des adressierten Problembereichs. Die Funktionalität wird dabei in dedizierte Funktionsbausteine unterteilt. Ein Referenzmodell dokumentiert diese Bausteine und die Informationsflüsse zwischen den Bausteinen [Bass et al. 2003]. Abbildung 6.5-3 illustriert die Struktur eines Referenzmodells exemplarisch.

Referenzmodelle

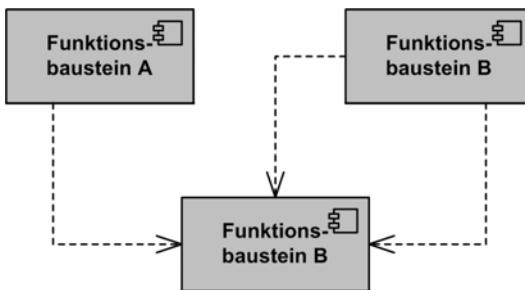


Abb. 6.5-3: Struktur eines Referenzmodells.

Die konzeptionellen Funktionsbausteine erfüllen in ihrem Zusammenspiel die Anforderungen des Problembereichs. Ein Referenzmodell besagt noch nicht, wie ein zu realisierendes IT-System diese Funktionalität erfüllt. Diese Aufgabe übernimmt die Referenzarchitektur, indem sie beschreibt, wie die Funktionsbausteine auf Systembausteine verteilt werden. Darüber hinaus erläutert sie deren Verantwortlichkeiten und ihr Zusammenspiel [Hofmeister et al. 1999].

Referenzarchitekturen als Abbild von Referenzmodellen

6.5.2 Einsatz und Vorteile von Referenzarchitekturen

Einsatz von Referenzarchitekturen

Ein Architekt bedient sich der Referenzarchitekturen während des Architektur-Entwurfs und überführt diese in konkrete Architekturen. Hierbei gilt es abzuwägen, welche Bestandteile einer Referenzarchitektur für die konkrete Problemstellung benötigt werden. Oftmals sind Referenzarchitekturen sehr umfangreich. Daher ist eine direkte Abbildung in aller Regel nicht sinnvoll. Stattdessen sollten immer nur die Bestandteile umgesetzt werden, die auch wirklich benötigt werden, um die Komplexität der zu entwerfenden Architektur zu reduzieren (siehe Abbildung 6.5-4).

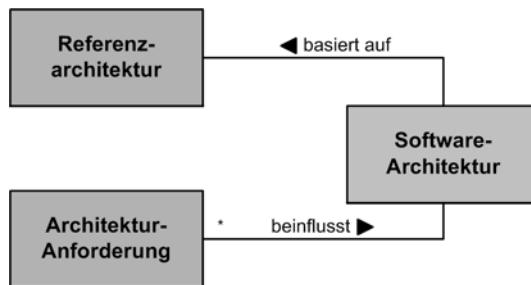


Abb. 6.5-4: Einsatz von Referenzarchitekturen.

Vorteile von Referenzarchitekturen

Der Einsatz von Referenzarchitekturen bringt folgende Vorteile:

- > Man kann auf dem Wissen und der Erfahrung anderer, die zum Entwurf der Referenzarchitektur beigetragen haben, aufbauen.
- > Eine Referenzarchitektur senkt das Risiko, eine nicht tragfähige Architektur zu entwerfen.
- > Eine Referenzarchitektur führt zu einer gesteigerten Qualität der eigentlichen Architektur, da diese auf einem bewährten Architekturmittel beruht.
- > Eine Referenzarchitektur senkt die Kosten des Architektur-Entwurfs, weil sie bereits wichtige Erkenntnisse aus dem Bereich der Problembereichsanalyse enthält und dadurch den Aufwand für diese Tätigkeit reduziert.
- > Der Einsatz einer Referenzarchitektur erlaubt die schnellere Entwicklung eines IT-Systems und ein besseres Time-to-Market.

6.5.3 Anforderungen an Referenzarchitekturen

Um auch wirklich Vorteile im Architektur-Entwurf zu bieten, müssen gute Referenzarchitekturen verschiedene Eigenschaften aufweisen:

- > Sie müssen auf bewährten Prinzipien, Muster, Stilen und Taktiken basieren.
- > Sie müssen erfolgreich eingesetzt worden sein. Erst durch den erfolgreichen Einsatz beweisen Referenzarchitekturen ihre Praxistauglichkeit. Bevor man sich für eine Referenzarchitektur entscheidet, sollte also immer sichergestellt sein, dass diese bereits in einem ähnlichen Kontext eingesetzt worden ist.
- > Sie müssen an konkrete Bedürfnisse anpassbar sein. Wie bereits zuvor kurz geschildert, können Referenzarchitekturen sehr umfangreich sein. Deshalb muss es möglich sein, die konkrete Architektur auf Basis der Referenzarchitektur zu entwerfen und diese schrittweise zu erweitern, wenn neue Anforderungen auftreten.
- > Sie müssen umfassend dokumentiert sein. Eine Referenzarchitektur kann erst erfolgreich eingesetzt werden, wenn dem Architekten eine aussagekräftige Dokumentation zur Verfügung steht. Aus dieser Dokumentation sollte klar hervorgehen, welche Architekturmittel angewandt wurden und wie das *Referenzmodell* auf die Referenzarchitektur abgebildet wurde. Ferner sollten die architektonischen Kräfte (englisch: *forces*) genannt und aufgezeigt werden, wie die Architektur diese ausgleicht. Deshalb bietet sich ein Dokumentationsstil wie bei Architektur-Mustern an.

Anforderungen an gute Referenzarchitekturen

6.5.4 Arten von Referenzarchitekturen

In der Praxis findet man verschiedene Arten von Referenzarchitekturen. Einige von ihnen besitzen einen Standardcharakter, das heißt, sie müssen, respektive sollen, genauso umgesetzt werden, wie sie beschrieben werden. So existieren z. B. Referenzarchitekturen für Komponentenplattformen, die illustrieren, wie man auf Basis der Plattform Architekturen für unterschiedliche Problemstellungen realisiert. Bekannte Vertreter dieser Art sind die JEE Blue-Prints von Sun Microsystems für die JEE-Plattform [Sun 2004a]. Die Realisierung einer serviceorientierten Architektur oder die Entwicklung einer web-basierten Shopping-Lösung sind Beispiele für solche Referenzarchitekturen. Der Fokus dieser Architekturen liegt dabei klar auf dem korrekten Einsatz der darunter liegenden Plattform. Eine Referenzarchitektur dieser Kategorie umfasst nicht nur konzeptionelle Artefakte, wie Architektur-Diagramme, sondern

Plattformbezogene Referenzarchitekturen

auch konkrete Implementierungen in Form von Quelltext oder lauffähigen Bausteinen.

Industriebezogene Referenzarchitekturen

Darüber hinaus sind umfangreiche, industriespezifische Referenzarchitekturen zu nennen, die auf die konkreten Bedürfnisse von Unternehmen zugeschnitten sind. Viele IT-Dienstleister und -Beratungshäuser bieten beispielsweise Referenzarchitekturen für die Telekommunikations-, Luftfahrt-, Banken- oder Versicherungsindustrie an, um nur einige Bereiche zu nennen. Ihre zugrunde liegenden Referenzmodelle umfassen hierbei Funktionsbausteine und Informationsflüsse, die in der Regel die Unterstützung aller wichtigen Geschäftsprozesse eines Unternehmens ermöglichen. Überdies zeigen diese Referenzarchitekturen, wie die Funktionsbausteine auf konkrete, kommerzielle Software-Produkte abgebildet werden. Die Individualentwicklung beschränkt sich normalerweise auf die Anpassung und Integration der Produkte in die IT-Systemlandschaft des Kunden. NGOSS und OSS/J sind Beispiele für solche Referenzarchitekturen (siehe Abschnitt 6.5).

Industrie-übergreifende Referenzarchitekturen

Zwischen die plattform- und industriespezifischen Referenzarchitekturen fallen Referenzarchitekturen, die ein in verschiedenen Industrien anzutreffendes Thema, wie z. B. Supply Chain Management (SCM) und Customer Relationship Management (CRM) behandeln.

Produktlinienarchitekturen

Eine besondere Form von Referenzarchitektur ist eine Produktlinienarchitektur, die die gemeinsame Architektur mehrerer ähnlicher Software-Produkte definiert [Hofmeister et al. 1999]. Sie umfasst die gemeinsamen Systembausteine, deren Verantwortlichkeiten und Zusammenarbeit. Produktlinienarchitekturen werden mit dem Ziel entworfen, Produkte kostengünstiger zu entwerfen, indem sie eine gemeinsame Architektur teilen und unter Umständen sogar vorgefertigte, lauffähige Software-Bausteine wiederverwenden können.

6.5 Beispiel für eine Referenzarchitektur

Beispiel einer industriebezogenen Referenzarchitektur

Die *Next-Generation-Operation-Support-Systems-Initiative (NGOSS)* des TeleManagement-Forums definiert eine umfassende, auf die Telekommunikationsindustrie zugeschnittene Referenzarchitektur [TMF 2004a].



Abb. 6.5-5: Das NGOSS-Referenzmodell im Überblick.

Diese basiert auf dem in Abbildung 6.5-5 vorgestellten und im Folgenden erläuterten Referenzmodell [TMF 2004b]:

- > *Gemeinsames Informations- und Daten-Modell* (englisch: *Shared Information and Data Model (SID)*):

Das gemeinsame Informations- und Daten-Modell ist ein standardisiertes Abbild der Telekommunikations-Domäne. Es definiert die Standardabstraktionen, wie Kunde, Bestellung und Netzwerkdienst. Des Weiteren trifft das Modell klare Aussagen über die Bedeutung der Abstraktionen, ihr Verhalten und ihre Kollaboration.

- > *Sicherheit* (englisch: *Security*):

Dieser Funktionsbaustein definiert die Sicherheitsmechanismen und -grundsätze. Diese orientieren sich an dem Information-Security-Management-Standard [ISO17799 2001].

- > *Grundsätze* (englisch: *Policy*):

NGOSS empfiehlt den Einsatz von Policy-Based-Management (PBM). PBM basiert auf Regeln, die festlegen, wie Bausteine behandelt werden sollen. Diese Regeln werden zur Laufzeit vom System ausgewertet und angewandt.

- > *Geschäftsprozesse* (englisch: *Business Process*):

Dieser Teil des NGOSS-Referenzmodells enthält standardisierte Geschäftsprozesse und -aktivitäten für den Telekommunikationsbereich. Diese werden in der Enhanced Telecom Operations Map (eTOM) zusammengefasst.

NGOSS-Referenzmodell

- > *OSS-Anwendungen* (englisch: *OSS Applications*):
Geschäftsbezogene Funktionalität, die von einem NGOSS-konformen IT-System unterstützt werden sollten, sind in diesem Teil des NGOSS-Referenzmodells zusammengefasst. OSS steht in diesem Zusammenhang für Operations Support System. Näheres zu diesem Themengebiet findet der interessierte Leser in [Terplan 2001].
- > *OSS-Framework-Dienste* (englisch: *OSS Framework Services*):
Grundlegende Dienste, die von verschiedenen OSS-Anwendungen genutzt werden können, werden durch OSS-Framework-Dienste definiert.
- > *Grundlegende Framework-Dienste* (englisch: *Basic Framework Services*):
Primär technische Dienste, wie das Auffinden eines Dienstes in einem Verzeichnis, werden durch grundlegende Framework-Dienste im NGOSS-Referenzmodell abgedeckt. Auf dieser Funktionalität bauen die höherwertigen Dienste (OSS-Framework-Dienste) auf.
- > *Grundlegende Mechanismen* (englisch: *Basic Mechanisms*):
Grundlegende Mechanismen behandeln die Funktionalität, die notwendig ist, um die Kommunikation zwischen den Bausteinen eines NGOSS-Systems und dem Aufruf von Diensten einzelner Bausteinen zu ermöglichen.

NGOSS und RM-ODP

Neben dem Referenzmodell basiert die NGOSS-Referenzarchitektur (Technology Neutral Architecture) auf allgemeinen architektonischen Mitteln und orientiert sich dabei an dem standardisierten Architektur-Modell RM-ODP (siehe Kapitel 4). Sie definiert die architektonischen Aspekte und Bausteine eines verteilten, NGOSS-konformen IT-Systems [TMF 2004b].

NGOSS-Implementierungen

Allerdings bietet das TeleManagement-Forum keine konkrete Implementierung ihrer Referenzarchitektur. Software-Hersteller und IT-Dienstleister nutzen die Referenzarchitektur zur Realisierung konkreter Lösungen. Das TeleManagement-Forum bietet die Möglichkeit, diese zu zertifizieren. Dies ist ein Vorteil für Kunden, da sie bei der Auswahl von NGOSS-Implementierungen auf das TMF-Gütesiegel vertrauen können.

OSS/J als NGOSS-Implementierung

Die NGOSS-Referenzarchitektur ist eine technologieneutrale Architektur. Daher muss diese für den jeweiligen Anwendungsfall auf konkrete Komponentenplattformen abgebildet werden. Aus diesem Grund wurde im Rahmen des Java Community Process (JCP) für die JEE-Plattform die OSS-for-Java-Initiative (OSS/J) von führenden Herstellern ins Leben gerufen, die die NGOSS-Referenzarchitektur auf Basis von JEE implementiert [OSSI 2004].

Die OSS/J-Initiative verfolgt das Ziel, auf OSS/J basierende Software-Bausteine respektive -Produkte für den Telekommunikationsbereich anzubieten. Dadurch können Produkte unterschiedlicher Hersteller zu einer umfassenden Telekommunikationslösung kombiniert werden, indem die Integrationskosten erheblich reduziert werden.

Ziel von OSS/J

Hierzu definiert die OSS/J-Initiative verschiedene Application Programming Interfaces, die auf eTOM von NGOSS basieren. Die API-Spezifikationen spiegeln zum einen die benötigte Funktionalität und zum anderen die architektonisch relevanten Systembausteine wider. Eine Übersicht der APIs gibt Tabelle 6.5-1.

Bestandteile von OSS/J

Tab. 6.5-1: Die OSS/J-APIs im Überblick.

Java API	Beschreibung
OSS Common API	Bestandteil dieses APIs sind grundlegende Kommunikationsmechanismen und Entwurfsrichtlinien, denen alle anderen APIs genügen müssen.
OSS Service Activation API	In einer Telekommunikationsarchitektur (TK-Architektur) muss es möglich sein, Dienste, wie z. B. SMS für einen Kunden nach Vertragsabschluss automatisch zu aktivieren. Das Service Activation API legt die hierfür benötigte Funktionalität und modelliert die relevanten Bausteine.
OSS Quality of Service API	Das Quality of Service API definiert die Funktionalität und Bausteine zur Überwachung und Ermittlung der Qualität von Telekommunikationsdiensten (TK-Diensten). Es ist beispielsweise wichtig zu bestimmen, ob die zur Verfügung stehende Bandbreite in einem Netzwerk unter einen bestimmten Wert sinkt.
OSS Trouble Ticket API	Das Trouble Ticket API wird zur Verwaltung von Fehler-Tickets verwendet. Es umfasst die notwendige Funktionalität und Bausteine im Bereich Fehler-Management und -Verfolgung.
OSS IP Billing API	Die Nutzung von TK-Diensten muss Kunden in Rechnung gestellt werden. Aus diesem Grund bedarf es Bausteinen, die die Rechnungsstellung übernehmen. Diesen Bausteinen und deren Funktionalität widmet sich das IP Billing API, indem es ihre notwendigen Charakteristika definiert.
OSS Inventory API	Eine TK-Architektur besteht aus verschiedenen Netzwerk-Bausteinen, wie Servern, Routern und Switches. Diese Bestandteile zu inventarisieren und deren Lokation z. B. im Fehlerfall schnell abrufen zu können, sind wichtige Bedürfnisse von TK-Anbietern. Aus diesem Grund müssen Inventare geführt werden können. Die hierfür relevante Funktionalität und die benötigten Bausteine werden durch das Inventory API modelliert.
OSS Service Quality Management API	Dieses API widmet sich der benötigten Funktionalität zur Bestimmung der Qualität eines TK-Dienstes und zeigt die hierfür benötigten Bausteine auf.

Verantwortlichkeiten

In ihrer Gesamtheit modellieren die OSS/J-APIs die benötigte Funktionalität einer TK-Architektur. Das Common API adressiert hierbei hauptsächlich die OSS-Framework-Dienste des NGOSS-Referenzmodells. Die anderen APIs widmen sich hingegen der durch die OSS-Anwendungen modellierten Funktionalität und den Bausteinen. Der JEE-Plattform kommt bei OSS/J die Aufgabe der grundlegenden Framework-Dienste sowie der grundlegenden Mechanismen und Dienste zu. Die Verwendung von JEE als Architektur-Plattform hat den großen Vorteil, dass man sich auf eine bewährte Plattform verlassen kann, die wichtige Basisdienste wie Skalierbarkeit und Transaktionssteuerung bietet.

OSS/J-Architektur-Beispiel

Ein Beispiel für eine einfache, auf OSS/J basierende Architektur kann Abbildung 6.5-6 entnommen werden.

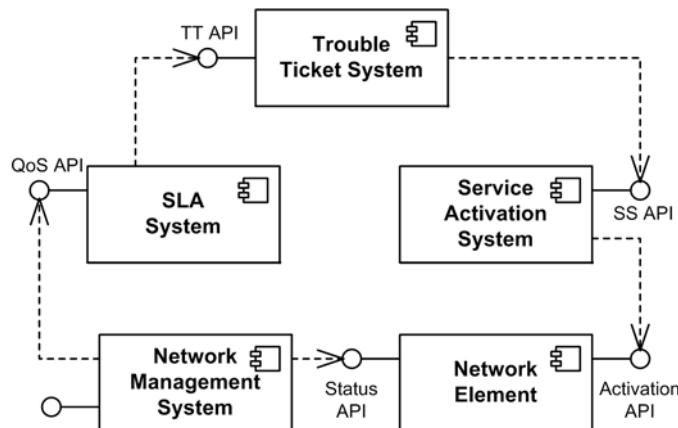


Abb. 6.5-6: Eine einfache OSS/J-basierte Architektur.

Die dargestellte Architektur basiert auf der Verwendung der standardisierten OSS/J-APIs zur Integration verschiedener Subsysteme eines Telekommunikationssystems. Da die benötigten Systeme entsprechende Schnittstellen anbieten, können die Systeme miteinander kommunizieren, ohne dass auf systemspezifische Schnittstellen zurückgegriffen werden muss. Dies reduziert den Integrationsaufwand erheblich.

Bei dem zugrunde liegenden Szenario überwacht ein Netzwerkmanagementsystem Netzwerkelemente. Sobald ein Fehler an einem Netzwerkelement erkannt wird, leitet das Netzwerkmanagementsystem eine entsprechende Nachricht an das SLA-System weiter. Die Nachricht genügt dabei einem durch die OSS/J-Initiative definierten Format. Das SLA-System prüft nun, basierend auf den definierten Service-Level-Vereinbarungen, ob eine Verletzung der Vereinbarungen vorliegt. Falls

dies zutrifft, fordert das SLA-System das Trouble Ticket System auf, ein entsprechendes Trouble Ticket zu erzeugen. Sobald der Fehler des Netzwerkelements durch einen Techniker behoben wurde, wird das Trouble Ticket geschlossen und eine OSS/J-konforme Dienstaktivierungsanfrage an das Service Activation System gestellt. Dieses aktiviert das entsprechende Netzwerkelement wieder.

Dieses Beispiel zeigt vereinfacht, welche Arten von Systemen benötigt werden, um dieses Szenario zu realisieren. Darüber hinaus verdeutlicht es die Struktur einer entsprechenden TK-Architektur sowie die Verantwortlichkeiten und das Zusammenspiel der Subsysteme.

6.5.6 Zusammenfassung

- > Referenzarchitekturen kombinieren allgemeines Architektur-Wissen und allgemeine Erfahrung mit spezifischen Anforderungen zu einer architektonischen Gesamtlösung für einen bestimmten Problembe-reich.
- > Referenzarchitekturen werden gebildet auf Basis von bewährten Architektur-Mitteln (z. B. Prinzipien, Muster, Stilen und Taktiken) und zum anderen auf der Grundlage von spezifischen Anforderungen in Form von gewünschter Funktionalität, die in einem Referenzmodell zum Ausdruck kommt.
- > Ein Referenzmodell enthält die spezifischen Charakteristika des adres-sierten Problembereichs.
- > Die Wahl einer Referenzarchitektur als architektonisches Gestal-tungsmittel bietet große Vorteile, da man auf dem Wissen und der Er-fahrung anderer, die zum Entwurf der Referenzarchitektur beigetra-gen haben, aufbauen kann.
- > In der Praxis findet man verschiedene Arten von Referenzarchiteku-ren: plattformbezogene Referenzarchitekturen, industriebezogene Re-ferenzarchitekturen, industrieübergreifende Referenzarchitekturen.
- > Eine besondere Form von Referenzarchitektur ist eine Produktlinien-architektur, die die gemeinsame Architektur mehrerer ähnlicher Software-Produkte definiert.
- > Die Next-Generation-Operation-Support-Systems-Initiative (NGOSS) des TeleManagement-Forums definiert eine umfassende, auf die Te-lekommunikationsindustrie zugeschnittene Referenzarchitektur.

**Zusammenfassung:
Referenzarchitekturen**

6.6 Architektur-Modellierungsmittel

Modellbildung als fester Bestandteil moderner Software-Entwicklung

Ständige Innovationen hinsichtlich neuer Konzepte der Abstraktion haben die Software-Entwicklung von Beginn an begleitet. Inzwischen hat die Modellbildung als Abstraktionskonzept einen festen Platz in der modernen Software-Entwicklung eingenommen. Im Hinblick auf Software-Architekturen dient die Modellierung als Mittel zur Dokumentation, Spezifikation, Kommunikation, Analyse und Validierung von Architekturen, teilweise auch zur Unterstützung automatischer Code-Generierung. Der letztgenannte Aspekt wurde bereits im Kontext architekturzentrierter MDSD (siehe Abschnitt 6.2.6) beleuchtet.

Aufbau dieses Abschnitts

Zu Beginn dieses Abschnitts werden wichtige Begrifflichkeiten und Konzepte der Modellbildung im Kontext der Software-Entwicklung eingeführt und erläutert. Im weiteren Verlauf dieses Abschnitts wird auf Modellierungssprachen eingegangen, die im Rahmen der Architektur-Modellierung sinnvollerweise zum Einsatz kommen sollten. Es werden mit der Unified Modeling Language (UML), Domain Specific Languages (DSL) und einer speziellen Sprachfamilie von DSLs, den Architecture Description Languages (ADL), drei wichtige Mittel hinsichtlich ihrer Möglichkeiten betrachtet, Architektur präzise zu modellieren. Diese Liste erhebt dabei keinen Anspruch auf Vollständigkeit. So gibt es weitere Möglichkeiten, wie beispielsweise Entity-Relationship-Diagramme oder Notationen aus dem Umfeld von Structured Analysis/Design (SA/D), die ebenfalls dazu verwendet werden können, zumindest Teilespekte einer Architektur formal zu erfassen und zu dokumentieren. Abbildung 6.6-1 gibt einen Überblick über die in diesem Abschnitt vorgestellten Konzepte und Architektur-Modellierungsmittel.

Architektur-Modell ist kein Vorgehensmodell

Es soll an dieser Stelle angemerkt werden, dass eine Architektur-Spezifikation in Form eines Modells kein Vorgehensmodell (siehe Abschnitt 6.6.5) darstellt. Ein Vorgehensmodell definiert konkrete Methoden sowie einen Entwicklungsprozess, die in ihrer Kombination vorgeben, *wie* und *wann was* gemacht werden muss. Ein Vorgehensmodell gibt also einen geordneten Rahmen vor, welche Modelle, beispielsweise das Architektur-Modell, im Rahmen des Lebenszyklus eines Projekts erstellt werden sollten. Soll allerdings das Vorgehensmodell präzise formuliert werden, so ist die Modellierung ein geeignetes Mittel. Ein entsprechendes Metamodell wurde in Abschnitt 6.5 betrachtet.

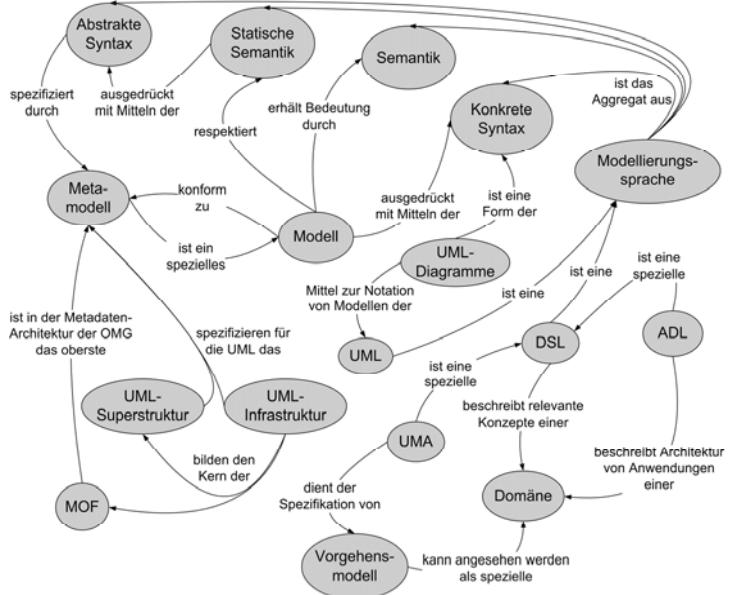


Abb. 6.6-1: Grundlegende Konzepte der Modellierung und Architektur-Modellierungsmittel – Überblick.

6.6.1 Grundlegende Konzepte der Modellierung

In der Literatur findet sich eine Vielzahl verschiedener Definitionen des Modellbegriffs. Dies ist unter anderem dadurch begründet, dass sich der Einsatz von Modellen in verschiedensten Ingenieurdisziplinen und Wissenschaftsgebieten etabliert hat. Eine allgemein anerkannte und domänenübergreifende Definition des Modellbegriffs ist in [Stachowiak 1973] zu finden:

Der Modellbegriff

- > **Abbildung**. Ein Modell ist immer ein Abbild von etwas, eine Repräsentation natürlicher oder künstlicher Originale, die selbst wieder Modelle sein können.
- > **Verkürzung**. Ein Modell erfasst nicht alle Attribute des Originals, sondern nur diejenigen, die dem Modellschaffer bzw. Modellnutzer relevant erscheinen.
- > **Pragmatismus**. Pragmatismus bedeutet so viel wie „Orientierung am Nützlichen“. Ein Modell ist einem Original nicht von sich aus zugeordnet. Die Zuordnung wird durch die Fragen für wen, warum und wozu relativiert. Ein Modell wird vom Modellschaffer bzw. Modellnutzer innerhalb einer bestimmten Zeitspanne und zu einem bestimmten Zweck für ein Original eingesetzt. Das Modell wird somit interpretiert.

Definition:
Modell

Modelle sind demnach eine Abstraktion des Originals weil sie nicht sämtliche seiner Details berücksichtigen, sondern nur solche, die dem Interessenvertreter des Modells für seine Zwecke nützlich sind.

Modelle in der Software-Entwicklung

Im Rahmen dieses Buches werden Modelle im Kontext des Entwurfs objektorientierter Software-Systeme betrachtet. Auch in der Software-Entwicklung versteht man unter einem Modell das abstrahierende Abbild eines Originals. Dieser Abschnitt führt die wichtigsten Begrifflichkeiten im Kontext der Modellierung in der Software-Entwicklung ein und entwickelt einen konzeptionellen Überbau zur Thematik.

Metamodellierung

Die Grundlage jeglicher automatisierten Verarbeitung von Modelldaten ist die eindeutige Definition eines Modells in einer formalen Sprache. Dieser Forderung Rechnung tragend hat sich in der Software-Entwicklung eine spezielle Art der Modellierung herauskristallisiert. Hier werden Modelle in der Regel wiederum durch Modelle beschrieben, welche dann *Metamodelle* genannt werden. Der Vorgang der Konstruktion solcher Modelle wird als *Metamodellierung* bezeichnet. So gesehen stellt die Metamodellierung einen Spezialfall der Modellierung dar. Dabei ist in den Formalismus bzw. den Sprachgebrauch der Metamodellierung insofern ein Gedanke der Objektorientierung eingeflossen, dass ein Metamodell Typen spezifiziert, welche im Modell instanziert werden. Ein Modell wird somit auch als *Instanz* des zugehörigen Metamodells bezeichnet.

Modellierungsebenen

Die Kaskade der fortlaufenden Abstraktion durch Bildung eines Metamodells lässt sich potenziell unbegrenzt fortsetzen. Die hieraus erzeugte Hierarchie von Modellen spannt dabei eine Folge von *Modellierungsebenen* auf, in die sich die jeweiligen Modelle einordnen lassen. Die Einordnung eines Modells in eine konkrete Modellierungsebene kann immer nur in Bezug auf eine bestimmte *Modellhierarchie* eindeutig bestimmt werden.

Vierschichtige Modellierungsarchitekturen

In der Praxis hat sich hinsichtlich der Hierarchie von Modellierungsebenen das Konzept der *vierschichtigen Modellierungsarchitektur* etabliert. Die einzelnen Modellierungsebenen werden dabei meist mit M0 bis M3 bezeichnet, so auch in diesem Buch. Das auf M3 angesiedelte Modell schließt die Modellhierarchie nach oben ab. Konzeptionell ist dies konform zu einem Metamodell einer Ebene M4, welches wiederum isomorph zu dem auf Ebene M3 definierten Modell ist. Unter einer isomorphen Abbildung versteht man in diesem Zusammenhang die umkehrbar eindeutige (bijektive) Abbildung der Modellelemente des einen Modells auf bedeutungsgleiche Elemente des anderen Modells. Mit der vierschichtigen Modellierungsarchitektur der OMG wird in

Abschnitt 6.6.2 ein architektonischer Ordnungsrahmen für eine Familie von Modellierungssprachen vorgestellt.

Indem es die (in der Regel unendliche) Menge gültiger Instanzen definiert, kann ein Metamodell auch als Mittel zur Beschreibung einer *Modellierungssprache* verstanden werden. Formal exakt definiert wird in der Regel lediglich die abstrakte Syntax der spezifizierten Modellierungssprache, weshalb die Begrifflichkeiten des Metamodells und der abstrakten Syntax oftmals synonym verwendet werden. Da die Semantik einer Modellierungssprache häufig informell in Form von natürlicher Sprache definiert wird, spricht man in diesem Fall auch von semi-formalen Sprachen.

Modellierungssprachen

Während die *abstrakte Syntax* lediglich die Struktur einer Modellierungssprache beschreibt, wird durch die *konkrete Syntax* ein Instrument zur textuellen oder grafischen Notation syntaktisch korrekter Modelle bereitgestellt. Zur Erläuterung soll an dieser Stelle eine Analogie zu klassischen Programmiersprachen bemüht werden. Hier wird durch die konkrete Syntax festgelegt, welche Eingaben ein Parser für diese Sprache akzeptiert. Die abstrakte Syntax spezifiziert lediglich, wie die Struktur der Sprache aussieht. Von Details, wie beispielsweise der Schreibweise von Schlüsselwörtern, wird dabei abgesehen.

Abstrakte und konkrete Syntax

Die *statische Semantik* einer Sprache legt die Wohlgeformtheitskriterien fest, welche deren Syntax nicht festlegen kann. Ohne auf die genauen theoretischen Hintergründe einzugehen, soll hier abermals ein illustrierendes Beispiel aus der Welt der klassischen Programmiersprachen bemüht werden: In einigen Sprachen existiert die Regel, dass Variablen deklariert werden müssen, bevor ihnen Werte zugewiesen werden können. Hinsichtlich der einzelnen Phasen eines Compilers würde der Parser eine diesbezügliche Regelverletzung nicht erkennen können. Erst die statische Analyse des Compilers würde fehlschlagen. Im Kontext von Modellierungssprachen wird die statische Semantik meist durch eine Reihe von Einschränkungen (englisch: *constraints*) festgelegt. Constraints beziehen sich dabei stets auf dedizierte Modellelemente. Typische Constraints sind beispielsweise die Einschränkung des Wertebereichs von Attributen eines Modellelements oder die Einschränkung der Beziehungen zwischen Modellelementen.

Statische Semantik

Neben der abstrakten und konkreten Syntax sowie der statischen Semantik muss jede Sprache auch eine *Semantik* besitzen, welche die Bedeutung der Modelle genau definiert. Im Falle der UML wird die Semantik der Modellelemente in Rahmen der UML-Superstruktur (siehe Abschnitt 6.6.2) informell durch natürlichsprachige, englische Erläute-

Semantik

Eine der wichtigsten Standard-Modellierungssprachen

rungen im Rahmen der Definition der abstrakten Syntax beschrieben. Im Falle von MDSD (siehe Abschnitt 6.2.6) spricht man auch von einer transformationellen Definition der Semantik, was bedeutet, dass die Modelle mittels Transformationen auf eine andere, wohlbekannte Sprache (oft eine 3GL-Sprache) abgebildet werden. Diese Abbildungsregeln definieren somit die Bedeutung der eingesetzten Modellierungssprache.

6.6.2 Unified Modeling Language (UML)

Die *Unified Modeling Language* (UML) entstand Ende der 90er-Jahre durch die Zusammenführung der unterschiedlichen Notationen von Rumbaugh, Booch und Jacobson. Mit dem großen Erfolg der UML fand die babylonische Sprachverwirrung bei den Notationen in der objekt-orientierten Gemeinde ein Ende (siehe Tabelle 6.6-1).

Tab. 6.6-1: Vom Notationen-Babylon zur UML.

Zeitraum	Notation	Bemerkung
2005	UML 2.0	Überarbeitetes Metamodell
2001	UML 1.4	Marktbeherrschung erreicht
1999	UML 1.3	XML Metadata Interchange (XMI)
1998	UML 1.2	OMG übernimmt Obhut
1997	UML 1.0	Object Constraint Language (OCL)
1996	Unified Modeling Language (UML) 0.9	Unified Method und OOSE
1995	Unified Method	OMT und OOD
1992	OOD und OOSE	Booch und Jacobson
1991	OMT	Rumbaugh
1987 - 1998	OMT, OOD, OOSE, OOSA und viele mehr	Notationen-Babylon

Unter der Obhut der Object Management Group (OMG) wurde die UML inzwischen zu einem der wichtigsten Standards der Software-Entwicklung. In diesem Buch wird die UML in der Version 2.0 behandelt, welche von annähernd allen Entwicklungswerkzeugen unterstützt wird. Bevor ein Blick auf die verschiedenen Diagrammarten geworfen wird, soll im Folgenden die Einbettung und Verflechtung der UML in die Modellierungsarchitektur der OMG kurz beleuchtet und in den Kontext der in Abschnitt 6.2.1 vorgestellten Konzepte gesetzt werden.

Die *Meta Object Facility* (MOF) [OMG 2008b] ist die Basis der Modellierungsarchitektur der OMG. Die MOF-Spezifikation definiert eine abstrakte Syntax sowie ein Rahmenwerk zur Konstruktion und zum Umgang mit technologienutralen Metamodellen, welche als MOF-basierte Metamodelle bezeichnet werden. Neben weiteren OMG-Standards wie beispielsweise dem *Common Warehouse Metamodel* (CWM) [OMG 2008a] kann auch das Metamodell der UML in die Menge der MOF-basierten Metamodelle eingeordnet werden. MOF basiert auf dem in Abschnitt 6.6.1 vorgestellten Konzept einer vierstufigen Modellierungsarchitektur.

Abbildung 6.6-2 illustriert die vierstufige Modellierungsarchitektur der OMG. Die linke Spalte bezeichnet die Modellebenen nach den durch die OMG standardisierten Nomenklatur. Die mittlere Spalte zählt konkrete, sich in der jeweiligen Ebene befindliche Modelle auf. Zu Illustrationszwecken wird an dieser Stelle abermals der Vergleich zu einer bekannten Spracharchitektur bemüht. Die rechte Spalte ordnet daher Begrifflichkeiten der Programmiersprache Java in die entsprechenden Ebenen ein.

Die Meta-Object Facility

Die vierstufige Modellierungsarchitektur der OMG

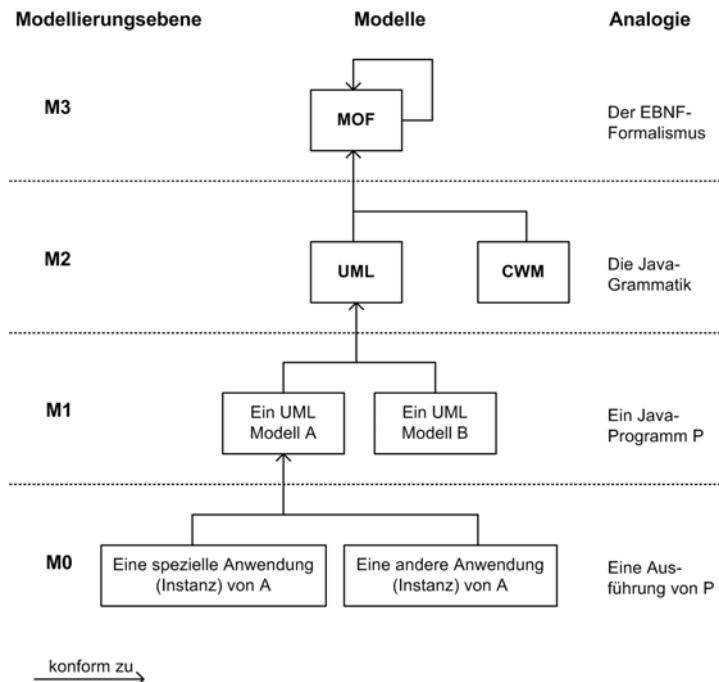


Abb. 6.6-2: Vierschichtige Modellierungsarchitektur der OMG: Modelle und Ebenen.

Modularer Aufbau der UML-2.0-Spezifikation

Insgesamt ist die Spezifikation der UML 2.0 sehr viel modularer aufgebaut, als die ihrer Vorgängerversionen. Zur Wiederverwendung bereits definierter Pakete und deren Elemente spezifiziert die UML 2.0 den *Package Merge Algorithmus* [Zito et al. 2006], eine komplexe Transformationsvorschrift zur Verschmelzung zweier Pakete. Eine zentrale Neuerung der UML 2.0 gegenüber ihren Vorgängerversionen ist die Trennung der UML-Spezifikation in Infra- und Superstruktur, was eine Wiederverwendung (mittels Package Merge) der Infrastruktur als Grundlage anderer Metamodelle ermöglicht.

Die UML-Infrastruktur

Das durch die *UML-Infrastruktur* (englisch: *UML Infrastructure Specification*) [OMG 2006a] spezifizierte Paket *Core* bildet den Sprachkern der UML. Es wurde so allgemein und wiederverwendbar konzipiert, dass es auch als Kern anderer Sprachdefinitionen dienen kann. Abbildung 6.6-3 zeigt die Verwendung des Paketes *Core* sowohl als Basis der MOF-Spezifikation, wie auch als Grundlage der UML- und der CWM-Spezifikation (englisch: *Common Warehouse Metamodel Specification*) [OMG 2008a]. Zusammen mit dem ebenfalls durch die Infrastruktur definierten Paket *Profiles* bildet es die *Infrastructure-Library* (IL). Es ist zu beachten, dass die IL in verschiedenen Modellierungsschichten der Modellierungsarchitektur genutzt werden kann. Mit der Aufnahme des Normierungsprozesses für die UML und MOF in der Version 2.0 wurde somit das Ziel verfolgt, den Kern von UML und MOF auf der Grundlage einer einheitlichen Menge von Basiskonzepten vollständig zu vereinen.

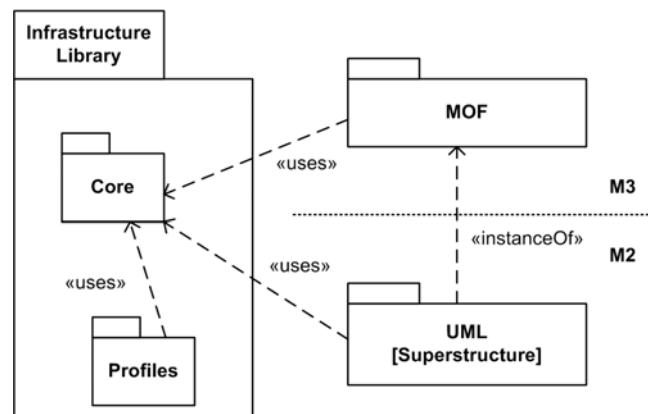


Abb. 6.6-3: Anwendung der Infrastructure-Library in MOF und der UML.

Die UML-Superstruktur

Auch die *UML-Superstruktur* (englisch: *UML Superstructure Specification*) [OMG 2005c] basiert auf dem Paket *Core* der Infrastruktur. Die in *Core* definierten Unterpakete verschmelzen mittels Package-Merge-Algorith-

mus mit den durch die Superstruktur definierten Paketen und werden anschließend weiter verfeinert, um die speziellen Modellelemente zu erhalten, welche die gemeinhin als UML-Metamodell bezeichnete Sprachdefinition ausmachen. Es soll an dieser Stelle angemerkt werden, dass die UML-Superstruktur lediglich die abstrakte Syntax der Sprache spezifiziert. Zur Notation, in Abschnitt 6.6.1 als konkrete Syntax bezeichnet, existieren sowohl grafische als auch textuelle Varianten, welche ihrerseits wiederum in separaten Spezifikationen definiert werden. Eine textuelle Repräsentationsmöglichkeit bietet der *XMI-Standard* [OMG 2007d]. Eine grafische Notationsvariante manifestiert sich in den im weiteren Verlauf dieses Abschnitts beschriebenen Diagrammarten.

Die Sprachkonstrukte der UML sind sehr generisch konzipiert. Im Zuge domänenpezifischer Sprachen (siehe Abschnitt 6.2.3) werden aber oftmals an eine bestimmte Anwendungsdomäne angepasste Modellierungssprachen benötigt, welche die Konzepte einer spezifischen Domäne möglichst präzise wiedergeben. Eine Möglichkeit zur Realisierung einer solchen Sprache besteht in der Anpassung der UML. Die beiden grundsätzlichen Möglichkeiten zur Adaption der UML an spezifische Einsatzzwecke werden im Folgenden kurz vorgestellt.

Eine Möglichkeit der Erweiterung der UML ist eine echte Erweiterung des UML-Metamodells. Zur Modellierung werden die Sprachmittel der nächsthöheren Meta-Ebene, in diesem Fall also der MOF, verwendet. Durch Instanziierung von durch die MOF spezifizierten Elementen können sowohl komplett neue Modellelemente definiert, als auch bestehende Elemente des UML-Metamodells (mittels Spezialisierung) verfeinert werden. Anzumerken ist, dass sich die Instanziierung von MOF-Elementen keineswegs nur auf die Instanziierung der MOF-Metaklasse *Class* reduziert, sondern alle zur Bildung des UML-Metamodells verwendeten Sprachmittel eingesetzt werden können. Eine derartige Erweiterung des Sprachumfangs wird aufgrund der tiefgreifenden Einwirkungen auf das Metamodell auch als *schwergewichtige Erweiterung* bezeichnet.

Mit der Definition der UML 2.0 wurde der *Stereotyp-Mechanismus* der Vorgängerversionen zu einem umfassenderen *Profilmechanismus* erweitert. Abbildung 6.6-4 zeigt, etwas vereinfacht, den relevanten Ausschnitt der UML-Superstruktur.

Anpassung der UML an spezifische Anwendungsdomänen

Erweiterung des UML-Metamodells

Profilmechanismus der UML 2.0

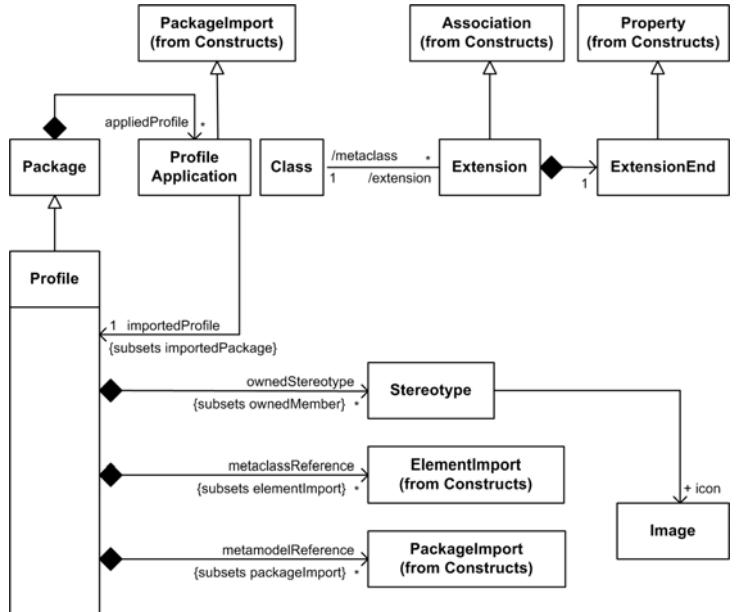


Abb. 6.6-4: Profilmechanismus der UML 2.0: Relevanter Ausschnitt der UML-Superstruktur.

Zentral ist das Konzept der *Extension*. Durch diese spezielle Assoziation wird die Metaklasse festgelegt, welche durch die im Rahmen einer Profildefinition spezifizierten Stereotype erweitert wird. Profile sind spezielle Pakete. Ein Paket kann profiliert werden, indem eine *ProfileApplication*, eine Spezialisierung von *Packagelimport*, angewendet wird. Die durch ein Profil definierten Stereotype können somit innerhalb des profilierten Pakets genutzt werden. Ein *Stereotyp* kann Attribut besitzen, welche als *Tagged Value* bezeichnet werden. Als weiteres Sprachmittel steht der Definition von Profilen lediglich die Spezifikation von Constraints (siehe Abschnitt 6.6.1) für die neu definierten Stereotype zur Verfügung. Neben der oben dargestellten schwergewichtigen Erweiterung des Metamodells erlaubt der Profilmechanismus der UML also eine Ergänzung vorhandener Modellierungskonzepte ohne einen Eingriff in das Metamodell. UML-Profilen werden daher auch als *leichtgewichtige Erweiterung* bezeichnet.

Jeder der beiden vorgestellten Erweiterungsmechanismen besitzt seine spezifischen Vor- und Nachteile, welche im Folgenden kurz skizziert werden sollen. Ziel einer Erweiterung der UML ist stets, zu dem erweiterten Metamodell konforme Modelle zu erstellen. Die funktionale Anforderung der Spezifikation von Profilen und deren Anwendung auf

konkrete Modelle ist in den meisten UML-Werkzeugen realisiert. Viele UML-Werkzeuge bieten dabei auch die Möglichkeit zur Überprüfung von durch ein angewendetes Profil spezifizierten Constraints. Ist der Zweck einer Erweiterung des UML-Metamodells lediglich die Einführung neuer Begrifflichkeiten (in Form von Stereotypen), so ist der Profilmechanismus ein geeignetes Mittel. Die Ausdrucksmächtigkeit von Profilen ist gegenüber schwergewichtigen Erweiterungen des UML-Metamodells oder der Definition eines eigenständigen, auf der MOF basierenden Metamodells jedoch sehr eingeschränkt. Viele Sachverhalte sind nur sehr umständlich und mithilfe komplexer, schwer zu durchschauender Constraints formulierbar. An dieser Stelle bietet sich der Einsatz sogenannter *Meta-Case-Werkzeuge* an, wie beispielsweise MetaEdit+ [MetaEdit 2008]. Meta-Case-Werkzeuge erlauben die Definition eigener, auf einem dedizierten Meta-Metamodell beruhender Metamodelle. Teilweise kann dabei jedem Sprachelement des erstellten Metamodells eine eigene Repräsentation zugewiesen, das heißt, die konkrete Syntax der Sprache definiert werden. Diese kann sowohl grafisch als auch textuell orientiert sein. Im ersten Fall werden den Sprachelementen beispielsweise bestimmte grafische Symbole, im zweiten Fall dedizierte Schlüsselwörter zugeordnet.

Die Leitidee der UML lautet „ein Modell, verschiedene Sichten“ und bedeutet, dass mit der UML verschiedene Aspekte eines Systems durch eine Vielzahl verschiedener Diagrammarten dargestellt werden können. In Tabelle 6.6-2 wird ein Kurzüberblick zu den Diagrammen (UML-Sichten) der UML gegeben. Der Fokus liegt dabei auf für die Architektur-Modellierung wichtigen Aspekten, wobei zwischen statischen und dynamischen Aspekten der Architektur unterschieden wird. Für einen umfassenden Überblick mit Details zu allen Notationselementen und den spezifischen UML-Begriffen sei auf [Jeckle et al. 2004 und Oestreich 2004] verwiesen.

Ein Modell, verschiedene Sichten

Tab. 6.6-2: Architektonische Bedeutung der UML-Diagramme.

Diagramm	zeigt	statisch/ dynamisch
Aktivitätsdiagramm	Schritte, die innerhalb eines Systems ablaufen, um eine bestimmte Aufgabe zu erfüllen. Unter Angabe der beteiligten Bausteine.	dynamisch
Anwendungsfall-diagramm	Anwendungsfälle eines geplanten oder existierenden Systems und den daran beteiligten Parteien.	dynamisch

Diagramm	zeigt	statisch/ dynamisch
Interaktionsübersichtsdiagramm	Wann welche Interaktion zwischen Bausteinen abläuft.	dynamisch
Klassendiagramm / Komponentendiagramm	Schnittstellen und Beziehungen von Bausteinen.	statisch
Kommunikationsdiagramm	Bausteine, die zusammenarbeiten bzw. kommunizieren.	dynamisch
Kompositionssstrukturdiagramm	Bausteine hinsichtlich ihrer Schnittstellen und Beziehungen sowie ihres Innenlebens.	statisch
Objektdiagramm	Innere Struktur eines Bausteins zu einem bestimmten Zeitpunkt zur Laufzeit.	statisch
Paketdiagramm	Logische Zusammenfassung von kohäsiven Bausteinen.	statisch
Sequenzdiagramm	Kommunikationsabläufe zwischen Bausteinen.	dynamisch
Timing-Diagramm	Zustände von Bausteinen in Abhängigkeit von der Zeit.	dynamisch
Verteilungsdiagramm	Physikalische Verteilung von Bausteinen zur Laufzeit.	statisch
Zustandsdiagramm	Zustände eines Bausteins und Ereignisse, welche diese Zustände bewirken.	dynamisch

Tabelle 6.6-3 zeigt, wie die UML verwendet werden kann, um statische und dynamische Aspekte von Architektur-Sichten (siehe Abschnitt 4.2) darzustellen. Dabei wird aufgeführt, welche Diagramme für die jeweiligen Architektur-Sichten des abstrakten Architektur-Sichtenmodells aus Abschnitt 4.2 idealerweise benutzt werden sollten.

Tab. 6.6-3: Architektur-Sichten mit UML darstellen.

Architektur-Sicht	UML-Diagramm
Anforderungssicht	<ul style="list-style-type: none"> > Aktivitätsdiagramm > Anwendungsfalldiagramm > Klassendiagramm > Paketdiagramm > Sequenzdiagramm > Zustandsdiagramm
Logische Sicht	<ul style="list-style-type: none"> > Aktivitätsdiagramm > Klassendiagramm > Komponentendiagramm

Architektur-Sicht	UML-Diagramm
Logische Sicht (Forts.)	<ul style="list-style-type: none"> > Kompositionssstrukturdiagramm > Paketdiagramm > Sequenzdiagramm > Zustandsdiagramm
Datensicht	<ul style="list-style-type: none"> > Klassendiagramm > Komponentendiagramm > Paketdiagramm
Verteilungssicht	<ul style="list-style-type: none"> > Komponentendiagramm > Paketdiagramm > Sequenzdiagramm > Verteilungsdiagramm > Zustandsdiagramm
Umsetzungssicht	<ul style="list-style-type: none"> > Klassendiagramm > Komponentendiagramm > Paketdiagramm > Sequenzdiagramm > Verteilungsdiagramm > Zustandsdiagramm

In Abbildung 6.6-5 ist ein Beispiel für ein statisches UML-Diagramm zu sehen. Es zeigt die logische Sicht (siehe Abschnitt 4.2) auf eine Mehrschichtenarchitektur, gemäß der technischen Referenzarchitektur aus Abschnitt 8.7. Dargestellt werden die wesentlichen Bausteine eines Online-Bestellsystems sowie deren Abhängigkeiten und Relationen. Zum Einsatz kommen hier die Architekturmuster *Front Controller*, *Business Delegate* und *Data Access Object* [Alur et al. 2003]. Modellelemente, welche die entsprechenden Rollen der genannten Muster realisieren, werden durch spezielle Stereotype gekennzeichnet, welche im Idealfall durch ein Profil definiert werden. In diesem Beispiel stehen die Bausteine und ihre Beziehungen im Vordergrund. Zur Notation werden stereotypisierte *Dependency-Relationships* (Abhängigkeiten), namentlich «uses» und «delegates», verwendet. Um auf die Schnittstellen einzugehen, würden typischerweise weitere Diagramme erstellt werden, welche mithilfe weiterer Notationselemente (z. B. Schnittstellenklasse oder Komponente) die Bausteine detaillierter darstellen würden. Es sollten in einem Diagramm nicht zu viele Aspekte auf einmal dargestellt werden. Damit wird erreicht, dass die Aussagekraft eines Diagramms zu bestimmten Gesichtspunkten nicht in einem Meer unterschiedlicher Aspekte verloren geht.

Beispielhafte logische Architektur-Sicht

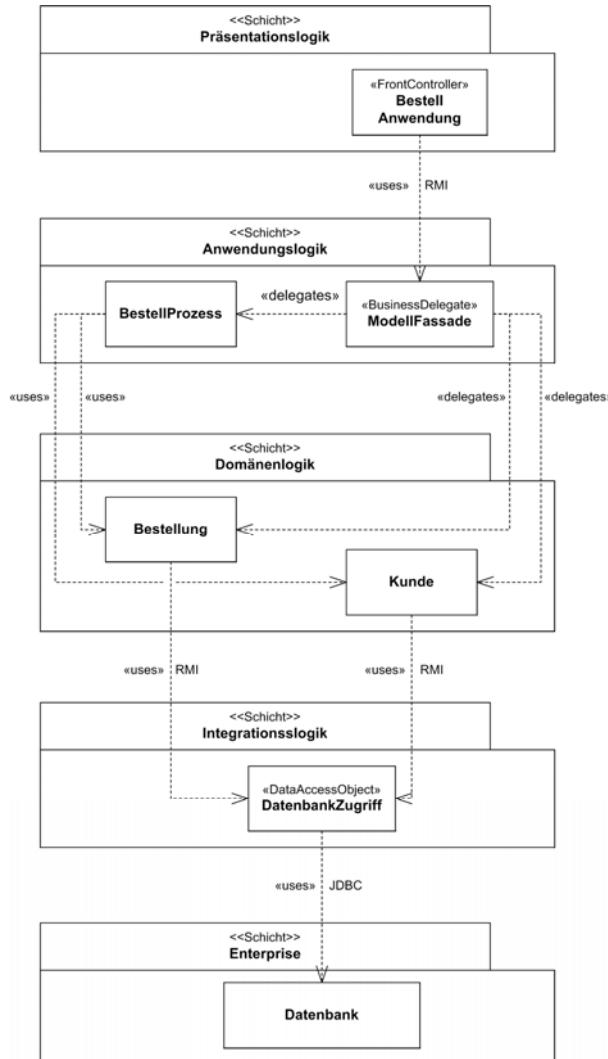


Abb. 6.6-5: Beispielhafte logische Sicht in UML.

6.6.3 Domain Specific Languages (DSL)

Domänen und domänen spezifische Sprachen

Unter einer *Domäne* versteht man in diesem Zusammenhang ein begrenztes Interessens- oder Wissensgebiet, welches sowohl fachlich, als auch technisch motiviert sein kann. Typische Domänen sind beispielsweise Eingebettete Systeme, Versicherungen, Finanzdienstleistungssysteme, aber eben auch Software-Architekturen. Ziel einer *domänenspezifischen Sprache* (englisch: *Domain Specific Language*, kurz *DSL*) ist es,

die für eine jeweilige Domäne relevanten Eigenschaften formal zu erfassen und in Form einer geeigneten Sprache abzubilden. Das Gegenstück zu DSLs bilden *General Purpose Languages* (GPL), also Sprachen, die so generisch konzipiert sind, dass sie für alle Anwendungen und Problemstellungen einsetzbar sind. Typische Beispiele für GPLs sind bekannte 3GL-Programmiersprachen wie beispielsweise Java oder auch die in Abschnitt 6.6.2 behandelte UML.

Ausgangspunkt der Konzeption einer DSL ist die Identifikation der relevanten Konzepte einer Domäne im Rahmen der *Domänenanalyse*. Als Ausgangspunkt eignet sich meist eine umfassende Begriffsbildung, beispielsweise in Form einer Ontologie. Eine Möglichkeit zur Formalisierung des durch die Domänenanalyse bereitgestellten Wissens besteht in der Definition einer Modellierungssprache, eingebettet in eine (in der Regel vierschichtige) Modellierungsarchitektur. Die Spezifikation einer DSL besteht gemäß den in Abschnitt 6.6.1 aufgeführten Betrachtungen also in der Definition eines zu einem Metametamodell (Ebene M3) konformen Metamodells (Ebene M2). Basierend auf diesem können dann konkrete Modelle definiert werden, welche Instanzen einer bestimmten Domäne darstellen.

Im Rahmen eines kurzen Exkurses sollen an dieser Stelle einige in der Praxis existierende Möglichkeiten zur Spezifikation einer DSL betrachtet werden. Zwei solcher Möglichkeiten wurden bereits in Abschnitt 6.6.2 durch die dort diskutierten Erweiterungsmöglichkeiten der UML vorgestellt. Deren praktischer Einsatz ist allerdings in hohem Maße von der Implementierung des Standards durch die am Markt verfügbaren UML-Werkzeuge abhängig, welche den Aspekt der Spracherweiterung oftmals in den Hintergrund stellen. Aufgrund der Popularität der Entwicklungsumgebung Eclipse [Eclipse 2008a] gewinnt das an die OMG-Spezifikationen der MOF und der UML angelehnte *Eclipse Modeling Framework* (EMF) [Eclipse 2008b] in der Praxis zunehmend an Interesse und Bedeutung. Das EMF stellt eine Java-basierte Realisierung einer vierschichtigen Modellierungsarchitektur dar. Als Äquivalent zu MOF kann das *Ecore-Modell* betrachtet werden, welches die Hierarchie von EMF-Modellen nach oben abschließt und somit das Metametamodell der Modellierungsarchitektur darstellt. Des Weiteren bietet EMF generische Editoren und Generatoren zur Erstellung und Verarbeitung von Modellen, was den Einsatz von EMF in der Praxis zusätzlich attraktiv gestaltet.

Der Begriff der domänenspezifischen Sprache ist sehr weit gefasst. Im MDSD-Kontext (siehe Abschnitt 6.2.6) werden die in einer bestimmten DSL formulierten Modelle meist auf den Aspekt des Inputs für Transformationswerkzeuge und Code-Generatoren reduziert. Es soll an dieser

Domänenanalyse und Formalisierung von Domänenwissen

Exkurs: Spezifikation von DSLs in der Praxis

DSLs als Mittel zur Formalisierung von Architekturwissen

Stelle darauf hingewiesen werden, dass Code-Generierung keinesfalls den einzigen Anwendungsfall einer Formalisierung domänenrelevanter Konzepte darstellt. Im Kontext der Domäne Architektur ist das formale Erfassen einer Architektur dann ein geeignetes Mittel, wenn man eine Architektur, beispielsweise im Rahmen eines großen Projektes, „standardisieren“ möchte. Der Nutzen reicht von einem präzisen Dokumentationsmittel bis hin zur Grundlage für den Entwurf einer Produktlinie einer auf dieser Architektur basierenden Systemfamilie (siehe Abschnitt 6.2.6). Eine rein informell dokumentierte Architektur reicht zu diesem Zweck nicht mehr aus, da die nötige Präzision fehlt.

Architektur-Metamodell

Ein Architektur-Metamodell definiert formal die Bausteine, aus denen eine Architektur aufgebaut ist, deren Beziehungen untereinander sowie mögliche Constraints, die festlegen, wann ein System eine gültige Architektur hat und wann nicht. Durch Instanzen eines Architektur-Metamodells wird also eine konkrete Architektur spezifiziert.

Beispielhaftes Metamodell

Abbildung 6.6-6 zeigt ein einfaches Architektur-Metamodell. Das Modell sagt unter anderem aus, dass es im Rahmen der Architektur das Konzept einer Komponente (*Component*) gibt. Komponenten haben Konfigurationsparameter (*ConfigParameter*). Außerdem können Komponenten Abhängigkeiten zu anderen Komponenten besitzen (*ComponentDependency*). Eine Business-Komponente (*BusinessComponent*) besteht aus mindestens einer (technischen) Komponente, darunter genau eine Fassade (*FacadeComponent*). Neben Komponenten als Baustein für die Anwendungslogik beschreibt das Metamodell, wie aus Komponenten Systeme (*System*) zusammengestellt werden können. Ein System besteht aus einer Reihe von Knoten (*Node*), die jeweils wiederum Komponenten-*Deployments* haben können. Ein solches *Deployment* ist stets für genau eine Komponente „verantwortlich“. Für jede Abhängigkeit einer solchen Komponente muss eine entsprechende „Verdrahtung“ (*Wire*) definiert werden. Schlussendlich definieren die *Deployments* auch Werte für die Konfigurationsparameter der Komponente (*ConfigParameterValue*).

Bezug zur UML

Das Mittel zur Beschreibung des Metamodells ist in diesem Beispiel die UML, genauer gesagt deren Notation. Die Verwendung der grafischen Notation der UML zur Spezifikation eines Metamodells wurde schon im Rahmen der Einführung der MOF verwendet. Zur exakten Formulierung der in diesem Beispiel in Form einfacher *Notes* (Anmerkungen) informell spezifizierten Constraints wird oftmals auch die *Object Constraint Language* [OMG 2006b] verwendet. Hervorgehoben werden soll an dieser Stelle nochmals der große Unterschied zwischen DSLs und der UML. Die UML ist nicht spezialisiert auf bestimmte Domänen und kann

deshalb im Gegensatz zu DSLs für ein breites Spektrum von Bereichen eingesetzt werden. Der Preis für diese Flexibilität ist ein Verlust an Präzision. Damit verbunden sind die im Vergleich zu DSLs nur eingeschränkten Analyse- und Simulationsmöglichkeiten sowie mögliche Missverständnisse hinsichtlich der Semantik eines Modells.

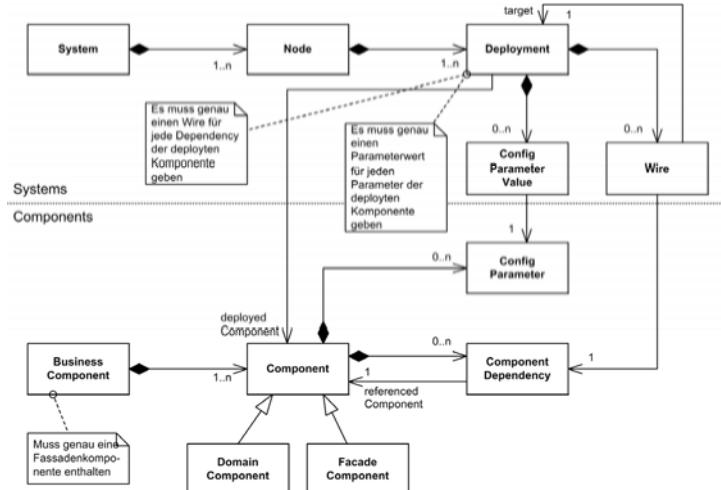


Abb. 6.6-6: Beispielhaftes Architektur-Metamodell.

Die Beschreibung der Architektur eines Systems mittels einer für die Architektur passenden DSL hat einige Vorteile. Z. B. wird die Kommunikation über das System bzw. die Architektur klarer, weil die Konzepte klar definiert sind. Letztendlich ist ein Architektur-Metamodell eine Art strukturiertes Glossar, welches einen wichtigen Bestandteil einer Architektur-Dokumentation darstellt. Der Anspruch, ein Architektur-Metamodell zu erstellen, hilft dabei, sich über die Architektur klar zu werden – die Formalisierung erzwingt dies. Der Ansatz legt auch eine technologiefreie Architektur-Definition nahe. Somit wird vermieden, sich zu früh auf Realisierungstechnologien festzulegen. Außerdem ebnet das Vorgehen den Weg hin zu Automatisierung in der Software-Entwicklung (siehe Abschnitt 6.2.6).

Konsequenzen

6.6.4 Architecture Description Languages (ADL)

Architecture Description Languages (ADLs) [Shaw und Garlan 1994] können als Spezialfall technisch motivierter, domänen spezifischer Sprachen betrachtet werden. Die Domäne ist hier *Software-Architektur*. ADLs sind also spezialisiert auf die präzise Darstellung von Architekturen, noch

Präzise Darstellung von Architektur

bevor ein System implementiert wird [Papoulias 2000]. ADLs unterstützen somit die architekturbasierte Software-Entwicklung. Mit ADLs und entsprechenden Werkzeugen lässt sich eine Architektur entwerfen, analysieren und simulieren. Insbesondere lässt sich so in einer frühen Phase feststellen, ob die Architektur den vorliegenden Anforderungen gerecht wird. Mit ADLs wird versucht, Verständlichkeit und Wiederverwendbarkeit von Architekturen zu steigern und auf diesem Gebiet bessere Analysemöglichkeiten zu erreichen [Papoulias 2000]. ADLs zeichnen sich aus durch [Opengroup 1999]:

- > Eine formale Repräsentation von Architektur mittels textuellen und grafischen Notationen auf sehr hohem Abstraktionsniveau.
- > Die Lesbarkeit durch Mensch und Maschine.
- > Analysemöglichkeiten verschiedener Architektur-Aspekte wie z. B. Vollständigkeit, Konsistenz, Performanz etc.
- > Teilweise Unterstützung automatischer Code-Generierung.

ADLs noch nicht im breiten kommerziellen Einsatz

ADLs sind noch im Entwicklungsstadium und weit von einer Standardisierung entfernt. Dies zeigt sich in folgenden Punkten [Opengroup 1999]:

- > Es ist umstritten, welche Architektur-Aspekte ADLs dokumentieren sollten und welche ADLs für bestimmte Probleme am besten geeignet sind.
- > Es gibt keine klare Abgrenzung zu anderen Mitteln wie z. B. formale Spezifikationen oder Simulationssprachen [Medvidovic und Taylor 1997].
- > Es gibt nicht *die eine* Standard-ADL, sondern eine Reihe von ADLs, die sich mit unterschiedlichen Architektur-Aspekten und Domänen beschäftigen.
- > Die verschiedenen ADLs unterscheiden sich stark in ihrem Aufbau und in der Mächtigkeit ihrer Analyse- oder Simulationswerkzeuge.
- > ADLs sind noch ein Forschungsthema an Universitäten und befinden sich nur selten im kommerziellen Einsatz. Sie sind tendenziell ausgerichtet auf akademische Zwecke ohne Bezug zur kommerziellen Nutzung.
- > Die Notationen der verschiedenen ADLs sind schwierig zu verarbeiten und werden von kommerziellen Entwicklungswerkzeugen nicht unterstützt.
- > ADLs sind meist stark vertikal spezialisiert auf die Analyse bestimmter Architektur-Aspekte.
- > Einige ADLs können direkt in Code übersetzt werden, für andere ist die Implementierung der spezifizierten Architektur offen.

Im weiteren Verlauf dieses Abschnitts werden die gemeinsamen Merkmale von ADLs behandelt. Tabelle 6.6-4 gibt in Anlehnung an [Medvidovic und Taylor 1997, ADML 2002, Chaudron 2002] eine Übersicht zu existierenden ADLs und ihren Einsatzschwerpunkten.

Tab. 6.6-4: ADL-Übersicht.

ADL	Beschreibung
ACME	Entwickelt (wie eine Reihe weiterer ADLs) an der Carnegie Mellon Universität (CMU) im Rahmen des ABLE-Projektes (englisch: <i>Architecture Based Languages and Environment</i>) [ABLE 2005]. Fokussiert auf statische Architektur-Aspekte und werkzeuggestützte Austauschbarkeit von Architekturdokumentation zwischen verschiedenen ADLs. Kann als Basis für neue Werkzeuge dienen.
ADML	Entwickelt von der open group. Basiert auf ACME und führt eine XML-basierte und damit standardisierte Form der Repräsentation ein.
Aesop	Entwickelt an der CMU. Unterstützung hierarchiebasierter Architekturstile bei der Spezifikation von Architekturen.
C2 SSDL	Entwickelt an der Universität von Kalifornien. Entwicklung von Architekturen für verteilte und dynamische Systeme.
Darwin	Ähnliche Ausrichtung wie C2, jedoch strengerer Formalismus bei der Beschreibung dynamischer Aspekte.
Koala	Entwickelt von Philips. Entwicklung von Produktlinien-Architekturen für den Embedded-Bereich.
MetaH	Entwickelt in den Honeywell Labs. Entwicklung von Architekturen für die Domäne Navigationssysteme.
Rapide	Entwickelt an der Universität von Stanford. Modellierung und Simulation des dynamischen Verhaltens von verteilten objektorientierten Systemen.
SSDL	Entwickelt am System Design Laboratory der SRI. Definition und formale Analyse architektonischer Hierarchien.
UniCon	Entwickelt an der CMU. Generierung von Konnektoren für existierende Komponenten unter Verwendung verbreiteter Interaktionsprotokolle.
Weaves	Entwicklung von Architekturen für Systeme mit Echtzeit-Verarbeitung großer Datenmengen.
Wright	Entwickelt an der CMU. Unterstützung im Bereich Konnektoren. Spezifikation und Analyse von Protokollen.

Allen ADLs ist gemeinsam, dass sie sich auf komponentenbasierte Architekturen fokussieren und sich im Kern mit den in Abbildung 6.6-7 illustrierten Architektur-Aspekten beschäftigen [Medvidovic und Rosenblum 1997, Torkler 2001]:

- *Komponenten:* Die Definition einer Komponente beinhaltet die syntaktische und semantische Spezifikation funktionaler und nicht-

Spezifikation von Komponenten und Konnektoren

funktionaler Aspekte eines Bausteins mittels Schnittstellen. Es werden sowohl die von einer Komponente exportierten, als auch die benötigten importierten Schnittstellen beschrieben. Ebenfalls beschrieben werden Daten und Datenintegrität von Komponenten. Das Verständnis einer Komponente basiert hier weitgehend auf der in Abschnitt 6.2.3 eingeführten Definition.

- > **Konnektoren:** Komponenten kommunizieren untereinander über Konnektoren, die festlegen, wie und nach welchen Regeln Komponenten miteinander interagieren. Konnektoren können verschiedene Kommunikationstechniken repräsentieren (z. B. RPC, HTTP oder Unix-Pipes).
- > **Architektur-Konfiguration:** Die Architektur-Konfiguration beschreibt die architektonische Struktur, indem sie festlegt, welche Komponenten auf welche Weise über Konnektoren verbunden werden.

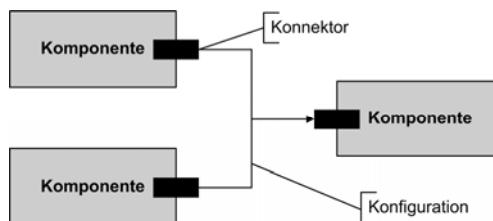


Abb. 6.6-7: ADL-Kernkonzepte.

Komponentendiagramme der UML 2.0

Im Hinblick auf die Komponentendiagramme der UML in der Version 2.0 ist ein Verschmelzen mit Konzepten der ADLs zu beobachten. Als reines Dokumentationsmittel wird daher in der Praxis meist die UML gegenüber ADLs bevorzugt. ADLs unterstützen jedoch eine Reihe statischer (zur Übersetzungszeit) und dynamischer (zur Laufzeit) Analysen. Gegenstände solcher Analysen können zum Beispiel sein:

- > Kompatibilität (Typisierung, Syntax und Verhalten) der Schnittstellen von Komponenten einer Konfiguration.
- > Erfüllungsgrad von spezifizierten Einschränkungen.
- > Performanz-, Sicherheits-, Stabilitäts- und Zuverlässigkeitssaspekte der Architektur.
- > Einhaltung von Architektur-Richtlinien.

Beispielhafte Architektur-Konfiguration

Abbildung 6.6-8 zeigt die Architektur-Konfiguration eines sehr einfachen Client-/Server-Beispiels, dargestellt in einer Lines-and-Box-Grafik. Die Komponente *Kunde* verwendet die Komponente *Datenbankzugriff* über einen RPC-Konnektor.



Abb. 6.6-8: Einfaches Client-/Server-Beispiel mit ADL.

Der folgende Beispiel-Code zeigt die formale Definition der oben dargestellten Architektur-Konfiguration in der ADL ACME [ACME 1998]. Es wird vorausgesetzt, dass die hier verwendeten Komponenten und der RPC-Konnektor bereits an anderer Stelle formal definiert wurden. Zunächst werden zwei Komponenten (*Kunde* und *Datenbankzugriff*) deklariert. Die Komponente *Kunde* (Klient) erhält einen *send-request* Port und die Komponente *Datenbankzugriff* (Server) erhält einen *receive-request* Port. Anschließend wird ein Konnektor (*rpc*) mit den Rollen *caller* und *callee* deklariert. Schließlich werden die beiden Komponenten über den Konnektor verbunden (Attachments), indem die Ports mit den entsprechenden Rollen des Konnektors assoziiert werden.

```

System BeispielSystem = {
Component kunde = {Port send-request}
Component datenBankZugriff = {Port receive-request}
Connector rpc = {Roles {caller, callee}}
Attachments : {
    kunde.send-request to rpc.caller;
    datenBankZugriff.receive-request to rpc.callee
}
}

```

ADLs definieren ein für Architekturen allgemein verwendbares Metamodell und stellen Mittel zur Modellierung einer Architektur zur Verfügung. Wie immer, wenn eine generische Lösung für ein Problem gesucht wird, ist diese auch meist entsprechend wenig spezifisch. ADLs kommen daher in der Praxis eher selten zum Einsatz. Es ist die Tendenz zu erkennen, dass ADLs immer spezifischer für bestimmte Domänen werden. Beispiele hierfür sind die EAST-ADL zur Beschreibung der Architektur von Software auf (Fahrzeug-) Steuergeräten [EAST 2004] oder das EDOC-Profil der OMG für verteilte Enterprise-Anwendungen [OMG 2005a].

Trend zur Annäherung an DSLs

6.6.5 Unified Method Architecture (UMA)

Die Entwicklung und Einführung von Software-Systemen umfasst typische Kerndisziplinen (Anforderungserhebung, Analyse, Entwurf, Implementation und Test), (Architektur-)Tätigkeiten, (Architektur-)Aktio-

Motivation

nen sowie eine Reihe querschnittlicher Aufgaben, welche in nahezu jedem Software-Projekt vollständig wieder zu finden sind (siehe Abschnitt 8.1). Eine strukturierte und planmäßige Vorgehensweise ist, insbesondere in größeren Projekten, unerlässlich.

Konzeptuelle Trennung von Methode und Entwicklungsprozess

Die Strukturierung der Vorgehensweise adressiert zwei zentrale Themenbereiche: Methoden und Entwicklungsprozesse. Als *Methode* bezeichnet man ein definiertes Verfahren zur Durchführung bestimmter Tätigkeiten. Eine Methode kann somit als „Rezept“ oder „Verfahrensanleitung“ aufgefasst werden. Während eine Methode also die durch Erfahrungen gewonnen „Best Practices“ einer spezifischen Tätigkeit aggregiert, beispielsweise des Erstellens eines Use-Case-Modells, und in Form einer Verfahrensanleitung zur Verfügung stellt, werden durch den *Entwicklungsprozess* einzelne Disziplinen und Tätigkeiten strukturiert und zueinander in Beziehung gesetzt. Insbesondere gibt der Entwicklungsprozess somit eine zeitliche Abfolge vor und nimmt Bezug auf den Lebenszyklus eines Projekts. Die konzeptuelle Trennung von Methode und Entwicklungsprozess soll anhand eines Beispiels veranschaulicht werden. Die oben genannte Beispieldurchlaufzeit des Erstellens eines Use-Case-Modells wird sowohl in einer Vielzahl von nach dem Wasserfallmodell durchgeföhrten Projekten, als auch in iterativ geföhrten Projekten Teil des Vorgehens sein. Hinsichtlich der angewendeten Methode wird sich die Tätigkeit, wenn überhaupt, nur marginal unterscheiden. Der Unterschied besteht lediglich darin, dass dies in nach dem Wasserfallmodell geföhrten Projekten typischerweise einmal während der Analysephase geschieht, während das zu erstellende Use-Case-Modell in iterativ geföhrten Projekten im Laufe verschiedener Phasen in mehreren Iterationen wiederkehrend und schrittweise verfeinert wird.

Vorgehensmodell

Hinsichtlich der Terminologie sind schon in der Literatur, insbesondere aber im Unternehmenskontext Inkonsistenzen zu finden. So wird der Entwicklungsprozess oftmals auch als Vorgehensmodell bezeichnet. In diesem Buch wird unter einem *Vorgehensmodell* das Aggregat aus Methoden und Entwicklungsprozess verstanden. Es existieren eine Reihe kommerzieller und freier Vorgehensmodelle wie beispielsweise der *Rational Unified Process* [Kruchten 2000], das *V-Modell 97* [Dröschel et al. 1998], das *V-Modell XT* [Rausch et al. 2007] oder der *Open Unified Process* [Eclipse 2008c].

Vorgehensmetamodell

Betrachtet man im Hinblick auf Vorgehensmodelle nochmals die in Abschnitt 6.6.1 eingeföhrten, grundlegenden Konzepte der Modellbildung, so kommt man an dieser Stelle zum Begriff des Vorgehensmetamodells. Ein *Vorgehensmetamodell* definiert eine allgemeingültige Ter-

minologie und Semantik zur Beschreibung von konkreten Vorgehensmodellen. Vorgehensmetamodelle existierten lange Zeit nur in den Köpfen der Prozess-Ingenieure oder in Form einer fest verdrahteten Implementierung in meist proprietären Werkzeugen. Um jedoch die Interoperabilität zwischen verschiedenen, auf Basis eines Metamodells formulierten Vorgehensmodellen gewährleisten zu können, ist ein standardisiertes Vorgehensmetamodell und damit die Einbettung in eine bestimmte Modellierungsarchitektur (siehe Abschnitt 6.6.1) von Nöten. Abbildung 6.6-9 zeigt die Einordnung der in diesem Abschnitt eingeführten Konzepte in die Modellierungsarchitektur der OMG (siehe Abschnitt 6.6.2). Die in Abbildung 6.6-9 exemplarisch gezeigten Vorgehensmetamodelle (UMA und SPEM) werden im weiteren Verlauf dieses Abschnitts genauer betrachtet.

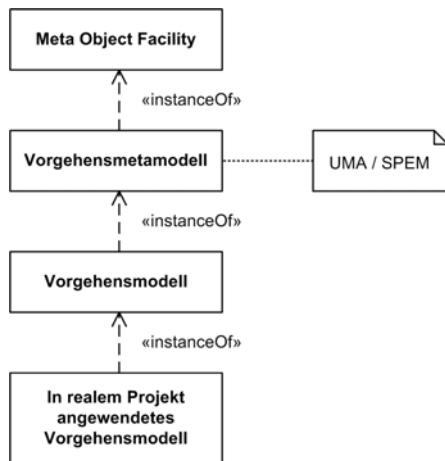


Abb. 6.6-9: Einordnung von Vorgehensmodell und Vorgehensmetamodell in die Modellierungsarchitektur der OMG.

Bereits im Jahr 2002 hat die OMG versucht, mit der Spezifikation des *Software Process Engineering Metamodel* (SPEM) [OMG 2005d] ein Vorgehensmetamodell zu standardisieren. SPEM in den Versionen 1.x wurde sowohl als eigenständiges Metamodell auf Basis der UML-Superstruktur als auch als UML-Profil spezifiziert (siehe Abschnitt 6.6.2). Zu Grunde liegende Basis war in beiden Fällen die UML in der Version 1.4. SPEM in der Version 1.x wurde jedoch kaum angenommen, sodass nur wenige, hauptsächlich kommerzielle Implementierungen hervorgebracht wurden. Da sich die UML 2 zunehmend durchgesetzt hat, wurde schnell der Wunsch erkennbar die Neuerungen der UML 2 auch für SPEM zu nutzen. Die *Unified Method Architecture* (UMA) [Gau 2006], ein durch die Firma IBM standardisiertes Vorgehensmetamodell,

Historische Entwicklung der Spezifikationen SPEM und UMA

adressierte einige der Schwächen von SPEM in den Versionen 1.x und beeinflusste, ebenso wie die Adaption an die UML 2, in entscheidender Weise die SPEM-2.0-Spezifikation [OMG 2007c]. Implementiert ist die UMA im Rahmen des frei verfügbaren *Eclipse Process Framework* (EPF) [Eclipse 2008c] sowie dem *Rational Method Composer* (RMC), einem kommerziellen Werkzeug der Firma IBM. Das EPF beziehungsweise der RMC stellen die Funktionalität zur Modellierung konkreter Vorgehensmodelle auf Basis der UMA zur Verfügung, welche anschließend in verschiedene Ausgabeformate (beispielsweise HTML) exportiert und dem Entwicklungsteam zur Verfügung gestellt werden können. Modelle werden also in diesem Kontext als Dokumentationsmittel eingesetzt.

UMA und SPEM als spezielle DSLs

Unter den bereits in Abschnitt 6.6.4 im Kontext von ADLs betrachteten Gesichtspunkten können SPEM und UMA als spezielle DSLs aufgefasst werden. Die Sprachkonstrukte dieser DSLs entstammen der fachlichen Domäne „Vorgehensmodelle zur Entwicklung von Software-Systemen“. Im weiteren Verlauf dieses Abschnitts wird die Umsetzung der domänenspezifischen Sprachkonstrukte als schwergewichtige Erweiterung des UML-Metamodells (siehe Abschnitt 6.6.2) anhand einiger ausgewählter Elemente der UMA kurz vorgestellt.

Trennung von Methode und Entwicklungsprozess in der UMA

Die grundlegende Philosophie einer Trennung von Methoden-Inhalten und Entwicklungsprozessen findet sich auch in der UMA wieder. Ein Modellelement ist somit entweder ein Baustein zur Modellierung von Methoden-Inhalten oder ein Mittel im Rahmen der Spezifikation von Prozessen oder Prozessbausteinen. Einzige Ausnahme bildet das Element *Guidance*, welches die Formulierung von Hilfestellungen und Richtlinien ermöglicht. Diese können sowohl im Methoden- als auch im Prozesskontext verwendet werden. Die Auszeichnung der Sprachkonstrukte der UMA als Methoden- oder Prozessbausteine wird über das objektorientierte Konzept der Generalisierung realisiert. So bilden *ProcessElement* beziehungsweise *MethodElement* die Wurzelemente der in den Abbildungen 6.6-10 und 6.6-11 dargestellten Vererbungshierarchien.

UMA-Sprachkonstrukte Role, Task und WorkProduct

Methoden-Inhalte ermöglichen die Modellierung von Verfahrensanweisungen zur Realisierung bestimmter Entwicklungsziele. Abbildung 6.6-12 bringt die für die Definition einer Methode zentralen Beziehungen zwischen den drei Sprachkonstrukten der Rolle (*Role*), der Aufgabe (*Task*) und des Arbeitsergebnisses (*WorkProduct*) zum Ausdruck. Diese drei Sprachkonstrukte bilden das Kernkonzept einer Methoden-Spezifikation.

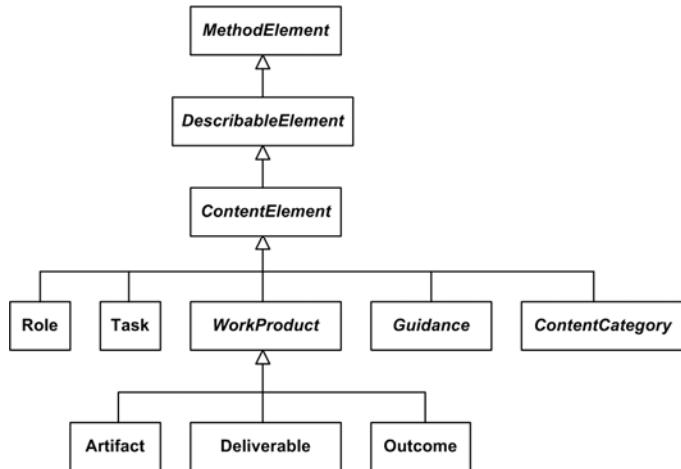


Abb. 6.6-10: UMA-Sprachkonstrukte zur Modellierung von Methoden-Inhalten (Ausschnitt).

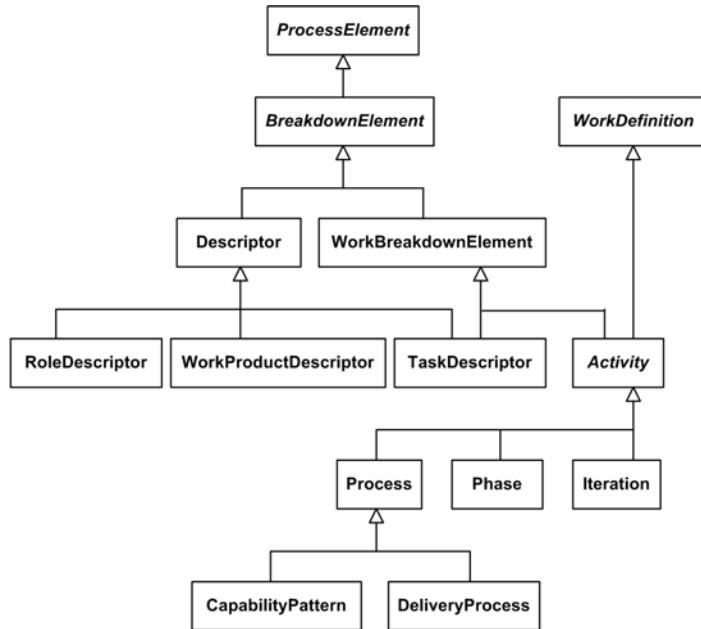


Abb. 6.6-11: UMA-Sprachkonstrukte zur Modellierung von Entwicklungsprozessen (Ausschnitt).

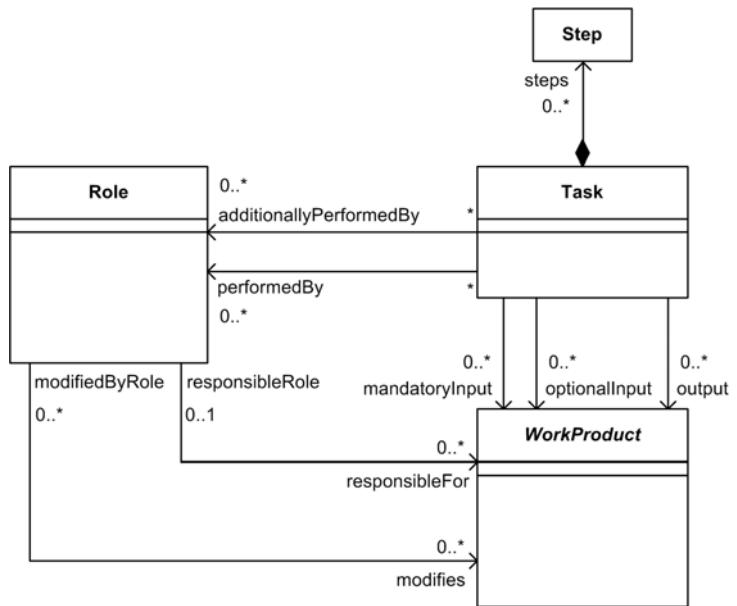


Abb. 6.6-12: Beziehungen zwischen den UMA-Sprachkonstrukten Role, Task und WorkProduct.

Eine Aufgabe (*Task*) definiert von bestimmten Rollen (*Role*) zu leistenden Arbeitsaufwände. Eine Aufgabe besitzt eine ausführende Rolle (*performedBy*) sowie beliebig viele zusätzliche bearbeitende Rollen (*additionallyPerformedBy*). Einer Aufgabe können Eingabe- und Ausgabe-Arbeitsergebnisse (*WorkProduct*) zugeordnet werden. Die zugeordneten Eingabeergebnisse werden als optional (*optionalInput*) oder notwendig (*mandatoryInput*) klassifiziert. Jede Aufgabe hat ein klar definiertes Ziel und stellt eine Schritt-für-Schritt-Anweisung in Form aller zur Erreichung des Ziels benötigten Arbeitsschritte (*Step*) bereit. Ein *WorkProduct* stellt eine Abstraktion jener Elemente dar, welche von Aufgaben erzeugt, benötigt oder modifiziert werden. Da Aufgaben von Projektteilnehmern einer bestimmten Rolle (ein Projektteilnehmer kann selbstverständlich mehrere Rollen einnehmen) bearbeitet werden, werden Arbeitsergebnisse somit von Rollen zur Ausführung ihrer Aufgaben genutzt beziehungsweise im Zuge einer Aufgabe erstellt. Darüber hinaus ordnet die UMA jedem Arbeitsergebnis eine verantwortliche Rolle zu.

Hierarchische Gliederung von Projektaktivitäten

Durch Aktivitäten und Prozess-Bausteine lassen sich Methoden-Inhalte strukturieren und zueinander in Beziehung setzen. Aktivitäten und Prozess-Bausteine legen fest, wann die als Methoden-Inhalte definierten Basis-Aufgaben im Rahmen eines projektspezifischen Entwicklungsprozesses durchzuführen sind. Die hierarchische Gliederung aller Arbeits-

pakete und Teilaufgaben eines Projekts wird im Rahmen der UMA als *WorkBreakdownStructure* (WBS) bezeichnet. Im Kontext des Projektmanagements wird die WBS teilweise auch als Projektstrukturplan (PSP) bezeichnet. Abbildung 6.6-13 zeigt den Kern des hinsichtlich der WBS relevanten Ausschnitts der UMA-Spezifikation. Zu sehen ist die Möglichkeit einer hierarchischen Gliederung von Aktivitäten (*Activity*) und in der Abbildung nicht dargestellten Subklassen der als abstrakt definierten Klasse *BreakdownElement*.

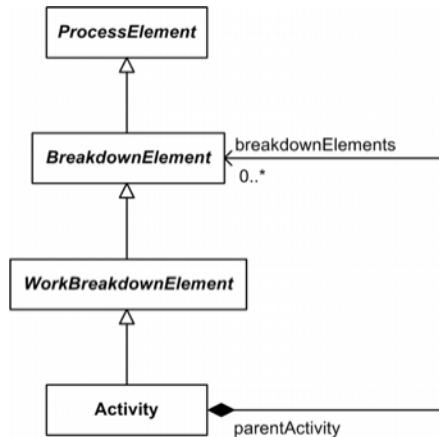


Abb. 6.6-13: Hierarchische Gliederung von Projektaktivitäten in der UMA.

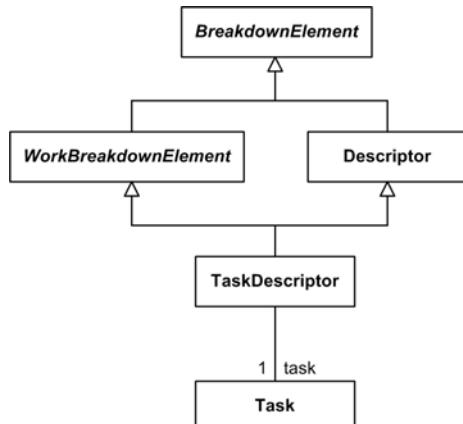


Abb. 6.6-14: Integration von Methoden-Inhalten am Beispiel Task.

Methoden-Inhalte werden über Deskriptoren (*Descriptor*) in die WBS integriert. Abbildung 6.6-14 zeigt dies exemplarisch am Beispiel des Methoden-Elements *Task*. Die durch das Konzept der Deskriptoren

eingeführte Indirektion ermöglicht es, Methoden-Inhalte individuell an spezielle Aktivitäten anzupassen, beispielsweise über die Selektion der in einem bestimmten Kontext durchzuführenden Arbeitsschritte (*Step*).

6.6.6 Zusammenfassung

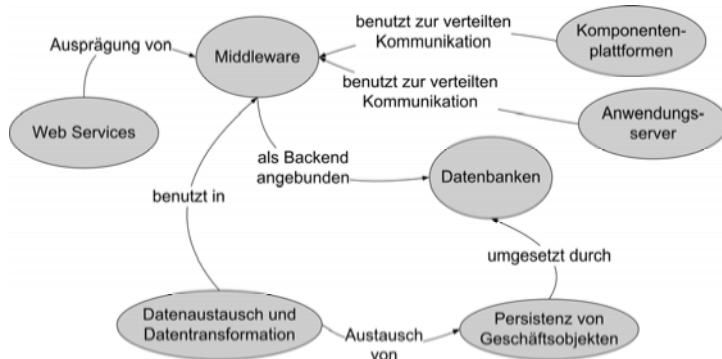
Zusammenfassung: Architektur- Modellierungsmittel

- > Eine Modellierungssprache ist das Aggregat aus konkreter Syntax, abstrakter Syntax sowie statischer und dynamischer Semantik. Die abstrakte Syntax wird auch als Metamodell bezeichnet. Die Begriffe Wohlgeformtheitskriterien und Constraints werden auch als Synonyme für statische Semantik gebraucht.
- > Modelle werden in einer konkreten Syntax notiert, erhalten Bedeutung durch die Semantik der Modellierungssprache, respektieren die definierten Wohlgeformtheitskriterien und sind konform zu einem dedizierten Metamodell.
- > Ein Metamodell ist ein spezielles Modell mit dem Zweck, die abstrakte Syntax, das heißt Modellelemente und deren Beziehungen, der Menge aller zu diesem Metamodell konformer Modelle zu definieren. Modelle werden dann auch als Instanzen des zugehörigen Metamodells bezeichnet.
- > Die Kaskade der fortlaufenden Abstraktion durch Bildung eines Metamodells spannt eine Hierarchie von Modellierungsebenen auf und wird als Modellierungsarchitektur bezeichnet.
- > In der Software-Entwicklung haben sich vierschichtige Modellierungsarchitekturen etabliert. Prominentester Vertreter ist die Modellierungsarchitektur der OMG mit der MOF als oberstem Metamodell, in diesem Fall auch als Meta-Metamodell bezeichnet.
- > Die UML ist eine spezielle Modellierungssprache. Die UML-Infrastruktur bildet den Kern der MOF und der UML-Superstruktur. Durch die UML-Superstruktur wird das UML-Metamodell spezifiziert. UML-Diagramme sind eine Möglichkeit der Notation von UML-Modellen.
- > Der Sprachumfang der UML lässt sich durch den Profilmechanismus der UML oder eine echte Metamodellerweiterung an spezifische Bedürfnisse anpassen und erweitern.
- > Domänenpezifische Sprachen (DSLs) sind, wie die UML, spezielle Modellierungssprachen. Durch eine DSL lassen sich die relevanten Konzepte einer spezifischen, fachlich oder technisch motivierten, Domäne präzise beschreiben.
- > Architecture Description Languages (ADLs) sind spezielle DSLs und dienen der präzisen Beschreibung der Architektur von Anwendungen.

- > Die Unified Method Architecture (UMA) ist eine spezielle DSL und dient der Spezifikation von Vorgehensmodellen, welche in diesem Buch als das Aggregat aus Methodik und Entwicklungsprozess betrachtet werden.

6.7 Architekturelle Technologien

In diesem Abschnitt werden einige Kategorien von Technologien behandelt, die in modernen Software-Architekturen zum Einsatz kommen und deshalb heutzutage in den „Werkzeugkasten“ eines Software-Architekten gehören. Im Detail werden Kommunikations-Middleware-Systeme, Datenbanken und Persistenz von Geschäftsobjekten, Datenaustausch und Datentransformation mit XML, Web-Anwendungsserver, Komponentenplattformen und Web Services behandelt. Abbildung 6.7-1 gibt einen Überblick über die behandelten Technologien.



Überblick

Abb. 6.7-1: Überblick der Technologien.

Die genannten Technologien stellen alle eine „generelle“ Infrastruktur in vielen Software-Architekturen bereit. Natürlich ist dies trotzdem nur eine kleine Auswahl von Technologien: Es existieren ferner viele grundlegende Technologien, wie Compiler oder virtuelle Maschinen, und auch viele spezifische Technologien, wie Content-Management-Systeme oder Enterprise-Resource-Planning-Systeme (ERP), die auch von großer Bedeutung für viele Architekturen sind, aber den Rahmen dieses Abschnitts sprengen würden.

6.7.1 Middleware-Systeme

Bereits im Abschnitt 6.4 wurde die Middleware als Basisarchitektur angesprochen. Im Folgenden sollen einige wichtige Middleware-Systeme beispielhaft genauer beleuchtet werden. Dies sind im Einzelnen: Transaktionsmonitore, RPC- und OO-RPC-Systeme, Message-oriented Middleware. Darüber hinaus wird ein kurzer Überblick über andere Systeme gegeben. Später werden noch Web Services behandelt, die ein Sonderfall sind, da sie zwar die Middleware-Basisarchitektur realisieren, aber auch die weiter gehende SOA-Basisarchitektur explizit unterstützen.

6.7.1.1 Transaktionsmonitore

Transaktionsmonitore: Historie

Transaktionsmonitore (englisch: *transaction processing monitors*, Abkürzung: TP-Monitore) sind eine der ältesten Formen einer Middleware. Sie stellen eine Infrastruktur zur Verfügung, um verteilte Transaktionen zu entwickeln, ablaufen zu lassen und zu kontrollieren.

Transaktionsmonitore sind in der Lage, eine Vielzahl von Anfragen von Klienten an Server oder Datenbanken effizient abzubilden. Viele Transaktionsmonitore unterstützen eine Reihe von Kommunikationsstilen, wie RPC, Publish/Subscribe und Nachrichtenwarteschlagen.

ACID-Eigenschaften

Das Konzept der Transaktion wurde im Datenbankumfeld entwickelt. Transaktionen sollen üblicherweise eine Reihe von Eigenschaften haben (sogenannte ACID-Eigenschaften):

- > *Atomarität* (englisch: *atomicity*): Eine Transaktion wird als eine unteilbare Einheit behandelt und wird entweder komplett oder gar nicht abgearbeitet.
- > *Konsistenz* (englisch: *consistency*): Wenn eine Transaktion zu Ende geht, muss sich das System in einem konsistenten Zustand befinden.
- > *Isolation* (englisch: *isolation*): Das Verhalten einer Transaktion darf nicht durch andere Transaktionen beeinflusst werden.
- > *Dauerhaftigkeit* (englisch: *durability*): Änderungen sind nach dem kompletten Durchführen einer Transaktion permanent bzw. persistent -überleben also auch einen Crash des Systems.

Two-Phase Commit

Eine verteilte Transaktion umfasst mehr als eine verteilte Ressource. RPC behandelt alle Aufrufe, als ob sie unabhängig voneinander wären. Im Gegensatz dazu erlauben Transaktionsmonitore dem Benutzer, eine Reihe von Aufrufen in einer Transaktion zusammenzufassen. Verteilte

Transaktionen können durch ein Two-Phase Commit Protocol (2PC, siehe auch [Gray 1978]) realisiert werden. Dieses Protokoll garantiert die ACID-Eigenschaften für eine Transaktion und unterstützt die verteilte Synchronisation von mehreren Transaktions-Ressourcen.

Einer der ersten Transaktionsmonitore war das Customer and Control System (CICS) von IBM [IBM 2003], das in den späten 1960er Jahren entwickelt wurde und noch immer im Einsatz ist. Andere bekannte kommerzielle Transaktionsmonitore sind Tuxedo von BEA [Bea 2003], Encina von Transarc [Transarc 2000] und MTS von Microsoft [Microsoft 2003].

6.7.1.2 RPC- und OO-RPC-Middleware

RPC- und OO-RPC-Middleware-Systeme benutzen den oben beschriebenen RPC-Verteilungsstil, um Prozeduren bzw. Methoden (in der Folge einheitlich als „Operationen“ bezeichnet) verteilt aufzurufen. Verteilte Aufrufe sollen so weit wie möglich – jedoch nicht weiter – aussehen wie lokale Operationsaufrufe. Intern sind RPC- und OO-RPC-Systeme sehr ähnlich realisiert. Der Hauptunterschied ist, dass OO-RPC zusätzlich objektorientierte Abstraktionen, wie insbesondere die Objektidentität, aber auch Klassen- und Vererbungsbeziehungen unterstützen.

RPC implementiert das Client-/Server-Modell wie folgt: Klienten rufen Operationen auf. Server akzeptieren Operationsaufrufe. Der Server stellt eine Menge an Operationen zur Verfügung, die Klienten verteilt aufrufen können.

Aus Klientensicht sehen RPC-Operationen identisch zu lokalen Operationen aus: Sie haben auch einen Operationsnamen, Parameter und einen Return-Typ. Ein wesentlicher Unterschied ist, dass es zu zusätzlichen Fehlermeldungen kommen kann, zum Beispiel weil das Netzwerk ausfällt oder der Server eine aufgerufene Operation nicht implementiert hat. Diese Fehler müssen dem Klienten mitgeteilt werden, das heißt, er muss in der Lage sein, sie zu behandeln.

Typischerweise wird der Klientenprozess so lange geblockt, bis die Antwort auf den Operationsaufruf vom Server zurückgeschickt wurde. Diese synchrone Art des RPC ist der Standardfall in den meisten RPC-Systemen. Einige RPC-Systeme unterstützen überdies asynchrones RPC. Bei asynchronen RPC blockt der Klient nicht, sondern setzt seine Arbeit nach der Anfrage unmittelbar fort. Üblicherweise gibt es verschiedene

**Transaktionsmonitore:
Produkte**

RPC/OO-RPC-Systeme

**Synchrones und
asynchrones RPC**

Arten von asynchronen Operationen – unter anderem solche, die ein Resultat zurücksenden, und solche, die kein Resultat senden (sogenannte Oneway-Operationen).

Prozedurale RPC-Systeme

Frühe, populäre RPC-Systeme sind das Distributed Computing Environment (DCE) [OSF 1991] und Sun RPC [Sun 1988]. Diese implementieren bereits den typischen einfachen Weg, RPC-Operationen aufzurufen: Der Server registriert eine Prozedur als einen sogenannten Endpoint in der Server-Anwendung und registriert diesen Service in einem Directory Server – der eventuell auf einem anderen Rechner läuft. Nun kann der Client diesen Service mittels des Directory Servers auffinden. Der Client nutzt in der Folge den Endpoint, um die verteilte Prozedur tatsächlich aufzurufen.

OO-RPC-Systeme

DCE ist primär als prozedurales RPC entstanden. Trotzdem verfügt es auch über eine Erweiterung, um verteilte Objekte zu unterstützen. Heute gibt es aber auch viele OO-RPC-Middleware-Systeme, die speziell für diesen Zweck entworfen wurden. Beispiele sind

- > Common Object Request Broker Architecture (CORBA) [OMG 2004c],
- > Microsoft's .NET Remoting [Microsoft 2004a],
- > diverse Web Service Frameworks (siehe Abschnitt zu Web Services unten),
- > Microsoft's DCOM [Grimes 1997] und
- > Sun's Java RMI [Grosso 2001].

Für ein tieferes Verständnis dieser Art von Middleware-Systemen sei das Buch *Remoting Patterns* [Völter et al. 2004] empfohlen, das die wesentlichen Entwurfs- und Architekturmuster von RPC- und OO-RPC-Middleware-Systemen diskutiert.

6.7.1.3 Message-oriented Middleware

Nachrichten und Nachrichtenwarteschlangen

Message-oriented-Middleware-Systeme (MOM) bedienen sich der Nachrichtenmetapher, um asynchrone, verteilte Kommunikation zu realisieren. Sowohl Client als auch Server senden oder empfangen Anfragen, Antworten oder andere Nachrichtentypen nicht direkt, sondern platzieren diese in Nachrichtenwarteschlangen bzw. erwarten diese aus Nachrichtenwarteschlangen. Dies bedeutet, dass Klienten nicht blockieren, sondern im Standardfall unmittelbar nach dem Senden einer Nachricht ihre Arbeit fortsetzen.

Resultate werden entweder durch

- > einen *Callback* (deutsch: *Rückruf*, eine asynchron als Ereignis aufgerufene Operation) zurückgeliefert oder
- > vom Client per Nachfrage an die Nachrichtenwarteschlange erlangt (sogenanntes „*Polling*“).

Asynchrone Resultate

Beide Varianten sind asynchron. Daher ist es notwendig, dass der Client in der Lage ist, eine spezifische Antwort einer vorangegangenen Anfrage zuzuordnen, denn die Antworten kommen nicht unbedingt in derselben Reihenfolge an, wie die Anfragen verschickt wurden. Die Zuordnung von Antworten zu Anfragen geschieht üblicherweise durch einen eindeutigen Identifizierer, der mit der Anfrage geschickt wird und mit der Antwort zurückkommt (ein sogenannter „Correlation Identifier“ [Hohpe und Woolf 2003]).

Typischerweise unterstützen MOM-Systeme mehrere Nachrichtenkanäle als Verbindung zwischen spezifischen Sendern und spezifischen Empfängern. Jeder dieser Kanäle hat seine eigene Sende- und Empfangsnachrichtenwarteschlange. Klienten- und Server-Anwendungen interagieren typischerweise nicht direkt mit Nachrichtenwarteschlängen oder Nachrichtenkanälen, sondern nutzen sogenannte Endpoints als Abstraktionen, welche alle Interaktion mit dem MOM-System übernehmen.

MOM-Systeme zeichnen sich überdies durch einige Eigenschaften aus. Diese sind typischerweise:

- > Nachrichten werden zuverlässig übermittelt. Das heißt, der temporale Ausfall von Systemressourcen, wie dem Netzwerk oder dem Server, kann toleriert werden.
- > Die Reihenfolge der Auslieferung und des Empfangs von Nachrichten kann garantiert werden.
- > Wenn eine Nachricht auch nach langer Zeit nicht ausgeliefert werden kann, kann sie durch das Setzen einer Expiration-Zeit (deutsch: „*Ablauf*-Zeit) automatisch aus dem MOM-System entfernt werden.
- > Das MOM-System erkennt und behandelt fälschlicherweise doppelt ausgesendete Nachrichten automatisch.

Charakteristische Funktionalitäten und Eigenschaften von MOM

Die MOM-Konzepte sind in einer Reihe von Middleware-Systemen implementiert. Beispiele sind IBM WebSphere MQ (früher MQ Series) [IBM 2004], JMS [Sun 2004b], Microsoft MSMQ [Microsoft 2004b] und Tibco [Tibco 2004].

MOM: Produkte

Für ein tieferes Verständnis von MOM-Systemen sei das Buch *Enterprise Integration Patterns* [Hohpe und Woolf 2003] empfohlen.

6.7.1.4 Weitere Middleware-Systeme

Die drei in etwas mehr Detail besprochenen Arten von Middleware-Systemen, Transaktionsmonitore, RPC-Systeme und MOM-Systeme, sind heute relativ weitverbreitet und kommerziell akzeptiert. Jedoch existiert eine Vielzahl anderer Middleware-Systeme, die auch häufig eingesetzt werden oder noch in der Entwicklung begriffen sind. Hier soll ein knapper Überblick über einige solche Systeme gegeben werden (siehe das Buch *Remoting Patterns* [Völter et al. 2004] für eine längere Diskussion):

- > *Peer-to-Peer-Systeme* (P2P) unterscheiden sich von anderen verteilten Architekturen, da sie nicht auf dem Client-/Server- oder n-Tier-Architektur-Stil aufbauen. Im Gegensatz dazu basieren sie auf einem Netzwerk von gleichberechtigten Peers, die untereinander kommunizieren und sich gegenseitig koordinieren. Viele P2P-Systeme werden intern mit verteilten Objekten realisiert. Beispiele für P2P-Systeme und Projekte sind: Napster [Roxio 2003], SETI@home [UPNP 2004] und JXTA [Sun 2003].
- > Eng verwandt mit P2P-Systemen sind *spontane Netzwerke*, die es erlauben, dass beliebige Services im Netzwerk zu jedem beliebigen Zeitpunkt angeboten und auch wieder entfernt werden können. Man kann spontane Netzwerke als eine Infrastruktur für die Implementierung eines P2P-Netzwerkes nutzen. Ein Beispiel für ein spontanes Netzwerk ist Jini [Jini 2003].
- > *Grid Computing* [Foster et al. 2001] hat den Zweck, verteilte Ressourcen, wie Rechnerleistung, Information oder Speicher, gemeinsam zu nutzen. Ein Netzwerk aus miteinander verbundenen Rechnern wird zu einem System zusammengeschlossen.
- > *Mobiler Code* [Fugetta et al. 1998] unterscheidet sich dadurch von anderen Middleware-Ansätzen, dass nicht nur Anfragen und Resultate verschickt werden – also Daten – sondern auch Code. Somit kann Code vom Klienten zur Verfügung gestellt und im lokalen Kontext des Servers ausgeführt werden. Dies hat z. B. den Vorteil, dass Klienten die Berechnungsvorschriften beeinflussen können, aber trotzdem die Daten für eine Berechnung nicht komplett übers Netz geschickt werden müssen.

6.7.2 Datenbanken und Persistenz von Geschäftsobjekten

In vielen Architekturen bilden Datenbanken einen wichtigen Teil der Software-Architektur. Hier soll keine Einführung in Datenbanken, sondern vielmehr ein Überblick über die architektonischen Anforderungen der persistenten Datenhaltung gegeben werden. Es sollen also die typischen Probleme und technologischen Lösungsmöglichkeiten besprochen werden, wenn man die Datenbanktechnologie nutzt, um Geschäftsobjekte persistent zu halten. *Persistenz* meint allgemein, dass man Daten von flüchtigem Speicher, wie dem Hauptspeicher (RAM), auf beständige Speichermedien, wie Festplatten oder optischen Speichermedien, bringt. Dies geschieht mit dem Ziel, die Geschäftsdaten „sicher“ zu speichern und, wenn nötig, bei Änderungen entsprechend zu aktualisieren.

Ein weiteres wichtiges Problem, das zu persistenter Datenhaltung führt, ist, dass der Speicherbedarf vieler Programme deutlich den zur Verfügung stehenden Hauptspeicher übersteigt. Generell lässt sich sagen, dass Speicher umso billiger wird, je langsamer er ist. Daher versucht man, soweit möglich, teuren Hauptspeicher durch billigere Speicher, wie Festplatten, optische Speichermedien oder Bandspeicher, zu ersetzen. Gewöhnlich basiert die persistente Speicherung von Geschäftsobjekten auf einer Datenbank, also der Speicherung auf Festplatten. Jedoch gibt es auch andere Formen der „sicheren“ Speicherung von Geschäftsdaten. Beispielsweise werden bei sehr großen Datenmengen oft optische Speichermedien mit sogenannten Jukeboxen (Plattenwechselautomaten für optische Speichermedien) verwendet. Diese erlauben heute einen Zugriff auf nahezu unbegrenzte Datenmengen, sind aber deutlich langsamer als Festplatten.

Neben der grundsätzlichen Persistenzanforderung an eine Datenbank – also ihrer Hauptfunktionalität – gibt es noch eine Reihe weiterer Anforderungen, die man an die Datenbank bzw. das Datenbankmanagementsystem stellen kann:

- In vielen Systemen kann eine Vielzahl von Anfragen zur gleichen Zeit an eine Datenbank gestellt werden. Daher ist eine gute *Performance* und *Skalierbarkeit* von Nöten.
- Da die Datenbank ein zentraler Bestandteil der IT-Architektur ist, ist auch die hohe *Verfügbarkeit* ein wichtiger Punkt.
- Die Datenbank sollte *Transaktionen* unterstützen – hier unterscheidet man zwischen kurz laufenden Transaktionen, wie einer Folge von Anfragen, und lang laufenden Transaktionen, wie zum Beispiel einem Workflow mit menschlichen Interaktionen. Bei lang laufen-

Persistenzanforderung

Weitere Anforderungen

den Transaktionen können mitunter die oben genannten ACID-Eigenschaften schwer einzuhalten sein. Daher müssen diese gesondert unterstützt werden, also in der Regel nicht nur durch die Datenbank, sondern auch durch Anwendungslogik.

- > Eine Datenbank soll *Sicherheit* der Daten unterstützen und verschiedene Sicherheitsrechte managen können, z. B. mit einem Benutzer- und/oder Rollenkonzept.
- > Die *Einfachheit* des Zugriffs auf die Daten ist wichtig, um die Datenmodelle verstehen zu können und leicht auf die Daten zugreifen zu können. Dies heißt insbesondere auch, dass die Datenmodelle und Zugriffsstrukturen einfach auf die Anwendungsarchitektur abbildbar sein sollen.
- > Die Datenbank sollte eine Wartungsschnittstelle für die *Administration* und die *Kontrolle* der Datenbank bereitstellen.

Strukturbruch zwischen Datenbank und objekt-orientierter Anwendung

Ein weiteres Problem ist, dass in vielen Projekten heutzutage objektorientiert entwickelt wird, aber die vorherrschenden Datenbanken auf relationalen Datenmodellen basieren. Mit anderen Worten, man muss sich entscheiden, ob man das übliche relationale Datenbankmanagementsystem (RDBMS) benutzen will oder aber aufgrund der objektorientierten Natur der Anwendung ein objektorientiertes Datenbankmanagementsystem (OODBMS) einsetzen will. Man beachte, dass sich die gleiche Problematik nicht nur bei der Objektorientierung, sondern auch bei anderen Programmierparadigmen stellt, wie z. B. prozeduraler Programmierung, logischer Programmierung oder Aspektorientierung. Bei jedem Programmierparadigma muss man sich Gedanken machen, wie man die Anwendungsdaten sinnvoll auf die Datenbank abbildet. Folgende beispielhaft angedeuteten Möglichkeiten gibt es immer:

- > Die Datenbank ist von sich aus in der Lage, die Datenabstraktionen und Beziehungen des Programmierparadigmas abzubilden.
- > Die Datenbank unterstützt andere Abstraktionen als das Programmierparadigma. Diese Situation bezeichnet man als einen *Strukturbruch*, auf den man reagieren muss. Das heißt, man muss sich überlegen, wie man das Anwendungsmodell auf das Datenbankmodell abbilden kann.

Speicherung objektorientierter Geschäftsobjekte

Betrachten wir nun die Lösungsmöglichkeiten für das Beispiel der Persistenz objektorientierter Geschäftsobjekte genauer. Wenn man objektorientierte Geschäftsobjekte speichert, möchte man generell gerne deren Eigenschaften, wie Objektbeziehungen, Objektidentität, Klassenbeziehungen, Vererbung, Polymorphismus etc., auf die Speicherung in der Datenbank abbilden. Ein OODBMS bietet hier den Vorteil, dass es

schon objektorientierte Eigenschaften abbildet, also die Abbildung der objektorientierten Modelle in der Programmiersprache sehr einfach ist. Der gravierende Nachteil von OODBMS-Systemen ist jedoch, dass diese deutlich seltener eingesetzt werden als relationale Datenbanksysteme. Da Datenbanken meistens äußerst wichtig für Unternehmen sind, wird in vielen Projekten das Risiko, ein OODBMS-System auszuwählen, als zu groß erachtet.

Bei der Speicherung von Geschäftsobjekten in relationalen Datenbanken hingegen kommt es zu dem oben erwähnten Strukturbruch: Relationen in Tabellen und Beziehungen werden über Fremdschlüssel abgebildet. Dieses Modell muss auf das objektorientierte Modell mit seinen Beziehungen, Hierarchien und Identitäten abgebildet werden. Diese Abbildung ist nicht eindeutig und daher gibt es eine ganze Reihe von Ansätzen für das Object-Relational-Mapping [Keller 1997].

Ein einfacher Ansatz ist, einfach den SQL-Code für den Zugriff auf die Datenbank in der Anwendungslogik einzubetten. Dies hat aber gravierende Nachteile: Schon bei einfachen Datenänderungen muss der Anwendungslogik-Code geändert werden. Der Aspekt „Persistenz“ ist über das gesamte Programm verteilt; das heißt, zentrale Änderungen werden schwierig. Daher ist es generell empfehlenswert, zumindest eigene „Datenklassen“ einzuführen, deren Aufgabe es ist, Datenbankzugriffe zu behandeln, und die unabhängig von der Anwendungslogik sind.

Oft jedoch sind viele Aufgaben beim Datenbankzugriff wiederkehrend. Um dieser Situation zu begegnen, führt man häufig eine eigene Datenbankzugriffsschicht ein, die alle Anfragen an eine Datenbank kapselt und nicht umgangen werden kann. Hier findet also das Object-Relational-Mapping statt. Oftmals besteht die Datenbankzugriffsschicht selbst aus zwei Schichten [Keller und Coldewey 1998] (siehe Abbildung 6.7-2): Eine logische Zugriffsschicht stellt eine stabile Schnittstelle für die Anwendungsschicht zur Verfügung. Eine physikalische Zugriffs- schicht darunter stellt den tatsächlichen Zugriff auf die Datenbank her und kann z. B. aus Performanzgründen oder bei Versionsänderungen der Datenbank modifiziert werden.

In vielen Sprachen stehen Standardbibliotheken für den Datenbankzugriff zur Verfügung. In Java zum Beispiel gibt es JDBC (Java Database Connectivity) – eine Bibliothek für den Zugriff auf relationale Datenbanken. Generell bietet JDBC Funktionen für den Verbindungsaufbau zu einer Datenbank, die Benutzung von SQL-Anweisungen für sogenannte CRUD-Operationen (create, read, update, delete) und die Auswertung

Object-Relational-Mapping

Datenbankzugriff

Standardbibliotheken für den Datenbankzugriff

der Ergebnisse an. Als Vorteil bieten solche Standardbibliotheken (meist) eine Unabhängigkeit von der Datenbankimplementierung. Allerdings ist die Datenbankprogrammierung hier immer noch auf sehr niedrigem Niveau: Entwickler müssen genaue Kenntnisse von SQL haben und alle technischen Details der Persistenz müssen vom Entwickler gelöst werden.

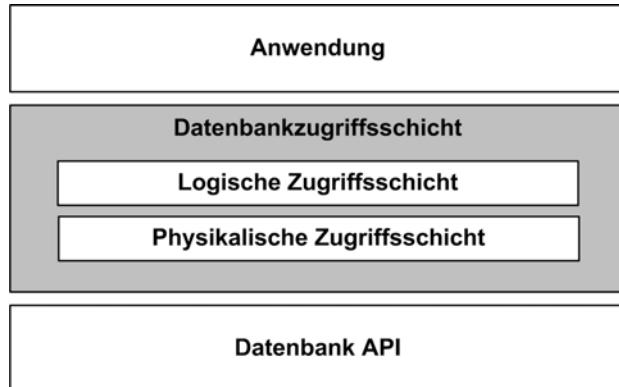


Abb. 6.7-2: Datenbankzugriffsschicht [Keller und Coldewey 1998].

Object-Relational-Mapping

Für viele Sprachen gibt es aber auch weiter gehende Standardbibliotheken, welche eine transparente Persistenz der Objekte ermöglichen. Diese Funktionalität wird als Object-Relational-Mapping (ORM) bezeichnet. Das heißt, der Entwickler muss die Persistenz nur konfigurieren (welche Objekte persistent gehalten werden, wo die Daten gespeichert werden etc.) und alle anderen Aufgaben der Persistierung werden automatisch übernommen. Beispiele in Java sind die Java Persistence API (JPA) in EJB 3, Hibernate und Java Data Objects (JDO).

6.7.3 Datenaustausch und Datentransformation mit XML

Beispiele für Datenaustausch und Datentransformation

Der reibungslose Austausch von strukturierten Daten und deren Transformation ist für viele Informationssysteme von enormer Bedeutung. Ein typisches Beispiel sind B2B-Transaktionen, wie einfache Online-Bestellungen. Hier ist es notwendig, dass der Einkäufer auf die Angebote des Anbieters zugreifen, diese interpretieren und dann automatisiert bestellen kann. Hier stellt sich das Problem, dass unter Umständen verschiedene Datenbeschreibungen bei den beiden beteiligten Unternehmen benutzt werden, denn meist fehlt ein industrieübereinander Daten-

standard. Die gleiche Situation kann selbst innerhalb eines einzigen Unternehmens entstehen. Z. B. entwickeln verschiedene Abteilungen größerer Unternehmen Software oft unabhängig voneinander, was zu unterschiedlichen Konventionen und Definition der Daten führt.

Dieses Problem besteht schon seit Langem und wurde durch eine Reihe von Lösungsansätzen abgedeckt. Der verbreitete EDI-Standard (Electronic Data Interchange) ermöglicht den elektronischen Austausch von Daten zwischen Unternehmen. EDI hat jedoch einige Nachteile hinsichtlich heutiger Anforderungen an einen Datenaustauschstandard. Insbesondere ist EDI relativ komplex und schwierig erweiterbar und benötigte – in der Vergangenheit – eine relativ teure, proprietäre Netzwerkinfrastruktur.

EDI

Daher wurde in der Vergangenheit – mit dem Aufkommen des Internets – der Datenaustausch mehr und mehr mit Web-Protokollen abgewickelt. HTML – die Sprache, mit der Web-Seiten im WWW beschrieben werden – ist jedoch nicht gut geeignet, um strukturierte Daten abzubilden. Daher wurde die Extensible Markup Language (XML) [Bray et al. 1998] mit dem Ziel entwickelt, einen flexiblen, erweiterbaren und einfachen Standard für den strukturierten Datenaustausch im Internet bereitzustellen. XML erlaubt es, Informationen strukturiert zu beschreiben. XML selbst ist kein Standard zum Datenaustausch im Unternehmen, sondern erlaubt es, XML-basierte Austauschformate und Austauschstandards zu definieren – also ist XML flexibel erweiterbar. Es ist auch recht einfach, proprietäre Datenformate in XML-Formate zu transformieren und umgekehrt, was für die Einbindung von Alt-Systemen wichtig ist, die oft proprietäre Datenaustauschformate verwenden. Ein weiterer wichtiger Vorteil von XML ist, dass es mittlerweile große Verbreitung auf vielen Plattformen, Sprachen und Systemen gefunden hat. Mittlerweile wird XML auch für viele anderen Aufgaben verwendet, als Datenaustausch und Datentransformation.

XML

Mit einer Document Type Definition (DTD) kann der Aufbau von XML-Dokumenten spezifiziert werden. DTDs sind einfach zu verstehen und Validatoren für DTDs sind verbreitet und effizient. DTDs haben jedoch einige Nachteile, insbesondere sind sie selbst keine XML-Dokumente und können daher nicht mit XML-Werkzeugen bearbeitet werden. Überdies sind die Möglichkeiten der Datenspezifikation begrenzt: Z. B. kann die Typisierung von Daten nicht vorgenommen werden.

DTD

Der XML-Schema-Standard [W3C 2004] hat das Ziel, diese Probleme zu lösen. Jedes Schema ist ein gültiges XML-Dokument und XML-Schemata

XML-Schema

Weitere XML-Standards

erlauben typisierte Daten, auf Basis von primitiven und selbst definierten Datentypen.

Die Definition der Datenformate – in DTDs oder Schemas – ist insbesondere für komplexe Anwendungen von großer Bedeutung.

Über diese Basiselemente der XML-Sprache hinaus gibt viele weitere Standards, welche XML in verschiedenen Bereichen komplettieren. Hier seien nur einige wenige, zentrale Standards von allgemeiner Bedeutung knapp aufgelistet:

- > XML Namespaces dienen der Unterscheidung von Bezeichnern in verschiedenen Kontexten. Dies gestattet dem Entwickler, Bezeichner frei zu wählen, ohne dass es zu Namenskonflikten bei der gemeinsamen Benutzung von zwei unabhängig voneinander entwickelten Dokumenten kommt, da beide ihre Bezeichner eindeutig im Kontext ihres Namensraums spezifizieren.
- > XHTML ist eine XML-Variante von HTML, die dieselben präsentationsorientierten Eigenschaften wie HTML hat, aber im XML-Sinne wohl geformt ist (beispielsweise jedes offene Tag muss auch beendet werden).
- > XLink ist ein Verknüpfungsmechanismus für XML-Dokumente. XLink-Verweise gehen deutlich über die aus HTML bekannten Links hinaus. Z. B. kann man auf eine Menge von Dokumenten verweisen und Traversierungen spezifizieren.
- > XPath ist eine Sprache zur Lokalisierung und Extraktion von Informationen innerhalb eines XML-Dokumentes.
- > XSLT erlaubt die Generierung beliebiger Dokumente aus XML-Dokumenten. Häufigstes Einsatzgebiet in der Praxis ist die Transformation von XML-Dokumenten. In XSLT wird XPath für die Spezifikation von XML-Strukturen benutzt.
- > XQuery ist eine SQL-ähnliche Abfragesprache für Abfragen an XML-Dokumente.
- > Das Resource Description Framework (RDF) ist eine XML-Sprache, die es erlaubt, Metadaten über Web-Ressourcen anzugeben. Diese Metadaten können unter Umständen in einer Ontologiesprache beschrieben sein.

Dies ist nur ein kleiner Ausschnitt der vielfältigen XML-Standards. Über diese allgemeinen Standards hinaus gibt es viele weitere domänen- oder branchenspezifische XML-Sprachen.

Neben den XML-Sprachen gibt es noch viele Standards, De-facto-Standards und APIs für die Verarbeitung von XML, wie z. B.:

- > SAX – eine programmatische API zur Verarbeitung von XML.
- > DOM – eine dokumentbaumbasierte API zur Verarbeitung von XML.
- > Redland – eine API zum Zugriff auf RDF Daten.

XML-Verarbeitung: APIs und Standards

XML wurde hier exemplarisch als ein Standard zur Definition von Datenaustauschsprachen erklärt – wegen seiner weiten Verbreitung. Viele andere Sprachen und Formate existieren. Oft ist es sinnvoll, XML zu verwenden. Die zentralen Vorteile sind, dass XML einfach, flexibel und erweiterbar ist. Allerdings gibt es auch einige Nachteile. XML benötigt gerade wegen seiner Eigenschaften relativ viel Speicherplatz (bzw. Bandbreite beim Datenaustausch), denn es benutzt sprechende Tags und meist eine Darstellung, die direkt von Menschen gelesen werden kann. Die String-Verarbeitung beim Parsen und Interpretieren von XML kann in einigen Fällen auch zu Performanz-Problemen führen. Überdies ist zwar XML selbst recht einfach, aber einige XML-Standards sind bereits sehr komplex. Diese Gründe mögen dafür sprechen, dass ein proprietäres Format doch besser für bestimmte Aufgaben geeignet ist als XML. Überdies kann die Migration zu XML, gerade in Altanwendungen, erhebliche Kosten verursachen.

XML: Vor- und Nachteile

Sicherlich wäre es sehr sinnvoll, eine einheitliche und allgemein akzeptierte Sprache für den unternehmensübergreifenden Datenaustausch zu haben. Jedoch zeigen selbst erfolgreiche Beispiele wie RosettaNet oder EDI, wie schwierig dieses Ziel zu erreichen ist. Daher scheinen domänen- bzw. branchenspezifische Lösungen die pragmatischere Lösung zu sein, wie das Beispiel RosettaNet – ein E-Business-Standard für die Hightech-Industrie – zeigt.

Domänen- und branchenspezifische Datenaustausch- formate

6.7.4 Dynamische Web-Seiten und Web-Anwendungsserver

Fast alle (größeren) Web-Sites benötigen heutzutage das dynamische Erzeugen von Web-Seiten. Das heißt, Daten werden dynamisch erzeugt oder von einem Backend, wie einer Datenbank oder Altanwendung, bei einer Web-Anfrage geholt. Dann werden diese Resultate mit HTML aufbereitet und anschließend ausgeliefert. Mittels Web-Anfragen (z. B. ausgelöst durch Klicken auf einen Link im Browser) oder durch HTML-Formulare können die Daten in einem Backend vom Browser aus verändert werden. Solche Änderungen müssen dann zurück ins Backend geschrieben werden.

Dynamische Web- Seiten

Web-Anwendungsserver

Um solche dynamischen Interaktionen im Web zu ermöglichen, muss hinter der Web-Seite ein Programm stecken, das die Seite dynamisch aus den aktuellen Daten des Backends erzeugt und, wenn nötig, Änderungen am Backend durchführt. Einen Web Server, der dies ermöglicht, bezeichnet man als Web-Anwendungsserver.

Techniken und Architekturen für serverseitige Programmmodul

Es gibt eine Reihe von Techniken und Architekturen, wie man serverseitige Programme in eine Web-Server-Architektur einbinden kann. Diese haben verschiedene Vor- und Nachteile:

- > Die *CGI-Schnittstelle* (Common Gateway Interface) ist eine der frühesten und immer noch benutzten Techniken, die praktisch von jedem Web Server unterstützt wird. Hier wird, wenn eine Web-Anfrage hereinkommt, dynamisch ein Prozess gestartet, der ein „kleines Programm“ ausführt. Oft werden hierzu Skriptsprachen wie Perl oder Tcl benutzt, aber jede andere Programmiersprache kann auch verwendet werden. Das Programm erhält die Anfrageparameter über UmgebungsvARIABLEN. Durch den neuen Prozess für jede Anfrage kann es aber zu Performanz-Problemen und hohem Ressourcen-Verbrauch kommen. *Fast CGI* ist eine CGI-Erweiterung, die dieses Problem durch Multi-Threading vermeidet. Jedoch stellt sich weiterhin das generelle Problem von CGI, dass „größere“ Interaktionen, wie ein kompletter Geschäftsvorgang, durch viele kleine Programme abgebildet werden, die nicht miteinander in Verbindung stehen. Somit sind komplexe CGI-Architekturen nur schwer zu verstehen und zu warten.
- > Eine Reihe von *Template-Sprachen* existiert, die Programmtext in HTML-Seiten einbetten und durch den Anwendungsserver dynamisch ersetzen lassen. Beispiele sind PHP, ColdFusion, Active Server Pages (ASP) und Java Server Pages (JSP). In diesen Sprachen besteht der Code aus normalem HTML-Code mit in Kommentaren (durch „<% ... %>“ abgetrennt) eingebetteten Code in der jeweiligen Template-Sprache. Alle genannten Ansätze bieten eine breite Unterstützung, gute Performanz, komfortable Bibliotheken, eine gute Datenbankanbindung und sind relativ leicht zu erlernen. Allerdings haben diese Ansätze zum Teil gerade für komplexe Projekte wiederum eine oder mehrere der folgenden Schwierigkeiten. Zum Teil fehlen einfache Mittel zur Kommunikation mit anderen Anwendungen. Auch kann Programmlogik in der jeweiligen Template-Sprache oft nicht für andere Zwecke verwendet werden. Die einzelnen Seiten bleiben relativ lose gekoppelt und sind also – genau wie bei CGI – nur durch Anfragen getrieben.
- > Einige der Nachteile von Template-Sprachen werden durch *Anwendungsserver*, wie Apache Tomcat, JBoss, BEA WebLogic oder IBM

WebSphere, gelöst. Diese sind oft Teil von größeren Standardarchitekturen, wie Microsoft's .NET oder JEE. Anwendungsserver sind professionelle, vollständige Systeme, die typischerweise ein serverseitiges Komponentenmodell, Transaktionsmanagement, Skalierung, Load Balancing, Sicherheitsmechanismen, Fail-Over und Integration mit anderer Middleware und Web Services bieten.

- › *Web-Content-Management- und Community-Systeme*, wie Zope oder Open ACS, basieren auf Web-Anwendungsservern und bieten zusätzlich viele Erweiterungsmodule, z. B. für Community-Funktionen, Foren, Wikis, Zusammenarbeit in virtuellen Gruppen etc.
- › *Agile Web Frameworks* bezeichnen Web Frameworks in dynamischen Programmiersprachen, wie Ruby on Rails, Seaside, Mason oder Grails. Sie werden als „agil“ bezeichnet, weil sie basierend auf Prinzipien wie „Don't Repeat Yourself“ (DRY) und „Convention over Configuration“ entworfen wurden. Das heißt diese Frameworks unterstützen die agile Software-Entwicklung, indem sie Programmierkonventionen über die Anwendungskonfiguration stellen und somit die rasche Umsetzung von Anforderungen erlauben.

Dieser kurze Abriss über Web-Architekturen hat die grundlegenden Architekturen für dynamische Web-Anwendungen knapp eingeführt. Weiter gehende Software-Muster zu dem Thema Generierung und Konvertierung von Inhalten im Web findet man in [Vogel und Zdun 2002].

6.7.5 Komponentenplattformen

In diesem Abschnitt sollen einige bekannte Plattformen, welche die Basisarchitektur Komponentenplattform umsetzen, unter architektonischen Gesichtspunkten kurz vorstellt werden.

Komponentenplattformen

6.7.5.1 SUN Java Enterprise Edition (JEE)

Die Java Enterprise Edition (JEE) ist eine auf der Java-Technologie basierende Komponentenplattform in der Programmiersprache Java. Damit ist diese Komponentenplattform plattformunabhängig. Die Herstellerunabhängigkeit ist nur eingeschränkt gegeben, weil sie auf Spezifikationen des Unternehmens Sun basiert. JEE ist zunächst eine Sammlung von Spezifikationen zu den Bausteinen einer Komponentenplattform. Wenn Sie JEE einsetzen möchten, müssen Sie aus einer Vielzahl von JEE-Implementierungen unterschiedlicher Hersteller (kommerziell oder open source) ein Produkt (Container und Dienste) auswählen. Je nach

JEE

Ausrichtung des Produkts werden nur bestimmte Teile der JEE-Spezifikationen umgesetzt. Auch kann es proprietäre Erweiterungen geben, die zu einer engen Herstellerbindung führen. All dies ist bei der Auswahl aus architektonischer Sicht zu berücksichtigen. Für welche Implementierungen Sie sich dann letztlich entscheiden, hängt vor allem von Ihren Anforderungen und Ihrem Budget ab.

JEE schränkt architektonische Freiheit ein

Für die Architektur eines Systems ist von Belang, dass mit JEE zahlreiche Schnittstellen einhergehen, welche befriedigt werden müssen, und dass JEE-Komponenten zum Zwecke der Auslieferung über XML-Konfigurationsdateien (sogenannte Deployment Deskriptoren) umfangreich konfiguriert werden müssen. Auch der Container muss konfiguriert werden. JEE setzt zahlreiche, der in den vorherigen Abschnitten beschriebenen Konzepte um. Diese Konzepte sollten wiederum auch angewendet werden, wenn JEE zum Einsatz kommt. Das heißt, JEE muss ganz gezielt in einen architektonischen Rahmen gesetzt werden und sollte nicht einfach in seiner „rohen“ Form zum Einsatz kommen. Dies ist wichtig, zum einen deshalb, weil mit JEE architektonische Zwänge entstehen und zum anderen, weil eine unsachgemäße Verwendung von JEE zu fatalen Problemen für ein System führen kann (z. B. schlechte Performanz). Sehr zu empfehlen ist es, in diesem Zusammenhang bewährte JEE-Entwurfstechniken zu verwenden, wie sie z. B. in [Alur et al. 2003] geschildert werden. Es folgt eine Übersicht zu den zentralen Bestandteilen von JEE. Abbildung 6.7-3 zeigt die JEE-Komponentenplattform im Überblick. Detaillierte Beschreibungen und weitere Literaturempfehlungen sind bei [Sun 2005] zu finden.

- > *Servlets/Java-Server-Pages (JSP)*: Diese Bausteine sind dafür verantwortlich, in JEE-basierten Web-Anwendungen zwischen Browser als Klient und Modellschicht zu vermitteln. Angesiedelt sind sie in der Darstellungsschicht.
- > *Enterprise Java Beans (EJB)*: Diese Bausteine stellen die eigentlichen Modellkomponenten dar und sind demzufolge in der Modellschicht vorzufinden. Sie liegen in den Ausprägungen Entity Bean (Entitäts-Komponenten), Session Bean (Session-Komponente) und Message Driven Bean (Nachrichten-Komponente) vor.



Abb. 6.7-3: JEE-Komponentenplattform im Überblick.

6.7.5.2 Microsoft .NET

Microsofts Komponentenplattform .NET ist konzeptionell JEE sehr ähnlich. So gibt es beispielsweise auch eine virtuelle Maschine, die Common Language Runtime (CLR). Es gibt jedoch eine ganze Reihe bedeutender Unterschiede in der Umsetzung der Konzepte. So unterstützt .NET verschiedene Programmiersprachen (VB, C++ etc.), wobei aber nur C# als die eigentliche .NET-Programmiersprache die Komponentenfähigkeit von .NET in vollem Umfang unterstützt. .NET ist mehr daten- als objektorientiert. Das heißt, der objektorientierte Entwurf wird stark von Datenstrukturen getrieben.

.NET vs. JEE

Diese Tatsache zeigt sich dann auch bei der Umsetzung des Komponentenansatzes. Als Komponente im engeren Sinne gibt es bei .NET nur Entitäts-Komponenten. Die gravierendsten Unterschiede zu JEE sind jedoch, dass .NET zwar grundsätzlich plattformunabhängig, in brauchbarer Form jedoch nur für die Windows-Plattform verfügbar ist und eine sehr enge Herstellerbindung besteht, weil es nur einen bedeutenden Hersteller, nämlich Microsoft, gibt.

Durch die enge Verzahnung mit dem Betriebssystem ergibt sich im Vergleich zu JEE insgesamt eine bessere Integration der verschiedenen Bausteine dieser Infrastruktur. Es bleibt festzuhalten, dass die architektonische Situation eine ähnliche ist wie bei JEE, jedoch erweitert um den Plattformaspekt. Für weitere Details zu .NET sei auf [Mircrosoft 2004a] verwiesen. Im Vergleich von Abbildung 6.7-4 mit Abbildung 6.7-3 werden die wesentlichen Gemeinsamkeiten und Unterschiede zwischen JEE und .NET ersichtlich.



Abb. 6.7-4: .NET-Komponentenplattform im Überblick.

6.7.5.3 Corba-Component-Model (CCM)

Das Corba-Component-Model (CCM) ist seit Version 3.0 Teil der Common Object Request Broker Architecture (CORBA). Es ist die Spezifikation eines Modells für verteilte Komponenten der Modellschicht und ihren Container. CCM ist programmiersprachen- und plattformunabhängig. JEE und .NET sind mit Abstrichen CCM-Anwendungen. Mit Ausnahme von wenigen Open-Source-Produkten gibt es bislang keine

CCM

vollständige Implementierung. Die OMG setzt mittlerweile mehr auf JEE als Komponententechnologie. Zu den weiter oben beschriebenen Komponentenarten kommen bei CCM noch die Prozesskomponenten hinzu, die einem Geschäftsvorgang entsprechen und optional persistent sowie von mehreren Klienten benutzt werden können. Zu einer CCM-Komponente können folgende Schnittstellen definiert werden:

- > *Facets*: Dienste, welche die Komponente nach außen anbietet.
- > *Receptacles*: Dienste, welche die Komponente von anderen Komponenten benötigt.

In Abbildung 6.7-5 sehen Sie CCM im Überblick. Alle weiteren Informationen zu CCM finden Sie bei [OMG 2005b].



Abb. 6.7-5: CCM-Komponentenplattform im Überblick.

6.7.6 Web Services

Web Services

Web Services sollen hier noch kurz gesondert als eine Middleware-Architektur, die die SOA-Basisarchitektur umsetzt und stark auf das zuvor beschriebene XML und Internet-Standards setzt, besprochen werden. SOA beschreibt eine allgemeine Basisarchitektur für die lose gekoppelte Interaktion zwischen verschiedenen, verteilten Software-Anwendungen. *Web Services* hingegen stellen eine mögliche – standariserte – Realisierung dieser Basisarchitektur dar.

Web Services: Entstehung

Web Services sind aus dem World Wide Web (WWW) entstanden. Das WWW wurde ursprünglich für den Austausch von unstrukturierter Information, wie HTML-Texten, entworfen. Aber mit der Zeit wurden Interaktionen zwischen Programmen immer wichtiger, beispielsweise im Bereich E-Commerce oder EDI. Hier kommen insbesondere XML [Bray et al. 1998] und darauf basierende Standards zum Einsatz. Die XML-RPC-Spezifikation [Winer 1999] stellte einen ersten Standard zur RPC-Kommunikation über XML dar.

Web-Service-Protokolle

Heutige Web Services basieren auf einer Schichtenarchitektur aus mehreren standardisierten Protokollen:

- > SOAP [Box et al. 2000] ist ein XML-basiertes Nachrichtenaustauschprotokoll, das schnell zum De-facto-Standard für Web Services ge-

worden ist. Es ist der – deutlich erweiterte – Nachfolger der XML-RPC-Spezifikation.

- Eine Alternative zu SOAP ist REST (Representational State Transfer), das kein Standard ist, sondern nur ein Architekturstil, der auf existierenden Web-Standards wie HTTP und URI basiert. Die Interaktion zwischen Client und Server wird bei REST-konformen Web Services unter Verwendung der durch HTTP definierten, einheitlichen Schnittstelle abgewickelt. Der konkrete Datenaustausch geschieht über benutzerdefinierte XML-Formate.
- In der Web-Service-Architektur spielt auch WSDL [Christensen et al. 2001] eine zentrale Rolle, da es eine SchnittstellenbeschreibungsSprache darstellt, die sowohl von den Nachrichtensendern als auch den Empfängern verstanden wird. Somit ist WSDL sehr wichtig für die Interoperabilität in heterogenen Systemen, beispielsweise wenn verschiedene Web-Service-Systeme interoperieren sollen. Auch WSDL basiert auf XML.
- Web Services benötigen kein spezielles Kommunikationsprotokoll. Zum Beispiel können HTTP oder andere Protokolle, wie SMTP, FTP, JMS, IIOP oder andere Protokolle, zum Einsatz kommen. HTTP wird von den meisten heutigen Web-Service-Systemen als Standardprotokoll unterstützt und meistens kann eine Reihe der anderen Protokolle als Plug-ins zusätzlich verwendet werden.
- UDDI ist ein Standard für einen Lookup-Service [OASIS 2002] und erlaubt somit das Auffinden von Web Services und deren WSDL-Beschreibung auf Basis von Eigenschaften. Allerdings hat sich UDDI bislang nicht wirklich kommerziell durchgesetzt und es sind viele andere – zum Teil proprietäre – Lookup-Services im Einsatz.
- Es existieren einige Standards zur Komposition von Web Services. Eine wichtige Kategorie dieser Standards beschreibt die Orchestrierung von Web Services durch Geschäftsprozessmodelle, die in einem Process Engine abgearbeitet werden. Die Aktivitäten der Prozesse (also die Prozessschritte) rufen die Web Services auf bzw. empfangen deren Resultate. Der wichtigste Standard in diesem Bereich ist momentan die Business Process Execution Language for Web Services (BPEL4WS) [Andrews et al. 2003], eine XML-basierte Workflow-Definitionssprache, die es erlaubt, Geschäftsprozesse zu beschreiben.

Über diese Standards hinaus gibt es eine Reihe anderer Web-Service-Standards, z. B. in den Bereichen Sicherheit und langlaufende Geschäftstransaktionen.

6.7.7 Zusammenfassung

Zusammenfassung: architekturelle Technologien

- > Es gibt architekturelle Technologien, die eine wesentliche Infrastruktur in vielen Software-Architekturen bereitstellen.
- > Die Kommunikations-Middleware ist eine zentrale Technologie für viele verteilte Systeme.
- > Wesentliche Middleware-Systeme sind Transaktionsmonitore, RPC- und OORPC-Systeme und Message-oriented Middleware.
- > Ein Strukturbruch ist eine wichtige architektonische Problematik, die den Bruch zwischen zwei Paradigmen bezeichnet. Z. B. bei der persistenten Datenhaltung kommt der Strukturbruch zwischen den Paradigmen der Datenbank und der Applikationslogik vor.
- > Object-Relational-Mapping ermöglicht die Integration einer objekt-orientierten Anwendung mit dem relationalen Paradigma. Dazu bietet ORM eine Datenbankzugriffsschicht für relationale Datenbanken und objektorientierte Applikationslogik. Object-Relational-Mapper, wie Hibernate, sind eine wichtige Technologie in diesem Bereich.
- > Eine weitere wichtige Technologie für die persistente Datenhaltung sind Standardbibliotheken für den Datenbankzugriff, wie JDBC.
- > Der reibungslose Austausch von strukturierten Daten und deren Transformation ist für viele Informationssysteme von enormer Bedeutung. Eine wichtige Beispieltechnologie in diesem Bereich ist XML.
- > XML erlaubt es, standardisiert die Struktur von Informationen zu beschreiben und dabei Inhalte und Struktur zu trennen. XML selbst ist kein Standard zum Datenaustausch zwischen Organisationen, sondern ermöglicht es, XML-basierte Austauschformate und Austauschstandards zu definieren.
- > Fast alle (größeren) Web-Sites benötigen heutzutage das dynamische Erzeugen von Web-Seiten.
- > Ein Web-Anwendungsserver ist ein Server für Web-Anwendungen, der dynamisch HTML-Seiten erzeugt und die Anwendungslogik zwischen dem Klienten (Browser) und nachgelagerten Systemen, wie einer Datenbank, ausführt.
- > Wichtige Komponentenplattformen sind Java Enterprise Edition (JEE), Microsoft .NET und das Corba-Component-Model (CCM).
- > Web Services sind eine Technologie, die sowohl die Middleware- als auch die SOA-Basisarchitektur umsetzt und stark auf XML und Internet-Standards setzt.

7 | Organisationen und Individuen (WER)

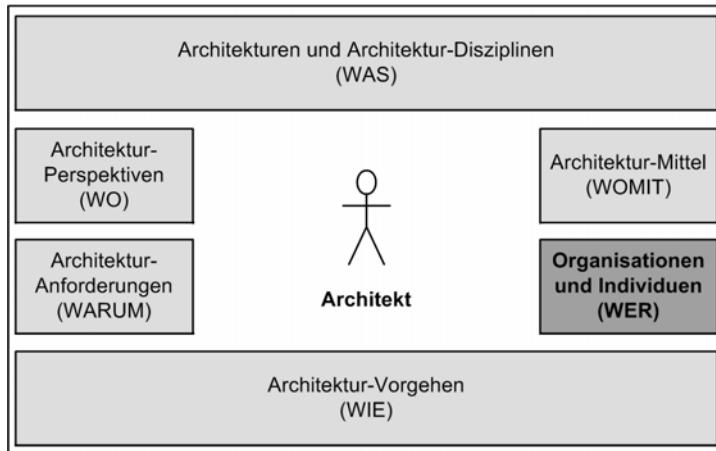


Abb. 7-1: Positionierung des Kapitels im Ordnungsrahmen.

In diesem Kapitel wird die *WER-Dimension* des architektonischen Ordnungsrahmens näher betrachtet und vertieft. Dabei werden organisatorische und soziale Einflussfaktoren aufgezeigt, die die Architektur eines Systems berühren und die Arbeit des Architekten beeinflussen können. Ferner wird grundlegendes Wissen zu Gruppen und ihrer Dynamik vermittelt. Darüber hinaus wird die Rolle des Architekten herausgearbeitet. Durch die Berücksichtigung der in dieser Dimension behandelten Themen sind Sie unter anderem in der Lage, die Relevanz der genannten Einflussfaktoren zu verstehen, die Rolle eines Architekten einzuzuordnen und gruppendifamische Prozesse besser zu beachten.

Übersicht

7.1	Allgemeines	312
7.2	Organisationen	316
7.3	Individuen	321
7.4	Individuen und Gruppen	324
7.5	Architektur und Entscheidungen	328
7.6	Architekt als zentrale Rolle	332
7.7	Zusammenfassung	337

Grundlegende Konzepte der WER-Dimension

Abbildung 7-2 stellt die grundlegenden Konzepte, welche in diesem Kapitel behandelt werden, vor und visualisiert ihren Zusammenhang.

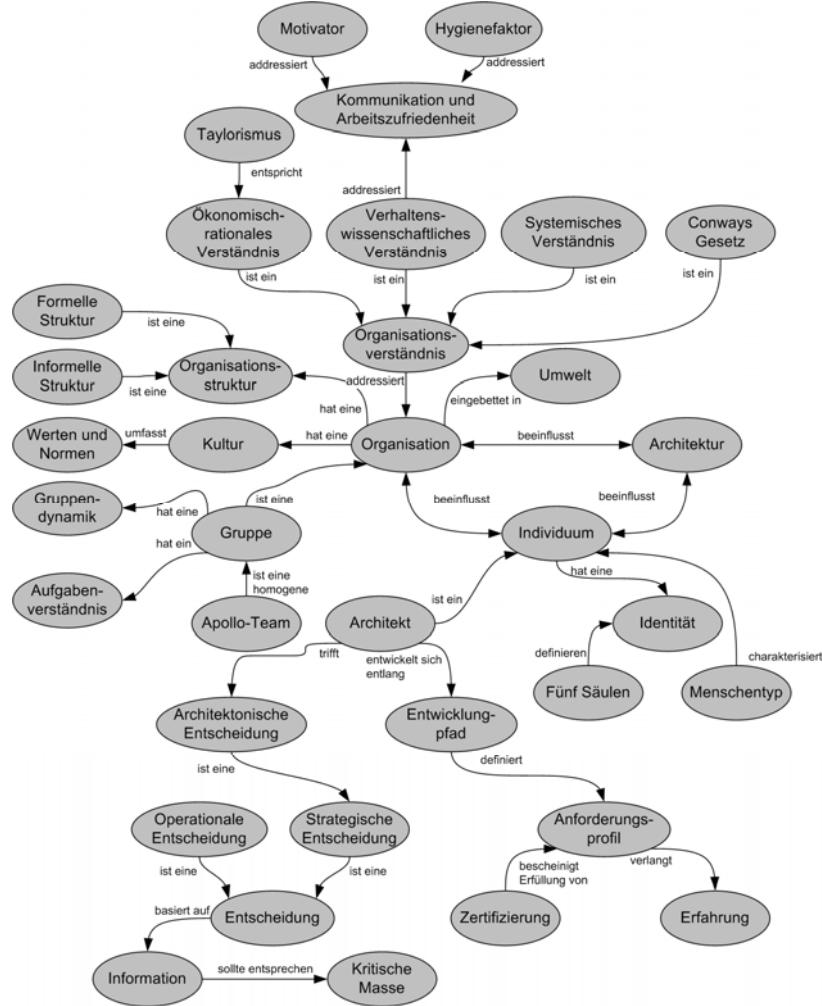


Abb. 7-2: Grundlegende Konzepte der WER-Dimension.

7.1 Allgemeines

Fokussierung auf technische und architektonische Aspekte reicht nicht aus

Bei der Gestaltung einer Architektur müssen vielfältige Einflüsse und Aspekte berücksichtigt werden. Am offensichtlichsten ist hierbei die Beachtung der an das IT-System gestellten funktionalen und nicht-funktionalen Anforderungen durch die Architektur. Um diesen Anforderungen Rechnung

zu tragen, können verschiedene Architektur-Mittel genutzt werden (siehe Kapitel 6). Neben diesen Mitteln kann die zu erstellende Lösung aus unterschiedlichen Perspektiven betrachtet werden (siehe Kapitel 4). Darüber hinaus kann man nach einem bestimmten Vorgehensmodell handeln, um so eine systematische und Erfolg versprechende Architektur-Gestaltung zu gewährleisten (siehe Kapitel 8). Im Sinne der im Rahmen der WAS-Dimension eingeführten Architektur-Definition (siehe Kapitel 3) werden damit alle wesentlichen Aspekte, die zur Ausübung seiner architektonischen Tätigkeit notwendig sind, behandelt. Dies allein reicht jedoch noch nicht aus. Vielmehr müssen auch soziale und organisatorische Einflussfaktoren berücksichtigt werden. Dies erfordert einen Blick über den technologischen Tellerrand hinaus und ein grundlegendes Verständnis von Organisationen und Individuen. Aus diesem Grund behandelt dieses Kapitel Organisationen und Individuen allgemein und zeigt dabei auch die Wechselwirkungen mit Architektur auf.

Architekturen beziehungsweise darauf basierende IT-Systeme werden stets durch und für Menschen entworfen. Ferner erfolgt die architektonische Tätigkeit in der Regel eingebettet in eine Organisation, sei es nun das Unternehmen, für das das IT-System entworfen wird, oder die Projektorganisation, die die involvierten Personen vereint. Der Organisationsbegriff wird hiermit bewusst weit gefasst. Er kann sich einerseits auf Unternehmen und andererseits auf Projektorganisationen beziehen. Wie in Abbildung 7.1-1 dargestellt, steht eine Architektur somit immer in Wechselwirkung mit der Organisation, in der sie entworfen wird, sowie den Individuen, die an der Architektur beteiligt sind und durch sie berührt werden.

Architekturen entstehen durch und für Menschen in Organisationen

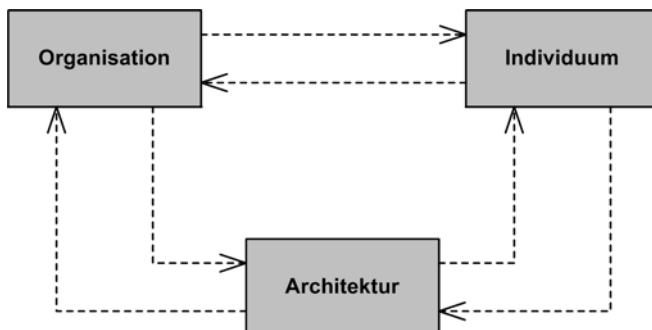


Abb. 7.1-1: Interdependenzen zwischen Organisation, Individuum und Architektur.

Eine Organisation zeichnet sich durch ihre Kultur sowie damit verbundene Werte und Normen aus. Diese wirken auf die Architektur ein,

Einflüsse durch die Organisationskultur

indem sie den normativen Rahmen und somit den Freiraum der Architektur-Gestaltung definieren. Dies kann sich dadurch äußern, dass ganz klare Standards und Richtlinien vorgegeben sind und dass die Organisation z. B. neue, unkonventionelle Ansätze ablehnt. Folglich besteht die Gefahr, dass eine Architektur scheitert, die den Werten und Normen der Organisation widerspricht, obwohl sie alle offensichtlichen funktionalen und nicht-funktionalen Anforderungen erfüllt. Ferner legt die Kultur einer Organisation die Art und Weise fest, wie Menschen innerhalb der Organisation miteinander umgehen und welche Erwartungen die Organisation an sie stellt. Dadurch existieren auch Wechselwirkungen zwischen der Organisation und dem einzelnen Individuum, die sich wiederum auf die Architektur auswirken. Eine Organisation wirkt somit auch auf die Architektur als Disziplin ein (siehe Kapitel 3). Eine Organisation kann aus weiteren Suborganisationen mit abweichenden Kulturen bestehen. So können in einem Unternehmen als übergeordnete Organisation verschiedene Abteilungen respektive Organisationseinheiten mit eigenen Kulturen existieren. Gerade in der heutigen Zeit, in der Unternehmen international vertreten sind, ist dieser Aspekt von besonderer Relevanz, da sich Organisationen somit über verschiedene Kulturreiche erstrecken, die sich durch andere Werte und Normen auszeichnen.

Einflüsse durch die Organisationsstruktur

Die Strukturen einer Organisation beeinflussen ebenfalls die Architektur eines IT-Systems. Existieren in einem Unternehmen z. B. dedizierte Abteilungen zur Entwicklung von verschiedenen Teilen eines IT-Systems, besteht die Gefahr, dass die Gesamtarchitektur diese organisatorische Trennung widerspiegelt. Dies liegt darin begründet, dass die organisatorische Trennung die Kommunikation über Abteilungsgrenzen hinweg erschwert. Empirische Studien zeigen, dass gerade die Integration zum Gesamtsystem organisatorisch getrennte Projektteams vor große Herausforderungen stellt [Herbsleb und Grinter 1999]. Der Einfluss von Organisationsstrukturen auf die Architektur eines IT-Systems wurde von Melvin Conway erkannt und ist als Conways Gesetz in die Literatur eingegangen [Conway 1968]:

Conways Gesetz

Organisationen sind darauf beschränkt, Systeme zu entwerfen, deren Strukturen Kopien ihrer organisatorischen Kommunikationsstrukturen sind.

Organisationen beeinflussen Architektur

Organisationen beeinflussen folglich die Architektur. Um dies zu verstehen, ist es wichtig, ein Verständnis von Organisationen zu besitzen. Aus diesem Grund behandelt Abschnitt 7.2 Organisationen genauer. Organisatorische Einflüsse sind vor allem im Zeitalter des Offshoring von großer Bedeutung, da nur eine übergreifende, regelmäßige Koordi-

nation und Kommunikation dazu führen kann, ein homogenes Gesamtsystem zu erhalten [Curtis et al. 1988].

Menschen unterscheiden sich durch ihre Stärken, Schwächen, Wünsche, Ängste und Mentalität. Menschen tragen durch ihre individuellen Eigenschaften zu einem IT-System bei. Sie agieren in unterschiedlichen Rollen, wie z. B. Architekt, Designer oder Entwickler, und erfüllen die an die Rollen gestellten Erwartungen auf unterschiedliche Art und Weise. Hierbei können Menschen unterschiedliche Rollen gleichzeitig wahrnehmen. Je nach Erfahrungshintergrund wird ein Architekt eine Architektur auf die eine oder andere Weise entwerfen. Ein Designer wird die architektonischen Vorgaben je nach Verständnis und Vorlieben in seinen Entwurf einfließen lassen und ein Entwickler wird diese auf seine ganz besondere Art und Weise umsetzen. Die Individualität jedes einzelnen Projektmitglieds beeinflusst somit die Architektur, da jedes Mitglied seine zugeteilten Rollen ganz individuell ausfüllen wird. Menschen sind einzigartig und werden auch immer ganz individuelle Beiträge leisten. Die unterschiedlichen Mentalitäten beeinflussen auch die Zusammenarbeit und die Kommunikation zwischen den Projektmitgliedern und wirken sich indirekt auf die Architektur aus, da eventuell wichtige Informationen nicht kommuniziert werden oder aber Verständnisprobleme nicht ausgesprochen werden. Dies kann dazu führen, dass die Architektur auf dem Papier sich deutlich von der letztlich realisierten Architektur unterscheidet. Ferner ist es möglich, dass architektonische Vorgaben nicht angenommen werden, da ihre Zweckmäßigkeit nicht erkannt wird oder aber Vorschläge anderer nicht angenommen werden. Dieses Verhalten wird in der Literatur oft als Not-invented-here-Syndrom bezeichnet [Cockburn 2002]. Darüber hinaus sind Sympathien und Antipathien zwischen Menschen von großer Bedeutung. Wenn z. B. die Zusammenarbeit zwischen dem Architekten und seinen Teammitgliedern aufgrund von zwischenmenschlichen Differenzen beeinträchtigt ist, wird die Architektur nicht von allen mitgetragen werden und letztlich nicht erfolgreich sein. Dies macht es unabdingbar, dass ein Architekt nicht nur ein Experte in fachlichen und methodischen Bereichen ist, sondern vielmehr auch über soziale Kompetenzen verfügt und ein grundlegendes Verständnis von Individuen besitzt. Diesem Verständnis widmet sich Abschnitt 7.3.

Es sind aber nicht nur die Organisation und das einzelne Individuum, die auf die Architektur einwirken. Vielmehr berührt auch die Architektur die Organisation und das einzelne Individuum. Eine Architektur definiert die Strukturen eines IT-Systems, indem es dessen Subsysteme und Bausteine identifiziert. Jedes Subsystem besitzt dedizierte Verant-

Einflüsse durch Individuen

Einflüsse durch Architekturen

wortlichkeiten und zwischen den Subsystemen existieren Abhängigkeiten. In der Regel wird jedes Subsystem von unterschiedlichen Teams weiter entworfen und umgesetzt. Es erfolgt also häufig eine Strukturierung der Organisation auf Basis der architektonischen Strukturen [Brooks 1995]. Dadurch wirkt die Architektur zum einen auf die Organisation ein. Zum anderen beeinflusst die Architektur auch jedes einzelne Teammitglied, da jedem Mitglied bestimmte Rollen zukommen. So wird es beispielsweise die Rolle des Designers geben, der das Subsystem weiterstrukturiert, oder die des Testers, der die Testfälle und -szenarien für das Subsystem entwirft.

7.2 Organisationen

Ziele

Nachdem im vorherigen Abschnitt der Einfluss von Organisationen auf Architektur motiviert wurde, wird im Folgenden wesentliches Wissen zum Verständnis von Organisationen vermittelt. Dabei werden einerseits allgemeingültige Themen aus der Organisationslehre behandelt. Andererseits werden diese Themen aber auch unter dem besonderen Blickwinkel von Architektur betrachtet. Das Thema wird dabei zunächst losgelöst von Architektur eingeführt und danach konkret mit Architektur in Beziehung gesetzt.

Definition: Organisation

Eine Organisation kann gemäß folgender Definition beschrieben werden [Kieser und Kubicek 1993]:

Eine Organisation ist ein soziales Gebilde, das dauerhaft ein Ziel verfolgt und eine formale Struktur aufweist, mit deren Hilfe Aktivitäten der Mitglieder auf das verfolgte Ziel ausgerichtet werden sollen.

Unterschiedliche Interpretationsmöglichkeiten

Im Laufe der Zeit haben sich unterschiedliche Verständnisse und Interpretationen von Organisationen entwickelt. Die wesentlichen Grundverständnisse werden im Folgenden näher besprochen. Dies sind im Einzelnen:

- > Ökonomisch-rationales Verständnis.
- > Verhaltenswissenschaftliches Verständnis.
- > Systemisches Verständnis.

Ökonomisch-rationales Verständnis

Das ökonomisch-rationale Verständnis hat seine Wurzeln in der frühen Industrialisierung. Ihm liegt das Prinzip der perfekten Arbeitsteilung zugrunde. Der Mensch als Individuum wird dabei als Produktionsfaktor wahrgenommen, der planbar, vorhersagbar und steuerbar ist. Organisationen, die durch dieses Prinzip geprägt sind, zeichnen sich durch starre

Organisationshierarchien aus. Die Kommunikation erfolgt streng über die Hierarchie. Dieser Ansatz kann zu einer starken Abgrenzung zwischen den einzelnen Organisationseinheiten respektive Teams führen. In seiner extremen Form erfolgt eine klare Trennung zwischen dem Planen und Ausführen. F. W. Taylor kann als Vater dieses Verständnisses angesehen werden [Taylor 1913]. Deshalb wird in diesem Zusammenhang auch oft vom *Taylorismus* gesprochen.

Wenn man sich nun Conways Gesetz noch einmal in Erinnerung ruft, wird auch evident, warum Organisationen, die unter rein ökonomisch-rationalen Gesichtspunkten aufgebaut sind, IT-Systeme entwickeln, die ihre Kommunikationsstrukturen widerspiegeln. Die klare Trennung ist in der Organisation vorhanden und ein übergreifendes Verständnis existiert nicht. Wenn nun z. B. unterschiedliche Teams für unterschiedliche Subsysteme eines Systems verantwortlich sind, wird auch die Architektur des IT-Systems diese Struktur aufweisen. Aufgrund dieses Sachverhalts muss sich ein Architekt bewusst sein, dass wesentliche, architektonische Prinzipien (siehe Kapitel 6), wie Separation of Concerns, Modularisierung und Information Hiding, zwar für die Architektur eines IT-Systems wichtig sind, dass sie jedoch nicht im gleichen Maße auf die Organisation, die das IT-System realisiert, angewendet werden dürfen. Stattdessen müssen wichtige Informationen fließen, die Kommunikation muss erleichtert werden und es müssen neue Formen der Zusammenarbeit geschaffen werden, beispielsweise durch den Einsatz von Kollaborationswerkzeugen, wie Wikis, Instant-Messaging-Systemen und Team-Rooms.

Conways Gesetz und Taylorismus

Das verhaltenswissenschaftliche Verständnis rückt den Menschen in den Mittelpunkt der Betrachtung, indem es ihn nicht mehr als reinen Produktionsfaktor, sondern als soziales Wesen wahrnimmt, das nach Anerkennung und Wertschätzung strebt. Die strikte Arbeitsteilung rückt dabei in den Hintergrund und das Hauptaugenmerk liegt auf der Schaffung von geeigneten Arbeitsbedingungen, die es dem Menschen erlauben, sich zu entfalten. Ein wesentlicher Aspekt ist hierbei die Förderung der Kommunikation und die Steigerung der Arbeitszufriedenheit durch geeignete Motivationsmaßnahmen. Herzberg spricht in diesem Zusammenhang von Motivatoren und Hygienefaktoren [Herzberg 1966]. Als Motivatoren können z. B. Arbeitsinhalte, Verantwortung und Anerkennung genannt werden. Diese steigern zwar zum einen die Arbeitszufriedenheit, zum anderen senken sie aber nicht die Arbeitsunzufriedenheit. Die Arbeitsunzufriedenheit wird durch Hygienefaktoren, wie die Beziehung zu Vorgesetzten und Gleichgestellten sowie die Unternehmenspolitik, beeinflusst. Wenn diese Faktoren als positiv wahr-

Verhaltenswissenschaftliches Verständnis

Verhaltenswissenschaftliches Verständnis und Architektur

genommen werden, können sie die Arbeitsunzufriedenheit senken, aber die Arbeitszufriedenheit nicht steigern [Drumm 1995].

Aus dieser Erkenntnis lässt sich für die Gestaltung einer Architektur ableiten, dass zur Minimierung der Arbeitsunzufriedenheit die Prinzipien und Konzepte der Architektur an Teammitglieder kommuniziert werden müssen und geeignete Kommunikationswege und -mittel geschaffen werden müssen. Hierzu kann das gemeinsame morgendliche Meeting gehören, in dem über Probleme und nächste Schritte gesprochen wird [Beedle und Schwaber 2001] oder die Etablierung einer Instant-Messaging-Umgebung. Zur Steigerung der Arbeitszufriedenheit reicht dies jedoch nicht aus. Vielmehr sollte jedes Teammitglied mit Tätigkeiten betraut werden, mit denen es sich identifizieren kann. Darüber hinaus sollten sie an der Architektur-Gestaltung beteiligt und zur konstruktiven Reflexion ermuntert werden. Architektur sollte letztlich als Teamleistung verstanden werden. Vor diesem Hintergrund wird auch deutlich, dass eine klare Rollentrennung, wie sie z. B. beim Taylorismus gegeben ist, bei der Realisierung von IT-Systemen nicht sinnvoll ist.

Systemisches Verständnis

Unter systemischen Gesichtspunkten ist eine Organisation nichts anderes als ein System (siehe Abschnitt 3.3) und erfüllt folglich die klassischen Eigenschaften von Systemen (siehe Abbildung 7.2-1).

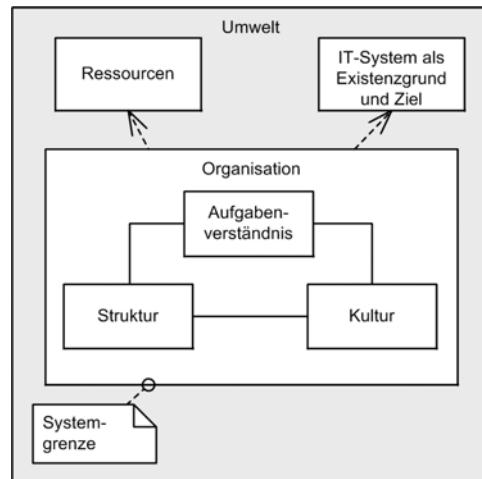


Abb. 7.2-1: Organisation als System (in Anlehnung an [Steiger und Lippmann 2003]).

Organisationen besitzen Ziele

Eine Organisation existiert zur Erreichung eines *Ziels*. Die Realisierung eines IT-Systems zur Erfassung und automatischen Abwicklung von Aufträgen ist ein exemplarisches Ziel einer Projektorganisation.

Die Umwelt, beispielsweise der Auftraggeber des IT-Systems, setzt der Organisation dieses Ziel und liefert ihr somit ihren Existenzgrund. Weiter stellt die Umwelt der Organisation *Ressourcen* zur Erfüllung ihrer *Aufgabe* bereit. Dies können materielle Dinge, wie Räumlichkeiten und Arbeitsmittel, oder aber auch immaterielle Dinge, wie Informationen, sein. Ferner können Menschen bis zur Erreichung des Ziels der Organisation zur Verfügung gestellt werden. Sie werden somit Bestandteil der Organisation. Da eine Organisation in ihre Umwelt eingebettet ist, gibt die Umwelt auch die *Rahmenbedingungen* vor, in denen sich die Organisation bewegen kann. Ein wichtiger Aspekt ist hierbei die *Kultur* der Umwelt und die damit verbundenen *Werte* und *Normen*. Darunter können die Art und Weise, wie miteinander umgegangen wird, und konkrete Prozessvorgaben fallen.

Organisationen interagieren mit ihrer Umwelt

Als umgebende Umwelt kann ein Unternehmen angesehen werden. Ein Architekt ist nun beispielsweise einerseits Mitglied des Unternehmens und andererseits Mitglied der Projektorganisation. Je nach Kultur wird ein Architekt motiviert sein, bei der Umsetzung der entworfenen Architektur selbst Hand anzulegen oder die eigentliche Handarbeit seinen Teammitgliedern zu überlassen.

Umwelt und Architektur

Eine Organisation entwickelt ein individuelles Verständnis, wie das von der Umwelt vorgegebene Ziel zu erreichen ist. Dieses *Aufgabenverständnis* basiert auf den Erfahrungen der Organisationsmitglieder. Daraus wird jede Organisation eine gestellte Aufgabe auf unterschiedliche Art und Weise lösen. Mit anderen Worten handelt sie *autonom*. Im Fall eines IT-Systems kann eine Architektur mit unterschiedlichen Mitteln gestaltet werden. So kann beispielsweise ein datenzentriertes System gemäß dem Repository-Architekturstil oder aber dem Blackboard-Architekturstil strukturiert werden (siehe Kapitel 6.4).

Organisationen besitzen ein Aufgabenverständnis

Zur Erreichung des Ziels der Organisation (z. B. zur Realisierung eines IT-Systems) strukturiert sich die Organisation. Die Struktur betrifft hierbei zum einen die Aufbau- und zum anderen die Ablauforganisation. Die *Aufbauorganisation* beschreibt dabei die zu besetzenden Stellen und ihre hierarchischen Beziehungen. Die Stellenbeschreibungen charakterisieren die innerhalb der Organisation zu besetzenden Rollen. Prozesse zur effizienten Zielerreichung werden durch die *Ablauforganisation* definiert. In diesem Zusammenhang können beispielsweise Entwicklungsprozesse wie der Unified Software Development Process [Jacobson et al. 1999] zum Einsatz kommen (siehe Abschnitt 8.1).

Organisationen besitzen Strukturen

Innerhalb von Organisationen bilden sich *formelle* und *informelle* Strukturen. Die formellen Strukturen werden durch die offizielle Projektorga-

Formelle und informelle Strukturen

nisation vorgegeben. Diese legt z. B. fest, dass die Kommunikation mit Mitgliedern anderer Teilteams eines Projekts nur über den Teamleiter zu erfolgen hat. Die informellen Strukturen umgehen diese Vorgaben und ermöglichen eine unmittelbare Kommunikation über Organisationsgrenzen hinweg. Diese Strukturen kommen durch Beziehungen zwischen Menschen zustande, die über die Projektorganisation hinausgehen. So hat man beispielsweise bereits bei einem vorherigen Projekt miteinander gearbeitet oder man geht demselben Hobby nach und spielt z. B. regelmäßig gemeinsam Tennis. Studien belegen, dass gerade diese informellen Brücken essenziell für die erfolgreiche Erreichung des Organisationsziels sind [Herbsleb und Grinter 1999].

Organisationen besitzen Kulturen

Das Aufgabenverständnis sowie die Strukturen der Organisation beruhen auf den Werten und Normen der beteiligten Menschen. Setzt ein Projektleiter beispielsweise auf einen sehr autoritären Führungsstil, wird die Organisationsstruktur streng hierarchisch ausgelegt sein und es wird sehr klare Vorgaben geben, wie die Aufgaben zu erfüllen und zu verstehen sind. Ähnliches gilt für den Architekten. Vertraut er auf die Fähigkeiten seiner Teammitglieder und lässt sie an der Architektur-Gestaltung partizipieren, werden die Organisationsstrukturen eher flach sein und es wird ein kollektives Aufgabenverständnis vorherrschen. Die besonderen Werte und Normen einer Organisation bezeichnet man als *Organisationskultur*. Die Organisationskultur legt auch fest, wie die Organisation mit ihrer Umwelt interagiert. Sie kann z. B. definieren, ob Teammitglieder, wie Entwickler oder Tester, direkt mit dem Kunden kommunizieren dürfen oder nicht.

Interdependenzen zwischen Struktur, Aufgabenverständnis und Kultur

Die Struktur, das Aufgabenverständnis und die Kultur einer Organisation sind eng miteinander verbunden. Sie beschreiben immer die gesamte Organisation, jedoch aus unterschiedlichen Perspektiven und beeinflussen sich gegenseitig. In komplexen Organisationen lassen sich Ursache und Wirkung nicht eindeutig voneinander trennen [Steiger und Lippmann 2003].

Organisationsverständnis in der IT

Dem Einfluss von Organisationen auf Architekturen wird in Entwicklungsprojekten und generell in der IT immer mehr Rechnung getragen. Im Laufe der Zeit haben sich verschiedene Organisationsprinzipien und -muster entwickelt, nach denen Organisationen strukturiert und gelebt werden sollten. Alistair Cockburn, Jim Coplien und Neil Harrison behandeln diesen Themenkomplex ausführlich [Cockburn 2002, Coplien und Harrison 2004]. Des Weiteren haben sich in letzter Zeit viele agile Software-Entwicklungs- respektive Projekt-Management-Methoden entwickelt, wie Scrum, die Crystal-Methodenfamilie oder eXtreme Pro-

gramming (XP), um nur einige zu nennen [Fowler 2003]. All diese Ansätze basieren auf dem verhaltenswissenschaftlichen und systemischen Organisationsverständnis. Sie rücken den Menschen ins Zentrum und betrachten ihn als motiviertes Individuum. Ferner sind nach ihrem Verständnis Organisationen so zu gestalten, dass sich Menschen in ihnen wohlfühlen und entfalten können. Letztlich muss es also das Ziel jeder Organisation sein, die Arbeitszufriedenheit zu erhöhen und die Arbeitsunzufriedenheit zu senken. Darüber hinaus vertrauen sie auf die *Selbstorganisation* von Organisationen [Cunningham et al. 2001]. Um diesen Forderungen gerecht zu werden, ist es wichtig, grundlegende Kenntnisse von Individuen und sich selbst organisierender Gruppen zu besitzen. Aufgrund dessen wird im nächsten Abschnitt auf den Menschen als Individuum näher eingegangen. Im Anschluss daran wird die Gruppe als sich selbst organisierende Einheit detaillierter behandelt.

7.3 Individuen

Erst durch Individuen werden Architekturen geschaffen. Menschen besitzen unterschiedliche Charaktereigenschaften, Stärken, Schwächen, Vorlieben und Abneigungen. Aus diesem Grund reicht es nicht aus, Menschen als reinen Produktionsfaktor zu betrachten. Stattdessen ist es für einen Architekten wichtig, seine Teammitglieder als Individuen wahrzunehmen und entsprechend zu behandeln. Deshalb wird in diesem Abschnitt grundlegendes Wissen zu Individuen vermittelt.

Um sich ein Bild von Menschen machen und sie besser verstehen zu können, ist es zunächst einmal bedeutend, zu erkennen, dass jeder Mensch eine eigene Identität besitzt. Die Identität kann dabei als auf fünf wesentlichen Säulen beruhend angesehen werden. Diese Säulen werden in Tabelle 7.3-1 vorgestellt [Petzold und Sieber 1993].

Tab. 7.3-1: Die fünf Säulen der Identität.

Soziales Netz	Beruf und Arbeit	Leiblichkeit	Materielle Werte	Werte und Normen
Familie	Status	Gesundheit	Geld	Religion
Freunde	Tätigkeit	Alter	Auto	Politik
Nachbarn	Verantwortung	Ernährung	Kleidung	Normen
Kollegen		Geschlecht		Tradition
		Sexualität		

**Individuen gestalten
Architektur**

**Fünf Säulen
der Identität**

Dabei ist bei jedem Menschen jede Säule unterschiedlich stark ausgeprägt. Dem einen ist das Vorankommen im Beruf wichtig und dem anderen eher unwichtig. Der eine legt viel Wert auf sein Prestige und der andere interessiert sich mehr für seine Familie.

Menschentypen nach Belbin

Obwohl jeder Mensch einzigartig ist, können die in Abbildung 7.3-1 dargestellten Menschentypen unterschieden werden [Belbin 1993]. Dies ist nur eine mögliche Theorie, die jedoch die grundsätzlichen Charaktereigenschaften von Menschen gut veranschaulicht. In der Literatur finden sich weitere Ansätze zur Interpretation von Menschen. So z. B. der von Meyer und Briggs entwickelte Ansatz, der auf den Theorien von C. G. Jung basiert. Er skizziert ebenfalls verschiedene Menschentypen und zeigt weiter auf, welche Typen in einem Team zusammenarbeiten können. Weiterführende Informationen hierzu findet man in [Briggs und Myers 1995].

Belbin identifiziert die in Abbildung 7.3-1 dargestellten Menschentypen. Für das Verständnis von gruppendynamischen Prozessen ist es wichtig, diese grundlegenden Typen zu kennen.

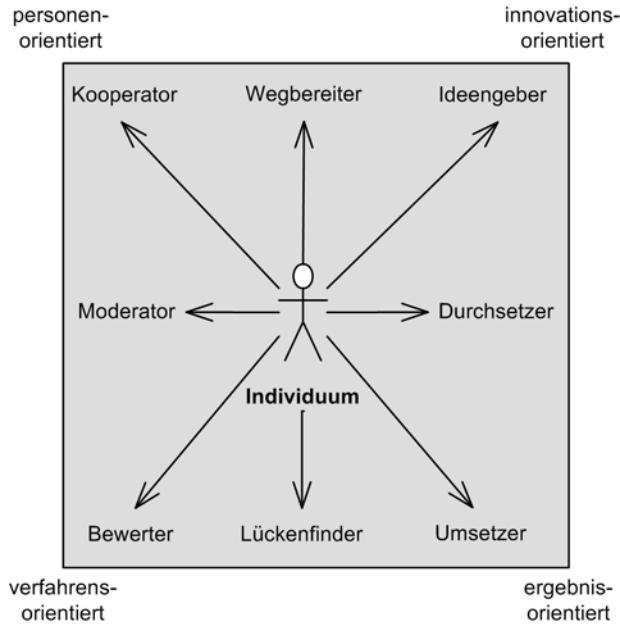


Abb. 7.3-1: Menschentypen nach Belbin.

Die einzelnen Menschentypen werden in Tabelle 7.3-2 beschrieben.

Tab. 7.3-2: Beschreibung der Menschentypen nach Belbin.

Typ	Beschreibung
Kooperator	Ein Kooperator besitzt die Fähigkeit, aufgrund seines sozialen Wesens auf Menschen einzugehen und den Teamgeist zu fördern. In Krisensituationen tendiert er eher zur Unentschlossenheit.
Wegbereiter	Ein Wegbereiter sucht Herausforderungen, ist extrovertiert und kommunikativ. Es ist eine seiner Stärken, menschliche Kontakte aufzubauen und neue Themen zu erforschen. Jedoch besteht die Tendenz, dass er das Interesse an einem Thema verliert, wenn es zur Routine wird.
Ideengeber	Ein Ideengeber geht unkonventionelle Wege und kann aufgrund seines Wissens und seiner Vorstellungskraft zu Lösungen beitragen. Allerdings neigt er auch dazu, Vorschriften zu übersehen und in den Wolken zu schweben.
Durchsetzer	Ein Durchsetzer besitzt eine dynamische Persönlichkeit, ist willensstark und in der Lage, Entscheidungen durchzusetzen. Er ist jedoch auch erregbar und neigt zur Provokation.
Umsetzer	Ein Umsetzer ist ein gewissenhafter Mensch, der eine Aufgabe sorgfältig erledigt. Er neigt teilweise zum Perfektionismus und kann sich an Kleinigkeiten stören.
Lückenfinder	Ein Lückenfinder untersucht Sachverhalte neutral und ist gut im Analysieren. Es liegt ihm jedoch nicht, eigene Ideen einzubringen und andere Menschen zu motivieren.
Bewerter	Ein Bewerter ist ein disziplinierter und hart arbeitender Mensch, der Problemlösungen pragmatisch angeht. Jedoch kann er sich nicht schnell auf sich ändernde Situationen einstellen und ungeprüfte Ideen akzeptieren.
Moderator	Ein Moderator ist ein selbstsicherer Mensch, der kaum Vorurteile hat und ein ruhiges Wesen besitzt. Er kann andere Menschen gut in das Teamgeschehen einbinden und besitzt eine starke Wahrnehmungskraft. Er verfügt jedoch nicht über das übliche Maß an Kreativität.

Menschen lassen sich nicht eindeutig den verschiedenen Menschentypen zuordnen. Hierfür ist jeder Mensch zu einzigartig. Jedoch sind bei jedem Menschen Tendenzen zu erkennen. Da in jedem Team unterschiedliche Individuen zusammenkommen, erhält jede Architektur ihr einzigartiges Gesicht. Die Teamzusammensetzung sagt somit viel über den Erfolg eines Teams aus (siehe Abschnitt 7.4).

Zuordnung ist nie eindeutig

7.4 Individuen und Gruppen

Bei der Entwicklung eines IT-Systems respektive bei der Gestaltung einer Architektur kommen für die Dauer eines Projekts unterschiedliche Individuen zusammen, um gemeinsam das gestellte Ziel, die Realisierung des IT-Systems, zu erreichen. Individuen schließen sich somit zu einer Gruppe zusammen und übernehmen darin unterschiedliche Rollen und damit verbundene Aufgaben (siehe Abbildung 7.4-1).

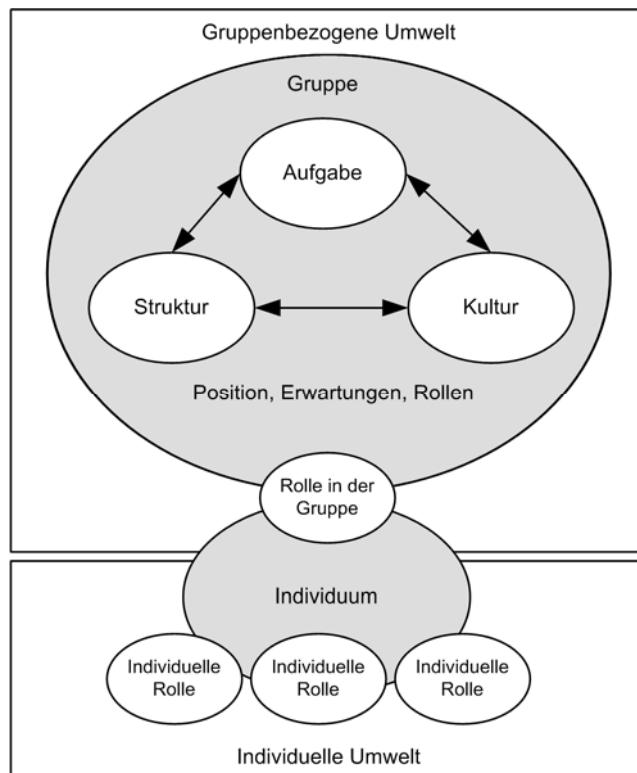


Abb. 7.4-1: Die Gruppe als System [Steiger und Lippmann 2003].

Gruppe als besondere Organisationsform

Eine Gruppe kann als eine besondere Form einer Organisation angesehen werden. Sie interagiert mit ihrer Umwelt, verfolgt eine Aufgabe, besitzt eine Struktur und entwickelt eine Kultur (siehe Abschnitt 7.2). Ein wichtiger Aspekt ist hierbei, dass jedes Individuum eine oder mehrere Rollen in der Gruppe wahrnimmt. Dies können zum einen die formellen Rollen, wie Architekt, Designer oder Entwickler, sein und zum anderen informelle Rollen, wie Spaßmacher oder Sündenbock. Über die Rollen definiert eine Gruppe ihre Erwartungen an Gruppenmitglieder.

Jeder Mensch besitzt dabei ein eigenes Rollenverständnis und erfüllt diese Erwartungen auf seine individuelle Art und Weise.

Darüber hinaus existiert der Mensch nicht nur in der Gruppe, sondern in seiner individuellen Umwelt. In dieser Umwelt nimmt er weitere Rollen wahr, wie z. B. Vater, Ehemann oder Freund, die nicht auf das Ziel der Gruppe ausgerichtet sind. Aufgrund dessen müssen Menschen auch die Erwartungen ihrer individuellen Umwelt befriedigen. Es ist somit essenziell, sich zu vergegenwärtigen, dass Gruppenmitglieder auch ein Leben außerhalb der Gruppe führen und ihnen hierzu genügend Freiraum eingeräumt werden muss. Dadurch wird letztlich auch die Erfolgswahrscheinlichkeit der Gruppe erhöht, da sich die Menschen in der Gruppe wohl- und nicht unter Druck gesetzt fühlen. Edward Yourdon empfiehlt hierzu Projektleitern ausdrücklich sicherzustellen, dass das individuelle Umfeld jedes Teammitglieds nicht beeinträchtigt wird. [Yourdon 1997].

Individuelle Umwelt

Die Erfahrung zeigt, dass die Art der Gruppenzusammensetzung wesentlich für deren Erfolg ist. Überraschenderweise zeigen Untersuchungen, dass Gruppen, die aus hochintelligenten, analytischen und mentalstarken Menschen bestehen, sogenannte Apollo-Teams, bei der Zielerreichung schlechter abschneiden als heterogene Gruppen. Ein Grund für die schlechte Effizienz von Apollo-Teams ist der Wunsch jedes Teammitglieds, seine Ideen durchzusetzen. Ferner achten Teammitglieder von Apollo-Teams seltener auf die Arbeit und Ideen anderer [Belbin 1993]. Belbin nennt folgende Faktoren für gut funktionierende, erfolgreiche Teams:

- > Gruppe wird durch einen guten, kooperativen Moderator geleitet.
- > Ein bis zwei Ideengeber sind in der Gruppe als absolute Erfolgsvoraussetzung vorhanden.
- > Teammitglieder sind ihren Fähigkeiten entsprechend eingesetzt.

Gruppenzusammensetzung und Erfolg

Gruppen sollten somit eine heterogene Gruppenstruktur aufweisen. Dies gilt einerseits für die Charaktereigenschaften und andererseits auch für die Fähigkeiten und den Erfahrungshintergrund. Um das Verständnis für die Problemstellung zu verbessern, schlägt Cockburn z. B. vor, Teams aus Analysten, Designern und Entwicklern zusammenzusetzen. Dies stellt sicher, dass das Geschäftsproblem, beispielsweise die Bewertung der Bonität von Unternehmen, im Team ebenso verstanden wird wie die entwurfs- und umsetzungsbezogenen Probleme [Cockburn 1996].

Heterogene Gruppenstruktur

Gruppenidentifikation und Erfolg

Neben einer heterogenen Gruppenzusammensetzung ist auch die Identifizierung mit der Gruppe ein kritisches Erfolgsmerkmal. Wenn Gruppen eingespielt sind und sich die einzelnen Gruppenmitglieder kennen, sind sie in der Lage, ein Vielfaches von dem uneingespielter Gruppen zu leisten [Cockburn 2002]. Dies hängt damit zusammen, dass sich die Gruppe bereits geformt, also eine Struktur beziehungsweise Hierarchie gebildet hat und sich nicht erst wieder finden muss, bevor sie sich an die Erreichung des Gruppenziels macht.

Gruppendynamik

Dieser Prozess zur Gruppenbildung wird in Tabelle 7.4-1 dargestellt. Dabei berücksichtigt das Modell den gesamten Lebenszyklus einer Gruppe von ihrer Bildung bis hin zu ihrer Neubildung.

Tab. 7.4-1: Gruppendynamik nach Tuckmann [Stahl 2002].

Phase	Primäre Aktivität	Gruppenleistung	Mittel
Gründungsphase (englisch: <i>forming</i>)	Sich-Kennenlernen Sich-Einschätzen Einordnung	Abgrenzung (Separation)	Konventionen
Streitphase (englisch: <i>storming</i>)	Sich-Zeigen Sich-Vertreten Auseinandersetzung	Zuspitzung (Amplifikation)	Konflikte
Vertragsphase (englisch: <i>norming</i>)	Sich-Festlegen Sich-Abfinden Einigung	Entscheidung (Selektion)	Vereinbarung
Arbeitsphase (englisch: <i>performing</i>)	Sich-Einbringen Sich-Engagieren Zusammenarbeit	Bewährung (Res-tabilisierung)	Kooperation
Orientierungsphase (englisch: <i>reforming</i>)	Bilanzieren Sich-Besinnen Erfahrungsaustausch	Veränderung (Variation)	Bilanzen

Gründungsphase

In der Gründungsphase lernen sich die Gruppenmitglieder kennen und schätzen einander ein. Es kommt hier bereits zu einer Einordnung der einzelnen Mitglieder. Des Weiteren grenzt sich die Gruppe gegenüber ihrer Umwelt ab. Dem Umgang miteinander liegen klare Konventionen zugrunde. Die Gruppenmitglieder sind höflich, nett und zuvorkommend. Es ist dabei wichtig, dass eine leitende Rolle, z. B. der Architekt, die Aufgabe und Ziele der Gruppe kommuniziert und den organisatorischen Rahmen vorgibt. Die Gruppenmitglieder müssen ein klares Bild von den an sie gestellten Erwartungen erhalten.

Auf Basis des in der Gründungsphase erlangten Gruppenverständnisses entscheidet sich in der Streitphase für jedes Gruppenmitglied, ob es in der Gruppe verbleiben will oder nicht. Ferner versucht jedes Mitglied, eine für sich adäquate Position innerhalb der Gruppe zu erhalten. Diese kann von der zugesagten Position respektive Rolle durchaus abweichen. Deshalb stehen Meinungsunterschiede, Konkurrenzverhalten und Konfrontationen in dieser Phase im Vordergrund. Diese Auseinandersetzung muss von der Gruppenleitung zugelassen werden, um das Sich-Finden der Gruppe zu ermöglichen. Jedoch sollte sie auf die in der Gründungsphase vereinbarten Regeln hinweisen und die Konfrontation auch nur in diesem Rahmen gestatten.

Streitphase

Die Vertragsphase schließt sich an die Streitphase an. In dieser Phase findet sich die Gruppe. Die Gruppenmitglieder identifizieren sich mit der verhandelten Rolle und einigen sich auf Regeln der Zusammenarbeit. Die Gruppe entwickelt eine Identität und es entsteht ein Wir-Gefühl. Ab diesem Zeitpunkt sollten Aufgaben delegiert werden, um die Selbstständigkeit der Gruppenmitglieder zu fördern. Dabei sind die Stärken und Schwächen der einzelnen Mitglieder zu beachten (siehe Abschnitt 7.3).

Vertragsphase

Die Arbeitsphase kennzeichnet sich durch das Engagement jedes Teammitglieds. Aufgrund des entwickelten Wir-Gefühls erfolgt eine zielorientierte, gemeinschaftliche Zusammenarbeit. Zu diesem Zeitpunkt hat sich die Gruppe eingespielt und ihr Leistungsvermögen erreicht. In dieser Phase sollten die Selbstständigkeit der Gruppe gewährleistet werden und störende Einflüsse von der Gruppe ferngehalten werden. Allerdings bedeutet dies nicht, dass Informationen von außen, die z. B. das Ziel der Gruppe in einem neuen Licht erscheinen lassen, abgeblockt werden sollen. Im Rahmen der Entwicklung eines IT-Systems können dies neue Anforderungen sein, die selbstverständlich berücksichtigt werden müssen.

Arbeitsphase

Nach der Erreichung des Gruppenziels erfolgt in der Orientierungsphase ein Besinnen auf die Leistung und Erlebnisse der Gruppe. Ferner kann sich die Gruppe auflösen oder auf ein neues Ziel ausrichten. Da die Phasen bis zur Arbeitsphase für die Gruppe sehr intensiv und langwierig sein können, empfiehlt es sich jedoch, ein eingespieltes Team beizubehalten und auf ein neues Ziel auszurichten.

Orientierungsphase

Die beschriebenen Phasen werden in der Regel mehrfach durchlaufen, bis sich die Gruppe gefunden und eingespielt hat. Dies gilt vor allem für die ersten drei Phasen des Modells.

Spiralförmiger Ablauf

7.5 Architektur und Entscheidungen

Treffen von Entscheidungen als Schlüsselaufgabe

Die Schlüsselaufgabe eines Architekten ist das Treffen von Entscheidungen. In diesem Zusammenhang stellt sich jedoch die Frage, was sind überhaupt Entscheidungen und was sind letztlich architektonisch relevante Entscheidungen, mit denen sich ein Architekt beschäftigen muss? Dieser Abschnitt widmet sich diesem Themenkomplex, indem zunächst das Thema *Entscheiden* allgemein betrachtet und anschließend auf Architektur übertragen wird.

Was ist eine Entscheidung?

Nach Vetter kann eine Entscheidung wie folgt definiert werden [Steiger und Lippmann 2003]:

1. Eine Entscheidung ist die Wahl einer Handlungs- oder Reaktionsmöglichkeit in einer Situation, in der mehrere Möglichkeiten bestehen.
2. Eine Entscheidung ist ein Schritt im Rahmen einer Problemlösung, bei dem nach der Bewertung von Handlungsalternativen eine Alternative ausgewählt wird.

Diese Definition hebt zwei wichtige Merkmale einer Entscheidung hervor. Zum einen bestehen stets mehrere Möglichkeiten, für die man sich entscheiden kann. Zum anderen ist die Wahl einer dieser Möglichkeiten, also die eigentliche Entscheidung, immer eingebettet in einen Problemlösungs- beziehungsweise Entscheidungsprozess.

Architektur als Resultat von Entscheidungen

Wenn man Architektur als das Resultat einer Reihe von Entscheidungen ansieht, wird auch deutlich, warum Architektur immer ein Kompromiss ist. Ein Architekt steht vor der Aufgabe, eine Architektur zu entwerfen, die die an sie gestellten funktionalen und nicht-funktionalen Anforderungen berücksichtigt. Dabei wird er entscheiden müssen, welche Anforderungen er priorisiert und welche er eher vernachlässigt. Er muss somit aus einer Reihe von Möglichkeiten wählen und dadurch einen Kompromiss eingehen.

Architektur als Resultat eines Entscheidungsprozesses

Architektur ist ferner das Ergebnis eines Entscheidungsprozesses. Unabhängig davon, nach welchem Vorgehen (siehe Kapitel 8) ein Architekt handelt, lässt sich das Treffen von Entscheidungen in folgende Schritte unterteilen [Steiger und Lippmann 2003]:

> Entscheidung vorbereiten

Um eine Entscheidung treffen zu können, muss diese vorbereitet werden. Mit anderen Worten sind die Ziele, die man erreichen möchte, zu definieren. Im Sinne von Architektur ist dies die Erfüllung der funktionalen und nicht-funktionalen Anforderungen. Des

Weiteren sind Restriktionen, die durch die Umwelt induziert werden, zu identifizieren und zu berücksichtigen. Ein Beispiel hierfür sind Standards hinsichtlich Plattformen, wie JEE oder .NET, die in einem Unternehmen (Umwelt) eingehalten werden müssen. Auf Basis der Anforderungen und der Restriktionen muss sich der Architekt die verschiedenen Möglichkeiten respektive Lösungsvarianten vergegenwärtigen. Es ist dabei empfehlenswert, bereits im Rahmen der Entscheidungsvorbereitung mit betroffenen Personen und Abteilungen zu kommunizieren, um zum einen Informationen zu erlangen und zum anderen die Akzeptanz verschiedener Möglichkeiten zu eruieren.

> *Entscheidung treffen*

In diesem Schritt gilt es, die festgehaltenen Möglichkeiten gegenüberzustellen und sich für eine der Möglichkeiten zu entscheiden. Jede Lösungsvariante adressiert die Anforderungen und Restriktionen unterschiedlich. Die Herausforderung besteht in diesem Schritt darin, für die jeweilige Problemstellung die Variante zu selektieren, die die Anforderungen und Restriktionen zweckmäßig ausbalanciert. Ist in einem Unternehmen z. B. JEE als Standard gesetzt, so gilt es für den Architekten immer noch abzuwegen, welcher Anwendungsserver einzusetzen ist. Dabei sind neben benötigten Diensten (JSP, Java Servlets, EJB etc.) auch Kostenaspekte in die Entscheidung einzubeziehen. Es kann somit zweckmäßig sein, einen Open-Source-Anwendungsserver einzusetzen, um den Budgetrahmen nicht zu sprengen. Die möglichen Entscheidungsalternativen, die letztlich getroffene Entscheidung sowie die Gründe für die gefällte Entscheidung sind entsprechend zu dokumentieren. Dies stellt sicher, dass man zu einem späteren Zeitpunkt die Entscheidung nachvollziehen kann.

> *Entscheidung kommunizieren*

Eine Entscheidung muss aktiv kommuniziert werden, um das durch die Entscheidung angestrebte Ziel zu erreichen. Es genügt nicht, eine Entscheidung im stillen Kämmerlein zu treffen und diese erst auf Nachfrage zu kommunizieren (siehe Abschnitt 7.2).

> *Entscheidung realisieren*

Neben der Kommunikation der Entscheidung ist es natürlich auch wichtig, diese zu realisieren.

> *Entscheidung beurteilen*

Jeder Entscheidung sollte im Nachhinein beurteilt werden, um die Zweckmäßigkeit der Entscheidung zu verifizieren und um aus möglichen Fehlern zu lernen.

Informationsmenge und Entscheidungen

Im Rahmen des Entscheidungsprozesses ist es wichtig, Informationen zu sammeln, auf deren Grundlage Entscheidungen getroffen werden können. Hier stellt sich die Frage, wie viel Informationen benötigt werden, um zweckmäßige Entscheidungen zu fällen. Man könnte annehmen, dass Entscheidungen umso leichter fallen, je mehr Informationen vorliegen. Psychologische Untersuchungen zeigen jedoch, dass die Entscheidungsfreudigkeit mit zunehmender Informationsmenge sogar deutlich sinken kann [Dörner 1989]. Dies ist letztlich gar nicht so unverständlich, wie es zunächst scheint. Damit man keine Entscheidung aus Unsicherheit heraus trifft, sammelt man Informationen über den zu entscheidenden Sachverhalt. Je mehr man jedoch über den Sachverhalt weiß, umso schwieriger wird die eigentliche Entscheidung, da die Komplexität des Sachverhalts zunimmt. Infolgedessen können Entscheidungen auf purem Aktionismus beruhen, weil man die Informationsflut nicht mehr überblicken kann oder Entscheidungen werden verschleppt, da sich eine Entscheidungsunfähigkeit eingestellt hat. Um diesem Problem zu begegnen, sollte die Menge an Informationen einer kritischen Masse entsprechen, das heißt, gerade so groß sein, dass eine zweckmäßige Entscheidung getroffen werden kann [Rüping 2004]. Es ist allerdings relativ schwierig, diese kritische Masse zu bestimmen. Hierbei kann der Architekt oftmals nur auf seine Erfahrung vertrauen. Die Berücksichtigung von Architektur-Mitteln, wie z. B. Architekturstilen und -Mustern können die Entscheidungsfindung erleichtern.

Entscheidungsarten und -methoden

Entscheidungen können auf *rationale* oder *intuitive* Art und Weise getroffen werden. Rationale Entscheidungen beruhen auf systematischem Vorgehen und dem analytischen Vorbereiten von Entscheidungen. Hierbei werden Entscheidungen in kleinere, überschaubarere Teilentscheidungen unterteilt. Die Summe der Teilentscheidungen ergibt dann die Gesamtentscheidung [Steiger und Lippmann 2003]. Einige Methoden, die zum Treffen rationaler Entscheidungen herangezogen werden, sind:

- > Entscheidungsbäume [Vroom und Yetton 1976]
- > Lineare Programmierung
- > Nutzwertanalyse
- > Metaplan-Technik

Bei der intuitiven Entscheidung lässt man sich von seinem Gefühl leiten und vertraut auf seine Intuition. Es erfolgt eine ganzheitliche Betrachtung des zu entscheidenden Sachverhalts und man fällt auf Basis seines Gefühls und seiner Erfahrung eine Entscheidung. Tony Bouzan, der Erfinder der Mind-Mapping-Methode, bezeichnet diese Art der Entschei-

dung als Superlogik. Das Gehirn analysiert die Problemstellung intuitiv und liefert ein Gefühl, mit dem man „aus dem Bauch heraus“ entscheiden kann. Die Relevanz dieser Entscheidungsmethode wird durch eine Untersuchung der Harvard Business School untermauert, nach der Angestellte und Vorstände von Unternehmen 80 % ihres Erfolgs auf intuitives Handeln zurückführen [Bouzan und Bouzan 1997].

In der Literatur wird zwischen strategischen und operationalen Entscheidungen unterschieden [Steiger und Lippmann 2003]. Strategische Entscheidungen haben längerfristigen Charakter und umfassende Bedeutung. Dahingegen betreffen operationale Entscheidungen kurzfristige Belange und haben ein geringeres Ausmaß.

Architektonische Entscheidungen sind von strategischer Natur, da sie ein System maßgeblich prägen und langfristig beeinflussen. Sie betreffen also das Gesamtsystem im jeweiligen Kontext.

Architekten müssen Entscheidungen in verschiedenen Bereichen treffen. Zum einen sind an dieser Stelle die Entscheidungen zu nennen, die die Architektur des Systems unmittelbar betreffen, und zum anderen die Entscheidungen, die eher von organisatorischer Natur sind. Zu Letzteren gehört z. B. die Auswahl von geeigneten Mitarbeitern, um die Realisierung des Systems sicherzustellen. Entscheidungen, die das System an sich betreffen, lassen sich aus dem in Abschnitt 3.4 eingeführten Modell ableiten. Abschließend werden typische Entscheidungen vorgestellt, die ein Architekt im Hinblick auf die konkrete Architektur eines Systems fällen muss.

Ein Architekt muss entscheiden,

- › aus welchen Subsystemen das zu realisierende System besteht und welche Verantwortlichkeiten die jeweiligen Subsysteme besitzen;
- › wie das zu realisierende System auf unterschiedliche Schichten organisiert ist und welche Subsysteme auf welchen Schichten angeordnet sind;
- › welche Beziehungen zwischen den Subsystemen bestehen und wie die Subsysteme miteinander kommunizieren sollen;
- › welche architekturtragenden Bausteine im System vorzusehen sind und wie die Schnittstellen dieser Bausteine aussehen;
- › auf welcher Plattform das System betrieben wird und welche Dienste benötigt werden;
- › wie Software-Bausteine auf der Plattform installiert und über die Hardware-Bausteine verteilt werden;

Strategische und operative Entscheidungen

Charakter architektonischer Entscheidungen

Entscheidungsbereiche

Beispiele architektonischer Entscheidungen

- > welche konkreten Hardware-Bausteine vorzusehen sind und wie diese zu dimensionieren sind.

7.6 Architekt als zentrale Rolle

Architekt als zentrale Rolle

Ein Architekt ist in vielfältige Aufgaben eingebunden und kommuniziert mit unterschiedlichen Interessenvertretern. Er ist in der Regel schon im Rahmen von Vorstudien, beispielsweise zur Verifizierung der Machbarkeit eines IT-Vorhabens, über die Analyse-Phase eines Projekts bis hin zur Inbetriebnahme eines IT-Systems involviert. Währenddessen interagiert er mit vielen verschiedenen Rollen, wie in Abbildung 7.6-1 illustriert.

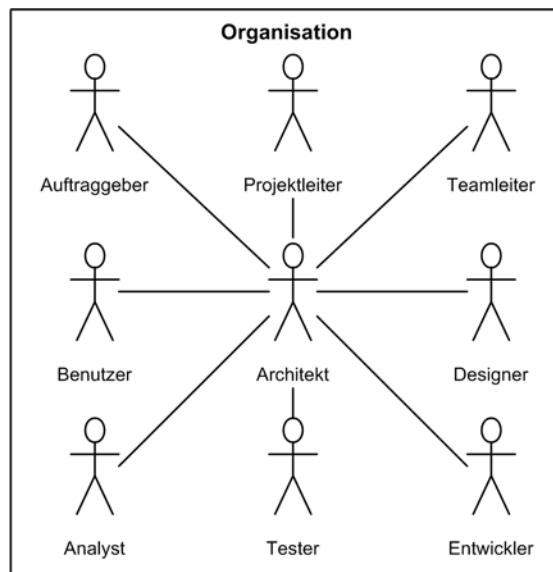


Abb. 7.6-1: Architekt im organisatorisch sozialen Umfeld.

Architekt als Gestalter

Aufgrund dieses zentralen Charakters ist ein Architekt mehr als ein Gestalter einer Architektur, obwohl dies eine seiner wichtigsten Aufgaben ist. Hierbei bringt er all seine Erfahrung ein, identifiziert die architektonisch relevanten Anwendungsfälle, berücksichtigt Architektur-Prinzipien, wählt passende Architektur-Stile aus und adaptiert erprobte Referenzarchitekturen. Dies ist jedoch nur ein Teil seines Aufgabenfeldes.

Architekt als Problemlöser

Durch die Architektur-Gestaltung agiert ein Architekt als Problemlöser, indem er die architektonische Basis für die Erfüllung von funktionalen und nicht-funktionalen Anforderungen schafft.

Mit anderen Worten beantwortet er die in Abschnitt 7.5 vorgestellten klassischen, architektonischen Fragen und trifft strategische Entscheidungen. Von einem Architekten wird somit auch eine Entscheidungsfähigkeit in unsicheren Situationen erwartet.

Architekt als Entscheider

Darüber hinaus muss er als Visionär der Architektur agieren. Er darf die Vision, die durch die Architektur verwirklicht werden soll, nicht aus den Augen verlieren. Vielmehr muss er sicherstellen, dass jeder Beteiligte stets die Grundprinzipien der Architektur kennt und sein Handeln entsprechend ausrichtet. Dies gilt sowohl in Richtung Kunde als auch in Richtung Team.

Architekt als Visionär

Deshalb muss ein Architekt auch über ausgesprochen kommunikative Fähigkeiten verfügen. Er muss aktiv auf Menschen zugehen, Informationen vermitteln und für einen adäquaten Wissensstand aller Beteiligten hinsichtlich der Architektur sorgen. Zu diesem Zweck muss er in der Lage sein, Ideen zu kommunizieren und zielgruppengerichtet zu präsentieren.

Architekt als Kommunikator

Neben der Vermittlung von Informationen sollte ein Architekt natürlich auch offen für Ideen und Fragen anderer sein. Er sollte somit stets aktiv zuhören und beispielsweise auf Vorschläge seiner Teammitglieder eingehen. Die Teammitglieder sind die Spezialisten in ihrem Bereich und können für ihr Gebiet die Zweckmäßigkeit der Architektur beurteilen. Allerdings darf dabei die Gesamtsicht der Architektur nicht aus den Augen verloren werden. Der Architekt muss immer die Zweckmäßigkeit für die Gesamtarchitektur beurteilen.

Architekt als Zuhörer

Darüber hinaus ist es die Aufgabe eines Architekten, seine Teammitglieder für die gewählte Architektur zu gewinnen. Aus diesem Grund sollte er sich auch als Motivator verstehen. Ein Architekt sollte im Sinne des Schriftstellers Antoine de Saint-Exupéry motivieren:

„Wenn Du ein Schiff bauen willst, so trommle nicht die Männer zusammen, um Holz zu beschaffen und Werkzeuge vorzubereiten oder die Arbeit einzuteilen und Aufgaben zu vergeben, sondern lehre die Männer die Sehnsucht nach dem endlosen, weiten Meer.“

Architekt als Motivator

Aus De Saint-Exupérys Worten wird deutlich, dass richtige Motivation nur geweckt werden kann, wenn Menschen auf ein Ziel hinarbeiten, mit dem sie sich voll und ganz identifizieren können. Für einen Architekten bedeutet dies, dass er seine Teammitglieder mit ins Boot holen muss. Dies kann durch die Delegation von Verantwortlichkeiten und die Be-

rücksichtigung von Ideen und Vorschlägen anderer gelingen. Ferner sollte jedem Teammitglied seine Nützlichkeit und Wichtigkeit klar kommuniziert werden, um seine Zufriedenheit zu steigern.

Architekt als Leiter

Ein Architekt ist auch der technische Leiter eines IT-Vorhabens. Er muss somit Führungsqualitäten besitzen. Einige dieser Qualitäten, wie die Motivation seines Teams, wurden bereits explizit erwähnt. Für die Ausübung seiner Führungsrolle ist es wichtig, einen für die jeweilige Situation und das jeweilige Individuum geeigneten Führungsstil anzuwenden. Sind Teammitglieder beispielsweise hochqualifiziert und motiviert, sollten Aufgaben delegiert werden. Ein delegativer Führungsstil zeichnet sich dadurch aus, dass der Architekt das Problem aufzeigt und die Grenzen des Entscheidungsspielraums festlegt. Das Team respektive das Individuum entscheidet selbstständig im Rahmen seines Entscheidungsspielraums. Bei Teammitgliedern, die zwar hoch motiviert, jedoch noch nicht über ausreichende Erfahrung verfügen, sollte der Architekt bei der Entscheidungsfindung unterstützend einwirken. Dies kann in seiner Rolle als Zuhörer und durch das Beantworten von Fragen geschehen. Der Entscheidungsspielraum ist dabei durch die Architektur vorgegeben, indem z. B. Architektur-Prinzipien festgelegt und Subsysteme inklusive ihrer Verantwortlichkeiten sowie der Art und Weise der Kommunikation beschrieben werden.

Architekt als Praktiker

Ein Architekt soll jedoch nicht nur Vorgaben entwerfen, sondern auch aktiv mitarbeiten, da dies signifikant zum Architektur-Erfolg beiträgt. Ein Architekt kann hierdurch unmittelbar die Realisierbarkeit architektonischer Ideen abschätzen. Darüber hinaus wird die Wahrscheinlichkeit erhöht, dass die Teammitglieder die Architektur verstehen, weil eine unmittelbare Kommunikation zwischen Architekt, Designern und Entwicklern entsteht. Ferner wird hierdurch die Akzeptanz des Architekten durch die Teammitglieder gesteigert, da eine direkte Zusammenarbeit soziale Barrieren abbaut [Ambler 2002]. Aus diesem Grund sollte sich ein Architekt immer auch als Praktiker verstehen. Das Organisationsmuster *ArchitectAlsoImplements* formuliert diese Anforderung [Coplien und Harrison 2004].

Architekt als Generalist

Ein Architekt sollte eine breite Wissensbasis besitzen, die es ihm erlaubt, Zusammenhänge zu erkennen, zu verstehen und daraus Konsequenzen abzuleiten. Im Hinblick auf die in Kapitel 3 vorgestellten Architektur-Disziplinen bedeutet dies, dass er über generelles Wissen in den einzelnen Disziplinen verfügt. Man kann ihn somit als Generalisten bezeichnen, der für tiefer gehende Fragen und Problemstellungen auf das Spezialwissen seiner Teammitglieder zurückgreift. Neben diesen

architekturbezogenen Wissensgebieten muss der Architekt jedoch auch über wesentliche Kenntnisse des Problembereichs verfügen, für den das IT-System entwickelt wird, um die Wünsche und Probleme des Auftraggebers und der Benutzer zu verstehen. Ferner sind Projektmanagementkenntnisse erforderlich, um im Zusammenspiel mit dem Projektleiter den Projektplan zu entwerfen. Nicht zuletzt ist auch fundiertes Wissen im Testen von Systemen notwendig, um die Testpläne und -szenarien mit Testern zu koordinieren.

Die geschilderten Kenntnisse eines Architekten sind sehr umfangreich, jedoch wesentlich für dessen Erfolg. Vitruvius umschrieb die Kompetenzen eines Architekten wie folgt:

The ideal architect should be a man of letters, a mathematician, familiar with historical studies, a diligent student of philosophy, acquainted with music, not ignorant of medicine, learned in the responses of juriconsults, familiar with astronomy and astronomical calculations.

Darin kommt die von einem Architekten erwartete, breite Wissensbasis zum Ausdruck, die sich über fachliche, methodische und soziale Kompetenzen erstreckt.

Die vorgestellten Aufgaben eines Architekten sind sehr umfangreich. Hier stellt sich die Frage, ob diese Aufgaben überhaupt durch eine Person alleine erfüllt werden können. Je nach den individuellen Stärken und Schwächen wird ein Mensch die Aufgaben besser oder schlechter erfüllen können und ggf. mit allen Aufgaben überfordert sein (siehe Abschnitt 7.3). Aus diesem Grund spricht sehr viel für die Etablierung von Architektenteams, deren Mitglieder sich entsprechend ergänzen.

Einige Firmen haben die von einem Architekten erwarteten Kompetenzen konkretisiert und dedizierte Anforderungsprofile für Architekten entwickelt. Ein Beispiel ist das Enterprise Architecture Skills Framework der Open Group [Jones 2004]. Die Open Group unterscheidet dabei zwischen folgenden Wissensbereichen:

- > Allgemeines Wissen (englisch: *generic skills*) ist auf die Sozialkompetenz von Architekten ausgerichtet und behandelt Themen wie Führung und Teamarbeit.
- > Geschäftswissen und -methoden (englisch: *business skills and methods*) vermittelt z. B. Geschäftsprozesse und strategische Planung.
- > Enterprise-Architektur-Wissen (englisch: *enterprise architecture skills*) vermittelt Themen rund um Enterprise-Architektur.

Kompetenzen eines Architekten

Architektenteams

Dedizierte Entwicklungspfade und Anforderungsprofile

- > Programm- oder Projektmanagement-Wissen (englisch: *program or project management skills*) fördert die Methodenkompetenz im Rahmen von Projekten.
- > Allgemeines IT-Wissen (englisch: *IT general knowledge skills*).
- > Technisches IT-Wissen (englisch: *technical IT skills*) beinhaltet elementare Themen wie Software-Entwicklung und Datenmodellierung.
- > Rechtliches Wissen (englisch: *legal environment*).

Architektur benötigt Erfahrung

Ein entsprechendes Entwicklungsprogramm ist eine nützliche Grundlage für die Weiterentwicklung als Architekt. Architektur ist jedoch immer das Resultat von Erfahrung. Aus diesem Grund muss man offen sein für Neues und erkannt haben, dass man nie auslernen wird. Jedes neue Architektur-Vorhaben ist somit eine Chance, um sich weiterzuentwickeln und seinen Erfahrungsschatz zu vergrößern

Zertifizierungen als Leistungsnachweis

Seit einigen Jahren bieten verschiedene Organisationen und Dienstleister, wie die Open Group, SUN, Microsoft, SAP etc., Architektur-Zertifikate an. Diese werden im Markt zunehmend als Leistungsnachweis anerkannt. Das *eine* standardisierte hersteller- und technologieunabhängige sowie international anerkannte Architektur-Zertifikat gibt es jedoch noch nicht. Als Folge der oft noch herrschenden Orientierungslosigkeit zum Thema Software-Architektur (siehe Kapitel 1) existiert keine einheitliche Vorstellung über das Berufsbild des Software-Architekten. Auch dadurch bedingt sind Inhalte, Aufbau und Umfang der Zertifizierungsprogramme sehr unterschiedlich. Kosten- (von 1.000 EUR bis über 20.000 EUR) und Zeitaufwand (von wenigen Tagen bis mehreren Monaten) für die Erlangung eines Architektur-Zertifikats variieren ebenfalls deutlich. Grundsätzlich werden gute Kenntnisse im Software-Engineering und teilweise umfangreiche Berufserfahrung (10 Jahre und mehr) für die Teilnahme an Zertifizierungsprogrammen vorausgesetzt. Zudem ist vor der Teilnahme bei einigen Zertifizierungsprogrammen ein Zulassungsverfahren zu durchlaufen. Abschlussprüfungen können Multiple-Choice-Tests und/oder Abschlussprojekt(e) umfassen. An dieser Stelle wird kurz auf die hersteller- und technologieunabhängigen Open-Group-Zertifizierungen eingegangen. Die Open Group unterscheidet drei Zertifizierungslevels:

- > *Level 1: Certified IT Architect*
- > Ein Level-1-zertifizierter IT-Architekt kann unter Aufsicht als Architekt agieren. Er verfügt über eine weite Bandbreite von notwendigem Architektur-Wissen.
- > *Level 2: Master Certified IT Architect*

- > Ein Level-2-zertifizierter IT-Architekt ist in der Lage, selbstständig Architekturen zu entwerfen und sich für diese verantwortlich zu zeichnen.

- > *Level 3: Distinguished Certified IT Architect*

Ein Level-3-zertifizierter IT-Architekt beeinflusst aufgrund seiner breiten und tiefen Architektur-Erfahrung seine Umgebung wesentlich. Er ist in der Regel als Chefarchitekt, Enterprise-Architekt oder Leiter der IT-Architektursparte seines Unternehmens tätig.

Die Kompetenz steigt von Level 1 zu Level 3. Weitere Informationen bei [Opengroup 2008b].

7.7 Zusammenfassung

- > Organisationen, Individuen und Architekturen beeinflussen sich gegenseitig.
- > Conways Gesetz besagt, dass Organisationen darauf beschränkt sind, Systeme zu entwerfen, deren Strukturen Kopien ihrer organisatorischen Kommunikationsstrukturen sind.
- > Architekturen beziehungsweise darauf basierende IT-Systeme werden stets durch und für Menschen entworfen.
- > Die architektonische Tätigkeit erfolgt stets eingebettet in eine Organisation, sei es nun das Unternehmen, für das das IT-System entworfen wird, oder die Projektorganisation, die die involvierten Personen vereint.

Zusammenfassung: Allgemeines

- > Eine Organisation ist ein soziales Gebilde, das dauerhaft ein Ziel verfolgt und eine formale Struktur aufweist, mit deren Hilfe Aktivitäten der Mitglieder auf das verfolgte Ziel ausgerichtet werden sollen.
- > In der Organisationslehre unterscheidet man zwischen dem ökonomisch-rationalem, dem verhaltenswissenschaftlichen und dem systemischen Verständnis.
- > Das ökonomisch-rationale Verständnis, auch Taylorismus genannt, betrachtet das Individuum als Produktionsfaktor, der planbar, vorhersehbar und steuerbar ist.
- > Das verhaltenswissenschaftliche Verständnis rückt den Menschen in den Mittelpunkt der Betrachtung, indem es ihn nicht mehr als reinen Produktionsfaktor, sondern als soziales Wesen wahrnimmt, das nach Anerkennung und Wertschätzung strebt.
- > Das systemische Verständnis betrachtet eine Organisation als ein System, welches zur Erreichung eines Ziels existiert und hierzu mit

Zusammenfassung: Organisationen

	<p>seiner Umwelt interagiert. Organisationen besitzen ein eigenes Aufgabenverständnis, eine eigene Kultur sowie formelle als auch informelle Strukturen.</p> <ul style="list-style-type: none"> > Agile Entwicklungsprozesse basieren auf dem verhaltenswissenschaftlichen und systemischen Organisationsverständnis. Sie rücken den Menschen ins Zentrum und betrachten ihn als motiviertes Individuum.
Zusammenfassung: Individuen	<ul style="list-style-type: none"> > Jeder Mensch besitzt eine eigene Identität. Die Identität beruht auf fünf wesentlichen Säulen: soziales Netz, Beruf und Arbeit, Leiblichkeit, materielle Werte, Werte und Normen. > Belbin unterscheidet zwischen folgenden Menschentypen: Kooperator, Wegbereiter, Ideengeber, Durchsetzer, Umsetzer, Lückenfinder, Bewerter und Moderator. > Menschen lassen sich nicht eindeutig den verschiedenen Menschen-typen zuordnen. Jedoch sind bei jedem Menschen Tendenzen zu erkennen.
Zusammenfassung: Individuen und Gruppen	<ul style="list-style-type: none"> > Eine Gruppe kann als eine besondere Form einer Organisation angesehen werden. Sie interagiert mit ihrer Umwelt, verfolgt eine Aufgabe, besitzt eine Struktur und entwickelt eine Kultur. > Belbin nennt folgende Faktoren für gut funktionierende, erfolgreiche Teams: <ul style="list-style-type: none"> > Gruppe wird durch einen guten, kooperativen Moderator geleitet. > Ein bis zwei Ideengeber sind in der Gruppe als absolute Erfolgsvoraus-setzung vorhanden. > Teammitglieder sind ihren Fähigkeiten entsprechend eingesetzt. > Gruppen sollten eine heterogene Gruppenstruktur aufweisen. Dies gilt einerseits für die Charaktereigenschaften und andererseits auch für die Fähigkeiten und den Erfahrungshintergrund. > Nach Tuckmann unterläuft jede Gruppe folgende Phasen: Gründungsphase (englisch: <i>forming</i>), Streitphase (englisch: <i>storming</i>), Ver-tragsphase (englisch: <i>norming</i>), Arbeitsphase (englisch: <i>performing</i>), Orientierungsphase (englisch: <i>reforming</i>) > Die beschriebenen Phasen werden in der Regel mehrfach durchlaufen, bis sich die Gruppe gefunden und eingespielt hat.
Zusammenfassung: Architektur und Entscheidungen	<ul style="list-style-type: none"> > Die Schlüsselaufgabe eines Architekten ist das Treffen von Entschei-dungen. > Eine Entscheidung ist die Wahl einer Handlungs- oder Reaktionsmög-ligkeit in einer Situation, in der mehrere Möglichkeiten bestehen. Sie

ist ein Schritt im Rahmen einer Problemlösung, bei dem nach der Bewertung von Handlungsalternativen eine Alternative ausgewählt wird

- > Architektur ist das Resultat von Entscheidungen.
- > Entscheidungen sind das Ergebnis eines Entscheidungsprozesses, der aus folgenden Phasen besteht: Entscheidung vorbereiten, treffen, kommunizieren, realisieren und beurteilen.
- > Die zum Treffen einer Entscheidung notwendige Menge an Informationen sollte einer kritischen Masse entsprechen, das heißt, gerade so groß sein, dass eine zweckmäßige Entscheidung getroffen werden kann.
- > Architektonische Entscheidungen sind von strategischer Natur, da sie ein System maßgeblich prägen und langfristig beeinflussen. Sie betreffen also das Gesamtsystem im jeweiligen Kontext.

- > Ein Architekt ist in der Regel schon im Rahmen von Vorstudien, beispielsweise zur Verifizierung der Machbarkeit eines IT-Vorhabens, über die Analyse-Phase eines Projekts bis hin zur Inbetriebnahme eines IT-Systems involviert.
- > Ein Architekt agiert in unterschiedlichen Rollen, wie z. B. Gestalter, Problemlöser, Entscheider, Visionär, Kommunikator, Zuhörer, Motivator, Leiter, Praktiker, Generalist.
- > Architektenteams bestehen aus Mitgliedern, die sich gegenseitig ergänzen, um die vielfältigen Architektur-Aufgaben ideal wahrzunehmen.
- > Seit einigen Jahren bieten verschiedene Organisationen und Dienstleister, wie die Open Group, SUN, Microsoft, SAP etc., Architektur-Zertifikate an.
- > Für die Erlangung eines Architektur-Zertifikats werden grundsätzlich gute Kenntnisse im Software-Engineering und teilweise umfangreiche Berufserfahrung (10 Jahre und mehr) vorausgesetzt.
- > Für die Teilnahme an einigen Zertifizierungsprogrammen ist ein Zulassungsverfahren zu durchlaufen.
- > Abschlussprüfungen von Zertifizierungsprogrammen können Multiple-Choice-Tests und/oder Abschlussprojekt(e) umfassen.

Zusammenfassung: Architekt als zentrale Rolle

8 | Architektur-Vorgehen (WIE)

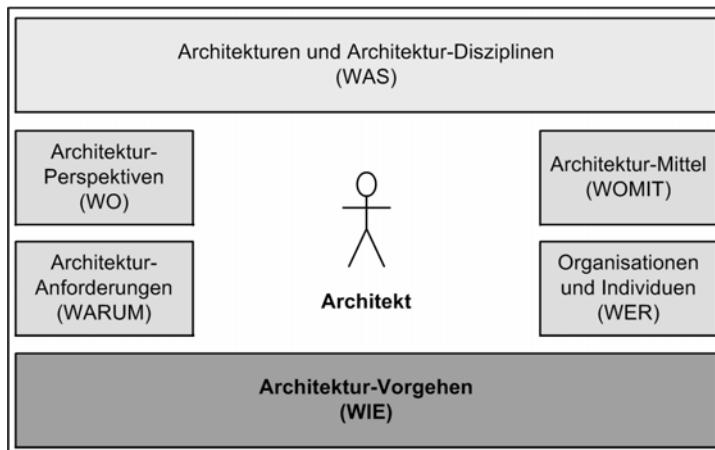


Abb. 8-1: Positionierung des Kapitels im Ordnungsrahmen.

In diesem Kapitel steht die *WIE-Dimension* des Ordnungsrahmens im Mittelpunkt. Zunächst wird für einen Architekten relevantes Wissen zu Entwicklungsprozessen vermittelt. Darauf aufbauend werden die einzelnen Tätigkeiten eines Architekten während der Erarbeitung eines Systems auf einem allgemeingültigen Niveau beschrieben. Abschließend erfolgt eine Konkretisierung dieser Darstellung anhand eines beispielhaften Anwendungsszenarios. Das Anwendungsszenario vernetzt den Ordnungsrahmen sowie den Theorieteil aus dem Kontext eines spezifischen Anwendungsfalls heraus und bietet auf diese Weise dem Leser einen problemorientierten Zugang zu den übrigen Kapiteln.

Übersicht

8.1	Architektur und Entwicklungsprozesse	342
8.2	Architektonisches Vorgehen im Überblick	350
8.3	Erstellen der Systemvision	357
8.4	Verstehen der Anforderungen	367
8.5	Entwerfen der Architektur	377
8.6	Umsetzen der Architektur	406
8.7	Kommunizieren der Architektur	413
8.8	Anwendungsszenario: Enterprise Application Integration	428

Grundlegende Konzepte der WIE-Dimension

Abbildung 8-2 stellt die grundlegenden Konzepte, welche in diesem Kapitel behandelt werden, vor und visualisiert ihren Zusammenhang. Die Abschnitte 8.1 bis 8.7 detaillieren diese Konzepte und präsentieren weitere relevante Konzepte.

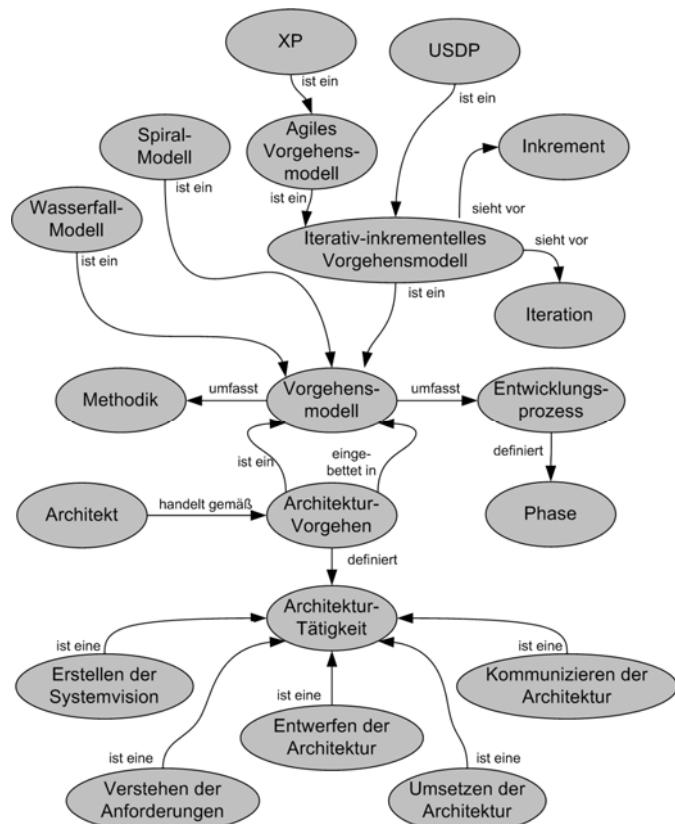


Abb. 8-2: Grundlegende Konzepte der WIE-Dimension.

8.1 Architektur und Entwicklungsprozesse

Der geplante Entwicklungspfad ist nur eine Leitlinie

Die Entwicklung eines Software-Systems erfolgt in den seltensten Fällen entlang dem ursprünglich geplanten Pfad (siehe Abbildung 8.1-1). Bereits zu Beginn kann der Start von der Planung abweichen, weil z. B. wichtige Projektmitarbeiter mit den benötigten Fähigkeiten nicht zur Verfügung stehen. Des Weiteren wird man im Verlauf vom geplanten Pfad abweichen und den Zielbereich nicht am geplanten Zielpunkt erreichen. Dies kann unterschiedliche Gründe haben. Systemanforderungen können sich ändern, wichtige Projektmitarbeiter können das

Projekt verlassen, Lerneffekte (auch für einen Architekten!), die sich „unterwegs“ ergeben haben, oder organisatorische Rahmenbedingungen können Kursänderungen bedingen. Solange auf solche Veränderungen reagiert werden kann und der von den Interessenvertretern vorgegebene Zielbereich trotzdem noch erreicht wird, ist dies kein Problem. Je nach gewähltem Vorgehensmodell kann auf solche Veränderungen unterschiedlich gut reagiert werden. Ein Architekt muss seine Tätigkeiten diesem Sachverhalt anpassen. Die Strukturierung von Kerndisziplinen (Anforderungserhebung, Analyse, Entwurf, Implementation und Test), (Architektur-)Tätigkeiten, (Architektur-)Aktionen und querschnittlichen Aufgaben sowie die Festlegung ihrer zeitlicher Abfolge ist also, insbesondere für große Projekte, unerlässlich und wurde bereits in Abschnitt 6.6.5 motiviert. Daher werden im Folgenden die in der Praxis am häufigsten anzutreffenden Vorgehensmodelle kurz besprochen. Für ein besseres Verständnis sei auf Abschnitt 6.6.5 verwiesen

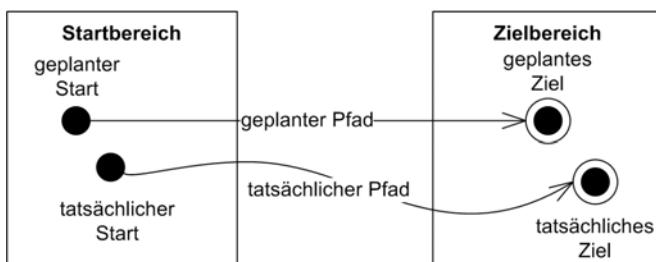


Abb. 8.1-1: Geplanter Entwicklungs- vs. tatsächlichem Entwicklungspfad.

Das Wasserfall-Modell wurde bereits in den siebziger Jahren das erste Mal in einem Artikel diskutiert [Royce 1970]. Es sieht die sequentielle und einmalige Abarbeitung verschiedener Software-Entwicklungsdisziplinen vor (siehe Abbildung 8.1-2). So werden beispielsweise die Disziplinen Anforderungserhebung, Analyse, Entwurf und Implementation nacheinander durchlaufen. Mit anderen Worten beginnt eine Disziplin erst, wenn eine andere vollständig abgeschlossen ist. Dies setzt voraus, dass die Arbeitsergebnisse der jeweiligen Disziplinen vollständig und fehlerfrei sind. Basierend auf den in der Anforderungserhebung dokumentierten Anforderungen werden passende Analyse-Modelle erzeugt und im Entwurf wird festgehalten, wie das System während der Implementation zu realisieren ist. Disziplinen werden in diesem Zusammenhang oft als Phasen bezeichnet.

Das Wasserfall-Modell

Das Wasserfall-Modell ist aufgrund seiner (realitätsfernen) Einfachheit auf den ersten Blick für viele sehr interessant. Einerseits lässt sich ein Entwicklungsprojekt mit diesem Ansatz sehr leicht planen. Dies gilt

Vor- und Nachteile des Wasserfall-Modells

zum einen für den Beginn und das Ende der verschiedenen Disziplinen und zum anderen für den Zeitpunkt, wann Personen für die speziellen Tätigkeiten (z. B. Anwendungsfällerstellung, Programmierung) benötigt werden [Cockburn 2002]. Andererseits birgt dieser Ansatz aber auch zahlreiche zeitlich inhärente Risiken. Beispielsweise erfolgt die Rückkopplung mit den Benutzern des Systems erst sehr spät und zwar während der Testphase im Entwicklungsprozess. Damit ist es nicht möglich, auf Fehler des Systems und auf geänderte Anforderungen zu reagieren. Ferner werden die erstellten Arbeitsergebnisse (z. B. der Entwurf) Lücken aufweisen, widersprüchlich oder schlicht fehlerhaft sein. Solche Fehler können ebenfalls nicht zeitnah behoben werden. Nicht selten kommen bestimmte Anforderungen erst während der Implementation zutage (z. B. Anforderungen an die Benutzeroberfläche). Diese können jedoch mit dem Wasserfall-Modell nicht angemessen berücksichtigt werden. Darüber hinaus dauert ein sequentielles Vorgehen per Definition länger als ein nicht-sequentielles Vorgehen, da eine Disziplin erst beginnen kann, wenn die vorherige abgeschlossen wurde.

Wasserfall-Modell ist selten erfolgreich

Wie sich in der Praxis gezeigt hat, ist dieser Ansatz in den seltensten Fällen erfolgreich [Parnas et al. 1986, Larman 2002, Cockburn 2002]. Trotzdem trifft man leider noch immer viele Projekte an, die das Wasserfall-Modell anwenden und damit scheitern. An dieser Stelle sollte noch festgehalten werden, dass auch Royce sich der Problematik eines sequentiellen Vorgehens bewusst war und er dies auch diskutiert hat [Royce 1970]. Das Wasserfall-Modell kann als historischer Irrtum bezeichnet werden [Oestereich und Weiss 2008].

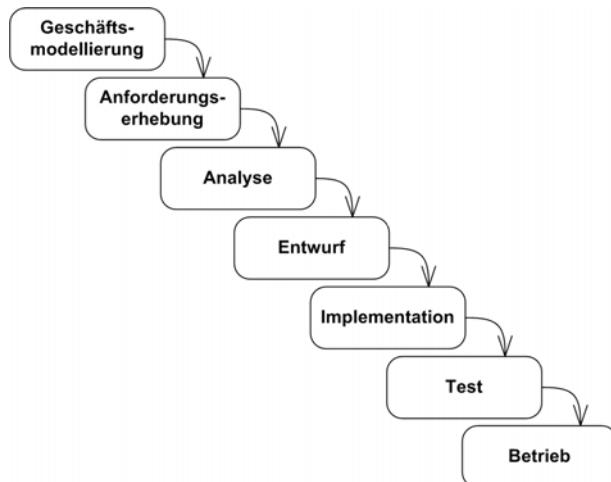


Abb. 8.1-2: Beispielhafte Darstellung des Wasserfall-Modells.

Wie an den Ausführungen deutlich geworden ist, kann man im Rahmen eines architektonischen Vorgehens nicht davon ausgehen, zuerst eine komplette Architektur bis ins Detail zu entwerfen, und dann die Architektur umzusetzen. Veränderungen an den Anforderungen können so nicht ausreichend schnell berücksichtigt werden. Außerdem erfolgt dadurch eine vollumfängliche Validierung der Architektur erst ganz am Ende eines Entwicklungsprozesses. Fehler in der Architektur lassen sich dann nur schwer und mit großem Aufwand beseitigen.

Eine Architektur entsteht nicht streng sequentiell

Das Spiral-Modell ist eine Verfeinerung des Wasserfall-Modells, welches die Schwächen des letzteren zu verbessern sucht. [Boehm 1998]. Statt die genannten Disziplinen nur ein einziges Mal sequentiell zu durchlaufen, wird ein Software-Entwicklungsprojekt in mehrere Zyklen unterteilt. In jedem Zyklus sind alle Disziplinen involviert. Ein Zyklus endet jeweils mit einem Review der Arbeitsergebnisse durch die betroffenen Interessenvertreter. So werden beispielsweise Benutzer gegen Ende eines Zyklus das System testen. Dadurch erhält man wichtige Informationen über den Zielerreichungsgrad und kann diese Informationen im nächsten Zyklus berücksichtigen. Die Arbeitsergebnisse, wie z. B. Anwendungsfallbeschreibungen, UML-Komponenten und -Sequenzdiagramme oder Quelltext, entstehen hierdurch schrittweise. Dies senkt das Risiko, am Ende eines Entwicklungsvorhabens den Zielbereich nicht zu erreichen. Ein wichtiger Bestandteil des Spiral-Modells ist die Entwicklung von Prototypen oder Simulationen zur Bewertung von Alternativen und zur Abschätzung von Risiken. Gerade Prototypen sind ein wertvolles Hilfsmittel, um Aspekte einer Architektur zu prüfen oder zu veranschaulichen (siehe Abschnitte 8.4 und 8.5).

Das Spiral-Modell

Neuere Vorgehensmodelle, wie der Unified Software Development Process (USDP), V-Modell XT, eXtreme Programming (XP), Feature Driven Development (FDD) oder Scrum, setzen ähnlich wie beim Spiral-Modell auf die Idee einer schrittweisen, durch mehrere Zyklen verlaufenden Entwicklung von Software-Systemen. In diesem Zusammenhang hat sich der Begriff des iterativ-inkrementellen Vorgehens etabliert. Bei solch einem Vorgehen wird der gesamte Entwicklungsprozess in einzelne, aufeinander folgende und aufbauende Entwicklungsschritte unterteilt, den sogenannten Iterationen. Das Besondere dabei ist, dass innerhalb jeder Iteration alle typischen Disziplinen und Tätigkeiten einer Software-Entwicklung stattfinden. Das heißt, in jeder Iteration wird die Aufgabenstellung ein Stück weit analysiert, für das Analysierte wird ein Entwurf erstellt und anschließend umgesetzt. Am Ende einer Iteration hat man sich so ein Stück weiter in die Aufgabenstellung eingearbeitet und die Lösung um ein weiteres Stück vervollständigt. Jedes dieser

Iterativ-inkrementelles Vorgehen

Stücke, welches die Lösung komplettiert, wird als Inkrement bezeichnet und stellt ein in Teilen bereits funktionierendes System dar. So kommt man schrittweise, Stück für Stück, sprich iterativ-inkrementell, zum endgültigen System.

Iterationsplanung und Anforderungen

Ein iterativ-inkrementelles Vorgehen benötigt natürlich Planung. Insbesondere sind hierbei die Iterationen zu planen. Dabei wird die Anzahl der benötigten Iterationen bestimmt und festgelegt, welche Anforderungen in den jeweiligen Iterationen realisiert werden. Ein Architekt muss den Projektleiter bei der Planung unterstützen, um die sinnvolle Priorisierung architektureller Anforderungen (siehe Abschnitt 8.4) zu gewährleisten. Die Analyse der Anforderungen liefert unter anderem die Gewichtung der einzelnen Anforderungen als Ergebnis. Beispielsweise könnten Anforderungen mit einem großen Nutzen für den Anwender und/oder mit einem hohen Risiko (z. B. Zeitbudget wird deutlich überschritten oder technische Umsetzung ist nicht ohne Weiteres möglich) eine hohe Gewichtung erhalten. Anforderungen mit einem geringen Nutzen und/oder einem geringen Risiko würden dagegen entsprechend eine geringere Gewichtung erhalten. Im Rahmen der Iterationsplanung erfolgt eine Verteilung der einzelnen Anforderungen anhand ihrer Gewichtung auf die einzelnen Iterationen. Anforderungen mit einer hohen Gewichtung werden möglichst früh in einer der ersten Iterationen realisiert. Anforderungen mit einer geringen Gewichtung werden für spätere Iterationen eingeplant. Mehr zum Gewichten von Anforderungen findet sich in Abschnitt 8.4. Diese Verteilung der Anforderungen auf einzelne Iterationen ist jedoch kein Fixum. Vielmehr kommt es während der Bearbeitung der einzelnen Iterationen auch zu Veränderungen in der Gewichtung der einzelnen Anforderungen, neue Anforderungen ergeben sich und bestehende Anforderungen verändern sich. Diese Veränderungen in den Anforderungen bewirken auch eine Veränderung und Anpassung der anfänglichen Iterationsplanung. Ein iterativ-inkrementelles Vorgehen bietet hier den entscheidenden Vorteil, dass sich der Entwicklungsprozess und seine Ergebnisse kontinuierlich an ihr sich veränderndes Umfeld anpassen können. Diese kontinuierliche Anpassung an das sich verändernde Umfeld erfordert aber auch ein besonderes Vorgehen eines Architekten bei seiner Arbeit.

Ressourcenplanung und Anforderungen

Neben der Iterationsplanung ist die Aufwandschätzung für die Ressourcenplanung (Budget, Zeit, Personen) ein weiterer Punkt, bei dem architekturelle Anforderungen zum Tragen kommen. Es ist wichtig, dass ein Architekt bei der Priorisierung und der Aufwandschätzung von Anforderungen ausreichend involviert ist, um Projektleitung und Domänenexperten zu unterstützen. Andernfalls besteht das Risiko, dass

technische Aspekte ungenügend Berücksichtigung finden bzw. falsch eingeschätzt werden (z. B. eingeschränkte Möglichkeiten eines Frameworks in Bezug auf Umsetzung bestimmter Anforderungen) mit negativen Folgen im weiteren Verlauf auf die Projektplanung (z. B. Umsetzung erfordert mehr Personen als geplant).

Mit XP, FDD und Scrum wurden bereits einige Vertreter agiler Vorgehensmodelle genannt. Aus architektonischer Sicht relevant ist bei agilen Vorgehensmodellen, dass diese ausschließlich ein iterativ-inkrementelles Vorgehen vorsehen und den Umfang von Dokumentation einschränken bzw. „überflüssige“ Dokumentation in Frage stellen [Cunningham et al. 2001]. Beispielsweise sollte Dokumentation nicht mögliche zukünftige, sondern nur aktuell vorliegende konkrete Anforderungen berücksichtigen. Für einen Architekten stellt sich die Frage, ob dies auch für die Architektur-Dokumentation gilt respektive inwieweit eine Architektur apriori festlegt werden muss und wie dies dann in einem agilen Kontext explizit gemacht werden kann. Ferner finden sich gerade im Bereich XP Aussagen, die einen Architektur-Entwurf als unnötig erachten. Diese Aussage wird jedoch in der XP-Gemeinschaft kontrovers diskutiert [Fowler 2004]. Generell kann gesagt werden, dass der Grad des Architektur-Entwurfs als auch der Architektur-Dokumentation getreu dem Sufficient-to-purpose-Prinzip zweckmäßig sein sollte [Cockburn 2002 und Fowler 2004]. Es kommt also sehr auf die aktuelle Situation an. Ambler argumentiert, dass auch bei agilen Vorgehensmodellen die Architektur nicht zu kurz kommen darf. In der ersten Iteration sollte sie bereits soweit gedeihen, dass sie als Vision und Orientierung für Teammitglieder dienen kann. Die Architektur selbst als auch die Dokumentation wird man auch in agilen Ansätzen über die Zeit, je nach Bedarf, anpassen [Ambler 2007].

Vorgehensmodelle sind Vorlagen, die an konkrete Situationen angepasst werden müssen respektive sollten. Die Projektgröße ist beispielsweise ein Indikator dafür, wie ein Vorgehensmodell in der Praxis adaptiert werden sollte. So existieren z. B. bereits Varianten von USDp für große und kleine Projekte. Neben der Adaption von Vorgehensmodellen können speziell in großen Projekten Teilprojekte unterschiedliche, für deren jeweiligen Kontext (z. B. Teammitglieder sind sehr erfahren so dass sich XP anbietet) geeignete Vorgehensmodelle einsetzen. Dies kann jedoch auch problematisch sein, weil sich die Abstimmung der Teilprojekte untereinander durch inhomogene Vorgehensmodelle erschwert. Des Weiteren können Vorgehensmodelle auch kombiniert werden. Projekte können ihren konkreten Entwicklungsprozess z. B. aus USDp und XP aufbauen. Zum einen kann USDp das generelle Vorgehen

Agile Vorgehensmodelle

Vorgehensmodelle können adaptiert und kombiniert werden

definieren, zum anderen können jedoch auch Best Practices aus XP wie „Fortlaufende Integration“ (englisch: *continuous integration*) und „Paarweises Programmieren“ (englisch: *pair programming*) aus dem Bereich Extreme Programming (XP) [Beck 2000] in den Prozess integriert werden. An dieser Stelle sei kurz das Software Engineering Meta Model (SPEM) der OMG sowie das Eclipse Process Framework (EPF) erwähnt. SPEM ist ein Metamodell zur Definition von Prozessen (siehe Abschnitt 6.6.5). Mit dem EPF können Prozesse auf Basis dieses Metamodells erstellt, adaptiert und kombiniert werden.

USDP als Beispiel eines iterativ-inkrementellen Entwicklungsprozesses

Im Folgenden wird USDP, häufig auch als UP (Unified Process) bezeichnet, als Beispiel eines iterativ-inkrementellen Vorgehensmodells näher vorgestellt [Jacobson et al. 1999]. Eine Variante dieses Vorgehensmodells ist der Rational Unified Process (RUP). Abbildung 8.1-3 zeigt die wesentlichen Elemente des USDP.

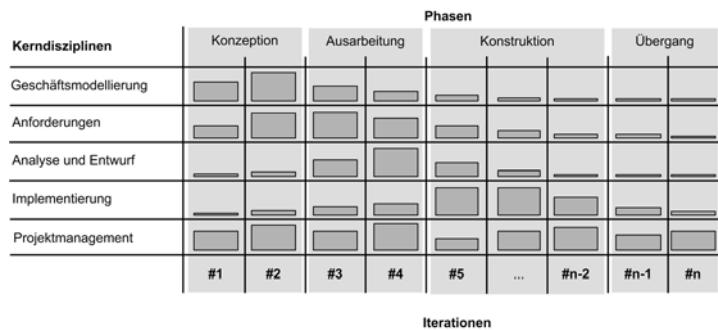


Abb. 8.1-3: Elemente des USDP.

In Abbildung 8.1-3 wird gezeigt, wie ein iterativ-inkrementelles Vorgehen auf der vertikalen Achse über die Zeit (Phasen und Iterationen) und auf der horizontalen Achse über Kerndisziplinen gegliedert ist. Hierbei werden nur die für einen Architekten relevantesten Kerndisziplinen (englisch: *core process disciplines*) dargestellt. Im USDP sind noch weitere Kerndisziplinen vorgesehen. In einem iterativ-inkrementellen Vorgehen werden zur besseren Übersicht bestimmte Phasen (Zeitabschnitte) unterschieden, die bei der Entwicklung eines Systems durchlaufen werden. Diese Phasen sind nicht zu verwechseln mit den Phasen des Wasserfallmodells. Jede dieser Phasen kann mehr oder weniger Iterationen umfassen. Die Kerndisziplinen sind dabei unterschiedlich stark gewichtet, abhängig von der Phase, zu der eine Iteration gehört. In Abbildung 8.1-3 wird die Gewichtung der Kerndisziplinen zu unterschiedlichen Zeitpunkten mittels verschiedener großer Blöcke in den Kreuzungspunkten der Zeit- und Aktivitätsachsen angedeutet. Es wird jedoch nur die

Tendenz in der Gewichtung der Kerndisziplinen gezeigt. In einem konkreten Projekt kann dies in absoluten Zahlen im Detail anders aussehen. Im weiteren Verlauf werden die verschiedenen Phasen kurz besprochen.

In der Konzeptionsphase (englisch: *inception*) liegt das Hauptaugenmerk auf der Übereinstimmung der Interessenvertreter in den Zielen des Vorhabens. Des Weiteren ist es zu diesem frühen Zeitpunkt ebenfalls wichtig, den Umfang des zu realisierenden Systems zu definieren. Hierzu gehört die Definition der wichtigsten Anforderungen und der Abnahmekriterien. Darüber hinaus sind Risiken zu identifizieren und abzuschätzen. In dieser Phase werden bereits erste Entscheidungen hinsichtlich der Architektur getroffen und ggf. eine erste Architektur entworfen, die die Machbarkeit demonstriert und somit Vertrauen in das geplante Vorhaben schafft.

Konzeption

Während der Ausarbeitungsphase (englisch: *elaboration*) ist man besonders gefordert, da in dieser Phase die Architektur soweit ausgearbeitet werden muss, dass eine stabile, architektonische Basis für die Tätigkeiten in den darauf folgenden Phasen vorhanden ist. Die Berücksichtigung aller architektonischen Anforderungen, das Entwerfen der Architektur, die Verifikation von Alternativen mittels Architektur-Prototypen sowie die Realisierung von evolutionären Architektur-Prototypen werden in dieser Phase behandelt. Am Ende der Ausarbeitungsphase muss sicher gestellt sein, dass die gewählte Architektur die identifizierten Risiken ausreichend minimiert.

Ausarbeitung

Bislang nicht im Detail betrachtete Anforderungen werden in der Konstruktionsphase (englisch: *construction*) genauer analysiert und dokumentiert. Ferner erfolgt die inkrementelle bzw. schrittweise Umsetzung der einzelnen Anforderungen. Mit anderen Worten wird in dieser Phase das System auf Basis der entworfenen Architektur entwickelt. In dieser Phase steht die Sicherstellung der Architektur-Konformität im Vordergrund. Dies kann man beispielsweise durch Reviews und Schulungen erreichen. Ferner hat man als Architekt die Aufgabe, den Umgang mit der Architektur so effizient wie möglich zu gestalten. Zu diesem Zweck könnte beispielsweise ein Skelettsystem erzeugt werden (siehe Abschnitt 8.6).

Konstruktion

In der abschließenden Übergangssphase (englisch: *transition*) liegen die Bestrebungen darauf sicherzustellen, dass das System auch an die Endbenutzer übergeben werden kann. Beispielsweise werden Fehler behoben, Benutzer geschult, das Wartungspersonal eingewiesen und die Betriebs- und Installationsdokumentation erstellt. Ein Architekt wird

Übergang

häufig in die Fehleranalyse und -behebung involviert werden. Ferner wird er auch für die Schulung der architektonischen Aspekte gegenüber dem Wartungs- respektive Betriebspersonal verantwortlich sein.

Zusammenfassung

- > Der geplante und der tatsächliche Entwicklungspfad weichen aufgrund verschiedener Einflussfaktoren voneinander ab. Dieses Sachverhalts muss man sich bewusst sein und sein Vorgehen darauf abstimmen.
- > Vorgehensmodelle sind Leitfäden zur Strukturierung einzelner Kerndisziplinen, (Architektur-)Tätigkeiten, (Architektur-)Aktionen sowie querschnittlichen Aufgaben und legen deren zeitliche Abfolge im Rahmen eines Entwicklungsprozesses fest.
- > Man unterscheidet wasserfallartige und iterativ-inkrementelle Vorgehensmodelle.
- > Iterativ-inkrementelle Vorgehensmodelle bieten die beste Möglichkeit, um auf Änderungen während der Entwicklung reagieren zu können.
- > Eine Architektur sollte nicht streng sequentiell, sondern vielmehr iterativ-inkrementell entstehen.
- > Ein Architekt muss den Projektleiter bei seiner Planung der Iterationen unterstützen, um sicherzustellen, dass architekturelle Anforderungen möglichst früh adressiert werden und so eine Risikominimierung stattfinden kann.
- > Vorgehensmodelle können kombiniert werden. Hierfür existieren im Markt Vorgehensmetamodelle (beispielsweise das SPEM oder die UMA) sowie Software-Werkzeuge (beispielsweise der RMC oder das EPF).
- > Im USDP werden die Phasen Konzeption (englisch: *inception*), Ausarbeitung (englisch: *elaboration*), Konstruktion (englisch: *construction*) und Übergang (englisch: *transition*) unterschieden.

8.2 Architektonisches Vorgehen im Überblick

Architektonisches Vorgehen als Voraussetzung für eine erfolgreiche Architektur

In den vorangegangen Kapiteln wurde essenzielles architektonisches Wissen vermittelt, welches für einen Architekten eine wichtige Grundlage bildet, um im Alltag erfolgreich zu handeln. Dieses Wissen ist allerdings wertlos, wenn man nicht in der Lage ist, es zielgerichtet und bewusst einzusetzen. Hierzu muss man ein architektonisches Bewusstsein entwickeln, architektonisch denken und vorgehen. Erst dann kommt das bislang behandelte architektonische Wissen sinnvoll zum

Tragen. Doch wie ist nun am Besten vorzugehen? Welches sind überhaupt die wesentlichen Tätigkeiten eines Architekten? Wie lässt sich das Vorgehen in einen bestehenden Software-Entwicklungsprozess integrieren? Und schließlich, wie wirken die anderen Architektur-Dimensionen (WAS, WO, WARUM, WOMIT und WER) auf die architektonischen Tätigkeiten ein? Dieser Abschnitt gibt einen kurzen Überblick über die Tätigkeiten eines Architekten. Die Abschnitte 8.3 bis 8.7 behandeln die einzelnen Tätigkeiten im Detail.

Wie bereits in Kapitel 3 verdeutlicht wurde, umfasst Software-Architektur nicht nur die Software-Strukturen eines Systems, sondern ebenso die Tätigkeiten, die zu diesen Strukturen führen. Für die Strukturierung existieren vielfältige Architektur-Mittel (siehe Kapitel 6). Deren alleinige Existenz führt jedoch noch nicht automatisch zu einer guten Architektur. Daher ist es wichtig, dass man sich bei der Auswahl der Mittel an einem systematischen Vorgehensmodell orientiert. Dabei ist es zunächst einmal unerheblich, nach welchem Software-Entwicklungsprozess vorgegangen wird. Ob das System nun wasserfallartig oder iterativ-inkrementell entwickelt wird, man wird immer vor der Herausforderung stehen, sein Vorgehen auf die Definition einer Architektur, die sowohl die funktionalen als auch die nicht-funktionalen Anforderungen erfüllt, auszurichten. Wie in Abschnitt 8.1 bereits deutlich wurde, ist ein iterativ-inkrementelles Vorgehen zu bevorzugen.

Ein gutes Vorgehensmodell allein führt jedoch noch nicht zu einer guten Architektur. Ein Architekt kann sich von einem Vorgehensmodell leiten lassen. Er wird jedoch über die generischen Hinweise eines Vorgehensmodells hinaus, immer auch aus seinem eigenen Erfahrungsschatz schöpfen müssen. Beispielsweise sind die Empfehlungen eines Vorgehensmodells aufgrund eigener Erfahrungen auf die konkrete Projektsituation zu adaptieren.

Aus welchen Tätigkeiten setzt sich nun ein architektonisches Vorgehen zusammen? Abbildung 8.2-1 visualisiert das allgemeine architektonische Vorgehensmodell mithilfe eines UML-Aktivitätendiagramms. Im weiteren Verlauf dieses Abschnitts wird das Modell kurz erläutert, bevor dessen Einbettung in Software-Entwicklungsprozesse analysiert wird.

Die in Abbildung 8.2-1 präsentierten Tätigkeiten orientieren sich an der Arbeit von Bass et al. [Bass et al. 2003] und unseren eigenen Erfahrungen. Bass et al. führen zusätzlich die Tätigkeiten „Analysieren und Beurteilen der Architektur“ und „Sicherstellen der Architektur-Konformität“ ein. Die Aufgabe der Architektur-Beurteilung wird in dem hier

Ein Architekt benötigt ein Vorgehensmodell ...

... und Erfahrung

Die typischen Tätigkeiten eines Architekten

vorgestellten Vorgehensmodell im Rahmen der Tätigkeit „Entwerfen der Architektur“ durchgeführt. Des Weiteren widmet sich die Tätigkeit „Umsetzen der Architektur“ auch dem Aspekt der Architektur-Konformität. Die Tätigkeiten sind unabhängig vom konkreten Entwicklungsprozess. Abgesehen von der Tätigkeit „Erstellen der Systemvision“ sollten die Tätigkeiten jedoch immer iterativ und inkrementell ausgeführt werden. Im weiteren Verlauf dieses Kapitels werden die Tätigkeiten deshalb im Kontext eines iterativ-inkrementellen Vorgehens betrachtet.

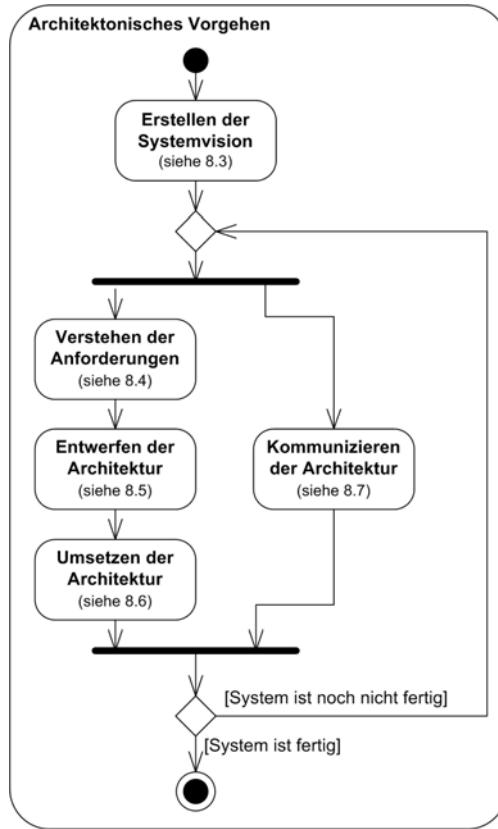


Abb. 8.2-1: Architektonisches Vorgehen im Überblick.

Erstellen der Systemvision

Eine Systemvision lotet zum einen die betriebswirtschaftliche Zweckmäßigkeit einer Initiative aus. Beispielsweise wird eine Systemvision zur Ablösung eines über Jahrzehnte gewachsenen Host-Systems durch ein JEE-basiertes System klare Aussagen über die angestrebte und erwartete Rentabilität beinhalten. Zum anderen definiert eine Systemvision jedoch auch die essenziellen Anforderungen an das zukünftige

System. In der Verantwortung eines Architekten liegt es, diese Anforderungen hinsichtlich ihrer architektonischen Machbarkeit kritisch zu hinterfragen, im gesamten bzw. globalen IT-Kontext einer Organisation zu sehen, auf sich widersprechende Anforderungen hinzuweisen und Alternativen aufzuzeigen. Daher ist es von erheblicher Bedeutung, dass bereits zu diesem frühen Zeitpunkt ein Architekt an der Erstellung der Systemvision mitwirkt. Sonst besteht die Gefahr, dass eine Systemvision aus architektonischer Sicht unrealistisch ist und die Realisierung eines Systems bereits unter schlechten Vorzeichen beginnt. Bezogen auf das architektonische Vorgehen muss man sich bei der Ausübung der Tätigkeit „Erstellen der Systemvision“ als architektonischer Berater sehen, der die Aufgabe hat, die Systemvision auf ein tragfähiges architektonisches Fundament zu stellen. Erst wenn die Systemvision verabschiedet wurde, kann man sich anderen architektonischen Tätigkeiten widmen.

Wie bereits in Kapitel 5 verdeutlicht, beschränken Anforderungen den gestalterischen Spielraum eines Architekten. Damit man sich diesen Spielraum vergegenwärtigen und auch nutzen kann, müssen die Interessenvertreter und die gestellten Anforderungen verstanden werden. Aus diesem Grund widmet sich die Tätigkeit „Verstehen der Anforderungen“ der Identifikation, der Priorisierung und der Detaillierung von architekturelevanten Anforderungen. Insbesondere die bewusste Auseinandersetzung mit nicht-funktionalen Anforderungen ist von großer Bedeutung, da diese oftmals gar nicht oder unscharf formuliert sind. Je deutlicher die architekturelevanten Anforderungen herausarbeitet werden, umso klarer wird der architektonische Gestaltungsspielraum und umso eher wird die im nächsten Schritt zu entwerfende Architektur, die an sie gestellten Anforderungen erfüllen.

Verstehen der Anforderungen

Die eigentliche Architektur entsteht während der Ausübung der Tätigkeit „Entwerfen der Architektur“. Hierzu bedient man sich eines breiten Fundus an Architekturmitteln (siehe Kapitel 6). So kommt beispielsweise die Verwendung einer bewährten Referenzarchitektur, eines erprobten Architekturmusters oder schlicht zeitloser Architekturprinzipien in Betracht. Darüber hinaus werden zu einem späteren Zeitpunkt auch konkrete Plattform- respektive Technologie-Entscheidungen getroffen. Aus der Summe aller Entscheidungen resultiert schließlich die Architektur des Systems. In der Regel gibt es mehr als eine Architektur-Alternative, um die gegebenen Anforderungen umzusetzen. Man wird sich also zwischen Alternativen entscheiden müssen. Aufgrund dessen kommt der Beurteilung von Architektur-Alternativen im Rahmen dieser Tätigkeit eine wichtige Bedeutung zu.

Entwerfen der Architektur

Umsetzen der Architektur

Eine Architektur darf nicht nur Entwurf bleiben, sondern muss sich auch technologisch manifestieren. Hierzu ist es notwendig, dass die Architektur erfolgreich umgesetzt wird. Das Umsetzen kann dabei auf unterschiedliche Arten erfolgen. Das Spektrum reicht dabei von der Definition reiner Entwicklungsrichtlinien und manueller Reviews bis hin zu der Etablierung einer Infrastruktur, welche die Entwickler bei der architekturkonformen Implementation des Systems führt und unterstützt. Oftmals wird die Umsetzung der Architektur vernachlässigt (siehe Kapitel 7). Das Ergebnis sind Systeme, die der Architektur nur teilweise oder gar nicht folgen und die architekturrelevanten Anforderungen nicht befriedigen.

Kommunizieren der Architektur

Ein Architekt muss nicht nur eine Architektur entwerfen, sondern er muss sie auch den unterschiedlichen Interessenvertretern vermitteln. „Kommunizieren der Architektur“ hat somit das Ziel, den einzelnen Interessenvertretern (z. B. Projektleitern, Entwicklern, Benutzern, Kunden) ein möglichst gutes Verständnis der Architektur sowie der Architektur-Entscheidungen zu vermitteln. Dieses Verständnis ist dann wiederum Grundlage für die Tätigkeiten der einzelnen Interessenvertreter. Kommunizieren umfasst hier die Dokumentation einer Architektur und deren mündliche Vermittlung auf Basis bzw. mithilfe der Architekturdokumentation. Die Kommunikation der Architektur läuft stets parallel zu den anderen Tätigkeiten ab. Als Architekt wird man bereits während des Verstehens der Anforderungen mit Interessenvertretern kommunizieren, um z. B. auf sich widersprechende Anforderungen hinzuweisen. Beim „Entwerfen der Architektur“ wird man seine Entwürfe Interessenvertretern präsentieren. Beim Umsetzen der Architektur wird man wiederum Quelltext-Reviews (englisch: *source code reviews*) durchführen, um die Architektur-Konformität zu gewährleisten. Hierbei wird man mit seinen Entwicklern kommunizieren und Review-Ergebnisse besprechen. Selbstverständlich findet eine Kommunikation auch während der Erstellung der Systemvision statt. Schwerpunktmäßig wird ein Architekt jedoch während der Entwicklung eines Systems kommunizieren.

Ein Architekt ist über den ganzen Software-Entwicklungszyklus hinweg involviert

Nachdem erfolgreich eine Vision für ein System erstellt und das eigentliche Software-Entwicklungsprojekt begonnen wurde, wird ein Architekt die anderen architektonischen Tätigkeiten über den gesamten Software-Entwicklungszyklus hinweg ausüben (siehe Abbildung 8.2-1). Die Tätigkeitsschwerpunkte werden sich dabei vom „Verstehen der Anforderungen“ über das „Entwerfen der Architektur“ hin zum „Umsetzen der Architektur“ verschieben. Das „Kommunizieren der Architektur“ wird jedoch kaum an Bedeutung verlieren, da eine Architektur in den Köpfen der Interessenvertreter lebendig gehalten werden muss.

Wichtiges Ergebnis der architektonischen Tätigkeiten ist die Erstellung der verschiedenen, in Kapitel 4 beschriebenen Architektur-Sichten [Rozanski und Woods 2005]. In Tabelle 8.2-1 wird aufgezeigt, welche Architektur-Sichten in den einzelnen architektonischen Tätigkeiten bearbeitet werden.

Architektonische Sichten werden erarbeitet

Tab. 8.2-1: Architektur-Sichten und architektonische Tätigkeiten.

Architektur-Sichten	Architektonische Tätigkeiten
Anforderungssicht	<ul style="list-style-type: none"> > Erstellen der Systemvision > Verstehen der Anforderungen > Kommunizieren der Architektur
Logische Sicht	<ul style="list-style-type: none"> > Erstellen der Systemvision > Entwerfen der Architektur > Kommunizieren der Architektur
Datensicht	<ul style="list-style-type: none"> > Entwerfen der Architektur > Kommunizieren der Architektur
Umsetzungssicht	<ul style="list-style-type: none"> > Umsetzen der Architektur > Kommunizieren der Architektur
Verteilungssicht	<ul style="list-style-type: none"> > Entwerfen der Architektur > Kommunizieren der Architektur
Prozesssicht	<ul style="list-style-type: none"> > Entwerfen der Architektur > Kommunizieren der Architektur

Die architektonischen Tätigkeiten werden von einem Architekten im Rahmen eines iterativ-inkrementellen Entwicklungsprozesses kontinuierlich ausgeführt. In jeder Iteration setzt sich die architektonische Arbeit aus einer Kombination dieser Tätigkeiten zusammen. Allerdings verändert sich der Anteil der einzelnen Tätigkeiten von Iteration zu Iteration. Während in der Startphase das Hauptgewicht auf den Tätigkeiten „Erstellen der Systemvision“ und „Verstehen der Anforderungen“ liegt, verschiebt sich der Schwerpunkt der Arbeit während des Entwicklungsprozesses hin zu den Tätigkeiten „Entwerfen der Architektur“, „Umsetzen der Architektur“ und „Kommunizieren der Architektur“. Damit passen sich die architektonischen Tätigkeiten in die Struktur des umgebenden iterativ-inkrementellen Entwicklungsprozesses ein. Das Handeln im Rahmen der einzelnen architektonischen Tätigkeiten kann sich so, im gleichen Maße wie der Entwicklungsprozess, an die sich verändernden Anforderungen anpassen.

Architektonisches Vorgehen im Rahmen eines iterativ-inkrementellen Vorgehens

Einbettung der Architektur-Tätigkeiten in Entwicklungsprozesse am Beispiel USDP

Die architektonischen Tätigkeiten können in die Disziplinen des USDP eingebettet werden. Abbildung 8.2-2 verdeutlicht, welche architektonischen Tätigkeiten innerhalb welcher Kerndisziplinen eines iterativ-inkrementellen Entwicklungsprozesses am Beispiel des USDP zum Tragen kommen. Manchmal wird sich eine Zuordnung nicht eindeutig treffen lassen. Dies ist jedoch nicht von so großer Bedeutung, da es wichtiger ist, die Tätigkeiten auszuüben, egal in welche Kerndisziplin diese nun genau einzubetten sind.

Kerndisziplinen und architektonische Tätigkeiten	Phasen								
	Konzeption	Ausarbeitung	Konstruktion			Übergang			
Geschäftsmodellierung Kein direkter Bezug	■	■	■	■	■	■	■		
Anforderungen Erstellen der Systemvision Verstehen der Anforderungen	■	■	■	■	■	■	■		
Analyse und Entwurf Entwerfen der Architektur	■	■	■	■	■	■	■		
Implementierung Umsetzen der Architektur	■	■	■	■	■	■	■		
Projektmanagement Kommunizieren der Architektur	■	■	■	■	■	■	■		
	#1	#2	#3	#4	#5	...	#n-2	#n-1	#n
	Iterationen								

Abb. 8.2-2: USDP-Kerndisziplinen und architektonische Tätigkeiten.

Praxisbeispiel: MIS

Im weiteren Verlauf wird das architektonische Vorgehen anhand eines konkreten Praxisbeispiels veranschaulicht. Als Beispiel dient hierzu das bereits in Abschnitt 3.4 vorgestellte Management-Informationssystem zur Erhebung und Auswertung von Geschäftskennzahlen.

Zusammenfassung

- > Ein Architekt benötigt ein Vorgehensmodell, welches ihn bei seinem Vorhaben leitet.
- > Neben einem Vorgehensmodell ist die eigene Erfahrung essenziell, um die Empfehlungen des Vorgehensmodells an die konkrete Projekt-situation zu adaptieren.
- > Das architektonische Vorgehen besteht aus den Tätigkeiten „Erstellen der Systemvision“, „Verstehen der Anforderungen“, „Entwerfen der Architektur“, „Umsetzen der Architektur“ und „Kommunizieren der Architektur“.
- > Die Tätigkeit „Kommunizieren der Architektur“ wird parallel zu den anderen Tätigkeiten ausgeführt.
- > Beim „Erstellen der Systemvision“ liegt es in der Verantwortung des Architekten, Anforderungen hinsichtlich ihrer architektonischen Machbarkeit kritisch zu hinterfragen, im globalen IT-Kontext einer Organisation zu sehen, auf sich widersprechende Anforderungen hinzuweisen und Alternativen aufzuzeigen.

- > Bezogen auf das architektonische Vorgehen muss sich der Architekt bei der Ausübung seiner Tätigkeit „Erstellen der Systemvision“ als architektonischer Berater sehen.
- > Die Tätigkeit „Verstehen der Anforderungen“ widmet sich der Identifikation, der Priorisierung und der Detailierung von architekturellen Anforderungen. Insbesondere die bewusste Auseinandersetzung mit nicht-funktionalen Anforderungen ist von großer Bedeutung.
- > Das „Umsetzen der Architektur“ befasst u. a. sich mit der Definition von Entwicklungsrichtlinien, manuellen Reviews und der Etablierung einer Infrastruktur.
- > Die Tätigkeit „Kommunizieren der Architektur“ hat das Ziel, den einzelnen Interessenvertretern ein möglichst gutes Verständnis der Architektur sowie der Architektur-Entscheidungen zu vermitteln.
- > Wichtiges Ergebnis der architektonischen Tätigkeiten ist die Erstellung der verschiedenen in Kapitel 4 beschriebenen Architektur-Sichten.
- > Der Anteil der einzelnen architektonischen Tätigkeiten verlagert sich von Iteration zu Iteration.

8.3 Erstellen der Systemvision

Abbildung 8.3-1 stellt die grundlegenden Konzepte der Tätigkeit „Erstellen der Systemvision“, welche in diesem Abschnitt behandelt und detailliert werden, vor und visualisiert ihren Zusammenhang.

Grundlegende Konzepte der Tätigkeit „Erstellen der Systemvision“

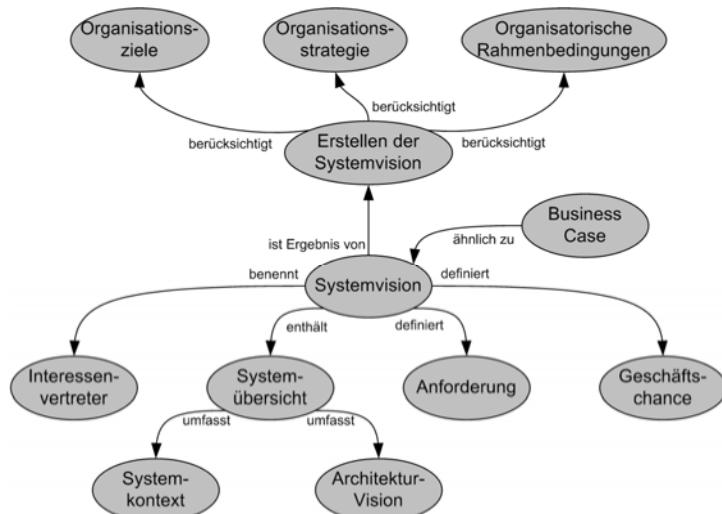


Abb. 8.3-1: Grundlegende Konzepte der Tätigkeit „Erstellen der Systemvision“.

Der Weg zu einer Systemvision

Abbildung 8.3-2 illustriert die einzelnen Aktionen der Tätigkeit „Erstellen der Systemvision“. Im Rahmen dieser Tätigkeit wird primär die Anforderungssicht eines Systems (siehe Abschnitt 4.2) erarbeitet. Nicht alle diese Aktionen sind primär architektonischer Natur. Eine Systemvision muss jedoch architektonisch machbar sein. Daher ist es wichtig, dass ein Architekt in allen Aktionen mitwirkt und die Systemvision durch die Architektur-Brille betrachtet. Ein Beispiel soll dies verdeutlichen. Es besteht die Anforderung, dass ein System bis zu einem bestimmten Zeitpunkt produktiv zur Verfügung stehen soll. Um eine möglichst große Freiheit bei der Weiterentwicklung zu haben, besteht außerdem die Forderung, das System selbst zu entwickeln. Allerdings ist der Funktionsumfang des Systems so groß, dass es nicht in dem gegebenen Zeitraum entwickelt werden kann. Es ist nun Aufgabe eines Architekten, auf diesen Konflikt zwischen den einzelnen Anforderungen hinzuweisen und Lösungsalternativen aufzuzeigen. Mögliche Lösungsalternativen können in diesem Fall sein, den gewünschten Produktivstart des Systems zu verschieben oder aber das System durch eine Integration von zugekauften Produkten und Eigenentwicklungen zu realisieren. Hierzu werden Eigenentwicklungen in den Bereichen eingesetzt, in denen es wichtig ist, nicht von Produktlebenszyklen eines Produktanbieters abhängig zu sein.

Architekt als architektonischer Berater

Ein Architekt übernimmt in dieser Tätigkeit die Rolle eines architektonischen Beraters in einem Team von Fachleuten aus unterschiedlichen Bereichen. Andere Teammitglieder sind zum Beispiel Domänenexperten. Erst durch eine Teamzusammensetzung mit Mitgliedern aus den unterschiedlichen Bereichen ist es möglich, eine ganzheitliche Systemvision zu erarbeiten.

Systemvision und Business Case

Manchmal spricht man statt von einer Systemvision auch von einem Business Case. Ein Business Case beleuchtet in der Regel den wirtschaftlichen Nutzen stärker als eine Systemvision. Der Übergang ist allerdings fließend. USDP unterscheidet Business Case und Systemvision. Allerdings wird auch der enge Zusammenhang betont.

Beschreibe Geschäftschancen

Die Geschäftschancen, die durch die Realisierung des Systems verwirklicht werden sollen, sind ein wesentlicher Bestandteil einer Systemvision. Der Nutzen des Systems und die Probleme, die das System löst, sollten klar wiedergegeben werden. Der Nutzen wird nicht nur qualitativ, sondern auch quantitativ in Form von Geschäftskennzahlen, wie dem Return on Investment (ROI), beziffert.

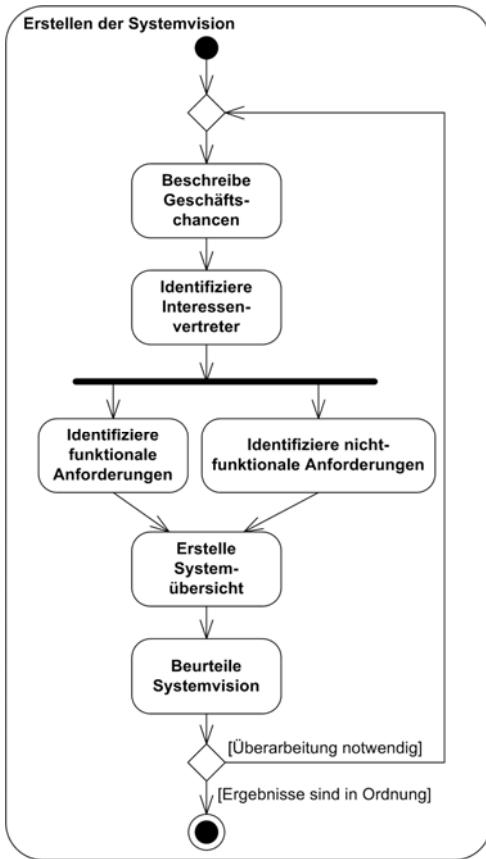


Abb. 8.3-2: Erstellen der Systemvision.

Tabelle 8.3-1 veranschaulicht hierzu den Geschäftskontext, die Geschäftschancen und die Problembeschreibung des MIS.

Tab. 8.3-1: Geschäftschancen und Problembeschreibung für MIS.

Geschäftskontext
Das MIS ist eingebettet in die globale Abteilung zur Betreuung der konzernweiten Warenwirtschaftssysteme (WWS-Support). Diese Abteilung bietet ihre Dienste allen Abteilungen an, welche die Warenwirtschaftssysteme benötigen.
Geschäftschancen
Das Management von WWS-Support möchte in der Lage sein, sowohl die Leistung ihrer globalen Abteilung als auch ihrer regionalen Unterabteilungen zu messen. Zu diesem Zweck sollen wesentliche Leistungsindikatoren (englisch: <i>key performance indicators</i>) erfasst und ausgewertet werden. Auf Basis der gewonnenen Erkenntnisse wird man in der Lage sein, Schwachstellen dediziert zu beheben und somit die Gesamtleistung zu verbessern.

Problembeschreibung	
Problem	Zurzeit stehen keinerlei Informationen über die qualitative und quantitative Leistung der WWS-Support-Abteilung zur Verfügung.
Betrifft	Management Team-Leiter
Auswirkungen	Es können keine Managementmaßnahmen zur Aufrechterhaltung und Verbesserung der Dienstleistungen unternommen werden.
Erfolgreiche Lösung	Die Transparenz über die tatsächliche Leistungsfähigkeit der Abteilung wird erhöht und fundierte Maßnahmen können ergriffen werden. Die Kundenzufriedenheit wird langfristig erhöht.
Aufgabe des Systems	Das MIS soll die Erfassung und Auswertung von Leistungsindikatoren ermöglichen.

Identifizierte Interessenvertreter

Neben den Geschäftschancen identifiziert eine Systemvision auch die Interessenvertreter des Systems. Da der Erfolg eines Systems von dessen Akzeptanz durch die Interessenvertreter abhängt, kommt dieser Aktion eine wichtige Bedeutung zu [Rozanski und Woods 2005]. Interessenvertreter sind zum einen die unmittelbaren Benutzer des Systems. Zum anderen gehören zu der Gruppe der Interessenvertreter auch Auftraggeber, Betreiber oder betroffene Abteilungen. Folgende Fragen können bei der Identifikation der Interessenvertreter nützlich sein:

- > Wer benutzt das System?
- > Wer ist von der Einführung des Systems betroffen?
- > Wem nutzt das System?
- > Wer betreibt das System?
- > Wer wartet das System?
- > Wer nimmt das System ab?
- > Welche Erwartungen haben die Interessenvertreter?
- > Welche Aufgaben, Kompetenzen und Verantwortlichkeiten haben die Interessenvertreter?

Es bietet sich an, die Interessenvertreter mittels eines einheitlichen Schemas zu beschreiben, welches die Antworten auf die oben gestellten Fragen widerspiegelt.

Erwartungsmanagement ist notwendig

Nicht immer sind Interessenvertreter positiv gegenüber der Entwicklung eines neuen Systems eingestellt. Beispielsweise könnte ein neues System das politische Machtgefüge zwischen Abteilungen verschieben. Im schlimmsten Fall könnte es zu einer Boykottierung des Vorhabens durch einzelne Interessenvertreter kommen. Daran wird wieder deut-

lich, dass auch soziale Aspekte für einen Architekten von großer Wichtigkeit sind (siehe Kapitel 7). Die Erwartungen von Interessenvertretern an das System und an die Architektur sollten also aktiv gehandhabt werden. Im Rahmen der Kommunikation der Architektur ist dieser Aspekt zu würdigen (siehe Abschnitt 8.7).

Tab. 8.3-2: Interessenvertreter von MIS.

Interessenvertreter	Rolle	Beschreibung
WWS-Management	Vertreibt Interessen des Managements.	Analysiert Leistungsindikatoren. Leitet daraus Aktionen ab.
Zentrale Architektur	Vertreibt zentrale architektonische Richtlinien.	Beurteilt die Konformität der MIS-Architektur mit den zentralen Richtlinien.
Systembetrieb	Vertreibt betriebliche Vorgaben.	Betreibt das MIS (Installation, Start, Stop, Update).
Teamleiter	Vertreibt die Anforderungen der Teamleiter.	Erfasst Daten zur Erhebung von Leistungsindikatoren.
Administrator	Vertreibt Anforderungen der Administratoren.	Verwaltet MIS-Benutzer. Initiiert Datenimports.
Controller	Vertreibt Anforderungen des Controllings.	Erstellt Leistungsberichte.

Tabelle 8.3-2 beinhaltet die beispielhaften Interessenvertreter des MIS. In Unternehmen existiert häufig eine zentrale Architektur-Abteilung, die unternehmensweite, architektonische Vorgaben definiert. Manchmal wird diese als Interessenvertreter zu spät wahrgenommen. Man sollte daher immer das organisatorische Umfeld analysieren und Interessenvertreter mit architektonischer Relevanz aufnehmen. Hierzu gehört beispielsweise auch der Systembetrieb, welcher später das System betreiben wird. Die Interessenvertreter können auch gemäß ihrer Wichtigkeit priorisiert werden, um deren Anforderungen später entsprechend zu beachten [Oestereich 2006].

Eine Systemvision sollte die wesentlichen Anforderungen, die durch das System zu erfüllen sind, dokumentieren. Dadurch wird gewährleistet, dass das Leistungsspektrum des Systems bereits früh festgehalten wird. Auf Basis dieses Spektrums werden während der Anforderungserhebung detaillierte Anforderungen abgeleitet. Neben den funktionalen Anforderungen müssen auch die relevanten nicht-funktionalen Anforderungen erfasst werden.

Interessenvertreter mit architektonischer Relevanz frühzeitig identifizieren

Wesentliche Anforderungen dokumentieren

Identifizierte funktionale Anforderungen

Bei der Identifikation der funktionalen Anforderungen geht es darum festzustellen, welche Funktionalität durch das System bereitgestellt werden soll (siehe Abschnitt 5.2). Dabei sollte der Fokus auf der primären Funktionalität liegen. Aus den wesentlichen funktionalen Anforderungen sind diejenigen auszuwählen, die architektonischen Charakter besitzen. Beispielsweise kann es notwendig sein, zur Realisierung mancher Anforderungen externe Systeme anzubinden. Die Integration dieser Systeme ist auf ihre Machbarkeit hin zu untersuchen. Tabelle 8.3-3 enthält einige beispielhafte funktionale Anforderungen von MIS.

Tab. 8.3-3: Wesentliche funktionale Anforderungen von MIS.

Funktionale Anforderung	Beschreibung
Anmelden	Benutzer müssen sich mit einer ID und einem Kennwort authentisieren, bevor MIS ihnen den Zugriff gewährt.
Leistungsdaten erfassen	Team-Leiter müssen Leistungsdaten für ihre Teams erfassen können. Administratoren müssen Leistungsdaten für beliebige Teams erfassen können.
Vorfallsdaten importieren	Vorfallsdaten müssen aus dem Vorfall-Management-System (VMS) in das MIS zur Leistungsindikatorenermittlung importiert werden.
CCM-Daten importieren	Configuration&Change-Management (CCM)-Daten müssen aus dem CCM-System (CCMS) in das MIS zur Leistungsindikatorenermittlung importiert werden.
Leistungsberichte erstellen	Controller müssen Leistungsberichte erstellen können.
Benutzerkonten verwalten	Administratoren müssen Benutzerkonten verwalten können.
Leistungsberichte anzeigen	Manager, Team-Leiter, Controller und Administratoren müssen Leistungsberichte betrachten können.

Identifizierte nicht-funktionale Anforderungen

Unglücklicherweise werden die nicht-funktionalen Anforderungen, die an ein System gestellt werden, häufig vernachlässigt. Für die Akzeptanz des Systems ist es jedoch essenziell, dass auch diese Anforderungen gewürdigt werden. Die unmittelbaren nicht-funktionalen Anforderungen bestimmen die erwartete Qualität der funktionalen Anforderungen (siehe Abschnitt 5.2). Beispielsweise kann für die Darstellung der Eingabemasken von MIS gefordert sein, dass 80 % der Masken in weniger als 5 Sekunden dargestellt werden. Des Weiteren können auch mittelbare nicht-funktionale Anforderungen auf das System einwirken. Solche, auch als Rahmenbedingungen bekannte Anforderungen, beschneiden den Gestaltungsraum eines Architekten. So kann die zentrale Architek-

tur für MIS beispielsweise vorgeben, dass das System auf Basis von JEE zu realisieren ist. Dadurch wird eine Realisierung des MIS mit PHP oder .NET bereits von vorneherein ausgeschlossen. Gerade die nicht-funktionalen Anforderungen müssen bereits früh begutachtet und hinterfragt werden. Darüber hinaus wird man stets vor der Herausforderung stehen, undokumentierte nicht-funktionale Anforderungen zu identifizieren und mit Interessenvertretern abzustimmen. Für das MIS können die in Tabelle 8.3-4 enthaltenen nicht-funktionalen Anforderungen genannt werden.

Tab. 8.3-4: Wesentliche nicht-funktionale Anforderungen von MIS.

Nicht-funktionale Anforderung	Beschreibung
Betreibbarkeit	MIS muss an die zentrale System-Management-Infrastruktur angeschlossen werden.
Erweiterbarkeit	MIS muss um neue Leistungsindikatoren und Leistungsberichte in durchschnittlich zehn Personentagen erweitert werden können.
Time-to-Market	MIS soll innerhalb von sechs Monaten entwickelt werden.
Einhaltung der IT-Standards	Die durch die zentrale Architektur vorgegebenen Richtlinien müssen eingehalten werden.
Sicherheit	Nur authentifizierte Benutzer dürfen auf Funktionalitäten und Daten, für die sie eine Berechtigung verfügen, zugreifen.
Bedienbarkeit	MIS muss eine kontextsensitive Online-Hilfe für Benutzer anbieten.
Verfügbarkeit	MIS soll rund um die Uhr verfügbar sein.
Performanz	80 % der Masken müssen in weniger als 5 Sekunden dargestellt werden.

Basierend auf den identifizierten wesentlichen Anforderungen kann mit der Erstellung der Systemübersicht begonnen werden. Nun wird das System das erste Mal in seinem Kontext betrachtet. Zu diesem Zweck behandelt man das System zunächst als Black Box (siehe Abschnitt 3.3) und identifiziert die menschlichen und systemischen Akteure, mit denen das System interagiert. Die menschlichen Akteure sind die Benutzer des Systems. Unter systemischen Akteuren werden die externen Systeme (Umsysteme) verstanden, mit denen das System kommuniziert. Der Systemkontext illustriert die Grenze des Systems. Im späteren Verlauf des Projekts wird der Systemkontext weiter verfeinert und konkretisiert werden.

Erstelle Systemübersicht

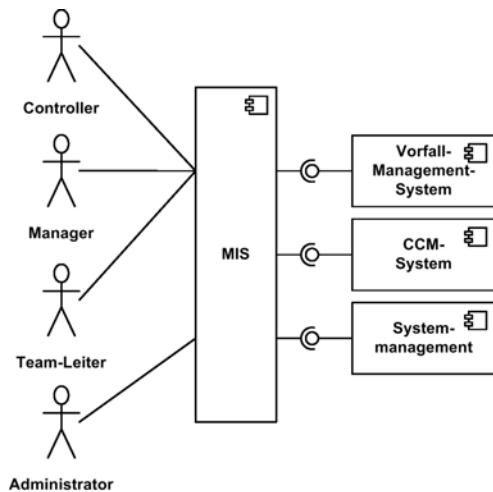


Abb. 8.3-3: Systemkontext von MIS.

Ein erster Systemkontext wird erstellt

Abbildung 8.3-3 veranschaulicht den Systemkontext von MIS. Darin finden sich zum einen die menschlichen Akteure (Rollen). Diese wurden aus den Interessenvertretern abgeleitet. Zum anderen enthält der Systemkontext auch die Umsysteme. In diesem Beispiel sind dies ein Vorfall-Management-System (VMS), ein Configuration&Change-Management-System (CCMS) sowie ein System-Management-System. Das VMS enthält Informationen zu Vorfällen, welche die WWS-Supportabteilung bearbeitet. Hierzu gehört z. B. das Einrichten oder Entsperren eines Benutzerkontos. Aus diesen Informationen kann geschlossen werden, wie lange die Bearbeitung eines Vorfalls gedauert hat. Falls Fehler im WWS festgestellt werden, müssen diese von der Entwicklungsabteilung behoben werden. Diese werden über Änderungsanträge (englisch: *change requests*) von der Support-Abteilung an die Entwicklungsabteilung übermittelt. Solche Änderungen werden nicht im VMS, sondern im CCMS verwaltet. Da für eine genaue Analyse sowohl Informationen aus dem CCMS als auch aus dem VMS vorliegen müssen, muss das MIS auch mit dem CCMS kommunizieren. Für den Betrieb von MIS ist es weiter notwendig, dass dieses an die zentrale System-Management-Infrastruktur angeschlossen wird (siehe Abschnitt 3.4). Dies resultiert nicht aus funktionalen Anforderungen, sondern vielmehr aus der nicht-funktionalen Anforderung der Betreibbarkeit heraus. Durch eine entsprechende Erweiterung des Systemkontexts, wird bereits der Grundstein für die spätere Würdigung dieser nicht-funktionalen Anforderung gelegt.

Neben dem Systemkontext entsteht während dieser Aktion eine erste grobe Dekomposition des Systems. Sie entspricht einer ersten architektonischen Vision. Die Vision ist in der Regel unscharf. Sie wird später beim Entwerfen der Architektur weiter konkretisiert. Die Darstellung kann an dieser Stelle sehr informell sein und sich auf die Identifikation wesentlicher Bausteine beschränken (siehe Abbildung 8.3-4). Häufig werden statt UML-Diagrammen auch sogenannte Boxes-and-Lines-Diagramme verwendet. Solche Diagramme sind sehr informell und erleichtern verschiedenen Interessenvertretern den Zugang zur Architektur-Vision. Manchmal dienen diese Art von Diagrammen auch als Einstieg für genauere formellere UML-Diagramme [Rozanski und Woods 2005].

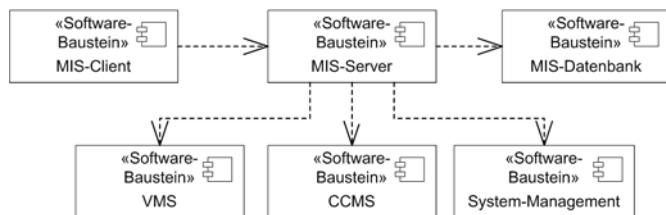


Abb. 8.3-4: Erste Architektur-Idee von MIS.

Nachdem die Systemübersicht erstellt wurde, sind die wesentlichen Aktionen zur Erstellung einer Systemvision beendet. Im Folgenden kann der Systemvision beurteilt werden. Diese Beurteilung erfolgt einerseits durch das Team, welches ihn erstellt hat, andererseits durch die Interessenvertreter. Im MIS-Fall beispielsweise durch das WWS-Management. In diesem Zusammenhang könnten auch nochmals die Anforderungen kritisch reflektiert werden. Der für das MIS verantwortliche Architekt könnte z. B. bemerken, dass die Verfügbarkeitsanforderung kein Wartungsfenster ermöglicht und auf diese Problematik hinweisen.

In Tabelle 8.3-5 ist festgehalten, welche Artefakte für die Architektursicht Anforderungssicht (siehe Abschnitt 4.2) im Rahmen der Tätigkeit Erstellen der Systemvision für MIS erarbeitet wurden.

Eine erste Architektur-Vision entsteht

Beurteile Systemvision

MIS:
Bearbeitete Architektur-Sichten

Tab. 8.3-5: Bearbeitete Architektur-Sichten beim „Erstellen der Systemvision“.

Architektur-Sicht	MIS-Artefakte
Anforderungssicht	<ul style="list-style-type: none"> > Geschäftschancen und Problembeschreibung (siehe Tabelle 8.3-1) > Interessenvertreter von MIS (siehe Tabelle 8.3-2) > Wesentliche funktionale Anforderungen (siehe Tabelle 8.3-3) > Wesentliche nicht-funktionale Anforderungen (siehe Tabelle 8.3-4)
Logische Sicht	<ul style="list-style-type: none"> > Systemkontext von MIS (siehe Abbildung 8.3-3) > Erste Architektur-Idee von MIS (siehe Abbildung 8.3-4)

**Checkliste:
Erstellen der
Systemvision**

- Wurden Geschäftschancen und Problembeschreibung dokumentiert?
- Haben sich alle Interessenvertreter eindeutig darauf verständigt, was die essentielle Aufgabe des zu entwickelnden Systems ist?
- Wurden alle architekturelevanten Interessenvertreter identifiziert?
- Sind alle wesentlichen Anforderungen dokumentiert?
- Werden nicht-funktionale Anforderungen ausreichend gewürdigt?
- Wurde ein Systemkontext erstellt?
- Umfasst der Systemkontext alle menschlichen und systemischen Akteure (Umsysteme)?
- Wurde eine Architektur-Vision erstellt?
- Wurde die Systemvision beurteilt?

Zusammenfassung

- > Ein Architekt übernimmt beim Erstellen der Systemvision die Rolle eines architektonischen Beraters in einem Team von Fachleuten aus unterschiedlichen Bereichen.
- > Manchmal spricht man statt von einer Systemvision auch von einem Business Case.
- > Eine Systemvision sollte die Geschäftschancen, die Interessenvertreter, die wesentlichen Anforderungen sowie eine erste Systemübersicht enthalten.
- > Die Interessenvertreter sind zu identifizieren und zu priorisieren.
- > Bereits während der Erstellung der Systemvision ist das Managen der Erwartungen der einzelnen Interessenvertreter notwendig.
- > Die wesentlichen funktionalen und nicht-funktionalen Anforderungen müssen dokumentiert werden.
- > Eine erste Systemübersicht sollte aus einem Systemkontext und einer ersten Architektur-Idee bestehen.

- > Die Darstellung der Architektur-Idee kann an dieser Stelle sehr informell sein und sich auf die Identifikation wesentlicher Bausteine beschränken.

8.4 Verstehen der Anforderungen

Abbildung 8.4-1 stellt die grundlegenden Konzepte der Tätigkeit „Verstehen der Anforderungen“, welche in diesem Abschnitt behandelt und detailliert werden, vor und visualisiert ihren Zusammenhang.

Grundlegende Konzepte der Tätigkeit „Verstehen der Anforderungen“

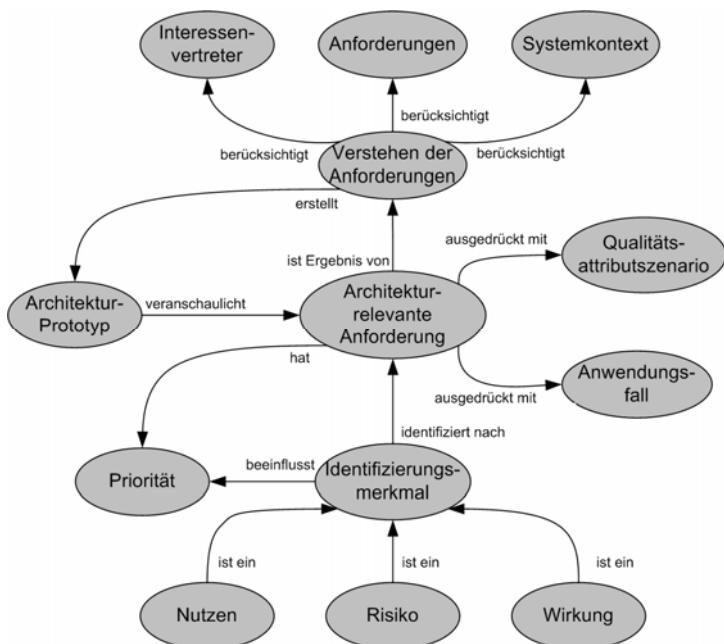


Abb. 8.4-1: Grundlegende Konzepte der Tätigkeit „Verstehen der Anforderungen“.

Ein gutes Verständnis der an eine Architektur gestellten Anforderungen ist eine wesentliche Voraussetzung für die Ausnutzung des Gestaltungsspielraums eines Architekten beim Entwurf der Architektur (siehe Abschnitt 8.5). Dies gilt sowohl für die funktionalen als auch nicht-funktionalen Anforderungen. Ohne ein adäquates Verständnis kann keine Architektur-Gestaltung stattfinden. Aufgrund dessen umfasst die architektonische Tätigkeit „Verstehen der Anforderungen“ die in Abbildung 8.4-2 dargestellten Aktionen. Im Rahmen dieser Tätigkeit wird primär die Anforderungssicht eines Systems (siehe Abschnitt 4.2) erarbeitet.

Verständnis von Anforderungen ist Voraussetzung für Architektur-Gestaltung

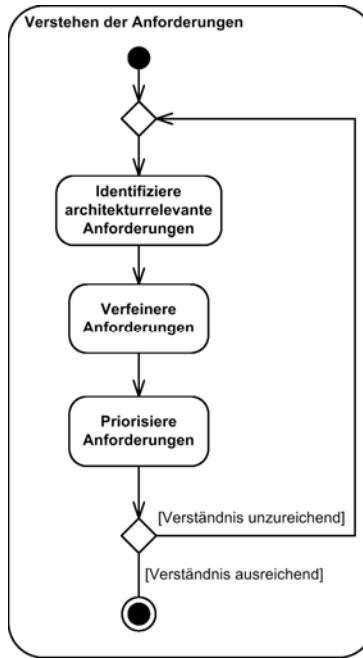


Abb. 8.4-2: Verstehen der Anforderungen.

Anforderungen sind nicht immer eindeutig

Unglücklicherweise sind Anforderungen nicht immer eindeutig dokumentiert. Vielmehr wird man in der Praxis häufig auf widersprüchliche und ungenaue Formulierungen treffen. Einige relevante Anforderungen sind wohlmöglich noch nicht einmal dokumentiert. Die Anforderung, dass das Beispielsystem MIS erweiterbar sein soll, ist eine sehr ungenaue Anforderung. Solch eine Anforderung muss konkretisiert werden. Eine genauere Formulierung wäre z. B. die Aussage, dass MIS um neue Leistungsindikatoren und Leistungsberichte in durchschnittlich zehn Personentagen erweitert werden können soll.

Anforderungen müssen greifbar sein

Widersprüche und Ungenauigkeiten bei den formulierten Anforderungen müssen ausgeräumt werden. Ferner müssen fehlende Anforderungen identifiziert und dokumentiert werden. Letztlich müssen Anforderungen greifbar gemacht werden. Durch klare, vorstellbare und messbare Formulierungen können Anforderungen mit Interessenvertretern leichter diskutiert werden.

Anforderungen müssen priorisiert werden

Anforderungen sind nicht gleichwertig. Einige werden einen höheren Nutzen als andere aufweisen. Ferner wird die Realisierung von Anforderungen mit unterschiedlichen Risiken behaftet sein. Basierend auf dem Nutzen und den Risiken müssen Anforderungen priorisiert werden.

Sowohl funktionale als auch nicht-funktionale Anforderungen können architekturelevant sein. In der Praxis werden funktionale Anforderungen, die sich auf Interaktionen mit einem System beziehen, häufig als Anwendungsfälle dokumentiert. Daher spricht man in diesem Zusammenhang auch oft von der Identifikation der architekturelevanten Anwendungsfälle. Nicht-funktionale Anforderungen lassen sich in Qualitäten und Rahmenbedingungen unterteilen. Qualitäten sind ein Maß für die Güte der Realisierung der funktionalen Anforderungen. Dass MIS rund um die Uhr verfügbar sein soll, ist ein Beispiel für eine Qualität. Rahmenbedingungen definieren die Art und Weise, wie funktionale Anforderungen und Qualitäten realisiert werden können (siehe Abschnitt 5.2). Die für MIS vorgegebene Entwicklungsdauer von sechs Monaten ist eine Bedingung, die den architektonischen Gestaltungsspielraum einschränkt. Sie stellt folglich eine Rahmenbedingung dar.

In dieser Tätigkeit sind die architekturelevanten Anforderungen zu identifizieren. Es stellt sich die Frage, was eine architekturelevante Anforderung wirklich ist und wie diese aus der Summe aller Anforderungen extrahiert wird. Häufig erscheinen alle Anforderungen auf dem ersten Blick als architekturelevant. Für einen Architekten ist es daher wichtig, Anforderungen systematisch zu klassifizieren. Anforderungen werden von Interessenvertretern gestellt. Interessenvertreter haben eine unterschiedliche Priorität. Daher kann eine erste Filterung der Anforderungen bereits nach der Priorität der Interessenvertreter erfolgen. Anforderungen sind für Interessenvertreter von unterschiedlicher Relevanz, weil sie sich in ihrem Nutzen unterscheiden. Darüber hinaus ist die Realisierung von Anforderungen mit verschiedenen Risiken behaftet. Ferner wirken manche Anforderungen stärker auf den architektonischen Gestaltungsspielraum ein als andere. Anforderungen sollten nach den Kriterien Nutzen, Risiko und Wirkung beurteilt werden (siehe Abbildung 8.4-3). Kandidaten für architekturelevante Anforderungen sind Anforderungen mit einem hohen Nutzen für die Interessenvertreter, mit einem hohen Umsetzungsrisiko oder einer großen Auswirkung auf die Architektur. Im Folgenden wird dies an einigen Beispielen für das MIS genauer betrachtet.

Wichtige Anforderungen sind die wesentlichen Anforderungen mit dem höchsten Nutzen für die Interessenvertreter. Das Erfassen von Leistungsdaten ist z. B. für die MIS-Interessenvertreter von hoher Wichtigkeit, da ohne dieses Feature keinerlei Leistungsberichte erstellt werden können. Dahingegen ist das Versenden von E-Mail-Benachrichtigungen, sobald neue Leistungsberichte zur Verfügung stehen, von niedrigerer Wichtigkeit.

Identifizierte architekturelevante Anforderungen

Was ist eigentlich architekturelevant?

Identifikation nach Nutzen

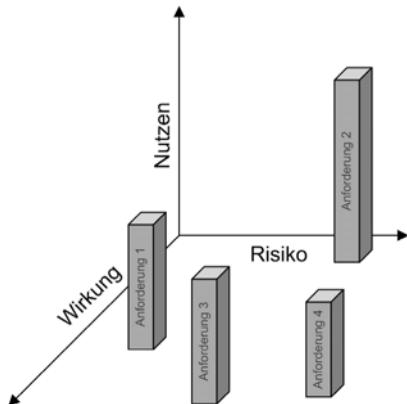


Abb. 8.4-3: Kriterien zur Identifikation von architekturelevanten Anforderungen.

Identifikation nach Risiko

Architektonisch risikoreiche Anforderungen sind typischerweise Anforderungen, mit deren Realisierung man noch keinerlei Erfahrungen gesammelt hat oder die Bereiche betreffen, die man nicht (vollständig) unter eigener Kontrolle hat (z. B. Bausteine eines externen Partners). Aufgrund dessen ist die Realisierung solcher Anforderungen mit einem hohen Risiko behaftet, da deren erfolgreiche Realisierung hinsichtlich Machbarkeit sowie Kosten- und Zeitvorgaben ungewiss ist. Anforderungen, die Umsysteme involvieren, sind hier als ein Beispiel zu nennen. Wenn man im Fall von MIS beispielsweise das VMS-System über einen JCA-Adapter an seine JEE-Anwendung anschließen muss, aber noch keine Erfahrungen in diesem Bereich gesammelt hat, sollte diese Anforderung als risikoreich betrachtet und als architekturelevant charakterisiert werden.

Identifikation nach Wirkung

Anforderungen, deren Umsetzung eine große Wirkung auf das zu realisierende System haben, sollten ebenfalls möglichst früh begutachtet und als architekturelevant betrachtet werden. Die zentrale Vorgabe von IT-Richtlinien hat z. B. eine große Wirkung auf ein zu realisierendes System, da die Nichteinhaltung dieser Richtlinien zur Verweigerung der Abnahme des Systems führen kann. Darüber hinaus gehören Anforderungen zu dieser Kategorie, deren Realisierung einen großen Aufwand mit sich bringt, wenn sie nicht früh bei der Architektur-Gestaltung berücksichtigt werden. So ist es beispielsweise häufig aus Sicherheits- und Nachvollziehbarkeitsgründen wesentlich, dass durchgeführte Aktionen innerhalb eines Systems protokolliert werden. In diesem Zusammenhang spricht man oft von Auditing. Eine solche Anforderung ist nur mit hohem Aufwand zu einem späten Zeitpunkt in die Architektur und vor allem in die existierende Systemimplementierung zu integrie-

ren. Ein weiteres Merkmal solcher Anforderungen ist, dass sie auf viele architekturrelevante Bausteine einwirken. Ein Anwendungsfall, dessen Realisierung sowohl Bausteine aus der Präsentationslogik-, Geschäftslogik- und Persistenzlogik-Schicht (siehe Abschnitt 3.4) involviert, ist in der Regel architekturrelevant. Dabei gilt es, aus der Fülle der Anwendungsfälle, Kandidaten auszuwählen, die unterschiedliche architekturrelevante Bausteintypen benötigen.

Das Verfeinern der architekturrelevanten Anforderungen beschäftigt sich primär mit dem Greifbarmachen von Anforderungen. Hierzu gehören die effektive Dokumentation der Anforderungen, die Verbindung von funktionalen und nicht-funktionalen Anforderungen sowie das Veranschaulichen von Anforderungen.

Wie bereits erwähnt, hat es sich in der Praxis etabliert, funktionale Anforderungen mittels Anwendungsfällen zu beschreiben. Ein in der Praxis noch nicht so häufig eingesetztes Mittel zur Dokumentation von Qualitäten ist das sogenannte Qualitätsattributszenario (siehe Abschnitt 6.3). Manche Qualitäten, wie die Verfügbarkeit von MIS, gelten für das System als Ganzes. Andere Qualitäten gelten jedoch nur für einen Teil der funktionalen Anforderungen. Bei MIS müssen Controller z. B. in der Lage sein, Leistungsberichte zu erstellen. Außerdem soll MIS den sicheren Zugriff auf Daten gewährleisten. Eine Kombination dieser beiden Anforderungen würde dazu führen, dass Controller nur für Teams Leistungsberichte erstellen dürfen, für die sie eine passende Berechtigung besitzen. Qualitätsattributszenarien können sowohl für systemweite als auch für anwendungsfallspezifische Qualitäten genutzt werden.

Es liegt nahe, architekturrelevante Anwendungsfälle mit architekturrelevanten Qualitäten in Verbindung zu bringen. Hierzu betrachtet man jeden architekturrelevanten Anwendungsfall hinsichtlich seiner qualitativen Realisierungsgüte. Je mehr verschiedene Qualitätsmerkmale (z. B. Sicherheit, Nachvollziehbarkeit, Performanz) ein Anwendungsfall ausweist, umso größer ist sein architektonischer Charakter. Für jede Kombination können Qualitätsattributszenarien erstellt werden. Dies wird im Folgenden am Beispiel des Erstellens von Leistungsberichten beim MIS besprochen (siehe Tabelle 8.4-1).

Verfeinerung der Anforderungen

Dokumentation von Qualitäten mit Qualitätsattributszenarien

Verbinden von funktionalen Anforderungen und Qualitäten

Tab. 8.4-1: Anwendungsfallspezifische Qualitätsattributszenarien.

Leistungsberichte erstellen und Sicherheit	
Kriterium	Bedeutung
Quelle	Identifizierter Controller
Stimulus	Versucht, einen Leistungsbericht für Teams zu erstellen, für die er nicht berechtigt ist.
Artefakt	Leistungsberichtsdaten im System
Kontext	System befindet sich im Normalzustand.
Reaktion	System verweigert den Zugriff auf die Leistungsdaten und protokolliert den Zugriffsversuch.
Reaktionsmessgröße	System hat den Zugriff verweigert und ein Protokolleintrag ist vorhanden.
Leistungsberichte erstellen und Nachvollziehbarkeit	
Kriterium	Bedeutung
Quelle	Identifizierter Controller
Stimulus	Erstellt Leistungsberichte
Artefakt	Leistungsberichtsdaten im System
Kontext	System befindet sich im Normalzustand.
Reaktion	System erzeugt die Leistungsberichte und protokolliert die durchgeführten Aktionen.
Reaktionsmessgröße	Leistungsberichte sind vorhanden und Aktionen wurden protokolliert.
Leistungsberichte erstellen und Performanz	
Kriterium	Bedeutung
Quelle	Identifizierter Controller
Stimulus	Erstellt Leistungsberichte
Artefakt	Leistungsberichtsdaten im System
Kontext	System befindet sich im Normalzustand.
Reaktion	System erzeugt die Leistungsberichte und protokolliert die durchgeführten Aktionen.
Reaktionsmessgröße	Die durchschnittliche Erstellungsdauer beträgt 5 Sekunden.

Architekturrelevante Anwendungsfälle und Qualitäten innerhalb der gesetzten Rahmenbedingungen betrachten

Abschließend sollte untersucht werden, wie die Qualitätsattributszenarien innerhalb der definierten Rahmenbedingungen realisiert werden können. Dabei wird häufig deutlich werden, dass die Erreichung eines gewünschten Qualitätsattributszenarios innerhalb der gesetzten Rahmenbedingungen nicht möglich ist. In diesem Fall ist man gefordert, auf diese Widersprüche hinzuweisen und Interessenvertretern Alternativen aufzuzeigen.

Zur Veranschaulichung von dokumentierten Anforderungen können unterschiedliche Mittel eingesetzt werden (z. B. Prototypen). Durch die Veranschaulichung von Anforderungen erscheinen diese nochmals unter einem anderen Licht und neue Erkenntnisse können gewonnen werden.

Anforderungen veranschaulichen

Eine Möglichkeit, ein besseres Verständnis für die einzelnen architekturelevanten Anforderungen zu erhalten und deren Machbarkeit zu prüfen, ist der Einsatz von Prototypen. Hinsichtlich erst später auftretender möglicher Probleme tragen Prototypen damit zur Risikominimierung bei. Ist man beispielsweise mit neuen Technologien konfrontiert, kann man diese mittels eines Prototyps beurteilen. Im Fall von MIS könnte sich der Architekt mit dem Einsatz von JCA als Integrationsmittel auseinandersetzen. Die Prototypen im Rahmen der Tätigkeit „Verstehen der Anforderungen“ spiegeln nicht die spätere Architektur des Systems wider, sondern sollen nur helfen, ein besseres Verständnis von gestellten Anforderungen zu gewinnen. Bei der Anforderungserhebung ist es auch üblich, Prototypen zur Veranschaulichung der Benutzeroberfläche zu erstellen. Hinsichtlich der Bedienbarkeit können aus diesen ebenfalls wertvolle architektonische Erkenntnisse gewonnen werden. Ist beispielsweise gefordert, dass Benutzeranfragen abgebrochen (Abort-Feature) oder mehrere Aktionen zurückgenommen (Undo-Feature) werden können, so sind solche Aspekte von architektonischer Bedeutung.

Architektur-Prototypen machen Anforderungen anschaulich

Wie in Kapitel 5 aufgezeigt, treten Anforderungen auf den unterschiedlichen Architektur-Ebenen auf. Die einzelnen Anforderungen stehen über die Architektur-Ebenen hinweg in Beziehung. So verfeinern sie von Architektur-Ebene zu Architektur-Ebene schrittweise das zu erstellende System. Beim Ausüben dieser Tätigkeit wird man also die Anforderungen von der Organisations-, über die System- bis hin zur Bausteinebene innerhalb mehrerer Iterationen verfeinern.

Schrittweises Verfeinern der Anforderungen

Architektonisch relevante Anforderungen werden erst richtig vollständig und konsistent im Zuge von Entwurf und Umsetzung einer Architektur. Dies wird impliziert durch die Iterationen im hier vorgestellten architektonischen Vorgehen. Während der Tätigkeiten „Entwerfen der Architektur“ und „Umsetzen der Architektur“ können sich als Rückkopplung neue Anforderungen ergeben oder es kann die Notwendigkeit entstehen, dass bestehende Anforderungen angepasst werden müssen. In der jeweiligen Folgeiteration sind dann im Rahmen der Tätigkeit „Verstehen der Anforderungen“ die neuen bzw. anzupassenden Anforderungen zu behandeln.

Keine vollständigen Anforderungen ohne umgesetzte Architektur

Anforderungen priorisieren

Eine weitere Aktion während des Verstehens der Anforderungen ist deren Priorisierung. Die Priorisierung ergibt sich aus der Klassifizierung der Anforderungen nach Nutzen, Risiko und Wirkung. Anforderungen mit einem hohen Nutzen, einem hohen Risiko oder einer großen Wirkung sollten möglichst früh adressiert werden. Dies sind die Anforderungen, denen eine Architektur gerecht werden muss. Eine beispielhafte Priorisierung der funktionalen und nicht-funktionalen Anforderungen für das MIS zeigen die Tabellen 8.4-2 und 8.4-3. Die Priorität wird hier jeweils mit der Formel Priorität = (Nutzen + Risiko + Wirkung) / 3 bestimmt.

Tab. 8.4-2: Priorisierte funktionale Anforderungen.

Funktionale Anforderung	Nutzen	Risiko	Wirkung	Priorität
	(1 = hoch, 3 = niedrig)			
Leistungsberichte anzeigen	1	1	1	1
Vorfallsdaten importieren	1	1	2	1
CCM-Daten importieren	1	1	2	1
Leistungsdaten erfassen	1	3	1	2
Leistungsdaten drucken	2	2	3	2
Leistungsberichte erstellen	2	3	2	2
Anmelden	3	1	2	2
Benutzerkonten verwalten	3	2	2	2
e-Mail versenden	3	2	3	3

Tab. 8.4-3: Priorisierte nicht-funktionale Anforderungen.

Nicht-funktionale Anforderung	Nutzen	Risiko	Wirkung	Priorität
	(1 = hoch, 3 = niedrig)			
Betreibbarkeit	1	1	1	1
Erweiterbarkeit	1	1	1	1
Time-to-Market	2	1	1	1
Einhaltung der IT-Standards	1	1	3	2
Sicherheit	1	1	3	2
Bedienbarkeit	2	2	2	2
Verfügbarkeit	2	3	3	3
Performanz	3	2	3	3

Risiko „zu guter“ Architekturen

Eine Schwäche (Risiko), der Architekten typischerweise leicht verfallen ist, dass sie die Architektur perfektionieren möchten. Dies ist dann der Fall, wenn bestimmte nicht-funktionale Anforderungen unangemessen

stark berücksichtigt werden, obwohl diese nur eine geringe oder gar keine Rolle im konkreten Fall spielen. Beispielsweise, wenn eine Austauschbarkeit der grafischen Benutzerschnittstelle in einer Architektur vorgesehen wird (nicht-funktionale Anforderung Erweiterbarkeit), obwohl das System ausschließlich über eine Desktop-Oberfläche einer bestimmten Plattform bedient werden soll. Unnötig komplexe Architekturen respektive Systeme, die nicht „fertig“ werden, oder Systeme, die nicht ausreichend den Anforderungen genügen, sind Beispiele für mögliche negative Folgen. Als Disziplinierungsmaßnahme im eigenen Interesse sollte ein Architekt deshalb z. B. mittels einer Tabelle explizit machen, welche der allgemein bekannten nicht-funktionalen Anforderungen überhaupt eine Relevanz für die Qualitätseigenschaften eines Systems haben. Tabelle 8.4-4 zeigt dies am Beispiel des MIS.

Tab. 8.4-4: Qualitätsrelevanz nicht-funktionaler Anforderungen.

Nicht-funktionale Anforderung	Relevanz für die Qualität (1 = hoch, 3 = niedrig)
Benutzerfreundlichkeit	1
Herstellerunabhängigkeit	1
Interoperabilität	1
Performanz	1
Portabilität	2
Robustheit	2
Sicherheit	2
Skalierbarkeit	2
Testbarkeit	2
Wartbarkeit	3
Erweiterbarkeit	3
Veränderbarkeit	3

Beachtet werden sollte jedoch, dass das Priorisieren von Anforderungen (wie andere Metriken auch) ein nützliches Hilfsmittel ist, das einem Architekten eine Orientierung (Tendenz), nicht jedoch die exakte Richtung an die Hand gibt (siehe hierzu auch Abschnitt 8.6).

In Tabelle 8.4-5 ist festgehalten, welche Artefakte für die Architektur-Sicht Anforderungssicht (siehe Abschnitt 4.2) im Rahmen der Tätigkeit „Verstehen der Anforderungen“ für MIS erarbeitet wurden.

Priorisierung von Anforderungen gibt „nur“ Orientierung

**MIS:
Bearbeitete
Architektur-Sicht**

Tab. 8.4-5: Bearbeitete Architektur-Sicht beim „Verstehen der Anforderungen“.

Architektur-Sicht	MIS-Artefakte
Anforderungs-sicht	<ul style="list-style-type: none"> > Anwendungsfallspezifische Qualitätsattributszenarien (siehe Tabelle 8.4-2) > Priorisierte funktionale Anforderungen (siehe Tabelle 8.4-3) > Priorisierte nicht-funktionale Anforderungen (siehe Tabelle 8.4-4) > Qualitätsrelevanz nicht-funktionaler Anforderungen (siehe Tabelle 8.4-5)

Checkliste: Verstehen der Anforderungen

- Wurden alle architekturelevanten Anforderungen identifiziert?
- Werden nicht-funktionale Anforderungen ausreichend gewürdigt?
- Sind alle architekturelevanten Anforderungen messbar?
- Schließen sich keine architekturelevanten Anforderungen aus?
- Wurden alle architekturelevanten Anforderungen priorisiert?
- Wurden alle architekturelevanten Anforderungen verfeinert?
- Wurden alle architekturelevanten Anforderungen dokumentiert?
- Wurden die architekturelevanten Anforderungen von den Interessenvertretern bestätigt?

Zusammenfassung

- > Anforderungen beschränken den Gestaltungsspielraum eines Architekten und sind selten genau.
- > Anforderungen müssen identifiziert, greifbar gemacht und priorisiert werden.
- > Architekturelevante Anforderungen können nach Nutzen, Risiko und Wirkung identifiziert werden.
- > Qualitäten können mit Qualitätsattributszenarien dokumentiert werden.
- > Der qualitative Charakter funktionaler Anforderungen kann ebenfalls mit Qualitätsattributszenarien festgehalten werden.
- > Architekturelevante Anwendungsfälle und Qualitäten sollten innerhalb der gesetzten Rahmenbedingungen betrachtet werden.
- > Architekturelevante Anforderungen können mit Architektur-Prototypen veranschaulicht werden.
- > Beim Ausüben der Tätigkeit „Verstehen der Anforderungen“ werden die Anforderungen von der Organisations-, über die System- bis hin zur Bausteinenebene innerhalb mehrerer Iterationen verfeinert.

8.5 Entwerfen der Architektur

Abbildung 8.5-1 stellt die grundlegenden Konzepte der Tätigkeit „Entwerfen der Architektur“, welche in diesem Abschnitt behandelt und detailliert werden, vor und visualisiert ihren Zusammenhang.

Grundlegende Konzepte der Tätigkeit „Entwerfen der Architektur“

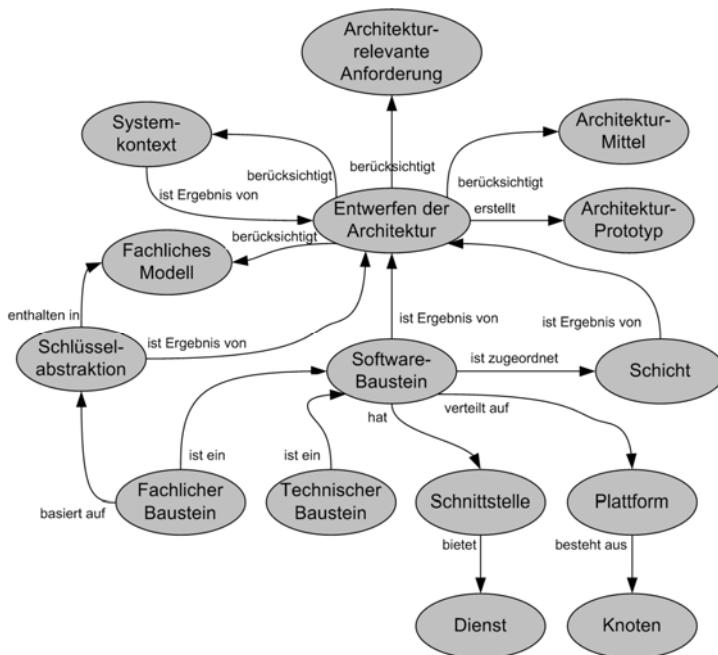


Abb. 8.5-1: Grundlegende Konzepte der Tätigkeit „Entwerfen der Architektur“.

Systeme sind eingebettet in Organisationen, werden für bestimmte Fachgebiete respektive Fachdomänen erstellt und müssen dedizierte Anforderungen erfüllen (siehe Abschnitt 3.3). Wie aus Abbildung 8.5-2 ersichtlich, muss diesem Sachverhalt beim Entwurf der Architektur Rechnung getragen werden. Die architekturelevanten Anforderungen werden beim Verstehen der Anforderungen identifiziert und genauer analysiert. Fachliche Modelle entstehen im Rahmen der Disziplin „Analyse und Entwurf“. Da das Entwerfen der Architektur ebenfalls innerhalb dieser Disziplin ausgeführt wird, erfolgt während dieser Tätigkeit auch eine architektonische Analyse fachlicher Modelle. Des Weiteren wird der Systemkontext berücksichtigt und verfeinert. Darüber hinaus dienen Architekturmittel der Gestaltung der Architektur (siehe Kapitel 6).

Einflussfaktoren auf den Entwurf einer Architektur

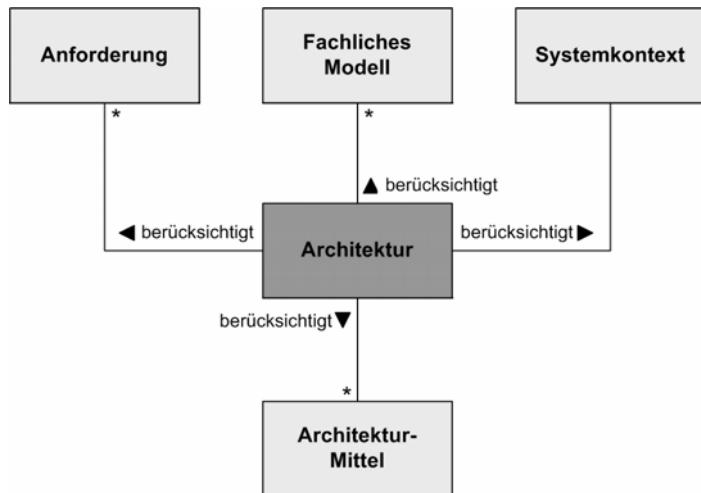


Abb. 8.5-2: Einflussfaktoren des Architektur-Entwurfs.

Entwurfsaktionen im Überblick

Die einzelnen Aktionen dieser Tätigkeit sind auf die Einflussfaktoren abgestimmt. Abbildung 8.5-3 zeigt diese Aktionen im Überblick. Das Ergebnis der ersten Iteration wird oftmals als Architektur-Vision bezeichnet, da in dieser Iteration die wesentlichen Entscheidungen zur Strukturierung des Systems getroffen werden. Das erste Architektur-Inkrement enthält die grundlegenden Systembausteine, deren Verantwortlichkeiten und Interaktionen. Die Architektur-Vision legt auch die Technologien (siehe Abschnitt 6.7) für die spätere Umsetzung fest. Im weiteren Verlauf werden die verschiedenen Aktionen näher besprochen. Im Rahmen dieser Tätigkeit werden primär die logische, die Daten-, die Prozess- sowie die Verteilungssicht eines Systems (siehe Abschnitt 4.2) erarbeitet.

Definiere Kontext

Beim Erstellen der Systemvision erfolgt bereits die Definition eines ersten Systemkontexts (siehe Abschnitt 8.3). Dieser wird im Rahmen dieser Tätigkeit verfeinert und konkretisiert. Oftmals arbeiten unterschiedliche Architekten an der Erstellung der Systemvision und dem Entwerfen der Architektur. Sollte dies der Fall sein, muss der Systemkontext an dieser Stelle umso genauer reflektiert werden, um mögliche Lücken zu schließen. Im Fall von MIS wird beispielsweise gefordert, dass sich Benutzer authentifizieren müssen und dass zentrale Richtlinien einzuhalten sind. Eine dieser Richtlinien könnte vorschreiben, dass jedes System an die zentrale Identity-and-Access-Management-Infrastruktur (IAM-Infrastruktur) angeschlossen werden muss. Daraus ergäbe sich die Notwendigkeit, den Systemkontext, um das IAM-System zu erweitern (siehe Abbildung 8.5-4).

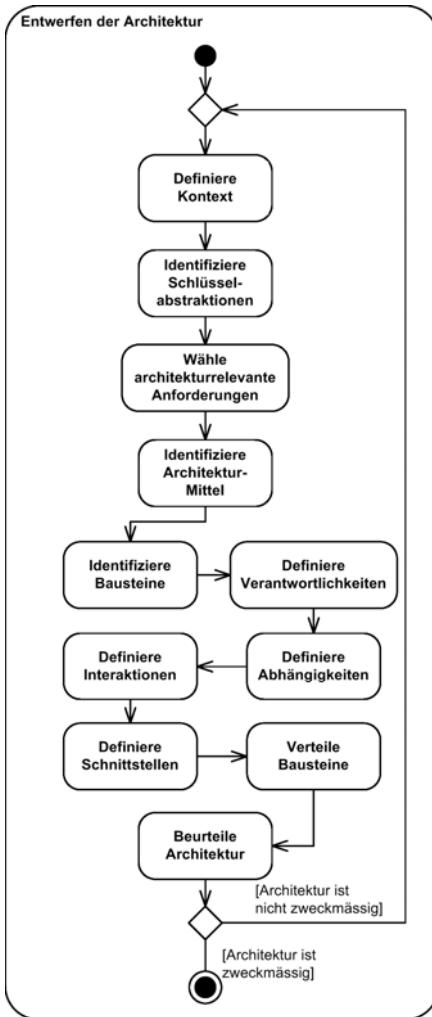


Abb. 8.5-3: Entwerfen der Architektur.

Ferner werden in dieser Aktion auch die Schnittstellen zwischen dem System und seinen Akteuren konkretisiert, indem beispielsweise folgende Fragestellungen behandelt werden:

- > Welche Dienste werden von den Akteuren des Systems benötigt?
- > Welche Dienste benötigt das System von den angeschlossenen Systemen?
- > Welche Entitäten werden über die Schnittstelle übertragen und welche Bedeutung haben diese?
- > Welche Schnittstellentechnologien kommen zum Einsatz?

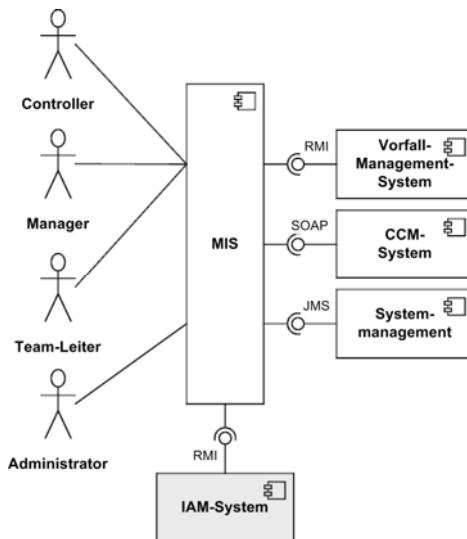


Abb. 8.5-4: Erweiterter Systemkontext von MIS.

Ein Systemkontext ist mehr als nur ein Diagramm

Ein Systemkontext ist mehr als nur ein Diagramm. Vielmehr ist es ein Artefakt, welches jede Schnittstelle detailliert dokumentiert. Näheres zum Thema Systemkontext findet sich in Abschnitt 8.3.2. Kontexte können nicht nur auf Systemniveau, sondern auch auf dem Niveau feingranularerer Systembausteine (z. B. Subsysteme) verwendet werden. Aus dem Systemkontext können auch dedizierte Schnittstellenvereinbarungen abgeleitet werden, die eigene Artefakte repräsentieren und eine Schnittstelle definieren.

Identifizierte Schlüsselabstraktionen

Wie in Abschnitt 3.3 bereits erwähnt, umfasst eine Architektur sowohl fachliche als auch technische Aspekte. Daher werden in der Regel eine fachliche und eine technische Architektur entworfen. Da sich die fachliche Architektur an der Fachdomäne orientiert, wird in dieser Aktion die Fachdomäne hinsichtlich ihrer Schlüsselabstraktionen analysiert. Schlüsselabstraktionen repräsentieren die wesentlichen Abstraktionen einer Fachdomäne, die von dem zu realisierenden System behandelt werden müssen. Beispiele sind Abstraktionen von Gegenständen, Konzepten, Orten oder Personen [Oestreich 2006]. Im Fall von MIS sind dies beispielsweise Leistungsbericht, Leistung oder Supportabteilung (siehe Abbildung 8.5-5). Begriffe wie Primärgeschäftsentität, -objekt oder Kernetität bzw. -objekt werden häufig als Synonyme für den Begriff Schlüsselabstraktion verwendet. Aus den Schlüsselabstraktionen und den funktionalen Anforderungen kann die fachliche Architektur abgeleitet werden.

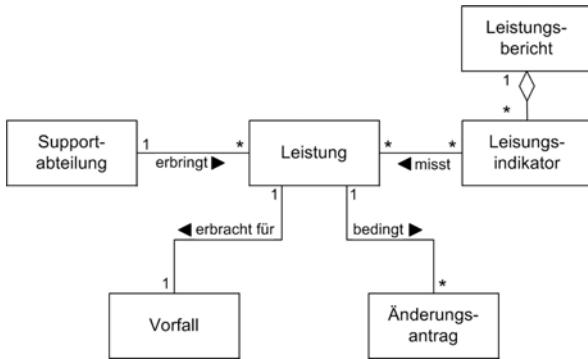


Abb. 8.5-5: Schlüsselabstraktionen von MIS.

In dieser Aktion werden die architekturelevanten Anforderungen selektiert, die man in der aktuellen Iteration bei der Architektur-Gestaltung berücksichtigen möchte. Hierzu greift man auf die beim Verstehen der Anforderungen entstandenen Qualitätsattributzenarien zurück. Anforderungen mit einer hohen Priorität sollten möglichst früh in eine Architektur einfließen.

Wähle architekturelle relevante Anforderungen

Nachdem nun neben den Anforderungen und dem Systemkontext auch die Schlüsselabstraktionen bekannt sind, können in einem nächsten Schritt geeignete Architektur-Mittel identifiziert werden. Sie sollten aus rein zweckmäßigen Überlegungen (an den vorliegenden Anforderungen ausgerichtet) ausgewählt werden. Man sollte nicht irgendeinem Hype folgen, dem Goldenen-Hammer- (englisch: *golden hammer*) bzw. Silberkugel-Syndrom (englisch: *silver bullet*) (Ein Mittel für ein bestimmtes Problem wird als Allheilmittel für alle anderen Probleme missbraucht bzw. ein neues Mittel soll ein bestehendes Problem vermeintlich ohne Weiteres lösen können) verfallen oder politischen Überlegungen Vorrang geben. Ein Architektur-Mittel alleine löst noch nicht alle Probleme. Vielmehr muss es korrekt angewendet, adaptiert und ggf. mit eigenen Entwurfsideen kombiniert werden. Dies impliziert, dass trotz Auswahl eines angemessenen Architektur-Mittels immer noch das Risiko besteht, es falsch einzusetzen. Zwei einfache Beispiele sollen dies verdeutlichen:

- > Wird das Muster MVC (siehe Abschnitt 6.3) benutzt, kann in der Folge ein MVC-Framework (z. B. Struts) zum Einsatz kommen. Werden dann jedoch (z. B. aus Bequemlichkeit) bestimmte Bausteine (z. B. Wertobjekte) dieses Frameworks in der Geschäftslogik-Schicht benutzt, ergibt sich dort eine enge Kopplung mit der Präsentationsschicht. Gerade diese Kopplung wollte man jedoch mithilfe des Musters MVC vermeiden.
- > Findet die Objektorientierung bei der Entwicklung eines verteilten Systems Anwendung (z. B. mittels Entitätsobjekten mit JEE), dann

Identifizierte Architektur-Mittel

Der Fokus verschiebt sich von der Fachlichkeit zur Technik

besteht die Gefahr einer unnötig hohen Netzwerklast und damit einer unzureichenden Performanz. Dies würde dann eintreten, wenn man auf die einzelnen Attribute der Entitätsobjekte gemäß der reinen Lehre der Objektorientierung mittels Methoden zugreifen würde.

Identifikation geeigneter Mittel

Sobald die grundlegenden Strukturen des zu realisierenden Systems entworfen wurden, wird sich der Fokus von der Fachlichkeit zur Technik verschieben. Damit werden die technologischen Architektur-Mittel an Bedeutung gewinnen. So wird man beispielsweise die konkreten Anwendungsserver, Datenbanken oder Middleware-Systeme festlegen.

Identifizierte Architektur-Mittel für die fachliche Architektur

Im Fall von MIS wäre es ideal, wenn bereits eine Referenzarchitektur für ein solches LeistungsindikatorSystem existieren würde. In vielen Fällen wird dies nicht der Fall sein. Dann lohnt sich der Blick über den eigenen Projekttellerrand, um festzustellen, ob bereits andere Projekte ähnliche Systeme entwickelt haben.

Warum werden Architektur-Mittel für fachliche Belange benötigt?

Um Architektur-Mittel für eine fachliche Architektur auszuwählen, sollte zunächst geprüft werden, ob für die gegebene Fachdomäne eine Referenzarchitektur existiert, da diese den höchsten Wiederverwendungsgrad aufweist. Abschnitt 6.5.4 nennt Beispiele für solche Referenzarchitekturen. Fachliche Referenzarchitekturen haben einen ähnlichen Charakter wie Analyse-Muster [Fowler 1996]. Allerdings befinden sie sich im Gegensatz zu Analyse-Mustern auf dem Niveau von grobgraularen Software-Bausteinen und nicht von Klassen. Eine fachliche Referenzarchitektur für das MIS-Beispiel würde bereits die fachlichen Bausteine benennen, die notwendig sind, um die Schlüsselabstraktionen zu behandeln und die funktionalen Anforderungen zu erfüllen. Finden sich keine geeigneten Referenzarchitekturen, können unter Verwendung der Architektur-Prinzipien, wie z. B. Separation of Concerns, lose Kopplung und hohe Kohäsion (siehe Abschnitt 6.1) die fachlichen Bausteine bestimmt werden.

Es stellt sich die Frage, warum man überhaupt eine fachliche Dekomposition des Software-Systems vornimmt. Existiert hierfür eine funktionale Anforderung? Solange das System die gewünschten funktionalen Anforderungen erfüllt, sollte es doch gleichgültig sein, wie es fachlich strukturiert ist. Auf den ersten Blick ist dies korrekt. Allerdings beeinflussen auch die nicht-funktionalen Anforderungen die fachliche Architektur. Beispielsweise erzwingt der Wunsch nach der Erweiterbarkeit des Systems oder nach der parallelen Entwicklung durch mehrere Teams die Strukturierung in sinnvolle, handhabbare Einheiten respektive Bausteine.

Der Fundus von Mitteln zur Gestaltung von technischen Architekturen ist größer als der von fachlichen Architekturen. Daher wird man hier in der Regel leichter passende Architektur-Mittel finden. Ausgangspunkt für die Wahl der Mittel sind die nicht-funktionalen Anforderungen. IT-Standards eines Unternehmens geben z. B. häufig die einzusetzende Komponentenplattform vor. Für die jeweilige Plattform, wie JEE oder .NET, finden sich in der Regel passende plattform-bezogene Referenzarchitekturen (siehe Abschnitt 6.5.4). Diese können in jedem Projekt unverändert oder mit leichten Anpassungen wiederverwendet werden [Siedersleben 2006]. Solche Referenzarchitekturen legen auch fest, welche Arten von technischen Bausteinen benötigt werden, um verschiedene nicht-funktionale Anforderungen zu erfüllen. Falls keine Referenzarchitekturen existieren, sollte man sein Augenmerk als nächstes auf Basisarchitekturen (siehe Abschnitt 6.4) sowie Architekturtaktiken, -Stile und -Muster (siehe Abschnitt 6.3) legen.

Eine erste Hilfestellung bei der Identifizierung der richtigen Architekturmittel findet man im Zusammenhang zwischen bestimmten nicht-funktionalen Anforderungen und Architektur-Prinzipien (siehe Abschnitt 6.1), die häufig zum Einsatz kommen, um die nicht-funktionalen Anforderungen zu befriedigen. Da diese Architektur-Prinzipien in allen anderen Architekturmitteln (z. B. Taktiken und Mustern) zum Tragen kommen, liefert dieser Zusammenhang Hinweise für mögliche in Frage kommende weitere Architekturmittel. Interessant sind hier insbesondere die von „hohe Kohäsion“ und „lose Kopplung“ abgeleiteten spezielleren Architektur-Prinzipien. Vollständig und exakt eindeutig ist dieser Zusammenhang jedoch nicht. Es handelt sich vielmehr um eine Heuristik, die ein Architekt aufgrund seiner im Laufe der Zeit gemachten Erfahrungen entwickelt. Solche Heuristiken festzuhalten und in verschiedenen Zusammenhängen wiederzuverwenden ist sehr nützlich. Tabelle 8.5-1 veranschaulicht beispielhaft die Architektur-Prinzipien, die zur Erfüllung einiger nicht-funktionaler Anforderungen in Frage kommen.

Tab. 8.5-1: Zusammenhang zwischen nicht-funktionalen Anforderungen und Architektur-Prinzipien.

Nicht-funktionale Anforderung	Architektur-Prinzip
Benutzerfreundlichkeit	Explizite Schnittstellen, Schnittstellen Segregation,...
Interoperabilität	Information Hiding, Trennung von Schnittstelle und Implementierung,...
Robustheit	Lose Kopplung, hohe Kohäsion,...

Identifizierte Architektur-Mittel für die technische Architektur

Zusammenhang zwischen nicht-funktionalen Anforderungen und Architektur-Prinzipien

Nicht-funktionale Anforderung	Architektur-Prinzip
Nicht-funktionale Anforderung	Architektur-Prinzip
Sicherheit	Information Hiding, Schnittstellen Segregation,
Testbarkeit	Lose Kopplung, Hohe Kohäsion, Trennung von Schnittstelle und Implementierung....
Erweiterbarkeit	Lose Kopplung, Kapselung, Abstraktion, Separation of Concerns, Information Hiding, ...
Wiederverwendbarkeit	siehe Erweiterbarkeit
...	...

Identifizierte Bausteine

Die Identifikation der Bausteine erfolgt unter Verwendung der ausgewählten Architektur-Mittel. Im einfachsten Fall kann auf eine Referenzarchitektur zurückgegriffen werden. Diese benennt die notwendigen Bausteine. Die Tätigkeit eines Architekten kann sich in diesem Fall auf die Verifikation beschränken, indem er prüft, ob jede Schlüsselabstraktion genau einem Baustein zugeordnet ist. Das Identifizieren der Bausteine und die Definition ihrer Verantwortlichkeiten stehen in einem engen Zusammenhang. Daher werden diesen beiden Aktionen gemeinsam behandelt.

Identifizierte fachliche Bausteine

Wie geht man am Besten vor, um fachlichen Bausteine zu identifizieren? Die fachlichen Bausteine befassen sich mit den Schlüsselabstraktionen und den damit verbundenen funktionalen Anforderungen. Mögliche Kandidaten für fachliche Bausteine ergeben sich daher, indem für fachlich zusammenhängende Schlüsselabstraktionen ein fachlicher Baustein identifiziert wird. Dabei wendet man einerseits das Separation-of-Concerns-Prinzip und andererseits das Prinzip der hohen Kohäsion an. Die Kohäsion innerhalb eines fachlichen Bausteins ist hoch, da die von dem Baustein behandelten Schlüsselabstraktionen in einer engen Beziehung stehen. Des Weiteren kümmert sich ein so identifizierter Baustein um einen wesentlichen fachlichen Belang (englisch: *concern*). Abbildung 8.5-6 veranschaulicht die fachlichen Bausteine von MIS.

Fachliche Bausteine der Architektur-Vision entsprechen fachlichen Subsystemen

Im Rahmen der Architektur-Vision werden in dieser Aktion die fachlichen Subsysteme des Systems identifiziert. Die fachlichen Bausteine aus Abbildung 8.5-6 entsprechen daher den fachlichen Subsystemen von MIS. In einem weiteren Schritt wird man die fachlichen Subsysteme in feinere fachliche Bausteine aufteilen.

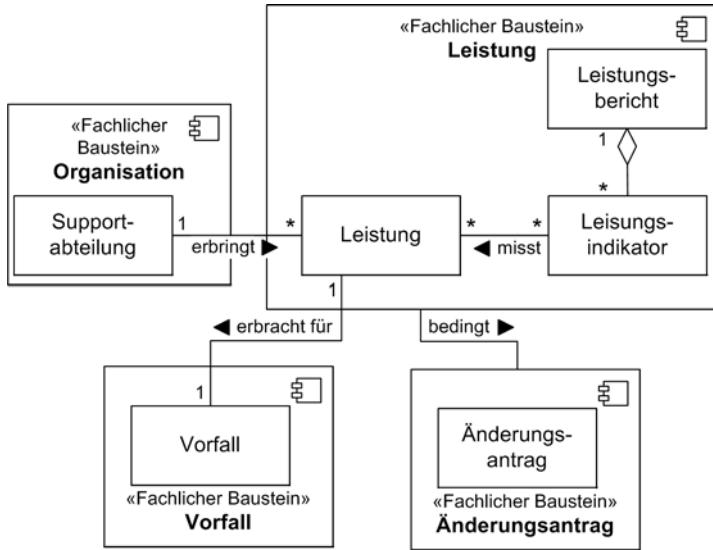


Abb. 8.5-6: Fachliche Bausteine von MIS.

Fachliche Bausteine sind für die Realisierung der funktionalen Anforderungen verantwortlich, die in engem Zusammenhang mit den durch sie behandelten Schlüsselabstraktionen stehen. Daher sollten die architekturelevanten Anforderungen nach den Schlüsselabstraktionen kategorisiert und den passenden fachlichen Bausteinen zugewiesen werden. Dadurch ergeben sich die Verantwortlichkeiten der Bausteine. Für MIS wurden diese in Tabelle 8.5-2 festgehalten.

Definiere die Verantwortlichkeiten der fachlichen Bausteine

Tab. 8.5-2: Fachliche Bausteine von MIS und ihre Verantwortlichkeiten.

Fachlicher Baustein	Schlüssel-abstraktionen	Verantwortlichkeiten
Organisation	Supportabteilung	Realisiert jegliche Funktionalität zur Verwaltung von Supportabteilungen respektive Organisationseinheiten.
Leistung	Leistung, Leistungsbericht, Leistungsindikator	Realisiert jegliche Funktionalität zur Verwaltung von Leistungen, Leistungsberichten und Leistungsindikatoren.
Vorfall	Vorfall	Realisiert jegliche Funktionalität zur Verwaltung von Vorfällen.
Änderungsantrag	Änderungsantrag	Realisiert jegliche Funktionalität zur Verwaltung von Änderungsanträgen.

Die Definition der Verantwortlichkeiten kann detaillierter erfolgen, indem jeder Schritt eines architekturrelevanten Anwendungsfalls einem fachlichen Baustein zuordnet wird. Man sollte sich noch keine Gedanken darüber machen, welche Bausteine, die durch die technische Architektur vorgegeben werden, den Schritt realisieren. Dadurch wird sicher gestellt, dass der Funktionsumfang der fachlichen Subsysteme respektive Bausteine losgelöst von ihrer technischen Realisierung definiert wird. Abbildung 8.5-7 visualisiert diese Definition der Verantwortlichkeiten am Beispiel „Leistungsindikator berechnen“.

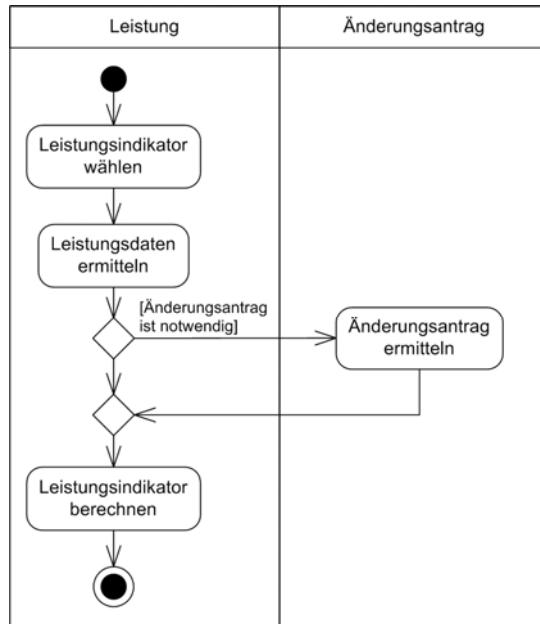


Abb. 8.5-7: Definition fachlicher Verantwortlichkeiten am Beispiel von MIS.

Identifizierte technische Bausteine und definie re deren Verantwortlichkeiten

Fachliche Bausteine allein sind nicht lebensfähig. Sie müssen sich auf technische Bausteine stützen, um auf einer Plattform existieren zu können. Daher müssen auch technische Bausteine identifiziert werden. Wie bei den fachlichen sollte man auch bei den technischen Bausteinen nach geeigneten Architekturmitteln Ausschau halten. In der Regel wird eine geeignete technische Referenzarchitektur zum Einsatz kommen. Eine, die in der Praxis oft anzutreffend ist, wird in Abbildung 8.5-8 dargestellt.

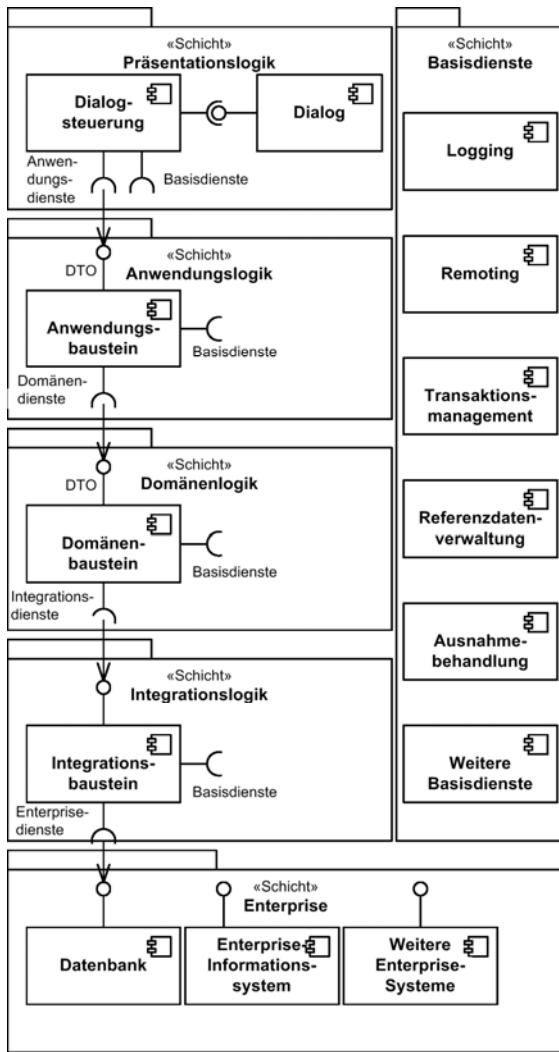


Abb. 8.5-8: Beispielhafte technische Referenzarchitektur.

Die technische Referenzarchitektur aus Abbildung 8.5-8 hat beispielhafte Charakter. Ähnliche Darstellungen finden sich in [Oestereich 2006], [Siedersleben 2006] und [Fowler 2003]. Im Folgenden werden die Bestandteile der Referenzarchitektur kurz erläutert. Tabelle 8.5-3 behandelt die Schichten und Tabelle 8.5-4 die Bausteine.

Tab. 8.5-3: Die Schichten und ihre Verantwortlichkeiten.

Schicht	Verantwortlichkeit
Basisdienste	Technische Bausteine, die Basisfunktionalität verkörpern, sind dieser Schicht zugeordnet. Sie können von Bausteinen der Präsentationslogik-, Anwendungslogik-, Domänenlogik- und Integrationslogik-Schicht verwendet werden. Sollten die Schichten über Prozess und Netzwerkgrenzen hinweg verteilt sein, müssen entsprechende Aufrufmechanismen (Stichwort: Entfernte Aufrufe (englisch: <i>remoting</i>)) vorgesehen werden.
Präsentationslogik	Die Präsentationslogik-Schicht enthält Bausteine, die zur Kommunikation mit dem Benutzer dienen.
Anwendungslogik	Die Anwendungslogik-Schicht enthält Bausteine, die Anwendungslogik realisieren. Anwendungslogik ist Logik, die eng mit der zu realisierenden Anwendung in Beziehung steht und nicht respektive nur schwer über Anwendungsgrenzen hinweg wiederverwendet werden kann. Die Darstellung eines Leistungsindikators in grafischer oder textueller Form ist beispielsweise vom konkreten Anwendungsfall abhängig und schwer wiederverwendbar.
Domänenlogik	Die Domänenlogik-Schicht enthält Bausteine, die Domänenlogik realisieren. Domänenlogik repräsentiert Fachlichkeit, die über Anwendungsgrenzen hinweg wiederverwendet werden kann. Sie ist somit anwendungsneutral und operiert allein auf fachlichen Abstraktionen. Die Berechnung eines Leistungsindikators ist beispielsweise unabhängig davon, ob der Indikator grafisch oder textuell dargestellt wird.
Integrationslogik	Die Integrationslogik-Schicht beherbergt Bausteine, welche die Anbindung von Enterprise-Systemen kapseln. Ein einfaches Beispiel sind Data-Access-Objekte, die den Datenbankzugriff abstrahieren [Fowler 2003].
Enterprise	Die Enterprise-Schicht, oftmals auch Backend-Schicht genannt, beinhaltet Systeme, mit denen das zu entwickelnde System interagieren muss. Dies können Datenbanken, Enterprise-Informationssysteme, System-Management-Systeme etc. sein.

Zur Unterscheidung zwischen Anwendungs- und Domänenlogik

Die Unterscheidung zwischen Anwendungs- und Domänenlogik ist nicht immer offensichtlich und die Grenze zwischen diesen ist manchmal fließend [Evans 2004]. Wie in Tabelle 8.5-3 bereits erwähnt, ist Domänenlogik anwendungsneutral und kann in verschiedenen Kontexten wiederverwendet werden. Die Berechnung von Telefonkosten ist beispielsweise unabhängig davon, ob die Kosten von einer Callcenter-Anwendung oder einer Selfservice-Anwendung abgefragt werden. Die Darstellung der Kosten ist hingegen anwendungsspezifisch. Anwendungslogik sollte nicht in die Domänenlogik-Schicht diffundieren. Sonst besteht die Gefahr, dass die Domänenlogik in anderen Anwendungsfällen nicht wiederverwendet werden kann und schwer testbar ist.

Tab. 8.5-4: Bausteine und Verantwortlichkeiten der technischen Referenzarchitektur.

Baustein	Verantwortlichkeit
Dialog	Ein Dialog entspricht dem View-Baustein innerhalb des Model-View-Controller-Architekturmusters [Buschmann et al. 1996].
Dialogsteuerung	Ein Dialogsteuerungsbaustein entspricht dem Controller-Baustein innerhalb des Model-View-Controller-Architekturmusters [Buschmann et al. 1996]. Er nutzt Dienste von Anwendungsbausteinen.
Anwendungsbaustein	Ein Anwendungsbaustein kapselt Anwendungslogik und stellt diese zur Verfügung. Der Datenaustausch erfolgt mittels Datentransferobjekten (DTO) [Fowler 2003].
Domänenbaustein	Ein Domänenbaustein kapselt Domänenlogik und stellt diese zur Verfügung. Der Datenaustausch erfolgt mittels Datentransferobjekten (DTO).
Integrationsbaustein	Ein Integrationsbaustein kapselt Integrationslogik (z. B. Datenbank, SAP, LDAP) und stellt diese zur Verfügung.
Basisdienstbaustein	Ein Basisdienstbaustein stellt Basisdienste zur Verfügung (z. B. Logging, Referenzdatenverwaltung).

Unabhängig davon, ob eine technische Referenzarchitektur verwendet werden konnte oder ob eine eigene technische Architektur entworfen wurde, können aus den identifizierten fachlichen Subsystemen und den durch die technische Architektur vorgegebenen Bausteintypen konkrete fachliche Bausteine für die identifizierten fachlichen Subsysteme abgeleitet werden. Die technische Architektur legt fest, wie fachliche Bausteine respektive Subsysteme realisiert werden. In dem hier verwendeten Beispiel identifiziert man für jedes fachliche Subsystem die benötigten Dialog-, Dialogsteuerungs-, Anwendungs-, Domänen- und Integrationsbausteine. Tabelle 8.5-5 verdeutlicht dies am Beispiel des fachlichen Subsystems „Leistung“. Dabei werden nur Bausteine für die Schlüsselabstraktion „Leistung“ dargestellt. Leistungsindikator und Leistungsbericht bleiben der Einfachheit halber außen vor.

Konkretisierung fachlicher Subsysteme

Tab. 8.5-5: Identifikation von Bausteinen auf Basis fachlicher und technischer Architektur am Beispiel von MIS.

Baustein	Bausteintyp	Verantwortlichkeit
Leistungsdialog	Dialog	Präsentiert leistungsbezogene Daten und nimmt Eingaben entgegen.
Leistungsdialogsteuerung	Dialogsteuerung	Steuert die Benutzerinteraktion für leistungsbezogene Aspekte.

Baustein	Bausteintyp	Verantwortlichkeit
LeistungAB	Anwendungsbaustein	Kapselt leistungsbezogene Anwendungslogik.
LeistungDB	Domänenbaustein	Kapselt leistungsbezogene Domänenlogik.
LeistungIB	Integrationsbaustein	Kapselt leistungsbezogene Integrationslogik.

Die so identifizierten fachlichen Bausteine strukturieren in ihrer Summe ein fachliches Subsystem, welches mit den Konzepten der technischen Architektur realisiert wird (siehe Abbildung 8.5-9).

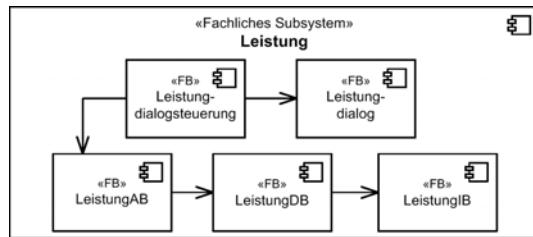


Abb. 8.5-9: Ein beispielhaftes fachliches Subsystem von MIS.

Verfeinerung der Verantwortlichkeiten

Sobald der Funktionsumfang der fachlichen Subsysteme definiert wurde, können die Verantwortlichkeiten auf einem detaillierteren Niveau betrachtet werden. Mit anderen Worten werden die Subsysteme nun als White Box behandelt und die Funktionalität auf die konkreten Bausteine aufgeteilt. In dem MIS-Beispiel kann ein funktionaler Schritt eines Anwendungsfalls in Präsentations-, Anwendungs-, Domänen- und Integrationslogik zerfallen. Darüber hinaus können auch Basisdienste, wie z. B. Sicherheit, involviert sein. Bei MIS muss sich ein Benutzer beispielsweise anmelden, bevor er Leistungsdaten erfassen oder Leistungsberichte ansehen darf. Dadurch ergeben sich die konkreten Verantwortlichkeiten der einzelnen Bausteine. Die eigentliche Berechnung eines Leistungsindikators ist z. B. in der Verantwortung des Domänenbausteins LeistungDB.

Definiere Abhängigkeiten

Nachdem die Bausteine identifiziert und ihre Verantwortlichkeiten festgehalten wurden, können nun ihre Abhängigkeiten definiert werden. Die Abhängigkeiten ergeben sich einerseits aus den fachlichen und andererseits aus den technischen Gegebenheiten.

Abhängigkeiten aus fachlichen Gegebenheiten

Die fachlichen Abhängigkeiten können aus den strukturellen Beziehungen der fachlichen Schlüsselabstraktionen abgeleitet werden. Im Fall von MIS steht Leistung beispielsweise in Beziehung mit Vorfall und

Änderungsantrag. Daraus kann eine Abhängigkeit des fachlichen Bausteins „Leistung“ von den fachlichen Bausteinen „Vorfall“ und „Änderungsantrag“ definiert werden (siehe Abbildung 8.5-10). Dadurch entsteht ein erstes Abhängigkeitsdiagramm. Es kann durchaus vorkommen, dass aufgrund technischer Gegebenheiten die fachlichen Abhängigkeiten nicht unmittelbar in der endgültigen Architektur sichtbar sind.

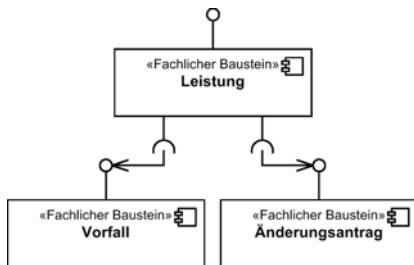


Abb. 8.5-10: Fachliche Abhängigkeiten am Beispiel Leistung.

Aus dem oder den gewählten Architekturmitteln für die Gestaltung der technischen Architektur ergeben sich die technischen Abhängigkeiten. Abbildung 8.5-11 veranschaulicht, welche Bausteine miteinander in Beziehung stehen dürfen, wenn diese technische Referenzarchitektur eingesetzt wird. So dürfen beispielsweise Anwendungsbausteine nicht direkt mit Integrationsbausteinen kommunizieren. Stattdessen interagieren Anwendungsbausteine mit Domänenbausteinen und diese wiederum mit Integrationsbausteinen. Damit soll eine möglichst hohe Plattform- und Technologieunabhängigkeit erreicht werden (siehe Abschnitt 3.4).

Abbildung 8.5-11 visualisiert die Abhängigkeiten der MIS-Bausteine am Beispiel von Leistung und Änderungsantrag, die aus den fachlichen und technischen Gegebenheiten hervorgehen. Wie hat sich die fachliche Abhängigkeit von Leistung und Änderungsantrag niedergeschlagen? Es wird deutlich, dass der Anwendungsbaustein AnwendungAB vom Domänenbaustein ÄnderungsantragDB abhängig ist und dass keine Abhängigkeit zwischen LeistungDB und ÄnderungsantragDB besteht. Folglich existiert eine Entkopplung zwischen den Domänen Leistung und Änderungsantrag auf der Stufe der Domänenbausteine. Dies ist ein Beispiel für die Anwendung des Prinzips der losen Kopplung (siehe Abschnitt 6.1.1). Diese Entkopplung hat den Vorteil, dass fachliche Domänen voneinander weitestgehend unabhängig bleiben und somit getrennt voneinander entwickelt werden können.

Abhängigkeiten aus technischen Gegebenheiten

Wie wirken sich fachliche und technische Abhängigkeiten aus?

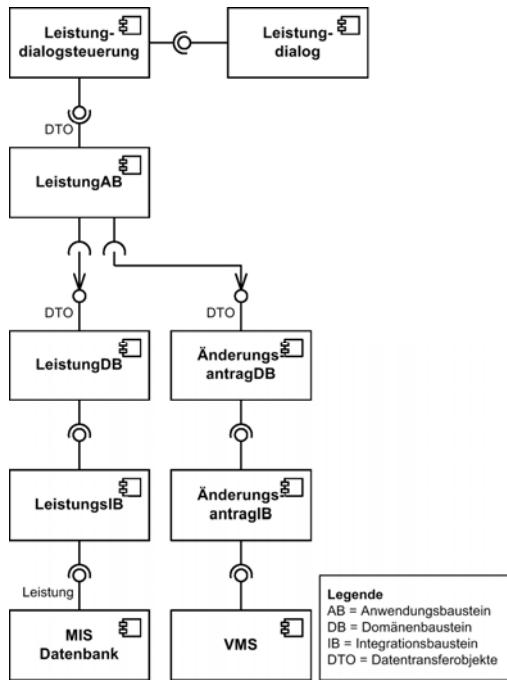


Abb. 8.5-11: Abhängigkeiten von MIS-Bausteinen.

Starke vs. weniger starke Trennung

Solch eine Trennung kann stark oder weniger stark sein. So kann die Kommunikation von Domänenbausteinen über Domänengrenzen hinweg generell verboten sein. Dies hat zur Konsequenz, dass Anwendungsbausteine alle domänenfremden Daten an Domänenbausteine übergeben müssen. Werden für die Berechnung eines Leistungsindikators z. B. Daten aus der Domäne Änderungsantrag benötigt, muss der Anwendungsbaustein LeistungAB diese Daten bei der Domäne Änderungsantrag beschaffen und diese an den Domänenbaustein LeistungDB übergeben. Bei einer starken Entkopplung müssen in dem Beispiel die Änderungsantragsdaten in eine Form konvertiert werden, die der Domänenbaustein LeistungDB erwartet. Bei einer weniger starken Trennung können einige domänenübergreifende Abhängigkeiten auf Stufe der Domänenbausteine erlaubt sein. Möchte man beispielsweise den Verwender der Leistungsdomäne von der Tatsache entkoppeln, dass zur Berechnung von Leistungsindikatoren Änderungsanträge notwendig sind, könnte der Zugriff auf die Änderungsantragsdomäne aus der Leistungsdomäne heraus erlaubt werden. Um die Kopplung möglichst gering zu halten, sollten dedizierte Schnittstellenbausteine vorgesehen werden, die als Fassaden [Gamma et al. 1995] in die Fremddomäne dienen.

Bei der Definition der Interaktionen werden die dynamischen Beziehungen zwischen den Software-Bausteinen auf einem architektonischen Niveau festgehalten. Hierzu werden die architekturelevanten Anwendungsfälle analysiert und bestimmt, welche Bausteine wie miteinander kommunizieren, um die zu erbringende Funktionalität zu realisieren. Dadurch wird deutlich, wie die Bausteine ihre Verantwortlichkeiten wahrnehmen. Diese Aktion steht in enger Verbindung mit der Definition der Schnittstellen, da die Interaktionen durch Aufrufe von Diensten und der Übergabe von Daten an Schnittstellen ausgedrückt werden. Die Aktionen werden in der Regel gleichzeitig ausgeführt. Es sollten Interaktionen für jede architekturelevante Anforderung definiert werden. Dabei wird man nicht alle Facetten einer Interaktion abbilden können. Durch die Darstellung der wesentlichen Merkmale der Interaktionen wird man jedoch in der Lage sein, die Stärken und Schwächen einer Architektur zu eruieren [Oestereich 2006]. Dies ist gerade bei der Beurteilung der Architektur von großer Bedeutung. Für die Modellierung der Interaktionen werden häufig Kommunikations- und Sequenzdiagramme der UML benutzt [Jeckle et al. 2004]. Interaktionen können sich auch über Systemgrenzen hinweg erstrecken. Als Architekt wird man auch gerade diese systemübergreifenden Interaktionen definieren und modellieren müssen. Abbildung 8.5-12 veranschaulicht eine stark vereinfachte Interaktion zwischen MIS-Bausteinen zur Erfassung von Leistungsdaten.

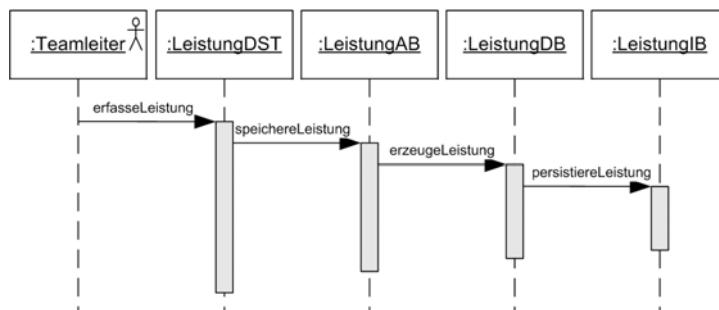


Abb. 8.5-12: Vereinfachte Interaktion zwischen MIS-Bausteinen.

Normalerweise können typische architekturelevante Interaktionen abstrahiert werden, sodass sie die gängigen Kollaborationen zwischen Bausteinen auf einer allgemeingültigen Ebene visualisieren. Beispiele hierfür sind die klassischen Create-Read-Update-Delete(CRUD)-Operationen. Diese werden von einer Architektur in der Regel identisch realisiert, unabhängig davon, um welche konkrete Abstraktion es sich handelt. Das Erfassen von Daten über eine Benutzerschnittstelle erfolgt beispielsweise gleich, egal ob nun Leistungsdaten oder Supportabtei-

Definiere Schnittstellen

lungen erfasst werden sollen. Solche Interaktionen werden ggf. bereits durch die eingesetzte Referenzarchitektur dokumentiert.

Schnittstellen sollten ihren Zweck und nicht ihre Implementierung offenbaren

Bausteine kommunizieren miteinander über Schnittstellen (siehe Abschnitt 3.4). Bausteine bieten ihre Funktionalität über Schnittstellen an und nutzen Funktionalität anderer Bausteine über deren Schnittstellen. Daher müssen innerhalb dieser Aktion sowohl die angebotenen als auch die benötigten Schnittstellen definiert werden. Man steht dabei vor der Herausforderung, einerseits die benötigte und angebotene Funktionalität mittels Schnittstellen auszudrücken und andererseits die Schnittstellen so zu definieren, dass die Abhängigkeiten zwischen den Bausteinen möglichst gering sind. Bei der Definition der Abhängigkeiten wurde bereits festgelegt, welche Bausteine prinzipiell voneinander abhängig sind. Die Definition der Schnittstelle bestimmt hierbei den Grad der Abhängigkeit. Je nachdem wie die Schnittstellen entworfen werden, können Bausteine stark oder weniger stark voneinander abhängen. Aus diesem Grund werden in dieser Aktion einige Aspekte guten Schnittstellenentwurfs behandelt. Im Folgenden wird der Begriff Dienst für ein Stück Funktionalität verwendet, welches über eine Schnittstelle von einem Baustein angeboten respektive benötigt wird.

Dienste sollten in Funktionen und Kommandos unterteilt werden

Schnittstellen sind Abstraktionen konkreter Bausteinimplementierungen (siehe Abschnitt 6.1.6). Daher haben implementierungsspezifische Aspekte in einer Schnittstelle nichts verloren. Dies beginnt bereits bei der Benennung von durch Schnittstellen angebotenen Diensten. Dienste sollten ihren Zweck klar zum Ausdruck bringen, indem dieser bereits in deren Namen reflektiert wird. In diesem Zusammenhang spricht man auch von zweckoffenbarenden Schnittstellen (englisch: *intention-revealing interfaces*) [Evans 2004]. Der Name eines Dienstes sollte also klarmachen, was mit ihm erreicht werden kann und nicht, wie er etwas erreicht.

Es gibt zwei Arten von Diensten. Dienste, die den Zustand eines Bausteins nicht ändern, und Dienste, die den Zustand eines Bausteins ändern [Evans 2004 und Siedersleben 2006]. Erstere bezeichnet man als Funktionen oder Abfragen (englisch: *queries*) und letztere als Kommandos. Funktionen haben im Gegensatz zu Kommandos keine Seiteneffekte zur Folge. Sie sind einfacher zu testen und weniger risikoreich als Kommandos. Sie werden auch als idempotente Funktionen bezeichnet. Das Lesen von Leistungsdaten ist ein Beispiel für eine Funktion. Kommandos versetzen Bausteine hingegen in einen anderen Zustand und erfordern daher ein umfangreicheres Testen. Das Anlegen von Leistungsdaten ist ein Beispiel für ein Kommando. Bei der Definition der

Schnittstelle sollte eine klare Segregation (siehe Abschnitt 6.1.6) zwischen Funktionen und Kommandos vorgenommen werden, indem Dienste entweder Funktionen oder Kommandos repräsentieren.

Um den Charakter eines Dienstes besser ausdrücken zu können, sollten dessen Vor- und Nachbedingungen dokumentiert werden. Vorbedingungen formulieren dabei Bedingungen, die erfüllt sein müssen, bevor der Dienst aufgerufen werden kann. Nachbedingungen sind die Bedingungen, die durch den Dienst garantiert werden, wenn die Vorbedingungen erfüllt sind. Dies entspricht dem Design-by-Contract-Prinzip (siehe Abschnitt 6.1.6).

Dienste sollten ihre Vor- und Nachbedingungen dokumentieren

Ein Dienst erwartet Eingabeparameter und gibt Ausgabeparameter zurück. Damit hängen Verwender des Dienstes von den Datentypen ab, die von ihm deklariert werden. Dadurch kann eine enge Kopplung (siehe Abschnitt 6.1.1) zwischen Bausteinen entstehen, wenn die Datentypen zu viele Details offenbaren. Im Sinne einer losen Kopplung sollte daher das Information-Hiding-Prinzip (siehe Abschnitt 6.1.5) beim Entwurf einer Schnittstelle berücksichtigt werden. Je nach der Art der Schnittstelle können unterschiedliche Klassen von Datentypen verwendet werden (siehe Tabelle 8.5-6). Generell kann man sagen, dass die Kopplung innerhalb eines Subsystems hoch (interne Kopplung) und zwischen Subsystemen gering sein sollte (externe Kopplung). Daher können innerhalb eines Subsystems Standarddatentypen, Entitätsobjekte [Evans 2004] und Wertobjekte (englisch: *value objects*) [Fowler 2003] genutzt werden. Für die Kommunikation zwischen Subsystemen sollten hingegen Datentransferobjekte statt Entitätsobjekte verwendet werden. Entitätsobjekte besitzen eine eigene Identität innerhalb einer Domäne. Ein Vorfall ist beispielsweise ein Entitätsobjekt im MIS-Fall. Dahingegen werden Wertobjekte ausschließlich über ihre Attribute definiert. Sie besitzen keine eigene Identität und sind unveränderlich (englisch: *immutable*). Ein Geldbetrag kann z. B. durch ein Wertobjekt vom Typ Geld repräsentiert werden. Im MIS-Fall könnten die monetären Kosten eines Vorfalls mittel Geld-Wertobjekten dargestellt werden. Die Kosten gehören zu einem Vorfall und besitzen keine eigene Identität. Die Unterscheidung zwischen Entitäts- und Wertobjekten ist domänenabhängig. So kann z. B. die Abstraktion Adresse in einer Domäne ein Entitätsobjekt und in einer anderen ein Wertobjekt sein. Datentransferobjekte entspringen zum einen der Idee, die Anzahl der notwendigen Client-/Server-Interaktionen (siehe Abschnitt 6.4) zu minimieren und zum anderen Klienten von der konkreten Repräsentation von Entitätsobjekten zu entkoppeln. Im Fall von MIS könnte ein LeistungDTO beispielsweise die Entität Leistung verkörpern.

Dienste sollten keine Details offenbaren

Tab. 8.5-6: Verwendung von Datentypen in Schnittstellen.

Klassen von Datentypen	Beispiele	Intern	Extern
Standarddatentypen	Integer, Double, String, etc.	X	X
Datentransferobjekte	LeistungDTO, AenderungsantragDTO, KundenDTO, VertragDTO, etc.		X
Entitätsobjekte	Leistung, Kunde, Vertrag, Adresse	X	
Wertobjekte	Geld, Farbe, Zeitraum	X	X

Datentransferobjekte innerhalb von Subsystemen?

Manchmal können sogar Datentransferobjekte zwischen Bausteinen eines Subsystems zum Einsatz kommen. Dies ist z. B. dann der Fall, wenn Daten über Netzwerk- oder Prozessgrenzen hinweg transportiert werden sollen. Ebenso kann dieser Ansatz gewählt werden, um Baustein einer höheren Schicht von Bausteindetails einer niedrigeren Schicht zu entkoppeln. Beispielsweise könnten im MIS-Fall Domänenbausteine Daten mit Anwendungsbausteinen über Datentransferobjekte, statt über Entitätsobjekte austauschen.

Schnittstellen für Entitäten?

Häufig wird die Frage diskutiert, ob für Entitätsobjekte Schnittstellen definiert werden sollen oder nicht. Diese Frage lässt sich pauschal nicht beantworten. Sollte es sich bei den Entitätsklassen respektive -objekten um echte Objekte im Sinne der Objektorientierung (siehe Abschnitt 6.2.2) handeln, ist die Verwendung von Schnittstellen angebracht. Ist die eigentliche Geschäftslogik jedoch primär in Service-Anbietern gekapselt, hat die Verwendung von Schnittstellen geringere Vorteile, da sich die Operationen dann auf das Setzen und Lesen von Attributen beschränken (sogenannte Setter- und Getter-Operationen).

Assoziationen zwischen Entitäten

Assoziationen zwischen Entitäten haben einen wesentlichen Einfluss auf die Kopplung von Bausteinen. Im MIS-Beispiel erbringt eine Supportabteilung eine oder mehrere Leistungen (siehe Abbildung 8.5-5). Es stellt sich nun die Frage, wie diese Beziehung abgebildet werden soll. Die Verwendung von echten Objektassoziationen führt zu einer engen Kopplung zwischen den Subsystemen, da Datentypen der Subsysteme über die Schnittstellen kommuniziert werden. Dahingegen entkoppelt der Einsatz von logischen Schlüsseln zur Referenzierung von Entitäten über Bausteingrenzen hinweg die Bausteine voneinander. Ein logischer Schlüssel identifiziert eine Entität eindeutig, ohne diese direkt über eine Objektassoziation zu referenzieren. Hierzu erhält z. B. jede Entität eine eindeutige Nummer oder einen eindeutigen Zeitstempel. Für subsystem- und systemgrenzenübergreifende Schnittstellen sollten logische Schlüs-

sel eingesetzt werden. Bei internen Schnittstellen sind Objektreferenzen zu bevorzugen.

Bei der Definition von Schnittstellen sollten die gegebenen Hinweise berücksichtigt werden. Es stellt sich jedoch die Frage, wie man überhaupt zu den Schnittstellen gelangt. Schnittstellen sind getrieben von den konkreten Anforderungen. Es geht also nicht darum, willkürliche Dienste in den Schnittstellen zu publizieren. Vielmehr sind die funktionalen Anforderungen genau zu analysieren und für diese zweckmäßige Dienste zu definieren. Sonst besteht die Gefahr, dass zu viele Dienste bzw. die falschen Dienste definiert werden. Aufgrund dessen sollte man, ausgehend von den architekturelevanten Anwendungsfällen, die Interaktion durchspielen und sich dabei stets die Frage stellen, welche Dienste von dem betrachteten Baustein in dieser Interaktion konkret benötigt werden. Im Laufe der Iterationen werden Dienste überarbeitet, indem diese aufgeteilt bzw. zusammengeführt werden. Dabei sollte man sich stets von den gegebenen Hinweisen zum zweckmäßigen Schnittstellenentwurf leiten lassen.

Zur Spezifikation von Schnittstellen können verschiedene Mittel eingesetzt werden. Tabelle 8.5-7 stellt einige dieser Mittel vor. In vielen Fällen wird man eine Kombination der aufgeführten Mittel verwenden.

Tab. 8.5-7: Mittel zur Spezifikation von Schnittstellen.

Mittel	Beschreibung
UML mit OCL	UML bietet gerade in Verbindung mit OCL geeignete Möglichkeiten zur Spezifikation von Schnittstellen (siehe Abschnitt 6.6.2).
IDL	Auch implementierungsneutrale Schnittstellenbeschreibungssprachen, wie man sie von CORBA und COM her kennt, können zur Spezifikation von Schnittstellen eingesetzt werden.
WSDL	Eine Schnittstelle kann auch allein durch die Summe der ausgetauschten Nachrichten spezifiziert werden. Dies ist insbesondere im SOA-Umfeld der Fall. Dabei ist häufig WSDL anzutreffen.
DSL	Domänenpezifische Sprachen können ebenfalls für die Spezifikation von Schnittstellen verwendet werden. In diesem Fall erstellt man eine solche Sprache, die eng an der Domäne angelehnt ist, um die Schnittstelle zu spezifizieren.

An das Definieren der Schnittstellen schließt sich das Verteilen der Bausteine auf Prozesse respektive leichtgewichtige Prozesse (englisch: *threads*) und die Plattform an. Es wird somit einerseits die Prozess- und

Wie kommt man zu Schnittstellen?

Welche Mittel setzt man zur Spezifikation von Schnittstellen ein?

Verteilte Bausteine

andererseits die Verteilungssicht der Architektur erarbeitet bzw. vervollständigt.

Verteilen der Bausteine auf die Plattform

Eine Plattform besteht sowohl aus Hardware- als auch aus Software-Bausteinen (siehe Abschnitt 3.4). In diesem Zusammenhang spricht man auch von den Knoten eines Systems. Beim Verteilen der Bausteine auf die Plattform werden die Software-Bausteine auf die Knoten verteilt. Knoten respektive Plattformbausteine können dabei sowohl Hardware- als auch Software-Bausteine sein. So kann eine JEE-Ausführungsumgebung beispielsweise auf einem Hardware-Server installiert sein. Die Software-Bausteine werden dann in der Ausführungsumgebung auf dem Hardware-Server ausgeführt. Die Verteilungssicht zeigt ebenfalls die Verbindungen zwischen den Knoten. Sie geht damit über das reine Verteilen der Software-Bausteine auf Knoten hinaus. Weitere relevante Punkte sind beispielsweise die Dimensionierung der Hardware-Server (Prozessor, Arbeitsspeicher, Plattenplatz etc.) und des Netzwerks sowie die genaue Spezifikation der eingesetzten Software (Betriebssystem, Anwendungsserver, Datenbank, Treiber etc.). Die Gesamtheit dieser Punkte wird häufig als operationale Architektur bezeichnet. Sie umfasst mehr als nur Software-Aspekte. Deren Entwurf bedingt Kenntnisse anderer Disziplinen, wie z. B. der Netzwerk- und der System-Management-Architektur (siehe Abschnitt 3.2). Der Fokus liegt in diesem Abschnitt auf der Verteilung der Bausteine. Abbildung 8.5-13 visualisiert die Verteilungssicht von MIS. Auf das IAM-System wurde aus Platzgründen verzichtet.

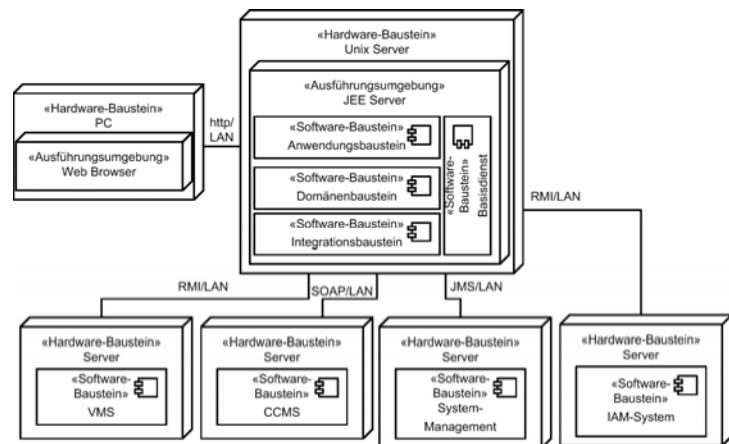


Abb. 8.5-13: Verteilungssicht von MIS.

Nebenläufigkeit ist ein wichtiges Thema beim Entwurf der Architektur heutiger Software-Systeme. Komponentenplattformen bieten zwar bereits entsprechende Dienste (z. B. entfernte Zugriffe, Lifecycle-Management, Clustering), um die Handhabung von Nebenläufigkeit zu vereinfachen, allerdings kann man sich nicht allein darauf verlassen. Diese Dienste schließen nämlich typische Probleme, wie Race Conditions, Deadlocks oder Dateninkonsistenzen nicht aus. Vielmehr muss man sich aktiv mit dieser Thematik auseinander setzen. Aus diesem Grund sollten in dieser Aktion Nebenläufigkeitsmodelle und Zustandsmodelle erstellt werden. Nebenläufigkeitsmodelle veranschaulichen die Verteilung von Software-Bausteinen auf Prozesse respektive leichtgewichtige Prozesse. [Schmidt et al. 2000] beinhaltet nützliche Architektur-Muster zum Thema Nebenläufigkeit. Zustandsmodelle artikulieren die möglichen Zustände, in denen sich ein System befinden kann, die erlaubten Zustandsübergänge sowie Ereignisse und Aktionen. Diese Informationen explizit zu machen, erleichtert die Vermeidung von möglichen Nebenläufigkeitsproblemen. Näheres zu diesem Thema findet sich in [Rozanski und Woods 2005].

Eine Architektur darf nicht unreflektiert eingesetzt werden. Vielmehr ist diese auf ihre Zweckmäßigkeit hin zu analysieren. Ferner gibt es immer mehr als eine Architektur-Alternative. Ein Architekt wird daher oft vor der Herausforderung stehen, aus den vorhandenen Alternativen die zweckmäßigste zu wählen. Darüber hinaus wird er auch gebeten werden, existierende Architekturen zu bewerten. Die Beurteilung einer Architektur ist sowohl während der Erarbeitung einer initialen Architektur relevant als auch während der nachfolgenden Entwicklung eines Systems, um so zu überprüfen, ob eine Architektur noch mit den anfänglich gestellten oder mittlerweile geänderten Anforderungen übereinstimmt. Für die Beurteilung von Architekturen stehen verschiedene Methoden zur Verfügung [Rozanski und Woods 2005]. Einige werden im Folgenden kurz vorgestellt.

Bei der Präsentationsform werden Aspekte einer Architektur Interessenvertretern informell vorgestellt. Interessenvertreter können unmittelbar Feedback zu den vorgestellten Aspekten geben.

Formale Reviews werden von einer Gruppe von Interessenvertretern durchgeführt. Diese analysiert die Architektur-Dokumentation systematisch, kommentiert formal und beschließt ggf. Aktionen zur Behebung von Mängeln.

Verteilen der Bausteine auf Prozesse

Beurteile Architektur

Präsentationen

Formale Reviews

Walkthroughs

Walkthroughs basieren auf architekturelevanten Szenarien. Die Szenarien werden durchgespielt, um festzustellen, ob und wie eine Architektur diese erfüllt. An Walkthroughs nehmen betroffene Interessenvertreter teil. Im MIS-Fall könnte beispielsweise der Anwendungsfall „Leistungsdaten erfassen“ durchgespielt werden.

Simulationen

Simulationen werden erstellt, um für bestimmte Fragestellungen, wie z. B. das Verhalten des Systems unter Last, Antworten zu erhalten, ohne das System selbst implementieren zu müssen.

Architektur-Prototypen bei der Architektur- Beurteilung

Architektur-Prototypen sind Implementierungen zur Beurteilung von Architektur-Alternativen oder Aspekten einer Architektur. Mithilfe von Prototypen können typische Abläufe einer Architektur realisiert werden. Eine Architektur wird dabei anhand von rudimentären Bausteinen implementiert, sodass die wichtigsten Abläufe im begrenzten Maße ausprobiert werden können. Wenn dabei sämtliche relevanten Systembausteine (z. B. für Benutzeroberfläche bis hin zur Persistenz) zum erfolgreichen Zusammenspiel gebracht werden, wird oft vom sogenannten architektonischen „Durchstich“ gesprochen. Ein solcher „Durchstich“ sollte zum Zwecke der Risikominimierung so früh wie möglich erfolgen.

Szenariobasierte Methoden

Die Software Architecture Analysis Method (SAAM) [Kazman et al. 1994] und die daraus weiterentwickelte Architecture Tradeoff Analysis Method (ATAM) [Kazman et al. 1998] sind Vertreter von szenariobasierten Methoden. Sie adressieren die Problematik des Interpretationsspielraums nicht-funktionaler Anforderungen. Hierzu erarbeiten sie konkrete Szenarien, welche eine nicht-funktionale Anforderung weiter konkretisieren. Die Erarbeitung der einzelnen Szenarien erfolgt in einer Gruppenarbeit, bei der alle Interessenvertreter – Benutzer, Auftraggeber, Architekt etc. – vertreten sind. Auf diese Weise entsteht bei allen Interessenvertretern ein besseres Verständnis für die einzelnen Anforderungen. Ein Beispilszenario für die Anforderung der Erweiterbarkeit könnte zum Beispiel lauten: „Das System soll mit einem Aufwand von maximal einem Personenjahr in ein unternehmensweites Portal integriert werden.“ Basierend auf den so erarbeiteten Szenarien erfolgen eine Bewertung einer Architektur und eine Beurteilung der Auswirkungen der Szenarios auf eine Architektur. So kann die Beurteilung anhand des Beispilszenarios hervorbringen: „Die Regeln zur Programmierung der Seitenabfolge sind noch nicht vollständig definiert, sodass diese Logik über die gesamte Präsentationsschicht verteilt ist. Im Falle einer Umstellung bei der Benutzerführung sind daher alle Programmteile der Präsentationsschicht anzupassen.“ Anhand dieser Beispilszenarien ist

eine detaillierte qualitative Beurteilung einer Architektur möglich. Allerdings basiert diese Beurteilungs-Methode auf der Architekturdokumentation (siehe Abschnitt 8.7). Sie bedingt eine sehr detaillierte Kenntnis einer zu bewertenden Architektur und der durchzuführenden Änderungen, um alle Auswirkungen eines solchen Szenarios einschätzen zu können.

Bei der Beurteilung einer Architektur können Checklisten sehr nützlich sein. Eine Checkliste besteht aus einer Liste detaillierter Fragen, welche die Umsetzung der unterschiedlichen Anforderungen widerspiegelt. Checklisten können für verschiedene Beurteilungsmethoden eingesetzt werden.

Am Beispiel der Anforderung „Erweiterbarkeit“ kann die Frage lauten: „Erlaubt es die Architektur, dass man nachträglich eine zusätzliche Web-Benutzeroberfläche zur Verfügung stellen kann?“. Anhand dieser Fragen kann festgestellt werden, wie gut eine Architektur die gegebenen Anforderungen unterstützt. Bei dieser Methode ergibt sich nur die Schwierigkeit, dass man zum Aufstellen der Checkliste bereits ein sehr genaues Verständnis der Anforderungen haben muss. In dem oben genannten Beispiel muss bereits bekannt sein, dass Erweiterbarkeit auf das zusätzliche Anbinden einer Web-Benutzeroberfläche bezieht und eine zusätzliche Benutzeroberfläche für mobile Geräte nicht benötigt wird.

In Tabelle 8.5-8 ist festgehalten, welche Artefakte für welche Architektursichten (siehe Abschnitt 4.2) im Rahmen der Tätigkeit Entwerfen der Architektur für MIS erarbeitet wurden.

Tab. 8.5-8: Bearbeitete Architektur-Sichten beim „Entwerfen der Architektur“.

Architektur-Sicht	MIS-Artefakte
Logische Sicht	<ul style="list-style-type: none">> Erweiterter Systemkontext (siehe Abbildung 8.5-4)> Schlüsselabstraktionen (siehe Abbildung 8.5-5)> Fachliche Bausteine (siehe Abbildung 8.5-6)> Fachliche Bausteine und ihre Verantwortlichkeiten (siehe Tabelle 8.5-2)> Definition fachlicher Verantwortlichkeiten (siehe Abbildung 8.5-7)> Identifikation von Bausteinen auf Basis fachlicher und technischer Architektur (siehe Tabelle 8.5-5)

Checklisten

Beispiel für eine Checklisten-Frage

MIS:
Bearbeitete
Architektur-Sichten

Logische Sicht (Forts.)	<ul style="list-style-type: none"> > Ein beispielhaftes, fachliches Subsystem (siehe Abbildung 8.5-9) > Fachliche Abhangigkeiten am Beispiel Leistung (siehe Abbildung 8.5-10) > Abhangigkeiten von MIS-Bausteinen (siehe Abbildung 8.5-11) > Vereinfachte Interaktion zwischen MIS-Bausteinen (siehe Abbildung 8.5-12)
Datensicht	<ul style="list-style-type: none"> > Verwendung von Datentypen in Schnittstellen (siehe Tabelle 8.5-6)
Verteilungs-sicht	<ul style="list-style-type: none"> > Verteilung der Software-Bausteine (siehe Abbildung 8.5-13)

Checkliste:
Entwerfen der Architektur

- Werden alle Umsysteme (auch mittelbare wie z. B. System-Management) im Systemkontext aufgefuhrt?
- Werden alle architekturelevanten Anwendungsfalle durch Bausteine abgedeckt?
- Ist jeder Baustein in mindestens einen Anwendungsfall involviert?
- Wurde jeder Baustein hinsichtlich seiner Verantwortlichkeit, Schnittstelle und Interaktionen mit anderen Bausteinen dokumentiert?
- Werden alle nicht-funktionale Anforderungen jeweils durch entsprechende Mittel abgedeckt?
- Wurde jede Schlusselabstraktion einem fachlichen Baustein zugewiesen?
- Wurden fachliche Bausteine gema der eingesetzten Referenzarchitektur entworfen?
- Wurden Bausteine fur Querschnittsaufgaben (z. B. Logging) vorgesehen?
- Besitzt jeder Baustein eine klare Verantwortlichkeit und keine Uberlappungen mit anderen Bausteinen?
- Wurde jeder Baustein einer passenden logischen Schicht zugeordnet?
- Existieren Schnittstellen zwischen dem System und seinen Umsystemen?
- Existieren Schnittstellen zwischen den Subsystemen?
- Erfolgen Interaktionen zwischen Bausteinen ausschlielich uber Schnittstellen?
- Finden keine unnotigen Interaktionen zwischen Bausteinen statt?
- Existieren keine zirkularen Abhangigkeiten zwischen Bausteinen?
- Wurde jeder Baustein auf einen Knoten verteilt?
- Wird der Belang Nebenlauigkeit ausreichend bercksichtigt?
- Wurde die Architektur einer Beurteilung unterzogen?
- Wurden die Ergebnisse der Beurteilung in die Architektur eingearbeitet?
- Wurde der Reifegrad der eingesetzten Technologien ausreichend beachtet und bewertet?
- Wurden organisatorische Rahmenbedingungen bercksichtigt?

Wurden alle Architektur-Prinzipien aufgrund der Anforderungen ausgewählt?
Werden alle, aufgrund der gegebenen Anforderungen relevanten Architektur-Prinzipien beachtet?
Werden, wo es möglich ist, konkretere Architektur-Mittel als Architektur-Prinzipien auf die architektonischen Probleme angewendet?
Werden alle ausgewählten Architektur-Prinzipien, die für die Architektur eine Rolle spielen, beachtet?
Werden potentielle Konflikte zwischen den Prinzipien behandelt?
Ist die Verwendung von Architektur-Prinzipien, die – z. B. durch die Nutzung anderer Mittel – zum Einsatz kommen, ausreichend dokumentiert?
Werden alle Prinzipien, die zum Einsatz kommen, konsequent umgesetzt?
Wenn Prinzipien verletzt werden oder wenn es zu Konflikten zwischen Prinzipien kommt, sind die Gründe dafür dokumentiert?

**Checkliste:
Architektur-Prinzipien**

Werden alle grundlegenden architektonischen Konzepte eingesetzt, die aufgrund der Anforderungen ausgewählt wurden?
Werden alle grundlegenden architektonischen Konzepte eingesetzt, die z. B. durch äußere Einflüsse, das Management oder verwendete Technologien erforderlich sind?
Kann man Strukturbrüche durch die Verwendung bestimmter grundlegender architektonischer Konzepte vermeiden?
Kann man durch zusätzliche grundlegende architektonische Konzepte näher an der Domäne oder der Problemstellung arbeiten?
Werden ausreichende Ressourcen für architekturverbessernde Maßnahmen im Rahmen des Reengineering und der Wartung bereitgestellt?
Finden architekturverbessernde Maßnahmen in ausreichendem Maße im Reengineering und der Wartung statt?

**Checkliste:
Grundlegende architektonische Konzepte**

Werden eine oder mehrere Mustersprachen in Hinblick auf die Anforderungen eingesetzt?
Werden einzelne Muster oder Stile in Hinblick auf die Anforderungen eingesetzt?
Werden Architektur-Taktiken dort eingesetzt, wo sie auf zentrale Qualitätsattribute in den Anforderungen passen?
Werden architektonische Taktiken, Muster oder Stile eingesetzt, deren Einsatz sich notwendigerweise aus der Anwendung anderer Taktiken, Muster oder Stile ergibt?
Werden diejenigen grundlegenden architektonischen Konzepte oder Technologien verwendet, die als ratsam für die Umsetzung der eingesetzten architektonischen Taktiken, Mustern oder Stilen empfohlen werden?
Werden diejenigen grundlegenden architektonischen Konzepte eingesetzt, die in allen Beispielen und bekannten Nutzungen der eingesetzten architektonischen Taktiken, Mustern oder Stilen genutzt werden?

**Checkliste:
Architektonische Taktiken, Stile und Muster**

Checkliste: Basisarchitekturen

Wurden die Basisarchitekturen aufgrund der gegebenen Anforderungen ausgewählt?
Werden die gewählten Basisarchitekturen auch durch Taktiken, Stile, Muster oder Referenzarchitekturen unterstützt?
Werden die Basisarchitekturen eingesetzt, die notwendigerweise oder üblicherweise zusammen mit der Verwendung anderer Basisarchitekturen zum Einsatz kommen?
Ist dokumentiert, welche alternativen Basisarchitekturen es zu einer gewählten Basisarchitektur gibt und was die Vorteile und Nachteile der gewählten Basisarchitekturen sind?
Werden bestehende Mittel genutzt, die eine gewählte Basisarchitektur unterstützen?

Checkliste: Referenzarchitekturen

Werden existierende industrieübergreifende, industriebezogene oder plattformbezogene Referenzarchitekturen für die Domäne der Architektur genutzt?
Gibt es erfolgreiche Beispiele für den Einsatz der gewählten Referenzarchitektur, an denen sich die Architektur orientiert?
Wenn eine Referenzarchitektur eingesetzt wird, gibt es eine Referenzimplementierung zu der ausgewählten Referenzarchitektur und ist diese zugreifbar (z. B. als Open Source)?
Basiert eine eingesetzte Referenzarchitektur auf bewährten Prinzipien, Mustern, Stilen und Taktiken und sind diese dokumentiert?
Ist eine eingesetzte Referenzarchitektur ausreichend anpassbar und wird sie in auf die Architektur-Domäne angepasster Weise eingesetzt?
Bringt eine eingesetzte Referenzarchitektur – verglichen mit einer einfachen proprietären Lösung – einen Nutzen und ist der Zusatzaufwand für die eingesetzte Referenzarchitektur rechtfertigbar?
Sind die Anpassung der und die Entscheidung für die Referenzarchitektur ausreichend dokumentiert?

Checkliste: Architektur- Modellierungsmittel

Werden Standardmodellierungsmittel, wie UML, eingesetzt oder eher proprietäre Modellierungsmittel? Wenn proprietäre Modellierungsmittel eingesetzt werden, wird dies in der Dokumentation begründet?
Werden, wo sinnvoll, domänenspezifische Sprachen eingesetzt?
Wenn domänenspezifische Sprachen eingesetzt werden, sind geeignete Basiskonzepte (dynamische Sprachen, Skriptsprachen, MDSD) bzw. Technologien zu deren Implementierung im Einsatz?
Wenn ein Modellierungsmittel nicht nur zur Dokumentation, sondern auch zur Generierung bzw. modellgetriebenen Entwicklung eingesetzt wird, sind die Modelle ausreichend formal?
Werden, wo sinnvoll, auch domänenspezifische Sprachen für die Architektur-Beschreibung, also eine ADL, eingesetzt?

Checkliste: Architekturrelevante Technologien

Werden Technologien, die sich direkt durch den Einsatz anderer Mittel ableiten lassen, da sie diese direkt unterstützen, auch genutzt?
Werden konsequent Technologien eingesetzt, die durch äußere Einflüsse, wie Unternehmensstrategien, vorgegeben werden?

Werden in der Architektur auch alternative Technologien – dort wo es sinnvoll ist – genutzt?

Sind alle technologische Alternativen und Abwägungen mit Vor- und Nachteilen dokumentiert?

Werden Einflüsse einer Technologie auf architektonische Eigenschaften, wie Qualitätsattribute, in der Dokumentation erläutert?

Wird dokumentiert, inwieweit eine Technologieauswahl die Auswahl möglicher anderer architektonischer Mittel einschränkt?

Werden Mittel, die man mit der Technologie automatisch auswählt, eingesetzt? Sind diese Konsequenzen einer Technologieauswahl auch dokumentiert?

Zusammenfassung

- > Der Entwurf einer Architektur wird von den architekturelevanten Anforderungen, fachlichen Modellen, dem Systemkontext und den einzusetzenden Architektur-Mitteln beeinflusst.
- > Die Architektur der ersten Iteration bezeichnet man auch als Architektur-Vision.
- > Beim Definieren des Systemkontexts wird der in der Systemvision enthaltene Systemkontext weiter konkretisiert und die Schnittstellen werden dokumentiert. Ein Systemkontext ist mehr als nur ein Diagramm.
- > Die Schlüsselabstraktionen der Fachdomäne müssen für den Architektur-Entwurf identifiziert werden.
- > Anforderungen mit einer hohen Priorität sollten möglichst früh beim Entwurf berücksichtigt werden.
- > Ein Architektur-Mittel allein löst noch nicht alle Probleme. Vielmehr muss der Architekt das Mittel adaptieren und ggf. mit eigenen Entwurfsideen kombinieren.
- > Man sollte bei der Wahl der Architektur-Mittel zunächst Referenzarchitekturen und Basisarchitekturen in Betracht ziehen. Falls sich keine passende findet, sollten Taktiken, Stile und Muster sowie Konzepte und Prinzipien betrachtet werden.
- > Der Fokus verschiebt sich beim Architektur-Entwurf von der Fachlichkeit zur Technik.
- > Für jeden Baustein müssen dessen Verantwortlichkeiten und Abhängigkeiten definiert werden.
- > Fachliche Bausteine werden mittels Schlüsselabstraktionen und den zu den Abstraktionen gehörenden funktionalen Anforderungen identifiziert.
- > Abhängigkeiten können aus fachlichen und technischen Gegebenheiten abgeleitet werden.
- > Die Interaktionen zwischen den Bausteinen müssen dokumentiert werden.

- > Schnittstellen sollten ihren Zweck und nicht ihre Implementierung offenbaren.
- > Das Verteilen der Bausteine umfasst deren Verteilung auf Knoten und auf Prozesse.
- > Zur Beurteilung einer Architektur stehen informelle und formale Methoden zur Verfügung (z. B. Präsentationen, Walkthroughs, Simulationen, Architektur-Prototypen, szenariobasierte Methoden).

8.6 Umsetzen der Architektur

Grundlegende Konzepte der Tätigkeit „Umsetzen der Architektur“

Abbildung 8.6-1 stellt die grundlegenden Konzepte der Tätigkeit „Umsetzen der Architektur“, welche in diesem Abschnitt behandelt und detailliert werden, vor und visualisiert ihren Zusammenhang.



Abb. 8.6-1: Grundlegende Konzepte der Tätigkeit „Umsetzen der Architektur“.

Architekt ist verantwortlich für die Umsetzbarkeit

Nachdem Entwerfen einer Architektur ist die Arbeit für einen Architekten noch nicht erledigt. Vielmehr ist er während der Implementierung des Systems gefordert, da er gewährleisten muss, dass das System auf Basis seiner Architektur umgesetzt werden kann. Die vom Architekten durchgeführten Aktionen werden in Abbildung 8.6-2 dargestellt. Im Rahmen dieser Tätigkeit wird primär die Umsetzungssicht eines Systems (siehe Abschnitt 4.2) erarbeitet. Insbesondere diese Tätigkeit führt man nicht allein durch, sondern arbeitet mit Experten aus seinem Team (z. B. Entwicklern, Konfigurationsmanagern, Testmanagern) zusammen und delegiert Aufgaben. Jedoch trägt man die architektonische Verantwortung für das Umsetzen der Architektur.

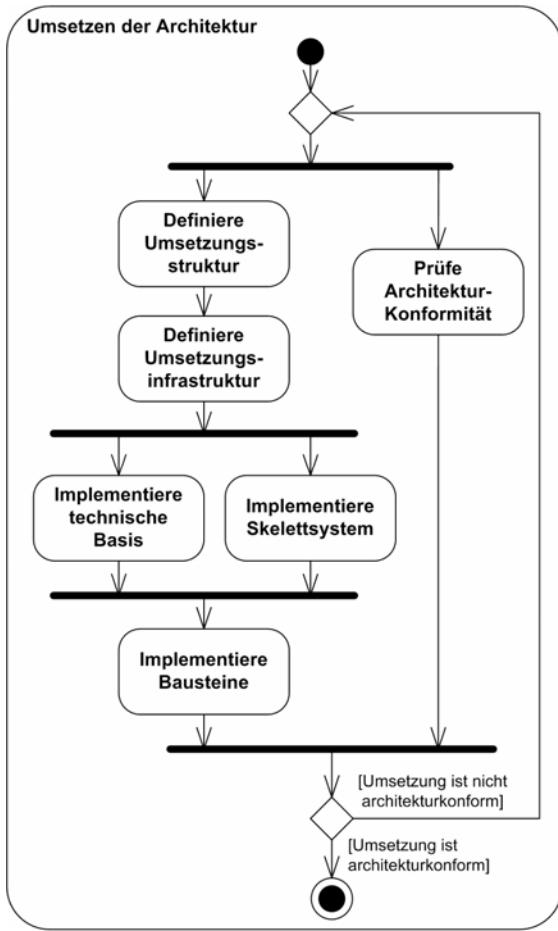


Abb. 8.6-2: Umsetzen der Architektur.

Die beim Entwerfen einer Architektur entstandene logische Sicht des Systems muss sich auch in der Realisierung widerspiegeln. Daher legt ein Architekt fest, wie die Bausteine der logischen Sicht auf die Bausteine der Umsetzungssicht abgebildet werden. Das Ergebnis dieser Aktion ist die Umsetzungsstruktur. Darunter ist die Dekomposition des Quelltextes in handhabbare Einheiten zu verstehen. Die Möglichkeiten werden dabei durch die eingesetzte Realisierungstechnologie beschränkt. Kommt für die Realisierung beispielsweise Java zum Einsatz, stehen dem Architekten die Einheiten Paket (englisch: *package*), Klasse und Schnittstelle zur Verfügung. Der Architekt des MIS müsste also festlegen, wie die Subsysteme, Bausteine und Schnittstellen von MIS auf Pakete, Klassen und Schnittstellen der Java-Technologie abzubilden sind.

Definiere Umsetzungsstruktur

Abbildungsregeln und Namenskonventionen festlegen

Neben einer ersten konkreten Abbildung sollten auch Abbildungsregeln und Namenskonventionen definiert werden. Alle Realisierungseinheiten, die die Schnittstelle eines Bausteins bzw. eines Subsystems definieren, sollten in dedizierten Schnittstellenpaketen zusammengefasst werden. Dahingegen sollten Realisierungseinheiten, die die Schnittstelle implementieren, in internen Paketen positioniert werden. Ferner sollte zwischen öffentlichen und publizierten Schnittstellen unterschieden werden [Fowler 2002]. Öffentliche Schnittstellen sind Schnittstellen, die innerhalb der eigenen Quelltextbasis benutzt werden. Solche Schnittstellen können leichter geändert werden, da Verwender der Schnittstelle identifiziert und werkzeuggestützt angepasst werden können. Publizierte Schnittstellen können hingegen nur schwer geändert werden, da sie über Systemgrenzen hinaus genutzt werden und sich der Quelltext nicht in der eigenen Hoheit befindet. Im MIS-Beispiel sind dies die Schnittstellen zu den Umsystemen, wie VMS und CCMS. Ab einer bestimmten Systemgröße gilt dies jedoch bereits für Subsystemgrenzen, da es auch hier ggf. eine klare Quelltextverantwortlichkeit gibt und Änderungen nicht einfach möglich sind. Zur besseren Unterscheidung zwischen öffentlichen und publizierten Schnittstellen sollte die Umsetzungsstruktur diese Schnittstellen klar voneinander trennen, indem z. B. unterschiedliche Pakete verwendet werden.

Definiere Umsetzungsinfrastruktur

Bei der Umsetzung einer Architektur muss nicht nur auf die Übereinstimmung der Realisierung mit dem ursprünglichen Entwurf geachtet werden, sondern auch darauf, dass die Umsetzung möglichst effektiv und reibungslos verläuft. Dies bedingt, dass neben der Strukturierung des Systems auch die Auswahl der Architekturmittel (Entwicklungsumgebung, Programmiersprache etc.) für die Umsetzung in die architektonische Arbeit einbezogen wird. Die Gesamtheit dieser Mittel bezeichnet man als Umsetzungsinfrastruktur. Die Umsetzungsinfrastruktur umfasst unter anderem:

- > Programmierumgebungen
- > Modellierungsumgebungen
- > Build-Umgebungen
- > Konfigurationsmanagement-Umgebungen
- > Testumgebungen

Definition der Umsetzungsinfrastruktur ist essenziell

Im Rahmen der Definition der Programmierumgebung, wie z. B. Eclipse, wird man festlegen, welche Quelltext-Formatierungen und -Vorlagen zu verwenden sind, wie ein Anwendungsserver einzubinden ist oder wie der Zugriff auf das Konfigurationsmanagement-Repository erfolgt. Oftmals wird die Definition der Umsetzungsinfrastruktur vernachlässigt.

Das Ergebnis ist eine heterogene Umsetzungsinfrastruktur, die die Effizienz der Umsetzung beeinträchtigt.

Zielsetzung bei der Auswahl der Architektur-Mittel für die Umsetzungsinfrastruktur ist ein möglichst effizientes Zusammenarbeiten der einzelnen Entwickler am Gesamtsystem. Um dies zu erreichen, muss es standardisierte Vorgehensweisen geben, die festlegen, wie z. B.

- > Änderungen in das Gesamtsystem integriert werden,
- > die einzelnen Bestandteile des Gesamtsystems erzeugt und installiert werden und
- > das Gesamtsystem getestet wird.

Effizientes Zusammenarbeiten ermöglichen

Die konkrete Ausgestaltung der Vorgehensweisen ist dabei stark von den jeweiligen Gegebenheiten abhängig. Neben den eingesetzten Architektur-Mitteln nehmen auch solche Faktoren wie die Anzahl der involvierten Entwickler, die Verteilung auf ein oder mehrere Entwicklungsstandorte etc. Einfluss auf die Vorgehensweisen.

Bevor ein System auf breiter Front realisiert werden kann, muss die technische Basis geschaffen werden. Zur technischen Basis zählen zum einen die benötigten technischen Basisdienste (siehe Abschnitt 8.5) und zum anderen die Implementierung der Frameworks, mit denen die fachlichen Bausteine realisiert werden. Sowohl die technischen Basisdienste wie auch die Frameworks sollten minimalinvasiv realisiert werden. Mit anderen Worten sollte die Abhängigkeit der fachlichen Bausteine von den technischen Bausteinen und Frameworks so gering wie möglich sein. Ebenso sollte die technische Basis Aspekte der Plattform soweit wie möglich abstrahieren und kapseln. Im Rahmen von MIS sollte es beispielsweise möglich sein, die fachlichen Bausteine sowohl innerhalb als auch außerhalb eines JEE-Servers zu testen. Eine sinnvolle technische Basis kann jedoch nur entstehen, wenn die Bedürfnisse der fachlichen Bausteine ausreichend bekannt sind. Dies ist ein Grund für die Implementierung eines Skelettsystems. Darüber hinaus wird durch ein Skelettsystem deutlich, wie man das System architekturkonform implementiert.

Implementiere technische Basis

Ein Skelettsystem entspricht in seiner Struktur bereits der Architektur des endgültigen Systems. Die einzelnen Bausteine hingegen stellen jedoch noch nicht eine vollständige Implementierung der Funktionalität zur Verfügung. Vielmehr stellen sie nur die Funktionalität zur Verfügung, die zur Abbildung eines klar abgegrenzten Anwendungsfalls benötigt wird. Alle weiteren Funktionalitäten sind entweder nicht im-

Implementiere Skelettsystem

plementiert oder mit provisorischen Implementierungen hinterlegt. So erhält man möglichst früh ein „komplettes, lauffähiges“ System für einen klar abgegrenzten Anwendungsfall, welches alle tragenden Bausteine bereits beinhaltet, wenngleich noch nicht vollständig realisiert. Der funktionale Umfang des Anwendungsfalls ist hierbei zweitrangig. Er muss jedoch so gewählt sein, dass auch alle tragenden Bausteine in die Realisierung des Anwendungsfalls involviert sind. Nachdem das Skelett des Gesamtsystems realisiert wurde, können nun die einzelnen Bausteine weiterentwickelt werden und so der funktionale Umfang des Systems weiter erhöht werden. Dieser Ansatz hat den Vorteil, dass zu jedem Zeitpunkt der Entwicklung ein „lauffähiges“ System zur Verfügung steht. Auf diese Weise ist eine kontinuierliche Überprüfung der gestellten Anforderungen an dem bereits existierenden System möglich. Außerdem kann das Augenmerk zu Anfang auf kritische oder komplexe Aspekte des Systems gerichtet werden, wodurch sich eine Risikoreduktion ergibt.

Technische Basis und Skelettsystem parallel entwickeln

Die technische Basis und das Skelettsystem sollten parallel implementiert werden, um Rückschlüsse aus der Implementierung des Skelettsystems in die Implementierung der technischen Basis einfließen lassen zu können. Die technische Basis wird dabei immer dem Skelettsystem eine Iteration voraus sein.

Implementiere Bausteine

Ein Architekt sollte nicht nur an der Implementierung der technischen Basis und des Skelettsystems beteiligt sein, sondern auch im weiteren Verlauf konkrete fachliche Bausteine implementieren. Hierdurch wird sichergestellt, dass man ein unmittelbares Feedback zur Zweckmäßigkeit seiner Architektur erhält und Schwachstellen frühzeitig eliminieren kann. Außerdem stärkt die Mitarbeit eines Architekten seine Akzeptanz im Team (siehe Kapitel 7).

Prüfe Architektur-Konformität

Eine wesentliche Aufgabe beim Umsetzen der Architektur respektive von Architektur-Richtlinien ist die Prüfung und Sicherstellung der Architektur-Konformität. Dies wird auch als Architektur-Durchsetzung (englisch: *architectural enforcement*) bezeichnet. Mit der ersten geschriebenen Zeile Quelltext oder der ersten modellierten Abstraktion ist die Konformität der Arbeitsergebnisse mit einer definierten Architektur sicherzustellen. Sonst werden mögliche Architektur-Erosionen zu spät erkannt. Man hat unterschiedliche Mittel zur Hand, um die Konformität eines implementierten Systems mit einer geplanten Architektur sicherzustellen.

Das einfachste Mittel ist die kontinuierliche, manuelle Überprüfung des geschriebenen Quelltextes auf seine Übereinstimmung mit der Architektur. Aufgrund seiner Einfachheit ist dieses Mittel recht schnell und kurzfristig verfügbar. Nachteil der manuellen Überprüfung ist der damit verbundene Aufwand und die unvollständige Abdeckung der überprüften Bausteine.

Kontinuierliche, manuelle Überprüfung

Neben der manuellen Prüfung können auch automatisierte Prüfungen von Quelltext durchgeführt werden. Hierzu existieren im Opensource-Bereich für Java verschiedene Werkzeuge, wie z. B. Findbugs, Checkstyle oder PMD. In die Analyse sollten unbedingt Metriken aufgenommen werden, die eine Aussage über die Qualität des Quelltextes erlauben. Hierzu gehören beispielsweise:

- > Zyklomatische Komplexität (englisch: *cyclomatic complexity*) [McCabe 1976]
- > Methodenlängen
- > Klassenlängen
- > Vererbungstiefe und -breite
- > Anzahl der Attribute und Methoden einer Klasse
- > Kohäsion von Methoden einer Klasse
- > Menge und Umfang von Copy&Paste-Verletzungen

Einsatz von Werkzeugen zur Quelltext- und Modellanalyse

Metriken dürfen jedoch nicht blind und als einzige Bewertungskriterien als Basis für architektonische Entscheidungen angewendet werden. Man muss Metriken von Fall zu Fall unter Einschluss weiterer Parameter (z. B. besondere Anforderungen) aus dem jeweiligen Kontext bewerten. Dabei ist es ebenfalls wichtig, den Einsatz von Metriken im Vorfeld zu kommunizieren (siehe Abschnitt 8.7). Des Weiteren lässt sich die Einhaltung von Richtlinien (für Programmierung, Namensgebung etc.) mittels solcher Werkzeuge überprüfen oder sogar erzwingen. Darüber hinaus bieten viele Modellierungsumgebungen die Möglichkeit, Modellierungsregeln zu definieren und bei der Modellierung automatisch zu prüfen. Dadurch wird gewährleistet, dass bereits beim Entwurf potenzielle Probleme identifiziert werden.

Metriken nicht blind anwenden

Eine weitere Möglichkeit, um die korrekte Umsetzung einer Architektur sicherzustellen, besteht darin, eine Architektur über ein Skelettsystem als Schablone vorzugeben. Den Entwicklern ist es dabei nur noch gestattet, sich im Rahmen des Skelettsystems zu bewegen. Voraussetzung hierfür ist jedoch, dass ein derartiges Skelettsystem alle architektonischen Entscheidungen auch umsetzt.

Skelettsystem als Schablone

Frameworks

Andere Möglichkeiten zur Erzielung einer guten Übereinstimmung von Architektur und Realisierung sind Frameworks (siehe Abschnitt 6.2). Frameworks geben dem Framework-Anwender anhand von Schnittstellen und abstrakten Implementierungen Richtlinien für die Realisierung einer Lösung vor. Diese Schnittstellen und abstrakten Implementierungen reflektieren dabei die vom Architekten erstellte Architektur. Damit bewirken sie eine gute Übereinstimmung der fertig implementierten Lösung mit einer vorgegebenen Architektur. Für typische Anwendungsfälle des Frameworks müssen Beispiele vorgegeben werden, welche die Entwickler als Referenz verwenden können, um eigene Funktionalität zu implementieren.

Generative Ansätze und modellgetriebene Software-Entwicklung

Eine andere Möglichkeit zur Erzielung einer möglichst guten Übereinstimmung von implementiertem System und Architektur-Vorgaben ist der Einsatz von Quelltext-Generatoren (siehe Abschnitt 6.2.5) und modellgetriebener Software-Entwicklung (siehe Abschnitt 6.2.6). Sie ermöglichen die Generierung von Quelltext anhand von Spezifikationen. Spezifikationen können in Form von Modellen vorliegen und ermöglichen so eine Beschreibung des gesamten Systems oder auch von Teilen auf einem höheren Abstraktionsniveau. Der Generator verwendet diese Modelle zur Erstellung von Quelltext für größere Teile des Gesamtsystems. Hierbei bewirkt der Generator, dass der erzeugte Quelltext der Architektur genügt, indem alle architekturrelevanten Realisierungsbauusteine generiert werden.

Checkliste: Umsetzen der Architektur

- Wird die Einhaltung von Architektur-Richtlinien automatisiert geprüft?
- Wird die korrekte Umsetzung der Architektur automatisiert sichergestellt?
- Existiert in der Dokumentation zu jeder Metrik je ein Beispiel für den Gute- und Schlecht-Fall?
- Existieren Regeln für die Abbildung von Bausteinen der logischen Sicht auf Bausteine der Umsetzungssicht?
- Findet sich jeder Baustein der logischen Sicht in der Umsetzungssicht wieder?
- Wurde die Umsetzungsinfrastruktur auf die organisatorischen Rahmenbedingungen abgestimmt?
- Wurde die Umsetzungsinfrastruktur gemäß Architektur-Richtlinien konfiguriert?
- Gibt es eine Anleitung (englisch: *cookbook*) für die Umsetzungsinfrastruktur?
- Existiert eine technische Basis?
- Genügt die technische Basis der vorgegebenen Architektur?
- Deckt die technische Basis alle architekturrelevanten Anwendungsfälle ab?

Zusammenfassung

- > Ein Architekt ist für die Umsetzung der Architektur verantwortlich.
- > Im Rahmen der Tätigkeit „Umsetzen der Architektur“ muss die Umsetzungsstruktur definiert werden. Diese sollte aus der logischen Sicht abgeleitet werden.
- > Es sind Abbildungsregeln und Namenskonventionen festzulegen.
- > Eine Umsetzungsinfrastruktur (Entwicklungsumgebung, Programmiersprache, etc.) muss definiert werden. Diese Infrastruktur muss ein effizientes Zusammenarbeiten ermöglichen.
- > Die technische Basis und das Skelettsystem sind zu implementieren. Diese sollten parallel entwickelt werden, wobei die Basis dem Skelettsystem immer eine Iteration voraus sein sollte.
- > Ein Architekt sollte auch selber Bausteine implementieren, um Rückmeldung (englisch: *feedback*) zur Zweckmäßigkeit seiner Architektur zu erhalten und eine höhere Akzeptanz im Team zu erreichen.
- > Die Konformität der Umsetzung mit der definierten Architektur muss geprüft werden. Dies sollte nach Möglichkeit automatisiert erfolgen.

8.7 Kommunizieren der Architektur

Abbildung 8.7-1 stellt die grundlegenden Konzepte der Tätigkeit „Kommunizieren der Architektur“, welche in diesem Abschnitt behandelt und detailliert werden, vor und visualisiert ihren Zusammenhang.

Grundlegende Konzepte der Tätigkeit „Kommunizieren der Architektur“

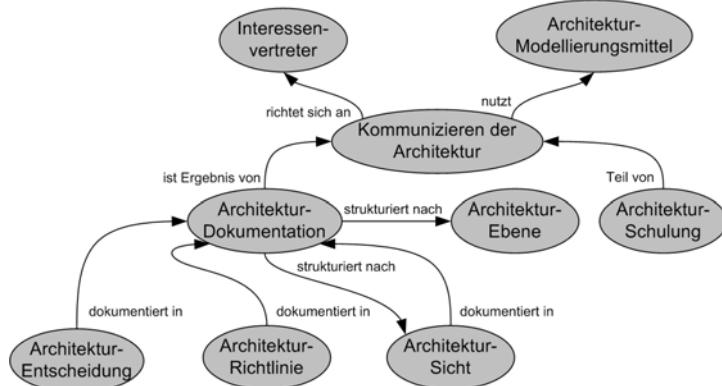


Abb. 8.7-1: Grundlegende Konzepte der Tätigkeit „Kommunizieren der Architektur“.

Kontinuierliche Kommunikation

Eine Architektur stellt für alle Interessenvertreter ein Modell zur Verfügung, mit dem sie arbeiten und untereinander kommunizieren können. Voraussetzung hierfür ist jedoch, dass ein Architekt gezielt und unmissverständlich eine von ihm entworfene Architektur laufend, also im Rahmen aller architektonischen Tätigkeiten (siehe Abbildung 8.2-1), an die jeweils relevanten Interessenvertreter vermittelt. Dadurch erhalten alle Interessenvertreter ein Verständnis sowohl für ein zu realisierendes System als auch für die vor ihnen liegende Arbeit und die Zusammenarbeit im Team bzw. in der Projektorganisation (siehe Kapitel 7). Folgende Beispiele sollen verdeutlichen, warum es von großer Relevanz ist, eine Architektur an alle Interessenvertreter zu kommunizieren [Bass et al. 2003]:

- > Entwickler müssen verstehen können, wie sie eine Architektur korrekt umsetzen müssen.
- > Tester müssen verstehen können, welche zu testenden Bausteine eine Architektur ihnen vorgibt.
- > Manager müssen verstehen können, inwieweit sich eine Architektur auf die Projektplanung auswirkt.
- > Kunden müssen verstehen können, warum die Berücksichtigung bestimmter Anforderungen einen Mehraufwand bei dem Entwurf einer Architektur mit sich bringt.

Team in architektonische Entscheidungen einbeziehen

Es ist von großer Relevanz für die Akzeptanz, das Verständnis und die korrekte Umsetzung einer Architektur, dass man „seine“ Entwickler in architektonische Entscheidungen einbezieht. In Hinblick auf die verschiedenen anderen Interessenvertreter ist die Kommunikation einer Architektur wichtig, weil diese es ermöglicht zu überprüfen, ob und wie ein zukünftiges System die gestellten Anforderungen erfüllt sowie die Akzeptanz des zukünftigen Systems zu verbessern.

In Abbildung 8.7-2 sind die einzelnen Aktionen der architektonischen Tätigkeit „Kommunizieren der Architektur“ und deren Ablauf dargestellt.

Dokumentiere Architektur-Richtlinien

Das Festlegen von Architektur-Richtlinien ist eine zentrale Aufgabe eines Architekten und geschieht im Rahmen aller architektonischen Tätigkeiten. Es gibt verschiedene Architektur-Richtlinien für verschiedene Aspekte und Interessenvertreter. Damit man wirkungsvoll Architektur-Richtlinien kommunizieren und deren Anwendung sicherstellen kann, muss man diese explizit als wichtigen Teil einer Architekturdokumentation festhalten. Es genügt nicht, Architektur-Richtlinien nur mündlich zu kommunizieren.

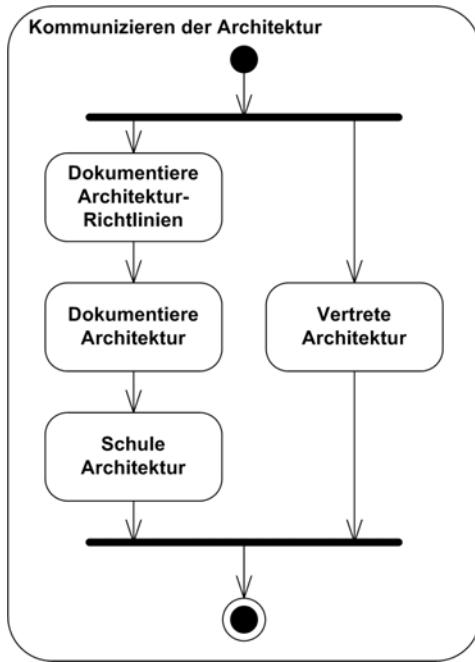


Abb. 8.7-2: Kommunizieren der Architektur.

Architektur-Richtlinien unterstützen einen Architekten bei der Sicherstellung von

- > Qualität,
- > korrekter Umsetzung und
- > Kommunikation

einer Architektur.

Architektur-Richtlinien werden für zwei Bereiche festgelegt:

- > Wie Architektur-Mittel zu verwenden/einzusetzen sind.
- > Wie Architektur zu dokumentieren ist (dazu später mehr in diesem Abschnitt).

Beispiele für Architektur-Mittel, für die Architektur-Richtlinien zu erstellen sind:

- > Verwendung der objektorientierten Konzepte in Java.
- > Namenskonventionen, z. B. für Bausteine.
- > Programmierung in Java.
- > Verwendung von UML-Diagrammen.

Zweck von Architektur-Richtlinien

Bereiche von Architektur-Richtlinien

Gegenstände von Architektur-Richtlinien

Wie sind Architektur-Richtlinien zu dokumentieren?

Bei der täglichen Arbeit müssen Architektur-Richtlinien einfach zu verwenden sein. Liegt die Dokumentation von Architektur-Richtlinien als bürokratisches Machwerk von etlichen Seiten Papier vor, werden Architektur-Richtlinien eher nicht konsequent beachtet oder sogar missachtet. Deshalb sollten sich Architektur-Richtlinien auf eine überschaubare Anzahl wirklich relevanter und nachvollziehbarer Festlegungen beschränken. Diese Festlegungen sind prägnant, einheitlich und verständlich zu beschreiben. Die Struktur der Dokumentation von Architektur-Richtlinien ist also so zu wählen, dass diese sich ohne unnötigen Aufwand benutzen lässt und die Kommunikation der Architektur-Richtlinien unterstützt. Die Dokumentation einer Richtlinie sollte deshalb immer aus mindestens folgenden Teilen bestehen:

- > Richtlinie als solche.
- > Begründung für die Richtlinie.
- > Beispiel für die Anwendung der Richtlinie.
- > Beispiel für die Verletzung der Richtlinie.

Eine beispielhafte Richtlinie für MIS wurde in Tabelle 8.7-1 festgehalten.

Tab. 8.7-1: Richtlinie am Beispiel von MIS.

Richtlinie	
Datenaustausch zwischen Bausteinen über die Grenzen logischer Schichten hinweg erfolgt über Datentransferobjekte (DTO).	
Begründung	Bessere Entkopplung der Schichten und Verringerung der Netzlast.
Beispiel für Anwendung	siehe Abbildung 8.5-11
Beispiel für Verletzung	Bausteine des Web-Rahmenwerks (englisch: <i>web framework</i>) werden benutzt, um Daten zwischen Präsentationslogikschicht und Anwendungslogikschicht auszutauschen.

Architektur-Richtlinien nicht auf der grünen Wiese erstellen

Bevor mit der Erstellung von Architektur-Richtlinien begonnen wird, sollte geprüft werden, ob an anderer Stelle die gewünschten Architektur-Richtlinien bereits existieren und diese als Basis verwendet werden. So gibt es beispielsweise im Umfeld von UML Regeln für die Schreibweise von Bezeichnern als Quasi-Standard. Derartige Standards sollten jedoch nicht ohne Weiteres benutzt werden, sondern auf die spezifischen Anforderungen (z. B. Projektgröße) angepasst werden.

Zu welchem Zeitpunkt sind Architektur-Richtlinien zu erstellen?

Die zweckmäßige Verwendung von Architektur-Mitteln und Erstellung von Artefakten erfordert die Beachtung von Architektur-Richtlinien. Daraus ergibt sich, dass Architektur-Richtlinien laufend und frühzeitig

erstellt werden müssen. Architektur-Richtlinien erst im Nachhinein zu erstellen und/oder zu berücksichtigen, ist, wenn überhaupt, nur mit großem Aufwand machbar.

Es ist erforderlich, dass Architektur-Richtlinien auch für die Erstellung der Architektur-Dokumentation und für die Verwendung von Architektur-Modellierungsmitteln festlegt werden. Dies sollte geschehen, noch bevor damit begonnen wird, eine Architektur-Dokumentation zu erstellen. Es folgt eine Liste der Fragen (kann bei Bedarf ergänzt werden), die durch Architektur-Richtlinien für die Architektur-Dokumentation beantwortet werden sollten:

- > Was sind die Inhalte der Architektur-Dokumentation?
- > Wie ist die Architektur-Dokumentation strukturiert?
- > Wie wird die Architektur-Dokumentation organisiert?
- > Wie wird die Architektur-Dokumentation auf einem aktuellen Stand gehalten?
- > Welche Regeln gelten für den Schreibstil?
- > Auf welche Weise sind Architektur-Modellierungsmittel zu verwenden?
- > Welche Landessprache ist für Bezeichner (aller Art) zu verwenden?
- > Welche Schreibweisen sind für Bezeichner zu verwenden?

Ein Architekt kommuniziert eine Architektur im Rahmen von Präsentationen, Reviews, Schulungen etc. Direkt oder indirekt kommt bei diesen Aktivitäten eine Architektur-Dokumentation zum Tragen. Der Architektur-Dokumentation kommt deshalb für das Kommunizieren einer Architektur eine Schlüsselrolle zu.

Nach folgenden zwei wesentlichen Zielsetzungen sollte Architektur-Dokumentation ausgerichtet werden [Bredemeyer und Malan 2004]:

- > Entscheidungen eines Architekten vollständig und unmissverständlich festhalten.
- > Architektur abgestimmt auf die verschiedenen Interessensvertreter bzw. Zielgruppen (Kunde, Projektleiter, Software-Entwickler etc.) kommunizieren.

Eine Architektur hat nur dann eine Chance, tatsächlich wie vom Architekten vorgesehen zur Umsetzung zu kommen, wenn die zugehörige, nach obigen Zielsetzungen und nach bestimmten Regeln erstellte Architektur-Dokumentation verfügbar ist und ein Architekt diese aktiv kommuniziert (siehe Abschnitt 7.6). Erst eine derart explizite Architektur

Architektur-Richtlinien für Architektur- Dokumentation auf- stellen

Dokumentiere Architektur

Wesentliche Zielsetzungen

Architektur- Dokumentation ist notwendig für korrekte Architektur-Umsetzung

kann vermittelt, verstanden und umgesetzt werden. Andernfalls wird eine Architektur, auch wenn diese noch so gut durchdacht ist, sich aber weitgehend nur im Kopf eines Architekten befindet, nur teilweise oder falsch umgesetzt werden. Oder diese wird dazu bestimmt sein, in Form der zugehörigen Architektur-Dokumentation im Regal zu verstauben. Ein bedeutender Nebeneffekt der Architektur-Dokumentation ist, dass es oft erst durch ihre Erstellung bestimmte Erkenntnisse zu einer Architektur selbst geben kann [Jeckle et al. 2004], die dann einer Architektur und damit ihrer Qualität zugute kommen. Architektur steht also in Wechselbeziehung zu ihrer Dokumentation.

Aspekte, die in keiner Architektur-Dokumentation fehlen sollten

Architektur-Dokumentation sollte in jedem Fall zumindest die folgenden Aspekte beschreiben [Bredemeyer und Malan 2004]:

- > Architektur-Entscheidungen (siehe Abschnitte 8.2 – 8.6)
- > Architektur-Sichten (siehe Kapitel 4)
- > Architektur-Anforderungen (siehe Kapitel 5 und Abschnitte 8.2 – 8.3)

Architektur-Entscheidungen dokumentieren

Die explizite Dokumentation der Architektur-Entscheidungen (siehe Abschnitt 7.5) ist ein ganz wesentlicher Punkt in einer Architektur-Dokumentation. Erst wenn die Architektur-Entscheidungen bekannt sind, die zu einer bestimmten Architektur geführt haben, kann diese nachvollzogen und auch begründet werden – und dies auch noch nach einem langen Zeitraum. Architektur-Entscheidungen manifestieren sich in den Ergebnissen des Architektur-Entwurfs, welche sich in der Architektur-Dokumentation wiederfinden. Jedoch gehen z. B. aus einem UML-Diagramm, welches eine logische 4-Schichtenarchitektur zeigt, nicht automatisch sämtliche zugrunde liegenden Architektur-Entscheidungen klar hervor. Deshalb ist es notwendig, zusätzlich zu den Ergebnissen des Architektur-Entwurfs auch die Architektur-Entscheidungen dediziert zu dokumentieren, z. B. tabellarisch.

Wichtige Architektur-Entscheidungen bei MIS sind zum Beispiel:

- > Priorisierung der Anforderungen (siehe Tabellen 8.4-2 und 8.4-3).
- > Schnittstellenprotokolle zu den Umsystemen (siehe Abbildung 8.5-4).
- > Identifikation der Schlüsselabstraktionen (siehe Abbildung 8.5-5).

Gegenstände von Architektur-Entscheidungen

Die Gegenstände von Architektur-Entscheidungen sind:

- > Architekturelevante Anforderungen (z. B. Skalierbarkeit).
- > Auswahl bestimmter Architektur-Mittel (z. B. Auswahl eines Musters oder einer Plattformtechnologie).

- > Art und Weise der Anwendung der ausgewählten Architektur-Mittel (z. B. es fehlt ein bestimmter Baustein, den ein ausgewähltes Muster jedoch vorsieht).
- > Struktur der Bausteine (z. B. bestimmte Bausteine kollaborieren nicht über ihre Schnittstellen, sondern direkt).

Daran schließen sich folgende Fragen an, die mittels explizit dokumentierten Architektur-Entscheidungen beantwortet werden können:

- > Welches sind die architekturelevanten Anforderungen und warum?
- > Warum wurden bestimmte Architektur-Mittel ausgewählt?
- > Warum wurden die Architektur-Mittel auf diese Art und Weise angewendet?
- > Warum liegen die Bausteine in dieser Struktur vor?

In Abbildung 8.7-3 wird der Kontext gezeigt, in dem sich eine Architektur-Dokumentation befindet. Auf die in der Abbildung 8.7-3 dargestellten und bisher noch nicht erläuterten Sachverhalte wird anschließend näher eingegangen.

Kontext von Architektur- Dokumentation

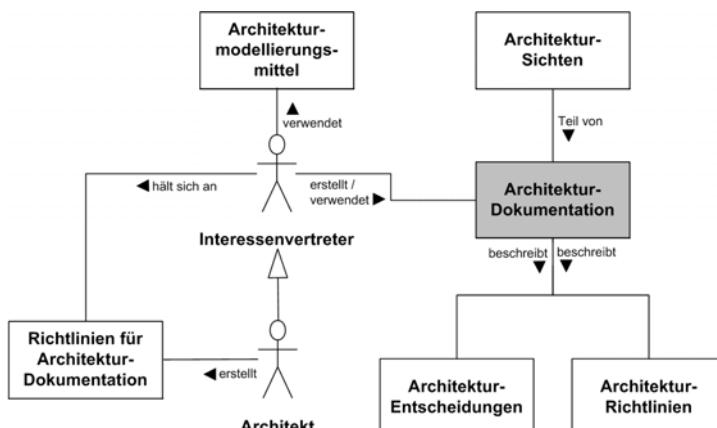


Abb. 8.7-3: Architektur-Dokumentation im Kontext.

Ein Architekt erstellt weite Teile einer Architektur-Dokumentation. Jedoch werden auch andere Interessenvertreter (Analysten, Entwickler etc.) Beiträge für eine Architektur-Dokumentation liefern. Die Architektur-Dokumentation wird unter Beachtung von durch den Architekten vorgegebenen Richtlinien erstellt.

**Architekt erstellt
Architektur-
Dokumentation
nicht allein**

Architektur-Dokumentation muss Zielgruppen berücksichtigen

Eine Architektur ist für ganz unterschiedliche Interessenvertreter von Bedeutung. Die Architektur-Dokumentation sollte deshalb in unterschiedlicher Form vorliegen, damit sie den verschiedenen Blickwinkeln der unterschiedlichen Zielgruppen gerecht werden kann. Z. B. ist für die Zielgruppe der Entscheidungsträger eine Folienpräsentation, welche mit einfachen grafischen Mitteln (Box-and-Lines-Diagramme) nur die wesentlichen Bausteine eines Systems aufzeigt, ohne dabei auf technische Details wie Schnittstellen einzugehen, völlig ausreichend. Die Darstellung eines semi-formalen Modells mittels der Unified Modeling Language (UML) oder einer Architecture Description Language (ADL) für diese Zielgruppe hätte nicht die gewünschte Wirkung. Hingegen wäre es fatal, ausgehend von einer solchen Folienpräsentation ein System zu entwerfen und zu realisieren, weil hier wichtige Architektur-Aspekte fehlen. Das heißt, man würde eine Pseudo-Architektur (siehe Abschnitt 1.1) verwenden. Die Auswahl der Architektur-Modellierungsmittel (siehe Abschnitt 6.6) und die Form einer Architektur-Dokumentation sollten also nicht dem Zufall überlassen werden, sondern müssen ganz bewusst mit einer bestimmten Zielsetzung erfolgen.

Standard-Architektur-Modellierungsmittel verwenden

Wichtig für die Wartung und die Kommunikation einer Architektur-Dokumentation ist die Auswahl eines Architektur-Modellierungsmittels. Die Verwendung von etablierten Standards (siehe Abschnitt 6.6) bringt hier erhebliche Vorteile:

- > Verwendung des Architektur-Modellierungsmittels ist dokumentiert
- > Architektur-Modellierungsmittel ist getestet und hat sich bewährt.
- > Bessere Unterstützung bei Fragen und Problemen.
- > Unterstützung des Architektur-Modellierungsmittels durch Werkzeuge verschiedener Hersteller.

Folgen falscher Verwendung von Architektur-Modellierungsmitteln

Bei der Verwendung von Architektur-Modellierungsmitteln gibt es Freiheitsgrade (z. B. in der Wahl der Landessprache für Bezeichner oder bei der Ausschöpfung bestimmter Möglichkeiten eines Architektur-Modellierungsmittels). Werden diese Freiheitsgrade ungeplant genutzt, kann die Qualität einer Architektur-Dokumentation in folgenden Punkten Schaden nehmen:

- > Konsistenz
- > Verständlichkeit
- > Eindeutigkeit
- > Wartbarkeit

Eine derart „schadhafte“ Architektur-Dokumentation hat weitreichende unerwünschte Auswirkungen. Sie wirkt sich in der Folge negativ auf eine Architektur an sich aus, das heißt, in letzter Konsequenz leidet schließlich die Software-Qualität des Systems. Beispielsweise wirkt sich eine ungeschickte Namensgebung (z. B. für Bausteine) in einer Architektur-Dokumentation bis auf den Quelltext aus. Der Quelltext wird als Folge unleserlich und damit schwer wartbar.

Mangelhafte Qualität der Architektur-Dokumentation bedingt mangelhafte Software-Qualität

Alle in Kapitel 4 beschriebenen Sichten eines Systems können und sollten mit den Möglichkeiten des ausgewählten Architektur-Modellierungsmittels beschrieben werden. Dabei sind dann sowohl die statischen wie auch die dynamischen Strukturen zu berücksichtigen.

Architektur-Sichten werden dokumentiert

Bei MIS wird beispielsweise mit Qualitätsattributszenarien (siehe Tabelle 8.4-1) und priorisierten Anforderungen (siehe Tabellen 8.4-2 und 8.4-3) ein Teil der Anforderungssicht dokumentiert.

Unterschiedliche Architektur-Ebenen und -Sichten nicht vermischen

In einer Architektur-Dokumentation sollten die unterschiedlichen Architektur-Ebenen und -Sichten (siehe Kapitel 4) nicht miteinander vermischt werden. Vielmehr sollte eine explizite Unterteilung einer Architektur-Dokumentation in die unterschiedlichen Ebenen und Sichten vorgenommen werden. So erleichtert man dem Leser der Architekturdokumentation ein schrittweises Verständnis der verschiedenen Architektur-Aspekte. Die Ausführungen zum MIS-Beispiel (siehe Abschnitte 8.2 – 8.6) und zum Anwendungsszenario „Enterprise Application Integration“ (Abschnitt 8.8) verdeutlichen dies beispielhaft. Falls Aspekte unterschiedlicher Architektur-Ebenen in einem Diagramm dargestellt werden, sollten die unterschiedlichen Ebenen hervorgehoben werden. Ist dies nicht (in überschaubarer Form) möglich sind verschiedene Diagramme zu verwenden (für Beispiele hierzu siehe Kapitel 9).

Strukturierung von Architektur-Dokumentation

Den einen Standard für die Strukturierung einer Architektur-Dokumentation gibt es nicht. Aber es gibt zahlreiche Vorgehensweisen. Beispielsweise kann eine Architektur-Dokumentation nach Bausteinen und/oder Architektur-Sichten aufgeteilt bzw. organisiert werden. Eine Architektur-Dokumentation sollte sich an der in Tabelle 8.7-2 aufgeführten Vorlage orientieren.

Tab. 8.7-2: Vorlage für Architektur-Dokumentation.

Einleitung								
Schnellüberblick								
Auswahl von Anforderungen	Auswahl und Anwendung von Architekturmitteln			Struktur der Bausteine				
Architektur-Sichten								
Anforderungssicht	Logische-Sicht	Daten-sicht	Umsetzungssicht	Prozess-sicht	Verteilungs-sicht			
Sichtenübergreifende Aspekte								
Architektur-Richtlinien			Sonstiges					
Architektur-Beurteilung								
Projektsaspekte								
Offene Punkte								
Glossar								

Diese Vorlage zeigt eine grundlegende Struktur in Anlehnung an [IEEE 2007] für Architektur-Dokumentationen. Eine projektspezifische Ergänzung und Verfeinerung dieser Struktur ist zu empfehlen. Die Hauptelemente dieser Vorlage haben folgende wesentlichen Inhalte:

- > *Einleitung:* Motivation, Ziele, Zielgruppen und Leserleitfaden.
- > *Schnellüberblick:* Essentielle Architektur-Aspekte zusammengefasst mit Verweisen auf Vertiefungsteile in der Architektur-Dokumentation.
- > *Architektur-Sichten:* Inhalte gemäß des jeweiligen Standpunktes (englisch: *viewpoint*) einer Sicht. Wichtig ist es hier nicht bloß die Artefakte einfach aufzuführen, sondern auch die Architektur-Entscheidungen aufzuzeigen, die zu den Artefakten führten.
- > *Sichtenübergreifende Aspekte:* Aspekte, die sich nicht einzelnen Architektur-Sichten zuordnen lassen. Dazu gehören z. B. Richtlinien zur Verwendung von UML Diagrammen oder die Definition von Standpunkten.
- > *Architektur-Beurteilung:* Protokolle zu Architektur-Beurteilungen und verworfenen Architektur-Alternativen.
- > *Projektsaspekte:* Aspekte wie z. B. Iterationsplanung, Aufgabenverteilung oder Ausbildung.
- > *Offene Punkte:* Punkte, die noch einer Klärung bedürfen, z. B. unklare Anforderungen.
- > *Glossar:* Darf nie fehlen! Zentrale Begriffe respektive Synonyme.

Eine solche Strukturierung kann sowohl für die Architektur-Dokumentation verwendet werden, die in einem einzigen Artefakt gebündelt ist,

als auch für solche, die über mehrere Artefakte verteilt ist. Auch die Ablagestrukturen (z. B. die Verzeichnisstrukturen in einem Dateisystem) für die Artefakte einer Architektur-Dokumentation können an dieser grundlegenden Strukturierung ausgerichtet werden. Die Struktur für eine Architektur-Dokumentation sollte in jedem Fall auf Basis einer bewährten Vorlage erstellt werden. Ein Beispiel für eine solche Vorlage ist der ANSI/IEEE Standard 1471-2000 [Chaudron 2002, IEEE 2000 und Opengroup 2001]. In [Bass et al. 2003] finden sich ebenfalls nützliche Anregungen und Vorlagen für die sinnvolle Strukturierung von Architektur-Dokumentationen.

Ein erhebliches Problem bei der Verwendung einer Architektur-Dokumentation stellen Begriffe (aus einer Architektur-Dokumentation) dar, wenn diese von verschiedenen Interessenvertretern unterschiedlich interpretiert werden. Dies gilt ganz besonders für Schlüsselbegriffe aus einer Domäne. Um dieses Problem zu entschärfen werden in [Evans 2004] folgende Architektur-Richtlinien vorgeschlagen:

- > Ausschließlich Begriffe aus dem Domänenmodell verwenden.
- > Alternativen zu einem Begriff hinsichtlich ihrer Bedeutung diskutieren und festlegen, welcher der zur Auswahl stehenden Begriffe zukünftig einheitlich verwendet werden soll.
- > Sicherstellen, dass sich Änderungen hinsichtlich verwendeter Begriffe im Domänenmodell widerspiegeln.
- > Begriffe bewusst auf Korrektheit (Konsistenz, Widersprüche, Verständlichkeit etc.) prüfen.

Die textuelle Architektur-Dokumentation ist sinnvoll zu ergänzen durch eine audio-visuelle Architektur-Dokumentation, um die Vermittlung einer Architektur zu erleichtern. Dazu stehen eine Reihe von Medien und Mitteln zur Verfügung:

- > Podcast
- > Hörbücher
- > Videos/DVDs
- > UML Diagramme
- > Freihandgrafiken

Um Architektur zu dokumentieren und ihre Vermittlung zu erleichtern, ist es unerlässlich, neben einer Architektur-Dokumentation in Textform die wesentlichen Merkmale einer Architektur auch visuell vor Augen zu haben. Hierzu ist das Modell des zu entwickelnden Systems zu visualisieren. Die visualisierten Artefakte eines Modells sind ein wichtiger Teil

Begriffe sorgfältig auswählen

Ergänzende audio-visuelle Architektur-Dokumentation

Visualisierung des Architektur-Modells ergänzt schriftliche Architektur-Dokumentation

einer Architektur-Dokumentation. Wichtig: Die verschiedenen Aspekte einer Architektur sollten durch verschiedene Artefakte (z. B. UML-Diagramme) visualisiert werden. Mittels solcher visueller Artefakte können eine Architektur und die Auswirkungen sich ändernder Anforderungen auf diese einfacher dargestellt, vermittelt und diskutiert werden. Dazu gibt es eine Reihe von mehr oder weniger formalen Architektur-Modellierungsmitteln (siehe Abschnitt 6.6). Eine visuelle Architektur-Dokumentation kann jedoch auf keinen Fall eine schriftliche Architektur-Dokumentation ersetzen, sondern nur ergänzen. Sie tut dies, indem sie dazu verhilft, rasch einen Überblick über eine Architektur erlangen zu können und bestimmte Aspekte gezielt zu fokussieren, ohne sich in den Details eines Textes zu verlieren. Zudem dient die visuelle Darstellung gleichsam als Wegweiser durch die textuelle Beschreibung. Der letztgenannte Sachverhalt kommt besonders zum Tragen, wenn man sich beispielsweise als Architekt und/oder Entwickler neu in eine Architektur einarbeiten muss.

Visuelle Architekturdokumentation

Damit die visuelle Architektur-Dokumentation einen wirklichen Nutzen bringt, sollte sie nicht versuchen, alle Details der textuellen Beschreibung darzustellen, sondern nur die wichtigen Aspekte herausheben (z. B. Schnittstellen und Schichtenzugehörigkeit). So erleichtert die visuelle Architektur-Dokumentation den Zugang zu den Konzepten, die in der textuellen Architektur-Dokumentation enthalten sind. Darüber hinaus sollten durch die visuelle Architektur-Dokumentation mindestens folgende Fragen zu Systembausteinen beantwortet werden:

- > Welche Rollen haben die Bausteine?
- > Welche Beziehungen zwischen den Bausteinen gibt es?
- > Wie kommunizieren die Bausteine untereinander?

Diese Aspekte werden in der logischen Sicht dokumentiert.

Einfachen Zugriff auf Architekturdokumentation sicherstellen

Wichtig ist es sicherzustellen, dass auf eine Architektur-Dokumentation schnell und einfach zugegriffen werden kann. Hierzu ist einerseits eine entsprechende technische Infrastruktur (z. B. Verwendung eines CMS oder eines Repositorys) mit der Möglichkeit der Versionskontrolle wichtig, da sich Architektur-Dokumente über die Zeit weiterentwickeln werden.

Umfang von Architekturdokumentation

Die Dokumentation einer Architektur kann mehr oder weniger umfangreich sein und ganz verschiedene Arten von Artefakten (Texte, Diagramme, Präsentationen etc.) umfassen. Es stellt sich nun die Frage, wie viel Architektur-Dokumentation braucht der Mensch? Grundsätzlich gilt: Der Umfang einer Architektur-Dokumentation sollte gerade so groß

gewählt werden, dass eine Architektur von allen Interessenvertretern richtig verstanden und korrekt umgesetzt wird. Unterschiedliche projektspezifische Faktoren wie Anforderungen, Systemgröße, Projektorganisation, fachliche Erfahrung der Projektmitarbeiter usw. beeinflussen den Umfang. Deshalb können für diesen keine fixen Größen pauschal benannt werden. Die Herausforderung ist, den Umfang zweckmäßig zu halten. Dabei ist zu bedenken, dass die meisten neu erstellten Artefakte in Zukunft auch gepflegt werden müssen. Es ist möglich, den Umfang teilweise unter Kontrolle zu bringen, wenn folgende Regeln konsequent beachtet werden:

- > Ausschließlich architekturelle Aspekte dokumentieren (z. B. Schnittstellen).
- > Ausschließlich Artefakte erstellen, die einen echten Mehrwert bringen und tatsächlich benutzt werden (z. B. ist zu überlegen, ob es Sinn macht, zu jedem Anwendungsfall ein UML-Aktivitätendiagramm zu erstellen)
- > Redundanzen strikt vermeiden (z. B. Dokumentation einer Schnittstelle sollte nicht bei den jeweils die Schnittstelle implementierenden Bausteinen wiederholt werden).
- > „Weniger ist manchmal mehr“ beachten.
- > Architektur-Dokumentation muss „100 %“ aktuell sein, das heißt der Umfang darf die Pflege der Dokumentation nicht beeinträchtigen.

Es kann nicht davon ausgegangen werden, dass eine Architektur und die Architektur-Richtlinien allein aufgrund der Architektur-Dokumentation (sei sie auch noch so ausgereift) korrekt verstanden und umgesetzt werden. Es ist hierzu erforderlich, dass Architektur und Architektur-Richtlinien fortlaufend vom Architekten aktiv in die Köpfe der jeweiligen Interessenvertreter gebracht werden. Architektur und Architektur-Richtlinien können nur dann ihren vom Architekten angestrebten Zweck erfüllen, wenn diese auch von den Interessenvertretern beachtet und korrekt angewendet werden. Voraussetzung dafür ist, dass Architektur und Architektur-Richtlinien bekannt sind sowie akzeptiert und verstanden wurden. Deshalb müssen Architektur und Architektur-Richtlinien bewusst und mit Bedacht kommuniziert werden. Für die erfolgreiche Kommunikation von Architektur und Architektur-Richtlinien ist es absolut notwendig, dass diese nicht bloß mittels einer Architektur-Dokumentation angewiesen, sondern auch mündlich vermittelt respektive geschult werden. Dies kann z. B. im Rahmen einer Teamsitzung oder der Etablierung eines Architektur-Boards geschehen. Damit wird eine echte Akzeptanz von Architektur und Architektur-Richtlinien erreicht und einer falschen Verwendung bzw. einem fal-

Schule Architektur

schen Verständnis vorgebeugt, die erheblichen Aufwand bei der anschließenden Behebung von Folgefehlern (z. B. bei der Umsetzung einer Architektur) nach sich ziehen würden.

Schulungsplanung

Schulungen beanspruchen Ressourcen und müssen geplant werden. Entscheidungen hierzu sollte die Projektleitung nicht allein treffen, weil diese oft die technischen Fakten nicht abschließend beurteilen kann (z. B. soll zukünftig ein anderes Persistenz-Framework eingesetzt werden oder können unsere Java-Entwickler ohne Weiteres auf Ruby umsteigen?). Hier ist man also einmal mehr gefordert, als technischer Berater zu fungieren und den Schulungsbedarf der einzelnen Interessenvertreter zu bestimmen. Grundsätzlich nimmt der Schulungsaufwand zu, je mehr es um die Umsetzung einer Architektur geht.

Schulungsarten

Schulungen können im Rahmen von

- > Externen/internen Seminaren oder Workshops,
 - > Coaching-Maßnahmen,
 - > Präsentationen bei einer Teamsitzung und bei der
 - > Mitarbeit an einer Architektur
- erfolgen.

Vertrete Architektur

Ein Architekt sollte jederzeit für seine Architektur auch gegen (politische) Widerstände einstehen. Dies muss an verschiedenen Fronten geschehen. Gefahren für eine Architektur müssen frühzeitig erkannt werden (z. B. die mögliche Ablehnung einer Architektur-Entscheidung durch einen wichtigen Interessenvertreter). Eine Architektur steht im Spannungsfeld der verschiedenen Interessenvertreter und mit ihr auch der Architekt. Deshalb dürfen bestimmte Interessenvertreter nicht ausgebündet werden, wenn es darum geht, eine Architektur zu positionieren. Beispielsweise läuft ein Architekt Gefahr mit seiner Architektur keine Akzeptanz zu finden wenn er unter der Annahme, dass er ja „nur Techniker“ sei, die Architektur nicht ausreichend aktiv Richtung Management vertritt.

Checkliste: Kommunizieren der Architektur

- | |
|---|
| Werden Entwickler in architektonische Entscheidungen mit einbezogen? |
| Wird die Architektur an alle Interessenvertreter kommuniziert? |
| Werden Richtlinien an die jeweils betroffenen Interessenvertreter kommuniziert? |
| Wird die Architektur gegenüber allen Interessenvertretern vertreten? |

Checkliste: Architektur-Richtlinien

- | |
|---|
| Gibt es Richtlinien für die Verwendung von Architektur-Mitteln? |
| Gibt es Richtlinien für die Erstellung der Architektur-Dokumentation? |
| Werden Architektur-Richtlinien dokumentiert? |

Basieren Richtlinien auf Standards?

Werden Richtlinien laufend und rechtzeitig erstellt?

Werden standardisierte Architektur-Modellierungsmittel verwendet?

Wird der IEEE-Standard 1471 beachtet?

Werden mindestens konzeptionelle, logische und Verteilungssicht dokumentiert?

Werden mindestens Architektur-Entscheidungen, Architektur-Sichten und Architektur-Anforderungen dokumentiert?

Werden unterschiedliche Architektur-Ebenen und -Sichten nicht vermischt?

Liegt eine ergänzende audio-visuelle Architektur-Dokumentation vor?

Fokussiert sich die visuelle Architektur-Dokumentation nur auf architekturelle Aspekte?

Kann auf die Architektur-Dokumentation schnell und einfach zugegriffen werden?

Ist der Umfang der Architektur-Dokumentation angemessen zu Anforderungen, Systemgröße und Projektorganisation?

Erlaubt der Umfang der Architektur-Dokumentation noch eine uneingeschränkte Pflege der Architektur-Dokumentation?

Sind Architektur-Richtlinien Teil der Architektur-Dokumentation?

Umfasst die Architektur-Dokumentation ein Glossar?

Steht die Architektur-Dokumentation unter Versionsverwaltung?

Enthält die Architektur-Dokumentation Kontextinformationen wie Autor(en), Änderungshistorie, Version usw.?

**Checkliste:
Architektur-
Dokumentation**

Wird ein standardisiertes Architektur-Sichtenmodell verwendet?

Wurde das ausgewählte Architektur-Sichtenmodell adaptiert?

Sind alle verwendeten Sichten spezifiziert?

Haben verwendete Sichten keine Redundanzen?

Sind alle Sichten kohärent?

Werden statische und dynamische Strukturen berücksichtigt?

**Checkliste:
Architektur-Sichten**

- > Damit eine Architektur durch die Interessenvertreter verstanden und korrekt umgesetzt wird, ist es notwendig, sie laufend, während aller architektonischen Tätigkeiten zu vermitteln.
- > Es ist von großer Relevanz für die Akzeptanz, das Verständnis und die korrekte Umsetzung einer Architektur, dass man die Entwickler in architektonische Entscheidungen mit einbezieht.
- > Das Kommunizieren einer Architektur erfolgt mündlich, schriftlich und visuell.
- > Der Architektur-Dokumentation kommt für das Kommunizieren einer Architektur eine Schlüsselrolle zu.

Zusammenfassung

- > Die Architektur-Dokumentation dokumentiert eine Architektur und Architektur-Richtlinien.
- > Architektur-Richtlinien sollten laufend und frühzeitig erstellt sowie kommuniziert werden.
- > Wesentliche Aspekte, die die Architektur-Dokumentation in jedem Fall abdecken sollte, sind Architektur-Entscheidungen, Architektur-Sichten und Architektur-Anforderungen.
- > Die explizite Dokumentation der Architektur-Entscheidungen ist ein ganz wesentlicher Punkt in einer Architektur-Dokumentation. Erst wenn die Architektur-Entscheidungen bekannt sind, die zu einer bestimmten Architektur geführt haben, kann diese nachvollzogen und auch begründet werden.
- > Für die Erstellung einer Architektur-Dokumentation sollten standardisierte Architektur-Modellierungsmittel verwendet werden.
- > Bei der Erstellung einer Architektur-Dokumentation ist auf Konsistenz, Verständlichkeit, Eindeutigkeit und Wartbarkeit zu achten und durch entsprechende Richtlinien sicherzustellen.
- > Die Textuelle Architektur-Dokumentation ist sinnvoll zu ergänzen durch eine audio-visuelle Architektur-Dokumentation.
- > Die Visuelle Architektur-Dokumentation sollte nicht versuchen, alle Details der textuellen Beschreibung darzustellen, sondern nur die wichtigen Aspekte herausheben.
- > Wichtig ist es sicherzustellen, dass auf eine Architektur-Dokumentation schnell und einfach zugegriffen werden kann. Hierzu ist eine entsprechende technische Infrastruktur vorzusehen.
- > Der Umfang einer Architektur-Dokumentation ist unter anderem von Anforderungen, Systemgröße und Projektorganisation abhängig. Er kann eingeschränkt werden, indem man sich auf die wirklich architekturelevanten Aspekte beschränkt und Redundanzen strikt vermeidet. Es sollte beim Umfang keine Grenze überschreiten werden, die die Pflege einer Architektur-Dokumentation mit den gegebenen Ressourcen einschränkt.

8.8 Anwendungsszenario: Enterprise Application Integration

Ausgangssituation

Unternehmen betreiben eine Vielzahl von IT-Systemen auf unterschiedlichen Plattformen, die in verschiedenen Programmiersprachen entwickelt wurden und differenzierte Geschäftsanforderungen erfüllen. Die wenigsten IT-Systeme können dabei ohne den Austausch von Informati-

onen mit anderen IT-Systemen existieren. Dies hat in der Praxis zu einer Integration von IT-Systemen in einer Ad-hoc-Manier geführt, indem IT-Systeme über Punkt-zu-Punkt-Verbindungen (PzP-Verbindungen) aneinander gekoppelt wurden. Wenn man davon ausgeht, dass ein durchschnittliches Unternehmen 50 IT-Systeme besitzt und diese alle miteinander verbunden sind, ergeben sich daraus 2450 PzP-Schnittstellen. Daraus resultieren heutzutage sehr komplexe IT-Systemlandschaften. Wenn sich Geschäftsprozesse ändern und neue Geschäftsstrategien umgesetzt werden sollen, ist die Integration weiterer IT-Systeme notwendig. Um der geschilderten komplexen Situation zu begegnen und neue Geschäftsstrategien zu unterstützen, haben sich Enterprise Application Integration (EAI) innerhalb von Unternehmen und B2B-Applikationsintegration (B2BAI) über Unternehmensgrenzen hinweg als konkrete Ausprägungen von Integrationsarchitektur zu unternehmenskritischen Architektur-Disziplinen in der Informationstechnologie entwickelt (siehe Kapitel 3). Im Folgenden liegt der Fokus auf EAI.

EAI verfolgt das Ziel, die Anzahl der Schnittstellen zwischen IT-Systemen zu verringern und zu standardisieren sowie die mit Integration verbundene Komplexität zu reduzieren.

Dieses Anwendungsszenario widmet sich Projekten, die das Ziel haben, heterogene IT-Systeme innerhalb von Unternehmen (EAI) zu integrieren. Dabei wird beschrieben, welche Gesichtspunkte zu beachten sind und welche Anforderungen an eine geeignete Integrationsarchitektur (EAI-Architektur) gestellt werden. Der Leser erhält mit diesem Szenario eine Orientierungshilfe, die er nutzen kann, um ein besseres Verständnis von EAI zu erlangen. Dieses Szenario beschränkt sich auf busbasierte EAI-Ansätze, wie sie beispielsweise mittels entsprechender Integrationsprodukte von Herstellern, wie IBM, TIBCO, Vitria oder BEA, umgesetzt werden können (siehe Kapitel 6.7.1). Primär datenorientierte Ansätze, wie sie mit entsprechenden ETL-Werkzeugen verfolgt werden können, werden nicht näher betrachtet. ETL steht für Extract, Transform and Load. ETL-Werkzeuge werden eingesetzt, um Daten aus einer Quell-Datenbank zu extrahieren (englisch: *extract*), diese in ein anderes Format zu transformieren (englisch: *transform*) und in eine Ziel-Datenbank zu laden (englisch: *load*). Das Szenario skizziert ein architektonisches Vorgehen, wie es in typischen Integrationsprojekten angewendet wird.

EAI behandelt nicht nur IT-Gesichtspunkte, sondern berücksichtigt auch geschäftliche Aspekte. EAI erstreckt sich über alle Architektur-Ebenen (siehe Kapitel 4). Daraus ergibt sich der in Abbildung 8.8-1 dargestellte Kontext von EAI. Es ist wichtig, sich dieser verschiedenen Ebenen be-

Ziel

Abgrenzung

Kontext

wusst zu sein, um die Anforderungen an EAI zu verstehen und eine entsprechende Architektur abzuleiten. EAI sollte also immer aus unterschiedlichen Architektur-Perspektiven betrachtet werden. Der Schwerpunkt dieses Anwendungsszenarios liegt jedoch auf der System- und Bausteinebene.

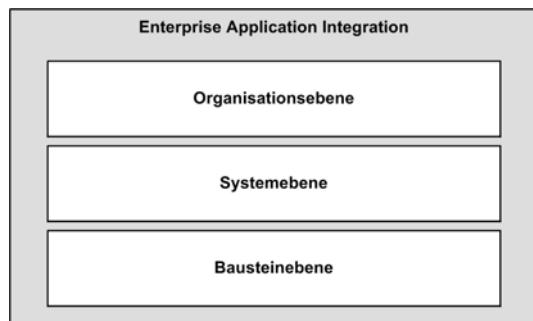


Abb. 8.8-1: Kontext von EAI.

8.8.1 Erstellen der Systemvision

EAI als Mittel zur Erreichung von Geschäftszielen

EAII erfolgt nicht zum Selbstzweck, sondern dient immer nur als Mittel, um geschäftliche Ziele zu erreichen. Einige der Gründe für EAII werden im Folgenden kurz vorgestellt. Diese, zusammen mit den im Anschluss geschilderten Vorteilen, entsprechen den Geschäftschancen im Sinne von Abschnitt 8.3.

Geschäftsstrategien und -prozesse als Grund für EAII

Customer Relationship und Supply Chain Management sind beispielsweise Schlüsselthemen, denen Unternehmen begegnen, um im Markt zu bestehen und erfolgreich zu sein. Hierzu ist eine ganzheitliche Sicht auf das Unternehmen notwendig. Dies bezieht sich sowohl auf Geschäftsprozesse als auch auf IT-Systeme, die die jeweiligen Geschäftsprozesse abbilden und unterstützen. Viele Systeme können jedoch als abteilungszentrisch charakterisiert werden. Dies bedeutet, dass sie für einzelne Abteilungen eines Unternehmens entwickelt wurden und die Anforderungen der jeweiligen Abteilung erfüllen. Zur Realisierung einer unternehmensweiten Geschäftsstrategie reicht dies jedoch nicht aus, da diese mehrere Abteilungen, wenn nicht sogar Unternehmen betreffen. Aus diesem Grund ist es nötig, IT-Systeme, die zu den übergreifenden Geschäftsprozessen beitragen, sinnvoll zu integrieren.

Kosteneinsparungen als Grund für EAII

Neben der Etablierung neuer Geschäftsstrategien können auch klare Kosteneinsparungsziele als Initiatoren für EAII genannt werden. Analys-

tenberichten zufolge wenden Unternehmen bis zu 40 % ihres IT-Budgets für die Integration von IT-Systemen auf. Dies liegt vor allem darin begründet, dass die klassische PzP-Integration mit zunehmender System-Anzahl im Vergleich zu EAI mit höheren Kosten verbunden ist.

Gerade im Rahmen von Fusionen besteht eine wichtige Aufgabe in der Konsolidierung der verschiedenen IT-Systemlandschaften. In diesem Zusammenhang kann EAI als Mittel der Integration einen wichtigen Beitrag leisten.

Im Vergleich zur PzP-Integration kann EAI folgende Vorteile bieten:

> *Standardisierte Integrationsmittel*

EAI etabliert standardisierte Integrationsmittel innerhalb eines Unternehmens oder sogar über Unternehmensgrenzen hinweg. Das Rückgrat einer EAI-Initiative bildet in aller Regel eine nachrichtenbasierte Middleware (siehe Abschnitt 6.7.1), an welche IT-Systeme angeschlossen werden. Diese bietet eine Vielzahl von Möglichkeiten, um IT-Systeme anzubinden. Durch standardisierte Integrationsmittel können weitere Systeme leichter angebunden werden.

> *Reduktion der Anzahl von Schnittstellen zwischen IT-Systemen*

Ein wesentlicher Vorteil von EAI ist die Reduktion vorhandener Schnittstellen. Dadurch reduziert sich die Komplexität der gesamten IT-Systemlandschaft. Hierdurch kann diese besser durchschaut, betrieben und erweitert werden.

> *Reduktion des Aufwands für den Unterhalt von IT-Systemen*

Aus der reduzierten Anzahl von Schnittstellen leitet sich auch ein geringerer Unterhaltsaufwand ab. Analysten sprechen hier von Einsparungen zwischen 25 und 70 % im Vergleich zur PzP-Integration.

> *Reduktion des Entwicklungsaufwands neuer IT-Systeme*

Durch EAI sinkt auch der Entwicklungsaufwand im Verhältnis zur PzP-Integration, da die Integration neuer IT-Systeme an Komplexität verliert. Hierbei finden sich Aussagen über Aufwandsreduktionen zwischen 20 und 40 %.

> *Unternehmensweite Verfügbarkeit von Informationen*

Durch EAI kann auf dieselben Informationen in derselben Qualität zur selben Zeit an verschiedenen Stellen in einem Unternehmen zugriffen werden. Im Rahmen von CRM ist es beispielsweise essenziell, dass jedes CRM-System über dieselben Kundenstammdaten verfügt. Dies kann durch EAI ermöglicht werden (siehe Kapitel 10).

Fusionen als Grund für EAI

Vorteile von EAI

- > *Schnellere Reaktionsfähigkeit auf sich ändernde Marktbedingungen*
Aufgrund der einfacheren Integration von IT-Systemen mittels EAI können Unternehmen auch schneller auf sich ändernde Marktbedingungen reagieren.
- > *Höherer Investitionsschutz*
Des Weiteren erhöht sich durch die Etablierung von EAI in Unternehmen der Investitionsschutz in die IT-Systemlandschaft. Dies liegt darin begründet, dass EAI auf standardisierter Middleware basiert. Diese kann zum einen für weitere Integrationsvorhaben genutzt werden. EAI rechnet sich somit mit jedem weiteren IT-System, welches an die Middleware angeschlossen wird. Zum anderen ist es für Unternehmen einfacher, qualifizierte Mitarbeiter auf dem Arbeitsmarkt zu finden, die über fundierte Erfahrung mit der Middleware verfügen. Dies steht im Gegensatz zu dem in aller Regel sehr proprietären Charakter existierender PzP-Verbindungen.

Nachteile und Risiken von EAI

Nach der Schilderung der Vorteile mag der Eindruck entstehen, dass mit EAI alle Integrationsprobleme in Unternehmen gelöst sind. Dies ist jedoch nicht der Fall. EAI ist kein Wundermittel. EAI kann zunächst einmal mit einem großen Aufwand und damit einhergehenden Kosten verbunden sein, da es eine große Herausforderung ist, IT-Systemlandschaften zu entflechten, Schnittstellen abzulösen und zu standardisieren sowie die eingesetzte Middleware zu verstehen. Ferner kann es notwendig sein, Änderungen an bestehenden IT-Systemen vornehmen zu müssen, um sie mittels EAI zu integrieren. Dies kann ein Risiko für den Betrieb der IT-Systeme bedeuten. Des Weiteren ist für den Erfolg von EAI-Projekten auch die breite Unterstützung des Managements wichtig, da Veränderungen auf Geschäftsprozessebene und etwaige Verschiebungen von Verantwortlichkeiten zu politischen Widerständen führen kann, die auf Managementebene abgebaut werden müssen.

EAI als Chance

EAI ist eine Chance für ein Unternehmen, wenn EAI als langfristige Investition betrachtet wird. Die Vorteile kommen erst dann richtig zum Tragen, wenn eine langfristige Strategie besteht, viele Systeme mittels EAI zu integrieren. Dann werden sich die Anfangsinvestitionen in eine EAI-Architektur voll auszahlen.

8.8.2 Verstehen der Anforderungen

Allgemeine architekturelle relevante Anforderungen an EAI

Das Verstehen der Anforderungen beschäftigt sich mit der Identifikation, Verfeinerung und Priorisierung von architekturelevanten Anforderungen (siehe Abschnitt 8.4). In diesem Abschnitt werden einige klassi-

sche architekturrelevante Anforderungen besprochen. Diese sind unabhängig davon, in welcher Industrie man tätig ist, welche Geschäftsstrategie realisiert und welche Geschäftsprozesse unterstützt werden sollen.

IT-Systemlandschaften bestehen aus verschiedenen Systemen, die auf unterschiedlichen Plattformen betrieben werden und in unterschiedlichen Programmiersprachen entwickelt wurden. Jede Plattform hat ihre eigenen Schnittstellen und Kommunikationsprotokolle. Im Sinne der PzP-Integration ergibt sich das in Abbildung 8.8-2 dargestellte Bild. Das Beispiel illustriert vereinfacht die im Rahmen einer Auftragsabwicklung involvierten IT-Systeme und ihre Beziehungen zueinander. Es wird ersichtlich, dass mit jedem hinzukommenden System die Schnittstellenkomplexität steigt.

Heterogene IT-Systemlandschaften mit vielfältigen Schnittstellen

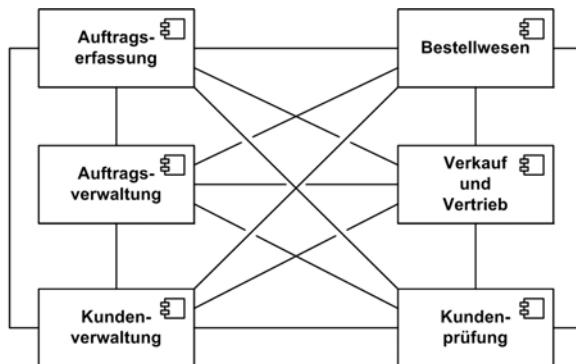


Abb. 8.8-2: Komplexe, auf Punkt-zu-Punkt basierende Systemlandschaft.

Eine Integrationsarchitektur muss diese unterschiedlichen Schnittstellen und Kommunikationsprotokolle unterstützen, indem sie diese abstrahiert und ineinander übersetzt. Darüber hinaus ist die Anzahl der Schnittstellen zu reduzieren, um die geschilderten EAI-Vorteile zu erreichen.

Neben existierenden IT-Systemen, oftmals als Legacy-Systeme bezeichnet, müssen auch häufig neue Paket-Lösungen, wie z. B. Siebel oder PeopleSoft im CRM-Bereich integriert werden. Dabei trifft man auf einen technischen Paradigmenwechsel zwischen Altsystemen, die beispielsweise in COBOL realisiert sind, und neuen Lösungen, die z. B. auf JEE beruhen. Integrationsarchitekturen müssen als Mittler zwischen diesen Paradigmen fungieren.

Integration existierender IT-Systeme und neuer Paket-Lösungen

Inkonsistente Datenmodelle und -konzepte

Wie bereits erwähnt, werden Systeme in der Regel auf die konkreten Bedürfnisse einer Abteilung zugeschnitten. Aus diesem Grund repräsentieren die zugrunde liegenden Datenmodelle und -konzepte die spezifische Sicht der jeweiligen Abteilung. Im Hinblick auf EAI ist dies ein großer Nachteil, da kein ganzheitliches, unternehmensweites Datenmodell existiert. Deshalb ist eine wichtige Aufgabe jedes EAI-Vorhabens die Konsolidierung, Transformation und Zuordnung von Daten.

Wechselnde Geschäftsanforderungen

EAI ist ein fortlaufendes Vorhaben, da sich mit der Zeit neue Geschäftsanforderungen entwickeln. Dies kann zu neuen Geschäftsprozessen und auf IT-Ebene zu neuen IT-Systemen führen, welche mit existierenden integriert werden müssen. Aufgrund dessen muss EAI diese Änderbarkeit und Erweiterbarkeit unterstützen.

Homogene und konsistente Geschäftsprozesse und -objektmodelle

EAI ist mehr als die reine Verbindung von IT-Systemen. Vielmehr sollten neben der Integration von IT-Systemen auch immer die Geschäftsprozesse und Geschäftsobjektmodelle auf ihre Homogenität und Konsistenz überprüft werden. Dadurch wird sichergestellt, dass der eigentliche Geschäftsnutzen durch die zu realisierende EAI-Lösung auch erreicht wird. Darauf hinaus sollte jede beteiligte Abteilung ein gleiches Verständnis der unternehmensweiten Geschäftsobjekte besitzen. Die Struktur eines Kunden oder eines Auftrags sollte sich beispielsweise in der Auftragserfassung nicht zu sehr von der Struktur im Verkauf und Vertrieb unterscheiden. Die Etablierung eines unternehmensweiten Geschäftsobjektmodells ist jedoch kein leichtes Unterfangen. Neben technischen Schwierigkeiten fehlt auch oft der politische Wille, um ein solches Modell zu realisieren, obwohl es die Chance für eine effizientere Geschäftsprozessabwicklung bietet. Der Aspekt der Geschäftsprozessanalyse und der Etablierung eines unternehmensweiten Geschäftsobjektmodells wird in diesem Szenario nicht näher behandelt.

Anforderungen an EAI zusammengefasst

Aus den gewonnenen Erkenntnissen ergeben sich zusammenfassend folgende Anforderungen an EAI:

- > Unterstützung unterschiedlicher Plattformen und Technologien.
- > Unterstützung unterschiedlicher Kommunikationsstile.
- > Unterstützung unterschiedlicher Datenformate und -strukturen.
- > Unterstützung des Informationsflusses zwischen Systemen.
- > Unterstützung von Geschäftsprozessen.

Die letzten beiden Punkte beziehen sich hierbei auf die Steuerung von Informationsflüssen und auf die Abbildung von Geschäftsprozessen durch EAI. Dabei soll EAI beispielsweise den Nachrichtentransport

(Routing) von einem Sender zum richtigen Empfänger gewährleisten. Ferner soll EAI auf Basis von übertragenen Informationen entscheiden, an welche angeschlossenen Systeme diese Informationen zu übermitteln sind. Man spricht in diesem Zusammenhang auch von inhaltsbasiertem Routing von Informationen [Keller 2002]. Im Sinne der Abbildung von Geschäftsprozessen soll es EAI ermöglichen, Geschäftsprozesse auf abstraktem Niveau zu modellieren und systemisch zu unterstützen. Die Abbildung der Systeminteraktionen durch einen Prozess hat den Vorteil, dass die Kommunikation zwischen Benutzern und den betroffenen Systemen nicht hart verdrahtet ist, sondern bei sich ändernden Anforderungen auf Prozessebene angepasst werden kann, ohne dass an der Implementierung Änderungen vorgenommen werden müssen.

Neben diesen spezifischen EAI-Anforderungen gelten beim Entwurf einer EAI-Architektur selbstverständlich auch die klassischen Architektur-Anforderungen, wie z. B. Bedienbarkeit, Wartbarkeit und Erweiterbarkeit (siehe Kapitel 5).

8.8.3 Entwerfen der Architektur

Dieser Abschnitt beschreibt die wesentlichen Tätigkeiten, die beim Entwurf einer EAI-Architektur durchgeführt werden müssen. Er orientiert sich dabei an Abschnitt 8.5.

In einem ersten Schritt sollte ein Systemkontext erstellt werden. Falls bereits während des Erstellens der Systemvision ein Systemkontext entworfen wurde, sollte er in diesem Schritt verfeinert werden. Normalerweise visualisiert ein Systemkontext die Abhängigkeiten zwischen dem zu realisierenden System und seiner Umwelt. Im Fall von EAI würde ein Systemkontext die Abhängigkeiten zwischen den zu integrierenden Systemen und der EAI-Infrastruktur aufzeigen. Es empfiehlt sich jedoch zunächst, nur die zu integrierenden Systeme zu betrachten und danach die EAI-Infrastruktur in den Kontext aufzunehmen.

Auf Basis der zu unterstützenden Geschäftsprozesse und -objektmodelle sind die Systeme zu identifizieren, die einzelne Prozessschritte unterstützen. Für das Auftragsabwicklungsbeispiel können die in Abbildung 8.8-3 dargestellten Systeme exemplarisch genannt werden. Die Auftragerfassung kommuniziert beispielsweise mit der Kundenprüfung, um die Bonität des Kunden festzustellen. Es empfiehlt sich, die Verantwortlichkeiten der einzelnen Systeme festzuhalten, um ihre Rolle im zu

Definiere Kontext

Identifikation der beteiligten Systeme

realisierenden IT-Systemverbund zu dokumentieren. Durch die Identifikation der beteiligten Systeme kann die grundlegende Frage „Welche Systeme sind miteinander zu verbinden?“ beantwortet werden.

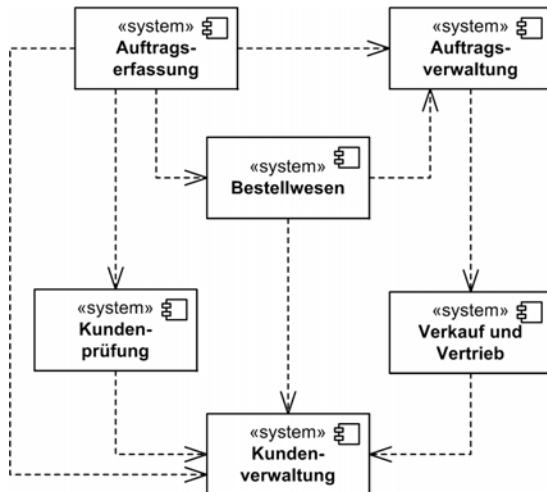


Abb. 8.8-3: Kontext-Diagramm zu integrierender Systeme.

Identifikation der Systemeigenschaften

Nachdem die Systeme identifiziert und dokumentiert wurden, müssen in einem nächsten Schritt die Eigenschaften der Systeme bestimmt werden. Die Eigenschaften ergeben sich aus den Architekturmitteln (z. B. Plattformen, Technologien etc.), die zur Realisierung der jeweiligen Systeme eingesetzt wurden (siehe Kapitel 6).

Plattformen und Technologien

Für jedes System sollte festgehalten werden, auf welcher Plattform es basiert und welche Technologien eingesetzt wurden. Das Spektrum der zu integrierenden Systeme kann sehr breit sein. Es kann von einer Host-Anwendung über die mittels IMS-Transaktionen zugegriffen wird, bis hin zu einem modernen Web-Service-orientierten System reichen. Aus diesem Spektrum lässt sich ableiten, welche Technologien sowie damit verbundenen Kommunikationsprotokolle und -formate von der EAI-Lösung unterstützt werden müssen.

Existierende Schnittstellen

Eng verbunden mit der Betrachtung der eingesetzten Plattformen und den verwendeten Technologien ist die Identifikation bereits existierender Schnittstellen. Dabei sind für jede Schnittstelle folgende Fragen zu beantworten:

Fragen im Rahmen der Schnittstellenanalyse

- > Welche Schnittstelle ermöglicht den Zugriff auf die benötigte Funktionalität?
- > Auf welcher Technologie basiert die Schnittstelle? Handelt es sich z. B. um ein Java-basiertes API oder eine DB-basierte Schnittstelle (beispielsweise Datenbank-Prozedur) und über welches Protokoll (z. B. SQL Net, TCP/IP, CORBA IIOP) wird zugriffen?
- > Wie sehen die über die Schnittstelle ausgetauschten Datenstrukturen aus? Welches Format und welche Semantik weisen sie auf?
- > Auf welchem Kommunikationsstil basiert die Schnittstelle? Erfolgt eine synchrone beziehungsweise asynchrone Kommunikation?

Es kann durchaus vorkommen, dass eine Schnittstelle nicht die vollständig benötigte Funktionalität bereitstellt. Aufgrund dessen ist es oftmals notwendig, ein System über mehrere Schnittstellen zu integrieren. Das fiktive Kundenprüfungssystem im Auftragsabwicklungsbeispiel verfügt über zwei getrennte Schnittstellen zur Prüfung der Bonität und der Adresse eines Kunden. Die Adressprüfungsschnittstelle ist dabei datenbankbasiert und ermöglicht die Prüfung von Adressen über den Aufruf einer Datenbank-Prozedur. Die Übergabe der Werte (Ort, Straße, Hausnummer) einer Adresse erfolgt über Parameter an die Datenbank-Prozedur. Die Rückgabe des Prüfungsergebnisses erfolgt synchron an den Aufrufer. Im Gegensatz hierzu basiert die Bonitätsprüfung auf einer asynchronen Kommunikation. Das aufrufende System stößt die Prüfung der Bonität eines Kunden durch das Absetzen einer HTTP-Anfrage an, bei der die Kundendaten in XML-Form übertragen werden. Das Kundenprüfungssystem teilt dem aufrufenden System mit, dass es den Prüfungsauftrag erhalten hat und initiiert die Bonitätsermittlung. Um das Ergebnis der Bonitätsprüfung zu erhalten, muss das aufrufende System nun in regelmäßigen Abständen ebenfalls über HTTP bei der Kundenprüfung nachfragen, ob die Bonitätsermittlung abgeschlossen ist. Liegt die Bonität vor, erhält das aufrufende System eine entsprechende XML-Nachricht als Antwort auf die HTTP-Anfrage. Die Adress- und die Bonitätsprüfung basieren somit sowohl auf unterschiedlichen Technologien, Datenstrukturen und -formaten als auch auf verschiedenen Kommunikationsstilen.

Neben der Analyse der existierenden Schnittstellen ist es auch wichtig, das Zusammenspiel der involvierten Systeme zu kennen. Daraus lassen sich die Informationsflüsse innerhalb des IT-Systemverbunds ableiten

Integration über mehrere Schnittstellen

Identifikation der Informationsflüsse

und die dabei anzusprechenden Schnittstellen bestimmen. Die Dokumentation kann mittels Sequenz-Diagrammen erfolgen.

Zu beantwortende Fragen

Folgende Fragen sollten beim Erarbeiten des Systemkontexts beantwortet werden:

Zu beantwortende Fragen:

- > Welche Systeme unterstützen die fachlichen Anforderungen?
- > Welches sind die Verantwortlichkeiten der Systeme?
- > Wie interagieren die Systeme miteinander?
- > Auf welcher Plattform und auf welchen Technologien beruhen die Systeme?
- > Über welche Schnittstellen verfügen die Systeme?
- > Welche Funktionalität bieten die Schnittstellen?
- > Welche Datenformate und -strukturen werden an den Schnittstellen erwartet?
- > Wie ist der Kommunikationsstil der einzelnen Schnittstellen charakterisiert?

Identifiziere Architektur-Mittel

Als Architekt steht man vor der Herausforderung, eine Architektur zu entwerfen, die die quadratische Komplexität von traditionellen PzP-Architekturen nicht aufweist (siehe Abbildung 8.8-2). Es gilt daher, eine architektonische Struktur zu wählen, die die Abhängigkeiten zwischen den zu integrierenden Systemen reduziert. Um eine solche Struktur zu entwerfen, sollte man sich geeigneter Architektur-Mitteln bedienen. Als architektonische Leitprinzipien können an dieser Stelle lose Kopplung und hohe Kohäsion genannt werden (siehe Abschnitt 6.1). In einer PzP-Architektur ist die Kopplung zwischen den IT-Systemen hoch, da tendenziell jedes System mit jedem anderem in Verbindung steht. Eine Integrationsarchitektur muss dagegen eine lose Kopplung ermöglichen, dass heißt, die Anzahl der Schnittstellen muss gering sein. Darüber hinaus sollte die Kohäsion der einzelnen Software-Bausteine hoch sein. Dies kann dadurch erreicht werden, dass jeder Software-Baustein eine dedizierte Verantwortlichkeit besitzt. Dies gilt zum einen für die fachlichen Anforderungen als auch für die nicht-funktionalen, im Sinne von EAI integrativen Anforderungen. Die fachlichen Anforderungen sind im eingeführten Beispiel klar auf die verschiedenen IT-Systeme verteilt. Die integrativen Anforderungen sollten ebenso klar auf entsprechende Software-Bausteine verteilt werden. Dies entspricht der Anwendung des Separation-of-Concerns- und Modularitäts-Prinzips. Für die Strukturierung der Integrationsarchitektur können Architektur-Stile und -Muster angewandt werden, die die genannten Prinzipien berücksichtigen.

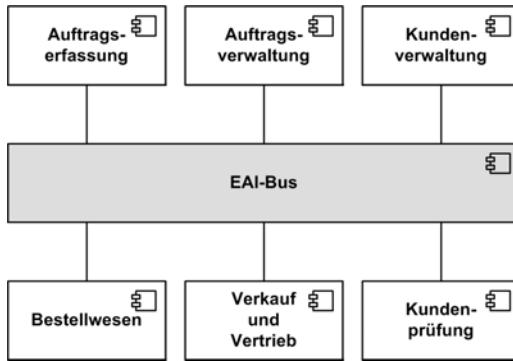


Abb. 8.8-4: Busbasierte EAI-Architektur.

Im Sinne der Modularisierung und des Separation of Concerns sollte aus diesem Grund zunächst ein grobgranularer Software-Baustein vorgesehen werden, welcher die integrativen Anforderungen realisiert. In Abbildung 8.8-4 übernimmt diese Aufgabe ein EAI-Bus. Die fachlichen IT-Systeme werden an den Bus angeschlossen und kommunizieren indirekt über den Bus miteinander. Dadurch wird eine Entkopplung der IT-Systeme und somit eine Reduktion der vorhandenen Schnittstellen erreicht. Die Komplexität aufgrund der Anzahl der Schnittstellen reduziert sich dabei von einer quadratischen auf eine lineare, da nur noch $O(n)$ statt im Extremfall $O(n^2)$ Schnittstellen benötigt werden.

Modularisierung und Separation of Concerns

Die gewählte architektonische Struktur aus Abbildung 8.8-4 basiert auf dem Component-Bus-Muster [Völter et al. 2002]. Eine Bus-Architektur sieht vor, dass die angeschlossenen Systeme nur über den Bus miteinander kommunizieren. Der Bus übernimmt dabei die Aufgabe der Übermittlung der Nachrichten zwischen den angeschlossenen IT-Systemen. Dies hat den Vorteil, dass sich die IT-Systeme nicht direkt kennen müssen. Die daraus resultierende Transparenz erlaubt es beispielsweise, mit geringerem Aufwand IT-Systeme auszutauschen oder neue hinzuzufügen, da die Systeme nur lose gekoppelt sind. Des Weiteren stellt der EAI-Bus sicher, dass die Nachrichten von den Systemen in der Form empfangen werden können, die sie benötigen. Die angeschlossenen Systeme müssen also nicht mehr das konkrete Format und die Technologie der Schnittstellen der anderen Systeme kennen. Die Aufgaben des EAI-Busses können durch verschiedene Middleware-Produkte realisiert werden (siehe Abschnitt 6.7.1).

Component-Bus-Muster als grundlegende Struktur

Nachdem die grundlegende EAI-Struktur gewählt wurde, kann der Systemkontext erweitert werden. In diesem Fall würde der Systemkontext um den EAI-Bus ergänzt.

Systemkontext um den EAI-Bus erweitern

EAI und Layering

Indem ein EAI-Bus in die Gesamtarchitektur integriert wird, entsteht ein dedizierter Baustein, der sich um die Aufgaben der Integration von Systemen kümmert. Der EAI-Bus kann als ein Subsystem der gesamten IT-Lösung betrachtet werden, welches auf der Integrationsschicht ange-siedelt ist. Dies entspricht der Anwendung des Layers-Architekturmusters [Buschmann et al. 1996]. Der EAI-Bus sollte weiter strukturiert werden, um eine erweiterbare Integrationsarchitektur zu erhalten. Zu diesem Zweck sollte man sich die allgemeinen EAI-Anforderungen (siehe Abschnitt 8.8.2) ins Gedächtnis rufen und die Anforderungen gemäß des Separation-of-Concerns-Prinzips unterschiedlichen Schichten zuordnen. In Abbildung 8.8-5 wird eine entsprechende Referenzarchitektur vorgestellt.

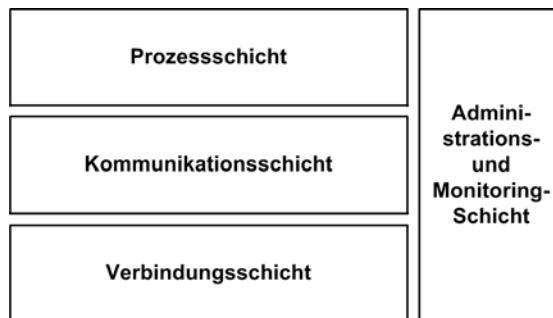


Abb. 8.8-5: Erweiterte EAI-Referenzarchitektur nach [Keller 2002].

Jeder Schicht der Architektur sind dedizierte EAI-Aufgaben zugeordnet, die im Folgenden kurz vorgestellt werden.

Verbindungsschicht

Die Verbindungsschicht hat die Aufgabe, die Anforderung nach der Unterstützung unterschiedlicher Plattformen und Technologien zu erfüllen. Aus diesem Grund enthält diese Schicht Funktionalität, um Verbindungen herzustellen. Integrationsprodukte bieten zu diesem Zweck beispielsweise Adapter zu verschiedenen ERP-Systemen, Datenbanken und Hostsystemen. Ein Adapter ist in der Lage, Nachrichten vom EAI-Bus an das angeschlossene IT-System mittels der hierfür notwendigen Technologie und dem passenden Kommunikationsprotokoll zu übermitteln. Darüber hinaus können Adapter Nachrichten des IT-Systems entgegennehmen und dem EAI-Bus zur Verfügung stellen. Dabei können sie die notwendigen Umwandlungen durchführen, um das Nachrichtenformat des angeschlossenen Systems in das des EAI-Busses zu transformieren.

Die Kommunikationsschicht kümmert sich um den Nachrichtentransport zwischen den angeschlossenen IT-Systemen. Sie kann sowohl synchrone als auch asynchrone Kommunikation unterstützen. Darüber hinaus hat sie die Aufgabe des Routings von Nachrichten. Sie muss also dafür Sorge tragen, dass zu einer Nachricht der richtige respektive die richtigen Empfänger gefunden werden. Um eine asynchrone Kommunikation zu gewährleisten, muss sie Nachrichten in Nachrichtenschlangen zwischenspeichern. Ferner können Datentransformationen und -formatierungen auch auf der Kommunikationsschicht durchgeführt werden. Sie müssen also nicht zwangsläufig nur auf der Verbindungsschicht erfolgen.

Kommunikations-schicht

Die Prozessschicht bietet Funktionalität zur Abbildung und Steuerung des Informationsflusses zwischen den partizipierenden Systemen auf Basis von Geschäftsprozessen. Hierbei kann ein Prozess Nachrichten versenden und auf eintreffende Nachrichten reagieren. Durch die Prozessschicht können die Interaktionen zwischen den Systemen, die zuvor identifiziert wurden, aus den betreffenden Systemen extrahiert und im EAI-Bus angesiedelt werden. Dies erhöht die Flexibilität, da Änderungen im Ablauf nicht mehr hart in der Programmlogik vorliegen, sondern konfigurativ angepasst werden können.

Prozessschicht

In komplexen Systemen ist die Nachvollziehbarkeit der über den EAI-Bus gesendeten Nachrichten essenziell, um bei eventuellen Unregelmäßigkeiten rechtzeitig reagieren zu können. Außerdem sollte es möglich sein, den Zustand des EAI-Systems zu überwachen und das System zu administrieren. Aus diesem Grund sollte bereits beim Entwurf der Integrationsarchitektur die Administrierbarkeit und Wartbarkeit des Systems berücksichtigt werden. Dies kommt in der Referenzarchitektur dahingehend zum Ausdruck, dass eine dedizierte Administrations- und Monitoring-Schicht vorgesehen ist, der entsprechende Funktionalität zugeordnet ist. Dies setzt natürlich voraus, dass die EAI-Bausteine der anderen Schichten auch so entworfen sind, dass sie sich administrieren und überwachen lassen. Bei der Wahl eines Produkts ist dies ein wichtiges Auswahlkriterium.

Administrations- und Monitoring-Schicht

Die Tabelle 8.8-1 ordnet die EAI-Anforderungen den Schichten der EAI-Referenzarchitektur zu.

Tab. 8.8-1: Zuordnung der EAI-Anforderungen zu den EAI-Schichten.

EAI-Schicht	EAI-Anforderung
Verbindungsschicht	<ul style="list-style-type: none"> > Unterstützung unterschiedlicher Plattformen und Technologien > Unterstützung unterschiedlicher Datenformate und -strukturen
Kommunikationsschicht	<ul style="list-style-type: none"> > Unterstützung unterschiedlicher Kommunikationsstile > Unterstützung des Informationsflusses zwischen Systemen
Prozessschicht	<ul style="list-style-type: none"> > Unterstützung von Geschäftsprozessen
Administrations- und Monitoring-Schicht	<p>allgemeine architektonische Anforderungen:</p> <ul style="list-style-type: none"> > Nachvollziehbarkeit > Bedienbarkeit > Wartbarkeit

Identifizierte Bausteine und definiere Verantwortlichkeiten

Für die einzelnen Schichten können Software-Bausteine identifiziert werden. Die Bausteine erhalten die Verantwortung, die schichtspezifischen EAI-Anforderungen zu realisieren. Auf eine konkrete Darstellung der Bausteine wird an dieser Stelle verzichtet.

Wahl der benötigten EAI-Dienste

Beim Entwurf der Integrationsarchitektur gilt es zu entscheiden, welche der genannten Dienste im Einzelnen überhaupt zur Erfüllung der konkreten, integrativen Anforderungen benötigt werden. Diese Frage kann mittels der dokumentierten Anforderungen im vorherigen Schritt beantwortet werden. Auf Basis der identifizierten Dienste kann auch ein entsprechendes EAI-Produkt beurteilt und ausgewählt werden.

Standardisierung der Schnittstellen

Ein wesentlicher Vorteil der Bus-Architektur ist die Verringerung der Schnittstellen. Dieser Vorteil lässt sich noch verstärken, indem die Schnittstellen hinsichtlich des Datenformats standardisiert werden. Es empfiehlt sich, zu diesem Zweck ein generisches Datenformat einzuführen, welches zur Kommunikation über den Bus genutzt wird [Vogel und Zdun 2002]. Dadurch sinkt die Anzahl notwendiger Transformationen, da nun für jede Verbindung mit einem System nur zwei Datenformate verwendet werden müssen. Zum einen das des adaptierten Systems und zum anderen das generische Format. Es ist vorteilhaft, das generische Datenformat in XML zu formulieren. Hierfür spricht unter anderem die mittlerweile recht große Verbreitung von XML auf unterschiedlichen Plattformen (siehe Abschnitt 6.7.3).

Benötigte Adapter und Integrationsarten identifizieren

Ein weiterer Aspekt, der auf architektonischem Niveau entschieden werden muss, ist die Art und Weise, wie die anzuschließenden Systeme in die EAI-Architektur integriert werden. Je nach Integrationsart sind für die Systeme unterschiedliche Adapter zu entwerfen. In EAI-Projekten kann man normalerweise folgende Integrationsarten unterscheiden [Linthicum 2001]:

> *Integration über eine Systemschnittstelle*

Bei dieser Integrationsart wird mit dem anzuschließenden System über seine hierfür vorgesehene Schnittstelle kommuniziert. Zu diesem Zweck programmiert man gegen das API des Systems. Dieser Ansatz hat den großen Vorteil, dass man gemäß dem Information-Hiding-Prinzip (siehe Abschnitt 6.1) nur die Aspekte des Systems kennen muss, die über das API zur Verfügung gestellt werden. Die internen Strukturen bleiben dem Verwender der Schnittstelle verborgen. Hierdurch können Änderungen im Innern des Systems leichter antizipiert werden, wenn sich die Schnittstelle nicht ändert. Nach Möglichkeit sollte ein System immer über dessen Schnittstelle respektive dessen API integriert werden. Verschiedene Integrationsprodukte verfügen zu diesem Zweck bereits über StandardadAPTER für Systeme, wie SAP oder Siebel.

> *Integration über eine Benutzerschnittstelle*

Falls das anzubindende System über keine Systemschnittstelle verfügt, bietet sich als Nächstes die Integration über die Benutzerschnittstelle an. Hierzu muss der Adapter die Aktionen eines Benutzers simulieren, um die gewünschten Aktionen im System auszulösen. Dies ist eine weitaus komplexere und fehleranfällige Integrationsart. Sie bietet jedoch trotzdem den Vorteil, dass man auch hier die interne Struktur und Charakteristik des Systems nicht kennen muss. Beispiele für Systeme, die über die Benutzerschnittstelle angeschlossen werden, sind alte Hostsysteme, die nur über ein Benutzerterminal angesprochen werden können. Bei dieser Art von Systemen ahmt der Adapter Benutzereingaben nach und interpretiert die daraus resultierenden Textausgaben. Diese Technik bezeichnet man auch als Screen Scraping. Sie wird nicht nur bei Hostsystemen angewendet, sondern kann auch zum Interpretieren von HTML-Seiten eingesetzt werden.

> *Integration über eine Datenschnittstelle*

Bei der Integration über die Datenschnittstelle greift man direkt auf das dem System zugrunde liegende Datenmodell zu, indem man die Datenbank des Systems anspricht. Diese Integrationsart sollte so weit wie möglich nur dazu verwendet werden, um lesend auf das System zuzugreifen. Sonst besteht die Gefahr, dass man die Integri-

tät des Systems verletzt. Selbst rein lesende Zugriffe sind problematisch, da man die interne Struktur des Systems kennen muss. Diese Struktur ist oftmals sehr komplex und es ist nicht auszuschließen, dass man die falschen Daten selektiert. Es kann beispielsweise vorkommen, dass ein System Kundendaten in verschiedenen Tabellen hält und man nicht entscheiden kann, welche der Tabellen für den Zugriff verwendet werden müssen. Für den Zugriff über eine Datenschnittstelle können ETL-Werkzeuge eingesetzt werden.

Minimal-invasive Integration

Eine Integration von Systemen sollte weitgehend minimal-invasiv sein. Mit anderen Worten sollten so wenige Änderungen wie möglich an dem bestehenden System vorgenommen werden müssen, um den Betrieb des Systems nicht zu gefährden. Sonst besteht das Risiko, dass wichtige Kernfunktionen eines Unternehmens nicht mehr zur Verfügung stehen.

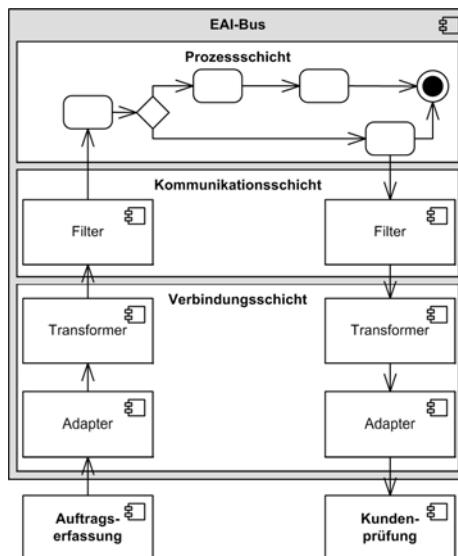


Abb. 8.8-6: EAI-Bus mit Pipes-and-Filters-Ansatz.

Einsatz von Architekturen-Stilen respektive -Mustern am Beispiel Pipes and Filters

Abbildung 8.8-6 konkretisiert die Architektur eines EAI-Busses und illustriert verschiedene benötigte Software-Bausteine. Dies sind nur beispielhafte Software-Bausteine, die in einer EAI-Architektur Verwendung finden können. Sie visualisieren jedoch, wie der Pipes-and-Filters-Architekturstil eingesetzt werden kann, um den Nachrichtenaustausch und die Verarbeitung zwischen den verbundenen Systemen zu ermöglichen (siehe Abschnitt 6.3.3). Filters sind Software-Bausteine, die Nachrichten verarbeiten. Pipes stellen dagegen die Verbindung zwischen

Datenquellen, Filtern und Datenzielen her. Im dargestellten Beispiel agieren die Adapter als Pipes, die die Nachrichten entgegennehmen. Im konkreten Fall empfängt der Adapter eine Nachricht von der Auftragserfassung und leitet diese an einen Transformer weiter. Dieser transformiert die empfangene Nachricht in das generische Datenformat. Es kann also durchaus vorkommen, dass es dedizierte Software-Bausteine gibt, die sich um die Transformation von Nachrichten kümmern. In diesem Fall hat der Adapter nur die Aufgaben des Entgegennehmens und des Versendens von Nachrichten. In genanntem Beispiel leitet der Transformer die umgewandelte Nachricht an einen Nachrichten-Filter weiter. Dieser entscheidet, ob die erhaltene Nachricht von Interesse ist und erzeugt ein entsprechendes Ereignis. Aufgrund dessen startet ein Prozess, der die Nachricht an den Nachrichten-Filter weiterleitet, der in die Verarbeitungskette zum Kundenprüfungssystem eingebettet ist. Dieser filtert beispielsweise Anfragen, die nicht vollständig sind, da die Kundenprüfung diese nicht verarbeiten kann. Gültige Anfragen übergibt der Filter an einen Transformer, der aus dem generischen Datenformat ein spezielles Datenformat für das Kundenprüfungssystem erzeugt. Im Anschluss ruft der Adapter das konkrete API des Kundenprüfungssystems auf. Es ist üblich, dass sich zwischen den einzelnen Software-Bausteinen Nachrichtenschlangen befinden, die dafür sorgen, dass die Nachricht von einem Software-Baustein zum nächsten gelangt. Dadurch wird sichergestellt, dass die Nachricht auch dann den Empfänger erreicht, wenn dieser zeitweise nicht zur Verfügung steht.

Der Pipes-and-Filters-Ansatz hat den Vorteil, dass auf einfache Art und Weise weitere Software-Bausteine integriert werden können. So ist es z. B. denkbar, einen weiteren Software-Baustein aufzunehmen, der einzelne Aufträge zu mehreren zusammenfasst und diese dann als Ganzes an die Auftragsverwaltung übergibt. Solch ein Software-Baustein übernimmt in einer EAI-Architektur die Aufgabe eines Aggregators.

Es ist sehr empfehlenswert, Erfahrungen und bewährte Lösungen aus verschiedenen EAI-Projekten für eigene EAI-Vorhaben zu nutzen. Für den Entwurf einer EAI-Architektur gibt es in der Literatur viele weitere Architektur- und Entwurfsmuster, deren Berücksichtigung zum Erfolg der Architektur beitragen kann. An dieser Stelle sei auf das Buch von Hohpe und Woolf verwiesen [Hohpe und Woolf 2003].

Ein oft genannter Nachteil beim Einsatz eines EAI-Produkts ist die Herstellerabhängigkeit aufgrund des proprietären Charakters der Middleware. Dies liegt vor allem daran, dass Standardadapter des Herstellers genutzt werden, mit denen die Systeme an den EAI-Bus angeschlossen

Vorteile des Pipes-and-Filters-Ansatzes

EAI-Muster

Abhängigkeit von eingesetztem EAI-Produkt

werden. Die Standardadapter stellen dabei jedoch nur die Verbindung zu einem System, wie SAP oder PeopleSoft, her. Die eigentliche Implementierung der benötigten Funktionsaufrufe und der Nachrichtenumwandlung innerhalb eines Adapters kann sehr aufwendig und somit kostenintensiv sein. Daher ist ein späterer Austausch des EAI-Produkts so gut wie ausgeschlossen, wenn sich die Investition rechnen soll. Um diese Problematik zu entschärfen, kann ein Enterprise Service Bus (siehe Abschnitt 6.4.11) entworfen werden. Ein ESB basiert auf einem EAI-Produkt. Allerdings erfolgt die Anbindung von Systemen über offene Standards. Viele Hersteller setzen dabei auf die Java Enterprise Edition mit ihrem Java Messaging Service und ihrer Web-Services-Unterstützung. Dies hat den Vorteil, dass die Adapter auf einem technologischen Standard und nicht auf einem herstellerspezifischen Format beruhen. Dies ist ein Schritt in die richtige Richtung. Allerdings reicht eine technologische Standardisierung noch nicht aus. Vielmehr ist es erstrebenswert, auch eine geschäftliche Standardisierung zu erreichen. Darunter ist zu verstehen, dass die fachliche Funktionalität, die von Paketlösungen bereitgestellt wird, ebenfalls standardisiert ist. In der Telekommunikationsindustrie arbeitet man beispielsweise an einer Standardisierung von Schnittstellen für TK-Produkte, die auf JEE basieren und definierte Methoden bereitstellen (siehe Abschnitt 6.5.5). Solch eine Standardisierung hätte für den Anwender den großen Vorteil, dass er für seine jeweilige Problemstellung das am besten geeignete Produkt auswählen kann und auf einfache Art und Weise mit anderen Produkten integrieren kann, die auf demselben Standard basieren.

Anbindung von Geschäftspartnern (B2BAI)

Die Integration von Geschäftspartnern im Rahmen von B2BAI sollte ebenso über technologische als auch geschäftliche Standards erfolgen. Um eine möglichst hohe Plattformunabhängigkeit zu erreichen, bieten sich hier der Einsatz von Web Services und die Verwendung von standardisierten XML-Vokabularen, wie RosettaNet, an (siehe Abschnitt 6.7.3).

Überprüfen der Architektur

Zum Überprüfen der Architektur sollte auf jeden Fall ein Architektur-Prototyp entwickelt werden. Dies gilt insbesondere zur Verifizierung der einzusetzenden Middleware. Der Architektur-Prototyp sollte dabei folgende architektonischen Aspekte verifizieren:

- > Benötigte EAI-Dienste (z. B. Verbindung, Transformation, Routing, Administration und Monitoring).
- > Verhalten der Middleware unter Last.
- > Einfache Abbildung wesentlicher Geschäftsprozesse und Systeminteraktionen.

- > Adapter-Entwicklung auf Basis der von der Middleware bereitgestellten Mittel.
- > Unterstützung der notwendigen Kommunikationsstile (synchron vs. asynchron).
- > Unterstützung der notwendigen Integrationsarten (Systemschnittstelle, Benutzerschnittstelle, Datenschnittstelle).
- > Unterstützung bei der Datentransformation.

Aufgrund der in einem Pilot-Projekt gewonnenen Erfahrungen kann über das weitere Vorgehen und über eine etwaige Ausweitung des EAI-Vorhabens auf andere anzuschließende Systeme entschieden werden.

8.8.4 Kommunizieren und Umsetzen der Architektur

Die Tätigkeiten Kommunizieren und Umsetzen der Architektur werden in diesem Szenario nur kurz behandelt, da das Hauptaugenmerk auf dem Verstehen der Anforderungen und dem Entwerfen der Architektur lag.

Bei der Kommunikation der Architektur ist es wichtig, die wesentlichen Architektur-Entscheidungen an Kunden, Systemverantwortliche und Teammitglieder zu kommunizieren. Zu den wesentlichen Architektur-Entscheidungen gehören:

- > die gewählte grundlegende EAI-Struktur.
- > die benötigten EAI-Dienste.
- > die benötigten Integrationsarten und Adapter.
- > die zu verwendenden Architektur- und Entwurfsmuster.
- > die Verantwortlichkeiten der IT-Systeme.

Diese Entscheidungen müssen zum einen von den Menschen verstanden und akzeptiert werden, die den EAI-Bus aufbauen. Zum anderen sind auch die Systemverantwortlichen der anzubindenden Systeme in das EAI-Vorhaben einzubinden. Sie sind die Experten der existierenden Systeme. Der Erfolg von EAI hängt sehr stark mit der Mitarbeit der Systemverantwortlichen zusammen. Dies gilt zum einen auf der technischen, fachlichen und zum anderen auf der politischen Ebene.

Die Umsetzung der Architektur erstreckt sich von der Definition eines generischen Datenformats über den Entwurf und die Umsetzung von Adapters bis hin zur Abbildung von Geschäftsprozessen auf der EAI-

Kurzer Überblick

Kommunizieren der Architektur

Systemverantwortliche involvieren

Umsetzen der Architektur

Evolution der Architektur

Plattform. Diese Tätigkeiten werden von unterschiedlichen Menschen in verschiedenen Rollen ausgeübt. Für den Architekten bedeutet dies, dass er ihnen das allgemeine Bild der Architektur und die Zusammenhänge erläutern und für die Bewahrung des Allgemeinbildes sorgen muss.

Eine Architektur kann einer Evolution unterliegen. Diese kann verschiedene Bereiche berühren. Ein Beispiel ist die Anbindung eines neuen Systems durch den Einsatz einer nach den vorgestellten Prinzipien und Mitteln strukturierten Architektur jedoch ohne größeren Aufwand möglich sein. Schwieriger gestaltet sich eine Änderung des generischen Datenformats, welches auf dem Bus gesprochen wird, da hiermit eine Änderung der Schnittstellenvereinbarung verbunden ist. Hiervon wären alle Adapter betroffen. Ebenso schwierig gestaltet sich ein möglicher Austausch der eingesetzten Middleware. Dies kann durch den Einsatz eines Enterprise-Service-Busses und offenen Standards zwar verringert werden, jedoch ist dies trotzdem kein leichtes Unterfangen. Ein Ersetzen des ESB ist zwar prinzipiell möglich, aber die im Bus integrierten Software-Bausteine, insbesondere das Workflow-Management, ist oft nicht ohne Weiteres zu ersetzen.

9 | Risikofallmanagementsystem

Diese Fallstudie bezieht sich vor allem auf die WARUM-Dimension und die WOMIT-Dimension des architektonischen Ordnungsrahmens.

Dabei wird die Entwicklung eines IT-Systems zur Risikoüberwachung für eine Bank vorgestellt. Verschiedene bereits isoliert bestehende IT-Systeme zur Risikoüberwachung sind dabei zu integrieren und die zugehörigen Geschäftsprozesse zu extrahieren. Eine entscheidende Anforderung ist darüber hinaus die leichte Erweiterbarkeit für weitere Geschäftsbereiche der Bank.

Der Leser soll durch diese Fallstudie insbesondere den Weg von den Anforderungen zur Strukturierung eines IT-Systems, die Einordnung eines konkreten Projekts in die Dimensionen des Ordnungsrahmens und den Einsatz modellbasierter Verfahren für die Umsetzung von Aspekten eines IT-Systems vertiefen.

Übersicht

9.1	Überblick	450
9.2	Architektur-Anforderungen (WARUM)	451
9.3	Architekturen und Architektur-Disziplinen (WAS)	460
9.4	Architektur-Perspektiven (WO)	462
9.5	Architektur-Mittel (WOMIT)	465
9.6	Organisationen und Individuen (WER)	470
9.7	Architektur-Vorgehen (WIE)	472

9.1 Überblick

RMS: Ein IT-System zur Risikoüberwachung

Eine europäische Großbank gibt die Entwicklung eines IT-Systems zur Risikoüberwachung in Auftrag, das verschiedene bereits isoliert bestehende IT-Systeme integrieren soll. Durch die Entwicklung eines Risikofallmanagementsystems (RMS) soll ein für die ganze Bank einheitliches Überwachungsinstrument geschaffen werden, das die Überwachung verschiedener Risikobereiche und die Abwicklung der damit verbundenen Risikofälle für alle Beteiligten erleichtert.

Standardisierter Bearbeitungsprozess für Risikobereiche

Über alle Risikobereiche hinweg sollen auf einheitliche Weise die Definitionen der nötigen Bearbeitungsprozesse möglich sein. Für jeden Risikobereich soll bei einem auftretenden Risikofall ein standardisierter Bearbeitungsprozess erzeugt werden. Das RMS soll alle weiteren nötigen Aktionen gemäß dem zugrunde liegenden Bearbeitungsprozess koordinieren und überwachen. Damit wird eine effiziente Überwachung der Risiken ermöglicht, die Bearbeitung jedes einzelnen Risikofalls gewährleistet und die Kontrolle und Übersicht über laufende und abgeschlossene Bearbeitungsprozesse sichergestellt.

Der zentrale Risikobereich Kredite

Als zentraler Risikobereich soll die Überwachung von Krediten integriert werden, für das bereits ein IT-System Kreditrisikodetektor existiert, das monatlich alle Kredite untersucht, dazu die Kreditrisikofälle auffindet und je einen standardisierten Kreditbericht erstellt.

Kurz- und langfristiges Business-Ziel

Die größten Einsparungen sollen durch Vereinheitlichung des Kreditfallbearbeitungsprozesses und einer damit verbundenen effizienteren Abwicklung der Risikofälle sowie einer Verminderung des Kreditrisikos erzielt werden. Eine weitere Kostensenkung soll vor allem auch die spätere Integration weiterer Risikobereiche ermöglichen.

Projektvolumen und -team

Das Projekt soll mit einem Team von weniger als acht Mitgliedern in zwei Jahren realisiert werden. Die Projektbudgetierung sieht ein Verhältnis von Personal zu Hardware- und Lizenzkosten von etwa eins zu drei vor. Das RMS wird zur Ausbreitung an ca. 1500 Mitarbeiter des Risikomanagements entwickelt.

Eingesetzte Architektur-Mittel

Im Projekt werden intensiv modellbasierte generierende und generische Verfahren eingesetzt.

9.2 Architektur-Anforderungen (WARUM)

Zunächst soll auf die WARUM-Dimension und dabei besonders auf die Motivatoren für das Projekt und die entscheidenden Anforderungen an das RMS eingegangen werden. Dabei werden diese gemäß Abschnitt 5.3 in funktionale, Entwicklungszeit- und Laufzeitanforderungen klassifiziert, was durch [F], [E] und [L] vermerkt werden soll.

Motivatoren und Anforderungen

9.2.1 Systemvision

Die involvierten Organisationseinheiten des Risikomanagements der Bank müssen eine Vielzahl von Berichten verschiedener Risikobereiche bearbeiten. Die Form, die Qualität, die Übermittlungswege dieser Informationen und deren Weiterverarbeitung sind sehr unterschiedlich. Dies führt zu einer wenig effizienten Überwachung und damit zu vermeidbaren Abwicklungs- und Risikokosten. Zur Verbesserung dieser Situation ist ein System zu entwickeln.

9.2.2 Organisationsanforderungen

Zunächst sollen die zentralen Geschäftsobjekte und Akteure, sowie deren Zusammenhänge dargestellt werden.

Die Mitarbeiter des Risikomanagements der Bank führen eine Risikoüberwachung für ein bankfachliches Risiko durch, das in einem relevanten Risikobereich vorliegt, zum Beispiel bei Krediten. In einem Risikobereich stellen sogenannte Risikofälle, die durch Berichte beschrieben werden, dieses Risiko dar. Für jeden Risikofall ist ein Mitarbeiter des Risikomanagements verantwortlich. Zur Risikoüberwachung setzen die Mitarbeiter ein bankfachliches Überwachungsinstrument ein, auf dem die Risikoüberwachung basiert. Diese wird ihrerseits durch einen Überwachungsprozess festgelegt, der die Definition eines Bearbeitungsprozesses für die Risikofälle beinhaltet. Solch ein Bearbeitungsprozess wird für jeden Risikofall initiiert und hängt vom Inhalt des jeweiligen Berichts ab. Die Geschäftsobjekte für den speziellen Risikobereich Kredite werden als kreditbezogene Spezialisierungen der beschriebenen Geschäftsobjekte aufgefasst.

Die zentralen Geschäftsobjekte und Akteure und deren Zusammenhänge

Das RMS stellt demzufolge die Realisierung des Kreditüberwachungsinstruments als IT-System dar. Die zentrale Organisationsanforderung

A-KRÜ [F]: Kreditrisikoüberwachung ermöglichen

an das RMS ist es, den Mitarbeitern des Risikomanagements zu ermöglichen, die Kreditrisikoüberwachung durchzuführen.

Aufbauorganisation im Risikobereich Kredite

In der Aufbauorganisation des Risikomanagements im Risikobereich Kredite sind die Mitarbeiter des Risikomanagements auf mehrere organisatorische Einheiten der Bank verteilt: Dies sind Kreditzentralen mit Unterabteilungen A und B für verschiedene Aufgaben sowie Beratergruppen von Kundenberatern.

Jeweils ein Kundenberater ist verantwortlich für einen Kreditrisikofall. Der Leiter einer Beratergruppe arbeitet zusätzlich auch als Kundenberater. Welche Mitarbeiter in die Kreditrisikoüberwachung für einen Kreditrisikofall involviert sind, kann durch festgelegte Regeln aus den Informationen zum verantwortlichen Kundenberater und aus der Organisationsstruktur abgeleitet werden. Die Teilnahme dieser verschiedenen Mitarbeiter an der Risikoüberwachung wird durch verschiedene Akteure modelliert, die am zentralen Anwendungsfall beteiligt sind.

9.2.3 Systemanforderungen

A-INT [F]: Anbindung IT-Systeme

Das RMS ist kein isoliertes System und muss sich in den Systemkontext der Bank integrieren. In Abbildung 9.2-1 sind die Akteure und die anderen IT-Systeme zu sehen, mit denen das RMS interagieren soll.

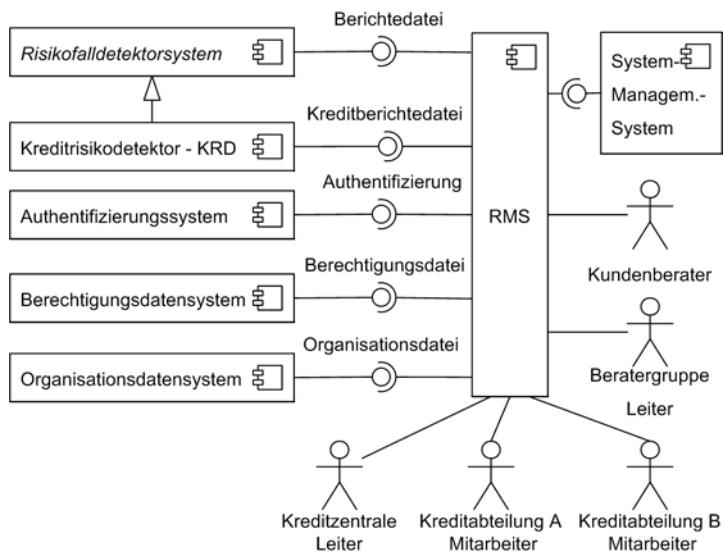


Abb. 9.2-1: RMS-Systemkontext.

Der Kreditrisikodetektor ist ein bereits bestehendes Host-System, das monatlich alle Kreditrisikofälle als Kreditberichtedateien zur Verfügung stellt.

Kreditrisikodetektor

Das Authentifizierungssystem ermöglicht es bankinternen web-basierten IT-Systemen, ohne eigene Benutzer- und Kennwortverwaltung auszukommen und somit die zentrale Benutzerverwaltung der Bank zu nutzen.

Authentifizierungs- system

Das Berechtigungsdatensystem verwaltet die Berechtigungen für alle IT-Systeme für sämtliche Mitarbeiter der Bank. Die entsprechenden Daten werden in einer Berechtigungsdatei mit täglicher Aktualisierung zur Verfügung gestellt. Die Bereitstellung dieser Datei erfolgt täglich bis spätestens 1:00 Uhr nachts.

Berechtigungsdaten- system

Im Organisationsdatensystem sind sämtliche verfügbaren Informationen über die Aufbauorganisation der Bank hinterlegt. Die Aktualisierung und Bereitstellung der Organisationsdatei, die diese Daten vollständig enthält, erfolgt völlig analog zur Berechtigungsdatei.

Organisationsdaten- system

Das System-Management-System wird in der Bank firmenweit verwendet, um das System-Management sämtlicher IT-Systeme zu ermöglichen.

System-Management- System

Die Anforderung zur Anbindung der aufgeführten IT-Systeme zerfällt in verschiedene einzelne Anforderungen:

A-AUT [F]: Zentrale Authentifizie- rung

Das RMS soll keine eigene Benutzerverwaltung mit eigener Kennwortverwaltung betreiben. Hier muss eine Anbindung an das bankeigene Authentifizierungssystem erfolgen.

A-BZV [F]: Abgeleitete Berechti- gungen, Zuständigkei- ten und Verantwor- tlichkeiten

Die Berechtigungs- und die Organisationsdatei muss einmal täglich aktuell per FTPS, einer sicheren, bankeigenen Variante des FTP-Protokolls, importiert werden. In welchen durch die Akteure spezifizierten Rollen ein Benutzer mit dem RMS arbeiten kann, muss aus der Berechtigungsdatei und der Organisationsdatei automatisch abgeleitet werden. Ebenso müssen sämtliche Zuständigkeiten und Verantwortlichkeiten bezogen auf die Kreditrisikofälle aus diesen Dateien abgeleitet werden.

A-KRD [F]: Import der Kreditrisiko- berichte vom Kredit- risikodetektor

Die monatlich erstellten Kreditberichte werden direkt nach der Bereitstellung durch den Kreditrisikodetektor als Datei per FTPS importiert und stehen den Mitarbeitern des Risikomanagements am nächsten Arbeitstag zur Verfügung. Das Format der Kreditberichtedatei ist durch

den Kreditrisikodetektor festgelegt, der allerdings zeitgleich zum RMS reimplementiert wird, sodass das endgültige Format erst in einer späteren Projektphase feststehen wird.

**A-PRI [F]:
Automatische Instan-
ziierung der Bearbei-
tungsprozesse**

**A-SYS [F]:
Anbindung System-
Management-System**

**A-ERW [E]:
Einfache Erweiterbar-
keit für Risikobereiche**

**A-TRS [L]:
Transaktionssicherheit
IT-Systeme**

**A-STD [E]:
Einhaltung der
IT-Standards**

**A-VFÜ [L]:
Verfügbarkeit**

Anwendungsfälle

Für jeden Kreditbericht muss automatisch eine Instanz des Kreditfallbearbeitungsprozesses erzeugt werden. Diese hat ebenfalls bis zum nächsten Arbeitstag den jeweiligen Mitarbeitern des Risikomanagements zur Weiterbearbeitung zur Verfügung zu stehen.

Das System-Management-System muss über die vorgegebene Schnittstelle angebunden werden. Fehler, die das RMS während des Betriebs feststellt, müssen an dieses IT-System automatisch übermittelt werden. Zudem werden alle Daten des RMS durch das System-Management-System der Bank einmal nächtlich gesichert. Dafür muss das RMS während eines Zeitfensters von 1 Uhr bis 3 Uhr heruntergefahren sein.

Bei Integration von Risikofalldetektorsystemen für weitere Risikobereiche und die zugehörigen Überwachungs- und Bearbeitungsprozesse soll der Aufwand für Entwicklung und Einführung gegenüber der Entwicklung für das Kreditrisiko halbiert werden. Es werden auch in diesen Risikobereichen Anforderungen analog zu den Anforderungen A-KRD und A-PRI bestehen.

Transaktionale Sicherheit soll während der gesamten Interaktion mit den anderen IT-Systemen gewährleistet sein. Wenn also die Übertragung oder Verarbeitung einer Datei unterbrochen wird, darf es nicht etwa zu inkonsistenten Zuständen im RMS kommen. Es soll möglich sein und automatisch versucht werden, Interaktionen so lange zu wiederholen, bis sie erfolgreich sind. Bis dahin kann im letzten gültigen Zustand weitergearbeitet werden.

Insgesamt wird gefordert, sämtliche Standardvorgaben bezüglich der eingesetzten Technologien und die Styleguides im IT-Bereich der Bank einzuhalten.

Das RMS soll den Mitarbeitern des Risikomanagements täglich von 8 Uhr bis 18 Uhr zur Verfügung stehen.

Die geforderten Funktionen für die Mitarbeiter des Risikomanagements der Bank lassen sich als Anwendungsfalldiagramm wie folgt charakterisieren.

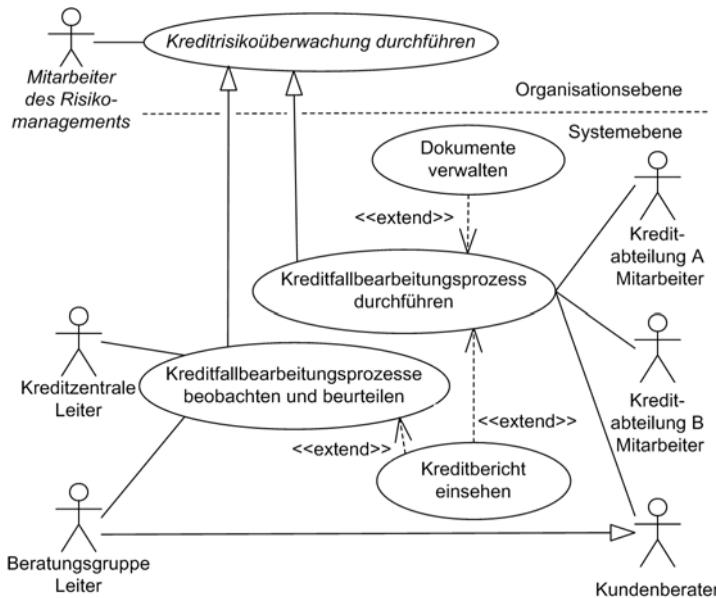


Abb. 9.2-2: Anwendungsfalldiagramm Kreditrisikoüberwachung.

Der Kreditfallbearbeitungsprozess, d. h. die Ablauforganisation für die Bearbeitung der Kreditberichte, ist wie folgt näher spezifiziert:

Für jeden Kreditrisikofall soll eine Prozessinstanz erstellt und die dafür definierten Aktionen zur Bearbeitung ermöglicht werden. Hierbei sind der verantwortliche Kundenberater und die zuständigen Mitarbeiter und Leiter aus der Kreditzentrale und den Kreditabteilungen beteiligt. Der betreffende Kreditbericht selbst und der Verlauf der Instanz des Kreditfallbearbeitungsprozesses sollen dabei archiviert werden.

Während der gesamten Abarbeitung einer Instanz des Kreditfallbearbeitungsprozesses wird Transaktionssicherheit bezüglich der Aktionen gefordert.

Die meisten Benutzer werden mit Aktionen des Kreditfallbearbeitungsprozesses beschäftigt sein. Deshalb ist es von besonderer Bedeutung, dass der Schulungsaufwand hierfür sehr gering ist.

Während der Abarbeitung, Beobachtung oder Beurteilung eines Kreditfallbearbeitungsprozesses soll jederzeit der ursprüngliche Bericht zum Risikofall einsehbar sein.

**A-PRB [F]:
Aktionen des Kreditfallbearbeitungsprozesses ermöglichen**

**A-TRP [L]:
Transaktionssicherheit für Prozessaktionen**

**A-SCH [L]:
Geringer Schulungsaufwand**

**A-BER [F]:
Berichtsansicht**

**A-DOK [F]:
Dokumente verwalten**

Während des Bearbeitungsprozesses für einen Risikofall sollen dem Bericht sowohl unstrukturierte als auch stark strukturierte Dokumente hinzugefügt und verwaltet werden können.

**A-STA [F]:
Status- und Archivansicht**

Die involvierten Akteure, also die Leiter der Beratergruppen und die Leiter der Kreditzentralen, sollen jeweils eine Liste der Kreditrisikofälle sehen, für die sie zuständig sind. Für Risikofälle, die noch in Bearbeitung sind, soll deren Bearbeitungszustand ersichtlich sein, für archivierte, der Verlauf des Bearbeitungsprozesses. Für die Liste der Risikofälle sollen Such- und Auswertemöglichkeiten bestehen.

**A-AUS [E]:
Einfache Ausbreitung**

Wegen der hohen Zahl und der örtlichen Verteilung der beteiligten Mitarbeiter des Risikomanagements wird ein einfacher und damit kostengünstiger Ausbreitungsprozess für das RMS und für neue oder geänderte Versionen des Kreditfallbearbeitungsprozesses benötigt.

**A-PRD [E]:
Einfach erstellbare
Prozessdefinition**

Die Erstellung und Änderungen der Definition des Kreditfallbearbeitungsprozesses soll versioniert möglich sein, wenig IT-Wissen benötigen und im Idealfall von einem Bankfachmann durchgeführt werden können.

**A-PER [L]:
Persistenz der Berichte,
Prozesse und Dokumente**

Die Anforderungen A-TRP, A-BER, A-STA, A-DOK und A-PRD bedingen sofort ihrerseits eine weitere Anforderung: Die Berichte, die Prozessdefinitionen und deren Instanzen sowie die Dokumente müssen persistent gehalten werden.

**Strukturierung und
Entscheidungen**

Es sollen nun bezüglich der Strukturierung des RMS die relevanten Systembausteine vorgestellt werden, um die zuvor formulierten Anforderungen auf die Bausteinebene herunterzubrechen und formulieren zu können. Um zu dieser Strukturierung zu gelangen und damit den Anforderungen der Organisations- und Systemebenen adäquat zu begegnen, sind natürlich bereits Entscheidungen verschiedener Architektur-Disziplinen zu treffen, Architektur-Mittel einzusetzen sowie Methoden und Vorgehen anzuwenden. Darauf wird allerdings erst etwas später, in den Abschnitten zu den entsprechenden Dimensionen, genauer eingegangen.

**E-REFA:
Einsatz der bankeigenen
Web-Referenzarchitektur**

Die Bank bietet bereits eine Referenzarchitektur für die Integration des bankeigenen Authentifizierungssystems für Web-Anwendungen an. Hierzu ist die in der Bank standardisierte Web-Server-Technologie vorgeschrieben, was hier durch Verwendung der Systembausteine Standard-Web-Client und Standard-Web-Server zum Ausdruck gebracht

wird. Damit ergeben sich die integrierenden RMS-Systembausteine RMS-Web-Client und RMS-Web-Server. Durch diese Architektur-Entscheidung kann so die Anforderung nach Integration des Authentifizierungssystems (A-AUT) und gleichzeitig die Anforderung der einfachen und kostengünstigen Ausbreitung an die Benutzer (A-AUS) gewährleistet werden. Zudem wird in dieser Struktureinheit die Standardisierungsanforderung erfüllt (A-STD).

Zur Verwaltung der Dokumente (A-DOK), der einfachen Erstellung der Definition von Bearbeitungsprozessen (A-PRD), der geforderten Status- und Archivansichten (A-STA) sowie der Transaktionssicherheit für die Prozessaktionen (A-TRP) wird die Architektur-Entscheidung getroffen, ein in der Bank zum Standard erklärtes IT-System einzusetzen. Es handelt sich um ein web-basiertes Document-Management-System (WDMS) mit grundlegender Workflow-Management-Funktionalität, das im Folgenden durch die Systembausteine Std-WDMS und Std-WDMS-Archiv repräsentiert wird. Dadurch können die Anforderungen A-PRD und A-PER zum größten Teil schon erfüllt werden. Es bleiben nur noch die Berichte. Diese sollen explizit nicht als Dokumente gespeichert werden, sondern separat und stark strukturiert. Dies verbessert den für später angedachten Austausch der WDMS-Systembausteine.

Sowohl das Organisations- und als auch das Berechtigungsdatensystem stehen ebenfalls bereits zur Verfügung. Wenn Daten aus diesen Bereichen benötigt werden, besteht nach Anforderung A-BZV die Verpflichtung, diese nicht redundant zu pflegen, sondern von den verfügbaren IT-Systemen zu beschaffen und davon abzuleiten.

Um das RMS jedoch von Änderungen der zugehörigen Dateiformate abzukapseln (siehe die beiden Interfaces Berechtigungsdatei und Organisationsdatei), werden diese jeweils in ein durch das RMS vorzugebendes, möglichst universelles und standardisiertes Format gebracht. Dazu wird ein eigener Systembaustein Org-Ber-Konverter zur Konvertierung der Dateien in XML-Dateien entwickelt. Dies wird durch die Interfaces Org-XML und Ber-XML ausgedrückt (siehe Abbildung 9.2-3). Der zentrale Systembaustein RMS-Kern wird damit so weit als möglich technologisch von den Zuliefersystemen unabhängig.

Dies bedeutet, dass die Schnittstellen Org-XML und Ber-XML zu entwerfen, in einem Systembaustein RMS-Kern zu realisieren und von den Konvertern zu benutzen sind. Auf die innere Struktur von RMS-Kern wird später noch genauer eingegangen.

E-WDMS:
Verwendung der Systembausteine
Std-WDMS mit Std-WDMS-Archiv

E-KONV:
OrgBer-Konverter und XML-Schnittstellen
Org-XML und Ber-XML

A-OBX [E]:
Dateiimporte via
Org-XML und Ber-XML

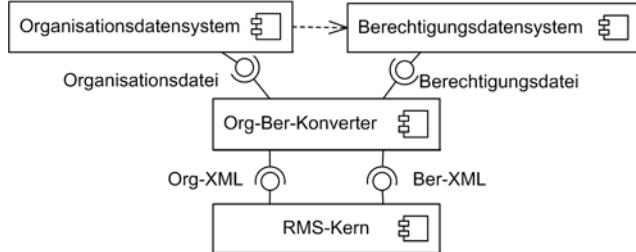


Abb. 9.2-3: Integration der Organisations- und Berechtigungsdatensysteme.

**A-OBV [L]:
Organisations- und
Berechtigungsdaten-
Verarbeitungsdauer**

Aus der Anforderung für die Dateiimporte und deren Aufbereitung (A-BZV) und der Verfügbarkeitsanforderung (A-VFÜ) folgt, dass die Verarbeitung und Aufbereitung der Dateien durch den Org-Ber-Konverter und den RMS-Kern zwischen 3 Uhr und 8 Uhr erfolgen muss.

**E-META:
Erweiterbarkeit durch
Berichte-Metamodell**

Aufgrund der Anforderung zur einfachen Erweiterbarkeit für neue Risikobereiche (Anforderung A-ERW) wird die weitreichende Entscheidung getroffen, die Berichte als zentralen Aspekt (siehe Abschnitt 6.2.6) im gesamten RMS zu betrachten. Dieser Aspekt wird durch ein Berichte-Metamodell technisch berücksichtigt. Hierzu wird ein spezielles Entwicklungswerkzeug (RMS-Gen) mit einem Editor für die Berichte-Modelle und einem Generator für die betroffenen RMS-Systembausteine entwickelt. Dies soll den Aspekt Berichte effizient und konsistent in die einzelnen RMS-Systembausteine einweben.

**A-BMM [E]:
Aspekt Berichte basiert
ausschließlich auf dem
Berichte-Metamodell**

Als Anforderung folgt hieraus, dass alle RMS-Systembausteine, die den Aspekt Berichte besitzen, auf dem Berichte-Metamodell basierend operieren. Für den jeweiligen Risikobereich sind die Systembausteine damit sozusagen über die Berichte-Modelle der Risikobereiche, also den Instanzen des Berichte-Metamodells, parametrisiert.

**E-GENX:
Berichte-Konverter mit
generischer Schnittstel-
le Berichte-XML basie-
rend auf Berichte-
Metamodell**

Von den Risikofalldetektorsystemen, insbesondere vom Kreditrisikodetektor, sind laut Anforderung A-KRD periodisch Risikoberichte zu importieren. Als Technologie an der Schnittstelle Berichte-XML soll wie bei den Schnittstellen Organisations- und Berechtigungsdatei XML verwendet werden. Das architektonische Konzept der generativen Erzeugung von Systembausteinen (siehe Abschnitt 6.2.5) wird hierbei, unterstützt durch ein extra zu entwickelndes Werkzeug RMS-Gen, basierend auf dem oben erwähnten Berichte-Metamodell eingesetzt. Wie damit die generalisierte Schnittstelle Berichte-XML genau umgesetzt ist, wird im Abschnitt zur Dimension Womit genauer diskutiert und erläutert.

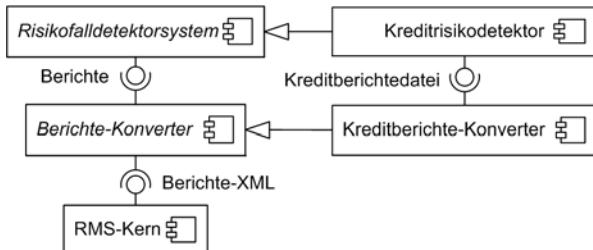


Abb. 9.2-4: Anbindung der Risikofalldetektorsysteme.

Damit wird durch Berichte-XML ein Teil der Anforderung A-ERW umgesetzt und die Realisierung und Benutzung der Schnittstelle selbst zu einer Bausteinanforderung für die Systembausteine Kreditberichte-Konverter und RMS-Kern.

Der abstrakte Systembaustein Berichte-Konverter versammelt dabei Funktionalität zur Validierung und Verifikation der XML-Dateien, die über die Schnittstelle Berichte-XML ausgetauscht werden. Dies ermöglicht es, die Berichte-Konverter, wie den Kreditberichte-Konverter, extern oder vom Kunden selbst entwickeln zu lassen und oft anzupassen ohne die Interna von RMS dabei anzutasten. Damit ist der Berichte-Konverter selbst vom RMS-Kern optimal entkoppelt. Gleichzeitig aber entkoppelt er das RMS zum einen vom Kreditrisikodetektor, der selbst noch unter laufender Entwicklung steht, und zum anderen von zukünftigen Risikodetektorsystemen weiterer Risikobereiche.

Aus der Anforderung für den Import der Kreditrisikoberichte (A-KRD) und der automatischen Instanziierung der Bearbeitungsprozesse (A-PRI), zusammen mit der Verfügbarkeitsanforderung (A-VFÜ) folgt, dass die Verarbeitung und Aufbereitung der Dateien durch den Berichte-Konverter und den RMS-Kern ebenfalls zwischen 3 Uhr und 8 Uhr erfolgen muss.

Für das Berichte-Archiv wird entschieden, dieses, ebenfalls auf dem oben bereits genannten Berichte-Metamodell basierend, als generisches Datenbankschema in der Berichtetedatenbank umzusetzen (siehe Abschnitt 6.2.4). Somit wird es ermöglicht, für einen neuen Risikobereich ein entsprechendes Berichte-Modell zu erstellen und die Instanzen dazu, d. h. die Berichte des Risikobereichs, ohne Änderung des Datenbankschemas zu speichern und wieder auf diese zuzugreifen. Dies setzt einen weiteren Teil der Anforderung A-ERW um. Zudem ist damit die Anforderung A-PER in Bezug auf die Persistenz für Berichte erfüllt.

A-GBI [E]:
Berichte-Import via generischer Schnittstelle Berichte-XML

A-KBV [L]:
Kreditberichte-Verarbeitungsdauer

E-GENDB:
Berichte-Archiv als relationale Datenbank mit Berichtezugriff, beides basierend auf Berichte-Metamodell durch Metaprogrammierung

**A-GBA [E]:
Generischer Berichte-
zugriff**

Auch für die Systembausteine RMS-Kern und RMS-WD bedeutet dies, dass diese im Hinblick auf Zugriffe auf das RMS-Berichte-Archiv ebenfalls generisch bezüglich der Berichte-Modelle sein müssen.

**E-SYSM :
Verwendung des
Sys-Mgmt-Client**

Schließlich muss nun noch das System-Management-System angebunden werden (Anforderung A-SYS). Hierzu bietet die Bank wieder eine Teillösung zur Verwendung an: den Systembaustein Sys-Mgmt-Client.

**A-SYC [E]:
Schnittstelle Sysmgmt**

Hieraus leitet sich aber die neue Bausteinanforderung ab, dass der RMS-Kern für diese Anbindung die Schnittstelle Sysmgmt des Sys-Mgmt-Clients benutzt. Wie zu sehen ist, übernimmt RMS-Kern auch Integrationsaufgaben zwischen den Systembausteinen des RMS.

9.3 Architekturen und Architektur-Disziplinen (WAS)

9.3.1 Disziplinen

Integrationsarchitektur

Architektur-Aufgaben ergeben sich bezüglich einer angemessenen Integrationsarchitektur, die durch die Anforderung zur Anbindung der verschiedenen IT-Systeme (siehe Anforderung A-INT) festgehalten sind.

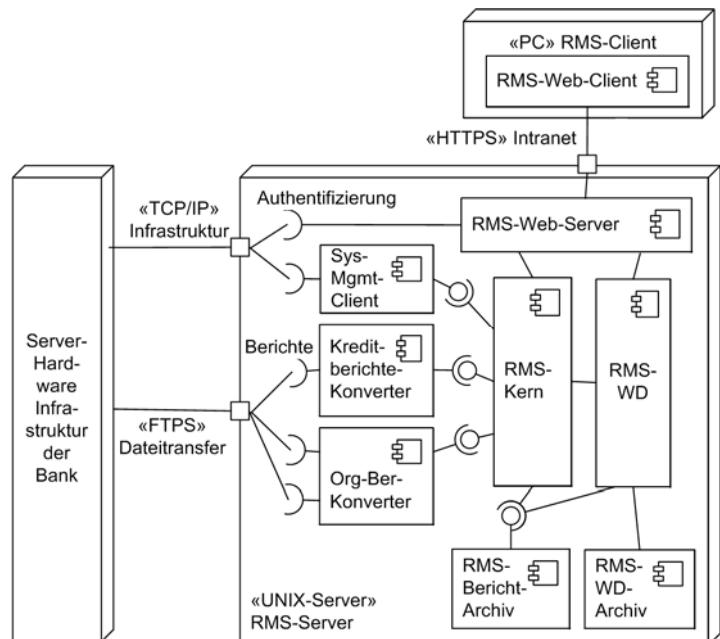


Abb. 9.3-1: RMS-Netzwerkarchitektur.

Die in Abbildung 9.3-1 dargestellte Netzwerkarchitektur wird weitgehend durch den Einsatz der bankeigenen Web-Referenzarchitektur und damit der Realisierung einer HTTPS-basierten Web-Anwendung festgelegt. Wegen der Verwendung der bankeigenen Systembausteine Std-Web-Server und Sys-Mgmt-Client ist die Kommunikation TCP/IP-basiert und durch die Infrastruktur bereits umgesetzt. Es bleibt der durch die Anforderungen A-BZV und A-KRD verlangte FTPS-Dateittransfer als Kommunikationsprotokoll.

Netzwerkarchitektur

Die Datenarchitektur ist einerseits durch den Einsatz des Std-WDMS-Archivs und die Anforderungen A-STD und A-SDB bezüglich der in der Bank vorgeschriebenen Datenbanksysteme für das Berichte-Archiv abgeleitet.

Datenarchitektur

Die System-Management-Architektur ist ebenfalls bereits durch die Anbindung des System-Management-System durch die Verwendung des Sys-Mgmt-Clients festgelegt (A-SYC).

System-Management-Architektur

Bezüglich der Sicherheitsarchitektur wurde durch die Integration des bestehenden Authentifizierungssystems, der Verwendung der bankeigenen sicheren FTP-Variante FTPS und durch die HTTPS-basierte Web-Referenzarchitektur Genüge geleistet.

Sicherheitsarchitektur

Die entscheidenden Herausforderungen stellen sich in der Disziplin Software-Architektur. Wo und womit diesen Herausforderungen begegnet wird, folgt ausführlich in den nächsten Abschnitten.

Software-Architektur

9.3.2 Entscheidungen zur Software-Architektur

Bereits in Abschnitt 9.2 wurden wichtige Entscheidungen zur Software-Architektur (siehe Abschnitt 7.5) getroffen. Nun sollen weitere Entscheidungen vorgenommen werden, die nicht die Systembaustein-Struktur des RMS betreffen.

Weitere Entscheidungen

Die Entscheidung, das Berichte-Metamodell als einen zentralen Aspekt Berichte (siehe Abschnitt 6.2.4) von RMS zu betrachten, führt konsequenterweise dazu, auch die Berichtsansicht (Anforderung A-BER) basierend auf dem Berichte-Metamodell zu realisieren. Dies führt zu einer weiteren Teilumsetzung der Erweiterbarkeit (Anforderung A-ERW).

E-BERA: Berichtsansicht basierend auf Berichte-Metamodell

Bei der Frage nach der Programmiersprache für die völlig neu zu entwickelnden Systembausteine des RMS, d. h. für die Konverter und den

E-JAVA: Einsatz von Java

E-RELDB:
Wahl des relationalen Standard-Datenbanksystems

RMS-Kern, kommt wegen den bankeigenen Standards (Anforderung A-STD) als objektorientierte Sprache (siehe Abschnitt 6.2.2) nur Java in Frage.

E-GENW:
Entwicklung eines Generatorwerkzeugs RMS-Gen

Bei der Wahl des Datenbanksystems für die Umsetzung des generischen Berichte-Archivs steht wegen Anforderung A-STD ebenfalls das relationale Standard-Datenbanksystem der Bank als zu verwendendes System fest. Das eingesetzte Std-WMDS-Archiv erfüllt bereits diese Anforderungen. Für die darüber hinausgehende Funktionalität ist dies also auch eine Anforderung an das RMS-WD-Archiv.

Da der Aspekt Berichte sehr stark in den verschiedenen Schnittstellen und Systembausteine vertreten ist, sollen die betroffenen Artefakte so weit als möglich automatisch erzeugt werden. Dazu wird ein eigenes RMS-spezifisches Werkzeug RMS-Gen entwickelt (siehe Abschnitt 9.5.3).

Im Folgenden wird nun näher darauf eingegangen, wo die beschriebenen Entscheidungen in der RMS-Architektur ihren Niederschlag finden.

9.4 Architektur-Perspektiven (WO)

Architektur-Ebenen

Nachdem bereits genau untersucht ist, warum das RMS entwickelt wird und was die Anforderungen auf den verschiedenen Ebenen sind, soll nun auf die innere Struktur der Systembausteine genauer eingegangen werden. In Abschnitt 9.2.3 wurde das RMS bereits in Systembausteine strukturiert. Nun soll diese Strukturierung anhand der Granularität der entsprechenden Architektur-Ebenen dargestellt werden. Die Ebene Organisation spielt allerdings hier keine Rolle.

9.4.1 Systemebene

E-PHYS:
Struktur der physikalischen Systembausteine

Das RMS wurde bereits in Abbildung 9.2-1 als Black Box dargestellt. Seine Interaktionspartner sind als IT-Systeme und Akteure modelliert. In Abschnitt 9.2 zur WARUM-Dimension wurde das RMS in Systembausteine auf der Systemebene strukturiert. Die Systembausteine dort besitzen zur Laufzeit jeweils eine direkte Entsprechung, die als Instanz zur Ausführung in die Laufzeitumgebung geladen wird. Solche Systembausteine sollen, in Anlehnung an die UML, physikalische Systembausteine genannt werden.

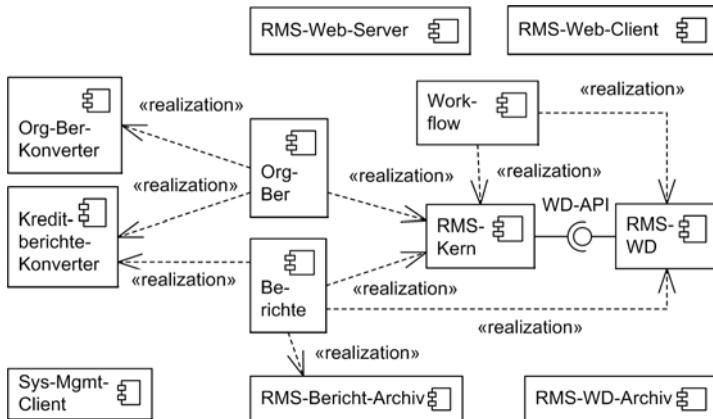


Abb. 9.4-1: Zusammenhang der physikalischen und logischen Systembausteine des RMS auf der Systemebene.

Es wird nun eine weitere Strukturierung vorgenommen, indem durch sogenannte logische Systembausteine für die RMS-Domänen Organisation und Berechtigungen, Berichte und Workflow eingeführt werden. Abbildung 9.4-1 zeigt den Zusammenhang durch Abhängigkeitsbeziehungen vom Stereotyp «realize». So sind die Systembausteine am Fuß des «realize»-Pfeils rein logische Systembausteine, also zunächst reine Spezifikation, und die RMS-Systembausteine an der Pfeilspitze sind die physikalischen Systembausteine, die die Funktionalität der logischen Systembausteine in der Laufzeitumgebung operativ machen.

Die Idee hinter der Entscheidung, die Spezifikation in diese logischen Systembausteine gegliedert vorzusehen, ist folgende: Die logischen Systembausteine sollen möglichst flexibel handhabbar und in späteren Versionen des RMS als Komponenten direkt instanzierbar sein, sobald eine, bislang fehlende, standardisierte Komponentenplattform in der Bank zur Verfügung steht.

E-LOGK:

**Logische Systembau-
steine Org-Ber, Berich-
te und Workflow**

**Fehlende Komponen-
tenplattform**

9.4.2 Bausteinebene

Auf der Bausteinebene werden nun die logischen Systembausteine in den folgenden Abschnitten weiter verfeinert.

Der logische Systembaustein Org-Ber ist spezifiziert durch eine vertikale Schicht auf der Bausteinebene, die eine eigene Domäne Organisation und Berechtigungen des Risikomanagements darstellt. Diese soll ein Konzept für Berechtigungen und Verantwortlichkeiten auf der Basis der

**Flexibles Konzept für
Berechtigungen und
Verantwortlichkeiten**

Organisation- und Berechtigungen-Metamodell

Informationen aus dem Organisations- und Berechtigungsdatensystem

DSL für Organisationspfadregeln

Berichte- und Berichte-Visualisierung-Metamodell

Rahmenwerke und Mikro-Architekturen

Organisations- und Berechtigungsdaten der Bank modellieren (siehe Anforderung A-BZV), das möglichst flexibel einsetzbar ist. Damit sollen auch alle in Zukunft noch zu erwartenden Risikobereiche modelliert werden können (siehe Anforderung A-ERW). Zudem soll das Konzept unabhängig von den entsprechenden Berechtigungs- und Rollen-Konzepten der anzubinden IT-Systeme Organisationsdatensystem, Berechtigungsdatensystem und dem Std-WDMS sein. Wie sind diese Konzepte aber aufgebaut?

Der eingesetzte Systembaustein Std-WDMS operiert bezüglich Rollen- und Berechtigungen lediglich mit dem Konzept von Benutzergruppen (WD-Benutzer und WD-Gruppe). Um hier aber flexibler zu sein, wird als allgemeineres Konzept im RMS ein Organisation-Metamodell und ein Berechtigungen-Metamodell eingesetzt.

Aus dem Organisations- und dem Berechtigungsdatensystem lassen sich die Instanzen der Metaklassen des Organisation- und Berechtigungen-Metamodells gewinnen. Zusammen stellen diese Instanzen dann ein Organisation- und ein Berechtigungen-Modell gemäß dem Organisation- und Berechtigungen-Metamodell dar.

Um die Berechtigungen der Akteure für einen Risikofall definieren und dynamisch berechnen zu können, wird eine spezielle regelbasierte DSL entwickelt (siehe Abschnitt 6.6.3). Diese erlaubt es, die Berechnungsregeln basierend auf dem Organisation- und Berechtigungen-Modell der Bank durch sogenannte Organisationspfadregeln einfach zu formulieren. Als zugrundeliegende Technologie kann hier XPath und XQuery eingesetzt werden.

In der Domäne Berichte werden die mögliche Struktur der Inhalte und deren mögliche Darstellung im RMS-Web-Client durch zwei verschiedene Metamodelle festgelegt. Zur Beschreibung der Berichte eines Risikobereichs wird ein Risikoberichte-Modell als Instanz eines Berichte-Metamodells erstellt. Die Darstellung der Berichte eines Risikobereichs wird durch ein separates Berichte-Visualisierung-Modell gemäß einem Berichte-Visualisierung-Metamodell festgelegt. Für die Erstellung beider Modelle wird ein speziell entwickeltes visuelles Generatorwerkzeug eingesetzt (siehe Abschnitt 9.5.3).

Für die Umsetzung der beschriebenen Konzepte werden verschiedene Rahmenwerke eingesetzt, insbesondere im Web-Bereich. Daneben werden zahlreiche Entwurfsmuster verwendet z. B. zur Entkopplung von Schichtung im Entwurf oder zur Umsetzung von Composite-Strukturen.

Das RMS wird in zahlreichen UML-Diagrammen dargestellt, die ein zugrunde liegendes UML-Modell für das RMS visualisieren. Die Diagramme selbst sind bereits feingranulare Bestandteile verschiedener Sichten auf dieses Modell. Zudem wurde die gesamte Fallstudie nach den Architektur-Dimensionen aufgegliedert, die ihrerseits bereits verschiedene Sichten auf das RMS sind.

Architektur-Modelle und Sichten

9.5 Architektur-Mittel (WOMIT)

9.5.1 Architektur-Prinzipien

Wie bereits im Abschnitt 9.2 zu sehen war, wird das Modularitätsprinzip zur Dekomposition des RMS in Systembausteine eingesetzt und dabei auf lose Kopplung und hohe Kohäsion geachtet (siehe Abschnitt 6.1). Da im RMS-Projektkontext keine Komponentenplattform zur Verfügung steht, werden die verschiedenen physikalischen Systembausteine des RMS als Betriebssystemprozesse instanziert, die über XML-Dateien kommunizieren oder sich gegenseitig mit Prozessparametern aufrufen. Das somit bei Komponentenplattformen automatisch durch das Konzept von Schnittstellen der Komponenten umgesetzte Prinzip des Information Hidings steht also im RMS nicht automatisch zur Verfügung.

Fehlende Komponentenplattform

Dem Fehlen einer Komponentenplattform wird durch die eingesetzte XML-Technologie für die RMS-internen Schnittstellen Org-XML, Ber-XML und Berichte-XML (siehe Abbildungen 9.2-3 und 9.2-4) wie folgt Rechnung getragen: Für jede dieser Schnittstellen wird eine zugehörige DTD erstellt. Für die Schnittstellen Org-XML und Ber-XML werden diese Organisation-DTD und Berechtigungen-DTD äquivalent zu den Metamodellen für die Organisation und die Berechtigungen erstellt. Für die Schnittstelle Berichte-XML wird die äquivalente Berichte-DTD (siehe Abbildung 9.5-1), der Entscheidung E-GENX und E-GENW folgend, generativ mit dem Entwicklungswerkzeug RMS-Gen erzeugt (siehe Abschnitt 9.5.3).

Kommunikation zwischen den physikalischen Systembausteinen über die XML-Schnittstellen

Die XML-Dateien sind somit die Übergabeparameter für eine zugehörige Schnittstellenoperation „import“ und die zugehörigen DTDs spezifizieren deren Typ. Die Operation „import“ liefert für Transaktionszwecke einen Fehlercode zurück. Wie der RMS-Kern die entsprechenden XML-Dateien weiter intern aufbereitet, ist somit vor den Konvertern versteckt. Hier findet also eine Trennung von Schnittstelle und Implementa-

XML-Dateien als Import-Parameter

Hohe Abstraktion durch Berichte- Metamodell

Modularitätsprinzip und Separation-of- Concerns-Prinzip durch logische Systembau- steine

Inkrementalitätsprinzip

tierung statt. Eine Typprüfung der „import“-Parameter (also der XML-Dateien) zur Laufzeit erfolgt insofern, als die XML-Dateien dabei mit einem validierenden XML-Parser untersucht werden. Für die Berichte-XML-Dateien können zusätzliche semantische Überprüfungen bei einem Aufruf von „import“ stattfinden, da hier im Berichte-Modell zusätzliche Angaben über den Inhalt der Berichte gemacht werden können. Die vergleichbare aktuelle Standardtechnologie, die hier eingesetzt werden kann, ist XML-Schema.

Beim gesamten Aspekt Berichte wird das Prinzip der Abstraktion besonders intensiv angewendet: Die Struktur der Berichte eines Risikobereichs, definiert durch das Berichte-Modell (der Typ des Berichts), wird durch eine noch abstraktere Struktur, das Berichte-Metamodell (der Typ des Berichte-Modells), beschrieben (siehe Abbildung 9.5-1).

Die logischen Systembausteine befinden sich auf einer höheren semantischen Ebene als die physikalischen Systembausteine. Dies führt dazu, dass das RMS für die in Zukunft für die Bank als IT-Standard zu erwartende Komponentenplattform gerüstet ist. Das Modularitätsprinzip wird hier angewendet, um die logische Struktur innerhalb der Systembausteine zu modularisieren. Dazu wird das Separation-of-Concerns-Prinzip zunächst grobgranular angewendet, um verschiedene Domänen zu identifizieren und zu trennen. In den Domänen bzw. den logischen Systembausteinen des Systembausteins RMS-Kern, werden durch den Einsatz der ausgewählten objektorientierten Programmiersprache natürlich zahlreiche dieser Prinzipien zusätzlich unterstützt. Dort wird auch im Design eine Schichtung bei den Klassen vorgenommen.

Das Inkrementalitätsprinzip wird angewendet, indem zunächst ein Prototyp des Systembausteins RMS-WD entwickelt wird, bei dem die Umsetzung der Anforderungen zur Prozessdefinition A-PRD, der Ansicht auf Berichte und Prozesse (Anforderungen A-BER und A-STA) und insbesondere auch die Verifikation des geforderten geringen Schulungsbedarfs (Anforderung A-SCH) im Vordergrund stehen. In einer späteren Phase des Projekts findet ein User-Acceptance-Test statt, bei dem die Funktionalitäten der Systembausteine RMS-Kern und RMS-WD und RMS-Web-Server weitestgehend bereits zur Verfügung stehen, sämtliche Infrastruktur-Anbindungen (A-INT) allerdings noch nicht umgesetzt sind.

9.5.2 Grundlegende architektonische Konzepte

Durch Verwendung objektorientierter Programmiersprachen und den Einsatz einer UML-basierten Java-Entwicklungsumgebung, die auch einfache Quelltext-Generierung aus UML-Modellen unterstützt, wird das Konzept der Objektorientierung nahezu im gesamten RMS-Projekt eingesetzt.

Auf die eingesetzten Konzepte Meta-Architekturen und modellgetriebene Software-Entwicklung in Verbindung mit generativer Erzeugung von Systembausteinen wird in den nachfolgenden Abschnitten ausführlicher eingegangen.

Für die eingesetzte Aspektorientierung gibt es drei erwähnenswerte Einsatzbeispiele im RMS:

- Ganz grobgranular der Aspekt Berichte, getragen durch das Berichte-Metamodell im Zusammenhang mit generativen und generischen Verfahren.
- Sehr feingranular auf der Ebene von Klassen und Objekten, wo es um Navigations- und Manipulationsoperationen in Bezug auf Assoziationen zwischen Klassen geht, umgesetzt durch Quelltext-Generierung aus dem verwendeten Java-Entwicklungswerkzeug.
- Ebenfalls sehr feingranular, wenn es um das Logging von Warn- und Fehlermeldungen geht, ebenfalls umgesetzt durch Quelltext-Generierung aus dem verwendeten Java-Entwicklungswerkzeug.

Im Folgenden werden nun die bereits erwähnten generativen und generischen Verfahren näher beschrieben.

9.5.3 Generative und generische Verfahren

Die schnelle Integration weiterer Risikobereiche wird entscheidend durch ein speziell für das RMS entwickeltes Generatorwerkzeug unterstützt (siehe Abschnitt 6.2.5). Dies verkürzt die Entwicklungs- und Testzeiten und verbessert die Stabilität und die Qualität der Implementierung entscheidend.

Damit beim Hinzufügen eines neuen Risikobereichs keine Datenbankschemaänderung nötig ist, wird ein generisches relationales Datenbankschema, das Berichte-Metamodell-Schema, für das RMS-Berichte-Archiv erstellt. Dieses ist äquivalent zum Berichte-Metamodell und

Objektorientierung

**Meta-Architekturen
und modellgetriebene
Software-Entwicklung**

Aspektorientierung

**RMS-Gen steigert die
Entwicklungseffizienz
und -qualität**

**Generisches Daten-
bankschema**

ermöglicht es einerseits jedes konforme Berichte-Modell als Berichte-Modell-Datensätze sowie andererseits die zugehörigen Berichte als Berichte-Instanz-Datensätze abzulegen.

Das Generatorwerkzeug RMS-Gen

Spezifikation für die Schnittstelle Berichte-XML

Mit dem RMS-Gen wird das Editieren von Berichte-Modellen und Berichte-Visualisierung-Modellen sowie das Generieren daraus abgeleiteter Artefakte, die auf den Modellen basieren, ermöglicht. Aus den Modellen für einen Risikobereich generiert RMS-Gen für die Schnittstelle Berichte-XML zum Beispiel folgende Artefakte:

- > Eine Berichte-Modell-XML-Datei, vergleichbar mit einem XML-Schema für die Struktur der Berichte eines Risikobereichs.
- > Eine Berichte-DTD-Datei, die der RMS-Kern beim Importieren über Berichte-XML benutzt, um die Berichte auf Korrektheit zu überprüfen.
- > Eine HTML-basierte Dokumentation für die Schnittstelle Berichte-XML, die den Entwicklern für einen Berichte-Konverter zusammen mit der Berichte-DTD-Datei als Spezifikation dient.

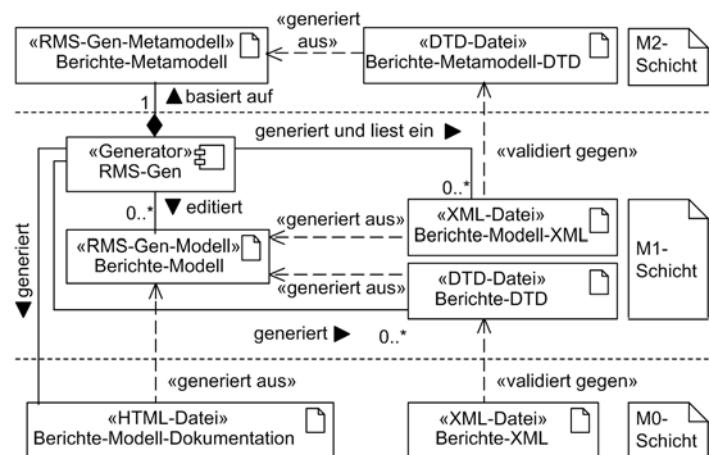


Abb. 9.5-1: Generierte Artefakte für die Schnittstelle Berichte-XML.

Darüber hinaus kann RMS-Gen Berichte-Modelle auch als Berichte-Modell-XML-Dateien einlesen, die mit anderen Werkzeugen erstellt wurden. Zur Validierung solcher Dateien mit XML-Standardwerkzeugen kann RMS-Gen auch eine Berichte-Metamodell-DTD-Datei erzeugen.

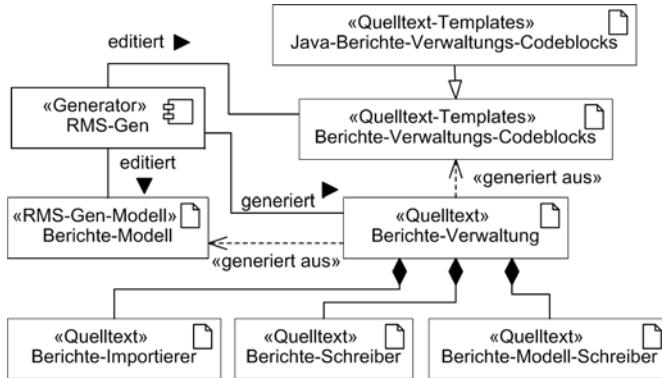


Abb. 9.5-2: Generierte Artefakte für die Berichte-Verwaltung.

Neben diesen XML-basierten Dateien generiert RMS-Gen pro Risikobereich auch Quelltext. Der Quelltext kann unabhängig von einer konkreten Programmiersprache, basierend auf sogenannten Codeblocks, die als Templates dienen, in einer eigenen Template-Sprache geschrieben werden. Es genügt dann, für die gewünschte Zielprogrammiersprache die Codeblocks zu spezialisieren, um den lauffähigen Quelltext dafür zu generieren.

Für die Berichte-Verwaltung in den Systembausteinen RMS-Kern und RMS-Berichte-Archiv wird auf diese Weise der Quelltext generiert. Dieser beinhaltet folgende Bestandteile:

- > *Berichte-Importierer*: implementiert das Importieren von Berichten nach RMS-Kern über die Schnittstelle Berichte-XML;
- > *Berichte-Schreiber*: implementiert das Ablegen der Berichte von RMS-Kern aus über die Schnittstelle Berichtezugriff ins Berichte-Archiv als Bericht-Instanz-Datensätze. Dazu liest RMS-Kern die Berichte-Modell-XML-Datei ein und benutzt diese als Metainformation;
- > *Berichte-Modell-Schreiber*: implementiert das Ablegen von Bericht-Modellen vom Systembaustein RMS-Kern aus als Berichte-Modell-Datensätze im RMS-Berichte-Archiv. Dazu liest RMS-Kern Bericht-Modelle als Berichte-Modell-XML-Datei ein.

Mit dem RMS-Gen wird passend zum Berichte-Modell ein Berichte-Visualisierung-Modell für jeden Risikobereich erstellt. Dabei kann RMS-Gen ebenfalls durch entsprechende Quelltext-Templates für verschiedene Visualisierungstechnologien konfiguriert werden. Dafür werden die nötigen Codeblocks für die entsprechenden Zieltechnologien implementiert. Dabei handelt es sich z. B. um die Std-WDMS-eigene HTML-basierte Template-Sprache WDMS-Skript oder auch XSLT. RMS-Gen

Quelltext-Template-Sprache

Quelltext für die Berichte-Verwaltung

Quelltext für die Berichte-Visualisierung

generiert je nach gewünschter Zielsprache folgende Bestandteile für die Berichte-Visualisierung:

- > *Berichte-Leser*: implementiert das Lesen von Berichten ausgehend von RMS-WD aus dem RMS-Berichte-Archiv über die Schnittstelle Berichtezugriff. Dazu liest der Systembaustein RMS-WD die Berichte-Modell-XML-Datei ein und benutzt diese als Metainformation;
- > *Berichte-Visualisierer*: implementiert das Visualisieren von Berichten durch RMS-WD basierend auf dem zugehörigen Berichte-Visualisierung-Modell. Dazu liest der RMS-WD Systembaustein die Berichte-Visualisierung-Modell-XML-Datei ein und benutzt diese als Metainformation.

Probleme und Lektionen

Komplexere Debugging-Struktur

Der Einsatz von generativen Verfahren bringt im Projekt nicht nur die bereits diskutierten Vorteile, sondern führt zu zusätzlichen Problemen. So wird mit dem RMS-Gen und der Quelltext-Generierung aus der Java-Entwicklungsumgebung im Grunde die Entwicklung eines Entwicklungswerkzeugs betrieben. Dies erhöht die Komplexität bei der Fehler suche zusätzlich. Auch bei der Bereinigung von Fehlern ist jeweils zu entscheiden, ob ein Fehler im Werkzeug oder in den Quelltext-Templates liegt. So muss bei jedem Verändern von RMS-Gen sichergestellt bzw. getestet werden, ob auch die anderen, bislang nicht fehlerhaft erzeugten Artefakte weiterhin fehlerfrei bleiben. Hinzu kommt, dass jede Zielsprache ihre eigene Debugging-Umgebung besitzt und im Grunde eine integrierte Umgebung hierzu benötigt wird, die bei einer Fehlermeldung direkt auf die zugehörigen Modellelemente verweist, aus denen der fehlerhafte Bestandteil des Artefakts generiert wurde. Eine solche Debugging-Umgebung zusätzlich zu entwickeln sprengt aber in der Regel den vertretbaren Rahmen eines Anwendungsentwicklungsprojekts.

9.6 Organisationen und Individuen (WER)

In diesem Abschnitt werden die Einflüsse und Auswirkungen von Organisation und Individuen auf das RMS-Projekt behandelt.

9.6.1 Organisation

Organisatorische Anforderungen der Bank

Die Projektorganisation, die Kommunikationswege wie auch der anzu wendende Projektmanagementprozess sind in unserem Projekt durch die Bank vorgeschrieben.

Diese Strukturen ändern sich aber in unserer Bank während des Projekts: In einem Outsourcing-Prozess wird der Projektauftrag an einen externen Auftragnehmer vergeben, gleichzeitig wird eine organisatorische Umstrukturierung in der Bank vorgenommen.

Als entscheidender Vorteil wirkt sich jetzt aus, dass das Projekt, bereits während der Entwicklungszeit innerhalb der Bank ähnlich einem externen Projekt durchgeführt wird.

Die organisatorische Umstrukturierung der Bank wirkt sich aber auch auf die betrieblichen Prozesse der Bank aus und beeinflusst insbesondere auch die Risikofallbearbeitungsprozesse. Nun zahlt sich die während des Projekts getroffene Architektur-Entscheidung E-KONV aus, die die Organisationsstruktur der Bank auf ein RMS-internes Organisationsmodell abbildet, das auf einem eigenen Metamodell basiert und damit die Risikofallbearbeitungsprozesse optimal entkoppelt.

Das Werkzeug RMS-Gen erzeugt Artefakte auf der Basis von Zielsprachen wie Java und XML. Diese Artefakte sind dabei so strukturiert, als wären sie manuell erstellt worden. Somit kann der Projektvertrag vorsehen, dass RMS-Gen, die Generierungskonfigurationen des Java-Entwicklungswerkzeugs und die zugehörigen Metamodelle nicht zum Lieferumfang gehören, sondern nur die generierten Artefakte. Die Bank kann somit zwischen dem Risiko für die eventuell entstehenden Mehrkosten für die Entwicklung des RMS-Gen und die Chance auf Kosteneinsparungen bei Folgeprojekten wählen.

9.6.2 Individuen

Im RMS-Projekt wird den Entwicklern ein hoher Grad an Abstraktionsvermögen abverlangt. Die stark modellgetriebene Vorgehensweise schafft nicht nur Vorteile, sondern auch einige Probleme im Team. Der Schulungsbedarf ist nicht allein dadurch zu bewältigen, dass die Mitarbeiter etwa auf einen UML-Modellierungskurs geschickt werden. Diese Maßnahme kann allenfalls als Basis dienen. Als wesentlich stellt sich heraus, dass nur ein Coaching durch erfahrenere Projektmitglieder und eine dem modellbasierten Vorgehen angepasste Werkzeugunterstützung falsche Vorgehensweisen und Verletzung getroffener Architektur-Entscheidungen verhindern.

Änderung der Organisation

Vorteile einer skalierten Struktur

Auswirkungen auf das IT-System RMS

Generative Verfahren: Risiko und Chance

Heterogene Teamzusammensetzung als Vorteil nutzen

Architektur muss unterschiedlich qualifizierte Entwickler zulassen

Als äußerst praktikabel erweist es sich auch, dass bei der Strukturierung des Systems in Systembausteine pro Systembaustein entschieden werden kann, wie hoch der Abstraktionsgrad der dafür eingesetzten Technologie ist. So können gezielt die nicht so zentralen Systembausteine ohne generative Verfahren, also zunächst auf herkömmliche Weise, entwickelt werden, solange sich dies bei der Kommunikation an den Schnittstellen zu den anderen Systembausteinen nicht zeigt.

9.7 Architektur-Vorgehen (WIE)

Das Vorgehen im Projekt orientiert sich am Entwicklungsprozess, wie er in Kapitel 8 dargestellt wird.

Auswechslung des Entwicklungsprozess

Wie in Abschnitt 9.6 bereits beschrieben, wird das RMS-Projekt zunächst in der Bank selbst, nach einem Fertigstellungsgrad von etwa 30 % aber bei einem externen Auftragnehmer weitergeführt. Dadurch ändert sich auch der zugrunde liegende Entwicklungsprozess. Hier zeigt sich, dass dies für den Projekterfolg unproblematisch ist, solange der Projektleiter und der maßgebliche Architekt dieses Projekt weiterführen.

Architektonische Tätigkeiten

Beide Vorgehen waren iterativ und inkrementell und im Wesentlichen analog zu den Grundlagen, die dazu in diesem Buch beschrieben sind (siehe Kapitel 8).

Anforderungen verstehen und Architektur entwerfen

Die zunächst zu erstellende Systemvision wurde in Abschnitt 9.2.1 als Ausgangspunkt für das Modellieren und Verstehen der Anforderungen im gesamten Abschnitt 9.2 formuliert. Das Entwerfen der Architektur wurde für das RMS durch das schrittweise Ableiten von Architektur-Entscheidungen aus diesen Anforderungen in den Abschnitten 9.3 und 9.4 vorgestellt. Für das Kommunizieren der Architektur wird als zentrales und durchgängiges Mittel die UML eingesetzt. Das Umsetzen der Architektur wurde dann ausführlicher in den Abschnitten 9.3. und 9.5 beleuchtet.

DSLs und MDSD

Der in den vorausgegangenen Abschnitten herausgearbeitete Aspekt Berichte spielt für das RMS eine zentrale Rolle für die Integration weiterer Risikobereiche. Folgerichtig werden daher auch Mittel wie Frameworks, DSLs und MDSD eingesetzt. Die Analyse, das Design und die Implementierung, bezogen auf eine Modellierung nach Domänen ist daher auch in dieser Fallstudie entscheidend und schlägt sich einerseits in domänen- und andererseits in anwendungsbezogenen Entwicklerrollen nieder.

10 | CRM-Kundendatenbank

Diese Fallstudie beschreibt den Entwurf und den Aufbau einer zentralen Kundendatenbank (KDB) im Rahmen der Einführung eines umfangreichen CRM-Programmes. Voraussetzung für die erfolgreiche Einführung des CRM-Programmes war es, einen konsistenten Bestand an Kundenstammdaten zu haben. Diese Daten sollten allen Systemen des Auftraggebers zentral zur Verfügung gestellt werden. Um Plattformunabhängigkeit zu gewährleisten, sollten alle Daten über Web Services angeboten werden. Eine weitere wichtige Anforderung war die fortlaufende Sicherstellung der Konsistenz der Kundendaten über die angeschlossenen Systeme und die Schaffung einer Möglichkeit, bei Datenverlust in einem System die verlorenen Daten wiederherstellen zu können.

Dieses Kapitel betrachtet den Aufbau der zentralen KDB, wobei besonders die Software- und Integrationsarchitekturen näher beleuchtet werden. Die im Anwendungsszenario EAI (siehe Abschnitt 8.8) beschriebenen Vorgehensweisen finden hier ihre praktische Anwendung.

IT-Architekten, die sich mit der Integration verschiedener Systeme beschäftigen, werden hier interessante Lösungsansätze für immer wieder auftretende Integrationsprobleme finden.

Übersicht

10.1 Überblick	474
10.2 Architektur-Anforderungen (WARUM)	475
10.3 Architekturen und Architektur-Disziplinen (WAS)	484
10.4 Architektur-Perspektiven (WO)	493
10.5 Architektur-Mittel (WOMIT)	494
10.6 Organisationen und Individuen (WER)	495
10.7 Architektur-Vorgehen (WIE)	496
10.8 Fazit	497

10.1 Überblick

Ziel: Aufbau einer optimalen Kundenbeziehung

Mit dem Wechsel in das neue Jahrtausend nahm mit dem verstärkten Trend zur Konzentration in internationalem Maßstab auch der Wettbewerb zwischen den verschiedenen Unternehmen zum Personentransport zu. Für den Auftraggeber des dieser Fallstudie zugrunde liegenden Projektes ergab sich daher die Notwendigkeit, sich auf die Kernkunden zu konzentrieren mit dem Ziel der langfristigen Bindung durch Aufbau einer optimalen Kundenbeziehung. Der Zielkunde sollte an allen Kundenkontaktpunkten schnell erkannt werden, damit ihm ein entsprechender Service geboten werden konnte. Die bestehende IT-Systemlandschaft ließ jedoch die Bereitstellung der benötigten Informationen aufgrund mangelnder Vernetzung nicht zu.

Mittel: Zentrale Datenbank für Kundendaten

Die folgende Fallstudie beschreibt den Entwurf und den Aufbau einer zentralen Datenbank für Kundendaten, das im Rahmen eines CRM-Programmes den Auftraggeber in die Lage versetzt, die benötigten Kundeninformationen an allen Kundenkontaktpunkten zur Verfügung stellen zu können.

Umfang des Projektes

Der initiale Aufbau der zentralen KDB hat etwa zwei Jahre gedauert und wurde Mitte 2003 abgeschlossen. Für die Anbindung weiterer Systeme und für das Anbieten neuer Dienste ist ein Prozess etabliert worden, der diese Aufgaben als Kombination von Projekt und Wartung abdeckt. Die damit verbundenen Änderungen waren aufgrund einer tragfähigen Architektur sehr gut umzusetzen. Die Kosten für die Einführung eines solchen CRM-Systems sind natürlich nicht unerheblich, besonders wenn bestehende Schnittstellen zwischen Bestandssystemen entfallen, neue geschaffen und Datenflüsse geändert werden. So kann für die Einführung des gesamten CRM-Systems eine Summe im mehrstelligen Millionenbereich veranschlagt werden, wovon etwa 30 % auf die KDB entfallen. Hiervon wiederum müssen die Anteile für Infrastruktur, Beratungsleistungen und Betrieb bestritten werden.

Englische Dokumentation

Da in dem Projekt die gesamte Dokumentation in englischer Sprache erstellt wurde, sind auch die Bezeichnungen in den Abbildungen in Englisch. Zum einfacheren Verständnis werden im Text die englischen Bezeichnungen verwendet.

10.2 Architektur-Anforderungen (WARUM)

Die Liberalisierung des Luftverkehrs, der daraus resultierende Wettbewerbsdruck sowie die zunehmende Markt- und Preistransparenz bestimmen das Markttumfeld, in dem der Auftraggeber bestehen muss. Dieses Umfeld zwingt zur Entwicklung neuer Strategien, die eine eindeutige Differenzierung gegenüber den Wettbewerbern ermöglichen. Produktqualität und Preis reichen für eine solche strategische Differenzierung nicht mehr aus und sind zudem leicht durch die Wettbewerber imitierbar. Eine langfristige Differenzierung und damit das Erlangen von Wettbewerbsvorteilen kann deshalb nur mit einer durchgängigen Ausrichtung an den Bedürfnissen der Kunden erreicht werden.

Basierend auf dieser Erkenntnis sollte ein CRM-Programm eingeführt werden mit dem Ziel, die wichtigen Kunden durch den Aufbau einer optimalen Kundenbeziehung langfristig an unseren Auftraggeber zu binden.

Die Konzentration auf den Kernkunden ermöglicht es, diesen gezielt anzusprechen und entsprechende Dienstleistungen anbieten zu können. Dadurch ist ein hoher Differenzierungsgrad im Vergleich zu den Mitbewerbern gegeben. Eine Imitation ist aufgrund der komplexen Kundenbeziehung kaum möglich.

Ebenso ist die Nutzung der Kundeninformationen durch Integration und Verteilung über die gesamte Prozesskette nicht auf Mitbewerber transferierbar und führt zu einer Verbesserung der internen Abläufe.

Voraussetzung für die Umsetzung dieser ehrgeizigen Ziele war die Realisierung einer Datenbank, die alle Daten der Kunden enthalten sollte, die aus CRM-Sicht für die Schaffung dieser komplexen Kundenbeziehung benötigt wurden.

Die Architektur der KDB wurde daher mit dem Ziel entworfen, eine gute Integration der betroffenen Anwendungen und Datenquellen in den CRM-Verbund zuzulassen und die Qualität der Kundeninformationen fortlaufend zu verbessern. Der Fokus wurde dabei sowohl auf die Geschäftslogik zur Sicherung der Datenkonsistenz als auch auf die Einhaltung der nicht-funktionalen Anforderungen gelegt. Im Rahmen des Anwendungsszenarios EAI (siehe Abschnitt 8.8) wird auf mögliche Treiber für eine Integrationslösung eingegangen.

Geschäftskontext

CRM-Programm

Konzentration auf den Kernkunden

Nutzung der Kundeninformationen

Zentrale Datenbank für Kundendaten

Architektur der KDB

Hauptziele	Für die zentrale KDB werden die folgenden drei Hauptziele formuliert: <ul style="list-style-type: none"> > Reduzierung und Vereinheitlichung der Schnittstellen und der Datenflüsse und damit Senkung der Kosten, > Verbesserung der Datenqualität und > Schaffung einer Möglichkeit, einfach auf konsistente Kundendaten zugreifen und diese modifizieren zu können.
	Aus diesen Hauptzielen ergeben sich eine Reihe von Anforderungen.
Funktionale Anforderungen	Wichtige funktionale Anforderungen sind dabei: <ul style="list-style-type: none"> > Die Schaffung einer einheitlichen Sicht auf Kundenstammdaten aus CRM-Sicht. > Die Bereitstellung von Diensten, über welche Kundendaten abgefragt und modifiziert werden können. > Die Sicherstellung der Konsistenz der Kundendaten über alle ange schlossenen Systeme.
Nicht-funktionale Anforderungen	Darüber hinaus lassen sich folgende nicht-funktionale Anforderungen ableiten: <ul style="list-style-type: none"> > Gewährleistung einer sehr hohen Verfügbarkeit und Skalierbarkeit. > Gute Betreibbarkeit mit wenig Aufwand trotz der hohen Komplexität. > Antwortzeit für die Abfrage einzelner Kunden in 90 % der Fälle von unter einer Sekunde. > Reduktion der Prozess- und Verwaltungskosten für das Management der Kundendaten.
Plattform vorgegeben	Eine wichtige organisatorische Rahmenbedingung für die Einhaltung dieser Ziele ist die Tatsache, dass die der KDB zugrunde liegende Plattform (siehe Abschnitt 3.4) durch den Auftraggeber vorgegeben ist.
	10.2.1 Ausgangssituation
Voraussetzung	Die wichtigste Voraussetzung für die erfolgreiche Umsetzung der Vision des Auftraggebers ist die Fähigkeit, Kundenwissen durchgängig an allen Kundenkontaktpunkten verfügbar zu machen.
Analyse	Ausgehend von dieser Vision wurde analysiert, in welchen Systemen wichtige Kundeninformationen zu finden waren und welche Datenflüsse bezüglich dieser Daten vorlagen.

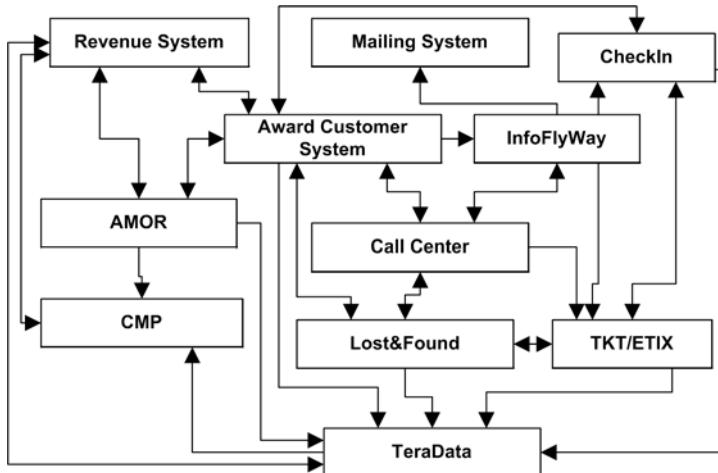


Abb. 10.2-1: Ausgangssituation – Schnittstellen zwischen den Systemen.

Das Ergebnis dieser Analyse zeigte, dass

- > die für das CRM-System wichtigen Kundendaten über eine Vielzahl von Systemen verstreut waren,
- > eine Vielzahl von Schnittstellen zwischen diesen Systemen existierten (Abbildung 10.2-1),
- > keine klare Hoheit über die Änderung von wichtigen Kundendaten zwischen den Systemen bestand,
- > Datenflüsse und die daraus resultierenden Änderungen nicht immer nachvollziehbar waren,
- > zwischen den einzelnen Systemen unterschiedliches Verständnis über den Inhalt von gleichnamigen Datenentitäten bestand und
- > Inkonsistenzen zwischen den Kundendaten in den Systemen nicht auszuschließen waren.

Analyseergebnisse

Die Umsetzung des für den Auftraggeber zu implementierenden CRM-Systems wurde aufgrund der hohen Komplexität in eine Reihe von Einzelvorhaben untergliedert, die unter gemeinsamer Steuerung in einer definierten Abfolge angegangen wurden. Diese Vorhaben waren:

- > Implementierung einer Customer-Relationship-Management-Software.
- > Gestaltung einer modernen Call-Center-Infrastruktur.
- > Aufbau eines zentralen Data Warehouse.
- > Schaffung einer einheitlichen Sicht auf Kundendaten durch eine zentrale Kundendatenbank.
- > Entwurf und Realisierung einer integrierten Systemlandschaft.

Umsetzung

Das Vorhaben, das in dieser Fallstudie weiter betrachtet werden soll, ist der Entwurf und die Implementierung der zentralen Kundendatenbank.

10.2.2 Anforderungen

Konsistente Kundendaten

Die Einführung einer zentralen Kundendatenbank soll die Basis für den Erfolg des gesamten CRM-Programms schaffen – konsistente Kundendaten.

Heterogene Systemlandschaft

Wie in der Mehrzahl aller Unternehmen ist auch die Systemlandschaft des Auftraggebers dieser Fallstudie stark heterogen. Host-Systeme bestehen neben UNIX-basierten Client-Server-Systemen. Es kommen verschiedene Technologien und Programmiersprachen zum Einsatz. Daraus und aus den übergeordneten Zielen ergibt sich eine Reihe von Anforderungen, anhand derer der Architekt unter Einsatz seiner Erfahrungen und eines bewährten methodischen Vorgehens eine tragfähige Architektur erstellen soll.

A-RedSch: Reduzierung und Ver- einheitlichung der Schnittstellen

Eine solche historisch gewachsene Systemlandschaft mit einer großen Anzahl von Systemen, deren Aufgaben sich zum Teil überschneiden und von denen einige zudem als Zwischenlösung vorgesehen waren, erfordert eine Neugestaltung, wenn neue Anforderungen nicht mehr mit sinnvollem Aufwand umgesetzt werden können. Die Forderung nach der Bereitstellung konsistenter Kundendaten an allen Kundenkontaktpunkten war mit der bestehenden Systemlandschaft so nicht mehr realisierbar, sodass eine Konsolidierung hinsichtlich der Reduzierung und Vereinheitlichung der Schnittstellen und Anpassung der Datenflüsse stattfinden musste.

A-Integ: Integration aller rele- vanten Systeme

Verstreut über diese Landschaft sind die wichtigen Daten – die Kundendaten als Herzstück und Grundlage jedes CRM-Systems. Das Kontextdiagramm (Abbildung 10.2-2) zeigt die Zusammenhänge zwischen den wichtigsten Systemen, die im Rahmen der ersten Projektphasen an die KDB angebunden werden sollten. Es gibt Aufschluss darüber, wie die KDB in die Systemlandschaft eingebettet ist und zwischen welchen Systemen Schnittstellen bestehen werden.

Touchpoint-Systeme

Die Touchpoint-Systeme dienen direkt der Unterstützung der Mitarbeiter des Auftraggebers bei verschiedenen Prozessen rund um den Kunden und haben keine eigene Datenhaltung.

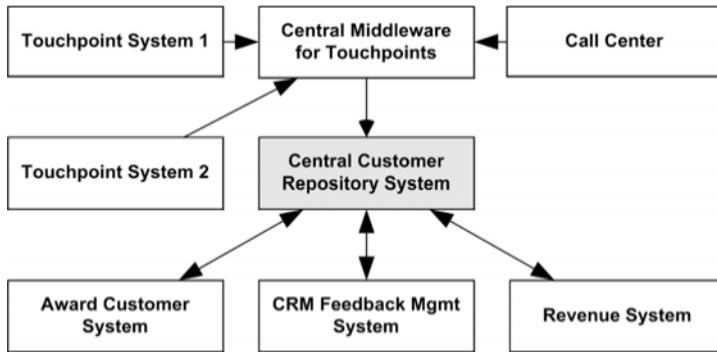


Abb. 10.2-2: Kontextdiagramm.

Das Call-Center ist ein Beispiel für ein Touchpoint-System, mit dem Mitarbeiter des Auftraggebers Kundenanfragen bearbeiten. Eine solche Bearbeitung kann auch die Änderung der Kundenstammdaten beinhalten.

Call-Center

Mithilfe des Revenue-Systems werden Dienstleistungen abgerechnet, die von Auftragnehmern wie Reisebüros erbracht wurden.

Revenue-System

Das Awardkundensystem dient den speziellen Belangen der Kunden, die im Besitz einer Kundenkarte sind und besondere Aufmerksamkeit verdienen.

Awardkundensystem

Die zentrale Middleware ist konzipiert, um Touchpoint-Systemen einfachen Zugang zu Kundendaten und der dazugehörigen Pflegefunktionalität zu ermöglichen und um Komplexität zu verbergen.

Central Middleware for Touchpoints

Das CRM-Feedback-Management-System hat neben der Verwaltung von Kundenfeedback die Aufgabe, Marketingmaßnahmen zu unterstützen.

CRM-Feedback-Mgmt-System

Alle diese Systeme haben mit Kundendaten zu tun, die auch für CRM interessant sind, und müssen sich daher nahtlos in ein CRM-System integrieren lassen.

A-Erw: Erweiterbarkeit

In der Systemlandschaft des Auftraggebers gibt es noch andere mit Kundendaten operierende Systeme, die an die KDB angeschlossen werden könnten. Es werden Systeme abgelöst und andere eingeführt werden. Um die Integration neuer Systeme zu ermöglichen, muss die KDB erweiterbar sein, wobei der Aufwand in einem vernünftigen Rahmen bleiben muss.

A-Sk: Skalierbarkeit

A-DQ:
Verbesserung der Datenqualität

Wie aus der Beschreibung der Ausgangssituation erkennbar wird, ist die Qualität der Kundendaten aufgrund der Verteilung über eine Anzahl von Systemen mit unterschiedlichen Blickwinkeln auf die Daten schlecht. Die KDB soll helfen, die Datenqualität spürbar zu verbessern.

A-DK:
Sicherstellung der Konsistenz der Daten über alle Systeme

Aus der Ausgangssituation ist darüber hinaus zu erkennen, dass die Konsistenz der Daten mangelhaft ist. Zur Erlangung qualitativ hochwertiger Daten soll die KDB sicherstellen, dass die Konsistenz der Kundendaten über alle angeschlossenen Systeme gewährleistet wird.

A-Tr:
Transaktionssicherheit

Eine der Hauptaufgaben der KDB ist das Ändern von Kundendaten. Dabei können verschiedene Teile der Kundendaten in einer Anfrage geändert werden. Die Anforderung in diesem Zusammenhang ist, dass entweder alle Änderungen innerhalb einer solchen Anfrage geändert werden oder die ganze Anfrage abgewiesen wird. Anfragen sollen also als atomare Operation ausgeführt werden.

A-HV:
Hochverfügbarkeit

Die Verfügbarkeit von Kundenstammdaten ist eine wichtige Voraussetzung für das Funktionieren einer Vielzahl von Systemen des Auftraggebers. Daher ist eine Verfügbarkeit von über 99,9 % sicherzustellen.

A-Betr:
Betreibbarkeit

Bei dem Entwurf der KDB ist dem Aspekt der Betreibbarkeit besonderes Augenmerk zu schenken. Die Hauptaufgabe des technischen Betreibers ist es, rund um die Uhr den Betrieb der KDB sicherzustellen. Dazu muss der Zustand aller Bausteine der KDB beobachtet und ein Ausfall sofort gemeldet werden. Alle System-Bausteine sollen im Fehlerfall über einen einheitlichen Logging-Mechanismus Meldungen in entsprechende Logdateien schreiben.

Tab. 10.2-1: Zusammenfassung der Anforderungen.

Anforderung	Art
Reduzierung und Vereinheitlichung der Schnittstellen (A-RedSch)	funktional
Integration aller relevanten Systeme (A-Integ)	funktional
Erweiterbarkeit, um weitere Systeme integrieren zu können (A-Erw)	funktional
Skalierbarkeit (A-Sk)	nicht-funktional
Verbesserung der Datenqualität (A-DQ)	funktional
Sicherstellung der Konsistenz der Daten über alle Systeme (A-DK)	funktional
Transaktionssicherheit (A-Tr)	funktional
Hochverfügbarkeit (A-HV)	nicht-funktional
Betreibbarkeit (A-Betr)	nicht-funktional

10.2.3 Anwendungsfälle

Aus dem Kontextdiagramm (Abbildung 10.2-2) ist ersichtlich, dass eine Vielzahl von Systemen als Akteur gegenüber der KDB auftritt.

Akteure

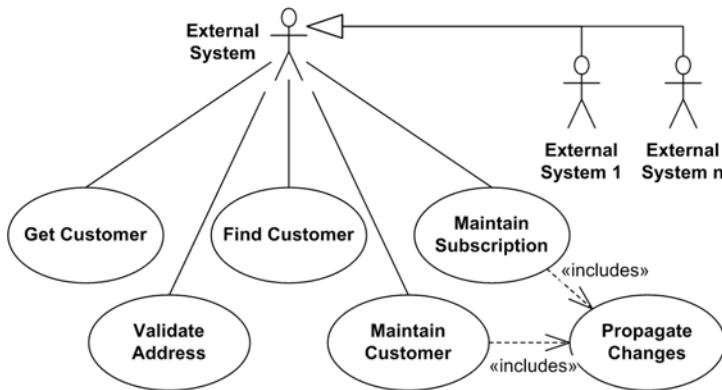


Abb. 10.2-3: Hauptanwendungsfälle und Akteure.

Die Definition und Beschreibung der Anwendungsfälle ist üblicherweise keine zentrale Aufgabe von Architekten. Dennoch hat es sich als zweckmäßig erwiesen, wenn diese Tätigkeit von Architekten begleitet wird. So kann bereits in dieser frühen Phase

- > die Machbarkeit sichergestellt,
- > die architektonische Relevanz der Anwendungsfälle untersucht,
- > eine Priorisierung der Anwendungsfälle vorgenommen und
- > eine genauere Schätzung für eine spätere Realisierung abgegeben werden.

Der Funktionsumfang der zentralen KDB soll anhand der wichtigsten Anwendungsfälle (Abbildung 10.2-3) näher beschrieben werden.

Anwendungsfälle

Der Anwendungsfall *Maintain Customer* beinhaltet alle Möglichkeiten, Kundenstammdaten zu modifizieren. So ist hier neben der Anlage von Kunden in der KDB auch deren Änderung möglich.

Maintain Customer

Der Anwendungsfall *Get Customer* erlaubt es, gezielt über einen Identifikator (z. B. die Kundennummer) die Daten eines Kunden abzurufen.

Get Customer

Der *Find Customer* erlaubt die Suche nach Kundenstammdaten über eine Reihe festgelegter Suchkriterien. Als Ergebnis der Suche wird eine Liste von Kundennummern zurückgegeben.

Find Customer

Maintain Subscription

Jedes der betrachteten Systeme verwendet Identifikatoren zur eindeutigen Kennzeichnung von Entitäten wie Kunde (Kundennummer) oder Adresse (Adress-ID). Gleiche Instanzen einer Entität haben in unterschiedlichen Systemen verschiedene IDs. Um eine integrierte Sicht der Daten zu bekommen und Daten zwischen den Systemen korrekt auszutauschen, ist das Wissen über die Beziehungen dieser IDs von essentieller Bedeutung. In der KDB werden neben Kundenstammdaten auch die Beziehungen der IDs für die einzelnen Instanzen jeder Entität zwischen KDB und externen Systemen als Crossreferenz hinterlegt. Dieses Verfahren wird auch als *KeyMapping* bezeichnet. Mit der Hinterlegung einer solchen Beziehung wird gleichzeitig das Interesse eines externen Systems bekundet, über jede Veränderung dieser Datenentität informiert zu werden. Der Anwendungsfall *Maintain Subscription* beinhaltet die Hinterlegung solcher Beziehungen.

Validate Address

Adressen sind ein zentraler Bestandteil von Kundenstammdaten. Um sicherzustellen, dass die Adressdaten eine hohe Qualität haben, wird externen Systemen über den Anwendungsfall *Validate Address* die Möglichkeit geboten, Adressen vor dem Einpflegen nach festgelegten Kriterien zu validieren und erforderlichenfalls zu korrigieren.

Propagate Changes

Ein aus der Sicht der Konsistenzhaltung der Daten im CRM-Verbund wichtiger Anwendungsfall ist *Propagate Changes*. Er stellt sicher, dass alle Änderungen an Kundenstammdaten über entsprechende Nachrichten an die angeschlossenen Systeme gemeldet werden. Jedes System schickt nach erfolgreicher Verarbeitung eine Bestätigung an die KDB.

Insert Customer

Externe Systeme im CRM-Verbund sollen die Möglichkeit haben, Kunden in der KDB anzulegen. Der Anwendungsfall *Insert Customer* betrachtet genau diese Funktionalität. Bei der Anlage eines Kunden kann nur eine Neuanlage der Kind-Entitäten, wie z.B. Adressen, erfolgen.

Update Customer

Der Anwendungsfall *Update Customer* beschreibt die ganze Palette möglicher Änderungen an Kundenstammdaten. Kind-Entitäten können u.a. Adressen, Präferenzen oder Sprachen sein. Im Rahmen des *Update Customer* kann mit einem Aufruf eine Vielzahl von Kind-Entitäten eines Kunden bearbeitet werden.

10.2.4 Architekturrelevante Anforderungen

In diesem Abschnitt soll der Einfluss der Anforderungen auf architektonische Entscheidungen untersucht werden.

Die Anforderung, Schnittstellen zu reduzieren und zu vereinheitlichen, hat einen wesentlichen Einfluss auf die Integrationsarchitektur. Sie führt letztlich zu der Definition und Einführung einer Schnittstelle, die in der Lage ist, alle oben genannten Anwendungsfälle abbilden zu können. Die Integrationsarchitektur muss Funktionalität für diese Schnittstelle vorsehen.

A-RedSch

Die Integrierbarkeit aller relevanten Systeme beinhaltet zumindest zwei wesentliche Aspekte:

- > Die KDB muss über eine Datenarchitektur verfügen, die generisch genug ist, um die Daten aus unterschiedlichen Systemen abbilden zu können. Dies ist sehr schwierig, weil die Liste der zu integrierenden Systeme ständig wächst.
- > Die zu integrierenden Systeme müssen in der Lage sein, mit der KDB zu kommunizieren. Dieser Aspekt hat starken Einfluss auf die Integrationsarchitektur, da diese für alle zu integrierenden Systeme eine Lösung bereitstellen muss. Die Ansätze unterscheiden hierbei nach Online- und Offline-Anbindung sowie danach, ob das zu integrierende System direkt die gemeinsame Schnittstelle bedienen kann oder ob eine Transformation erfolgen muss.

A-Integ

Die Integration neuer Systeme hat neben den bereits genannten Aspekten Einfluss auf die Software-Architektur. Diese muss eine einfache Erweiterung erlauben. Im Idealfall soll dies durch Anpassung der Konfiguration möglich sein, was jedoch nicht immer gewährleistet sein wird, wenn beispielsweise für ein System ein neuer Integrationsbaustein geschaffen werden muss.

A-Erw

Der Anforderung nach Skalierbarkeit wird in erster Linie durch eine entsprechende Gestaltung der Anwendungsarchitektur Rechnung getragen. Diese stellt sicher, dass Engpässe vermieden werden und stark frequentierte Bausteine redundant ausgelegt sind, sodass dynamisch auf die Auslastung des Systems reagiert werden kann. Die Auslegung der Software-Architektur erfolgt auch im Hinblick auf die optimale Ausnutzung der Möglichkeiten des Anwendungsservers bezüglich der Verteilung von Bausteinen. Die Anforderung hat auch Auswirkungen auf die Gestaltung der Hardwarearchitektur, die eine solche Verteilung ermöglichen muss.

A-Sk

Die Verbesserung der Datenqualität stellt Ansprüche an die Software-Architektur, da diese eine entsprechende Geschäftslogik bereitzustellen hat.

A-DQ

A-DK	Die Sicherstellung der Konsistenz der Daten über alle Systeme erfordert, dass die Hoheit über Daten und die Reaktion auf Konfliktsituationen festgelegt sind und sich in den Integrations- und Software-Architekturen widerspiegeln. Dies hat Auswirkungen auf die Gestaltung der Schnittstelle selbst wie auch auf die Auslegung der Integrationsbausteine der zu integrierenden Systeme.
A-Tr	Aus der Sicht der KDB ist eine Änderungsanfrage gleichzeitig eine Transaktion. Das bedeutet, dass entweder alle Änderungen innerhalb dieser Anfrage ausgeführt werden oder der vorherige Stand erhalten bleibt. Die Software-Architektur muss sicherstellen, dass jede Anfrage als eine Transaktion behandelt wird. Darüber hinaus muss auch die eingesetzte Datenbank Transaktionen unterstützen.
A-HV	Die KDB soll als zentrales System im CRM-Verbund und in der Systemlandschaft des Auftraggebers eine Verfügbarkeit von 99,9 % haben.
A-Betr	Die Anforderung, das System soll möglichst einfach betreibbar sein, hat starken Einfluss auf die Software-Architektur. So sind etwa zentrale Bausteine für Logging, Arbeit mit Konfigurationsdateien und Überwachung des Systems vorzusehen.

10.3 Architekturen und Architektur-Disziplinen (WAS)

10.3.1 Disziplinen

Wie weiter oben im Abschnitt WARUM (siehe Abschnitt 10.2) schon ausgeführt, gab es von Projektstart an drei Architektur-Arten, die Berücksichtigung finden mussten:

- > Die Integrationsarchitektur, die bestimmen soll, wie Kundendaten im CRM-Verbund ausgetauscht werden.
- > Die Datenarchitektur, die beschreibt, wie Kundendaten in der Datenbank abgebildet werden.
- > Die Software-Architektur, die Aussagen zu dem Aufbau des IT-Systems macht.

10.3.2 Architektonische Entscheidungen

Die in 10.2 genannten Anforderungen beeinflussen die Architektur der KDB, sodass sich direkt einige architektonische Entscheidungen ableiten lassen.

Entscheidungen ableiten

Die Vielzahl unterschiedlicher Funktionen, welche die KDB haben soll, macht es entsprechend des Architektur-Prinzips *Separation of Concerns* (siehe Abschnitt 6.3) erforderlich, in der Software-Architektur verschiedene Schichten vorzusehen, die Funktionalitäten auf dem gleichen Abstraktionsgrad beinhalten (Abbildung 10.3-1). Hier findet das Architekturmuster *Layers* (siehe Abschnitt 6.3.4) Anwendung. Solche Schichten sind

- > der *Published Services Layer*, der die aufrufbaren Services und die Schnittstelle nach außen realisiert,
- > der *CORE Services Layer*, welcher die Geschäftslogik für die einzelnen Entitäten beinhaltet,
- > der *Persistence Services Layer*, der für jede Entität ein entsprechendes Datenobjekt mit der benötigten Funktionalität zur Persistierung bereitstellt sowie
- > der *Enterprise Layer* für die persistente Speicherung der Daten.

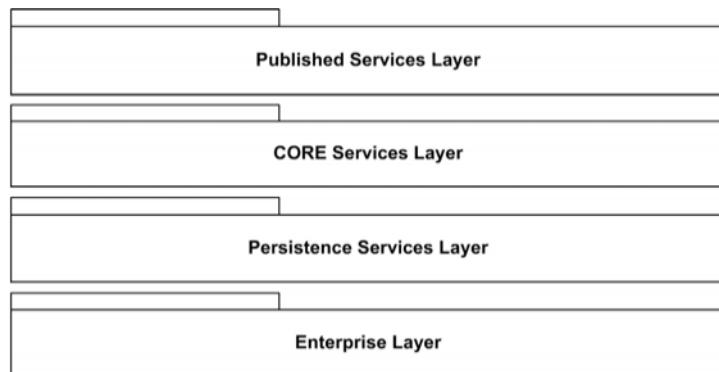


Abb. 10.3-1: Schichten der KDB.

Darüber hinaus lassen sich Schichten getrennt installieren, was eine bessere Skalierbarkeit ermöglicht.

Die durch die einzelnen Schichten bereitzustellende Funktionalität ist wiederum so komplex und vielfältig, dass diese aus Gründen der Wartbarkeit und der Vermeidung von Redundanzen in Subsysteme strukturiert wird. Hier finden unter anderem die Architektur-Prinzipien *Separation of Concerns* und *Modularisierung* Anwendung.

E-Layer

E-Subsystem

tion of Concerns, lose Kopplung (siehe Abschnitt 6.1.1) und hohe Kohäsion Anwendung (siehe Abschnitt 6.1.2), was am Beispiel des *CORE Services Layer* verdeutlicht wird (Abbildung 10.3-2).

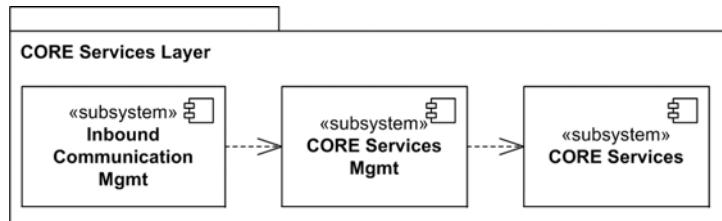


Abb. 10.3-2: Subsysteme des *CORE Services Layer*.

E-Web Service

Die Anforderung nach Reduktion der Schnittstellen führt letztlich dazu, eine generische Schnittstelle für Kundenstammdaten und die entsprechenden Services zu definieren. Auch wenn sich der Auftraggeber für eine strategische Zielplattform entschieden hat, existiert noch eine Vielzahl an Systemen, die nicht auf dieser Plattform basieren. Das Erfordernis, diese generische Schnittstelle in einer heterogenen Systemlandschaft zu etablieren, führt zu der Entscheidung, diese als Web Service auszuführen. Dabei ist der Web-Service das Realisierungsmittel einer serviceorientierten Architektur (SOA, siehe Abschnitt 6.4.11).

E-XML

In welchem Format sollen jedoch die Inhalte transportiert werden? Auch hierbei ist der heterogenen Systemumgebung Rechnung zu tragen. Um allen Systemen Zugang zu den Kundendaten zu geben, wird ein speziell auf diesen Anwendungsfall zugeschnittener XML-Dialekt für den Transport der Kundendaten eingeführt.

E-Adapter

Trotz der Verwendung von Web Services und XML gibt es Systeme, die aus unterschiedlichen Gründen nicht direkt mit der Schnittstelle für Kundendaten kommunizieren können. Für diese Systeme kommen Integrationsbausteine in Form von Adapters zum Einsatz, die zum Umsetzen der XML-Nachrichten in ein für das anzuschließende System verarbeitbares Format über eine systemspezifische Geschäftslogik verfügen. Diese Adapter werden entweder als eigenständige Bausteine oder als Bestandteil des anzuschließenden Systems ausgeführt.

E-PXMLFW

Um der Anforderung nach einfacher Erweiterbarkeit gerecht zu werden, wurde als Abbild des Kundendaten-XML eine Objektstruktur geschaffen, die auf einem mithilfe von Metadaten konfigurierbaren generischen Framework beruht. Das Hinzufügen von Attributen oder Entitäten ist damit sehr einfach möglich.

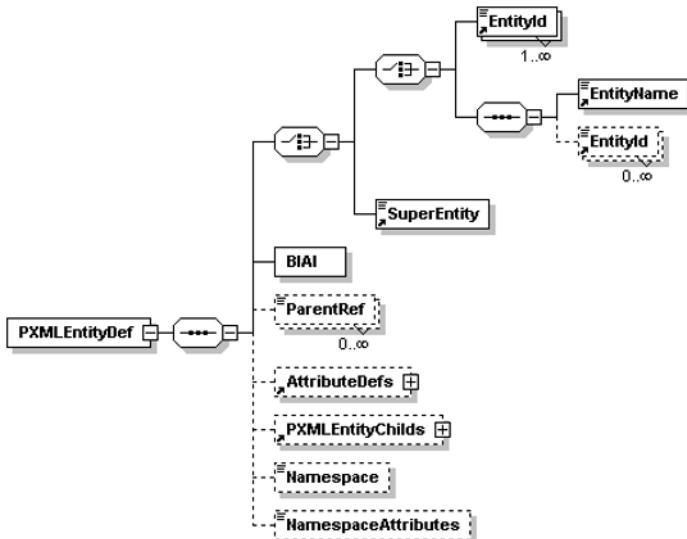


Abb. 10.3-3: XML-Schema PXML-Framework.

Auch das Mapping von durchzuführender Operation (Insert, Update, Delete) auf die Serviceoperation in der Geschäftslogik ist konfigurierbar. Abbildung 10.3-3 zeigt die Struktur der Konfigurationsdatei des PXML-Framework als XML-Schema.

Das gewünschte Antwortzeitverhalten wie auch die erforderliche Skalierbarkeit lassen sich nur realisieren, wenn die Verarbeitungsprozesse parallel abgearbeitet werden. Die für die Verarbeitung erforderliche Logik ist demzufolge so auf Bausteine zu verteilen, dass eine entsprechende Parallelität ermöglicht wird.

Hochverfügbarkeit kann nicht allein durch Software-Architektur sichergestellt werden, jedoch kann diese einen Beitrag dazu leisten. Wirksamstes Mittel zur Erhöhung der Verfügbarkeit ist Redundanz und die Vermeidung von Singularitäten, deren Ausfall den Ausfall des gesamten Systems nach sich zieht. Mit Redundanz auf Bausteinenebene kann die Software-Architektur für eine erhöhte Verfügbarkeit sorgen. Darüber hinaus soll sie sicherstellen, dass die Verteilung von Bausteinen und Subsystemen auf die Hardware ebenso möglich ist wie ein dynamisches Verschieben von Bausteinen. Das alles setzt jedoch voraus, dass der Anwendungsserver solche Mechanismen auch unterstützt. Die Integrationsarchitektur ermöglicht durch Nutzung einer generischen Schnittstelle den Einsatz von Hardware-Loadbalancern, was Redundanz auf Systemebene ermöglicht. Auch dieser Fakt schlägt sich in einer erhöhten Verfügbarkeit nieder. Nicht unerwähnt sollte dabei auch

E-Parallel

E-Redundanz

die Verfügbarkeit der Datenbank bleiben, da diese ein zentraler Bestandteil des Systems ist. Moderne Datenbank-Managementsysteme bieten hier Mechanismen wie Clustering und Parallelbetrieb, die auch in diesem Bereich eine entsprechende Verfügbarkeit sichern.

E-Robust

Robustheit ist die Fähigkeit eines Systems, seltener Vorkommnisse der Umwelt, die in ihren Eigenschaften stark abweichend sind, geschehen lassen zu können, ohne dass der Fortgang des Systems hiervon wesentlich betroffen ist. In der Informatik wird der Begriff Robustheit auch verwendet, um die Eigenschaft eines Verfahrens zu beschreiben, auch unter ungünstigen Bedingungen noch zuverlässig zu funktionieren. Neben der Redundanz ist Robustheit ein weiterer wesentlicher Faktor für eine hohe Verfügbarkeit. Die Software-Architektur hat einen wesentlichen Einfluss auf die zu erreichende Robustheit. Dabei muss jedoch berücksichtigt werden, dass Robustheit und Performanz komplementäre Eigenschaften sind, sodass hier ein sinnvoller Kompromiss gefunden werden muss.

E-Hub

Die KDB soll das zentrale System für Kundendaten werden und für Datenqualität und -konsistenz verantwortlich sein. Diese Anforderungen lassen sich nur mit einer Umgestaltung der bestehenden Integrationsarchitektur erreichen, indem alle relevanten Systeme Kundendaten nur von der KDB abfragen und Veränderungen an Kundendaten an die KDB melden.

E-Propagation

Allein das Melden von Veränderungen an Kundenstammdaten an die KDB reicht nicht aus, um die Konsistenz der Daten über die gesamte Systemlandschaft sicherzustellen. Dazu müssen die relevanten Systeme von Änderungen, die andere Systeme vorgenommen haben, informiert werden. Dieses Verfahren wird auch als *Propagation* bezeichnet. Die KDB soll hierbei die Priorität haben und als *single point of truth* fungieren, sodass die angeschlossenen Systeme Konflikte auflösen können.

E-DBMS

Die Transaktionssicherheit beim Ändern von Daten wird primär vom Datenbank-Managementsystem sichergestellt. Bei der Auswahl des DBMS ist darauf zu achten, dass Transaktionen entsprechend unterstützt werden.

E-Konfig

Die Betreibbarkeit eines Systems wird im Wesentlichen durch zwei Faktoren bestimmt. Einer davon ist die Art und Weise, wie die Konfiguration durchzuführen ist. Für die KDB sind dabei folgende Festlegungen getroffen worden:

- > Verwendung von XML-basierten Konfigurationsdateien.
- > Trennung in maschinenunabhängige und -spezifische Konfiguration.
- > Ähnlicher Aufbau von Konfigurationselementen für ähnliche Aufgaben.

Ein weiterer wichtiger Faktor für die Betreibbarkeit von Systemen ist deren Überwachung mit der Möglichkeit, regulierend eingreifen zu können. Zu diesem Zweck wird eine Konsole für die Überwachung, das Starten und Stoppen von Systembausteinen sowie eine Konsole für die Überwachung der Middleware und der Queues bereitgestellt. Die Struktur der Logdateien ist festgelegt. Es werden einheitliche Markierungen zur Erleichterung der Auswertbarkeit benutzt.

E-Monitor

10.3.3 Entscheidungen zur Software-Architektur

Wie sieht denn nun die Struktur eines Systems aus, das die oben genannten Anforderungen realisiert? Abbildung 10.3-4 zeigt die während des Entwurfs entstandene und in vielen Iterationsschritten verfeinerte Schichtenstruktur mit den wesentlichen Subsystemen und deren Abhängigkeiten.

Schichtenstruktur

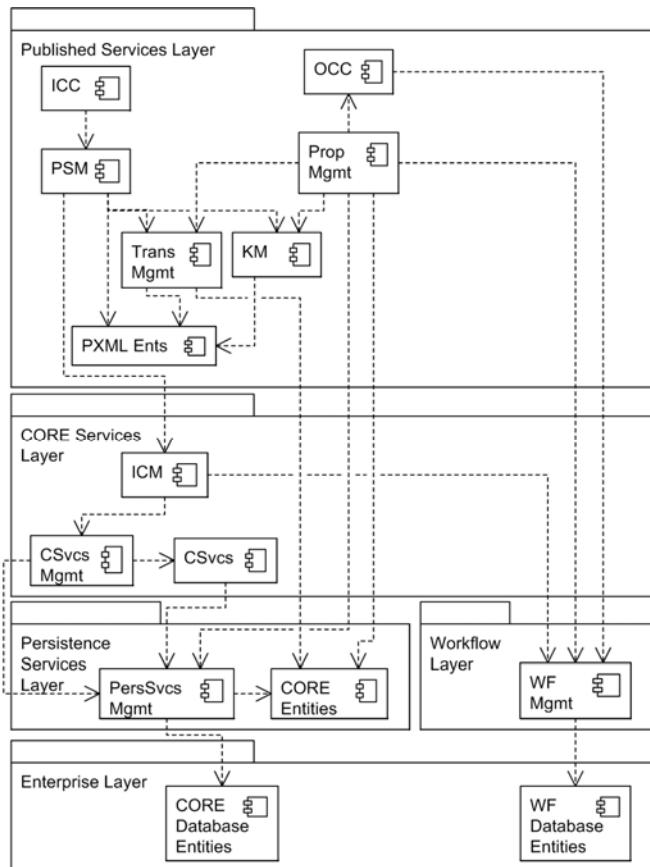


Abb. 10.3-4: Schichtenstruktur der KDB.

Lehre vs. Performanz

Betrachtet man die Schichtenstruktur, fällt auf, dass entgegen der reinen Lehre auch Schichten übersprungen werden. Es kann nicht nur sinnvoll, sondern aus verschiedenen Gründen (z. B. Performanz) sogar zwingend erforderlich sein, von der Lehre abzuweichen, um Ziele zu erreichen. Dabei darf nur nicht vergessen werden, dass eine solche Abweichung oft negative Konsequenzen hat. In dem vorliegenden Fall wurde der Performanz der Vorzug gegeben und Konsequenzen wie schlechtere Wartbarkeit bzw. Erweiterbarkeit bewusst in Kauf genommen.

Published Services Layer

Der *Published Services Layer* stellt alle benötigte Funktionalität für die Kommunikation mit den angeschlossenen Systemen, Prüfung und Aufbereitung der Anfragen und die erforderliche Transformationslogik zur Verfügung. Die Subsysteme in dieser Schicht sind in Tabelle 10.3-1 aufgeführt.

Tab. 10.3-1: Subsysteme der KDB.

Subsystem	Beschreibung
<i>Inbound Communication Controlling (ICC)</i>	Das Subsystem <i>Inbound Communication Controlling</i> stellt Funktionalität für die Behandlung von Anfragen und Antworten über SOAP/HTTP zur Verfügung. Angeschlossene Systeme kommunizieren mit Bausteinen dieses Paketes.
<i>Outbound Communication Controlling (OCC)</i>	Das Verpacken von Änderungsnachrichten in einen SOAP-Umschlag und dessen Versenden an angeschlossene Systeme über HTTP ist Aufgabe des Subsystems <i>Outbound Communication Controlling</i> . Die Änderungsnachrichten werden an die Web-Services-Schnittstelle, die jedes potenzielle Zielsystem bereitstellen muss, geschickt.
<i>Published Services Management (PSM)</i>	Das Subsystem <i>Published Services Management</i> implementiert die sogenannten <i>Published Services</i> (z. B. die <i>Customer</i> und die <i>Central Services</i>) und verarbeitet Anfragen und Antworten protokollneutral. <ul style="list-style-type: none">> Anfragen werden dabei zunächst in eine <i>CORE Services</i> konforme Repräsentation transformiert.> Diese wird an den <i>CORE Service Layer</i> delegiert und dort verarbeitet.> Das Ergebnis der Verarbeitung wird in eine Antwort des aufgerufenen <i>Published Service</i> transformiert und> in einen SOAP-Umschlag verpackt dem anfordern System zugestellt.
<i>Propagation Management (Prop Mgmt)</i>	Das Subsystem <i>Propagation Management</i> realisiert alle erforderliche Funktionalität zum Generieren einer PXML-Änderungsnachricht:

Subsystem	Beschreibung
<i>Propagation Management (Prop Mgmt) (Forts.)</i>	<ul style="list-style-type: none"> > Vervollständigung der Daten, um eine PXML-schemakonforme PXML-Nachricht erzeugen zu können, > zielsystemspezifisches Filtern der Elemente der vervollständigten Kundendaten, > Aufruf der Transformation für das Erzeugen einer PXML-Nachricht auf Basis der vervollständigten und gefilterten Kundendaten sowie > Aufruf des <i>Key Mappings</i> zur Anreicherung der PXML-Nachricht um die Schlüssel des Zielsystems.
<i>Transformation Management (Trans Mgmt)</i>	<p>Das Subsystem <i>Transformation Management</i> realisiert alle erforderlichen Funktionalitäten für</p> <ul style="list-style-type: none"> > die Transformation von <i>Published-Service</i>-Aufrufen in Aufrufe von <i>CORE-Services</i> sowie > die Transformation der Verarbeitungsergebnisse in <i>Published-Service</i>-Antworten.
<i>Key Mapping (KM)</i>	<p>Das Subsystem <i>Key Mapping</i> stellt Funktionalität zur Anreicherung von <i>Published-Services</i>-Anfragen und -Antworten um Schlüssel-Attribute bereit. Dabei werden</p> <ul style="list-style-type: none"> > <i>Published-Services</i>-Anfragen, die nur Schlüssel des anfragenden Systems enthalten, um die benötigten Schlüssel der KDB angereichert und > <i>Published-Services</i>-Antworten um die Schlüssel des anfragenden Systems ergänzt. <p>Die Ergänzung erfolgt natürlich nur dann, wenn auch entsprechende Einträge vorhanden sind.</p>
<i>PXML Entities (PXML Ents)</i>	<p>Die Kommunikation mit den angeschlossenen Systemen erfolgt mittels eines speziell dafür entworfenen XML-Dialektes, des <i>Published-Service</i>-XML (kurz PXML). Das Subsystem <i>PXML-Management</i> stellt eine objektorientierte Repräsentation von PXML-Nachrichten zur Verfügung. Mit seiner Hilfe können PXML-Nachrichten erzeugt und bearbeitet werden. Darüber hinaus erlaubt dieses Paket den Zugriff auf alle Elemente von solchen Nachrichten.</p>

Der *Core Services Layer* bietet granulare Abfrage- und Bearbeitungsfunktionalität für jede Entität der Kundenstammdaten. Diese ist als Operationen auf Entitäten und Transaktionen umgesetzt. Die gesamte Geschäftslogik zur Sicherung der Datenqualität ist mit einer Vielzahl von Bedingungen und Prüfungen auf diesen Operationen implementiert. Die einzelnen Subsysteme sind in Tabelle 10.3-2 mit ihrer Funktion kurz beschrieben.

Core Services Layer

Tab. 10.3-2: Subsysteme des Core Services Layer.

Subsystem	Beschreibung
<i>Inbound Communication Management (ICM)</i>	Das Subsystem <i>Inbound Communications Management</i> nimmt Anfragen in CORE-Services-Repräsentation entgegen und reicht diese an das <i>CORE Service Management</i> weiter. Dabei erfolgt eine Unterscheidung von synchronen und asynchronen Aufrufen und ob eine Anfrage im Zuge der Bereinigung von Fehlersituationen zum wiederholten Mal geschickt wurde. Darüber hinaus generiert dieses Subsystem eindeutige Transaktionsnummern und stellt die Kommunikation mit dem <i>Workflow Layer</i> sicher.
<i>CORE Services Management (CSvcs Mgmt)</i>	Das Subsystem <i>CORE Services Management</i> steuert anhand der Anfrage die Aufrufe der <i>CORE Services</i> und überwacht deren Ausführung. Darüber hinaus übernimmt es die Steuerung der Transaktionen und stellt die Änderungsnachrichten zur Verfügung.
<i>CORE Services (CSvcs)</i>	Die Dienste, die für jede Entität zur Verfügung stehen, werden durch das Subsystem <i>CORE Services</i> bereitgestellt. Darunter befinden sich Dienste für das Anlegen, Modifizieren, Löschen von und die Suche nach Adressen. In diesem Subsystem ist die Geschäftslogik implementiert, die durch die einzelnen Dienste sichergestellt werden muss.

Persistence Services Layer

Der *Persistence Services Layer* stellt die Abstraktionsschicht für den Zugriff auf die Datenbank über die Subsysteme *Persistence Services Management* und *CORE Entities* (siehe Tabelle 10.3-3) dar.

Tab. 10.3-3: Subsysteme des Persistence Services Layer.

Subsystem	Beschreibung
<i>Persistence Services Management (PersSvcs Mgmt)</i>	Das Subsystem <i>Persistence Services Management</i> stellt eine Fassade zur Verfügung, über die alle Zugriffe auf Datenbankobjekte der Kundenstammdaten gesteuert werden.
<i>CORE Entities</i>	Das Subsystem <i>CORE Entities</i> stellt eine objekt-orientierte Repräsentation für alle Entitäten der Kundenstammdaten zur Verfügung.

Workflow Layer

Der Workflow Layer stellt ein konfigurierbares Workflow-Management und Funktionalität zur Persistierung von Aktivitäten in Warteschlangen (englisch: *queues*) zur Verfügung (siehe Tabelle 10.3-4).

Tab. 10.3-4: Subsysteme des Workflow Layer.

Subsystem	Beschreibung
<i>Workflow Management (WF Mgmt)</i>	Das einzige Subsystem im Workflow Layer ist <i>Workflow Management</i> . Es bietet alle Funktionalität, um Aktivitäten in einen Workflow zu stellen und sichert die Abarbeitung der im Workflow definierten Arbeitsschritte. Jeder Aktivität kann verschiedene Daten als Prozessattribute halten. Aktivitäten werden zwischen den Arbeitsschritten in Queues persistiert.

Auf dem *Enterprise Layer* befindet sich ein relationales Datenbank-Managementsystem, welches die Persistenz der Kundenstammdaten sicherstellt. Es beinhaltet zwei Subsysteme, die in Tabelle 10.3-5 kurz beschrieben werden.

Enterprise Layer

Tab. 10.3-5: Subsysteme des Enterprise Layer.

Subsystem	Beschreibung
<i>CORE Database Entities</i>	Die <i>CORE Database Entities</i> sind die relationale Repräsentation der Entitäten der Kundenstammdaten in Form von Datenbank-Tabellen.
<i>Workflow Database Entities (WF Database Entities)</i>	Die <i>Workflow Database Entities</i> sind Datenbank-Tabellen, die für Workflow und Queuing benötigt werden.

10.4 Architektur-Perspektiven (WO)

Innerhalb des dieser Fallstudie zugrunde liegenden Projektes wurde auf unterschiedlichen Architektur-Ebenen operiert.

Die Operationen auf der Organisationsebene wie

- > die Identifikation der Geschäftsprozesse und der damit verbundenen Geschäftsobjekte,
- > die Identifikation der betroffenen IT-Systeme und
- > die Definition der Verantwortlichkeiten der IT-Systeme

waren bereits im Vorfeld durch den Auftraggeber abgeschlossen.

Organisationsebene

Auf der Systemebene waren die funktionalen Anforderungen für jedes zu integrierende System abzuleiten und zu konkretisieren. Basierend darauf mussten die Schnittstellen für die benötigte Funktionalität definiert und hinsichtlich ihres Funktionsumfangs, ihrer Technologie, der

Systemebene

verarbeiteten Datenstrukturen und -formate sowie ihres Kommunikationsstils untersucht werden.

Bausteinebene

Auf der Bausteinebene wurden Vorgaben bezüglich der Struktur und der Verteilung der Aufgaben unter den Bausteinen erarbeitet. Um eine Durchgängigkeit bei der Lösung von Kernproblemen zu gewährleisten und Doppelarbeit zu vermeiden, wurden systemübergreifende Frameworks geschaffen. Durch die Vorgabe von Implementierungs-Standards, die teilweise in der Form von Code-Mustern vorgegeben waren, wurden architektonische Richtlinien aufgestellt. Darüber hinaus gab es Vorgaben hinsichtlich der Namenskonventionen und Ableitungshierarchie für Bausteine.

10.5 Architektur-Mittel (WOMIT)

Werkzeuge

Das Projekt CRM-KDB hat in seiner Laufzeit eine Reihe von Änderungen hinsichtlich der Verantwortlichen wie auch der Beteiligten erfahren. Verbunden mit diesen Änderungen hat auch ein Wandel der Architektur-Mittel stattgefunden. Während zu Beginn in erster Linie auf die Unterstützung der Datenmodellierung fokussiert wurde, rückte später der gesamtheitliche Modellierungsaspekt in den Vordergrund und es wurden UML-Modellierungswerkzeuge eingesetzt.

Neben den Werkzeugen fanden auch Architektur-Prinzipien und -Muster wie *Separation of Concerns* und das Schichten-Architektur-Muster (*Layers*) Anwendung.

Dokumentation

Die Dokumentation der entstandenen Architektur erfolgte mittels UML, um eine konsistente Notation für die Beschreibung aller wichtigen Aspekte zu benutzen.

Technologien

Als Technologien kamen Datenbanken, Middleware, XML und Web Services zum Einsatz.

Plattform

Die eingesetzte Plattform beinhaltete eine integrierte Entwicklungsumgebung sowie ein einfaches Versionierungssystem. Darüber hinaus kamen ein Defekt-Verfolgungssystem und viele verschiedene Test-Werkzeuge zum Einsatz. Das Build-System wurde selbst entwickelt, da das mit der Entwicklungsumgebung ausgelieferte Toolkit nicht flexibel genug war und aus der sequentiellen Abarbeitung sehr lange Übersetzungszeiten resultierten.

10.6 Organisationen und Individuen (WER)

Auftraggeber für das gesamte CRM-Programm - und damit auch der KDB als dessen integraler Bestandteil - ist eine bedeutende europäische Fluggesellschaft.

Auftraggeber

Die Dienste der zentralen KDB stehen jedem Bereich des Auftraggebers zur Verfügung, der Zugriff auf Kundendaten benötigt. Das sind in erster Linie Bereiche, die in das CRM-Programm eingebunden sind, sowie Bereiche, die Dienstleistungen im Zusammenhang mit Kundenkontaktpunkten (Check-in, Lost-and-found, Call-Center, ...) anbieten.

Nutzer der KDB

Anwendungen mit Zugriff auf diese Services werden von den Geschäftsbereichen an Generalunternehmer innerhalb oder außerhalb des Konzerns in Auftrag gegeben und unter Mitwirkung Externer realisiert.

Der Generalunternehmer des dieser Fallstudie zugrunde liegenden Projektes war ein international agierendes Beratungsunternehmen, das Leistungen, die es nicht selbst erbringen konnte, bei Unterauftragnehmern einkaufte.

Generalunternehmer

Die fünf wesentlichen Rollen, die der Auftraggeber im Rahmen des Aufbaus sowie des Betriebs der zentralen KDB unterschied, waren:

- *Betreiber der anzuschließenden Systeme und Datenquellen:* Diese Gruppe ist zuständig für den Betrieb des jeweiligen Systems, arbeitete bei der Erstellung der entsprechenden Schnittstellendefinition mit und setzte sich aus Mitarbeitern des Auftraggebers und Externen zusammen.
- *Technischer Betreiber CRM-Kern:* Die Gruppe, die für die technische Infrastruktur der zentralen CRM-Anwendungen zuständig ist. Darunter sind im Wesentlichen Hardware, Netzwerk, Betriebssystem sowie entsprechende System-Management und Dienstleistungen wie Support zu verstehen. Für diese Aufgabe hat der Auftraggeber einen Dienstleister aus dem eigenen Konzern ausgewählt.
- *Fachlicher Betreiber CRM-Kern:* Die Gruppe, die fachliche Fragen rund um CRM und dessen Zusammenspiel klären und ihre Umsetzung initiieren soll. Mitglieder dieser Gruppe sind Mitarbeiter des Auftraggebers und Externe.
- *Anwender:* Diese Gruppe arbeitet mit den Daten aus dem CRM-Verbund und ist daher über das ganze Unternehmen des Auftraggebers verteilt.

Rollen und deren Beziehungen

- > *Projektteam*: Diese Gruppe hatte den Entwurf und die Implementierung des IT-Systems KDB als zentrale Aufgabe. Mitglieder dieser Gruppe waren Externe.

Die Beziehungen der einzelnen Rollen untereinander sind in Abbildung 10.6-1 dargestellt.

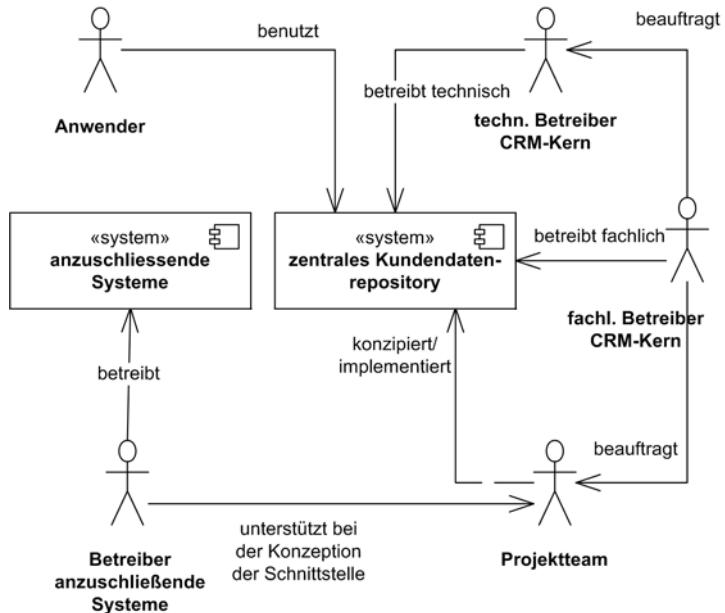


Abb. 10.6-1: Rollen und deren Beziehungen.

10.7 Architektur-Vorgehen (WIE)

Wasserfall

Das Projekt wurde nach einem vom Auftraggeber eingesetzten Vorgehensmodell umgesetzt, das sich stark am Wasserfallmodell (siehe Abschnitte 8.1 und 8.2) orientiert.

Dabei wurden alle Architektur-Tätigkeiten wie

- > Erstellen der Systemvision,
 - > Verstehen der Anforderungen,
 - > Entwerfen der Architektur,
 - > Umsetzen der Architektur und
 - > Kommunizieren der Architektur
- durchgeführt.

Für Teilaktivitäten, wie z. B. der Entwurf und die Implementierung einzelner Bausteine, kam eine iterative Vorgehensweise zum Einsatz.

Iterativ

10.8 Fazit

Im Verlauf des Projekts, auf dem diese Fallstudie basiert, wurden die Anforderungen, deren architektonische Relevanz, die Anwendungsfälle und neben der Software-Architektur auch weitere (Datenarchitektur, Integrationsarchitektur) erarbeitet und dokumentiert. Die Ausführung dieser Architektur-Tätigkeiten erfordert einen Großteil des in diesem Buch beschriebenen Wissens und ist aufwändig. Lohnt sich dieser Aufwand?

Lohnt sich der Aufwand?

Ein wesentlicher Bestandteil des Wissens ist, wie sinnvoll in Projekten vorgegangen wird, um die Ziele zu erreichen. Dazu gehört, welche Tätigkeit in welcher Reihenfolge auszuführen ist, was während der Anforderungsanalyse zu erfragen ist und wie Ergebnisse der verschiedenen Tätigkeiten dokumentiert werden können. Letztlich ergibt sich ein Leitfaden, anhand dessen sich alle Projektbeteiligten orientieren können, wo sie gerade im Projektverlauf stehen und was noch zu tun ist. Die Orientierung an diesem Leitfaden hat in diesem Projekt sehr dazu beigetragen, dass in einem großen Team effizient zusammengearbeitet werden konnte und dass Zeit- und Kostenaufwand in einem vernünftigen Rahmen blieben.

Vorgehen

Aus dem Wissen ergibt sich auch, welche Ergebnisse wie und am besten zu dokumentieren sind, um verständlich für alle Projektbeteiligten und möglichst vollständig zu sein. So ist es auch ohne das Architektur-Wissen möglich, sich in den Ergebnisdokumenten zu orientieren und diese zu verstehen.

Struktur

Architektur hilft also bei der Strukturierung von Projekten und Dokumenten und hat im vorliegenden Fall zu einem gemeinsamen Verständnis bei allen Projektbeteiligten geführt und war dementsprechend ein wichtiges Mittel zur Kommunikation sowohl zum Auftraggeber als auch zum Projektteam.

Mittel zur Kommunikation

Da die Ergebnisdokumente in der oben genannten Struktur erstellt wurden, beschreiben sie alle relevanten Aspekte der Problematik und deren Lösung. Insbesondere die Beschreibung der Lösung ist ein Maß, an dem die Realisierung gemessen werden kann. Reviews über die gesamte Projektlaufzeit können sicherstellen, dass

- > die Problematik richtig verstanden wurde,
- > die relevanten Anforderungen korrekt aufgenommen wurden,
- > eine tragbare Architektur als Lösung aufgezeigt wurde und
- > die Realisierung entsprechend der Architektur erfolgt ist.

Erfahrung

Neben dem Wissen spielt natürlich auch Erfahrung eine wesentliche Rolle, wenn es darum geht, die richtigen Fragen zu stellen und eine tragfähige Lösung zu entwerfen und umzusetzen. Das Wissen darum, welche Fähigkeiten gebraucht werden, befähigt dazu, die richtige Zusammensetzung für das Projektteam zu finden.

Wesentlicher Erfolgsfaktor

Software-Architektur ist folglich keinesfalls eine rein technische Disziplin, sondern steht im engen Zusammenhang mit Projektmanagement und Kommunikation. Software-Architektur ist bei richtiger Anwendung ein wesentlicher Beitrag für den Projekterfolg.

| Glossar

Begriff (Synonym)	Erklärung
.NET	.NET ist eine Komponentenplattform von Microsoft.
2-Tier-Architektur	Eine 2-Tier-Architektur besteht aus zwei grundlegenden Tiers. Das klassische Client-/Server-Modell basiert z. B. auf einer 2-Tier-Architektur.
3-Tier-Architektur	Eine 3-Tier-Architektur erweitert die 2-Tier-Architektur, indem sie einen Zwischentier (englisch: <i>intermediate tier</i>) zwischen Client und Datenbanksystem respektive Enterprise Information System einführt.
4+1-Sichtenmodell	Das 4+1-Sichtenmodell ist ein von Philippe Kruchten im Umfeld des USDP (Unified Software Development Prozess) entwickeltes Architektur-Sichtenmodell. Es definiert sechs in der Praxis häufig benötigte Architektur-Sichten.
Abstraktion	Abstraktion bedeutet, ein komplexes Problem dadurch besser zu verstehen, dass man wichtige Aspekte identifiziert und die unwichtigen Details vernachlässigt.
Abstraktionsprinzip	Ein Abstraktionsprinzip wendet Abstraktion an. Es hilft, ein komplexes Problem dadurch besser zu verstehen, indem wichtige Aspekte identifiziert und die unwichtigen Aspekte vernachlässigt werden.
Agiles Vorgehensmodell	Ein agiles Vorgehensmodell basiert auf einem iterativ-inkrementellen Vorgehen und ist bestrebt, unnötige Tätigkeiten auf ein Minimum zu reduzieren (z. B. die Erstellung überflüssiger Dokumentation). Beispiele sind XP, Scrum, FDD.
Anforderung	Eine Anforderung ist eine vom Anwender benötigte Fähigkeit (englisch: <i>capability</i>) des Systems, um ein Problem zu lösen oder ein Ziel zu erreichen, respektive eine Fähigkeit, die das System besitzen muss, damit es einen Vertrag, einen Standard, eine Spezifikation oder ein anderes formelles Dokument erfüllt.
Anwendungsbaustein	Ein Anwendungsbaustein kapselt Anwendungslogik und stellt diese zur Verfügung.
Anwendungslogik	Anwendungslogik ist Logik, die eng mit der zu realisierenden Anwendung in Beziehung steht und nur schwer über Anwendungsgrenzen hinweg wiederverwendet werden kann.
Apollo-Teams	Apollo-Teams sind Gruppen, die aus hochintelligenten, analytischen und mentalstarken

Begriff (Synonym)	Erklärung
	Menschen bestehen und in der Regel bei der Zielerreichung schlechter abschneiden als heterogene Teams.
Arbeitsphase	Die Arbeitsphase ist die vierte Phase der Gruppenbildung nach Tuckmann. Aufgrund des entwickelten Wir-Gefühls erfolgt eine zielorientierte, gemeinschaftliche Zusammenarbeit. Zu diesem Zeitpunkt hat sich die Gruppe eingespielt und ihr Leistungsvermögen erreicht.
Architecture Description Langauge (ADL)	Eine Architecture Description Langauge ist eine spezielle DSL und dient der präzisen Beschreibung der Architektur von Systemen.
Architektonischer Durchstich	Ein Architektonischer Durchstich ist ein Architektur-Prototyp, mit dem sämtliche relevanten Systembausteine (z. B. für Benutzeroberfläche bis hin zur Persistenz) zum erfolgreichen Zusammenspiel gebracht wurden.
Architektonischer Ordnungsrahmen	Der architektonische Ordnungsrahmen ist ein Rahmen, in den architektonisches Wissen und Erfahrung eingebettet werden kann. Er ist nach verschiedenen Architektur-Dimensionen strukturiert.
Architektonischer Setzkasten	Der architektonische Setzkasten ist die in diesem Buch verwendete Metapher zur Strukturierung der Architektur-Domäne.
Architektonisches Bewusstsein	Das architektonische Bewusstsein ist eine Geisteshaltung, die die Wichtigkeit von Architektur erkennt und zu einem architektonischen Handeln führt. Die Güte dieses Bewusstseins ist von strategischer und langfristiger Relevanz, da ein architektonisches Bewusstsein als Grundlage für ein lebenslanges Lernen und somit für ein erfolgreiches Handeln angesehen werden kann.
Architektur-Aktion (Architektur-Schritt)	Eine Architektur-Aktion ist eine Aktion innerhalb einer Architektur-Tätigkeit.
Architektur-Dimension	Eine Architektur-Dimension ist eine Kategorie des architektonischen Ordnungsrahmens, welche zur Strukturierung der Architektur-Domäne dient und einen Teilbereich abdeckt. Jeder Architektur-Dimension ist ein Fragewort zugewiesen.
Architektur-Disziplin	Eine Architektur-Disziplin befasst sich mit den architektonischen Tätigkeiten und den damit verbundenen Entscheidungen zum Entwurf und zur Umsetzung einer Architektur. Es existieren unterschiedliche Arten von Architektur-Disziplinen.
Architektur-Dokumentation	Eine Architektur-Dokumentation umfasst alle Artefakte, die eine Architektur beschreiben und ist im Wesentlichen strukturiert nach den

Begriff (Synonym)	Erklärung
	Themen Architektur-Entscheidungen, Architektur-Sichten, sichtenübergreifende Aspekte, Architektur-Bewertung, Projektaspekte, offene Punkte und Glossar.
Architektur-Ebene	Eine Architektur-Ebene umfasst architekturelle relevante Elemente, die sich auf der gleichen Abstraktionsstufe befinden.
Architektur-Entscheidung	Eine Architektur-Entscheidung ist eine Entscheidung mit strategischem Charakter und Einfluss auf eine Architektur.
Architektur-Mittel	Architektur-Mittel dienen einem Architekten dazu, eine Architektur zu entwerfen und umzusetzen. Das Spektrum der möglichen Architektur-Mittel reicht dabei von elementaren Prinzipien bis hin zu konkreten Technologien.
Architektur-Muster (Muster)	Ein Architektur-Muster ist eine dreiteilige Regel, die die Beziehung zwischen einem bestimmten Kontext, einem bestimmten System an Kräften, die in diesem Kontext wiederkehrend auftreten, und einer bestimmten Baustein-Konfiguration ausdrückt, die diesen Kräften erlaubt, sich gegenseitig aufzulösen, ausdrückt.
Architektur-Prinzip	Ein Architektur-Prinzip ist ein bewährter und erprobter Grundsatz, der bei der Gestaltung einer Architektur zur Anwendung kommen sollte.
Architektur-Prototyp	<p>Ein Architektur-Prototyp dient grundsätzlich dazu, Risiken zu untersuchung, indem eine lauffähige Implementierung für konkrete Problemstellungen erstellt wird, die ein besseres Verständnis der Risiken ermöglichen.</p> <p>Im Rahmen der Tätigkeit „Verstehen der Anforderungen“ reflektieren Prototypen noch nicht die eigentliche Architektur, sondern werden dazu eingesetzt, um ein besseres Verständnis für die einzelnen architekturellen Anforderungen zu erhalten und deren Machbarkeit zu prüfen (beispielsweise um neue Technologien zu untersuchen).</p> <p>Im Rahmen der Tätigkeit „Entwerfen der Architektur“ nehmen Prototypen Bezug zur eigentlichen Architektur und werden eingesetzt zur Beurteilung von Architektur-Alternativen oder Aspekten einer Architektur.</p>
Architektur-Rahmenwerk	Ein Architektur-Rahmenwerk ist eine Sammlung von Standards, Festlegungen, Richtlinien, Best-Practices, Methoden und Referenzmodellen sowie Architektur-Sichtenmodellen. Architektur-Rahmenwerke, wie das Zachman-Framework, RM-ODP und TOGAF, adressieren die Enterprise-Architektur eines Unternehmens und unterscheiden sich in ihrem konkre-

Begriff (Synonym)	Erklärung
	ten Umfang sowie ihrer konkreten Ausgestaltung.
Architekturrelevante Anforderung	Eine architekturrelevante Anforderung ist eine Anforderung mit einem hohen Nutzen für die Interessenvertreter, mit einem hohen Umsetzungsrisiko oder einer großen Auswirkung auf die Architektur.
Architektur-Richtlinie	Eine Architektur-Richtlinie ist eine Vorgabe, mit der Zielsetzung, dass Architekturentscheidungen beachtet werden und eine Architektur damit korrekt entworfen und umgesetzt wird.
Architektur-Sicht (Sicht, englisch: view)	Eine Architektur-Sicht zeigt ein System aus dem Blickwinkel einer Menge von zusammenhängenden Interessen.
Architektur-Sichtenmodell	Ein Architektur-Sichtenmodell definiert Architektur-Sichten und legt deren Inhalte fest.
Architektur-Stil (Stil)	Ein Architektur-Stil ist eine spezielle Art von Architektur-Muster. Stile drücken insbesondere die strukturelle Organisation einer Familie von Systemen aus.
Architektur-Tätigkeit	Eine Architektur-Tätigkeit ist eine innerhalb eines Architektur-Vorgehens ausgeübte Tätigkeit.
Architektur-Vision	Eine Architektur-Vision ist eine erste grobe Dekomposition eines Systems. Sie ist in der Regel unscharf und wird später beim Entwerfen der Architektur weiter konkretisiert. Die Darstellung ist in der Regel informell.
Architektur-Vorgehen	Ein Architektur-Vorgehen definiert das Vorgehen zum Entwurf und zur Umsetzung einer Architektur.
Aspektorientierung (AOP, AO)	Die Aspektorientierung vermeidet über den Quelltext oder den Entwurf verstreute Lösungen für sogenannte Crosscutting Concerns. Stattdessen werden solche Lösungen in einem Aspekt gekapselt und somit von dem durch den Aspekt betroffenen System separiert.
Ausführungsumgebung	Eine Ausführungsumgebung ist ein Software-Baustein einer Plattform, der Software-Bausteinen eines Systems Dienste bereitstellt. Ein JEE-Applikationsserver bietet beispielsweise Ausführungsumgebungen für JEE-Bausteine, wie Java Servlets oder Enterprise Java Beans.
Basisdienstbaustein	Ein Basisdienstbaustein stellt Basisdienste zur Verfügung (z. B. Logging, Referenzdatenverwaltung).
Bausteinebene	Auf der Bausteinebene im Bereich Makro-Architektur befinden sich architekturrelevante Systembausteine. Auf der Bausteinebene im

Begriff (Synonym)	Erklärung
	Bereich Mikro-Architektur befinden sich hingen- gen architekturirrelevante Systembausteine.
Bedienbarkeit	Bedienbarkeit (englisch: <i>usability</i>) eines Sys- tems drückt sich in erster Linie in der Gestal- tung der Benutzeroberfläche aus und steht nicht im unmittelbaren Zusammenhang mit der Software-Architektur eines Systems.
Bewerter	Ein Bewerter ist ein disziplinierter und hart arbeitender Mensch, der Problemlösungen pragmatisch angeht. Jedoch kann er sich nicht schnell auf sich ändernde Situationen einstel- len und ungeprüfte Ideen akzeptieren.
Business Case	Ein Business Case besitzt Ähnlichkeiten mit einer Systemvision. Ein Business Case beleuch- tet in der Regel den wirtschaftlichen Nutzen stärker als eine Systemvision. Manchmal spricht man statt von einer Systemvision auch von einem Business Case. Der Übergang ist allerdings fließend.
Checkliste	Eine Checkliste besteht aus einer Liste detail- lierer Fragen, welche die Umsetzung der un- terschiedlichen Anforderungen widerspiegelt. Checklisten können für verschiedene Beurtei- lungsmethoden eingesetzt werden.
Conways Gesetz (eng- lisch: <i>conway's law</i>)	Conways Gesetz besagt, dass Organisationen darauf beschränkt sind, Systeme zu entwerfen, deren Strukturen Kopien ihrer organisatori- schen Kommunikationsstrukturen sind.
CORBA-Component- Model	Das CORBA-Component-Model ist eine Spezifi- kation einer Komponentenplattform der OMG.
Datenarchitektur	Datenarchitektur umfasst die datenorientier- ten Aspekte eines Systems. Der Entwurf logi- scher und physischer Datenmodelle, die Aus- wahl von Persistenzmechanismen (z. B. Da- tenbank oder Dateisystem), die Konfiguration einer Datenbank oder der Entwurf eines Data Warehouse sind Tätigkeiten dieser Disziplin.
Datenflussarchitektur (Pipes and Filters, Batch Sequential)	Eine Datenflussarchitektur strukturiert eine Architektur entlang der Datenflüsse und macht insbesondere dann Sinn, wenn eine komplexe Aufgabe in eine Reihe einfacher Aufgaben zer- teilt und dann als Kombination unabhängiger Aufrufe dargestellt werden kann.
Datensicht	Die Datensicht beschreibt Aspekte bezüglich Speicherung, Manipulation, Verwaltung und Verteilung von Daten.
Datentransferobjekt (DTO)	Ein Datentransferobjekt ist ein Objekt, das die Anzahl der notwendigen Client/Server- Interaktionen minimieren und zum anderen Klienten von der konkreten Repräsentation von Entitätsobjekten entkoppeln soll.

Begriff (Synonym)	Erklärung
Dezentralisierung	Die Dezentralisierung bezeichnet die Verteilung eines Belangs auf mehrere Systembausteine. Meist muss eine Abwägung zwischen der Zentralisierung und der Dezentralisierung getroffen werden.
Dialogbaustein	Ein Dialogbaustein entspricht dem View-Baustein innerhalb des Model-View-Controller-Architektur-Musters.
Dialogsteuerungsbaustein	Ein Dialogsteuerungsbaustein entspricht dem Controller-Baustein innerhalb des Model-View-Controller-Architektur-Musters. Er nutzt Dienste von Anwendungsbausteinen.
Dienst (Plattformdienst)	Ein Dienst ist ein Software-Baustein, der Basisfunktionalität anbietet, die in der Regel unabhängig von jeglicher durch das System realisierten Geschäftsfunktionalität ist. Mit anderen Worten bietet ein Dienst Funktionalität zur Befriedigung nicht-funktionaler Anforderungen. Ein Dienst kann ebenfalls ein Stück Funktionalität sein, das über die Schnittstelle eines Bausteins angeboten wird.
Domänenbaustein	Ein Domänenbaustein kapselt Domänenlogik und stellt diese zur Verfügung.
Domänenlogik	Domänenlogik repräsentiert Fachlichkeit, die über Anwendungsgrenzen hinweg wiederverwendet werden kann. Sie ist somit anwendungsneutral und operiert allein auf fachlichen Abstraktionen.
Domänspezifische Sprache (DSL)	Eine domänspezifische Sprache ist, wie die UML, eine spezielle Modellierungssprache. Durch eine DSL lassen sich die relevanten Konzepte einer spezifischen, fachlich oder technisch motivierten Domäne präzise beschreiben.
Durchsetzer	Ein Durchsetzer besitzt eine dynamische Persönlichkeit, ist willensstark und in der Lage, Entscheidungen durchzusetzen. Er ist jedoch auch erregbar und neigt zur Provokation.
Dynamische Sprachen	Eine dynamische Sprache ist eine Sprache auf hohem Abstraktionsniveau, die während der Laufzeit viele Aufgaben ausführt, die andere Sprachen zu Kompilierzeit durchführen.
Emergenz	Emergenz besagt, dass ein System Eigenschaften besitzt, die es von seinen Systembausteinen unterscheidet. Demnach weist kein Systembaustein diese Eigenschaften auf, sondern diese entstehen erst durch das Zusammenspiel der einzelnen Systembausteine.
Enterprise-Architektur (Unternehmensarchitektur)	Enterprise-Architektur (deutsch: Unternehmensarchitektur) ist eine Disziplin, die unter Berücksichtigung von Geschäftsstrategien, -prozessen und -daten eine unternehmensweite IT-Architektur entwirft.

Begriff (Synonym)	Erklärung
Entitätsobjekt	Ein Entitätsobjekt ist eine Abstraktion, welche eine eigene Identität innerhalb einer Domäne besitzt. Beispiele sind Kunde, Auftrag und Produkt.
Entscheidung	Eine Entscheidung ist die Wahl einer Handlungs- oder Reaktionsmöglichkeit in einer Situation, in der mehrere Möglichkeiten bestehen. Sie ist ein Schritt im Rahmen einer Problemlösung, bei dem nach der Bewertung von Handlungsalternativen eine Alternative ausgewählt wird.
Entwerfen der Architektur	Das Entwerfen der Architektur ist die Architektur-Tätigkeit, bei der die eigentliche Architektur entsteht. Hierzu bedient man sich eines breiten Fundus an Architekturmitteln und trifft Entscheidungen. Aus der Summe aller Entscheidungen resultiert schließlich die Architektur des Systems.
Entwicklungsprozess	Ein Entwicklungsprozess strukturiert einzelne Disziplinen, Aufgaben und Tätigkeiten und bringt sie in eine zeitliche Abfolge.
Entwicklungszeit-anforderung	Eine Entwicklungszeitanforderung ist eine Qualität oder Rahmenbedingung, die bei der Entwicklung eines Systems schwerpunktmäßig berücksichtigt werden muss.
Entwurf (englisch: <i>design</i>)	Entwurf umfasst den Prozess, um Architektur, Bausteine, Schnittstellen und andere Eigenschaften eines Systems oder eines Systembausteins festzulegen sowie das Ergebnis dieses Prozesses selber. Abhängig vom Detaillierungsgrad der Systembausteine wird beim Entwurf zwischen Makro- und Mikro-Architektur unterschieden.
Erstellen der Systemvision	Das Erstellen der Systemvision ist die Architektur-Tätigkeit, bei der sichergestellt wird, dass die Systemvision architektonisch machbar ist.
Erweiterbarkeit	Erweiterbarkeit ist eine Entwicklungszeitanforderung, die besagt, dass ein System um neue Funktionalität erweiterbar sein muss. Die Erweiterbarkeit eines Systems ist umso besser, je geringer die Kopplung zwischen Systembausteinen ist.
Explizite Schnittstelle	Eine explizite Schnittstelle ist losgelöst vom eigentlichen Systembaustein. Das Konzept der expliziten Schnittstelle wird beispielsweise durch Technologien wie Enterprise Java Beans oder Web Services umgesetzt.
eXtreme Programming (XP)	eXtreme Programming ist ein agiles Vorgehensmodell, Quelltext und Auftraggeber sowie die flexible und zeitnahe Umsetzung von Anforderungen stehen im Mittelpunkt.

Begriff (Synonym)	Erklärung
Fachliche Architektur	Eine fachliche Architektur widmet sich der Realisierung von funktionalen Anforderungen und definiert fachliche Bausteine.
Fachlicher Baustein	Ein fachlicher Baustein befasst sich mit den Schlüsselabstraktionen einer Domäne und den damit verbundenen funktionalen Anforderungen.
Feature Driven Development (FDD)	Feature Driven Development ist ein agiles Vorgehensmodell, das eine Sammlung Industrie-bekannter, bewährter Praktiken umfasst. Die zeitnahe Umsetzung von geforderten Eigenschaften (englisch: <i>feature</i>) in kurzen Iterationen steht im Mittelpunkt.
Formales Review	Ein formales Review wird von einer Gruppe von Interessenvertretern durchgeführt. Diese analysiert die Architektur-Dokumentation systematisch, kommentiert formal und beschließt ggf. Aktionen zur Behebung von Mängeln.
Funktion (Abfrage, englisch: <i>queries</i>)	Eine Funktion ist eine spezielle Art von Dienst, die über eine Schnittstelle angeboten wird. Sie hat keine Seiteneffekte zur Folge. Sie ist einfacher zu testen und weniger risikoreich als ein Kommando. Sie wird auch als idempotente Funktion bezeichnet.
Funktionale Anforderung	Eine funktionale Anforderung definiert benötigte Funktionalitäten.
Funktionale Baustein-anforderung	Eine funktionale Bausteinanforderung ist eine funktionale Anforderung, die ein Baustein eines Systems erfüllen muss, damit das System in der Lage ist, seine Anforderungen zu erfüllen.
Funktionale Organisationsanforderung	Eine funktionale Organisationsanforderung ist eine funktionale Anforderungen, die an Organisationen z. B. von deren Kunden, Mitarbeitern, Geschäftspartnern oder von Behörden gestellt wird.
Funktionale System-anforderung	Eine funktionale Systemanforderung drückt konkrete funktionale Bedürfnisse von Interessenvertretern bzw. von Systemen aus, die mit dem betrachteten System interagieren.
Generierung	Unter Generierung versteht man die generative Erzeugung von Systembausteinen. Sie verfolgt das Ziel, den Automatisierungsgrad bei der Erstellung von Software zu erhöhen.
Geschäfts chance	Eine Geschäftschance ist ein Nutzen für eine Organisation, die durch die Realisierung des Systems verwirklicht werden soll. Sie ist ein wesentlicher Bestandteil einer Systemvision. Der Nutzen des Systems und die Probleme, die das System löst, gehören zu den Geschäftschancen. Der Nutzen wird nicht nur qualitativ, sondern auch quantitativ in Form von Geschäftskennzahlen fassbar gemacht.

Begriff (Synonym)	Erklärung
Geschlossenes System	Ein geschlossenes System tauscht mit seiner Umwelt keine Informationen aus, jedoch steht es mit ihr in einer energetischen Beziehung.
Gründungsphase	Die Gründungsphase ist die erste Phase der Gruppenbildung nach Tuckmann. Die Gruppenmitglieder lernen sich kennen und schätzen einander ein. Es kommt hier bereits zu einer Einordnung der einzelnen Mitglieder. Des Weiteren grenzt sich die Gruppe gegenüber ihrer Umwelt ab.
Gruppe	Eine Gruppe kann als eine besondere Form einer Organisation angesehen werden. Sie interagiert mit ihrer Umwelt, verfolgt eine Aufgabe, besitzt eine Struktur und entwickelt eine Kultur.
Gruppendynamik	Die Gruppendynamik bezeichnet die dynamischen Aspekte der Gruppenbildung.
Holismus	Der Holismus betrachtet ein System in seiner Gesamtheit. Es erfolgt eine Konzentration auf die emergenten Systemeigenschaften, die durch die Interaktion der Systembausteine entstehen.
Ideengeber	Ein Ideengeber geht unkonventionelle Wege und kann aufgrund seines Wissens und seiner Vorstellungskraft zu Lösungen beitragen. Allerdings neigt er auch dazu, Vorschriften zu übersehen und in den Wolken zu schweben.
Implizite Schnittstelle	Eine implizite Schnittstelle ist ein direkter Bestandteil eines Software-Bausteins. Ein C-Modul ist ein Beispiel für einen Software-Baustein, der über eine implizite Schnittstelle verfügt.
Information-Hiding-Prinzip	Das Information-Hiding-Prinzip besagt, dass man einem Klienten nur den wirklich notwendigen Teilausschnitt der gesamten Information zeigt, der für die Aufgabe des Klienten gebraucht wird und alle restliche Information verbirgt.
Inkrement	Ein Inkrement ist ein Arbeitsergebnis, welches am Ende einer Iteration entsteht. Es ist in der Regel ein lauffähiges Stück Software.
Integrationsarchitektur	Integrationsarchitektur beschäftigt sich mit der Planung und Realisierung von integrativen Lösungen mit dem Ziel, mehrere Applikationen oder Systeme eines oder mehrerer Unternehmen miteinander zu verbinden.
Integrationsbaustein	Ein Integrationsbaustein kapselt Integrationslogik (z. B. Datenbank, SAP, LDAP) und stellt diese zur Verfügung.
Integrationslogik	Integrationslogik verkörpert Logik zur Anbindung von Enterprise-Systemen. Ein einfaches

Begriff (Synonym)	Erklärung
	Beispiel ist Logik zum Zugriff auf Datenbanken.
Interesse (Belang, englisch: <i>concern</i>)	Unter Interesse (von lateinisch <i>interesse</i> : dabei sein, teilnehmen an, "dazwischenstecken/sein") versteht man die kognitive Anteilnahme respektive die Aufmerksamkeit, die eine Person an einer Sache oder einer anderen Person nimmt [Wikipedia 2008b]. Bei einem Interesse kann es sich unter anderem um eine Anforderung, eine Fragestellungen, eine Sorge, ein Ziel oder einen Wunsch handeln.
Interessenvertreter	Ein Interessenvertreter ist eine natürliche oder juristische Person respektive Organisation, die ein Interesse an einem zu realisierenden System hat. Hierzu gehören zum einen die unmittelbaren Benutzer eines Systems. Zum anderen gehören zu der Gruppe der Interessenvertreter auch Auftraggeber, Betreiber oder betroffene Abteilungen.
Iteration	Eine Iteration ist ein einzelner Entwicklungsschritt innerhalb eines iterativ-inkrementellen Entwicklungsprozesses. Innerhalb einer Iteration werden alle typischen Tätigkeiten einer Software-Entwicklung wie Analyse, Entwurf etc. durchgeführt.
IT-Standard und -Richtlinie	Ein IT-Standard bzw. -Richtlinie ist eine organisationsweite Vorgabe, denen zu entwickelnde IT-Systeme innerhalb einer Organisation genügen müssen.
IT-System (System)	Ein System ist eine Einheit, die aus miteinander interagierenden Software- und Hardware-Bausteinen besteht und zur Erfüllung eines fachlichen Ziels existiert. Es kommuniziert zur Erreichung seines Ziels mit seiner Umwelt und muss den durch die Umwelt vorgegebenen Rahmenbedingungen Rechnung tragen.
Java Enterprise Edition (JEE)	Die JEE ist eine Komponentenplattform von SUN.
Knoten (englisch: <i>node</i>)	Ein Knoten ist eine Systemresource, wie z. B. eine physische Recheneinheit, eine Ausführungsumgebung oder ein Anwendungsserver.
Kommando (englisch: <i>command</i>)	Ein Kommando ist eine spezielle Art von Dienst, der über eine Schnittstelle angeboten wird. Ein Kommando verändert den Zustand eines Bausteins.
Kommunikations-Middleware	Eine Kommunikations-Middleware ist eine zentrale Technologie für viele verteilte Systeme (siehe Middleware). Sie ist eine Plattform, die Anwendungen Dienste für alle Aspekte der Verteilung anbietet, wie verteilte Aufrufe, effizienten Zugriff auf das Netzwerk, Transaktionen und viele andere.

Begriff (Synonym)	Erklärung
Kommunizieren der Architektur	Das Kommunizieren der Architektur ist die Architektur-Tätigkeit, mit dem Ziel, den einzelnen Interessenvertretern (z. B. Projektleitern, Entwicklern, Benutzern, Kunden) ein möglichst gutes Verständnis der Architektur sowie der Architektur-Entscheidungen zu vermitteln.
Komponentenorientierung	Die Komponentenorientierung ist eine Weiterentwicklung der Objektorientierung und bietet Komponenten als wiederverwendbare, in sich geschlossene Bausteine eines Systems an.
Komponentenplattform	Eine Komponentenplattform basiert auf der Trennung von technischen Belangen und fachlichen Belangen. Die technischen Belange werden automatisiert von einem Container übernommen. Beispiele für technische Belange im Enterprise-Umfeld sind Verteilung, Sicherheit, Persistenz, Transaktionen, Nebenläufigkeit und Ressourcenmanagement.
Konzeptionelle Sicht	Die konzeptionelle Sicht beschreibt fachliche Aspekte ohne auf technische Details einzugehen.
Kooperator	Ein Kooperator besitzt die Fähigkeit, aufgrund seines sozialen Wesens auf Menschen einzugehen und den Teamgeist zu fördern. In Krisensituatoren tendiert er eher zur Unentschlossenheit.
Laufzeitanforderung	Eine Laufzeitanforderung beinhaltet Erwartungen hinsichtlich des Verhaltens eines Systems zur Laufzeit.
Leistungsverhalten (Performanz)	Leistungsverhalten ist eine Laufzeitanforderung und beschreibt das Leistungsvermögen respektive die Performanz des Systems bei der Reaktion auf diese äußeren Ereignisse.
Logische Sicht (funktionale Sicht)	Die logische Sicht beschreibt Verantwortlichkeiten, Schnittstellen und Interaktionen der Systembausteine.
Lückenfinder	Ein Lückenfinder untersucht Sachverhalte neutral und ist gut im Analysieren. Es liegt ihm jedoch nicht, eigene Ideen einzubringen und andere Menschen zu motivieren.
Message-oriented Middleware	Eine Message-oriented Middleware ist eine weit verbreitete Art von Middleware-System (siehe Kommunikations-Middleware), die asynchrone Nachrichten als zentrale Abstraktion benutzt.
Metamodell	Ein Metamodell ist ein spezielles Modell mit dem Zweck, die abstrakte Syntax, das heißt Modellelemente und deren Beziehungen, der Menge aller zu diesem Metamodell konformen Modelle zu definieren. Modelle werden dann

Begriff (Synonym)	Erklärung
	auch als Instanzen des zugehörigen Metamodells bezeichnet.
Meta-Programmierung	Die Idee der Meta-Programmierung ist es, durch eine zusätzliche Abstraktionsebene in Software-Systemen eine höhere Flexibilität und Kontrolle zu erreichen.
Methode	Eine Methode ist eine definierte Verfahrensanleitung zur Durchführung einer bestimmten Aufgabe oder Tätigkeit der Software-Entwicklung.
Methodik	Eine Methodik ist die Gesamtheit aller im Rahmen eines Vorgehensmodells definierten Methoden.
Middleware	Die Middleware beschäftigt sich mit der Verteilungsarchitektur eines Software-Systems. Sie bietet Anwendungen Dienste für alle Aspekte der Verteilung an, wie verteilte Aufrufe, effizienten Zugriff auf das Netzwerk, Transaktionen und viele andere.
Mikro-Architektur (Detail- oder Fein-Entwurf, englisch: <i>detail design</i>)	Eine Mikro-Architektur detailliert eine Software-Architektur, indem nicht-tragende Systembausteine definiert und deren Strukturierung festgelegt werden.
Mittelbare nicht-funktionale Anforderung (Rahmenbedingungen)	Eine mittelbare nicht-funktionale Anforderung wirkt auf die Art und Weise der Realisierung der gewünschten Funktionalitäten und Qualitäten ein. Sie repräsentiert Vorgaben oder Gegebenheiten, die eingehalten respektive berücksichtigt werden müssen und somit den Realisierungsrahmen vorgeben.
Modellgetriebene Software-Entwicklung (MDA, MDD, MDE, MDSD MDE)	Die modellgetriebene Software-Entwicklung setzt Modelle nicht lediglich zu Dokumentationszwecken ein, sondern behandelt sie als zentrale Artefakte eines lauffähigen Systems. Sie ist ein Oberbegriff für Techniken, die aus Modellen automatisiert lauffähige Software erzeugen.
Modellierungssprache	Modellierungssprachen dienen dazu, Systeme zu spezifizieren. Eine Modellierungssprache ist das Aggregat aus konkreter Syntax, abstrakter Syntax sowie statischer und dynamischer Semantik. Die abstrakte Syntax wird auch als Metamodell bezeichnet. Die Begriffe Wohlgeformtheitskriterien und Constraints werden auch als Synonyme für statische Semantik gebraucht.
Moderator	Ein Moderator ist ein selbstsicherer Mensch, der kaum Vorurteile hat und ein ruhiges Wesen besitzt. Er kann andere Menschen gut in das Teamgeschehen einbinden und besitzt eine starke Wahrnehmungskraft. Er verfügt jedoch nicht über das übliche Maß an Kreativität.

Begriff (Synonym)	Erklärung
Modularitätsprinzip	Das Modularitätsprinzip besagt, dass Systeme aus wohl definierten Systembausteinen bestehen sollten, deren funktionalen Verantwortlichkeiten klar abgegrenzt sind.
Mustersprache	Eine Mustersprache ist eine Sammlung von semantisch zusammengehörenden Mustern, welche Lösungsszenarien für Probleme in einem bestimmten Kontext bietet.
Netzwerkarchitektur	Netzwerkarchitektur beleuchtet die Infrastruktur von Systemen beziehungsweise gesamter Unternehmungen. Die Planung und der Entwurf der Funktionen, Dienste, Bausteine und Protokolle eines Netzwerks sind die Hauptaufgaben dieser Disziplin.
Next-Generation-Operation-Support-Systems-Initiative (NGOSS)	Die Next-Generation-Operation-Support-Systems-Initiative des TeleManagement Forums definiert eine umfassende, auf die Telekommunikationsindustrie zugeschnittene Referenzarchitektur.
Nicht-funktionale Anforderung	Eine nicht-funktionale Anforderung verkörpert eine Erwartung und Notwendigkeit, die von Interessenvertretern neben den funktionalen Anforderungen als wichtig erachtet werden und die über die reine gewünschte Funktionalität hinausgehen.
Nicht-funktionale Bausteinanforderung	Eine nicht-funktionale Bausteinanforderung formuliert den Qualitätsanspruch, der zur Erbringung der funktionalen Bausteinanforderungen definiert ist. Sie lässt sich nur schwer in einem Baustein lokalisieren.
Nicht-funktionale Organisationsanforderung	Eine nicht-funktionale Organisationsanforderung formuliert den Qualitätsanspruch der Umwelt an die Erbringung der funktionalen Organisationsanforderungen.
Nicht-funktionale Systemanforderung	Eine nicht-funktionale Systemanforderung formuliert den Qualitätsanspruch der Umwelt an die Erbringung der funktionalen Systemanforderungen. Sie lassen sich nur schwer in einem System lokalisieren.
n-Tier-Architektur	Eine n-Tier-Architektur dient der logischen Strukturierung eines Systems in n Tiers. n ist die Anzahl an Tiers. 2-Tier- und 3-Tier-Architekturen sind häufig anzutreffende Beispiele für n-Tier-Architekturen.
Object-Relational-Mapping (ORM)	Object-Relational-Mapping ermöglicht die Integration einer objekt-orientierten Anwendung mit dem relationalen Paradigma. Dazu bietet ORM eine Datenbankzugriffsschicht für relationale Datenbanken und objektorientierte Applikationslogik. Object-Relational-Mapper, wie Hibernate, sind eine wichtige Technologie in diesem Bereich.

Begriff (Synonym)	Erklärung
Objektorientierung (OO)	Die Objektorientierung basiert auf der Idee, die Daten, die eine Reihe von zusammenhängenden Methoden gemeinsam bearbeiten, mit diesen Methoden zu bündeln. Objektorientierung ist ein heute vorherrschendes architektonisches Konzept.
Offenes System	Ein offenes System steht in Kontakt mit seiner Umwelt und tauscht mit ihr Informationen aus. Darüber hinaus kann ein Energieaustausch stattfinden. Es muss mit seiner Umwelt interagieren, um existieren zu können.
Ökonomisch-rationales Organisationsverständnis (Taylorismus)	Das ökonomisch-rationale Verständnis hat seine Wurzeln in der frühen Industrialisierung. Ihm liegt das Prinzip der perfekten Arbeitsteilung zugrunde. Der Mensch als Individuum wird dabei als Produktionsfaktor wahrgenommen.
OORPC-System	Ein OORPC-System ist eine verbreitete Art von Middleware-System (siehe Kommunikations-Middleware), die auf objekt-orientierten, verteilten RPC-Aufrufen basiert.
Operationale Entscheidung	Eine operationale Entscheidung adressiert kurzfristige Belange und hat ein geringeres Ausmaß.
Organisation	Eine Organisation ist ein soziales Gebilde, das dauerhaft ein Ziel verfolgt und eine formale Struktur aufweist, mit deren Hilfe Aktivitäten der Mitglieder auf das verfolgte Ziel ausgerichtet werden sollen.
Organisationsebene	Die Organisationsebene umfasst architektur-relevante Elemente auf der Abstraktionsstufe von Organisationen.
Organisationskultur (Kultur)	Eine Organisationskultur definiert den normativen Rahmen und somit den Freiraum der Architektur-Gestaltung. Dies kann sich dadurch äußern, dass ganz klare Standards und Richtlinien vorgegeben sind. Ferner legt die Kultur einer Organisation die Art und Weise fest, wie Menschen innerhalb der Organisation miteinander umgehen und welche Erwartungen die Organisation an sie stellt.
Organisatorische Rahmenbedingung	Eine organisatorische Rahmenbedingung ist eine Vorgabe, wie Budget und Time-to-Market. Ferner gehören hierzu auch Restriktionen hinsichtlich der Architektur-Gestaltung aufgrund von vorhandenem Wissen und Erfahrung innerhalb des Teams.
Orientierungsphase	Die Orientierungsphase ist die fünfte Phase der Gruppenbildung nach Tuckmann. Nach der Erreichung des Gruppenziels erfolgt in der Orientierungsphase ein Besinnen auf die Leistung und Erlebnisse der Gruppe. Ferner kann

Begriff (Synonym)	Erklärung
	sich die Gruppe auflösen oder auf ein neues Ziel ausrichten.
Peer-to-Peer	Peer-to-Peer ist eine Basisarchitektur, die eine Reihe von gleichwertigen Peers zur (verteilten) Kommunikation nutzt.
Plattform	Eine Plattform ist ein System, welches aus Software- und ggf. Hardware-Bausteinen bestehen kann. Sie dient zur Ausführung von Software-Bausteinen eines Systems.
Plattformarchitektur	Eine Plattformarchitektur definiert die Bausteine einer Plattform und legt deren Strukturierung fest.
Plattformunabhängigkeit (Portierbarkeit)	Plattformunabhängigkeit ist eine Entwicklungszeitanforderung und fordert, dass ein System auf unterschiedlichen Plattformen betrieben respektive auf unterschiedliche Plattformen portiert werden kann.
Präsentation	Eine Präsentation kann als informelle Reviewmittel genutzt werden. Bei der Präsentationsform werden Aspekte einer Architektur Interessenvertretern informell vorgestellt. Interessenvertreter können unmittelbar Feedback zu den vorgestellten Aspekten geben.
Präsentationslogik	Präsentationslogik ist Logik, die zur Kommunikation mit dem Benutzer dient.
Prinzip der hohen Kohäsion	Kohäsion ist ein Maß für die Abhängigkeiten innerhalb eines Systembausteins. Das Prinzip der hohen Kohäsion besagt, dass diese Kohäsion innerhalb eines Systembausteins möglichst hoch sein soll.
Prinzip der losen Kopplung	Das Prinzip der losen Kopplung ist ein zentrales Prinzip und besagt, dass die Kopplung zwischen Systembausteinen möglichst niedrig gehalten werden soll.
Prinzip des Entwurfs für Veränderung	Das Prinzip des Entwurfs für Veränderung besagt, dass man versuchen soll, die Architektur so zu entwerfen, dass man leicht mit den wahrscheinlichen Änderungen eines Software-Systems umgehen kann.
Produktlinienarchitektur	Eine besondere Form von Referenzarchitektur ist eine Produktlinienarchitektur, die die gemeinsame Architektur mehrerer, ähnlicher Software-Produkte definiert.
Prozesssicht	Die Prozesssicht beschreibt Nebenläufigkeitsaspekte der Systembausteine.
Publish-/Subscribe-Architektur	Eine Publish-/Subscribe-Architektur ist eine Basisarchitektur bei der Aufrufe nicht direkt unter den Kommunikationsteilnehmern versendet, sondern durch einen Vermittler weitergeleitet werden. Typischerweise wird in

Begriff (Synonym)	Erklärung
	einer Publish-/Subscribe-Architektur über asynchrone Ereignisse kommuniziert.
Qualität	siehe unmittelbare nicht-funktionale Anforderung
Qualitätsattribut	siehe unmittelbare nicht-funktionale Anforderung
Qualitätsattributszenario	Ein Qualitätsattributszenario macht nicht-funktionale Anforderungen greifbar, indem sie nach einem bestimmten Schema beschrieben werden. Insbesondere die Messbarkeit steht dabei im Vordergrund. Man kann Qualitätsattributszenarien in Szenariotypen einteilen. Beispiele sind: Verfügbarkeitsszenarien, Änderbarkeitsszenarien, Performanzszenarien etc.
Querschnittsaspekt/-belang (englisch: <i>cross-cutting concern</i>)	Ein Querschnittsaspekt ist ein technischer Aspekt (z. B. Logging), der orthogonal zu fachlichen Aspekten liegt. Querschnittsaspekte liegen verstreut über fachliche Bausteine vor.
Rahmenbedingung	siehe mittelbar nicht-funktionale Anforderung
Reduktionismus (Dekomposition)	Der Reduktionismus untersucht Systembausteine getrennt voneinander. Diese Sichtweise hilft, konkrete Aussagen über das Verhalten und die Funktionsweise einzelner Systembausteine zu treffen.
Reference Model for Open Distributed Processing (RM-ODP)	Das RM-ODP ist ein von der ISO standardisiertes Architektur-Rahmenwerk für offene, verteilte Systeme. Es umfasst neben einem Architektur-Sichtenmodell unter anderem ein Objektmodell und eine Sammlung von Funktionsdefinitionen.
Referenzarchitektur	Eine Referenzarchitektur kombiniert allgemeines Architektur-Wissen und allgemeine Erfahrung mit spezifischen Anforderungen zu einer architektonischen Gesamtlösung für einen bestimmten Problembereich.
Referenzmodell	Ein Referenzmodell enthält die spezifischen Charakteristika des adressierten Problembereichs (im Zusammenhang einer Referenzarchitektur).
Reflection	Reflection erlaubt Programmen beispielsweise Typinformationen, Informationen über Klassen (Attribute und Operationen) und Vererbungshierarchien abzufragen.
RPC-System	Ein RPC-System ist eine verbreitete Art von Middleware-System (siehe Kommunikations-Middleware), das verteilte Prozeduraufrufe als zentrale Abstraktion nutzt.
Schicht (englisch: <i>layer</i>)	Eine Schicht dient zur logischen Strukturierung der Software-Bausteine eines Systems. Software-Bausteine einer Schicht sind kohäsiv und können auf die Software-Bausteine einer direkt nachfolgenden unteren Schicht zugreifen.

Begriff (Synonym)	Erklärung
	fen. Schichten werden bei der Erarbeitung der logischen Sicht eingesetzt.
Schichtenarchitektur (englisch: <i>layered architecture</i>)	Eine Schichtenarchitektur dient der logischen Strukturierung eines Systems in Schichten.
Schlüsselabstraktion	Eine Schlüsselabstraktion repräsentiert eine wesentliche Abstraktion einer Fachdomäne, die von dem zu realisierenden System behandelt werden muss. Beispiele sind Abstraktionen von Gegenständen, Konzepten, Orten oder Personen.
Schnittstelle	Eine Schnittstelle definiert einen Vertrag zwischen dem Systembaustein, der die Schnittstelle anbietet, und den Systembausteinen, die sie nutzen. Des Weiteren legt eine Schnittstelle die Operationen fest, die von dem Systembaustein angeboten werden.
Scrum	Scrum ist ein agiles Vorgehensmodell, das eine Sammlung von Sitzungen, Artefakten, Rollen, Werten und Grundüberzeugungen umfasst. Organisation der Arbeit sowie Auswahl der Mittel und Methoden erfolgt hier weitgehend selbstständig durch die Projektmitarbeiter.
Separation-of-Concerns-Prinzip	Das Separation-of-Concerns-Prinzip besagt, dass man verschiedene Aspekte eines Problems voneinander trennen soll und jedes dieser Teilprobleme für sich behandeln soll.
Serviceorientierte Architektur (SOA)	Eine serviceorientierte Architektur ist eine Basisarchitektur, welche die funktionalen Schnittstellen von Software-Bausteinen als wiederverwendbare, verteilte, lose gekoppelte und standardisiert zugreifbare Services repräsentiert.
Shared-Repository-Architektur	In einer Shared-Repository-Architektur stellt ein Baustein des Systems einen zentralen Datenspeicher zur Verfügung.
Sicherheit	Sicherheit ist eine nicht-funktionale Anforderung mit durchdringendem Charakter. Sie beschäftigt sich mit Vertraulichkeit (englisch: <i>confidentiality</i>), Authentifizierung (englisch: <i>authentication</i>), Integrität (englisch: <i>integrity</i>), Privatsphäre (englisch: <i>privacy</i>), Unleugbarkeit (englisch: <i>nonrepudiation</i>) sowie Schutz vor Zerstörung und Schutz des Betriebs (englisch: <i>intrusion protection</i>).
Sicherheitsarchitektur	Sicherheitsarchitektur als Disziplin beschäftigt sich mit der Gewährleistung von Sicherheitsaspekten wie Identitäts- und Berechtigungsüberprüfung sowie Nachweisbarkeit und Unleugbarkeit sicherheitsrelevanter Vorgänge. Sicherheitsarchitektur als Basisarchitektur bezieht sich auf eine zu schützende Anwen-

Begriff (Synonym)	Erklärung
	dung und eine darunterliegende Sicherheitsinfrastruktur.
Sicht (englisch: view)	siehe Architektur-Sicht
Simulation	Eine Simulation gibt Antworten für bestimmte Fragestellungen, wie z. B. das Verhalten des Systems unter Last, ohne das System selbst implementieren zu müssen.
Skalierbarkeit	Skalierbarkeit ist eine Entwicklungszeitanforderung, die besagt, dass ein System mit steigenden Lasten umgehen können muss. Man unterscheidet in der Regel vertikale und horizontale Skalierbarkeit. Bei ersterer wird z. B. ein Server durch einen leistungsfähigeren Server ausgetauscht. Bei letzterer wird die Last auf mehrere Server verteilt.
Skelettsystem	Ein Skelettsystem entspricht in seiner Struktur bereits der Architektur des endgültigen Systems. Die einzelnen Bausteine hingegen stellen noch nicht eine vollständige Implementierung der Funktionalität zur Verfügung. Vielmehr stellen sie nur die Funktionalität zur Verfügung, die zur Abbildung eines klar abgegrenzten Anwendungsfalls benötigt wird.
Skriptsprachen	Eine Skriptsprache ist ursprünglich eine Programmiersprache, die Software-Systeme kontrolliert oder steuern sollen. Heute werden Skriptsprachen aber auch für alle möglichen anderen Zwecke als Sprachen auf hoher Abstraktionsebene eingesetzt.
Software Reengineering	Software Reengineering beschäftigt sich mit den folgenden wesentlichen Aufgaben: Reverse Engineering, Restrukturierung, Software-Evolution und Wrapping.
Software-Architektur (Anwendungsarchitektur, Applikationsarchitektur, Makro-Architektur, Grob-Entwurf, englisch: <i>high-level design</i>)	Software-Architektur _{Gesamt} = Software-Architektur _{Struktur} + Software-Architektur _{Disziplin} Software-Architektur _{Struktur} : Die Software-Architektur eines Systems beschreibt dessen Software-Struktur respektive dessen -Strukturen, dessen Software-Bausteine sowie deren sichtbaren Eigenschaften und Beziehungen zueinander als auch zu ihrer Umwelt. Software-Architektur _{Disziplin} : Software-Architektur als Disziplin befasst sich mit den architektonischen Tätigkeiten und den hiermit verbundenen Entscheidungen zum Entwurf und zur Umsetzung einer Software-Architektur.
Spiral-Modell	Das Spiral-Modell ist eine Verfeinerung des Wasserfall-Modells, welches die Schwächen des letzteren adressiert. Statt die genannten Disziplinen nur ein einziges Mal sequentiell zu durchlaufen, wird ein Software-Entwicklungsprojekt in mehrere Zyklen unterteilt.

Begriff (Synonym)	Erklärung
Standpunkt (englisch: <i>viewpoint</i>)	Ein Standpunkt ist die systemunabhängige Spezifikation einer bestimmten Architektursicht.
Strategische Entscheidung	Eine strategische Entscheidung hat längerfristigen Charakter und umfassende Wirkung.
Streitphase	Die Streitphase ist die zweite Phase der Gruppenbildung nach Tuckmann. Für jedes Gruppenmitglied entscheidet sich, ob es in der Gruppe verbleiben will oder nicht.
Strukturbruch	Ein Strukturbruch ist eine wichtige architektonische Problematik, die den Bruch zwischen zwei Paradigmen bezeichnet. Z. B. bei der persistenten Datenhaltung kommt der Strukturbruch zwischen den Paradigmen der Datenbank und der Applikationslogik vor.
Subsystem	Ein Subsystem vereint kohärente Funktionalität und ist in sich selbst abgeschlossen. Somit bietet ein Subsystem zusammengehörende Funktionalität, die einen Teil der an das System gestellten Anforderungen befriedigt.
Systemarchitektur	$\text{Systemarchitektur}_{\text{Gesamt}} = \text{Systemarchitektur}_{\text{Struktur}} + \text{Systemarchitektur}_{\text{Disziplin}}$ $\text{Systemarchitektur}_{\text{Struktur}}$: Die Systemarchitektur eines Systems beschreibt dessen Struktur respektive dessen Strukturen, dessen Bausteine (Software- und Hardware-Bausteine) sowie deren sichtbaren Eigenschaften und Beziehungen sowohl zueinander als auch zu ihrer Umwelt. $\text{Systemarchitektur}_{\text{Disziplin}}$: Systemarchitektur als Disziplin befasst sich mit den architektonischen Tätigkeiten und den hiermit verbundenen Entscheidungen zum Entwurf und zur Umsetzung einer Systemarchitektur.
Systembaustein (Baustein)	Ein Systembaustein repräsentiert den abstrakten Typ aller konkreten Bausteine eines Systems. Er kann andere Systembausteine benötigen und kann über eine oder mehrere Schnittstellen verfügen respektive eine oder mehrere Schnittstellen anderer Systembausteine erfordern.
Systemebene	Die Systemebene umfasst architekturellelemente auf der Abstraktionsstufe von IT-Systemen.
Systemisches Organisationsverständnis	Das systemische Organisationsverständnis betrachtet die Organisation als ein System, welches ein Ziel sowie eine Systemgrenze besitzt und mit seiner Umwelt interagiert.
Systemkontext	Ein Systemkontext umfasst einerseits eine grafische Darstellung des Systems und seiner Umwelt inklusiver seiner menschlichen Akteure und Umsysteme. Ferner dokumentiert ein

Begriff (Synonym)	Erklärung
	Systemkontext auch die Schnittstellen zwischen dem System und seinen menschlichen Akteuren sowie Umsystemen.
System-Management-Architektur	System-Management-Architektur beinhaltet hauptsächlich den operationalen Aspekt von Systemen. Der Entwurf von Betriebsstrategien zentraler und dezentraler Systemlandschaften und die Definition von Service Level Agreements sind Aufgaben, denen ein Architekt in dieser Disziplin gegenübersteht.
Systemübersicht	Eine Systemübersicht besteht aus einem Systemkontext und einer Architektur-Vision.
Systemvision	Eine Systemvision lotet die betriebswirtschaftliche Zweckmäßigkeit eines Systems aus, benennt die Interessenvertreter, definiert die essenziellen Anforderungen und enthält eine erste Systemübersicht.
Szenariobasierte Methode	Eine szenariobasierte Methode ist eine Beurteilungsmethode, die die Problematik des Interpretationsspielraums nicht-funktionaler Anforderungen adressiert. Hierzu erarbeitet sie konkrete Szenarien, welche eine nicht-funktionale Anforderung weiter konkretisieren. Die Beurteilung der Architektur erfolgt auf Basis der Szenarien.
Taktik	Eine Taktik ist eine Entwurfsentscheidung, die Einfluss nimmt auf die Realisierung der Reaktion eines Qualitätsattributszenarios.
Technische Architektur	Eine technische Architektur widmet sich primär der Realisierung von nicht-funktionalen Anforderungen und definiert technische Bausteine.
Technische Basis	Eine technische Basis umfasst zum einen die benötigten technischen Basisdienste und zum anderen die Implementierung der Frameworks mit denen die fachlichen Bausteine realisiert werden.
Technischer Baustein	Ein technischer Baustein kapselt Funktionalität für nicht-funktionale Aspekte, wie z. B. Logging, Auditing, Sicherheit, Referenzdaten, Persistenz und Transaktionsmanagement. Er nutzt Dienste der Plattform und abstrahiert sie so, dass sie von fachlichen Bausteinen plattformneutral genutzt werden können.
Tier	Ein Tier ist ein Mittel zur Strukturierung der Verteilung von Software-Bausteinen auf Hardware-Bausteine. In einem Tier sind kohäsive Systembausteine (Software- und Hardware-Bausteine) angesiedelt, die über ein Netzwerk miteinander verbunden sind. Tiers werden bei der Erarbeitung der Verteilungssicht verwendet.

Begriff (Synonym)	Erklärung
Transaktionsmonitor	Ein Transaktionsmonitor beschäftigt sich mit der Realisierung von verteilten Transaktionen. Historisch sind Transaktionsmonitore eine verbreitete Art von Middleware-System (siehe Kommunikations-Middleware).
Umsetzen der Architektur	Das Umsetzen der Architektur ist die Architektur-Tätigkeit, bei der die entworfene Architektur umgesetzt wird. Dabei wird einerseits die Umsetzungsinfrastruktur etabliert und andererseits die Architektur-Konformität überwacht.
Umsetzer	Ein Umsetzer ist ein gewissenhafter Mensch, der eine Aufgabe sorgfältig erledigt. Er neigt teilweise zum Perfektionismus und kann sich an Kleinigkeiten stören.
Umsetzungsinfrastruktur	Die Umsetzungsinfrastruktur umfasst Architektur-Mittel (Entwicklungsumgebung, Programmiersprache etc.) für die Umsetzung einer Architektur.
Umsetzungssicht (Entwicklungssicht)	Die Umsetzungssicht beschreibt Aspekte bezüglich Implementation, Build, Konfiguration, Test und Auslieferung sowie Wartung.
Umsetzungsstruktur	Die Umsetzungsstruktur legt fest, wie die Bausteine der logischen Sicht des Systems auf die Bausteine der Umsetzungssicht abgebildet werden.
Unified Method Architecture (UMA)	Die Unified Method Architecture (UMA) ist eine spezielle DSL und dient der Spezifikation von Vorgehensmodellen.
Unified Modeling Language (UML)	Die UML ist eine spezielle Modellierungssprache. Die UML-Infrastruktur bildet den Kern der MOF und der UML-Superstruktur. Durch die UML-Superstruktur wird das UML-Metamodell spezifiziert. UML-Diagramme sind eine Möglichkeit der Notation von UML-Modellen.
Unified Software Development Process (USDP)	USDP ist ein iterativ-inkrementelles Vorgehensmodell und legt sehr umfassend Artefakte, Tätigkeiten und Rollen fest.
Unmittelbare nicht-funktionale Anforderung (Qualität, Qualitätsattribut)	Eine unmittelbare nicht-funktionale Anforderung wird auch als Qualität bzw. als Qualitätsattribut bezeichnet, da sie den qualitativen Charakter der durch Organisationen, IT-Systeme oder Bausteine erfüllten funktionalen Anforderungen widerspiegelt. Der Wunsch von Kunden, eine Bestellung innerhalb von 24 Stunden zu erhalten, kann z. B. als nicht-funktionale Anforderung an eine Organisation angesehen werden.
Verfügbarkeit	Verfügbarkeit ist eine Laufzeitanforderung und drückt sich in dem Verhältnis der Fehlzeiten zu den Produktivzeiten aus. Je kleiner die Fehlzeiten gegenüber den Produktivzeiten, desto höher die Verfügbarkeit des Systems.

Begriff (Synonym)	Erklärung
Verhaltenswissenschaftliches Organisationsverständnis	Das verhaltenswissenschaftliche Verständnis rückt den Menschen in den Mittelpunkt der Betrachtung, indem es ihn nicht mehr als reinen Produktionsfaktor, sondern als soziales Wesen wahrnimmt, das nach Anerkennung und Wertschätzung strebt.
Verstehen der Anforderungen	Das Verstehen der Anforderungen ist eine Architektur-Tätigkeit, die sich mit der Identifikation, der Priorisierung und der Verfeinerung von architekturelevanten Anforderungen beschäftigt. Insbesondere die bewusste Auseinandersetzung mit nicht-funktionalen Anforderungen ist von großer Bedeutung, da diese oftmals gar nicht oder unscharf formuliert sind.
Verteilungssicht (Ausführungssicht)	Die Verteilungssicht beschreibt die physikalische Verteilung der Systembausteine zur Laufzeit.
Vertragsphase	Die Vertragsphase ist die dritte Phase der Gruppenbildung nach Tuckmann. Die Gruppenmitglieder identifizieren sich mit der verhandelten Rolle und einigen sich auf Regeln der Zusammenarbeit.
V-Modell	Das V-Modell ist ein Vorgehensmodell, welches in Deutschland für Projekte der öffentlichen Hand entwickelt wurde. Sein Name kommt von der V-förmigen Darstellung der Tätigkeiten.
V-Modell XT	Das V-Modell XT ist eine Weiterentwicklung des V-Modells, welches die Anpassung des Vorgehensmodells an konkrete Projektbedürfnisse erlaubt.
Vorgehensmodell	Unter einem Vorgehensmodell wird in diesem Buch das Aggregat aus Methodik und Entwicklungsprozess verstanden.
Walkthrough	Ein Walkthrough basiert auf architekturelevanten Szenarien. Bei Walkthroughs werden diese Szenarien durchgespielt, um festzustellen, ob und wie eine Architektur diese erfüllt. An Walkthroughs nehmen betroffene Interessenvertreter teil.
Wartbarkeit	Wartbarkeit ist eine Entwicklungszeitanforderung, die besagt, dass Fehler in einem System in einem angemessen Rahmen behoben werden können. Je einfacher Fehler behoben werden können, umso wartbarer ist das System. Ein wartbares System zeichnet sich durch Systembausteine mit hoher Kohäsion aus.
Wartung	Wartung von Software-Systemen beschäftigt sich mit Änderungen am System nach dessen Auslieferung. Dies umfasst beispielsweise das Korrigieren von Fehlern, die Verbesserung von

Begriff (Synonym)	Erklärung
	Qualitätsattributen wie Performanz und die Evolution bzw. Weiterentwicklung des Systems.
WARUM-Dimension (Architektur-Anforderungen)	Die WARUM-Dimension widmet sich Anforderungen, die an IT-Systeme im Allgemeinen und Architekturen im Speziellen gestellt werden.
WAS-Dimension (Architekturen und Architektur-Disziplinen)	Die WAS-Dimension beinhaltet Architektur-Grundlagen und -Definitionen. Sie bildet hiermit die Basis für die Arbeit als Architekt.
Wasserfall-Modell	Das Wasserfall-Modell ist ein Entwicklungsprozess, der die sequentielle und einmalige Abarbeitung verschiedener Software-Entwicklungsdisziplinen vorsieht. So werden beispielsweise die Disziplinen Anforderungs'erhebung, Analyse, Entwurf und Implementation nacheinander durchlaufen.
Web Service	Ein Web Service ist eine Technologie, die sowohl die Middleware- als auch die SOA-Basisarchitektur umsetzt und stark auf XML und Internet-Standards setzt.
Web-Anwendungsserver	Ein Web-Anwendungsserver ist ein Server für Web-Anwendungen, der dynamisch HTML-Seiten erzeugt und die Anwendungslogik zwischen dem Klienten (Browser) und nachgelagerten Systemen, wie einer Datenbank, ausführt.
Wegbereiter	Ein Wegbereiter sucht Herausforderungen, ist extravertiert und kommunikativ. Es ist eine seiner Stärken, menschliche Kontakte aufzubauen und neue Themen zu erforschen. Im Gegensatz hierzu besteht die Tendenz, dass er das Interesse an einem Thema verliert, wenn es zur Routine wird.
WER-Dimension (Organisationen und Individuen)	Die WER-Dimension ist eine Architektur-Dimension und behandelt die Rolle des Architekten sowie den Einfluss von Individuen und Organisationen auf Architektur.
Wertobjekt	Ein Wertobjekt wird allein über seine Attribute definiert. Es besitzt keine eigene Identität und ist unveränderlich (englisch: <i>immutable</i>). Ein Geldbetrag kann z. B. durch ein Wertobjekt vom Typ Geld repräsentiert werden.
Wiederverwendbarkeit	Wiederverwendbarkeit ist eine Entwicklungszeitanforderung, die besagt, dass Systembausteine so entworfen und umgesetzt werden sollen, dass sie in anderen Kontexten einsetzbar respektive wiederverwendbar sind.
WIE-Dimension (Architektur-Vorgehen)	Die WIE-Dimension ist eine Architektur-Dimension und dient zur Strukturierung von architektonischem Vorgehen. Sie enthält die wichtigsten architektonischen Tätigkeiten, die ein Architekt während seiner Arbeit ausübt.

Begriff (Synonym)	Erklärung
WO-Dimension (Architektur-Perspektiven)	Die WO-Dimension ist eine Architektur-Dimension und umfasst die unterschiedlichen Ebenen, auf denen sich Architektur abspielt, und die Sichten, mit denen Architektur betrachtet werden kann.
WOMIT-Dimension (Architektur-Mittel)	Die WOMIT-Dimension strukturiert die unterschiedlichen architektonischen Mittel, denen sich ein Architekt während seines Handelns bedienen kann.
XML	XML erlaubt es, standardisiert die Struktur von Informationen zu beschreiben und dabei Inhalte und Struktur zu trennen. XML selbst ist kein Standard zum Datenaustausch zwischen Organisationen, sondern ermöglicht es, XML-basierte Austauschformate und Austausch-standards zu definieren.
Zachman-Framework	Das Zachman-Framework ist ein bei IBM entstandenes domänen- und technologieneutrales Architektur-Rahmenwerk mit Fokus auf Unternehmensarchitektur. Es bietet ein mächtiges Architektur-Sichtenmodell, das neben Sichten noch zu den Sichten orthogonal liegende Sichtenspekte und Rollen je Sicht definiert.
Zentralisierung	Die Zentralisierung bezeichnet die Lokation eines Belangs in genau einem Systembaustein. Meist muss allerdings eine Abwägung zwischen der Zentralisierung und der Dezentralisierung getroffen werden.

| Abkürzungsverzeichnis

Abkürzung	Erklärung
.NET	Microsoft Komponentenplattform
2PC	Two-Phase Commit Protocol
3GL	3rd Generation Language
ABLE	Architecture Based Languages and Environment
ACID	Atomicity, Consistency, Isolation, Durability
AC-MDSD	Architecture Centric MDSD
ACME	Spezifische ADL
ACS	Ars Digita Community System
ADL	Architecture Description Language
ADML	Architecture Description Markup Language
AM	Access Management
ANSI	American National Standards Institute
AOP	Aspektorientierte Programmierung
API	Application Programming Interface
ASP	Active Server Pages
ASP	Application Service Provider
AST	Abstract Syntax Tree
ATAM	Architecture Tradeoff Analysis Method
BAM	Business Activity Monitoring
B2B	Business to Business
B2BAI	Business to Business Application Integration
BPEL	Business Process Execution Language
BPEL4WS	Business Process Execution Language for Web Services
CASE	Computer-Aided Software Engineering
CCM	Corba Component Model
CCM	Configuration&Change-Management
CCMS	Configuration&Change-Management-System
CGI	Common Gateway Interface
CICS	Customer and Controller Systems
CLOS	Common Lisp Object System
CLR	Common Language Runtime
CMS	Content Management System
CMU	Carnegie Mellon Universität

Abkürzung	Erklärung
COBOL	Common Business Oriented Language
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
CRM	Customer Relationship Management
CRUD	Create, Read, Update, Delete
CVS	Concurrent Versions System
DCE	Distributed Computing Environment
DCOM	Distributed Component Object Model
DLL	Dynamic Link Library
DOM	Document Object Model
DSL	Domain Specific Language
DTD	Document Type Definition
DTO	Datentransferobjekt
EAI	Enterprise Application Integration
EAST	ADL zur Beschreibung eingebetteter System
EDA	Event Driven Architecture
EDI	Electronic Data Interchange
EDOC	Enterprise Distributed Object Computing
EJB	Enterprise Java Beans
EMF	Eclipse Modeling Framework
EPF	Eclipse Process Framework
ERP	Enterprise Resource Planning
ESB	Enterprise Service Bus
ETL	Extract Transform and Load
eTOM	Enhanced Telecom Operations Map
FDD	Feature Driven Development
FTP	File Transfer Protocol
GoF	Gang of Four
GPL	General Purpose Language
HTML	Hyper Text Markup Language
HTTP	Hyper Text Transfer Protocol
IAM	Identity and Access Management
IBM	International Business Machines
IDL	Interface Description Language
IEEE	Institute of Electrical and Electronics Engineers
IIOP	Internet Inter ORB Protocol
IMS	Information Management System

Abkürzung	Erklärung
ISO	International Organization for Standardization
ISP	Internet Service Provider
IT	Informationstechnologie
ITU	International Telecommunication Union
J2EE	Java 2 Enterprise Edition
JAAS	Java Authentication and Authorization Service
JDBC	Java Database Connectivity
JCA	Java Connector Architecture
JDO	Java Data Objects
JEE	Java Enterprise Edition
JMS	Java Messaging Service
JSP	Java Server Pages
JVM	Java Virtual Machine
KDR	Kundendatenrepository
LDAP	Lightweight Directory Access Protocol
MDA	Model Driven Architecture
MDD	Model Driven Development
MDE	Model Driven Engineering
MDSD	Model Driven Software Development
MIS	Management-Informations-System
MOF	Meta Object Facility
MOM	Message Oriented Middleware
MOP	Meta Object Protocol
NGOSS	Next Generation Operations Support Systems
MVC	Model View Controller
NTLM	NT Lan Manager
OCL	Object Constraint Language
OMA	Object Management Architecture
OMG	Object Management Group
OMT	Object Modeling Technique
OOD	Object Oriented Design
OODBMS	Object Oriented Database Management System
OO-RPC	Object Oriented Remote Procedure Call
OOSA	Object Oriented System Analysis
OOSE	Object Oriented Software Engineering
ORB	Object Request Broker
ORM	Object Relational Mapping

Abkürzung	Erklärung
OSS	Operations Support System
OSS/J	OSS for Java Initiative
P2P	Peer-to-Peer
PBM	Policy Based Management
PC	Personal Computer
PDF	Portable Document Format
PHP	PHP Hypertext Preprocessor
PIM	Platform Independent Model
PKI	Public Key Infrastructure
PL	Product Line
PLE	Product Line Engineering
POC	Proof of Concept
POSA	Pattern Oriented Software Architecture
PSM	Platform Specific Model
PSM	Protocol State Machine
PSP	Projektstrukturplan
P2P	Peer-to-Peer
PzP	Punkt-zu-Punkt
QoS	Quality of Service
QVT	Query, Views, Transformation
QWAN	Quality without a Name
RAM	Random Access Memory
RDBMS	Relational Database Management System
RDF	Resource Description Framework
REST	Representational State Transfer
RMC	Rational Method Composer
RMI	Remote Method Invocation
RM-ODP	Reference Model for Open Distributed Processing
RPC	Remote Procedure Call
RUP	Rational Unified Process
SA/D	Structured Analysis / Design
SAAM	Software Architecture Analysis Method
SADL	Structural Architecture Description Language
SAP	Systeme, Anwendungen und Produkte
SAX	Simple API for XML
SCM	Supply Chain Management
Scrum	Agile Software-Entwicklungs methode

Abkürzung	Erklärung
SDLM	Software Design Level Model
SID	Shared Information and Data Model
SKM	Software-Konfigurations-Management
SLA	Service Level Agreement
SMTP	Simple Mail Transport Protocol
SOA	Serviceorientierte Architektur
SOAP	Simple Object Access Protocol
SOI	Serviceorientierte Infrastruktur
SPEM	Software Process Engineering Metamodel
SQL	Structured Query Language
SSF	Software System Family
Tcl	Tool Command Language
TCP/IP	Transmission Control Protocol/Internet Protocol
TK	Telekommunikation
TMF	Tele Management Forum
TOGAF	The Open Group Architecture Framework
TP	Transaction Processing
TT	Trouble Ticket
UDDI	Universal Description, Discovery and Integration
UMA	Unified Method Architecture
UML	Unified Modeling Language
UP	Unified Process
USDP	Unified Software Development Process
USE	Unanticipated Software Evolution
VM	Virtuelle Maschine
VMS	Vorfall-Management-System
WAM	Web-Access-Management
WBS	Work Breakdown Structure
WSDL	Web Services Description Language
WWS	Warenwirtschaftssystem
WWW	World Wide Web
XMI	XML Metadata Interchange
XML	eXtensible Markup Language
XP	eXtreme Programming
XSLT	eXtensible Stylesheet Language Transformations

| Literaturverzeichnis

[ABLE 2005]

ABLE, *Architecture Based Languages and Environment*,
<http://www-2.cs.cmu.edu/afs/cs/project/able/www/able.html>, 2005

[ACME 1998]

ACME, *The Acme Architectural Description Language*,
http://www-2.cs.cmu.edu/~acme/language_overview.html, 1998

[ActiveCharts 2007]

ActiveCharts - UML 2 Activity Diagram Execution and Visualization,
<http://activecharts.informatik.uni-ulm.de/publications.html>, 2007

[ADML 2002]

ADML, *Architecture Description Markup Language*,
http://www.opengroup.org/architecture/adml/adml_home.htm, 2002

[Alexander 1977]

Alexander, Christopher; Ishikawa, Sara; Silverstein, Murray; Jacobson, Max;
Fiksdahl-King, Ingrid; Angel, Shlomo, *A Pattern Language*, Oxford University
Press, 1977

[Alur et al. 2003]

Alur, Deepak; Crupi, John; Malks, Dan, *Core J2EE Patterns*, Prentice Hall PTR,
2003

[Ambler 2002]

Ambler, Scott, *Agile Enterprise Architecture: Beyond Enterprise Data Modeling*,
<http://www.agiledata.org>, 2002

[Ambler 2007]

Ambler, Scott, *Agile Architecture: Strategies for Scaling Agile Development*,
<http://www.agilemodeling.com/essays/agileArchitecture.htm>, 2007

[Andrews et al. 2003]

Andrews, T.; Curbera, F.; Dholakia, H.; Goland, Y.; Klein, J.; Leymann, F.; Liu, K.;
Roller, D.; Smith, D.; Thatte, S.; Trickovic, I.; Weerawarana, S., *Business Process
Execution Language for Web Services*, Version 1.0,
<http://www.ibm.com/developerworks/webservices/library/ws-bpel>,

[Antlr 2005]

Parr, Terence, *ANTLR - ANOther Tool for Language Recognition*,
<http://www.antlr.org>, 2005

[AOSD 2005]

Aspect-Oriented Software Association, <http://aosd.net>, 2005

[Apache 2003]

Apache Software Foundation, *Web Services - Axis*, <http://ws.apache.org/axis/>,
2003

[Apache 2008a]

Apache Software Foundation, *Formatting Objects Processor (FOP)*,

<http://xmlgraphics.apache.org/fop/>, 2008

[Apache 2008b]

Apache Software Foundation, *Velocity Project*, <http://velocity.apache.org/>, 2008

[Ascet 2005]

ETAS GmbH, *ASCET-SD Overview*, http://en.etasgroup.com/products/ascet_sd/,

2005

[Autosar 2005]

The Autosar Consortium, Automotive Open System Architecture,

<http://www.autosar.org>, 2005

[Avgeriou and Zdun 2005]

Avgeriou, Paris und Zdun, Uwe. *Architectural Patterns Revisited - A Pattern Language*. In Proceedings of 10th European Conference on Pattern Languages of Programs (EuroPlop 2005), Irsee, Germany, 2005

[Bass et al. 2003]

Bass, Len; Clements, Paul; Kazman, Rick, *Software Architecture in Practice*, Second Edition, Addison-Wesley, New York, 2003

[Bea 2003]

BEA, *BEA Tuxedo 8.1*. <http://www.bea.com/products/tuxedo/index.shtml>, 2003

[Beck 2000]

Beck, Kent, *Extreme Programming*, Addison-Wesley, München, 2000

[Beedle und Schwaber 2001]

Beedle, Mike; Schwaber, Ken, *Agile Software Development with Scrum*, Prentice-Hall, Upper Saddle River, NJ, 2001

[Belbin 1993]

Belbin, Meredith, *Team Roles at Work*, Butterworth-Heinemann, 1993

[Boehm 1998]

Boehm, Barry, *A Spiral Model of Software Development and Enhancement*, IEEE, 21, 61 72, 1998

[Bonér und Vasseur 2004]

Bonér, Jonas; Vasseur, Alexandre, *Aspectwerkz*, <http://aspectwerkz.codehaus.org>, 2004

[Bosch 2000]

Bosch, Jan, *Design and Use of Software Architectures*, Addison-Wesley, 2000

[Booth et al. 2003]

Booth, D.; Haas, H.; McCabe, F.; Newcomer, E.; Champion, M.; Ferris, C.; Orchard, D., *Web Services Architecture*, W3C Working Draft 8, <http://www.w3.org/TR/2003/WD-ws-arch-20030808/>, 2003

[Bouzan und Bouzan 1997]

Bouzan, Tony; Bouzan, Barry, *Das Mind-Map-Buch, Die beste Methode zur Steigerung ihres geistigen Potentials*, mvg-verlag, 1997

[Box et al. 2000]

Box, D.; Ehnebuske, D.; Kakivaya, G.; Layman, A.; Mendelsohn, N.; Nielsen, H. F.; Thatte, S.; Winer, D., *Simple object access protocol (SOAP) 1.1*, <http://www.w3.org/TR/SOAP/>, 2000

[Bray et al. 1998]

Bray, T.; Paoli, J.; Sperberg-McQueen, C. M., *Extensible markup language (XML) 1.0*, <http://www.w3.org/TR/1998/REC-xml-19980210>, 1998

[Bredemeyer und Malan 2004]

Bredemeyer, Dana; Malan, Ruth, *Software Architecture Action Guide*, <http://www.ruthmalan.com/>, 2004

[Bredemeyer 2002]

Bredemeyer, Dana, *Introduction to Software Architecture*, <http://www.bredemeyer.com/papers.htm>, 2002

[Briggs und Myers 1995]

Briggs, Isabel; Peter B. Myers, *Gifts Differing: Understanding Personality Type*, Davies-Black Publishing, 1995

[Brooks 1995]

Brooks, F., *The Mythical Man-Month*, Addison Wesley, New York, 1995

[Brown et al. 1998]

Brown, William, J.; Malveau, Raphael, C.; McCormick III, Hays, W., "Skip"; Mowbray, Thomas, J., *Anti Patterns – Refactoring Software Architectures, and Projects in Crisis*, John Wiley & Sons, New York, 1998

[Burke 2004]

Burke, B., *JBoss aspect oriented programming*, <http://www.jboss.org/developers/projects/jboss/aop.jsp>, 2004

[Buschmann et al. 1996]

Buschmann, Frank; Meunier, Regine; Rohnert, Hans; Sommerlad, Peter; Stal, Michael, *Pattern-Oriented Software Architecture Vol. 1, A System of Patterns*, John Wiley & Sons, New York, 1996

[Chaudron 2002]

Chaudron, Michel, *Documenting Architectures & Architecture Description Languages*, http://www.win.tue.nl/~mchaudro/swads/ADLs_v2002.ppt, 2002

[Chikofsky und Cross 1990]

Chikofsky, Elliot J.; Cross, James H., *Reverse Engineering and Design Recovery: A Taxonomy*, IEEE Software, 7(1), 1990

[Christensen et al. 2001]

Christensen, E.; Curbera, F.; Meredith, G.; Weerawarana, S., *Web services description language (WSDL) 1.1*, <http://www.w3.org/TR/wsdl>, 2001

[Chughtai und Vogel 2001]

Chughtai, Arif; Vogel, Oliver, *Software-Wiederverwendung, Theoretische Grundlagen, Vorteile und realistische Beurteilung*, <http://www.ovogel.de>, 2001

[Clements und Northrop 2001]

Clements, P.; Northrop, L., *Software Product Lines: Practices and Patterns*, Addison-Wesley, 2001

[Cockburn 1995]

Cockburn, Alistair, *Growth of Human Factors in Application Development*, <http://alistair.cockburn.us/crystal/articles/gothiad/growthofhumanfactorsinsd.htm>, 1995

[Cockburn 1996]

Cockburn, Alistair, *The Interaction of Social Issues and Software Architecture*, Communications of the ACM, Ausgabe 39, Nummer 10, 1996

[Cockburn 2000]

Cockburn, Alistair, *Writing Effective Use Cases*, Addison-Wesley, New York, 2000

[Cockburn 2002]

Cockburn, Alistair, *Agile Software Development*, Addison-Wesley, New York, 2002

[Conway 1968]

Conway, Melvin E., *How Do Committees Invent?*, Datamation magazine, F. D. Thompson Publications, Inc., April, 1968

[Coplien 2004]

Coplien, James O., *A Pattern Definition*, <http://hillside.net/patterns/definition.html>, 2004

[Coplien und Harrison 2004]

Coplien, James O.; Harrison, Neil B., *Organizational Patterns of Agile Software Development*, Prentice Hall, Upper Saddle River, NJ, 2004

[Cunningham et al. 2001]

Cunningham, Wart et al., *Manifesto for Agile Software Development*, <http://agilemanifesto.org/>, 2001

[Curtis et al. 1988]

Curtis, B.; Krasner H.; Iscoe N., *A Field Study of the Software Design Process for Large Systems*, Communications of the ACM, Ausgabe 31, Nummer 11, 1988

[Czarnecki und Eisenecker 2000]

Czarnecki, Krysztof; Eisenecker, Ulrich W., *Generative Programming - Methods, Tools and Applications*, Addison-Wesley, 2000

[Czarnecki und Helsen 2006]

Czarnecki K., Helsen S., *Feature-model-based survey of model transformation approaches*, IBM Systems Journal archive, 45(3):621 645, 2006

[Davis 1993]

Davis, Alan, *Software Requirements: Objects, Functions and States, 2nd edition*, Prentice Hall, Upper Saddle River, NJ, 1993

[Dijkstra 1972]

Dijkstra, Edsger W., *The Humble Programmer*, Communications of the ACM, 1972

[Dorfmann und Thayer 1990]

Dorfman, Merlin; Richard H. Thayer, *Guidelines and Examples of System and Software Requirements Engineering*, IEEE Computer Society Press, Los Alamitos CA., 1990

[Dörner 1989]

Dörner, Dietrich, *Die Logik des Mißlingens, Strategisches Denken in komplexen Situationen*, Rowohlt Verlag, Reinbek bei Hamburg, 1989

[Dröschel et al. 1998]

Dröschel, Wolfgang; Heuser, Walter; Midderhoff, Rainer (Hrsg.), *Inkrementelle und objektorientierte Vorgehensweisen mit dem V-Modell 97*, Oldenbourg, München, 1998

[Drumm 1995]

Drumm, Hans-Jürgen, *Personalwirtschaftslehre*, 3. neu bearbeitete und erweiterte Auflage, Springer-Verlag GmbH, Heidelberg, 1995

[DSTG 2008]

Delta Software Technology GmbH, *ANGIE*, <http://www.d-s-t-g.de>, 2008

[Dyson und Longshaw 2004]

Dyson, Paul; Longshaw, Andrew, *Architecting Enterprise Solutions*, Wiley, 2004

[EAST 2004]

EAST, *Embedded Electronic Architecture*, <http://www.east-eea.net/>, 2004

[Eclipse 2008a]

The Eclipse Project, <http://www.eclipse.org/>, 2008

[Eclipse 2008b]

Eclipse Modeling Framework Project (EMF),
<http://www.eclipse.org/modeling/emf/>, 2008

[Eclipse 2008c]

Eclipse Process Framework Project (EPF),
<http://www.eclipse.org/epf/>, 2008

[Evans 2004]

Evans, Eric, *Domain-Driven Design*, Addison-Wesley, Boston, 2004

[EWITA 2003]

EWITA, *Enterprise Architecture and Related Definitions*,
http://www.ewita.com/EA_Overview/Definitions/ArchitectureDefinitions.htm, 2003

[Filman und Friedman 2000]

Filman, Robert E.; Friedman, Daniel P., *Aspect-Oriented Programming is Quantification and Obliviousness*, Proceedings of the Workshop on Advanced Separation of Concerns, OOPSLA 2000, Minneapolis, Minnesota, USA, 2000

[FODA 2005]

SEI, *Feature-Oriented Domain Analysis*, <http://www.sei.cmu.edu/domain-engineering/FODA.html>, 2005

[Foote und Yoder 1999]

Foot, Brian; Yoder, Joseph, *Big Ball of Mud*,
<http://www.laputan.org/mud/mud.html>, 1999

[Fowler 1996]

Fowler, Martin, *Analysis Patterns: Reusable Object Models*, Addison-Wesley, 1996

[Fowler 2002]

Fowler, Martin, *Public vs. published Interfaces*,
<http://www.martinfowler.com/ieeeSoftware/published.pdf>, 2002

[Fowler 2003]

Fowler, Martin, *The New Methodology*,
<http://www.martinfowler.com/articles/newMethodology.html#N10233>, 2003

[Fowler 2004]

Fowler, Martin, *Is Design Dead?*,
<http://www.martinfowler.com/articles/designDead.html>, 2004

[Fowler 2005]

Fowler, Martin, *Language Workbenches: The Killer-App for Domain Specific Languages?*, <http://martinfowler.com/articles/languageWorkbench.html>, 2005

[Foster et al. 2001]

Foster, I.; Kesselman, C.; Tuecke, S, *The Anatomy of the Grid: Enabling Scalable Virtual Organizations*, International Journal of Supercomputer Applications, 15(3), 2001

[Fuggetta et al. 1998]

Fuggetta, A.; Picco, G. P.; Vigna, G., *Understanding code mobility*, IEEE Transactions on Software Engineering, 24(5), 342-361, May 1998

[Fricke und Völter 2000]

Fricke, A.; Völter, M, *SEMINARS – A Pedagogical Pattern Language on how to Teach Seminars Efficiently*, <http://www.voelter.de/publications/seminars.html>, 2000

[Gamma et al. 1995]

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John, *Design Patterns*, Addison-Wesley, Reading, 1995

[Gau 2006]

Gau, Thosten, UMA und EPF: Einführung und Anwendung in der Praxis,
http://www.sigs.de/publications/os/2006/06/gau_OS_06_06.pdf, 2006

[Goedicke et al. 2000]

Goedicke, M.; Neumann, G.; Zdun, U., *Object system layer*, Proceedings of 5th European Conference on Pattern Languages of Programs (EuroPlop 2000), 397-410, Universitätsverlag Konstanz, Irsee, Germany, 2000

[Gray 1978]

Gray, J. N., *Notes on Database Operating Systems. Operating Systems: An Advanced Course*. Lecture Notes in Computer Science 60, 393-481, Springer-Verlag, 1978

[Greenfield und Short 2004]

Greenfield, Jack; Short, Keith, *Software Factories*, Wiley, 2004

[Grimes 1997]

Grimes, R.; *Professional DCOM Programming*, Wrox Press Inc., 1997

[Grosso 2001]

Grosso, W., *Java RMI*, O'Reilly & Associates, 2001

[Henning und Vinoski 1999]

Henning, Vinosiki, *Advanced CORBA Programming with C++*, Addison-Wesley, 1999

[Herbsleb und Grinter 1999]

Herbsleb, James D.; Grinter, Rebecca E., *Splitting the Organization and Integrating the Code: Conway's Law Revisited*, International Conference on Software Engineering, Los Angeles, 1999

[Herzberg 1966]

Herzberg, Friedrich, *Work and the Nature of Man*, Harpercollins, 1966

[Hofmeister et al. 1999]

Hofmeister, Christine; Nord Robert; Soni Dilip, *Applied Software Architecture*, Addison-Wesley, New York, 1999

[Hohpe und Woolf 2003]

Hohpe, Gregor; Woolf, Booby, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*, Addison-Wesley, New York, 2003

[IBM 2003]

IBM, *CICS (Customer Information Control System) Family*,

<http://www.ibm.com/software/htp/cics/>, 2003

[IBM 2004]

IBM, *WebSphere MQ Family*,

<http://www-306.ibm.com/software/integration/mqfamily/>, 2004

[IBM 2005]

IBM, <http://www-306.ibm.com/software/websphere/>, 2005

[IEEE 2000]

IEEE, *Recommended Practice for Architectural Description of Software-Intensive Systems*,

http://standards.ieee.org/reading/ieee/std_public/description/se/1471-2000_desc.html, 2000

[IEEE 2004]

IEEE, *Guide to the Software Engineering Body of Knowledge*,

<http://www.swebok.org/>, 2004

[IEEE 2007]

IEEE, *Recommended Practice for Architectural Description of Software-intensive Systems*, <http://www.iso-architecture.org/ieee-1471/>, 2007

[ISO10746 1998]

International Organization for Standardization, *Information technology – Open Distributed Processing – Reference model: Overview*,
<http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=20696&ICS1=35&ICS2=80&ICS3=>, 1998

[ISO17799 2001]

International Organization for Standardization, *Information technology – Code of practice for information security management*,
<http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=33441&ICS1=35&ICS2=40&ICS3=>, 2001

[iText 2008]

The iText Project, <http://www.lowagie.com/iText/>, 2008

[Jacobson et al. 1999]

Jacobson, Ivar; Booch Grady; Rumbaugh James, *The Unified Software Development Process*, Addison-Wesley, New York, 1999

[JCC 2005]

JavaCC - the Java Compiler Compiler, <https://javacc.dev.java.net/>, 2005

[Jekkle et al. 2004]

Jekkle, Mario; Rupp, Chris; Hahn, Jürgen; Zengler, Barbara; Queins, Stefan, *UML 2 glasklar*, Hanser, München, 2004

[Jini 2003]

The Jini Community, *Jini Community Homepage*, <http://www.jini.org/>, 2003

[Johnson und Foote 1988]

Johnson, R. E.; Foote, B., *Designing reusable classes*, Journal of object-oriented programming, 1(2), 22-35, 1988

[Jones et al. 1993]

Jones, N. D.; Gomard, C. K.; and Sestoft, P, *Partial Evaluation and Automatic Program Generation*, Prentice Hall, Englewood Cliffs, NJ, 1993

[Jones 2004]

Jones, Judith, *Architecting the Enterprise, Developing Architecture Skills*, http://www.opengroup.org/architecture/0404brus/presents/jones/Developing_Architecture_Skills1.pdf, TOGAF, 2004

[Kazman et al. 1994]

Kazman, Rick; Bass, Len; Abowd, Gregory; Webb, Mike, *SAAM: A Method for Analyzing the Properties Software Architectures*, Proceedings of the 16th International Conference on Software Engineering, Sorrento, 1994

- [Kazman et al. 1998]
 Kazman, R.; Klein, M.; Barbacci, M.; Lipson, H.; Longstaff, T.; Carrière, S.J., *The Architecture Tradeoff Analysis Method*, Proceedings of ICECCS, Monterey, 1998
- [Keller 1997]
 Keller, Wolfgang, *Mapping Objects to Tables: A Pattern Language*, Proceedings of the 1997 European Pattern Languages of Programming Conference, Irsee, Germany, Siemens Technical Report 120/SW1/FB, 1997
- [Keller 2002]
 Wolfgang Keller, *Enterprise Application Integration, Erfahrungen aus der Praxis*, dpunkt.verlag, Heidelberg, 2002
- [Keller und Coldevey 1998]
 Keller, Wolfgang; Coldevey, Jens, *Accessing Relational Databases: A Pattern Language*, In Martin, Robert; Riehle, Dirk; Buschmann, Frank (Eds.): Pattern Languages of Program Design 3, Addison-Wesley, 1998
- [Kelter et al. 2005]
 Kelter, Udo; Wehren, Jürgen; Niere, Jörg, *A Generic Difference Algorithm for UML Models*, Proceedings of SE 2005, Essen, Germany, 2005
- [Kelter 2007]
 Kelter, Udo, *Lehrmodul Dokumentdifferenzen*, Fachgruppe Praktische Informatik, Universität Siegen, 2007
- [Kelter et al. 2008]
 Kelter, Udo; Schmidt, Maik; Wenzel, Sven, *Architekturen von Differenzwerkzeugen für Modelle*, GI-Fachtagung Software Engineering 2008 (Proceedings), 2008
- [Kiczales et al. 1991]
 Kiczales, Gregor; des Rivieres, Jim; Bobrow, Daniel G., *The Art of the Metaobject Protocol*, MIT Press, 1991
- [Kiczales et al. 1997]
 Kiczales, Gregor; Lamping, John; Mendhekar, Anurag; Maeda, Chris; Videira Lopes, Cristina; Loingtier, Jean-Marc; Irwin, John, *Aspect-Oriented Programming*, ECOOP 1997, 220-242, 1997
- [Kiczales et al. 2001]
 Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J.; Griswold, G., *Getting Started with AspectJ*, Communications of the ACM, 44 (10), 59–65, 2001
- [Kieser und Kubicek 1993]
 Kieser, Alfred; Kubicek, Herbert, *Organisation*, Schäffer-Poeschel, 1993
- [Kniesel et al. 2002]
 Kniesel, G.; Noppen, J.; Mens, T.; Buckley, J., *The first workshop on unanticipated software evolution (USE 2002)*, ECOOP 2002 Workshop Reader, Springer Verlag, LNCS 2548, 2002
- [Kruchten 2000]
 Kruchten, Philippe, *The Rational Unified Process - An Introduction Second Edition*, Addison-Wesley, Boston, 2000

[Larman 2002]

Larman, Craig, *Applying UML and Patterns*, Second Edition, Prentice Hall PTR,
Upper Saddle River, NJ, 2002

[Leffingwell et al. 2003]

Leffingwell, Dean; Widrig, Don, *Managing Software Requirements: - A use case approach*, Addison-Wesley Professional, 2003

[Lehmann und Belady 1985]

Lehman, M.M.; Belady, L.A., *Program Evolution - Processes of Software Change*, Academic Press, London, 1985

[Lieberherr und Holland 1989]

Lieberherr, Karl; Holland, Ian, *Assuring Good Style for Object-Oriented Programs*, IEEE Software, 38-48, September 1989

[Lieberman 2007]

Lieberman, Ben, *Analyzing use cases by architectural relevance*,
<http://citeseer.ist.psu.edu/medvidovic97domains.html>, Januar 2007

[Linthicum 2001]

Linthicum, David S.; *B2B Application Integration, e-Business-Enable Your Enterprise*, Addison-Wesley, New York, 2001

[Liskov 1988]

Liskov, B, *Data Abstraction and Hierarchy*, SIGPLAN Notices, 23(5), May 1988

[Maier und Rechtin 2000]

Maier M.; Rechtin E., *The Art of Systems Architecting*, Second Edition, CRC Press, 2000

[Malveau und Mowbray 2001]

Malveau, Raphael; Mowbray, Thomas, J, *Software Architect Bootcamp*, Prentice Hall, London, 2001

[Martin 2000]

Martin, Robert C., *Design Principles and Design Patterns*,
http://www.objectmentor.com/resources/articles/_Principles_and_Patterns.PDF, 2000

[McCabe 1976]

McCabe, Thomas J, *A Complexity Measure*, IEEE Transactions on Software Engineering, 2(4), 308-320, 1976

[Medvidovic und Rosenblum 1997]

Medvidovic, Nenad; Rosenblum, David, S., *Domains of Concern in Software Architectures and Architecture Description Languages*,
<http://citeseer.ist.psu.edu/medvidovic97domains.html>, 1997

[Medvidovic und Taylor 1997]

Medvidovic, Nenad; Taylor, Richard, N., *A Classification and Comparison Framework for Software Architecture Description Languages*,
<http://citeseer.ist.psu.edu/medvidovic97framework.html>, 1997

- [MetaEdit 2008]
MetaEdit+, <http://www.metacase.com>, 2008
- [Meyer 1997]
Meyer, Bertrand; *Object-Oriented Software Construction, second edition*, New Jersey, Prentice Hall, 1997
- [Microsoft 2003]
Microsoft, *Microsoft Transaction Server (MTS)*,
<http://www.microsoft.com/com/tech/MTS.asp>, 2003
- [Microsoft 2004a]
Microsoft, *Microsoft .NET*, <http://www.microsoft.com/net/>, 2004
- [Microsoft 2004b]
Microsoft, *MSMQ Microsoft Message Queue Server*,
<http://www.microsoft.com/msmq/default.htm>, 2004
- [Miller 1956]
Miller, G., *The Magical Number Seven, Plus Or Minus Two: Some Limits on Our Capacity for Processing Information*, The Psychological Review, 63(2), 81-97, 1956
- [Morgan 1960]
Vitruvius, Morgan Morris (Übersetzer), *Ten Books on Architecture*, Dover Publications, 1960
- [Nosek und Palvia 1990]
Nosek, J.; Palvia, P., *Software Maintenance Management: Changes in the Last Decade*, Journal of Software Maintenance, 2(3), 157-174, 1990
- [OASIS 2002]
Organization for the Advancement of Structured Information Standards (OASIS),
UDDI Version 3.0 Published Specification, <http://www.uddi.org/specification.html>, 2002
- [OAW 2005]
The openArchitectureWare Generator Framework,
<http://www.openarchitectureware.org>, 2005
- [Oestereich 2004]
Oestereich, Bernd, *Objektorientierte Softwareentwicklung - Analyse und Design mit der UML 2.0*, Oldenbourg Verlag, München, 2004
- [Oestereich 2006]
Oestereich, Bernd, *Objektorientierte Softwareentwicklung - Analyse und Design mit der UML 2.1*, Oldenbourg Verlag, München, 2006
- [Oestereich und Weiss 2008]
Oestereich, Bernd; Weiss, Christian, *APM - Agiles Projektmanagement*, dpunkt.verlag, Heidelberg, 2008
- [OMG 2004]
Object Management Group, *Common Object Request Broker Architecture (CORBA/IOP)*, <http://www.omg.org/technology>, 2004

[OMG 2005a]

Object Management Group, *UML Profile for Enterprise Distributed Object Computing (EDOC)*, <http://www.omg.org/technology/documents/formal/edoc.htm>, 2005

[OMG 2005b]

Object Management Group, *CORBA Component Model, V3.0*,
<http://www.omg.org/technology/documents/formal/components.htm>, 2005

[OMG 2005c]

Object Management Group, *UML 2.0 Superstructure Specification*,
<http://www.omg.org/docs/formal/05-07-04.pdf>, 2005

[OMG 2005d]

Object Management Group, *Software Process Engineering Metamodel Specification, v1.1*, <http://www.omg.org/docs/formal/05-01-06.pdf>, 2005

[OMG 2006a]

Object Management Group, *UML 2.0 Infrastructure Specification*,
<http://www.omg.org/docs/formal/05-07-05.pdf>, 2006

[OMG 2006b]

Object Management Group, *Object Constraint Language 2.0*,
<http://www.omg.org/docs/formal/06-05-01.pdf>, 2006

[OMG 2007a]

Object Management Group, *Model Driven Architecture*,
<http://www.omg.org/mda/>, 2007

[OMG 2007b]

Object Management Group, *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*, <http://www.omg.org/docs/ptc/07-07-07.pdf>, 2007

[OMG 2007c]

Object Management Group, *Software and Systems Process Engineering Metamodel Specification, v2.0 (Beta 2)*,
<http://www.omg.org/docs/ptc/07-11-01.pdf>, 2007

[OMG 2007d]

XML Metadata Interchange (XMI), v2.1.1,
<http://www.omg.org/technology/documents/formal/xmi.htm>, 2007

[OMG 2008a]

Object Management Group, *Common Warehouse Metamodel (CWM) Specification*,
<http://www.omg.org/technology/cwm/>, 2008

[OMG 2008b]

Object Management Group, *Meta-Object Facility (MOF)*
<http://www.omg.org/mof/>, 2008

- [Opengroup 1999]
Opengroup, *Architecture Description Languages: An Overview*,
http://www.opengroup.org/architecture/togaf/bbs/9910wash/adl_over.ppt, 1999
- [Opengroup 2001]
Opengroup, *Developing Architecture Views - Introduction*,
http://www.opengroup.org/public/arch/p4/views/vus_intro.htm, 2001
- [Opengroup 2008a]
Opengroup, *The Open Group Architecture Framework*,
<http://www.opengroup.org/togaf/>
- [Opengroup 2008b]
IT Architect Certification Program
<http://www.opengroup.org/itac/>, 2008
- [OPERA 2005]
Opera Software, <http://www.opera.com/>, 2005
- [Osek 2005]
OSEK/VDX, <http://www.osek-vdx.org/>, 2005
- [OSF 1991]
Open Software Foundation, *DCE Application Development Guide*, Revision 1.0, Cambridge, MA, 1991
- [OSSJ 2004]
OSS through Java Initiative, <http://www.ossj.org/>, 2004
- [Papoulias 2000]
Papoulias, Athanasios; *Architekturbeschreibungssprachen im Hinblick auf die Bedeutung des Verteilungsaspektes am Beispiel*, Diplomarbeit am Fachbereich Informatik Universität Dortmund Lehrstuhl Software-Technologie, 2000
- [Parnas 1976]
Parnas, David L., *On the Design and Development of Program Families*, IEEE Transactions on Software Engineering, 1976
- [Parnas et al. 1986]
Parnas, David L., Clements, Paul, Naval Research Laboratory, *A rational Design Process: How and Why to fake it*, 1986
- [Parnas 1994]
Parnas, David. L., *Software Aging*, In Proceedings of ICSE 1994, Sorrento, Italy, 1994
- [Perry und Wolf 1992]
Perry, Dewayne E.; Wolf, Alexander L., *Foundations for the Study of Software Architecture*, ACM SIGSOFT Software Engineering Notes, 17(4), October, 1992
- [Petzold und Sieber 1993]
Petzold Hilarion G.; Sieper Johanna, *Integration und Kreation*, Junfermann, 1993

[Pohl et al. 2005]

Pohl, Klaus; Böckle, Günter; Van der Linden, Frank, *Software Product Line Engineering. Foundations, Principles, and Techniques*, Springer, Berlin, 2005

[Popma 2004a]

Popma R., *Jet tutorial part 1 - introduction to jet*,
http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html, 2004

[Popma 2004b]

Popma R., *Jet tutorial part 2 (write code that writes code)*,
http://www.eclipse.org/articles/Article-JET2/jet_tutorial2.html, 2004

[Pree 1995]

Pree, W., *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, 1995

[Raistrick et al. 2004]

Raistrick, Chris; Francis, Paul; Wright, John; Carter, Colin; Wilkie, Ian, *Model driven architecture with Executable UML*. Cambridge University Press, 2004

[Rausch et al. 2007]

Rausch, Andreas; Broy, Manfred; Bergner, Klaus; Höhn, Reinhard; Höppner, Stephan, *Das V-Modell XT. Grundlagen, Methodik und Anwendungen*, Springer, Heidelberg, 2007

[Rechtin 1991]

Rechtin, Eberhard, *Systems Architecting - Creating and building complex systems*, CRC Press, 1991

[Roxio 2003]

Roxio, Inc., *The Napster Homepage*, <http://www.napster.com>, 2003

[Royce 1970]

Royce, Winston, Managing the Development of Large Systems, IEEE WESCON Proceedings, 26, 1-9, 1970

[Rozanski und Woods 2005]

Rozanski, Nick und Woods, Eoin, *Software Systems Architecture*, Addison-Wesley, 2005

[Rüping 2004]

Rüping, Andreas, *Insights into Decision Making*, Proceedings of 9th European Conference on Pattern Languages of Programs (EuroPLoP 2004), 1-26, Irsee, Germany, 2004

[Samek 2002]

Samek, Miro, *Practical Statecharts in C/C++*, CMP Books, 2002

[Schmidt et al. 2000]

Schmidt, Douglas C.; Rohnert, Hans; Stal, Michael; Buschmann, Frank, *Pattern-Oriented Software Architecture Vol. 2, Patterns for Concurrent and Networked Objects*, John Wiley & Sons, New York, 2000

[Schneier 2001]

Schneier, Bruce, *Secret & Lies, IT-Sicherheit in einer vernetzten Welt*, dpunkt. verlag, Heidelberg, 2001

[SEI 2004]

Carnegie Mellon University Software Engineering Institute, *How Do You Define Software Architecture?*

<http://www.sei.cmu.edu/architecture/definitions.html>, 2004

[Shaw und Garlan 1994]

Shaw, Mary; Garlan, David, *Characteristics of Higher-level Languages for Software Architecture*, Technical Report CMU-CS-94-210, School of Computer Science, Carnegie Mellon University, Pittsburgh, 1994

[Shaw und Garlan 1996]

Shaw, Mary; Garlan, David, *Software Architecture - Perspectives on an Emerging Discipline*, Prentice Hall, Upper Saddle River, N. J., 1996

[Siedersleben 2006]

Siedersleben, Johannes; Moderne Software-Architektur, dpunkt.verlag, Heidelberg, 2006

[Stachowiak 1973]

Stachowiak, Herbert, *Allgemeine Modelltheorie*, Springer, Wien, 1973

[Stahl 2002]

Stahl, Eberhard, *Dynamik in Gruppen*, BeltzPVU, 2002

[Stahl und Völter 2005]

Stahl, Thomas; Völter, Markus, *Modellgetriebene Softwareentwicklung*, dPunkt, 2005

[Standish 1994]

The Standish Group International Inc., *The CHAOS Report*,

http://www.standishgroup.com/sample_research/chaos_1994_1.php, 1994

[Schumacher et al. 2005]

Schumacher, Markus, Fernandez, Eduardo, Hybertson, Duane, Buschmann, Frank, Sommerlad, Peter. *Security Patterns: Integrating Security and Systems Engineering*, John Wiley & Sons, 2005

[Steiger und Lippmann 2003]

Steiger, Thomas; Lippmann, Erich (Hrsg.), *Handbuch angewandte Psychologie für Führungskräfte, Führungskompetenz und Führungswissen*, 2. Auflage, Springer, Berlin 2003

[Sun 1988]

Sun Microsystems, *RPC: Remote Procedure Call Protocol Specification*, Tech. Rep. RFC-1057, Sun Microsystems, Inc., June 1988

[Sun 2003]

Sun Microsystems, *Project JXTA*, <http://www.jxta.org>, 2003

[Sun 2004a]

Sun Microsystems, *Java Blue Prints*,

<http://java.sun.com/blueprints/enterprise/index.html>, 2004

[Sun 2004b]

Sun Microsystems, *Java Message Service (JMS)*,

<http://java.sun.com/products/jms/>, 2004

[Sun 2005]

Sun Microsystems, *Java 2 Platform - Enterprise Edition (J2EE)*,

<http://java.sun.com/j2ee/>, 2005

[Szyperski 1998]

Szyperski, Clemens; *Component Software - Beyond Object-Oriented Programming*,

Addison-Wesley, 1998

[Tanenbaum und van Steen 2003]

Tanenbaum, Andrew S.; van Steen, Maarten, *Distributed Systems*, Prentice Hall,

New York, 2003

[Tarr 2004]

Tarr, P., *Hyper/J*, <http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm>,

2004

[Taylor 1913]

Taylor, F.W., *The principles of scientific management*, 1913

[Terplan 2001]

Terplan, Kornel, *OSS Essentials, Support System Solutions for Service Providers*,

John Wiley & Sons, New York, 2001

[Tibco 2004]

Tibco, *Messaging Solutions*,

http://www.tibco.com/software/enterprise_backbone/messaging.jsp, 2004

[TMF 2004a]

TeleManagement Forum, *Next Generation Operations Support Systems Initiative*

(NGOSS), <http://www.tmforum.org/browse.asp?catID=1911>, 2004

[TMF 2004b]

Strassner, John; Fleck, Joel; Huang, Jenny; Fauer, Cliff; Richardson, Tony, Tele-
Management Forum, *TMF White Paper on NGOSS and MDA*,

<http://www.tmforum.org/browse.asp?catID=1875&sNode=1875&Exp=Y&linkID=28972>, 2004

[Torkler 2001]

Torkler, Stephan, *Acme: Beschreibung der Architektur komponentenbasierter*

Systeme, Seminar 1910 Komponenten-Software: Konzepte und Programmierung

FernUniversität – Gesamthochschule Hagen Wintersemester 2000/2001

[Transarc 2000]

Transarc, *Encina*, <http://www.transarc.com>, 2000

- [UPNP 2004]
UPNP Forum, *Universal Plug and Play*, <http://upnp.org/>, 2004
- [Van Deursen et al. 2007]
Van Deursen, A.; Visser, E.; Warmer, J., *Model-Driven Software Evolution: A Research Agenda*, In Dalila Tamzalit (Eds.), *Proceedings 1st International Workshop on Model-Driven Software Evolution (MoDSE)*, Nantes, France, 2007
- [Vinoski 2003]
Vinoski, S.; IEEE Internet Computing, *Toward Integration Column: Integration With Web Services*, November/Dezember 2003
- [Vogel und Zdun 2002]
Vogel, Oliver; Zdun, Uwe, *Content Conversion and Generation on the Web: A Pattern Language*, EuroPLoP, 2002
- [Völter et al. 2002]
Völter, Markus; Schmid, Alexander; Wolf, Eberhard, *Server Component Patterns*, John Wiley & Sons, New York, 2002
- [Völter et al. 2004]
Völter, Markus; Kircher, Michael; Zdun, Uwe, *Remoting Patterns - Foundations of Enterprise, Internet, and Realtime Distributed Object Middleware*, John Wiley & Sons, 2004
- [Vroom und Yetton 1976]
Vroom, Victor H.; Yetton, Philip W., *Leadership and Decision-Making*, University of Pittsburgh Press, Pittsburgh, 1976
- [W3C 1999]
W3C XML Path Language (XPath), v. 1.0, <http://www.w3.org/TR/xpath>, 1999
- [W3C 2004]
W3C, *XML Schema*, <http://www.w3.org/XML/Schema>, 2004
- [W3C 2006]
W3C Extensible Stylesheet Language (XSL) Specification, v. 1.1, <http://www.w3.org/TR/xsl/>, 2006
- [Weiss und Lai 1999]
Weiss, D. M.; Lai, C. T. R., *Software Product-Line Engineering: A Family Based Software Development Process*, Addison-Wesley, 1999
- [Wiegers 2003]
Wiegers, Karl E., *Software Requirements*, Second Edition, Microsoft Press, 2003
- [Wikipedia 2008a]
Systemdefinition, <http://de.wikipedia.org/wiki/System>, 2008
- [Wikipedia 2008b]
Definition zu Interesse, <http://de.wikipedia.org/wiki/Interesse>, 2008
- [Winer 1999]
Winer, D., *XML-RPC Specification*, <http://www.xmlrpc.com/spec>, 1999

[Yourdon 2004]

Yourdon, Edward, *Death March*, Prentice Hall, New York, 2004d

[Yourdon 1997]

Yourdon, Edward, *Death March, The Complete Software Developer's Guide to Surviving "Mission Impossible" Projects*, Prentice Hall, Upper Saddle River, N. J., 1997

[Yourdon und Constantine 1978]

Yourdon, E.; Constantine, L., *Structured Design: Fundamentals of a Discipline of Computer Programming and Design*, Prentice Hall, 1978

[Zachman 1987]

Zachman, John, A., *A Framework for Information Systems Architecture*, IBM Publication, 1987

[Zdun 2004]

Zdun, Uwe, *Pattern Language for the Design of Aspect Languages and Aspect Composition Frameworks*, IEE Proceedings Software, 151(2), 67- 83, 2004

[Zdun et al. 2006]

Zdun, U. Henrich, C. van der Aalst, W.M.P. *A Survey of Patterns for Service-Oriented Architectures*, International Journal of Internet Protocol Technology, 1(3), Inderscience, 2006

[Zdun und Hendrich 2006]

Hendrich, C. Zdun, Uwe, *Patterns for Process-Oriented Integration in Service-Oriented Architectures*, Proceedings of 11th European Conference on Pattern Languages of Programs (EuroPLoP 2006), Irsee, Germany, 2006

[Zito et al. 2006]

Zito, A.; Diskin, Z.; Dingel, J. *Package merge in UML 2: Practice vs. theory?* Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006), Genova, Italy, 2006

| Index

2-Tier-Architektur	224	Architecture Tradeoff Analysis
3-Tier-Architektur	225	Method 400
4+1-Sichtenmodell <i>siehe</i>		Architekt
Architektur-Sicht		Akzeptanz 334
A		Aufgabenfeld 24
Abstraktion	142	Entwicklungspfad 335
ACID-Eigenschaft	292	Generalist 334
Änderung		Kompetenz 335
erwartbare	135	Rolle 17, 23, 37, 332
nicht-erwartbare	135	Architektenteam 335
Anforderung	32, 104	architektonischer
Abstraktionsniveau	119	Ordnungsrahmen 17, 23
als Kraft	104	als gemeinsames
architekturelle 119, 369		Vokabular 30
Art	32, 107	Architekt im Mittelpunkt 25
Beziehung	110	Dimension 26
Checkliste	376	Erklärungsmodell 23, 26
Definition	105	in der Praxis 28
Detaillierungsgrad 110, 119		Verständnis von
Eigenschaft	105	Architektur 26
funktionale	32, 107	Visualisierung 28
im Architektur-Kontext 119		architektonisches
klassische	43	Handeln 25, 28
nicht-funktionale	32, 108	architektonisches Konzept 152
Organisations-		Checkliste 403
anforderung	80	Architektur
Spannungsfeld	104	als Kompromiss 45
Wechselwirkung	110	als Resultat von
Anforderungskatalog	106	Entscheidungen 328
Anwendungsarchitektur	53	als Struktur 43
Anwendungslogik	388	als Struktur und
Anwendungsserver	304	Disziplin 50
Anwendungsszenario	38	als Tätigkeit 43
Enterprise Application		Architektur-Denken 8, 61f
Integration	428	Architektur-Disziplin 30, 43,
AOP	139	54, 334
Apollo-Team	325	Baukunst 30
Architecting <i>siehe</i>		Big Ball of Mud 4
Architekturieren		Definition 9, 42
		Definition als Disziplin 50

Definition als Struktur.....	49	WO-Dimension.....	71
Definition nach		WOMIT-Dimension.....	125
Bass et al.	48	Zusammenhang.....	29
Definition nach IEEE.....	49	Architektur-Disziplin ...	30, 43,
Einfluss von		54, 334	
Anforderungen.....	43	Zusammenspiel	57
Einfluss von Architektur-		Architektur-Dokumentation	
Mitteln	43	417	
Entwicklungsprozess.....	342	Architektur-Richtlinie	417
fachliche.....	52, 382	audio-visuelle.....	423
Folgen mangelhafter		Auswahl der Mittel.....	420
Architektur	7	Checkliste	427
Fragen zur.....	3	Erstellung	419, 421
Gemeinsamkeiten mit klas-		Inhalt	418
sischer Architektur.....	45	Kontext	419
Grundlagen	30f	Notwendigkeit	417
inhärente Komplexität.....	7	Richtlinie	414
klassische.....	42	Umfang	424
Kommunizieren der.....	414	unzureichende.....	421
Makro-Architektur	78	Verwaltung	424
mangelhafte	6	visuelle.....	423
Mikro-Architektur.....	78	Vorlage.....	421
Orientierungslosigkeit.....	3	Zielgruppe.....	420
Plattformarchitektur.....	53	Zielsetzung	417
Pseudo-Architektur.....	7	Architektur-Ebene ...	10, 31, 72,
Resultat von Erfahrung...336		119	
soziale und organisatorische		Bausteinebene	73, 82
Einflussfaktoren.....	313	Ebenenmodell	73
technische.....	52, 383	Ebenenwechsel	82
und Kundenseite.....	8	Organisationsebene...73f, 80	
vs. Entwurf.....	77	Software Design Level Model	
Zertifizierung.....	336	76	
Architektur-Anforderung		Systemebene.....	73f, 81
<i>siehe</i> Anforderung		und architektonischer	
Architektur-Beurteilung	399	Ordnungsrahmen	76
szenariobasierte.....	400	Architektur-Entscheidung... 24,	
Architektur-Bewusstsein.....	25	328, 418	
Architektur-Denken.....	8, 61f	Art und Methode	330
Architektur-Dimension.....	26	Definition allgemein.....	328
WARUM-Dimension.....	103	Entscheidungsbereich....	331
WAS-Dimension.....	41	Entscheidungsfreudigkeit	
WER-Dimension.....	311	330	
WIE-Dimension.....	341	Entscheidungsprozess....	328

Informationsmenge.....	330
strategischer Charakter	331
Superlogik.....	331
Architektur-Entwurf.....	377
Einflussfaktor.....	377
Architektur-Erfahrung.....	43
Architekturieren	50
Architektur-Konformität....	410
Architektur-Konzept	
Aspektorientierung.....	181
dynamische Sprache.....	185
generative Erzeugung von	
Systembausteinen.....	166
Generierung	166
Komponenten-	
orientierung	161
MDSD	170
Meta-Architektur.....	164
Objektorientierung.....	155
prozeduraler Ansatz.....	153
Reflection	164
Skriptsprache.....	185
Wartung	189
Architektur-Metamodell	279
Architektur-Mittel.....	33, 436
Basisarchitektur.....	35, 216
Konzept	34, 152
Modellierungsmittel	36, 264
Muster	34, 194
Prinzip	33, 128
Referenzarchitektur	35, 253
Stil	34, 194
Taktik	34, 194
Technologie.....	36, 291
Architektur-Modellierungs-	
mittel	36, 264
Architecture Descripion	
Language.....	279
Architektur-	
Metamodell.....	279
Checkliste.....	404
Glossar.....	279
Liste der.....	264
Unified Modeling	
Language.....	268
Architektur-Muster.....	34, 202,
205	
Checkliste	403
Architektur-Perspektive.....	31,
430	
Architektur-Prinzip	33, 129
Abstraktion	142
Bezug zu Anwendungs-	
fällen	150
Entwurf für	
Veränderung	135
Information Hiding	140
Inkrementalität.....	149
Kohäsion	133
Konsistenz	150
Konvention anstatt	
Konfiguration.....	150
Kopplung.....	130
Modularität	145
Rückverfolgbarkeit.....	148
Selbstdokumentation.....	148
Separation of Concerns... ..	137
Vermeidung überflüssiger	
Komplexität.....	150
Architektur-Prototyp	373, 400,
446	
Architektur-Richtlinie	417
Checkliste	426
Architektur-Sicht.....	31, 83, 94,
365, 375, 401	
4+1-Sichtenmodell.....	98
Abhängigkeit.....	93
abstraktes Architektur-	
Sichtenmodell	90
Anforderungssicht.....	91
Ausführungssicht	90
bei IEEE.....	84
Checkliste	427
Datensicht	92
Definition	83
grundlegende.....	89

im Entwicklungsprozess	87	
konzeptionelle Sicht.....	90	
logische Sicht.....	90f	
Prozesssicht.....	92	
Reihenfolge.....	88	
Relevanz.....	87	
RM-ODP.....	97	
Spezifikation	89	
Standpunkt	86	
Umsetzungssicht	92	
und Architektur-Tätigkeit	355	
Verteilungssicht.....	93	
Zachman-Framework.....	94	
Architektur-Sichtenmodell ...	32	
Architektur-Stil.....	34, 43, 199	
Architektur-Struktur	438	
Architektur-Taktik.....	34, 197	
Architektur-Tätigkeit.....	496	
architekturverbessernde		
Maßnahme	191	
Architektur-Vision.....	365	
Architektur-Vorgehen	37f, 350	
Checkliste	401	
Entwerfen der		
Architektur	353, 377	
Entwurfsaktionen	378	
Erstellen der		
Systemvision.....	352, 358	
iterativ-inkrementell.....	355	
Kommunizieren der		
Architektur	354	
Tätigkeit.....	38, 351	
Umsetzen der		
Architektur	354, 406	
Verstehen der		
Anforderungen	367	
Aspektorientierung....	181, 467	
Advice.....	184	
AOP	139	
Crosscutting Concern	181	
Introduction	184	
Joinpoint.....	183	
Pointcut	184	
ATAM <i>siehe</i> Architecture		
Tradeoff Analysis Method		
Aufbauorganisation.....	452	
Ausführungsumgebung.....	68	
B		
Basisarchitektur	35, 216	
Bausteinanforderung...33, 107,		
114		
funktionale.....	108, 114	
nicht-funktionale	114	
Bausteinebene	494	
Bedienbarkeit	115	
Betreibbarkeit.....	480, 484	
Big Ball of Mud.....	4	
Black-Box-Prinzip.....	142	
Broker-Muster	208	
Buch		
E-Mail.....	21	
Fallstudien.....	14	
Kapitelüberblick.....	15	
Lesereihenfolge Teil II.....	16	
Lesereihenfolge Teil III.....	16	
Motivation.....	2	
Teil I im Detail.....	17	
Teil II im Detail.....	18	
Teil III im Detail.....	19	
Web Site.....	21	
Ziel	3f	
Zielpublikum	15	
Bucharchitektur.....	13	
Landkarte zu Teil II.....	14	
Modell für Themengebiete	12	
Budget.....	118	
C		
CGI-Schnittstelle.....	304	
Checkliste		
für Anforderungen.....	376	
für Architektur- Konzepte		
403		
für Architektur-		
Dokumentation.....	427	

für Architektur-Muster ..	403	Domäne	175, 463
für Architektur-Richtlinien	426	Domänenlogik.....	388
für Architektur-Sichten	427	DTD	301, 465
für das Entwerfen der		dynamische Sprache.....	185
Architektur.....	402	für Webanwendungen	188
für das Erstellen der		dynamische Web-Seite	303
Systemvision.....	366		
für das Kommunizieren der		E	
Architektur.....	426	EDI	301
für Architektur-		Emergenz	59
Modellierungsmittel...	404	Enterprise Application	
für Referenzarchitekturen	404	Integration	428
für Technologien	404	Adapter	440
für das Umsetzen der		Administrations- und	
Architektur.....	412	Monitoring-Schicht.....	441
für das Verstehen der		Anforderungen.....	434
Anforderungen.....	376	Ansatz	429
Client-/Server-Modell.....	223	B2B-Applikations-	
Container.....	234	integration	429
Conways Gesetz	314, 317	Dienst.....	442
Crosscutting Concern	181	EAI-Bus	439
Crystal-Methodenfamilie ...	320	EAI-Muster.....	445
D		ETL-Werkzeug.....	429
Data Hiding	140	generisches	
Datenarchitektur.....	55, 483,	Datenformat.....	442
484		Gründe	430
Datenbank	298	Herstellerabhängigkeit...	445
Zugriff.....	299	Integrationsart	443
Zugriffsschicht	299	Kommunikationsschicht	441
Datenflussarchitektur.....	219	Kontext	429
Datenmodell	434	minimal-invasiv	444
Datenqualität.....	480, 483	Nachteile	432
Dekomposition	138	Prozessschicht	441
Dependency Injection	133, 161	Punkt-zu-Punkt-	
Dependency Inversion	133	Verbindung	429
Design-by-Contract	144	Risiken	432
Dezentralisierung	221	Routing	435, 441
Dienst	68, 395	Schicht	440
Domain Specific		Screen Scraping	443
Language.....	171, 187	Verbindungsschicht.....	440
		Vorteile	431
Enterprise Architecture Skills		Enterprise Framework.....	335

Enterprise Service Bus	446
Enterprise-Architektur	56
Entität.....	396
Entwerfen der Architektur	
Checkliste	402
Entwicklungszeit-	
anforderung	33, 109, 116
Erweiterbarkeit.....	117
Plattformunabhängigkeit	116
Skalierbarkeit.....	117
Wartbarkeit.....	117
Wiederverwendbarkeit	117
Entwurf für Veränderung	135f
Entwurfsmuster.....	205
Erklärungsmodell <i>siehe</i>	
architektonischer	
Ordnungsrahmen	
Erstellen der Systemvision	357
Checkliste	366
Erweiterbarkeit	117, 458, 479
ESB <i>siehe</i> Enterprise Service	
Bus	
Evolution der Software-	
Entwicklung.....	2
eXtreme Programming	321,
345	
F	
fachliche Architektur.....	52
Fallstudie	
CRM-Kundendatenbank ..	20,
473	
Risikofallmanagement-	
system.....	20, 449
Feature Driven Development	
345	
File Sharing.....	223
Forward Engineering	189, 192
Framework <i>siehe</i>	
Rahmenwerk	
FTP	453, 461
Führungsstil.....	334
Delegation	333
G	
gemeinsame Speicher	233
generative Erzeugung von	
Systembausteinen	166
generatives Verfahren.....	470
Generator.....	458
Generatorwerkzeug.....	464,
467	
Generierung.....	166
Gesetz von Demeter.....	132
Grid.....	296
Gruppe.....	324
als besondere	
Organisationsform.....	324
Arbeitssphase.....	327
Architekturententeam.....	335
Erfolgsfaktor.....	325
Gründungsphase	326
Gruppendynamik.....	326
Identifizierung.....	326
Orientierungsphase.....	327
Streitphase.....	327
Vertragsphase.....	327
H	
Hardware-Baustein.....	46, 64
Hochverfügbarkeit....	480, 484,
487	
Holismus.....	60
Systembetrachtung	49
Hollywood-Prinzip.....	133, 160
Hot Spots.....	136
HTML	469
HTTPS	461
I	
Individuum.....	36, 321
Arbeitszufriedenheit	317f
Einfluss auf Architektur	315
Einfluss durch	
Architektur	315
Hygienefaktor.....	317
Identität	321

Menschentyp.....	322
Motivator.....	317
Rolle.....	315, 324
Rollenverständnis	325
soziales Wesen.....	317
Information Hiding	140
Informationstechnologie.....	32
Inkrementalität.....	149
Integration.....	475, 478
Integrationsarchitektur.....	55,
429, 433, 435, 483f	
Interesse.....	84
Interessenvertreter	45, 332,
360f	
Interface <i>siehe</i> Schnittstelle	
Introspektion.....	164
Inversion of Control	160
IT-Richtlinie	112
IT-Standard.....	112
IT-System	
Baustein	47
Definition	46
K	
Kohäsion.....	133
Kollaborationswerkzeug....	317
Kommunikations-	
Middleware	231
Kommunikationsstruktur	317
Kommunizieren der	
Architektur.....	413
Checkliste.....	426
Komponente	162, 233
Komponentenmodellierung	
163	
Komponentenorientierung	161
Komponentenplattform....	233,
305	
.NET	307
CCM.....	307
JEE (J2EE)	305
Konsistenz.....	150, 480
Kopplung.....	130
L	
Laufzeitanforderung....	33, 109,
114, 509	
Bedienbarkeit.....	115
Leistungsverhalten	115
Sicherheit.....	116
Verfügbarkeit.....	115
Layer <i>siehe</i> Schicht	
Layers-Architektur-Muster	34,
440	
Legacy-System	433
Leistungsverhalten	115
Liskov-Substitutions-	
Prinzip.....	143
M	
Makro-Architektur.....	78
Makro-Sprache.....	165
Mainframe.....	222
MDA.....	176
MDSD.....	170, 412
Message-oriented-	
Middleware	294
Messaging.....	233
Meta-Architektur	164
Metamodell	458f, 464
Meta-Objekt-Protokoll.....	164
Meta-Programm	164
Middleware	229, 292, 479
Mikro-Architektur	78
Mind-Mapping	330
minimal-invasiv.....	161
mobiler Code.....	296
Modell	171
Definition	265
modellgetriebene Software-	
Entwicklung.....	170
Modellierungsmittel	
<i>siehe</i> Architektur-	
Modellierungsmittel	
Modularität	145
Modularitätsansatz.....	145
Monolith.....	216

Motivation.....	333	Hierarchie.....	317
Muster.....	202f	ökonomisch-rationales	
Kraft.....	204	Verständnis	316
verwandte	211	Organisationskultur	313, 320
zusammengesetzte	211	Organisationsstruktur...	314,
Mustersequenz	212	319	
Mustersprache.....	211f	systemisches Verständnis	
Mustersystem	212	318	
 N		verhaltenswissenschaft-	
Netzwerkarchitektur	55	liches Verständnis.....	317
Next-Generation-Operation-		Verständnis	316
Support-Systems.....	258	Organisationsanforderung	33,
n-Tier-Architektur.....	225	107, 111	
 O		funktionale.....	108, 111
Object-Relational-		nicht-funktionale.....	112
Mapping	299	Organisationsebene.....	493
Objektorientierung	155	Organismusmuster....	37, 320
Aggregation.....	157	ArchitectAlsoImplements.	334
Assoziation.....	157	Organisationsstruktur.....	314,
Klasse	156	319, 452	
Objekt.....	156	organisatorische Rahmen-	
Objektidentität.....	156	bedingung	33, 110, 118, 512
Polymorphie.....	158	Budget.....	118
Schnittstelle und abstrakte		Projektplanung.....	118
Klasse	157	Stellenbesetzung	118
Vererbung	157	OSS-for-Java-Initiative	260
Offen-/Geschlossen-		Outsourcing	471
Prinzip.....	147	 P	
Offshoring	314	Peer-To-Peer-Architektur....	228
OODBMS	298	Peer-to-Peer-System	296
OO-RPC-Middleware-		Performanz <i>siehe</i>	
System.....	293	Leistungsverhalten	
Organisation	36, 316	Persistenzanforderung.....	297
Arbeitsbedingung	317	Pipes and Filters.....	200
Arbeitszufriedenheit.....	321	Plattform.....	67
Definition	316	Ausführungsumgebung....	68
Einfluss auf Architektur	313	Dienst.....	68
Einfluss durch		Plattformarchitektur.....	53
Architektur	315	Plattformunabhängigkeit ...	116
formelle und informelle		Polymorphie	158
Struktur.....	319	Prinzip der hohen	
		Kohäsion.....	134

Prinzip der losen	synchron	293
Kopplung.....		131
Product Line Engineering	Rückverfolgbarkeit.....	148
177,		
472		
Produktlinienarchitektur...		258
Projektplanung.....		118
Prototyp		
Architektur-Prototyp.....		400
Prototyping.....		150
Proxy-Muster.....		206
prozeduraler Ansatz.....		153
Prozedurdefinition		153
Publish/Subscribe.....		228
Q		
Qualitätsattributszenario..		196,
371		
R		
Rahmenwerk.....		160, 412
Reduktionismus.....		60
Reengineering.....		190
Werkzeug		192
Referenzarchitektur....		35, 253,
387, 440, 456		
Anforderung.....		257
Art		257
Bestandteil		255
Checkliste.....		404
Definition		254
Einsatz.....		256
Vorteil.....		256
Referenzmodell		255
Reflection		164
Remote-Procedure-Call		232
Repository		220
Restrukturierung		190
Reverse Engineering.....		190
Rich Client.....		226
RM-ODP		260
Robustheit.....		488
RPC.....		293
asynchron.....		293
S		
SAAM <i>siehe</i> Software		
Architecture Analysis		
Method		400
Schicht		66, 485, 489
Schichtenarchitektur.....		217
Schlüsselabstraktion		380
Schnittstelle 65, 394, 397, 437,		
478		
Analyse		437
explizite		65, 143
Fokus auf		142
generische.....		486
implizite		66
Komplexität		439
Reduzierung und		
Vereinheitlichung.....		476
Trennung von		
Implementierung		143
Schnittstellen-Segregations-		
Prinzip.....		143
schrittweises Wachsen.....		150
Scrum.....		320, 345
Selbstdokumentation.....		148
Separation of Concerns.....		137
serverseitige		
Komponentenplattform ..		233
serviceorientierte		
Architektur		236
Setzkasten <i>siehe</i>		
architektonischer		
Ordnungsrahmen		
Sicherheit.....		116
Sicherheitsarchitektur		55, 243
Skalierbarkeit		117, 479, 487
Skelettsystem		409
Skriptsprache.....		185
SOA <i>siehe</i> serviceorientierte		
Architektur		
SOAP		308

Software Architecture Analysis	
Method	400
Software-Architektur <i>siehe</i>	
Architektur	
Software-Baustein....	46, 48, 64
Beziehung	49
sichtbare Eigenschaft	48
Software Design Level Model	
76	
Software-Evolution.....	190
Software-Krise.....	6
Software-Muster.....	202
Software-Struktur	49
dynamische Struktur.....	49
statische Struktur	49
Software-Systemfamilie.....	177
Software-Wartung.....	189
Sozialkompetenz.....	37
Spiralmodell.....	345
spontanes Netzwerk	296
Sprachunterstützung für	
Abstraktionen	144
Stakeholder <i>siehe</i>	
Interessenvertreter	
Standpunkt	86
Stellenbesetzung.....	118
Streaming.....	233
Subsystem.....	58, 64, 389, 485
System.....	64
allgemeine Definition	58
geschlossenes System	59
Grenze.....	58
IT-Definition.....	46
offenes System.....	59
Wahl der Systemgrenze....	61
Ziel	58
Systemanforderung	33, 107, 113
architekturelle	369, 381
funktionale.....	108, 113, 362
nicht-funktionale ..	113, 362, 383
Priorisierung.....	374
wesentliche	361
Systemarchitektur	
Definition.....	51
Systembaustein	58, 62, 64
Art.....	62
Beziehung	64
Blickwinkel.....	63
fachlicher.....	384
Modell.....	63
nicht-tragender	78
Taxonomie	63
technischer.....	386
tragender	48, 78
Verteilung.....	397
Vokabular	63
Systemebene.....	493
Systemgrenze	58
Systemkontext.....	81, 363, 378, 452
System-Management-Architektur	55
Systemtheorie	57
T	
Taylorismus	317
TCP/IP.....	461
Team <i>siehe</i> Gruppe	
technische Architektur	52, 383
Technologie.....	36
Checkliste	404
Template-Sprache.....	304, 469
Thin Client.....	226
Transaktionsmonitor.....	292
Transaktionssicherheit.....	480, 484
Two-Phase Commit.....	292
U	
UDDI	309
Umsetzen der Architektur	406
Checkliste	412
Unified Software Development Process.....	319, 345, 348

Ausarbeitungsphase	349	WAS-Dimension.....	18, 26, 30, 38, 41, 121
Auslieferungsphase	349	Wasserfallmodell.....	343
Konstruktionsphase.....	349	Web Service.....	308
Konzeptionsphase.....	349	Web-Application-Server.....	304
Unternehmensarchitektur		Web-Content-Management	305
<i>siehe</i> Enterprise-Architektur		Web-Seite.....	303
V			
Verfügbarkeit.....	115	WER-Dimension.....	19, 27, 36, 121, 311, 521
Verstehen der Anforderungen		WIE-Dimension.....	19, 27, 37, 122, 341, 521
367		WO-Dimension...18, 27, 31, 71, 119, 522	
Checkliste.....	376	WOMIT-Dimension	27, 122, 125, 522
Verteilungsstil.....	232	Wrapping	190
Viewpoint <i>siehe</i> Standpunkt		WSDL	309
Vitruvius	43		
V-Modell XT	345		
Vorgehensmodell			
Adaption.....	347		
agiles	347		
Iterationsplanung.....	346		
iterativ-inkrementell.....	345		
W			
Wartbarkeit.....	117	XML.....	301
Wartung	189	XML-Schema.....	301
WARUM-Dimension.....	18, 27, 32f, 103, 521		
Z			
Zachman-Framework.....	94		
Zentralisierung	221		
Zertifizierung	336		
Zielpublikum	15		