Institut für Informatik
der Technischen Universität München

# Logische Architekturen

## Eine Theorie der Strukturen und ihre Anwendung in Dokumentation und Projektmanagement

*Gerd Hinrich Beneken*

Institut für Informatik
der Technischen Universität München

# Logische Architekturen

## Eine Theorie der Strukturen und ihre Anwendung in Dokumentation und Projektmanagement

*Gerd Hinrich Beneken*

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

| | |
|---|---|
| Vorsitzender: | Univ.-Prof. Dr. Hans Michael Gerndt |
| Prüfer der Dissertation: | |
| 1. | Univ÷Prof. Dr. Dr. h.c. Manfred Broy |
| 2. | Univ÷Prof. Bernd Brügge, Ph.D. |

Die Dissertation wurde am 14.01.2008 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 01.07.2008 angenommen.

# Kurzfassung


**Beitrag der Arbeit**

Architektur und Projektmanagement sind Mittel, um große Software-Entwicklungsprojekte beherrschbar zu machen. Die Verbindung zwischen beiden Disziplinen soll in der vorliegenden Arbeit hergestellt werden. Das wird als *architekturzentrisches Projektmanagement* bezeichnet. Damit wird die Zusammenarbeit von Architekt und Projektleiter verbessert: Ein Verfahren wird vorgeschlagen, mit dem die Architekturbeschreibung und die Projektplanung iterativ abgeglichen und verbessert werden können. Zusätzlich werden zwei Verfahren zur Optimierung der Aufgabenverteilung und zur Verringerung der Entwicklungsaufwände vorgestellt.

Die Dokumentation von Architekturen mithilfe von Architektursichten ist der zweite Schwerpunkt dieser Arbeit. Ein neuartiges Ordnungsschema erlaubt es, Architektursichten einzuordnen und zu vergleichen. Mathematisch fundierte Verfahren zur konsistenten Erzeugung von Architektursichten werden vorgestellt. Darüber hinaus schlägt die vorliegende Arbeit eine Reihe neuartiger Architektursichten vor, die das Projektmanagement und die Kommunikation innerhalb des Projektes unterstützen. Eine Sicht stellt etwa dar, welches Team für welche Komponente verantwortlich ist und erleichtert so das Auffinden von Ansprechpartnern in großen Projekten.

**Vorgehensweise**

Bereits während der Anforderungsanalyse in einem Software-Entwicklungsprojekt entstehen erste grobe Darstellungen der System- und Software-Architektur. Diese Grobentwürfe werden hier *logische Architekturen* genannt. Eine logische Architektur beschreibt, wie der Funktionsumfang des IT-Systems auf logische Komponenten aufgeteilt wird und wie diese über Konnektoren Nachrichten austauschen. Wie diese Bestandteile später in Software oder Hardware implementiert, wiederverwendet oder zugekauft werden, wird erst später in der technischen Architektur festgelegt. Die logische Architektur ist für die Planung, Organisation und Steuerung von Projekten die Grundlage: Das IT-System muss so in logische Komponenten zerlegt werden, dass diese arbeitsteilig spezifiziert, entworfen und entwickelt werden können. Die Arbeitsaufteilung geschieht typischerweise entlang der Komponentengrenzen.

Die hier vorgeschlagene Erzeugung von Sichten und die Vorschläge zum architekturzentrierten Projektmanagement basieren auf einer Architekturtheorie: Logische Architekturen von IT-Systemen werden als gerichtete Multigraphen formalisiert. Komponenten sind die Knoten und die Konnektoren sind die Kanten. Die hierarchische Struktur der Komponenten wird über eine spezielle Kantenmenge beschrieben. Den Komponenten und Konnektoren werden Zusatzinformationen mithilfe von Attributen zugeordnet. Diese Zusatzinformationen stammen beispielsweise aus der Projektplanung oder dem Projekt-Controlling, beschreiben Qualitätseigenschaften oder sind Informationen zur Qualitätssicherung. Diese Informationen werden zur Definition und Erstellung von Architektursichten genutzt. Zusätzlich werden zur logischen Architektur Anwendungsfälle als Kantenmengemenge (Sze-

narios) modelliert. Szenarios stellen dar, welche Komponenten und Konnektoren an der Erbringung eines Anwendungsfalls beteiligt sind. Dies schafft eine Verbindung zu den funktionalen Anforderungen.

Zu den logischen Architekturen werden Projektionsfunktionen zur Erzeugung von Sichten definiert. Die Projektionen sind eine formale Beschreibung der Blickwinkel (Viewpoints), welche zur Definition von Architektursichten (Views) verwendet werden. Die hier definierten Grundprojektionen können flexibel zu komplexeren Projektionen verkettet werden. Die Projektionen berücksichtigen auch die Attribute, welche den Komponenten und Konnektoren zugeordnet sind und ebenso die Szenarios.

Schließlich werden auch Projektpläne als gerichtete Graphen formalisiert. Eine Abbildungsvorschrift erlaubt die Generierung eines initialen Projektplans aus einer logischen Architektur und den iterativen Abgleich zwischen dem Projektplangraphen und dem Multigraphen, der die logische Architektur beschreibt.

# Danksagung

Die vorliegende Arbeit ist am Lehrstuhl für Software- und Systems-Engineering von Professor Dr. Dr. h.c. Manfred Broy an der Technischen Universität München entstanden. Das produktive Arbeitsumfeld des Lehrstuhls und die Diskussionen mit Professor Broy und den Kolleginnen und Kollegen haben wesentlich zum Gelingen dieser Arbeit beigetragen.

Besonderer Dank gilt Professor Broy für die sehr gute Unterstützung sowie für die Freiheit bei der Themenfindung. Ebenso danke ich Professor Bernd Brügge, Ph.D., der das zweite Gutachten übernommen hat, für das Interesse der Arbeit. Beiden danke ich für die vielen hilfreichen Kommentare, Anregungen und Diskussionen zur Arbeit.

Tilman Seifert und meiner Zimmerkollegin Ulrike Hammerschall gilt besonderer Dank. Beide haben sich immer Zeit genommen, eine Idee zu diskutieren oder zu einem Text oder Vortrag Feedback zu geben. Sie haben mir viele nützliche Kommentare und Anregungen zu den diversen Fassungen und Ideen dieser Arbeit gegeben. Für die Endphase der Arbeit gilt der gleiche Dank auch Alexander Ziegler. Für die kompetente Unterstützung im Umgang mit AutoFOCUS2 danke ich Florian Hölzl.

Dem BMBF danke ich für die Finanzierung meiner Stelle im Rahmen des VSEK-Projektes. Die im Rahmen des Projektes veranstalteten Worshops haben Denkanstösse zu dieser Arbeit gegeben, ebenso die Architektenworkshops, die Prof. Dr. Andreas Rausch und ich gemeinsam veranstaltet haben.

Auch meinen Lehrstuhlkollegen Dr. Marco Kuhrmann, Florian Deissenböck, Patrick Keil, Dr. Michael Gnatz, Dr. Frank Marschall und Dr. Marcus Pizka danke ich für die sehr gute Zusammenarbeit in den Projekten und in der Lehre.

Meiner Mutter danke ich für die Tolleranz und die Unterstützung, sie musste im Verlauf der Arbeit viel auf Sohn, Schwiegertochter und Enkel verzichten. Dina, Du hast mit Deiner großartigen Unterstützung, Deiner Tolleranz und Geduld diese Arbeit erst möglich gemacht. Lasse, Dir danke ich für die Spaziergänge im Park, die mir Zeit zum Nachdenken und viele gute Ideen verschafft haben.

Ich widme diese Arbeit meinem Vater, Johann-Hinrich Beneken.

Neubiberg im August 2008,

Gerd Beneken.

# Inhaltsverzeichnis

# Kapitel 1

# Einführung

Architektur und Projektmanagement sind Mittel, um große Software-Entwicklungsprojekte beherrschbar zu machen. Die Verbindung zwischen beiden Disziplinen soll in der vorliegenden Arbeit hergestellt werden, um architekturzentriertes Projektmanagement methodisch zu unterstützten. Zusätzlich werden systematisch Architektursichten erzeugt.

Diese Einführung gibt einen Überblick über die vorliegende Arbeit: Die Bedeutung von Beschreibungen der Software- und Systemarchitektur als Kommunikationsmittel und der Zusammenhang zwischen Architektur und dem Projektmanagement werden in Abschnitt 1.1 verdeutlicht. Offene Fragestellungen und der Handlungsbedarf in diesem Bereich werden in Abschnitt 1.2 abgeleitet. Um die aufgezeigten Probleme anzugehen, wird eine Architekturtheorie auf der Grundlage der Graphentheorie vorgeschlagen. Diese wird in Abschnitt 1.3 skizziert und von vorhandenen Arbeiten aus der Literatur abgegrenzt. Die Anwendungen der Theorie, insbesondere die Erzeugung von Architektursichten und das architekturzentrische Projektmanagement werden im Abschnitt 1.4 vorgestellt. Abschnitt 1.5 enthält die Abgrenzung zu verwandten Arbeiten anderer Autoren und schließlich gibt Abschnitt 1.6 einen Überblick über die Kapitel dieser Arbeit.

## Übersicht

## 1.1 Architektur: Erfolgsfaktor für Software-Entwicklungsprojekte

### 1.1.1 Projektmanagement und Architektur hängen zusammen

Große IT-Systeme können nur arbeitsteilig in großen Teams erstellt werden. Kleine Teams würden zu viel Zeit benötigen, wie Brooks in [Bro03] vorrechnet. Die arbeitsteilige Entwicklung muss geplant und organisiert werden. Die Software- bzw. Systemarchitektur ist dafür eine Grundlage [Bur02, S.76ff]: Typischerweise wird die Erstellung einer Komponente an ein Teilteam, eine Organisationseinheit oder einen Lieferanten vergeben. Die Architektur eines IT-Systems und die Organisation des Teams, das es entwickelt, ähneln sich daher.

Die Architektur eines IT-Systems beeinflusst die Kommunikation in den Entwicklungsteams: Je klarer und stabiler die Schnittstellen zwischen den Komponenten des IT-Systems formuliert sind und je weniger die Komponenten untereinander vernetzt sind, desto weniger müssen die Entwickler direkt (oder indirekt über den Architekten) miteinander kommunizieren, um Unklarheiten zu beseitigen [HG99a, GHP99]. Dies zeigt auch eine Untersuchung[1] von Brügge und Dutoit [BD97]. Schnittstellen zwischen den Komponenten sind Kommunikationsschnittstellen zwischen den Entwicklern. Mehrdeutig oder fehlerhaft spezifizierte Schnittstellen erhöhen den Kommunikationsaufwand während der Entwicklung und der Integrationsphase. Dies zeigen auch die Beobachtungen von Küderli [Küd05].

Eine Liste mit Arbeitspaketen bildet die Grundlage für Verfahren zur Expertenschätzung. Die Aufwände zur Erstellung eines IT-Systems werden über die Schätzung der Aufwände für jedes Arbeitspaket ermittelt. Die Arbeitspakete werden in der Regel entlang der Grobarchitektur des IT-Systems definiert. Damit ist die Grobarchitektur Voraussetzung für viele Verfahren zur Expertenschätzung [Sie02b, Bur02].

Im Rahmen des Projekt-Controllings muss der Fertigstellungsgrad eines IT-Systems ermittelt werden: Diese Kennzahl kann aus den Fertigstellungsgraden der Arbeitspakete[2] berechnet werden. Diese Art der Fortschrittsermittlung kann ungenau und fehlerhaft sein, da sie das eigentliche Ergebnis der Arbeitspakete nicht berücksichtigt [WM04, S.188ff], beispielsweise erfolgreich getestete Komponenten. Eine Architektursicht, die den Fertigstellungsgrad der einzelnen Komponenten des IT-Systems darstellt, liefert ein plastischeres Bild der bis dahin tatsächlich erarbeiteten Ergebnisse.

Die Software- bzw. Systemarchitektur bildet daher die Grundlage für die Organisation und das Management eines Entwicklungsprojekts. Die Entwicklung von Komponenten wird an eigene Teams und Lieferanten vergeben. Die Schnittstellen der Komponenten beeinflussen daher die Schnittstellen zwischen den Organisationen und bilden auch die Grundlage für Verträge zu deren Zusammenarbeit. Die Abstimmung zwischen Architektur und Projektmanagement ist daher eine Grundlage für den Projekterfolg.

### 1.1.2 Zusammenarbeit zwischen Architekt(en) und Projektleiter(n)

In kleineren Projekten werden die Rollen Architekt und Projektleiter von derselben Person ausgefüllt. In großen Projekten sind beide Rollen mit verschiedenen Personen oder mit verschiedenen Teilteams besetzt. Da ein enger Zusammenhang zwischen Architektur und Projektmanagement besteht, sollten Architekt(en) und Projektleiter zusammenarbeiten[3]. Planung und Architektur müssen

[1] Brügge und Dutoit untersuchen die Kommunikation in Projekten: Ein Projekt, das mit strukturierten Methoden (SA/SD) arbeitet, wird mit einem inhaltlich ähnlichen Projekt verglichen, das mit der objektorientierten Methode OMT (Object Modeling Technique) arbeitet. Im Projekt, das die OMT verwendet, wird zwischen verschiedenen Teilteams weniger kommuniziert, insbesondere während der Implementierung und Integration. Brügge und Dutoit schließen: *Our interpretation is, that the use of OMT minimized the dependencies accross modules, and thus accross teams, therefore reducing the cost of inter-team communication.*

[2] $= \frac{VerbrauchterAufwand}{VerbrauchterAufwand+Restaufwand}$

[3] *Leider zeigt die Praxis oftmals ein anderes Bild: Projektleiter planen ohne Kenntnis von Architekturen. Dies führt im weiteren Projektverlauf zu Problemen, weil es zwei unterschiedliche Planungsstrukturen gibt: Eine gemäß der Projektplanung und eine gemäß der*

iterativ entwickelt und untereinander abgeglichen werden, damit sie zueinander passen. Erfolgt die Projektplanung auf der Grundlage erster Grobentwürfe des IT-Systems wird dieses Vorgehen auch als *architekturzentriertes Projektmanagement* bezeichnet [BD04, S.612].

### 1.1.3 Architekturgrafiken als Kommunikationsmittel

Kommunikation innerhalb des Projektteams und mit allen anderen Beteiligten ist ein wichtiger Erfolgsfaktor für ein Projekt [CHA99, S.4],[BD04, S.101ff]. Keller [Kel03, S.99] kommt in seiner Untersuchung von 13 Software-Projekten zu dem Schluss: *Architekturbeschreibungen [nehmen] eine zentrale Rolle bei der Projektplanung und Koordination ein. Sie ermöglichen eine unmissverständliche Kommunikation zwischen den beteiligten Stakeholdern über die wichtigsten Strukturen des zu realisierenden Systems*. Smolander und Präivärinta kommen nach der Untersuchung von drei Unternehmen im Telekommunikationsbereich zu einer ähnlichen Beobachtung: *Whereas only Designers emphasized architecture as basis for further design and implementation, the other stakeholders emphasized it rather as means for communication, interpretation and decision making* [SP02].

Eine grobe Architekturbeschreibung dient dem Auftraggeber als Übersicht über das zu entwickelnde System. Bestehende Probleme oder Engpässe in der Entwicklung können anhand dieser Darstellung visualisiert, verortet und kommuniziert werden. Diese Beschreibung dient dem Projektleiter als Grundlage für die Aufgabenverteilung im Team und für die Planung. Ebenso ist die Architekturbeschreibung für die Entwickler eine Möglichkeit, sich in der Organisation und der Software großer Projekte zu orientieren, wenn die zuständigen Teams für jede Komponente bekannt sind.

## 1.2 Handlungsbedarf: Architekturbeschreibung und Projektmanagement

### 1.2.1 Zusammenhang von Architektur und Projektmanagement unklar

Die Disziplinen Software- bzw. Systemarchitekturentwurf und Projektmanagement werden in der Literatur bis auf wenige Ausnahmen isoliert betrachtet. Der Zusammenhang zwischen Architektur und Planung wird in der Praxis kaum dargestellt und kaum wissenschaftlich untersucht. In der Regel wird lediglich darauf hingewiesen, dass Komponenten auch Arbeitspakete in der Planung sein können. Demzufolge findet sich kaum methodische Unterstützung für die Zusammenarbeit von Architekt und Projektleiter[4]: Planung und Architektur werden typischerweise iterativ entwickelt und sollten wie oben dargestellt, untereinander abgeglichen werden. Für diesen Informationsabgleich zwischen Architektur und Planung fehlt ein Verfahren. Eine Schnittstelle zur Projektplanung und dem Projekt-Controlling wird von keinem der bekannten Architekturbeschreibungsansätze (ADL, UML) unterstützt.

### 1.2.2 Architektursichten für das Projektmanagement fehlen

In der Literatur fehlen Vorschläge für Sichten, welche die Projektdurchführung unterstützten: Beispielsweise eine Darstellung, welches Team für welche Komponente verantwortlich ist oder die Testüberdeckung jeder Komponente als Hilfe für die Beurteilung des Qualitätszustands der Software. Wenn Vorschläge für derartige Sichten gemacht werden, sind diese vage und Beispiele fehlen vgl. z.B. Keller [Kel03, S.99ff] oder Paulish [Pau02].

*Architektur*[Sta05]

[4]Einzig Paulish macht dazu Vorschläge [Pau02]

Dem Auftraggeber ist beispielsweise häufig nicht bewusst, wie viel Aufwand bestimmte Anforderungen kosten. Dieses Wissen kann aber dabei helfen, mit dem Auftraggeber über die Reduktion oder Änderung der Anforderungen und damit der Erstellungskosten zu diskutieren. Eine Architektursicht, die Komponenten und Konnektoren zusammen mit ihren Erstellungskosten darstellt, würde diese Informationen liefern. Derartige Vorschläge sind bislang nicht publiziert worden.

Die derzeit verfügbaren ADL[5] oder die UML können nicht in ausreichendem Umfang um Zusatzinformationen etwa aus Projektplänen angereichert werden. Architektursichten, die für die Kommunikation zwischen den beteiligten Stakeholdern hilfreich wären, können daher kaum[6] mit den Mitteln dieser Sprachen erstellt werden. Ein Ansatz fehlt, der die Anreicherung von Architekturbeschreibungen mit Zusatzinformationen unterstützt.

### 1.2.3 Wenig systematischer Umgang mit Architektursichten

Konzepte für spezifische Architektursichten [IEE00] werden in der Literatur häufig pragmatisch sowie über Erfahrungen begründet, vgl. beispielsweise [Kru95, Sie04, Sta05]. Die Inhalte jeder Sicht werden dort jeweils informell und über Beispiele beschrieben. Eine nachvollziehbare Systematik fehlt, nach der Sichten eingeordnet, verglichen oder sogar erzeugt werden können[7].

## 1.3 Theorie logischer Architekturen als Ziel dieser Arbeit

Die vorliegende Arbeit erarbeitet einen Vorschlag zum architekturzentrierten Projektmanagement sowie eine mathematisch fundierten Ansatz und eine Systematik zur Erzeugung von Architektursichten. Grundlage dafür bilden Modelle, die hier als logische Architekturen bezeichnet werden. Gründzüge der Theorie und ihrer praktischen Anwendung werden im Folgenden dargestellt.

Bereits während oder sogar vor der Anforderungsanalyse in einem Projekt zur Entwicklung eines IT-Systems entstehen erste grobe Darstellungen der System- und Software-Architektur [BD04, S.593],[HNS99a]. Diese Grobentwürfe werden in der vorliegenden Arbeit *logische Architekturen* genannt.

Die logische Architektur beschreibt, wie der Funktionsumfang des IT-Systems auf logische Komponenten aufgeteilt wird und wie diese über Konnektoren Nachrichten austauschen[8]. Wie diese Bestandteile später in Software oder Hardware implementiert, wiederverwendet oder zugekauft werden, wird noch nicht festgelegt. Dies geschieht später in der technischen Architektur.

Die logische Architektur ist für die Planung, Organisation und Steuerung von Projekten die Grundlage: Für die detaillierte Spezifikation, den Feinentwurf und die Implementierung muss das IT-System so in logische Komponenten zerlegt werden, dass diese arbeitsteilig spezifiziert, entworfen und entwickelt werden können. Die Arbeitsaufteilung geschieht typischerweise entlang der Komponentengrenzen. Die Dekomposition in logische Komponenten berücksichtigt damit neben den anderen Architekturtreibern wie Qualitätsanforderungen und wirtschaftlichen Anforderungen auch Aspekte des Projektmanagements.

---

[5] Architecture Description Languages

[6] Denkbar wäre ein UML-Profil für das Projektmanagement, dieses könnte auf dem unten vorgestellten Ansatz aufbauen. Problematisch beim Einsatz der UML ist jedoch, dass Bedeutungsträger wie Farbe, Position oder Größe ihrer Beschreibungselemente nicht in der UML genutzt werden [RH06, S.243ff]. Gerade diese Bedeutungsträger können für verschiedene Stakeholder für die Akzeptanz der Architekturbeschreibungen und zu deren Verständnis wichtig sein.

[7] vgl. Abschnitt 2.3.4

[8] Ein Konnektor stellt hierbei lediglich dar, dass eine Komponente einer anderen Nachrichten schicken kann. Nachrichteninhalte oder -typen sind noch ebensowenig festgelegt, wie Protokolle. In der technischen Architektur kann ein Konnektor etwa als Warteschlange, als Paar von Schnittstellen oder als gemeinsam genutzte Datenbanktabelle verfeinert werden.

### 1.3.1 Logische Architektur als gerichteter Multigraph

Die vorliegende Arbeit schlägt eine einfache Architekturtheorie auf der Grundlage der Graphentheorie vor. Betrachtungsgegenstand sind logische Architekturen, insbesondere deren Struktur. Diese werden als gerichtete Multigraphen dargestellt:

- Komponenten werden als Knoten und Konnektoren werden als Kanten eines Multigraphen modelliert. Zwischen zwei Komponenten können mehrere Konnektoren verlaufen.
- Der hierarchische Aufbau der Komponenten aus Teilkomponenten wird durch eine zweite Kantenmenge innerhalb des Multigraphen beschrieben.
- Kommunikationsstruktur und Hierarchiedefinition werden zusammen als Konfiguration[9] bezeichnet. Die Konfiguration ist also ein Multigraph mit zwei Kantenmengen: Eine Kantenmenge stellt die Kommunikation dar und die andere den hierarchischen Aufbau. Derartige Strukturen werden in der Literatur auch als hierarchische Graphen bezeichnet.

Die Darstellung einer Architektur als hierarchischer (Multi-)Graph[10] ist weit verbreitet und in vielen Publikationen zu finden [Gar03]. Grafische Architekturbeschreibungen etwa einfache Box-And-Arrow-Diagramme[11] oder UML 2.0 Beschreibungen können als gerichtete Graphen aufgefasst und beschrieben werden.

Zwei Aspekte der Architekturtheorie gehen über die typische Darstellung einer Architektur als hierarchischer Graph hinaus:

1. Jeder Komponente und jedem Konnektor können Attribute zugewiesen werden, beispielsweise Erstellungsaufwand, Fertigstellungstermin, aktuelle Testüberdeckung oder verantwortliches Team. Diese Zusatzinformationen ermöglichen es, unter anderem eine Verbindung zwischen einer Architekturbeschreibung und Informationen aus der Projektplanung und dem Projekt-Controlling herzustellen.

   ACME [GMW97] und xADL 2.0 [DvdHT05] sind die einzigen[12] Architekturbeschreibungsansätze, die es erlauben, Architekturelementen beliebige Zusatzinformationen zuzuordnen. Die Architekturtheorie geht jedoch über die dort vorgesehenen Möglichkeiten hinaus, da diese Informationen auch bei der Generierung von Architektursichten berücksichtigt werden (siehe unten).

2. Anwendungsfälle (Nutzungsszenarios) werden explizit als Teil der Architektur modelliert. Sie werden als Teilmenge der Konnektoren darstellt. Damit dokumentieren Szenarios, welche Komponenten und Konnektoren diese implementieren. So wird eine einfache Verbindung der logischen Architektur zu den funktionalen Anforderungen hergestellt.

Die Architekturtheorie beinhaltet keine formalen Verhaltensbeschreibungen, etwa in Form endlicher Automaten. Diese können in einem Verfeinerungsschritt hin zur technischen Architektur ergänzt werden.

### 1.3.2 Projektionen zur Erzeugung von Architektursichten

Ausgehend von einer Darstellung der logischen Architektur können mit den Mitteln der Theorie konsistente Architektursichten über mathematisch definierte Projektionen erzeugt werden. Die Pro-

[9] Dieser Begriff ist bei ADL gebräuchlich [MT00] und bezeichnet die Software- bzw. Systemstruktur. In der Disziplin Konfigurationsmanagement hat der Begriff Konfiguration eine andere Bedeutung

[10] Einige Ansätze beschreiben die Architektur über einen Hierarchiebaum und einen gerichteten (Kommunikations-)Graphen, wobei die Knoten des Graphen die Blätter des Baumes sind, vgl. etwa [NL05].

[11] Jede Box wird als Knoten und jeder Pfeil als Kante aufgefasst.

[12] dem Autor bekannten

jektionen berücksichtigen auch die Attribute und die Nutzungsszenarios. Zusätzlich können Projektionen über Attributwerte beeinflusst werden. Damit werden mehrere pragmatische Vorschläge für Architektursichten, etwa die von Siedersleben et al. [Sie02a], auf eine formalere Grundlage gestellt. Die Abbildung 1.1 stellt die Idee grafisch dar.

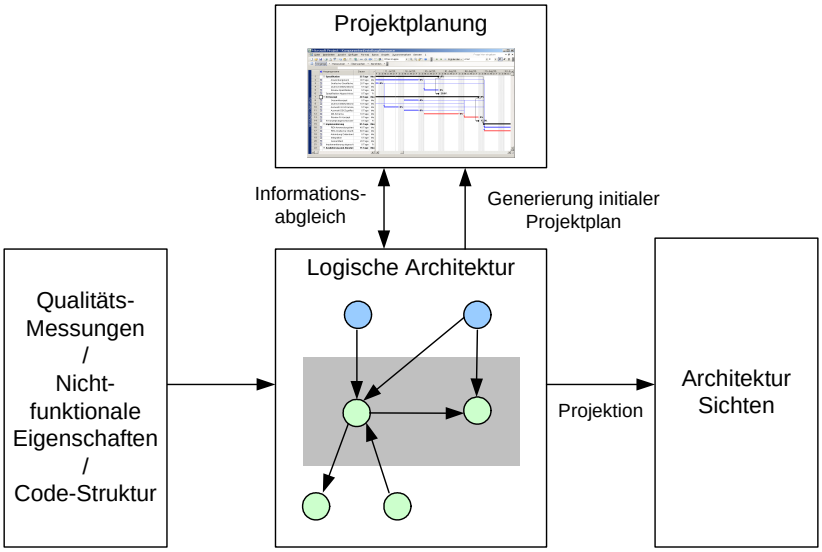


Abbildung 1.1: Logische Architektur als Kernmodell

Die Theorie nutzt zur Definition der Projektionen einfache Gleichungen (Mengenlehre), welche die Abbildung eines Multigraphen in einen anderen mathematisch beschreiben. Alternative Ansätze verwenden Graphgrammatiken [FH00, WF02] oder Modelltransformationen [OMG05]. Darüber werden in der Literatur auch Architekturrekonstruktion und -evolution beschrieben.

In Graphgrammatiken wäre die Berücksichtigung der oben dargestellten Attribute kaum möglich, denn Rechenregeln für die Attribute, etwa deren Summation, müssten mit den Mitteln der Graphgrammatik beschrieben werden. Die Erstellung von Sichten über Modelltransformation erfordert ein Metamodell für logische Architekturen [OMG05], das alle Attribute enthalten muss, die Komponenten oder Konnektoren zugeordnet werden können. Die Transformationsregeln werden mithilfe dieses Metamodells formuliert. Die vorgeschlagene Theorie betrachtet Strukturen von Architekturen und deren Veränderung über Projektionen auf möglichst einfache mathematische Weise und sie beschreibt Anforderungen an die Typen der Attribute. Dies bildet die Grundlage für eine eventuelle spätere Formulierung der Projektionen über Modelltransformationen.

Die Architekturtheorie stellt keine konkrete Syntax bereit. Sie beschreibt Strukturen mit mathematischen Mitteln. Auch Informationen zum grafischen Layout der Multigraphen sind nicht enthalten. Abbildungen auf eine konkrete grafische Syntax und eine stärkere Berücksichtigung des Layouts sind mögliche Weiterentwicklungen, wie sie etwa Ernst et al. in [ELSW06][13] im Bereich der Software-Kartographie vorschlagen.

## 1.4 Praktische Anwendung der Architekturtheorie

Mit der vorliegenden Arbeit soll die Zusammenarbeit von *Architekt* und *Projektleiter* verbessert werden. Zwei Anwendungsbereiche der Architekturtheorie werden dazu beispielhaft vorgestellt: Das

[13] Im Sinne von Ernst et al. ist die Architekturtheorie eine Vorstufe zu einem Informations-Metamodell, das den Inhalt von Architektursichten beschreibt. Ein Visualisierungs-Metamodell, das eine grafische Sprache beschreibt und eine Abbildungsvorschrift zwischen beiden, die etwa eine Komponente auf ein Rechteck abbildet, sind in darauf aufbauenden Arbeiten zu ergänzen.

sind die Erzeugung von Architektursichten und das architekturzentrierte Projektmanagement. Die vorgestellten Anwendungsbereiche sollen insgesamt große, verteilte Software-Entwicklungsprojekte besser planbar und durchführbar machen sowie deren Transparenz erhöhen.

### 1.4.1 Architektursichten - Systematik und Generierung

Die Projektionen der Architekturtheorie bilden eine logische Architektur in eine andere logische Architektur ab. Mehrere typische Architektursichten können so aus einer zentralen logischen Architektur erzeugt werden. Die Theorie liefert hierbei nur die Struktur- und die Zusatzinformationen, diese müssen noch in eine konkrete Syntax wie Box-and-Arrow-Diagrammen, UML 2 oder AutoFOCUS 2 umgesetzt werden. Nachfolgend werden zwei Typen von Architektursichten beispielhaft vorgestellt:

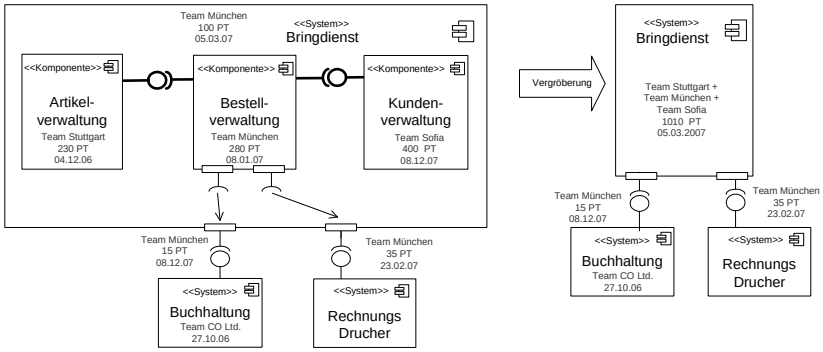


Abbildung 1.2: Architektursichten und Planungsinformationen in UML-Syntax

**Vergröberung:** Die zentrale logische Architektur (Kernmodell) beschreibt den vollständigen hierarchischen Aufbau eines IT-Systems aus Komponenten und Konnektoren. Für Übersichtsbilder sind nur Ausschnitte davon erforderlich. Ein Umgebungsdiagramm [CBB+03, S. 195f] stellt beispielsweise das System als Black-Box zusammen mit Nachbarsystemen und Nutzergruppen dar. Die Darstellung der ersten Dekompositionsebene des IT-Systems (Top Level Komponenten) ist ebenfalls gebräuchlich. Die in der Architekturtheorie definierte Vergröberungsprojektion erlaubt es, Darstellungen für jede Dekompositionsebene zu erzeugen und so Umgebungsdiagramme, Diagramme der ersten Dekompositionsebene etc. zu generieren. In Abbildung 1.2 ist eine solche Vergröberungsprojektion dargestellt, diese erzeugt aus dem Kernmodell (links) ein Umgebungsdiagramm (rechts).

**Sichten mit Zusatzinformationen:** Zusammen mit den Komponenten und Konnektoren werden Attribute visualisiert, etwa Erstellungsaufwände in einer Aufwandssicht, Fertigstellungsgrade der jeweiligen Komponenten in einer Controlling-Sicht, Testüberdeckung in einer QS-Sicht oder die Struktur der Implementierung (etwa Namen von Java-Packages und -Interfaces) in einer Code-Landkarte. Die Abbildung 1.2 zeigt beispielsweise links und rechts neben den Strukturinformationen auch Planungsdaten, wie das zuständige Team, den Aufwand in Personentagen und den Fertigstellungstermin.

Die Kommunikation in einem Entwicklungsteam kann mit derartigen Sichten verbessert werden: Eine Architektursicht stellt dar, welches Team für welche Komponente zuständig ist, wie es etwa Paulish vorschlägt [Pau02]. So werden Ansprechpartner in großen Teams schneller gefunden.

Andere Ansätze zu Architektursichten definieren ihre Sichten informell [Kru95, Sie04] oder über Metamodelle [HNS99a]. Vorschriften zur Erzeugung spezifischer Sichten (z.B. Vergröberung) werden nicht angegeben, dies leistet jedoch die vorgeschlagene Architekturtheorie.

Viele der publizierten Sichtenkonzepte verwenden Strukturinformationen und zusätzliche Beschreibungen der dargestellten Komponenten, etwa zusätzliche Datenmodelle oder Anwendungsfälle. Attribute, etwa aus der Projektplanung oder der Qualitätssicherung, werden nicht berücksichtigt. Einzig die Arbeiten von Matthes et al. im Bereich der Software-Kartographie für Landschaften aus IT-Systemen [LMW05] verwenden Zusatzinformationen in ihren Darstellungen. Im Gegensatz zu Matthes et al. werden hier jedoch Einzelsysteme betrachtet und mehrere konkrete Sichten vorgeschlagen, die für das Management entsprechender Entwicklungsprojekte hilfreich sind, außerdem werden Operatoren (etwa Summation) für die Berücksichtigung der Attribute in Projektionen angegeben.

### 1.4.2 Architekturzentriertes Projektmanagement

Auf eine logische Architektur können Informationen aus den Arbeitspaketen eines Projektplans projiziert werden. Zu jeder logischen Komponente und zu jedem Konnektor ist also bekannt, wer sie/ihn erstellt, wann sie/er integrierbar ist und wie viel Aufwand die Erstellung voraussichtlich kostet. Die Abbildung 1.2 stellt die Idee grafisch in UML-Syntax dar: Ein Bringdienstsystem besteht aus den drei logischen Komponenten *Kundenverwaltung*, *Artikelverwaltung* und *Bestellverwaltung*, diesen werden Termin, Aufwand in Personentagen und zuständiges Team zugeordnet, ebenso werden dem System als Ganzem das Integrationsteam und entsprechende Aufwände zugewiesen.

Ähnlich wie Architekturen können auch Projektpläne mit ihren Arbeitspaketstrukturen[14] sowie die dazu passenden Informationen aus dem Projekt-Controlling als gerichtete Graphen modelliert werden. Ein Vorschlag dazu wird in der vorliegenden Arbeit gemacht. Zwischen Architektur und Arbeitspaketstruktur wird über die Formalisierung als Graphen ein Zusammenhang formal beschreibbar: Eine Abbildungsvorschrift bildet etwa die Knoten (Arbeitspakete) und Kanten (Abhängigkeiten) des Planungsgraphen auf die Knoten (Komponenten) und Kanten (Kommunikation, Hierarchie) der logischen Architektur ab. Zusätzlich werden Informationen aus der Planung als Attribute auf die Architektur übertragen. Eine umgekehrte Abbildung von der Architektur auf die Planung wird ebenfalls angegeben.

Die vorliegende Arbeit stellt damit erstmals einen einfachen Zusammenhang zwischen Architektur und Planung her und macht damit die Zusammenarbeit zwischen Architekt und Projektleiter über Werkzeuge besser unterstützbar. Andere bislang publizierte Ansätze bleiben in diesem Bereich nur vage und informell, etwa Paulish [Pau02].

Anwendungsmöglichkeiten der Architekturtheorie für das Projektmanagement sind unter anderem:

1. **Architekturbasierte Plausibilisierung von Plänen**: Mit Hilfe der Architekturtheorie wird eine Architektursicht erzeugt, welche die Produktstruktur (die logische Architektur) zusammen mit Planungsinformationen darstellt, wie es in Abbildung 1.2 links gezeigt wird. Die Darstellung kann mit der Vergröberungsprojektion aus der Architekturtheorie auf das gewünschte Abstraktionsniveau vergröbert werden. Dies ist in Abbildung 1.2 rechts dargestellt.

   Diese Darstellung kann zusätzlich zu den in der Planung typischerweise verwendeten Terminplänen (Balken-Diagramme, Gantt-Charts) und Organigrammen verwendet werden. Die dargestellten Informationen wie die Kosten und Termine pro Komponente machen komplexe Planungen plastischer: Die Integrationsplanung kann so beispielsweise graphisch geschehen, da für jede Komponente der Fertigstellungstermin angegeben ist. Die Darstellung erlaubt zusätzlich eine inhaltliche Plausibilisierung von Planungsabhängigkeiten, die sich aus der Architektur begründen, etwa: Eine Vaterkomponente kann erst fertig gestellt sein, wenn alle Teile fertig gestellt sind oder ein Szenario ist erst dann testbar, wenn die implementierenden Komponenten und Konnektoren fertig gestellt sind.

2. **Iterative Entwicklung von Planung und Architektur**: Planung und logische Architektur können iterativ entwickelt werden: Aus der ersten Grobarchitektur wird ein erster Terminplan

[14] Work-Breakdown-Structure [PMI96]

generiert. Der Terminplan wird im Rahmen der Aufwandsschätzung sowie der Termin- und Einsatzmittelplanung überarbeitet. Diese Daten werden auf die erste Grobarchitektur projiziert und daraus werden verschiedene Sichten erzeugt. Mit den unten angegebenen Verfahren werden die Architektur und die Aufgabenverteilung optimiert. Die entstehenden Änderungen werden zurück auf die Planung gespiegelt usw.

Die Anwendung der Architekturtheorie erlaubt so die fortlaufende Synchronisation der Planung (z.B. in einem Projektmanagementwerkzeug) und der logischen Architektur.

3. **Optimierung der Aufgabenverteilung**: In der Software-Entwicklung ist die formelle und informelle Kommunikation ein wichtiger Erfolgsfaktor [HG99a, DRC+04]. Die Architekturtheorie unterstützt den Projektleiter und den Architekten bei der Optimierung der Aufgabenverteilung: Es kann eine Architektursicht erzeugt werden, die nur die Konnektoren zeigt, bei denen sich die Entwicklungsstandorte der Anfangs- und Endkomponente unterscheiden. Die Zahl solcher Konnektoren sollte gering sein, da jeder dieser Konnektoren die direkte oder indirekte Kommunikation zwischen den Teams notwendig macht [DRC+04]. Derartige Konnektoren müssen mit den dazu passenden Schnittstellen möglichst exakt spezifiziert und stabil sein [GHP99]. Dieses Verfahren ist ein neuer Beitrag der vorliegenden Arbeit.

4. **Architekturbasierte Reduktion des Lieferumfangs**: Nutzungsszenarios dokumentieren in der Theorie, welche Komponente und welcher Konnektor an der Erbringung eines Anwendungsfalls beteiligt sind. Sie dokumentieren also den Zusammenhang zwischen Funktionalität und Architektur.

   Mithilfe dieser Informationen sind Planspiele möglich: Wenn eine Komponente oder ein Konnektor weggelassen wird, welcher Nutzen wird dann nicht mehr erbracht, d.h. welche Anwendungsfälle können nicht mehr ausgeführt werden, und wie stark sinkt damit der Gesamtaufwand? In der logischen Architektur aus der obigen Abbildung könnte beispielsweise der Konnektor zum Nachbarsystem *Buchhaltung* weggelassen werden, damit können alle Anwendungsfälle, die diesen Konnektor verwenden, nicht mehr ausgeliefert werden, dafür sinkt der Gesamtaufwand um die Erstellungskosten dieses Konnektors. Solche Planspiele sind die Grundlage für eine Diskussion zwischen Auftragnehmer (vertreten durch Architekten und Projektleiter) und Auftraggeber zur Reduktion des Lieferumfangs. Zusätzlich ist es wichtig, die Anwendungsfälle mit geringer Priorität erst am Ende des Projektes einzuplanen, um diese ggf. weglassen zu können. Hierzu müssen alle Arbeitspakete, welche die Komponenten und Konnektoren der Anwendungsfälle umsetzen, an das Projektende verschoben werden. Auch dabei unterstützt das Verfahren. Es ist ein neuer Beitrag der vorliegenden Arbeit.

### 1.4.3 Werkzeugunterstützung

Das im Verlauf der vorliegenden Arbeit entwickelte prototypische Werkzeug AutoARCHITECT macht die Theorie direkt für praktische Problemstellungen anwendbar. Idee ist es, die Architektur in einem Modellierungswerkzeug zu erstellen und die konkrete Syntax dieses Werkzeugs zu verwenden. AutoARCHITECT reichert die Architekturbeschreibung um Informationen (aus einem Projektplan) an und führt die Projektionen durch. Die Visualisierung geschieht über die Darstellung des entstandenen gerichteten Graphen. Mit dieser Datenbasis können verschiedene Architektursichten erzeugt werden. Diese Idee wird in Abbildung 1.3 schematisch dargestellt.

## 1.5 Einordnung, verwandte Arbeiten und Abgrenzung

Die vorliegende Arbeit hat viele Berührungspunkte zu anderen verwandten Arbeiten aus der Software- und Systemarchitektur und dem Projektmanagement. Diese Ansätze werden im Folgenden dargestellt und von der vorliegenden Arbeit abgegrenzt. Die Abgrenzung geht auf die wichtigsten Eigenschaften der Architekturtheorie und ihrer Anwendungen ein, das sind: die Attributierung von Ar-

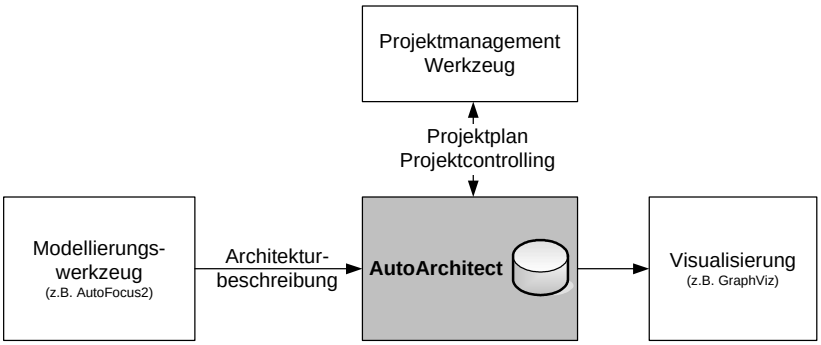


Abbildung 1.3: Werkzeug AutoARCHITECT als Ergänzung zu AutoFOCUS 2

chitekturbeschreibungen mit Zusatzinformationen, die Spezifikation von Szenarios zur Darstellung, welche Komponenten an welchem Anwendungsfall beteiligt sind, die Projektion zur Erzeugung von Architektursichten sowie die Systematik der Architektursichten und das architekturzentrierte Projektmanagement.

**Ansätze zur Architekturbeschreibung mit Zusatzinformationen** Eine Reihe von Sprachen wurde für die Beschreibung von Software-Architekturen vorgeschlagen: Die UML ist mittlerweile Industriestandard und wird in vielen Entwicklungsprojekten eingesetzt (vgl. z.B. [RH06, PBG07]). Mittlerweile verfügt die UML 2.0 über Sprachmittel, mit denen Architekturen beschrieben werden können [AGM04] und der Komponentenbegriff der UML 2.0 hat sich dem allgemeinen Komponenten-Verständnis [SGM02] angenähert. UML-Beschreibungen können über entsprechende UML-Profile mit Zusatzinformationen angereichert werden, so ist etwa ein UML-Profil denkbar, das auch Daten des Projektmanagements zeigen kann.

In der Wissenschaft haben Architekturbeschreibungssprachen (ADL) Verbreitung gefunden. Medvidovic und Taylor geben eine Übersicht über (im Jahr 2000) bekannte Architekturbeschreibungssprachen [MT00]. Eine aktuelle Übersicht findet sich etwa bei Padberg in [RH06, S.465ff]. Ziele der in den Übersichten vorgestellten ADL sind häufig die Prognose bzw. Analyse von Qualitätseigenschaften oder die spätere Generierung der Quelltexte aus der Architekturbeschreibung. Die ADL beschäftigen sich jedoch nicht mit der Erzeugung von Architektursichten oder einer Schnittstelle zum Projektmanagement.

Auf der Grundlage einer erweiterbaren ADL können einer Architekturbeschreibung grundsätzlich beliebige Informationen zugeordnet werden, also auch Informationen zur Projektplanung oder der aktuellen Qualitätszustand der Software. Zwei erweiterbare Architekturbeschreibungssprachen haben Bekanntheit erlangt, das sind ACME [GMW00] und xADL 2.0 [DvdHT01]. Beide Sprachen geben Erweiterungsmechanismen an: ACME erlaubt Erweiterbarkeit über die Zuordnung von beliebigen Attributen zu Architekturelementen. Für xADL 2.0 erfüllen Erweiterungen der zugrunde liegenden XML Schemata denselben Zweck. Die textuelle Syntax beider Sprachen ist für die Erzeugung von Sichten offenbar schwierig verwendbar.

Die vorliegende Arbeit schlägt nicht wie die UML 2.0, ACME oder xADL 2.0 eine konkrete Syntax zur Architekturbeschreibung vor, sondern fokussiert über die Darstellung als gerichteter Multigraph nur die Struktur mit Zusatzinformationen. Eine konkrete Syntax könnte eine Konkretisierung der Architekturtheorie sein.

Arbeiten zur Software-Kartographie [LMW05] machen Anleihen bei der geographischen Kartographie: Sie ordnen der Struktur der Anwendungslandschaft eines Unternehmens verschiedene Kennzahlen zu (etwa Planerische oder Wirtschaftliche Aspekte [LMW05]) und erzeugen so verschiedene Karten der Landschaft. Die vorliegende Arbeit überträgt diese Idee die Architektur von Einzelsystemen und arbeitet in Form anwendbarer Beispiele in Kapitel 7 weiter aus.

Arbeiten im Bereich der Software-Visualisierung ordnen Klassen und teilweise auch Komponenten Eigenschaften, wie Zuverlässigkeit oder andere Metriken zu. Termeer et al. visualisieren etwa Metriken als Ergänzung zu UML (1.5) Diagrammen[15] mithilfe von zusätzlichen Symbolen und der Farbgebung in ihrem Werkzeug MetricView [TLTC05]. Byelas und Telea umgeben etwa Komponenten mit ähnlichen Eigenschaften in UML-Diagrammen grafisch mit einer Hülle (Area of Interest), die Farbe der Hülle deutet auf eine Zusatzinformation hin [BT06]. Den genannten Ansätzen fehlt ein allgemeines Konzept zur Erzeugung üblicher Architektursichten, das auch Attributwerte (der Metriken) und Szenarien berücksichtigt. Daten des Projektmanagements werden ebenfalls nicht betrachtet.

**Architekturen und Nutzungsszenarios** Szenarios in Architekturen werden beispielsweise von Buhr und Casselman mit ihren Use Case Maps (UCM) beschrieben [Buh98]. Die UCM stellen die Szenarios zusammen mit der Struktur der Architektur dar. Szenarios können in einer Architektur alternativ über UML-Sequenz- oder Kommunikationsdiagramme dargestellt werden, diese visualisieren den Nachrichtenaustausch zwischen den Komponenten. UCM und Sequenzdiagramme stellen jeweils Beispiele für die Interaktion von Komponenten dar, sie sind typischerweise keine vollständigen Verhaltensbeschreibungen.

Die vorliegende Arbeit bietet ebenfalls eine Möglichkeit zur Spezifikation von Szenarios an. Szenarios dokumentieren, welche Komponenten und Konnektoren an der Durchführung eines Anwendungsfalls beteiligt sind. Szenarios sind in der Architekturtheorie als Teilmenge der Konnektoren modelliert, eine Reihenfolge oder eine Verwendungsanzahl ist nicht spezifiziert. Sie enthalten damit weniger Informationen als die Sequenzdiagramme oder UCM. Für die hier vorgestellten Verfahren und die generierten Architektursichten genügen diese Informationen:

Die Szenarios werden verwendet, um bestimmte Architektursichten zu erzeugen. Sie werden daher auch von allen Projektionen berücksichtigt. Sie dienen zusätzlich als Grundlage für verschiedene Verfahren des hier ausgeführten architekturzentrierten Projektmanagements.

**Projektionen** Die vorliegende Arbeit erzeugt Architektursichten über die Projektion einer logischen Architektur auf eine Bildarchitektur, die ebenfalls eine logische Architektur ist. Die Projektionen transformieren auch die Attribute, die den Komponenten und Konnektoren zugeordnet sind, und die Szenarios. Die Projektionen werden in Kapitel 6 über Gleichungen und unter Verwendung der Graphentheorie und Mengenlehre beschrieben, ohne dabei jedoch eine Graphgrammatik [Roz97] zu verwenden oder (Meta)Modellbasierte Transformationen [OMG05]. Um die Projektionen zu Steuern, werden einstellige Prädikate[16] verwendet, um ausgewählte Komponenten, Konnektoren und Attribute zu kennzeichnen und Äquivalenzrelationen[17], die über ihre Äquivalenzklassen Komponenten und Konnektoren aufzeigen, die zusammengefasst werden können.

Mehrere Ansätze im Bereich der Architekturevolution bzw. -rekonfiguration verwenden Graphgrammatiken zur Beschreibung von Veränderungen in einer Architektur, etwa Fahmy und Holt [FH00] oder von Wermelinger und Fiadeiro [WF02]. Über Graphgrammatiken sind grundsätzlich auch Projektionen zur Erzeugung von Architektursichten beschreibbar.

Im Gegensatz zu Graphgrammatiken können jedoch über die hier verwendeten Prädikate direkt Anfragen (Queries) in einfacher Weise auf einer logischen Architektur formuliert werden: Ein Prädikat gibt direkt an, welche Komponente, welcher Konnektor oder welches Attribut in der Bildarchitektur vorkommen soll. In der Graphgrammatik müssten Produktionsregeln die nicht mehr gewünschten Komponenten und Konnektoren entfernen, die Anfrage wäre nur indirekt formuliert und damit umständlicher. Außerdem müsste beispielsweise die Summierung von Attributwerten in der Vergröbe-

[15] Dies entspricht der Idee der Software-Kartographie.

[16] true für ausgewählte Komponenten, Konnektoren oder Attribute, false sonst

[17] Eine Äquivalenzrelation teilt die Komponenten einer Architektur in Äquivalenzklassen ein. Beispielsweise können die Software-Kategorien nach Siedersleben [SAFW99] zur Definition der Äquivalenzklassen 0-Software, A-Software, T-Software und AT-Software verwendet werden: Komponenten mit derselben Vaterkomponente und derselben Software-Kategorie sind äquivalent. Alle Komponenten einer Äquivalenzklasse werden zu einer neuen Komponente zusammengefasst.

rungsprojektion (vgl. Abbildung 1.2) mit den Mitteln der Graphgrammatik beschrieben werden.

Die Projektionen können auch als Sonderfall allgemeiner Modelltransformationen aufgefasst werden, wie sie etwa von der Model Driven Architecture benötigt werden [Fra03]. Mehrere Autoren beschreiben die Transformationsvorschrift über das Metamodell, das den Modellbeschreibungen zugrunde liegt etwa [ELSW06]. Ein Vorschlag auf der Grundlage der Meta Object Facility 1.x (MOF), in der das Metamodell der UML 1.x formuliert ist, stammt von Marschall und Braun und ist unter dem Namen BOTL bekannt [BM03]. Im Jahr 2005 ist mit QVT (Query, View, Transformationen) [OMG05] von der Object Management Group (OMG) ein Industriestandard zur Modelltransformation auf der Grundlage von Metamodellen verabschiedet worden. QVT hat MOF 2.0 als Grundlage, das Metamodell der UML 2.0 ist damit formuliert worden.

Die Verwendung von BOTL oder QVT ist möglich, um Projektionen zu beschreiben: Grundlage dazu kann ein Metamodell für logische Architekturen sein, dieses wird in Kapitel 5 umrissen. Ziel der vorliegenden Arbeit ist es, die Strukturen logischer Architekturen und die Erzeugung von Architektursichten besser zu verstehen. In diesem Zusammenhang sind Einfachheit und Freiheit bei der Gestaltung besonders wichtig, daher werden die Projektionen mit den Mitteln der Mengenlehre und der Graphentheorie formuliert. Die Verwendung der modellbasierten Transformationen ist in der Berücksichtigung von Strukturen aufwändiger, etwa bei der Berechnung der internen Konnektoren einer beliebigen Komponentenmenge[18].

Ähnlich zur hier vorgestellten Theorie sind Ansätze zur Projektion hierarchischer Graphen wie sie etwa Noack und Lewerenz skizzieren [NL05]: Auf dem Pfad von jedem Blatt der Hierarchie zur Wurzel wird ein Knoten ausgewählt. Auf diese Knoten werden dann die Konnektoren umgehängt. Ein solcher Schnitt erlaubt eine elegante Darstellung der hier vorgestellten Vergröberungsprojektion. Die vorliegende Arbeit bietet Ähnliches. Sie bildet darüber hinaus zusätzlich noch Attribute und Szenarios ab.

**Architektursichten** Die Beschreibung von Software- und Systemarchitekturen über Sichten ist etabliert. Bücher in diesem Themenfeld verwenden bekannte Sichtenkonzepte [RH06, S.35ff] oder schlagen eigene vor (vgl. z.B. [VAC+05, S.75ff], [GA03], [RW05], [Sta05, S.78ff], [BCK03] oder [HNS99a]). Begriffe und Zusammenhänge sind über den Standard ANSI/IEEE 1471-2000 [IEE00] definiert.

Typischerweise werden die Sichtenkonzepte pragmatisch gerechtfertigt. Eine Systematik, die es erlauben würde, Sichten einzuordnen oder festzustellen, ob etwas Relevantes vergessen wurde, wird selten angegeben. Beispiele sind bei Zachman und Sowa [SZ92] oder Schlosser [Sch05] zu finden. Die vorliegende Arbeit bietet zur Einordnung von Sichten in Abschnitt 7.1 eine umfassendere Systematik mit drei Dimensionen Auflösung, Sachverhalt und Architekturart an.

Die Inhalte der Sichten werden in der Literatur in der Regel informell beschrieben und mit Notationselementen und Beispielen dargestellt, vgl. z.B. [Kru95]. Hofmeister et al. geben einfache Metamodelle für ihre Sichten an [HNS99a]. Informell definierte Sichten müssen manuell erstellt und konsistent gehalten werden. Wegen der fehlenden formalen Beschreibung ist eine werkzeuggestützte Erzeugung und Sicherung der Konsistenz offenbar nur schwer möglich. Die vorliegende Arbeit bietet für logische Architekturen ein Verfahren zur Erzeugung konsistenter Sichten in den Dimensionen Sachverhalt und Auflösung der oben genannten Systematik für Sichten an. Dazu werden die oben beschriebenen Projektionen verwendet.

Lange et al. [LWC07] schlagen ein Konzept für Sichten mit UML-Modellen vor. Das Konzept berücksichtigt auch Zusatzinformationen in Diagrammen, hierfür steht eine eigene MetricView bereit. Zusammenhänge zwischen verschiedenen UML-Diagrammen werden über eine MetaView gezeigt. Dem Ansatz fehlen die hier ausgeführten Konzepte zur Sichtenerzeugung über Projektionen, die auch Attributwerte und Szenarios berücksichtigen. Sichten mit Informationen zum Projektmanagement fehlen.

[18] In diesem Zusammenhang wurden Experimente mit BOTL durchgeführt.

**Architekturzentriertes Projektmanagement** Bekannte Ansätze zur gemeinsamen Betrachtung von Architekturen und Planung stammen von Herbsleb und Grinter [HG99b] sowie von Paulish [Pau02] und Hofmeister et al. [HNS99a, S.87ff]. Diese Ansätze erwähnen zwar, dass Architektur und Projektplanung korreliert sind, geben jedoch keine konkreten Theorien, Verfahren oder Beschreibungstechniken an, welche diesen Zusammenhang explizit und für einen Architekten anwendbar machen:

Die genannten Ansätze geben keine Abbildung von Planungsdaten auf eine Architektur an, häufig wird nur empfohlen, die Architektur und Teamstruktur ähnlich zu gestalten [HG99b] - Die Literatur ist an diesen Stellen vage. Die vorliegende Arbeit macht einen Vorschlag für die systematische Abbildung zwischen Planung, Organisation und der Architektur. Diese soll Architekten und dem Projektmanagement helfen, iterativ Planung und Architektur zu entwickeln. Damit kann die vielfach geforderte optimierte Angleichung beider erreicht werden.

Gnatz betrachtet in seiner Dissertation die Generierung von Projektplänen aus Beschreibungen von Vorgehensmodellen [Gna05]. Er würdigt dabei jedoch nicht explizit die Betrachtung projektspezifischer Software- und Systemarchitekturen. Die vorliegende Arbeit konzentriert sich auf diesen Aspekt.

Einfache Diagramme, die Architektur- und Planungsinformationen visualisieren und so die Kommunikation im Projektteam verbessern, fehlen sowohl in der vorliegenden Projektmanagement-Literatur etwa [Bur02, HHMS04, WM04] als auch in der Literatur zum Thema Software- und Systemarchitekturen. Typische Visualisierungen der Planungen sind eher Netzpläne [DIN87b], Tabellen oder Balkendiagramme.

Die vorliegende Arbeit bietet einfache Architektursichten zur Visualisierung von Informationen des Projektmanagements (Termine, Kosten, Ressourcen, etc.) zusammen mit Informationen zur Architektur. Dies erleichtert Projektleiter und Architekt die Zusammenarbeit und schafft einen visuellen Zugang zu komplexen Daten des Projektmanagements. Zusätzlich bietet die vorliegende Arbeit zwei Verfahren an, welche den Zusammenhang von Architektur und Planung ausnutzen zur iterativen Optimierung der Planung und der Architektur.

## 1.6 Aufbau der Arbeit

Die Abbildung 1.4 gibt einen Überblick über die Themengebiete dieser Arbeit von der Analyse bis zur Validierung. Die Schritte werden in den folgenden Abschnitten im Detail dargestellt. Die Abbildung soll als Landkarte für die Kapitel dieses Textes dienen.

#### Teil I: Grundlagen und Begriffe

Der Teil *Grundlagen und Begriffe* hat vier Aufgaben: (1) Definition der wichtigsten Begriffe, wie *Architektur*, *Komponente*, *Projekt* oder *Qualität*, (2) Einordnung dieser Begriffe in ihren wissenschaftlichen und praktischen Kontext und (3) Beschreibung des State-Of-The-Art in den betrachteten Disziplinen, insbesondere in der Software-Architektur und dem Projektmanagement.

**Kapitel 2 Grundbegriffe am Beispiel betrieblicher Informationssysteme:** Zunächst werden grundlegende Begriffe wie IT-System, Software-System und Trägersystem sowie Projekt eingeführt. Die Begriffe Original, Modell, Produktmodell, Prozessmodell, Modellbeschreibung, Beschreibungssprachen und Sichtenkonzepte werden definiert und in Beziehung gesetzt. Die in Kapitel 4 vorgestellten Projektpläne werden als spezielle Prozessmodelle aufgefasst und die in Kapitel 3 dargestellten Software- und Systemarchitekturen werden als spezielle Produktmodelle aufgefasst. Darauf aufbauend wird die Domäne der *betrieblichen Informationssysteme* charakterisiert, um die Architekturtheorie anhand konkreter Problemstellungen und Beispiele zu erarbeiten. Die wichtigsten Architekturtreiber aus dem Bereich der wirtschaftlichen und Qua-

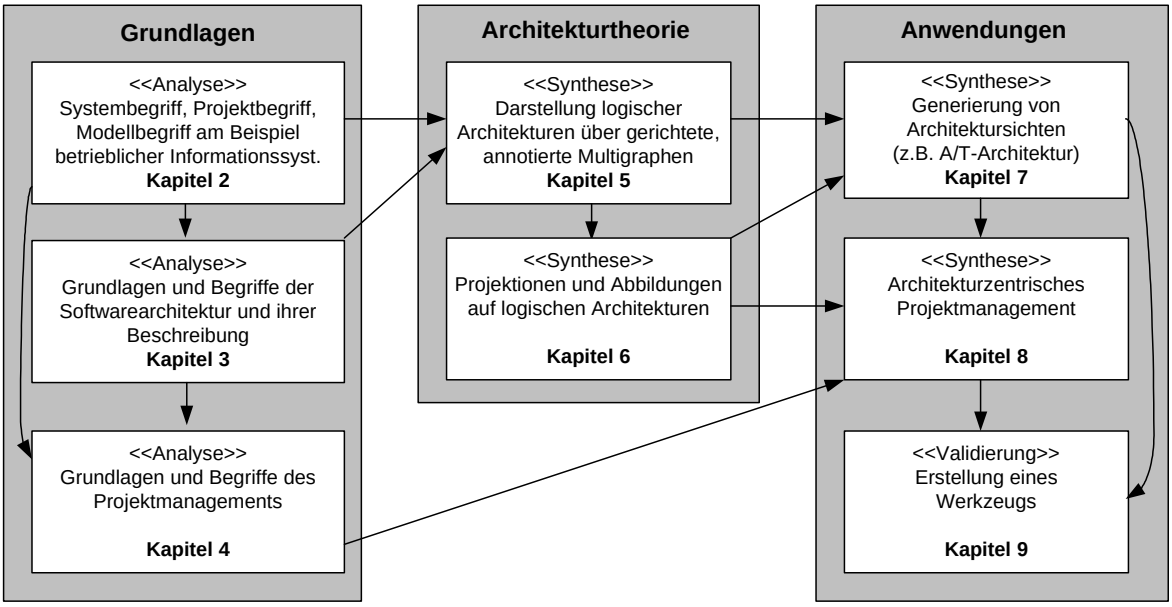


Abbildung 1.4: Übersicht über die Zusammenhänge in der Arbeit

litätsanforderungen werden betrachtet. Zusätzlich wird ein IT-System für einen Bringdienst vorgestellt, dieses wird als durchgehendes Beispiel im Verlauf des Textes verwendet.

**Kapitel 3 Architekturbeschreibung:** Eine Einführung in die Begriffe und Problemstellungen der Software- und Systemarchitekturen wird gegeben. Begriffe wie Komponente, Konnektor und Konfiguration werden definiert und hinterfragt. Das zu beschreibende Modell, die logische Architektur, wird charakterisiert und von anderen Architekturarten, wie der technischen Architektur oder der Verteilungsarchitektur abgegrenzt.

Die wichtigsten Ansätze zur Beschreibung von Architekturen werden dargestellt. Betrachtet werden unter anderem Architekturbeschreibungssprachen (ADL), die Unified Modeling Language und verschiedene Sichtenkonzepte.

**Kapitel 4 Projektmanagement:** Eine wichtige Anwendung der erarbeiteten Architekturtheorie ist das architekturzentrierte Projektmanagement. Als Grundlage für die erarbeiteten Vorschläge werden die Begriffe des Projektmanagements definiert. Zusätzlich werden Verfahren zur Projektplanung und -steuerung vorgestellt. Auf diese Verfahren wird in Kapitel 8 Bezug genommen.

#### Teil II: Architekturtheorie

Der Teil *Architekturtheorie* definiert eine auf der Graphentheorie beruhende Theorie zur Darstellung und Manipulation von logischen Architekturen. Neben den grundlegenden Beschreibungsmitteln werden auch Projektionen auf logischen Architekturen eingeführt.

**Kapitel 5 Logische Architektur als gerichteter Multigraph:** Die grundlegenden Elemente der hier vorgeschlagenen Architekturtheorie werden definiert. Für jede Komponente wird ein Knoten und für jede Kommunikationsbeziehung wird eine Kante eingeführt. Der hierarchische Aufbau der Komponenten aus ihren Teilkomponenten wird als zusätzliche Kantenmenge explizit modelliert. Zusätzlich werden Nutzungsszenarios (Anwendungsfälle) als Teilmengen der Kanten des Kommunikationsmultigraphen dargestellt.

Jeder Komponente und jedem Konnektor können Informationen zugewiesen werden, etwa:

- Ein oder mehrere Typen, beispielsweise A-Software und T-Software, wie es in [Sie04] vorgeschlagen wird.
- Qualitätseigenschaften wie Antwortzeit, Durchsatz oder Angriffswahrscheinlichkeit
- Informationen aus einem Projektplan, wie Aufwand, Termine oder verantwortliche Teams.

Die im Kapitel 6 vorgestellten Abbildungen auf logischen Architekturen berücksichtigen die zugewiesenen Informationen.

**Kapitel 6 Projektionen:** Projektionen machen einen wichtigen Mehrwert der Architekturtheorie aus. Eine logische Architektur wird in eine andere projiziert. So lassen sich aus einer zentralen, vollständigen Architekturbeschreibung (dem Kernmodell) verschiedene Sichten über Projektionen erzeugen. Es werden drei Typen von Projektionen definiert, das sind die Vergröberungsprojektion, die Auswahlprojektion und die Zusammenfassungsprojektion.

Die Vergröberungsprojektion fasst Komponenten und Konnektoren zusammen, so können etwa Sichten auf die verschiedenen Dekompositionsebenen eines IT-Systems erzeugt werden. Die Auswahlprojektion wählt Komponenten, Konnektoren und/oder Attribute nach bestimmten Kriterien aus und stellt nur diese dar, etwa alle Komponenten, die von einem bestimmten Team entwickelt werden. Die Zusammenfassungsprojektion ist eine besondere Form der Vergröberung. Sie fasst nach einem bestimmten Kriterium ausgewählte Komponenten innerhalb einer Architektur zusammen.

**Teil III: Anwendungen und praktische Umsetzung**

Die Anwendbarkeit der Architekturtheorie wird an praktischen Problemstellungen der Architekturdokumentation und des Projektmanagements demonstriert. Projektleiter und Architekt sollen bei der Erstellung und Weiterentwicklung von IT-Systemen unterstützt werden. Ein Werkzeug wird vorgestellt, das die Architekturtheorie aus Teil II und die Anwendungen unterstützt.

**Kapitel 7 Architektursichten:** Zunächst wird eine Systematik für Architektursichten im Rahmen eines Ordnungsschemas vorgeschlagen, in das Architektursichten nach Auflösung (Organisation, System oder Bausteine), dargestelltem Sachverhalt (Daten, Funktionen, Planung, ...) und Architekturart (logisch, technisch, ...) eingeordnet werden können.

Die Generierung von Architektursichten wird mit den Mitteln der Architekturtheorie dargestellt. Sichten wie Aufwands- und Planungssicht oder eine Karte der Zuständigkeiten werden ebenso erzeugt, wie klassische Architekturdarstellungen etwa das Umgebungsdiagramm oder eine Darstellung der Bausteinebene [VAC+05, S.66ff].

**Kapitel 8 Architekturzentriertes Projektmanagement:** Zwischen einem Projektplan und der logischen Architektur bestehen Gemeinsamkeiten, häufig ist eine Komponente der Architektur auch Element eines Projektplans in Form eines Arbeitspakets. Das Kapitel 8 stellt dar, wie aus der logischen Architektur ein einfacher Projektplan generiert werden kann und wie Informationen aus dem Projektplan, etwa die Aufwände für die Erstellung einer bestimmten Komponente, in die Beschreibung der logischen Architektur abgebildet werden können. Zusätzlich werden Verfahren zur architekturbasierten Reduktion des Lieferumfangs und für die Reduktion von Abstimmungsaufwänden vorgestellt. Grundlage dafür ist ein vorgestelltes Verfahren, das den iterativen Abgleich zwischen Architekturbeschreibung und Projektplan unterstützt.

**Kapitel 9 Werkzeugprototyp:** Die Anwendbarkeit der Architekturtheorie zur Erzeugung von Architektursichten und für das Projektmanagement wird über ein prototypisches Werkzeug demonstriert: Das Werkzeug AutoARCHITECT liest die Strukturinformationen aus einem Modellierungswerkzeug und kombiniert diese mit anderen Informationen, etwa einem Projektplan oder den Darstellungen nichtfunktionaler Eigenschaften. Das Werkzeug implementiert die Projektionen und Abbildungen, wie sie in Kapitel 6 eingeführt werden.

**Zusammenfassung und Ausblick:** Die wesentlichen Ergebnisse werden zusammengefasst. Vorschläge für weitere Forschungs- und Entwicklungsaktivitäten schließen das Kapitel ab.

**Anhang:** Der Anhang enthält eine Übersicht über die verwendete mathematische Notation und die verwendeten UML-Notation zur Darstellungen der Zusammenhänge zwischen Begriffen. Das Glossar stellt die in der vorliegenden Arbeit verwendeten Begriffe zusammenfassend dar.

# Teil I

# Grundlagen und Begriffe

# Kapitel 2

# Grundbegriffe am Beispiel betrieblicher Informationssysteme

Dieses Kapitel definiert die begrifflichen Grundlagen für die vorliegende Arbeit. Die Begriffe IT-System und seine Bestandteile Software-System und Trägersystem werden in Abschnitt 2.1 eingeführt. IT-Systeme werden in der Regel im Rahmen von Projekten erstellt. Ein Projektbegriff wird in Abschnitt 2.2 eingeführt und die Rollen Projektleiter und Architekt werden als Zielgruppe der vorliegenden Arbeit definiert. Die Beschreibung von System- und Software-Architekturen ist Thema der vorliegenden Arbeit. Architekturen und ihre Beschreibungen sind Modelle eines IT Systems, die im Laufe eines Projektes erstellt werden. Abschnitt 2.3 stellt die entsprechenden Begriffe Original, Modell und Beschreibung vor. Anwendungsfeld der vorliegenden Arbeit sind betriebliche Informationssysteme: Sie haben typische funktionale, qualitative und wirtschaftliche Anforderungen, die den Entwurf und die Beschreibung der System- und Software-Architektur beeinflussen. Abschnitt 2.4 diskutiert diese Anforderungen. Ein (Pizza-)Bringdienst System dient als durchgehendes Beispiel. Es wird im Überblick in Abschnitt 2.5 dargestellt.

## Übersicht

## 2.1 Systembegriff

### 2.1.1 IT-Systeme

Die Abbildung 2.1 zeigt ein Beispiel für ein IT-System und seine Nachbarsysteme. Dargestellt ist ein IT-System für einen (Pizza-)Bringdienst, bei dem Kunden im Internet oder per Telefon Speisen und Getränke bestellen können. Das Bringdienstsystem unterstützt die Geschäftsprozesse des Bringdienstes und wickelt die Bestellungen ab. Das IT-System besteht aus mehreren Rechnern, den entsprechenden Betriebssystemen, einer Netzwerkinfrastruktur sowie aus Software, welche auf diesem Trägersystem läuft und die Daten verarbeitet. Das Bringdienstsystem wird in der vorliegenden Arbeit als durchgängiges Beispiel verwendet. Details dazu finden sich in Abschnitt 2.5.

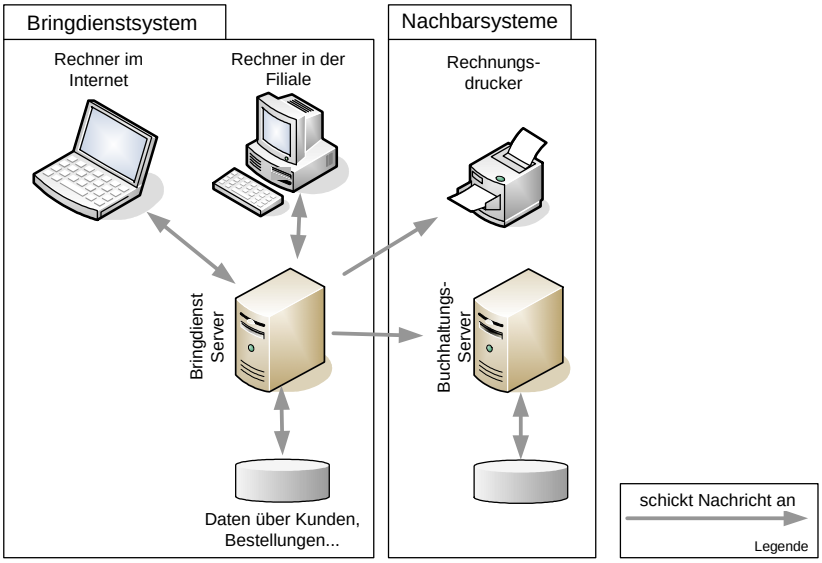


Abbildung 2.1: Informelle Beschreibung eines IT-Systems für einen Bringdienst

Ein IT-System besteht allgemein aus Hardware- und Software-Anteilen. Verallgemeinert führt dies zu folgender Definition des Begriffs *IT-System*, angelehnt an [VAC+05, S. 44]:

**Definition 2.1 (IT-System)**
Ein IT-System ist eine Einheit, die aus interagierenden Software- und Hardware-Komponenten besteht. Es existiert zur Erfüllung eines fachlichen oder technischen Ziels. Das IT-System kommuniziert zur Erreichung seines Ziels mit seiner Umgebung und muss den durch die Umgebung vorgegebenen Rahmenbedingungen Rechnung tragen.

Innerhalb des IT-Systems können zwei Bestandteile unterschieden werden: Das *Software-System* und das programmierbare *Trägersystem*, welches das Software-System ausführt. Dem Software-System kann eine formale Beschreibung seines Verhaltens zugeordnet werden, dies ist das Programm bzw. die Software.

**Definition 2.2 (Software-System)**
Ein Software-System ist Bestandteil eines IT-Systems. Es implementiert die fachlichen oder technischen Ziele des IT-Systems. Dem Software-System kann eine formale Beschreibung seines Verhaltens zugeordnet werden. Diese wird als Programm bezeichnet.

Das Programm setzt auf der von einem Trägersystem angebotenen Funktionalität auf, um zusammen mit dem Trägersystem seine Aufgaben zu erledigen. Das Trägersystem ist die Laufzeitumgebung für das Programm.

**Definition 2.3 (Trägersystem)**
Ein Trägersystem ist Bestandteil eines oder mehrerer IT-Systeme. Es ist die Laufzeitumgebung für Software-Systeme. Das Trägersystem besteht einer oder mehreren Hardware-Komponenten, die über ein Netzwerk verbunden sind und darauf aufbauenden Betriebssystemen und weiteren Laufzeitumgebungen wie virtuellen Maschinen, Datenbanksystemen oder Applikationsservern. Das Trägersystem verwaltet die Daten des IT-Systems, etwa in einem Dateisystem oder einem Datenbanksystem.

Als Beispiel für IT-Systeme werden in der vorliegenden Arbeit betriebliche Informationssysteme verwendet. Diese werden nach Stahlknecht und Hasenkamp [SH02, S. 330ff] für alle Arbeitsgebiete innerhalb eines Unternehmens verwendet, sowohl für primäre Prozesse der Wertschöpfungskette wie Beschaffung, Produktion, Vertrieb als auch für sekundäre Prozesse wie Personalabrechnung, Rechnungswesen und Controlling.

**Definition 2.4 (Betriebliches Informationssystem)**
Ein betriebliches Informationssystem ist ein IT-System. Sein Zweck ist die Unterstützung und Steuerung betrieblicher Prozesse. Das sind primäre Prozesse der Wertschöpfungskette eines Unternehmens wie Beschaffung oder Produktion und auch sekundäre Prozesse wie Personalabrechnung oder Rechnungswesen. Alternative Begriffe sind betriebliches Anwendungssystem und betriebswirtschaftliches Informationssystem.

In den vergangenen Jahren hat die Unterstützung von Geschäftsprozessen mit Software in Unternehmen deutlich zugenommen und der Integrationsgrad ist gestiegen. Viele Systeme bieten mittlerweile über das Internet direkte Nutzerschnittstellen für Kunden oder Partner an. Teilweise werden die Systeme auch über Unternehmensgrenzen hinweg integriert, beispielsweise über EDI[1] oder WebServices [Erl05]. Die Grenzen zwischen verschiedenen Systemen verschwimmen. Betriebliche Informationssysteme sind keine isolierten Inseln, sondern sie sind in eine komplexe Landschaft [Kel06] eingebunden.

### 2.1.2 Umgebung eines IT-Systems

Das Verhalten des IT-Systems hängt nicht nur von den genannten Bestandteilen ab, sondern auch von seiner Umgebung, die Nachbarsysteme und Benutzer enthält. Die hier vorgestellten IT-Systeme sind demnach *offene Systeme*, da ihr vollständiges Verhalten nur mit Informationen über die Umgebung beschrieben werden kann.

**Definition 2.5 (Umgebung eines IT-Systems)**
Zur Umgebung eines IT-Systems zählen andere IT-Systeme (Nachbarsysteme) und Benutzer, die jeweils mit dem IT-System direkt kommunizieren. Ein IT-System und seine Nachbarsysteme können sich ein Trägersystem teilen.

Die Kommunikation zwischen IT-System und seiner Umgebung findet an der *Systemgrenze* statt, diese wird innerhalb des Spezifikations- bzw. Entwurfsprozesses festgelegt. Das IT-System tauscht an seiner Systemgrenze Nachrichten mit seiner Umgebung aus.

Die Abbildung 2.2 stellt ein betriebliches Informationssystem in seiner Umgebung dar. Zur Umgebung gehören die Benutzer des Systems, welche ihre tägliche Arbeit damit erledigen und die Betriebsführung, welche für die Verfügbarkeit des Systems verantwortlich ist. Nachbarsysteme greifen auf das betriebliche Informationssystem zu.

**Benutzer (Sachbearbeiter, Kunden, Geschäftspartner)** Benutzer eines betrieblichen Informationssystems sind beispielsweise Sachbearbeiter oder Führungskräfte eines Unternehmens aber auch Kunden und Geschäftspartner. Sie verwenden ein System über seine textuelle oder grafische Oberfläche und verrichten damit ihre tägliche Arbeit.

[1] Electronic Data Interchange

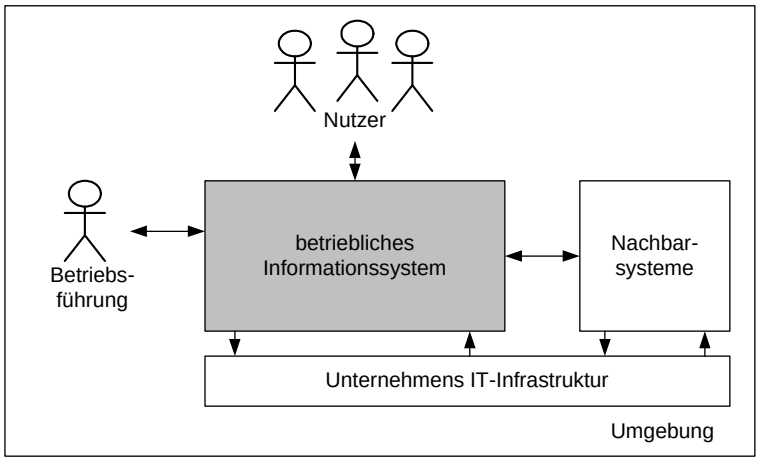


Abbildung 2.2: Ein betriebliches Informationssystem in seiner Umgebung

**Betriebsführung** Die Betriebsführung (Systemadministration) stellt eine besondere Benutzergruppe dar. Sie ist in der Regel zentral organisiert, etwa in einem Rechenzentrum. Ihre Aufgabe besteht darin, die Verfügbarkeit, Sicherheit und Performance des Systems für die verteilten Benutzer sicherzustellen [Bri00, S. 107], darauf hat sie wesentlichen Einfluss. Sie konfiguriert das System abhängig von aktuellen Erfordernissen. Während die Benutzer das System zur Abwicklung ihres täglichen Geschäfts verwenden, stellt die Betriebsführung in regelmäßigen Intervallen oder problemgetrieben sicher, dass die Benutzer arbeiten können.

**Nachbarsysteme und IT-Infrastruktur** Ein betriebliches Informationssystem kann den Funktionsumfang eines Nachbarsystems nutzen, um seinen eigenen Funktionsumfang bereitzustellen. Umgekehrt kann auch ein Nachbarsystem den Funktionsumfang eines Systems nutzen. Nachbarsysteme kommunizieren nicht immer direkt über die Fachlogik miteinander. Die Integration über einen gemeinsamen Datenbestand oder eine gemeinsame Oberfläche (Portal) ist ebenfalls gebräuchlich [Kel02].

Zentrale Dienste eines Unternehmens gehören zum Trägersystem eines betrieblichen Informationssystems und sind Bestandteil der technischen Infrastruktur eines Unternehmens, etwa ein Berechtigungssystem, ein Druckdienst oder ein Workflow-System.

### 2.1.3 Zusammenfassung der Begriffe

Die Abbildung 2.3 zeigt die hier definierten Begriffe im Zusammenhang: Ein betriebliches Informationssystem ist ein IT-System. Ein IT-System besteht aus einem Software-System und einem Trägersystem. Das Trägersystem führt das Software-System aus. Das Trägersystem umfasst die komplette Laufzeitumgebung, diese enthält Hardware-Komponenten, Netzwerke, Betriebssysteme, Datenbanksysteme und andere Laufzeitumgebungen.

Die Aufteilung in Software-System und Trägersystem entspricht dem typischen Verständnis betrieblicher Informationssysteme. Das Software-System wird im Rahmen von Projekten entwickelt und das Trägersystem steht beim Bau eines neuen Systems in der Regel fest: Die Anschaffung neuer Arbeitsplatzrechner für Sachbearbeiter, die Änderung der vorhandenen Netzwerktopologie sowie die Anschaffung etwa eines neuen zentralen Host-Rechners ist in der Regel für neu erstellte betriebliche Informationssysteme zu teuer. Betriebssysteme werden häufig von dem zentralen IT-Betrieb festgelegt, um die IT-Landschaft zentral betreiben zu können und diese nicht zu heterogen werden zu lassen. Auch weitere Elemente des hier definierten Trägersystems sind aus demselben Grund über unternehmensweite Vorgaben festgelegt, etwa die zu verwendenden Virtuellen Maschinen (Java Virtual Machine oder Common Language Runtime) sowie die Applikationsserver und Datenbankser-

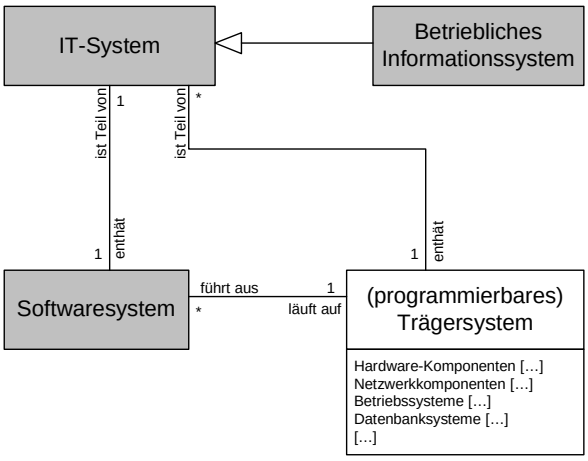


Abbildung 2.3: Begriffe zu betrieblichen Informationssystemen

ver.

Innerhalb eines typischen Projekts, das ein betriebliches Informationssystem entwickelt, werden keine neue Hardware, kein Betriebsystem und auch kein neuer Datenbankserver entwickelt. Aus den oben genannten Gründen wird in solchen Projekten selten ein Element des Trägersystems ausgetauscht. Daher werden die vorhandenen Hardware- und Infrastrukturkomponenten als Trägersystem zusammengefasst auf dessen Grundlage das Software-System erstellt wird.

## 2.2 Projektbegriff

### 2.2.1 Projekte

Betriebliche Informationssysteme werden in der Regel im Rahmen von einem oder mehreren Projekten individuell entwickelt und gepflegt [WM04, Sie02b]. In diesem Fall wird von Individual-Software gesprochen. Von diesen Systemen wird ein betriebswirtschaftlicher Nutzen erwartet. Der erwartete Nutzen bestimmt den zu implementierenden Leistungsumfang (die funktionalen Anforderungen). Der Nutzen sollte die Kosten für die Erstellung und Pflege des Systems übersteigen, sonst wäre die Entwicklung unwirtschaftlich. Das Budget, das für ein Entwicklungsprojekt zur Verfügung steht, ist damit begrenzt. Der Nutzen des Systems sollte zu einem planbaren Zeitpunkt verfügbar sein: Ein IT-System muss beispielsweise eine Gesetzesänderung termingerecht unterstützen oder neue Produkte und Dienstleistungen des Unternehmens sind von dem neuen System abhängig und sollten so früh wie möglich verfügbar sein, um sich einen Wettbewerbsvorteil gegenüber Konkurrenten mit ähnlichen Produkten und Dienstleistungen zu sichern. Der Liefertermin für ein neues oder geändertes Software-System steht damit in der Regel zu Beginn des Projektes bereits fest und eine Verschiebung dieses Termins ist teuer (Konventionalstrafen, Wettbewerbsnachteile).

Ein Projekt ist damit gekennzeichnet durch die drei Größen geforderte Leistung (diese liefert den Nutzen), beanspruchte Einsatzmittel (Budget, Mitarbeiter, Sachmittel) und benötigte Zeit [Bur02, S.23]. Im Folgenden wird eine konkretisierte Form der Definition des Projektbegriffs nach DIN 69901 [DIN87c] verwendet, welche die erwähnten Aspekte zusammenfasst:

**Definition 2.6 (Projekt)**
Ein Projekt ist ein Vorhaben, das im Wesentlichen durch die Einmaligkeit seiner Bedingungen in ihrer Gesamtheit gekennzeichnet ist. Ein Projekt ist definiert durch die Zielvorgabe (die geforderte Leistung) und die zeitlichen, finanziellen und personellen Rahmenbedingungen (nach DIN 69901 [DIN87c]).

Zeit, Einsatzmittel (Budget) und Leistung bilden das Spannungsfeld in dem sich ein Projekt bewegt, in dem es gesteuert werden muss. Dabei hängen die Parameter Zeit, Einsatzmittel und Leistung von einander ab, wenn etwa der Leistungsumfang über einen Änderungswunsch erhöht wird, kann dies eine Verschiebung des Fertigstellungstermins oder eine Aufstockung des Personals erfordern.

### 2.2.2 Auftraggeber und Auftragnehmer

In Entwicklungsprojekten für Individual-Software werden zwei grundsätzliche Parteien unterschieden: *Auftraggeber* und *Auftragnehmer*.

Der Auftraggeber definiert das Ziel des Projektes und bezahlt seine eigenen Leistungen und die Leistungen des Auftragnehmers. Er übernimmt die Rolle des Kunden in einer Vertragssituation. Häufig setzt der Auftraggeber den Termin für die zu erbringende Leistung fest. Der Auftraggeber erwartet vom Projekt einen (wirtschaftlichen) Nutzen. Der erwartete Nutzen ist der Grund, warum das Projekt durchgeführt wird.

Auf der Seite des Auftraggebers sitzen typischerweise Fachabteilungen, mit wenig Software Engineering Hintergrund, sondern mit Fachkenntnissen beispielsweise in der Bank- oder Versicherungswirtschaft oder in der Tourismusbranche. Software-Entwicklungsprojekte für betriebliche Informationssysteme sind daher in der Regel interdisziplinär.

Der Auftragnehmer wird beauftragt, das Projektziel zu erreichen, gemäß den Vorgaben des Auftraggebers. Er hat die Rolle des Lieferanten in einer Vertragssituation. Der Auftragnehmer übernimmt abhängig von der Art und Weise, wie die Zusammenarbeit vereinbart ist[2], beispielsweise Spezifikation, Entwurf, Implementierung und Modultest des Systems.

Die vorliegende Arbeit unterstellt eine Aufgabenverteilung, wie sie im V-Modell XT [V-M06] definiert ist: Die Anforderungen werden auf der Seite des Auftraggebers in einem Lastenheft definiert, die Spezifikations-, Entwurfs-, Implementierungs- sowie Test- und Integrationsarbeit findet auf der Seite des Auftragnehmers statt. Die Abnahme des Projektergebnisses geschieht wiederum auf der Seite des Auftraggebers. Projektgegenstand ist dabei die individuelle Entwicklung eines betrieblichen Informationssystems.

### 2.2.3 Projektleiter und Architekt

In Entwicklungsprojekten müssen auf Auftraggeber- und Auftragnehmerseite jeweils Rollen besetzt werden, welche die Verantwortungsbereiche und Aufgaben innerhalb des Projekts festlegen. Projektleiter und Architekt (Chef-Designer) sind Rollen, die in vielen Vorgehensmodellen definiert werden [V-M06, IBM04, Sie02b]. Auf der Seite des Auftragnehmers gibt es in der Regel beide Rollen [Sie02b]. Auf der Seite des Auftraggebers gibt es in der Regel ebenfalls einen Projektleiter. Beide Rollen sind Zielgruppe der vorliegenden Arbeit, ihre Zusammenarbeit soll unterstützt werden.

**Definition 2.7 (Projektleiter)**
Projektleiter ist eine Rolle in einem Software-Entwicklungsprojekt. Aufgabe des Projektleiters ist die Planung, die Organisation sowie die operative Kontrolle und Steuerung eines Projekts. Er stimmt die Planung mit allen Projektbeteiligten ab. Der Projektleiter ist dafür verantwortlich, dass die Ergebnisse des Projekts zum geplanten Termin, innerhalb des geplanten Budgets und in angemessener Qualität vorliegen.

In der Regel berichtet der Projektleiter auf der Seite des Auftragnehmers regelmäßig dem Auftraggeber über den Projektfortschritt. Der Bericht beschreibt den Fertigstellungsgrad von Ergebnissen, das

[2] Das V-Modell XT definiert beispielsweise über seine Entscheidungspunkte eine Schnittstelle zwischen Auftragnehmer und Auftraggeber. Die Entscheidungspunkte definieren, welche fertig gestellten Produkte (Dokumente, Hardware, Software) jeweils von welcher Partei (zu einem im Projekt festgesetzten Termin) zu erbringen sind. Auf dieser Grundlage wird jeweils entschieden, ob das Projekt fortgesetzt wird [V-M06]

Resultat von Qualitätsprüfungen und den bis zum Berichtszeitpunkt verbrauchten Aufwand. Absehbare Änderungen von Budget, (Fertigstellungs-)Terminen und Leistungsumfang werden kommuniziert.

**Definition 2.8 (Architekt (Chefdesigner))**
Architekt ist eine Rolle in einem Software-Entwicklungsprojekt. Aufgabe des Architekten ist der Entwurf der System- und der Software-Architektur. Er stimmt die Architektur mit allen Projektbeteiligten ab. Der Architekt ist verantwortlich dafür, dass das fertige IT-System alle funktionalen, wirtschaftlichen und Qualitätsanforderungen erfüllt. Der Architekt wirkt an der Projektplanung, -kontrolle und -steuerung mit.

In kleinen Projekten fallen beide Rollen in einer Person zusammen. In größeren Projekten sind diese Rollen auf zwei Personen aufgeteilt. Sehr große Projekte werden typischerweise in Teilprojekte aufgeteilt, in diesen Teilprojekten werden wiederum die Rollen des Teilprojektleiters und des Teilarchitekten besetzt.

## 2.3 Modellbegriff

### 2.3.1 IT-Systeme und Projekte als Originale

*Modelle sind stets Modelle von etwas, nämlich Abbildungen, Repräsentationen natürlicher oder künstlicher Originale, die selbst wieder Modelle sein können* [Sta73, S. 131]. Damit von Modellen gesprochen werden kann, muss zunächst also das Original festgelegt werden. Originale können Produkte sein, wie IT-Systeme oder deren Beschreibungen, oder sie können Prozesse sein, wie etwa ein Software-Entwicklungsprojekt. Betrachtet werden zwei Arten von Originalen für Produktmodelle:

1. Eigenschaften wie Antwortzeiten, Durchsatz und auch Schutz gegen unbefugten Zugriff hängen bei betrieblichen Informationssystemen unter anderem von Hardware, Netzwerkeigenschaften, Client-Zahl sowie dem Volumen und der Komplexität der Anwendungsdaten ab. Daher kann nur das installierte, laufende Software-System zusammen mit dem Trägersystem und den Anwendungsdaten das in Modellen mittelbar oder unmittelbar abgebildete *Original* sein.
2. Während der Erstellung eines betrieblichen Informationssystems entsteht eine Reihe von Modellen, beispielsweise der Quelltext. Diese Modelle werden über zusätzliche Modelle beschrieben, beispielsweise ein UML-Paketdiagramm, das die Struktur des Quelltextes beschreibt. Damit werden Modelle des IT-Systems zu Originalen für andere Modelle.

Die Festlegung des jeweiligen Originals ist für das Verständnis von Produktmodellen wichtig: Beispielsweise ist die mit einem Paketdiagramm beschriebene Quelltextstruktur am System zur Laufzeit nicht mehr feststellbar und dafür ohne Bedeutung. Demnach kann das IT-System nicht das Original für das Paketdiagramm sein.

Für Prozessmodelle sind Software-Entwicklungsprojekte die Originale. Beispiele für Prozessmodelle sind allgemeine Vorgehensmodelle, wie etwa der Rational Unified Process [IBM04] und Projektpläne für ein konkretes Projekt. Dabei kann ein Vorgehensmodell seinerseits Modell für einen Projektplan sein [Gna05].

### 2.3.2 Produkt- und Prozessmodelle

Vor und bei der Erstellung und der Dokumentation von IT-Systemen werden *Modelle* des IT-Systems und des Erstellungsprozesses (des Projektes) entwickelt. Modelle können unterschiedliche Beschreibungen haben. Die in betrieblichen Informationssystemen gebräuchlichste Form ist das Dokument

[Den91]. Es enthält natürlich sprachliche Beschreibungen, Tabellen, Screenshots und einfache (Box-and-Arrow oder UML) Diagramme. Quelltexte in einer Programmiersprache, in Bibliotheken zusammengefasste, übersetzte Quelltexte oder Testfälle sind ebenso Beschreibungen des IT-Systems. Ein Projektplan ist eine mögliche Beschreibung eines Projekts. Ein Projektplan wird typischerweise als Dokument abgefasst [V-M06], darin werden Terminpläne beispielsweise als (Gantt-) Diagramm dargestellt.

Normalerweise werden nur bestimmte Eigenschaften eines Originals in einem Modell berücksichtigt, nicht relevante Eigenschaften werden wegabstrahiert. Stachoviak bezeichnet dies als Verkürzungsmerkmal [Sta73, S. 132f]: *Modelle erfassen im Allgemeinen nicht alle Attribute des durch sie repräsentierten Originals, sondern nur solche, die den jeweiligen Modellerschaffern und/oder Modellbenutzern relevant erscheinen.*

Modelle sind auf Modellerschaffer bzw. Modellbenutzer (auf *Stakeholder*) ausgerichtet und erfüllen für diese einen bestimmten Zweck. Stachowiak [Sta73, S. 132f] bezeichnet dies als *pragmatisches Merkmal*: *Modelle sind ihren Originalen nicht per se eindeutig zugeordnet. Sie erfüllen ihre Ersetzungsfunktion a) für bestimmte – erkennende und/oder handelnde, modellbenutzende – Subjekte, b) innerhalb bestimmter Zeitintervalle und c) unter Einschränkung auf bestimmte gedankliche oder tatsächliche Operationen.*

In der Software Engineering Literatur gibt es ähnliche Modellbegriffe: Jacobson et al. [JRB99, S. 22] fordern beide Eigenschaften nach Stachowiak für ein Modell: *A model is an abstraction of a system, specifying the modeled system from a certain viewpoint and at a certain level of abstraction. [...]*

Ein Modell beschreibt bestimmte Eigenschaften des Originals, etwa dessen Struktur oder Teile seines Verhaltens. Für diese Eigenschaften muss eine eindeutige Abbildung zwischen Modell und Original existieren. Damit können mit dem Modell Aussagen über das Original gewonnen werden. Mit entsprechenden Abbildungen können ggf. Teile des Originals aus dem Modell erzeugt werden.

Da das Modell eine Abstraktion des Originals ist, beschreibt es genau genommen eine unendliche Menge von denkbaren Originalen, welche die beschriebenen Eigenschaften erfüllen. Alle Eigenschaften, über die das Modell keine Aussagen macht, können als beliebig angenommen werden.

**Definition 2.9 (Modell)**
Ein Modell ist eine Beschreibung[3] von Eigenschaften eines Originals. Ein betriebliches Informationssystem oder ein Projekt sind Beispiele für Originale. Das Modell ist eine Abstraktion des Originals: Es stellt einen bestimmten Ausschnitt der Eigenschaften des Originals dar, dieser erfüllt für eine Menge von natürlichen Benutzern und/oder IT-Systemen einen definierten Zweck. Weiterhin muss eine Abbildung zwischen Modell und Original existieren, welche das Modell auf feststellbare Eigenschaften des Originals abbildet. Das Modell kann auch ein gedachtes (mentales) Modell sein.

Die Aussage: *Das System verwaltet Name und Anschrift von Kunden* ist am laufenden System verifizierbar, indem die Dialoge zum Verwalten des Namens und der Anschrift des Kunden aufgerufen werden. Die Aussage *'Das System ist schnell'* ist dagegen nicht verifizierbar, da der Begriff *schnell* nicht genauer bestimmt ist. Ein Modell muss daher aus Aussagen bestehen, die am Original überprüft werden können.

**Definition 2.10 (Produktmodell)**
Ein Produktmodell ist ein Modell, dessen Original ein Produkt oder das Modell eines Produktes ist. Ein IT-System ist ein Beispiel für ein Produkt; Seine Software-Architektur ist ein Beispiel für ein Produktmodell.

**Definition 2.11 (Prozessmodell)**
Ein Prozessmodell ist ein Modell, dessen Original ein Prozess ist. Beispiel für Prozesse sind Software-Entwicklungsprojekte oder Geschäftsprozesse.

[3] Die Form der Beschreibung ist hier nicht definiert, die Aussage kann beispielsweise ein Idee des Fachbereichs, ein Quadrat auf einem Blatt Papier oder mehrere Sätze in einem Dokument sein.

### 2.3.3 Beschreibung von Modellen

Ein Datenmodell kann in verschiedenen Notationen dargestellt werden, etwa mit einem Klassendiagramm. Dieselben Aussagen lassen sich jedoch ebenso als Text mit Tabellen, als formaler Text, etwa im XMI-Format, oder auch in einer anderen grafischen Notation, etwa ER-Diagrammen [Che76], darstellen. Daher wird der eigentliche Sachverhalt (z.B. das Datenmodell) von seiner Beschreibungen (z.B. Diagramme, Tabellen, Texte) unterschieden.

Ein Modell wird in einer oder mehreren Sprachen durch eine oder mehrere Beschreibungen dargestellt. Die konkrete Syntax der Sprachen kann grafisch oder textuell sein. Dasselbe Modell kann mehreren Sprachen mit unterschiedlicher Syntax formuliert werden. Dies wurde anhand des Klassendiagramms illustriert. Mehrere Autoren treffen die Unterscheidung zwischen Modell und Beschreibung nicht (vgl. z.B. [IEE00]), das wird auch an Begriffen wie *UML-Modell* deutlich. Ein UML-Modell ist eigentlich nur die Beschreibung eines Modells unter Verwendung der UML.

**Definition 2.12 (Beschreibung eines Modells)**
Die Beschreibung eines Modells ist die Formulierung des Modells oder eines Ausschnitts in einer oder in mehreren Sprachen. Die Beschreibung kann beispielsweise ein Diagramm, ein Text, eine mathematische Formel oder einige Zeilen Quelltext enthalten. Auch bei der physischen Speicherung der Beschreibung in einer oder mehreren Dateien wird hier von der Beschreibung eines Modells gesprochen.

Eine Beschreibung kann in verschiedene Beschreibungselemente unterteilt werden. Die Gestaltung der Beschreibungselemente liegt dabei in Ermessen des Autors der Beschreibung.

**Definition 2.13 (Beschreibungselement)**
Ein Beschreibungselement ist Teil einer Beschreibung. Es ist in nur einer Sprache abgefasst. Eine Textpassage, ein UML-Diagramm oder eine Java-Klasse sind Beispiele für Beschreibungselemente.

#### Viele Beschreibungen und ein zentrales Modell (im Repository)?

Im Laufe eines Projektes entstehen mehrere Beschreibungen von Eigenschaften eines IT-Systems bzw. des Projekts selbst. Gründe dafür sind unter anderem, dass unterschiedliche Sachverhalte dargestellt werden müssen, etwa Anforderungen, Informationen zur Verteilung und Quelltexte. Außerdem arbeiten verschiedene Experten mit ihren spezifischen Modellen, Beschreibungstechniken und Werkzeugen mit. Im Entwicklungsprojekt entstehen also Beschreibungen in verschiedenen Spezialsprachen (UML, SQL, Java, Netzpläne, etc.). Je zwei Beschreibungen können inkonsistent sein, wenn sie die gleichen Eigenschaften eines IT-Systems oder eines Projektes widersprüchlich beschreiben, d.h. es ist kein Original denkbar, das beiden Beschreibungen genügt. Um die Konsistenz der Beschreibungen sicherzustellen, gibt es mehrere Möglichkeiten, unter anderem sind das:

1. Die Beschreibungen werden paarweise abgeglichen und konsistent gehalten, dazu muss eine Abbildung zwischen den Elementen von je zwei Beschreibungen definiert sein (vgl. z.B. [FMP99]). Ein gemeinsames Modell aller Beschreibungen existiert höchstens gedacht.

2. Alle Beschreibungen werden als Sichten aus einer gemeinsamen, konsistenten Beschreibung eines zentralen Modells erzeugt. Ein solches Modell wird hier auch *Kernmodell* genannt. Änderungen an den Beschreibungen werden auf das Kernmodell abgebildet. Die Abbildung zwischen dem Kernmodell und den Beschreibungen muss also bidirektional sein. Das Kernmodell kann in Form eines zentralen Repositories beschrieben und gespeichert werden (vgl. z.B. [Her02]).

3. Alle Beschreibungen können auf eine gemeinsame Beschreibung abgebildet werden, etwa die Implementierung. Gelingt die Abbildung nicht, sind die Beschreibungen inkonsistent.

Ein zentrales Produktmodell wird im Folgenden als *Kernmodell* bezeichnet. Dieser Ansatz wird weiter verfolgt. Damit das Kernmodell als gemeinsame Grundlage insbesondere für Verhaltensbeschreibungen des IT-Systems dienen kann, muss seine Semantik exakt definiert sein, beispielsweise mit mathematischen Mitteln. Für das Kernmodell ist ein *Systemmodell*[4] [RKB95] erforderlich, welches die Grundelemente des Kernmodells und deren Eigenschaften festlegt. Ein Vorschlag für ein Systemmodell, das auf betriebliche Informationssysteme ausgerichtet ist, findet sich beispielsweise bei Schwerin [Sch04].

### 2.3.4 Sichten und Blickwinkel

Projektbeteiligte haben Interessen, die sie mit einem Modell bzw. einer Beschreibung verfolgen. Das Projektmanagement ist beispielsweise lediglich an einem groben Überblick über das IT-System interessiert, während die Betriebsführung die Liste der betroffenen Rechner, Betriebssysteme und Netzwerke fordert. Für Entwickler ist wiederum die Strukturierung des Software-Systems in Bibliotheken von Interesse. Um die Anforderungen unterschiedlicher Projektbeteiligter zu erfüllen, sind unterschiedliche Beschreibungen notwendig, diese werden auch als Sichten bezeichnet. Diese Idee wird in den Ingenieurwissenschaften und der Gebäudearchitektur seit vielen Jahrzehnten erfolgreich betrieben. Die Architektur eines Einfamilienhauses wird beispielsweise dokumentiert über

- Vorderansicht, Seitenansicht und Draufsicht,
- Raumplan für jedes Stockwerk und
- Ansichten für Spezialisten z.B. für Elektriker mit den zu verlegenden Stromkabeln und Steckdosen, für Installateure mit den zu verlegenden Wasserleitungen, Stücklisten und Arbeitspakete für den Bauherren, mathematische Modelle für den Statiker usw.

Die Idee der Darstellung von Sichten wurde in die Informatik von mehreren Autoren übertragen: Einer der bekanntesten allgemeinen Vorschläge stammt von Zachman [Zac87]. Die aktuelle Fassung [SZ92] definiert sechs Gruppen von Stakeholdern, nämlich Planner, Owner, Designer, Builder, Subcontractor sowie User. Für jede dieser Gruppen werden in einem zweidimensionalen Schema jeweils 6 Sachverhalte definiert, die mit den Fragen What (Data), How (Function), Where (Network), Who (People), When (Time) und Why (Motivation) überschrieben sind. Dies ergibt insgesamt 36 mögliche Sichten, mit denen die verschiedenen Aspekte eines IT-Systems für die unterschiedlichen Nutzergruppen aufbereitet werden. Im deutschen Sprachraum ist mit dem ARIS[5]-Haus eine ähnliche Systematik verbreitet [Sch97].

Für Software-Architekturen definiert der Standard IEEE 1471 [IEE00] einen Begriffsapparat (Konzeptuelles Modell) für Sichten, dort wird zwischen Sicht (View) und Blickwinkel (Standpunkt [RH06, S.36], Viewpoint) unterschieden. Eine Sicht ist dabei eine konkrete Ausprägung eines Blickwinkels. Sicht und Blickwinkel verhalten sich wie Instanz zu Klasse in der Objektorientierung:

> view: A representation of a whole system from the perspective of a related set of concerns.
> viewpoint: A specification of the conventions for constructing and using a view. A pattern or template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis [IEE00, S. 2f].

Diese Begriffe werden auf alle Beschreibungen von Modellen ausgedehnt. Wir definieren analog:

**Definition 2.14 (Sicht, View)**
Eine Sicht (View) ist Teil einer Beschreibung eines Modells. Sichten strukturieren Beschreibungen.

[4] Abstraktes mathematisches Modell informationsverarbeitender Systeme. Mit *Systemmodell* im hier verwendeten Sinn, ist nicht die Modellierung eines konkreten IT-Systems gemeint.

[5] Architektur integrierter Informationssysteme

Eine Sicht bezieht sich in der Regel auf einen Teil eines größeren Modells und zeigt einen zweckgebundenen Ausschnitt. Der Zweck wird durch die Interessen der Ersteller und Nutzer der Sicht bestimmt.

**Definition 2.15 (Blickwinkel, Viewpoint)**
Ein Blickwinkel legt Verfahren zur Modellierung und Sprachen zur Beschreibung von Modellen fest. Darüber hinaus verkörpert er eine Menge verwandter Interessen von Erstellern und Nutzern der Modelle. Dies grenzt auch die Inhalte der Beschreibung ein, etwa auf Verteilungsaspekte für die Betriebsführung oder die Strukturierung der Quelltexte für die Entwickler. Eine Sicht heißt *konform* zu einem Blickwinkel, wenn sie die Beschreibungssprachen verwendet, über die Modellierungsverfahren entstanden ist und denselben Zwecken dient.

Weitere bekannte Vorschläge für Blickwinkel stammen aus dem Bereich der Software-Architektur, etwa die 4+1 Blickwinkel nach Kruchten [Kru95], dieses Konzept ist Bestandteil des Rational Unified Process [Kru00] oder die 4 Blickwinkel nach Hofmeister et al. [HNS99a], diese sind auch bekannt als Siemens-Sichten. Weitere Blickwinkel Definitionen finden sich bei Herzum und Sims [HS00, S. 49ff] oder im RM-ODP[6] Standard [CBB+03, S. 372], [VAC+05, S.87f]. Clements et al. führen darüber hinaus drei Kategorien von Blickwinkeln ein, die Viewtypes [CBB+03]. Auf den Standard IEEE 1471 und auf die Sichtenkonzepte aus der Software-Architektur geht das Kapitel 3 im Detail ein.

### 2.3.5 Zusammenfassung der Begriffe

Ein Modell hat jeweils eine Menge von Originalen. Diese Menge kann auch leer oder unendlich groß sein. Ebenso kann es zu jedem Original eine Menge von Modellen geben, diese Menge kann ebenfalls auch leer oder unendlich groß sein. Diese Beziehung wird durch eine Assoziation zwischen Modell und Original in Abbildung 2.4 dargestellt. Prozesse und Produkte sind Originale. Diese werden über Prozessmodelle bzw. Produktmodelle beschrieben. Die Assoziation zwischen Produkt und Produktmodell bzw. Prozess und Prozessmodell ist eine Spezialisierung der oben genannten allgemeinen Assoziation zwischen Modell und Original

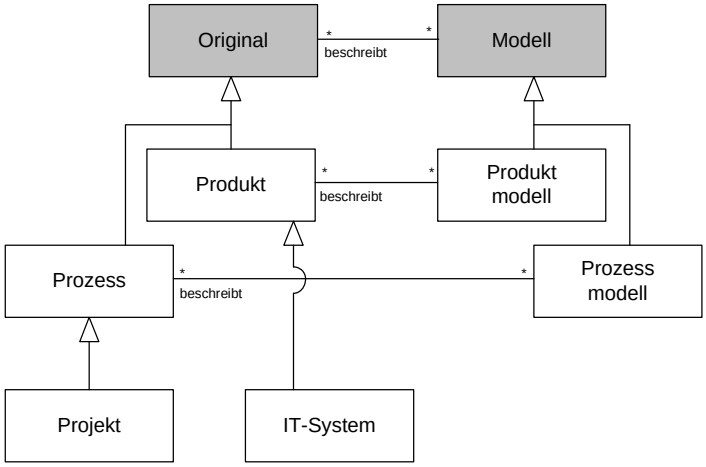


Abbildung 2.4: Begriffe der Produkt- und Prozessmodelle

Ein Modell kann mehrere Beschreibungen haben. Die Beschreibungen bestehen aus mindestens einem Beschreibungselement und sind über eine oder mehrere Sichten strukturiert. Jede Sicht enthält mindestens ein Element der Beschreibung. Jede Sicht ist weiterhin konform zu genau einem Blickwinkel. Der Blickwinkel definiert die Sprachen, in der die Beschreibungselemente einer Sicht abgefasst sein dürfen. Diese Zusammenhänge sind in Abbildung 2.5 dargestellt.

[6] Reference Model for Open Distributed Processing

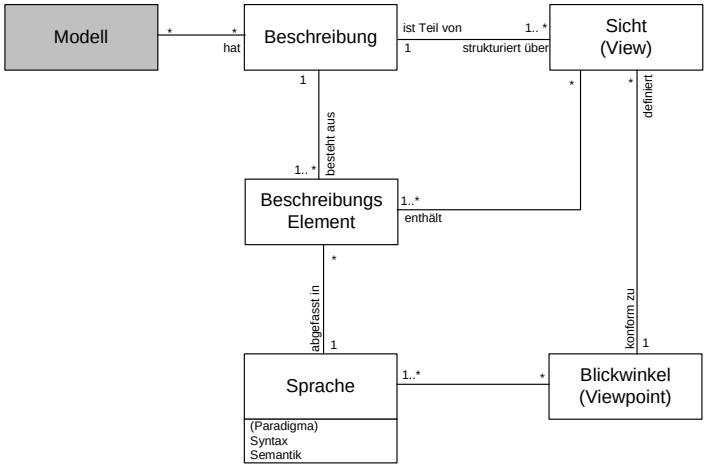


Abbildung 2.5: Begriffe der Produktmodelle

## 2.4 Anforderungen an betriebliche Informationssysteme

Die Systemkategorie "Betriebliches Informationssystem" wird durch eine Reihe typischer Anforderungen charakterisiert. Ihre Ausprägung ist von System zu System verschieden, auch die Prioritäten sind unterschiedlich. Sie sind jedoch für eine Schwerpunktsetzung im Architekturentwurf und der Architekturbeschreibung wichtig, denn sie beeinflussen die Gestalt der Architektur zum Teil wesentlich. Solche Anforderungen werden auch Architekturtreiber genannt[7]:

**Definition 2.16 (Architekturtreiber)**
Anforderungen, die Auswirkungen auf die Gestalt der System- und Software-Architektur haben, werden als Architekturtreiber bezeichnet.

Im Folgenden werden funktionale, wirtschaftliche und Qualitätsanforderungen definiert und diskutiert, die Architekturtreiber sein können.

### 2.4.1 Funktionale Anforderungen

Die funktionalen Anforderungen tragen wesentlich zum wirtschaftlichen Nutzen eines betrieblichen Informationssystems bei. Ihretwegen werden betriebliche Informationssysteme gebaut. Die funktionalen Anforderungen bestimmen letzthin, wie Geschäftsprozesse in welchem Umfang unterstützt werden. Die *Korrektheit* der Umsetzung der funktionalen Anforderungen ist eines der wichtigsten Qualitätsmerkmale eines Systems.

Auf eine umfangreiche Betrachtung der funktionalen Anforderungen verschiedener Branchen oder betrieblicher Aufgabenbereiche wird hier verzichtet. Detaillierte Informationen und Referenzmodelle für Geschäftsprozesse finden sich bei Stahlknecht und Hasenkamp [SH02, S. 338ff], bei Scheer [Sch97, S. 96ff] oder bei Mertens [Mer04].

**Definition 2.17 (Funktionale Anforderung)**
Eine funktionale Anforderung ist eine Bedingung oder Eigenschaft, die ein IT-System benötigt, um sein fachliches oder technisches Ziel zu erreichen. Sie bezieht sich nur auf die Außensicht des IT-Systems, wie sie sich Benutzern und Nachbarsystemen darstellt, insbesondere das Verhalten bei korrekten und fehlerhaften Eingaben. Eine Beschreibung der Bedingung oder Eingenschaft wird ebenfalls als funktionale Anforderung aufgefasst.

[7] *An architecture is "shaped" by some collection of functional, quality and business requirements. We call these shaping requirements archtectural drivers* [...] [BCK03, S.154f]

Ein betriebliches Informationssystem ist in die Struktur und die Abläufe eines Unternehmens eingebettet. Abteilungsgrenzen, Geschäftsprozesse und Organisationsstrukturen spiegeln sich in der Regel im System wider [LA02, WBFG03, CD94], dies ist in Abbildung 2.6 dargestellt. Beispielsweise richtet sich die Aufteilung der Dialoge und der Berechtigungen zur Datenpflege nach den Abteilungsgrenzen bzw. Aufgabenbereichen. Die Einkaufsabteilung könnte die Bestellungen pflegen, während die Vertriebsabteilung die Kundendaten pflegt.

Ändert sich die Organisation eines Unternehmens oder werden wesentliche Geschäftsprozesse modifiziert, hat das in der Regel auch Änderungen im IT-System zur Folge. Das System hängt von seiner Umwelt ab.

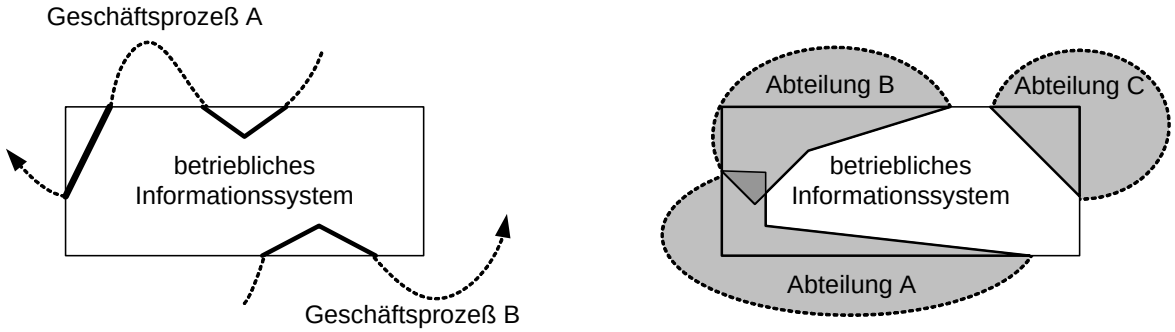


Abbildung 2.6: Einbettung in Organisationsstrukturen und Geschäftsprozesse

Während der Erstellung eines betrieblichen Informationssystems wird seine Systemgrenze festgelegt. Es wird entschieden, welche Teile der Geschäftsprozesse vom System als Anwendungsfälle zu unterstützten sind, und welche nicht. Dies ist ein aktiver Entwurfsschritt, der durch Kosten, Risiken und Erstellungszeiten beeinflusst wird.

**Geschäftsprozesse:**

Ein betriebliches Informationssystem unterstützt und steuert Geschäftsprozesse eines Unternehmens. Selten kann ein Geschäftsprozess, z.B. vom Bestellungseingang bis zur Auslieferung der Ware, vollständig über ein einziges System laufen. Teile des Geschäftsprozesses finden in der Umwelt des Systems statt. Sie werden ausgeführt durch Menschen, beispielsweise einen Sachbearbeiter, der einen Wahreneingang prüft oder durch Nachbarsysteme.

Geschäftsprozesse und IT-Systeme eines Unternehmens beeinflussen sich gegenseitig: Das IT-System wird so konstruiert, dass es bestehende Prozesse unterstützt, umgekehrt werden Geschäftsprozesse so geändert, dass sie durch IT-Systeme unterstützt werden können.

Die Art und Weise, wie ein Geschäftsprozess über ein betriebliches Informationssystem unterstützt wird, kann über Anwendungsfälle (Use Cases) beschrieben werden. *Anwendungsfälle zerlegen die Geschäftsprozesse in Portionen, die aus der Sicht des Anwenders sinnvoll und programmierbar sind* [Sie02b, S.53]. Anwendungsfälle stellen in der vorliegenden Arbeit die Verbindung zwischen den funktionalen Anforderungen und der Architektur her, daher folgt nun eine Definition:

**Definition 2.18 (Anwendungsfall, angelehnt an [Sie02b])**
Ein Anwendungsfall ist eine Folge von Interaktionen zwischen Akteuren und einem IT-System. Er beschreibt das sichtbare Verhalten des IT-Systems aus der Sicht und in der Sprache der Akteure. Akteure können menschliche Benutzer oder Nachbarsysteme sein. Mit einem Anwendungsfall wird durch das IT-System ein für den Anwender sinnvoller Dienst erbracht oder ein benutzbares Ergebnis erzielt. Ein Anwendungsfall ist Teil funktionalen Anforderungen an ein IT-System.

**Unternehmensorganisation:**

Die Organisationsstruktur definiert Zuständigkeiten und Aufgabenbereiche innerhalb eines Unternehmens. Diese Zuständigkeiten erstrecken sich auch auf Teile der Geschäftsprozesse. Eine Abteilung bearbeitet beispielsweise den Auftragseingang während eine andere die Aufträge in die Produktion weiterleitet. Das System ist in die Organisationsstruktur eingebunden. Es kann beispielsweise die Kommunikation zwischen zwei Abteilungen über einen automatisierten Workflow regeln.

Unternehmensorganisation und IT-Systeme beeinflussen sich gegenseitig, das IT-System wird beispielsweise so konstruiert, dass es die Kommunikation in der bestehenden Organisation optimal unterstützt, umgekehrt kann die Organisation an den zentralen Strukturen des IT-Systems ausgerichtet werden.

### 2.4.2 Qualitätsanforderungen

Die Norm ISO 9126-1991 [ISO91, DIN94] definiert sechs zentrale Qualitätsmerkmale von IT-Systemen, das sind Funktionalität, Zuverlässigkeit, Benutzbarkeit, Effizienz, Änderbarkeit und Übertragbarkeit. Die Merkmale können grundsätzlich auf jede Art von Software angewandt werden [ISO91, S.3]. Die Norm macht zusätzlich einen Vorschlag für die Detaillierung dieser Eigenschaften in 21 Teilmerkmale, die Detaillierung ist jedoch nicht Bestandteil der Norm.

Anforderungen an Qualitätsmerkmale eines IT-Systems sind für den Entwurf ihrer Architekturen wichtig: Sie beeinflussen die verwendeten Modelle im Entwicklungsprozess und bestimmen deren Aussehen wesentlich [RW05] - Sie sind in der Regel Architekturtreiber. Dies wird im Abschnitt 3.3 im Detail dargelegt.

**Definition 2.19 (Qualitätsanforderung)**
Qualitätsanforderungen definieren gewünschte Qualitätsmerkmale eines IT-Systems. Beispiele für Merkmale, zu denen häufig Anforderungen formuliert werden, sind Sicherheit, Verfügbarkeit, Änderbarkeit oder Effizienz.

Im Folgenden werden Qualitätsmerkmale beschrieben, bei denen Anforderungen häufig Architekturtreiber beim Entwurf betrieblicher Informationssysteme sind [RW05].

**Sicherheit (Security)**

Die Bedeutung der Sicherheit (im Sinne von Security) hat in den vergangenen Jahren zugenommen. Durch die weltweite Vernetzung der Computer über das Internet, kommt es vermehrt zu Virenbefall, Hacker-Angriffen (z.B. Industriespionage) oder Denial-Of-Service Attacken. Davor müssen insbesondere geschäftskritische Systeme geschützt werden. Teileigenschaften der Security sind etwa Vertraulichkeit und Verbindlichkeit, diese werden hier beispielhaft dargestellt.

Daten, die betriebliche Informationssysteme verwalten, sind in der Regel vertraulich und nur für einen bestimmten Personenkreis bestimmt. Daten über Kunden, Verträge oder Konten fallen unter das Datenschutzgesetz. Daten über die Leistungen und die Produktivität von Mitarbeitern werden vom Betriebsrat reglementiert. Daten über Umsätze, Gewinne oder Kunden fallen unter das Betriebsgeheimnis von Unternehmen. Vertraulichkeit ist daher eine zentrale Eigenschaft. Mechanismen, die zu ihrer Sicherstellung eingesetzt werden, sind Authentifizierung, Zugriffskontrolle und Verschlüsselung.

Mit einem betrieblichen Informationssystem werden Teile von Geschäftsprozessen abgewickelt, auch direkt durch Kunden oder Geschäftspartner. Die mit dem System durchgeführten Aktionen müssen verbindlich sein. Das bedeutet, dass der- oder diejenige, der/die die Aktion durchgeführt hat, zweifelsfrei nachgewiesen werden kann. Diese Eigenschaft wird auch als Zurechenbarkeit bezeichnet. Bei

geschäftskritischen Systemen werden häufig die Aktionen der Benutzer protokolliert, um dieses bei Bedarf nachvollziehen zu können, dies wird auch als *Audit-Trailing* bezeichnet, die dazu gehörende Eigenschaft wird auch als Revisionsfähigkeit bezeichnet.

#### Zuverlässigkeit

Die Norm ISO 9126 betont die Eigenschaft Zuverlässigkeit (reliability) mit den Teileigenschaften Reife, Fehlertoleranz und Wiederherstellbarkeit. Ein Maß dafür ist die Zeit zwischen zwei Ausfällen, die MTBF[8]. Der Ausfall eines geschäftskritischen Systems bedeutet den Stillstand der Geschäftstätigkeit eines Unternehmens.

Ein betriebliches Informationssystem darf durchaus ausfallen, solange beim Ausfall keine Daten beschädigt werden und das System schnell (innerhalb weniger Sekunden bis Minuten) wieder zur Verfügung steht. Wenn etwa das Checkin-System der Fluglinie für eine Minute ausfällt, wird dies von den Passagieren vermutlich kaum bemerkt. Ein Ausfall von mehreren Stunden ist dagegen nicht akzeptabel. Ausfälle eines betrieblichen Informationssystems werden also durchaus toleriert, lange Ausfallzeiten jedoch nicht.

Die Verfügbarkeit während der Geschäftszeiten ist also wichtiger als die Zuverlässigkeit. Ein Maß für die Verfügbarkeit ist die Wahrscheinlichkeit, dass das System zu einem bestimmten Zeitpunkt seine geforderten Aufgaben erledigt. Einflussfaktoren auf die Verfügbarkeit sind beispielsweise die Startzeit des Systems, also die Zeit bis das System nach einem Absturz wieder gestartet ist, die Zeit zur Korrektur einer kaputten Transaktion, sowie die Zeit, um ein Backup in die Datenbank einzuspielen. Kurze Start- bzw. Reparaturzeiten führen zu einer guten Wiederherstellbarkeit des Systems.

#### Effizienz (efficiency)

ISO 9126 schlägt das Zeitverhalten und das Verbrauchsverhalten als Teil der Effizienz vor. Das Zeitverhalten eines Systems wird häufig als Performance bezeichnet. Dahinter verbergen sich die Eigenschaften Antwortzeit und Durchsatz.

Die Antwortzeiten eines Systems im Online-Betrieb werden in Sekunden gemessen und sind spezifisch für jeden aufgerufenen Dialog. Komplexe Operationen dauern in der Regel länger, als einfache Anfragen. Typischerweise liegen geforderte Antwortzeiten zwischen 3 und 10 Sekunden für häufig verwendete Dialoge. Für selten verwendete Dialoge werden Überschreitungen dieser Zeiten akzeptiert. Definierte Antwortzeiten im Sinne der Echtzeitfähigkeit werden nicht gefordert.

Batches haben häufig harte obere Grenzen für ihre Laufzeiten, solche Batches laufen in Zeiten, in denen das System nicht online zur Verfügung steht. Diese Zeiträume werden auch Batch-Fenster genannt. Batches sind technisch häufig so programmiert, dass sie das System oder die Datenbank exklusiv belegen, d.h. andere Batches oder Online-Betrieb sind gleichzeitig nicht möglich. Daher ist die Laufzeit auf die Dauer des Batch-Fensters oder einen definierten Teil davon beschränkt.

Der Durchsatz eines Systems kann in Transaktionen pro Sekunde gemessen werden. Im Rahmen der Anforderungsanalyse muss der geforderte Durchsatz über die Nutzerzahl und ihr Arbeitsprofil bestimmt werden. Dabei ist die theoretische Nutzerzahl nicht von Bedeutung, sondern die maximale Zahl der Nutzer, die gleichzeitig das System verwenden (in einer Lastspitze). Die tatsächliche Systemlast hängt zusätzlich von der Art ab, wie die Nutzer das System verwenden. Handelt es sich beispielsweise um die reine Betrachtung gespeicherter Daten, Reporting oder Modifikation von Daten? Die Transaktionslast ist bestimmt durch die Zahl der Datenänderungen in einem bestimmten Zeitintervall.

Das **Verbrauchsverhalten** ist in betrieblichen Informationssystemen weniger kritisch als beispielsweise in eingebetteten Systemen. Moderne Bürorechner haben eine in der Regel ausreichende Re-

[8] Mean Time Between Failure

chenleistung, so dass der Ressourcenverbrauch dort für den Entwurf immer weniger wichtig wird. Begrenzt sind zentrale Rechenkapazitäten an Applikationsservern bzw. Transaktionsmonitoren, zentralen Diensten und Datenbanken. Diese Ressourcen werden von vielen Clients gleichzeitig genutzt. Entwurfsziel ist daher, den Ressourcenverbrauch pro Client möglichst gering zu halten. Netzwerkkapazitäten sind ebenfalls begrenzt. Ein weiteres Entwurfsziel ist es damit, Netzwerkkommunikation gering zu halten. Das bedeutet, möglichst wenig Datenbankanfragen zu starten und möglichst wenig Aufrufe an zentrale Server zu machen. Weiterhin wird versucht, die übertragenen Datenmengen zwischen Datenbanken, Applikationsservern und Clients gering zu halten, beispielsweise durch Caching von Daten.

#### Änderbarkeit (maintainability)

Änderbarkeit ist nach DIN 66272 die deutsche Übersetzung der Eigenschaft Maintainability. Betriebliche Informationssysteme stellen erhebliche Investitionen dar und können, da sie häufig geschäftskritisch sind, nur mit hohem Risiko ausgetauscht werden. Änderungen in den Anforderungen treten häufig auf, Geschäftsprozesse und Trägersysteme ändern sich laufend. Die Änderbarkeit und Anpassbarkeit eines Systems sind daher eine wichtige Anforderungen.

Eine schlechte Verständlichkeit der Quelltexte ist häufige Ursache für das Quelltext-Wachstum in der Wartungsphase, obwohl nicht immer Funktionalität ergänzt wird: Wartungsprogrammierer fassen einen Quelltext, den sie nicht verstehen, lieber nicht an [FY97]. Die Analysierbarkeit ist daher für die Änderbarkeit eines Systems wichtig.

Die Modifizierbarkeit eines Systems spielt ebenfalls eine wichtige Rolle. Die Modifizierbarkeit bestimmt die Zeit, um eine Änderung im System zu implementieren und die notwendigen Verwaltungsaktivitäten (Konfigurationsmanagement etc.) durchzuführen. Die Modifizierbarkeit ist vor allem von den verwendeten Werkzeugen abhängig. Entwicklungsumgebungen für aktuelle Programmiersprachen erlauben beispielsweise sehr schnelle Übersetzung des Codes und teilautomatisierte Änderungen wie das Refactoring.

Nach einer Modifikation muss das System geprüft werden, um festzustellen ob es noch korrekt funktioniert und die Änderung richtig umgesetzt wurde. Die Prüfbarkeit (Testbarkeit) ist daher ein wichtiger Teil der Änderbarkeit. Sind beispielsweise automatisierte Regressionstests vorhanden und arbeitet das System mit diesen gut zusammen, verbessert das die Prüfbarkeit des Systems.

Die Stabilität der verwenden Bestandteile hat Einfluss auf die Änderbarkeit eines Systems. Stabilität bedeutet, dass bei Änderungen keine unerwarteten Seiteneffekte auftreten. Änderungen in der Software sollen nur lokale Auswirkungen haben.

### 2.4.3 Wirtschaftliche Anforderungen

Der Wert eines betrieblichen Informationssystems wird durch seinen (betriebswirtschaftlichen) Nutzen bestimmt. In einigen Fällen ist dieser Nutzen direkt messbar, etwa wenn durch das System Sachbearbeiter ersetzt werden können. Das eingesparte Gehalt ist der Nutzen des Systems in Euro. Die Messung des Nutzens ist in den meisten Fällen problematisch. Die Kundenzufriedenheit oder der Einfluss auf den Umsatz und die Marktposition eines Unternehmens sind kaum in Euro zu bewerten.

Festzuhalten ist, dass die im System implementierten funktionalen Anforderungen den Nutzen des Systems bestimmen. Über die Architektur des IT-Systems ist der wirtschaftliche Nutzen kaum beeinflussbar, dagegen werden die Entwicklungs-, Wartungs- und Betriebskosten wesentlich dadurch bestimmt [BCK03]. Über eine gute Architektur können diese Kosten gemindert werden und über die Erstellungsreihenfolge der Systembestandteile kann der Zeitpunkt beeinflusst werden, wann welcher Nutzen für Kunden, Sachbearbeiter oder Geschäftspartner zur Verfügung steht. Anforderungen

wie Kostenminderung, Erstellungszeit oder beherrschbare Risiken können damit auch Architekturtreiber sein. Die genannten Anforderungen werden als *wirtschaftliche Anforderungen* bezeichnet:

**Definition 2.20 (Wirtschaftliche Anforderung)**
Anforderungen an Kosten, Zeitrahmen und Risiken zum Bau, Betrieb und der Weiterentwicklung von IT-Systemen werden als wirtschaftliche Anforderungen bezeichnet.

#### Kostenminderung

Drei große Kostenblöcke fallen beim Bau und Betrieb betrieblicher Informationssysteme an, das sind Erstellungskosten, Betriebskosten und Wartungskosten. Diese Kosten werden unter dem Begriff Lebenszykluskosten zusammengefasst. Die Lebenszykluskosten dürfen den wirtschaftlichen Nutzen nicht übersteigen. Die Erstellungskosten werden durch den Entwicklungsprozess beeinflusst, z.B. durch Zahl und Umfang der zu erstellenden Produkte oder die Ausprägung der Qualitätssicherung. Weitere Einflussgröße ist die Architektur des Systems. Die Betriebskosten werden durch die Eigenschaften wie Security oder Verfügbarkeit bestimmt. Die Wartungskosten hängen von den Erstellungskosten ab. Boehm gibt empirische Hinweise darauf, dass beide Kosten proportional zueinander sind [Boe81]. Weiterhin werden die Wartungskosten durch die Änderbarkeit des Systems bestimmt.

#### Erstellungs- und Änderungszeit (Time-to-Market)

Ein betriebliches Informationssystem wird häufig über einige Jahre hinweg in Stufen [Sie02b, Tau04] und Iterationen [JRB99] entwickelt. Die Stufen werden jeweils an die Benutzer ausgeliefert. Wichtig sind in diesem Zusammenhang zwei Anforderungen: Erstens muss die Zeit bis zur Einführung des Systems kurz sein, sonst ändern sich in der Zwischenzeit Geschäftsprozesse oder konkurrierende Unternehmen bieten die selbe Dienstleistung früher an und schöpfen damit anvisierte Marktanteile ab. Zweitens müssen die Stufen für sich genommen wirtschaftlich nützlich sein. Die nützlichste Stufe wird nach Möglichkeit zuerst ausgeliefert.

#### Investitionssicherheit und Risikobeherrschung

Software ist Investitionsgut und in der Regel langlebig. Investitionssicherheit ist daher eine wichtige Anforderung. Die Lebensdauer eines Systems hängt von seiner Änderbarkeit ab, da sich in der Regel die durch das System unterstützten Geschäftsprozesse permanent ändern und das System diesen veränderten Bedingungen angepasst werden muss.

Ein besonderes Risiko stellen Fremdkomponenten (COTS[9]) und -produkte in der Architektur dar. Die Fremdprodukte sind zum Betrieb des Systems zwingend erforderlich, beispielsweise Datenbankserver, Applikationsserver oder GUI-Bibliotheken. Sie werden auch für die Änderung des Systems gebraucht, beispielsweise die Entwicklungsumgebung, ein Compiler oder ein Konfigurationsmanagement Werkzeug. Ohne die verwendeten Fremdprodukte kann das System weder betrieben noch geändert werden[10]. Das System hängt also von diesen Produkten und damit von ihren Herstellern ab. Die Investitionssicherheit eines IT-Systems wird durch die wirtschaftliche Stabilität und Größe sowie die Produktpolitik der Hersteller von Fremdprodukten beeinflusst. Diese Abhängigkeit kann gemildert, jedoch nicht ausgeschlossen werden.

[9] Commercial off the Shelf
[10] Seifert und Beneken diskutieren dieses häufig übersehene Problem am Beispiel der MDA [SB05]

## 2.5 Ein Bringdienstsystem als durchgehendes Beispiel

Ein betriebliches Informationssystem für einen (Pizza-)Bringdienst dient im Laufe der vorliegenden Arbeit als Beispiel. Einen Eindruck von dem Funktionsumfang eines solchen Systems vermittelt etwa `www.bringdienst.de`. In der vorliegenden Arbeit werden Planung und Architekturentwürfe des Systems zur Veranschaulichung und als durchgehende Fallstudie präsentiert. Das System ist software-technisch jedoch nicht komplett umgesetzt. Eine erste Architekturskizze des Bringdienstsystems ist in Abbildung 2.1 dargestellt.

#### Ziel des Systems

Von dem Bringdienst wird angenommen, dass er über mehrere Filialen verfügt und sich somit die Kosten für den Bau eines eigenen IT-Systems auf die verschiedenen Standorte umlegen lassen. Das Bringdienstsystem soll die betrieblichen Abläufe innerhalb der Filialen des Bringdienstes automatisieren und Bestellungen von Kunden über das Internet ermöglichen. Es ist nach der Definition 2.4 ein betriebliches Informationssystem.

#### Grobe funktionale Anforderungen

Die Mitarbeiter in der Filiale müssen (telefonische) Bestellungen mit dem System erfassen können. Die Bestellungen werden als Rechnungen ausgedruckt und dienen als Arbeitsaufträge an die Küche und die Fahrer, sowie später als Beleg für die belieferten Kunden. Zur Auslieferung wird eine fertig gestellte Bestellung direkt mit dem Fahrer abgerechnet, dieser rechnet seinerseits mit dem Kunden vor Ort ab. Die Daten über ausgelieferte Bestellungen sollen innerhalb des Systems vorgehalten werden und an ein Buchhaltungssystem übermittelt werden.

Um Bestellungen verwalten zu können, benötigt das Bringdienstsystem eine Verwaltung der lieferbaren Artikel, also eine Verwaltung der Speisekarte mit Speisen und Getränken. Die lieferbaren Artikel müssen angelegt, geändert und gelöscht werden können. Auch Daten über Kunden müssen im System verwaltet werden können, etwa um die Anschrift des Kunden auf die Rechnung drucken zu können.

Kunden sollen in einer zweiten Ausbaustufe aus dem Internet auf das System zugreifen können: Nach ihrer Identifizierung können sie online Speisen und Getränke bestellen. Diese Bestellungen werden in der zuständigen Filiale als Rechnungen gedruckt und danach wie telefonische Bestellungen bearbeitet. Die zuständige Filiale wird nach der Adresse des Kunden ermittelt.

#### Das Entwicklungsprojekt

Das Bringdienstsystem wird in einem fiktiven Projekt entwickelt. Das Projektteam ist über mehrere Standorte in Europa verteilt. Es stehen Teams in Deutschland und bei einem Nearshore-Partner in Bulgarien (Sofia) zur Verfügung. Die Buchhaltungs-Software wird von einer britischen Firma geliefert.

In der vorliegenden Arbeit wird die Planungs- und Entwurfsphase des fiktiven Projekts dargestellt. Hierfür wird die grobe logische Architektur des Systems entworfen und Teile der Projektplanung werden durchgeführt. Die Aufgaben und (Entwurfs-)Probleme des Architekten und des Projektleiters werden diskutiert.

## 2.6 Zusammenfassung

Thema der vorliegenden Arbeit ist die Beschreibung von System- und Software-Architekturen von betrieblichen Informationssystemen. Dieses Kapitel definiert die begrifflichen Grundlagen für die nachfolgenden Kapitel:

Betriebliche Informationssysteme sind IT-Systeme. Diese sind strukturiert in Software-Systeme und Trägersysteme. Trägersysteme führen die Software-Systeme aus. Typischerweise wird ein Software-System im Rahmen eines oder mehrerer Projekte entwickelt. Das Trägersystem wird zugekauft oder ist bereits vorhanden. Geschieht die Entwicklung eines Systems speziell für einen Auftraggeber, wird von Individual-Software gesprochen.

Projektleiter und Architekt(Chefdesigner) sind zwei zentrale Rollen innerhalb eines Software-Entwicklungsprojekts. Der Projektleiter ist verantwortlich für die Planung, Kontrolle und Steuerung des Projektes. Der Architekt ist dafür verantwortlich, dass das entwickelte IT-System funktionale, wirtschaftliche und Qualitätsanforderungen erfüllt. Beide Rollen müssen eng zusammenarbeiten. Die vorliegende Arbeit hat das Ziel, diese Zusammenarbeit zu unterstützen.

Die Architekturen und ihre Beschreibungen sind Modelle von IT-Systemen. Die IT-Systeme sind die entsprechenden Originale. Beschreibungen werden über Sichten strukturiert. Eine Sicht beschreibt dabei die Architektur für einen Problembereich oder spezifisch für eine Stakeholdergruppe. Ein Blickwinkel legt Sprache und Inhalte einer Sicht fest. Das nachfolgende Kapitel stellt Details zu System- und Software-Architekturen und ihrer Beschreibung dar. Dort werden auch die wichtigsten Begriffe dieses Themenfeldes definiert. Vorschläge für neue Architektursichten bzw. -blickwinkel zur Unterstützung der Zusammenarbeit von Architekt und Projektleiter bilden einen wesentlichen Beitrag dieser Arbeit.

Betriebliche Informationssysteme bilden den Anwendungsbereich der vorliegenden Arbeit. Sie unterstützen und automatisieren die Geschäftsprozesse von Unternehmen und werden von Sachbearbeitern, Führungskräften sowie Kunden und Partnern eines Unternehmens verwendet. Betriebliche Informationssysteme sind eingebunden in eine Umgebung aus Nachbarsystemen und vorhandener IT-Infrastruktur. Ein System für einen Bringdienst dient als Anschauungsbeispiel. Daran werden Aspekte des Architekturentwurfs sowie der Projektplanung und -steuerung demonstriert.

Architekturtreiber sind Anforderungen, welche die Gestalt und die Inhalte einer Architektur wesentlich beeinflussen. Dieses Kapitel stellt Qualitätsanforderungen und wirtschaftliche Anforderungen an betriebliche Informationssysteme als Beispiele für Architekturtreiber vor. Qualitätsanforderungen wie Sicherheit (Security), Verfügbarkeit, Effizienz und Änderbarkeit beeinflussen den Architekturentwurf. Wirtschaftliche Anforderungen, etwa Kostenminderung, Erstellungs- und Änderungszeit sowie Investitionssicherheit und Risikobeherrschung sind ebenfalls Architekturtreiber.

# Kapitel 3

# Architekturbeschreibung

Dieses Kapitel definiert die Begriffe Systemarchitektur und Software-Architektur. Die begrifflichen Grundlagen für die Theorie logischer Architekturen im Kapitel 5 werden in Abschnitt 3.2 geschaffen. Der zentrale Begriff *logische Architektur* wird in Abschnitt 3.3 definiert und von anderen Architekturarten abgegrenzt. Bekannte Ansätze zur Beschreibung von System- und Software-Architekturen werden in den Abschnitten 3.4 und 3.5 dargestellt. Betrachtet werden Architektursichten, UML 2.0, Architecture Description Languages und (Auto-)FOCUS 2.

## Übersicht

## 3.1 Rolle der Software- und Systemarchitektur

**Beherrschung von Komplexität**

Wenn die Anforderungen an IT-Systeme komplex und umfangreich sind, ist damit in der Regel auch die Implementierung dieser Anforderungen komplex und umfangreich. Häufig ist die Komplexität der Implementierung noch höher, als die des eigentlichen Problems[1]. Einer der wichtigsten Ansprüche der Disziplin Software-Architektur ist daher Umfang und Komplexität durch Strukturierung beherrschbar zu machen. Jazajeri et al. formulieren dies so [JRvdL02, S. 4]: *We understand software architecture as a conceptual tool for dealing with the complexity of software.*

Typische Methoden zur Beherrschung eines komplexen Problems sind hierarchische Zerlegung (Dekomposition) in kleinere, einfachere Einzelteile einerseits und Abstraktion andererseits.

Die beherrschbaren Einzelteile werden typischerweise Komponenten oder Module genannt. Wichtiges Merkmal dieser Einzelteile ist, dass sie nach dem *Geheimnisprinzip*[2] [Par72] ihren inneren Aufbau verbergen und nur eine schmale Schnittstelle veröffentlichen. Es genügt dann die Außensicht eines Einzelteils zu kennen, um es verwenden zu können, sein möglicherweise komplexer interner Aufbau bleibt verborgen. Die Schnittstellen sind eine Abstraktion des jeweiligen Einzelteils.

**Projektmanagement**

IT-Systeme mit umfangreichen Anforderungen können nur arbeitsteilig in (großen) Teams erstellt werden. Eine Erstellung in kleinen Teams oder durch Einzelpersonen würde zu lange dauern [Bro03]. Die arbeitsteilige Entwicklung muss geplant und organisiert werden. Die Architektur ist dafür eine Grundlage [Bur02, S.76ff].

Typischerweise wird die Erstellung eines Einzelteils an ein Teilteam, eine Organisationseinheit oder einen Lieferanten vergeben. Software-Architektur und die Organisation des Entwicklungsteams gleichen sich daher häufig. Conway [Con68] weist bereits 1968 auf diesen Sachverhalt hin, diese Beobachtung ist als Conway's Law bekannt geworden[3].

Die Architektur beeinflusst die Kommunikation in den Entwicklungsteams: Je klarer und stabiler die Schnittstellen zwischen den Einzelteilen des IT-Systems formuliert sind und je weniger die Einzelteile untereinander vernetzt sind, desto weniger müssen die Entwickler miteinander kommunizieren, um Unklarheiten zu beseitigen [HG99a]. Schnittstellen zwischen den Einzelteilen sind auch Kommunikationsschnittstellen zwischen den Entwicklern [DeM05, S.223].

Die Architektur bildet daher die Grundlage für die Organisation des Projekts sowie der Zulieferer und Konsortialpartner. Die Schnittstellen der Komponenten sind zum Teil auch die jeweiligen Grenzen der Organisationen und Grundlage für Verträge zu deren Zusammenarbeit.

Für Aufwandsschätzungen zum Bau eines IT-Systems ist eine grobe Architekturbeschreibung erforderlich [Sie02b, S. 328], die dort identifizierten Komponenten dienen als Schätzpositionen in einer Stückliste (einem Projektstrukturplan). Die Erstellungsaufwände für die Schätzpositionen werden dann beispielsweise von Experten geschätzt. Ergebnis der Schätzung ist der Nettoaufwand, also die reine Spezifikations- bzw. Entwicklungszeit ohne Qualitätssicherung oder Projektleitung.

Bei der Implementierung eines Software-Systems sind Maße zur Feststellung des Projektfortschritts erforderlich, um das Projekt steuern zu können. Verbrauchter Aufwand ist bekanntlich ein schlechtes Maß. Besser geeignet ist der Fertigstellungsgrad [HHMS04, S.73ff] des IT-Systems. Die Zahl der im-

[1] Brooks spricht in diesem Zusammenhang von *accidential complexity* [Bro87]

[2] Information Hiding

[3] Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations. Conway's Law wird in verschiedenen Veröffentlichungen bis heute diskutiert, vgl. Endres und Rombach [ER03, S.82f] oder Herbsleb und Grinter [HG99a]

plementierten und getesteten Komponenten gibt eher einen Anhaltspunkt für den Projektfortschritt.

#### Kosten-Nutzen Diskussion

Die Software-Architektur beeinflusst Qualitätseigenschaften wesentlich, wie Antwortzeiten oder Verfügbarkeit [BCK03, RW05]. Die Umsetzung vieler Qualitätseigenschaften ist mit zum Teil hohen Kosten verbunden: Wenn etwa vom IT-System gefordert wird, dass es von einem Benutzer bis zu 1.000.000 Nutzern skalieren soll, ist dies in der Regel nur mit großen Anstrengungen in Hardware und Software möglich. Je flexibler ein IT-System sein soll, desto mehr Abstraktionsstufen, Schichten und Komponentenaufteilungen sind normalerweise erforderlich. Entwurf und Umsetzung dieser Flexibilisierungen kostet Entwicklungsaufwand.

Die Anforderungen zusammen mit dem Auftraggeber so zu präzisieren, dass ein IT-System mit einem angemessenen Kosten-Nutzen-Verhältnis erstellt werden kann, ist eine wichtige Aufgabe im Architekturentwurf. Eine Beschreibung der Architekturalternativen bildet die Grundlage, um diese Diskussion zu führen.

#### Prognose von Qualitätsmerkmalen

Die Erfüllung von Qualitätsanforderungen sollte früh im Entwicklungsprozess sichergestellt werden, da entsprechende Änderungen in späteren Projektphasen aufwändig oder sogar unmöglich sind. Drei Verfahren können in frühen Projektphasen helfen, die Qualitätsmerkmale eines IT-Systems vorherzusagen und damit eventuelle Risiken aufzeigen: (1) Modelle des IT-Systems werden erstellt und mit Werkzeugen analysiert oder simuliert, (2) der Entwurf wird mithilfe von Experten überprüft und/oder (3) die wichtigsten Elemente der Architektur werden prototypisch erstellt und der entstandene technische Durchstich [HT99, Sie02b, Bec99] wird analysiert.

Eine ganze Reihe von Architekturbeschreibungssprachen dient dazu, Qualitätsmerkmale eines IT-Systems bereits in frühen Entwurfsphasen prognostizierbar zu machen. Über Beschreibungen in der Sprache WRIGHT [All97] können beispielsweise Nebenläufigkeitseigenschaften untersucht werden, die Sprache AEMILIA ist für Performance-Untersuchungen verwendbar [BBS03].

Qualitätssichernde Review-Verfahren wie ATAM[4] [BCK03] verwenden eine Architekturbeschreibung und analysieren diese mit Hilfe von Experten. Im Zentrum der Analyse stehen Qualitätsanforderungen sowie Risiken in Bezug auf deren Erfüllung. Zusätzliches Ergebnis von ATAM-Workshops ist in der Regel ein besseres Verständnis der Qualitätsanforderungen [BCK03].

#### Erhaltung der Änderbarkeit und Minderung der Lebenszykluskosten

Wenn Software über einen längeren Zeitraum entwickelt wird, wachsen in der Regel Abhängigkeiten in den Quelltexten. Die Software altert [Par94]. Häufig entsteht ein Software-Monolith[5], bei dem Änderungen wegen der starken Abhängigkeiten immer aufwändiger werden. Typische Merkmale eines Software-Monolithen werden von Foote und Yoder in [FY97, S.6] dargestellt, etwa viele globale Variable, auf die aus allen Programmteilen unkontrolliert zugegriffen wird, ein kaum verständlicher Kontrollfluss und duplizierte Quelltexte.

Eine der Ursachen für ein solches Wachstum ist häufig eine nicht oder nur schlecht definierte grobteilige Struktur eines IT-Systems, insbesondere seiner Quelltexte, und fehlende kontinuierliche Kontrollen der Quelltexte auf die Einhaltung von Architekturvorgaben [FY97, Lak96].

Die langfristige Erhaltung der Änderbarkeit einer Software und die Vermeidung des unkontrollierten, monolithischen Systemwachstums ist eine der zentralen Aufgaben der Software-Architektur.

[4] Architecture Tradeoff Analysis Method
[5] Alias *Big Ball of Mud* [FY97]

Die Architektur eines IT-Systems muss Mechanismen definieren und bereitstellen, um große Mengen von Quelltexten, an denen viele verschiedene Entwickler arbeiten, über einen langen Zeitraum zu beherrschen.

#### Grundlage der Implementierung

Die Architektur wird über Programme (ihre Implementierung) umgesetzt. Aus den Architekturkomponenten werden beispielsweise Pakete oder Namensräume in den Quelltexten. Die in der Architektur definierten Schnittstellen der Komponenten werden mit Hilfe einer Programmiersprache umgesetzt. Die Umsetzung geschieht in der Regel teilautomatisiert oder manuell.

Die Architekturbeschreibung wird auch als Grundlage für die Generierung von Quelltexten bzw. Quelltextrahmen verwendet. Dies ist eine der Kernideen der Model Driven Architecture (MDA) [Fra03].

Ein wichtiges Problem bei der Definition von Software-Architekturen besteht darin, diese so zu formulieren, dass sie alle Entwickler verstehen. Was Entwickler nicht verstanden haben, können sie auch nicht umsetzen. Architekturbeschreibungen sind Kommunikationsmittel.

#### Illustrationen und Kommunikationsmittel

Architekturdiagramme werden als informelles Kommunikationsmittel während der Software-Entwicklung eingesetzt [SP02]. Beispielsweise wird in einer Besprechung ein UML-Klassendiagramm an ein Flipchart gemalt, um die Struktur eines Problems oder eines Entwurfs darzustellen. Das Diagramm wird mit den anwesenden Personen diskutiert und ggf. wird eine Entscheidung dazu im Team getroffen. Die verwendete Notation kann informell sein, muss aber von allen beteiligten Personen verstanden werden.

Ähnlich werden (UML-)Diagramme in Architekturdokumenten verwendet: Die Software wird durch einen Text, Tabellen und informelle Grafiken beschrieben. Diagramme dienen zur Verdeutlichung und zur Illustration bestimmter Aspekte [Sie02b]. In der Architekturdokumentation eines betrieblichen Informationssystems wird dessen Grobstruktur beispielsweise durch ein UML-Komponentendiagramm dargestellt. Der begleitende Text beschreibt die im Diagramm enthaltenen Komponenten und Tabellen beschreiben die Operationen ihrer Schnittstellen. Diese Form der Software-Entwicklung wird auch *dokumentenbasiert* [Den93] genannt.

#### Vereinheitlichung und Wiederverwendung

Referenzarchitekturen [Ben06, BKPS04] sind abstrakte Architekturen bzw. abstrakte Teilarchitekturen. Sie sind in unterschiedlichen Anwendungskontexten über Unternehmens- und Produktgrenzen wiederverwendbar. Innerhalb eines Unternehmens kann eine Referenzarchitektur zur Einordnung wiederverwendbarer Software-Komponenten und anderer Artefakte genutzt werden, da sie Strukturen von IT-Systemen vereinheitlicht. Im Rahmen der Strukturen können Lösungselemente implementiert werden[6].

[6] vgl. z.B. Kössler in [BKPS04]

## 3.2 Architekturbegriffe

### 3.2.1 Systemarchitektur und Software-Architektur

Im Folgenden werden mehrere allgemeine Definitionen der Begriffe Software-Architektur und Systemarchitektur aus der Literatur betrachtet. Eigenschaften und Gemeinsamkeiten dieser Definitionen werden herausgearbeitet. Die Definitionen sind in der Regel informell, ihnen liegt keine explizite Architekturtheorie bzw. ein formales, mathematisches Systemmodell zugrunde. Alle Definitionen betonen die Struktur(en) bzw. die Organisation der Software oder des Systems.

Eine häufig zitierte Definition des Begriffs Software-Architektur stammt von Bass, Clements and Kazman [BCK03, S. 3]:

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.

Bass et al. sehen Software-Architektur als Struktur eines Programms oder eines Systems. Die Strukturen bestehen aus Software-Elementen. Ein *Software-Element* kann ein Objekt, ein Prozess, eine Datenbank, ein kommerzielles Produkt oder ein beliebiges anderes Element sein [BCK03, S. 22]. Die Definition legt sich hier bewusst nicht fest. Auch die Beziehungen zwischen den Software-Elementen und ihre außen beobachtbaren Eigenschaften gehören zur Software-Architektur.

Dieser Architekturbegriff ist schwach: Er legt sich nicht fest, was unter einem Software-Element oder unter einer Struktur zu verstehen ist. Wie noch in Abschnitt 3.3.1 dargestellt wird, gehören zu einer Software bzw. einem IT-System mehrere wesentlich verschiedene Darstellungen, etwa Bibliotheken, Prozesse, Quelltexte oder strukturierte Anforderungen. In allen Darstellungen gibt es verschiedene Strukturen, die verschiedenen Anforderungen genügen müssen. Demzufolge unterscheiden sich auch die strukturbildenden Elemente. Schon Parnas zeigt diesen Sachverhalt in [Par74] am 'Buzzword' *hierarchische Struktur* auf. Daher sind spezifischere Architekturbegriffe erforderlich, die genauer festlegen, was dort genau unter Software-Elementen zu verstehen ist[7][8].

In der Definition von Bass et al. wird der Begriff *computing system* verwendet, wenn damit IT-System und nicht Software-System gemeint ist, führt dies auf eine häufig auftretende Unschärfe: Der Begriff Software-Architektur legt nahe, dass es sich nur um Software-Strukturen handeln muss. Ein IT-System beinhaltet auch die Hardware des Trägersystems, deren Strukturen auch Strukturen des IT-Systems sind. Demnach wäre der Begriff Systemarchitektur als Name für die Strukturen eines IT-Systems angebrachter[9]. Im Abschnitt 3.3 werden daher verschiedene Architekturarten dargestellt, angefangen bei der logischen Architektur bis hin zur Laufzeitarchitektur, einzig die Implementierungsarchitektur stellt ausschließlich Software dar. Diese Architekturarten sind Teil der unten definierten Systemarchitektur. Der Begriff Software-Architektur wird unten auf das Software-System eingeschränkt.

Die Architektur ermöglicht erst die arbeitsteilige Entwicklung eines IT-Systems. Auch die Erfüllung der Qualitätsanforderungen und der wirtschaftlichen Anforderungen wird über die Architektur unterstützt. Dies ist im einführenden Abschnitt bereits angeklungen. Damit werden die Begriffe der Software-Architektur und der Systemarchitektur eines IT-Systems wie folgt definiert:

**Definition 3.1 (Software-Architektur eines IT-Systems)**
Die Software-Architektur ist ein Modell eines Software-Systems und beschreibt dessen grundlegende

[7] Clements et al. verwenden dazu beispielsweise die Elemente von Architekturstilen [CBB+03], siehe dazu auch [SG96].

[8] Der Architekturbegriff wird zusätzlich von den verwendeten Paradigmen geprägt, zurzeit werden verstärkt Dienste anstelle von Komponenten als strukturbildende Elemente von Architekturen verwendet [Erl05].

[9] Diese begriffliche Unschärfe zieht sich durch große Teile der Literatur über Software-Architektur. Hofmeister et al. beschreiben beispielsweise in ihrem Buch *Applied Software Architecture* [HNS99a] vier Architektursichten. Von diesen beschreiben die Conceptual View und die Execution View (siehe Abschnitt 3.4.4) Strukturen des IT-Systems. Nur zwei Sichten beschreiben ausschließlich Software.

Strukturen. Die Strukturen bestehen aus interagierenden Software-Komponenten, sowie deren sichtbaren Eigenschaften und den Beziehungen zwischen den Software-Komponenten. Die Aufteilung in Software-Komponenten ist eine Voraussetzung für die arbeitsteilige Entwicklung der Software. Die Software-Architektur ermöglicht die Erfüllung von Qualitätsanforderungen und der wirtschaftlichen Anforderungen an das IT-System, dessen Bestandteil das Software-System ist.

Die Systemarchitektur umfasst die Software-Architektur und beschreibt zusätzlich die Strukturen des Trägersystems, also Hardware, Netzwerke, Betriebssysteme und weitere beteiligte Laufzeitumgebungen und wie das Software-System auf dem Trägersystem verteilt wird.

**Definition 3.2 (Systemarchitektur eines IT-Systems)**
Die Systemarchitektur ist ein Modell des IT-Systems. Sie beschreibt die grundlegenden Strukturen des IT-Systems. Die Strukturen bestehen aus Elementen wie logischen Komponenten, Software- und Hardware-Komponenten, Prozessen, Bibliotheken oder Quelltexten. Die Beziehungen zwischen den Elementen und deren sichtbare Eigenschaften, zu denen auch ihr Verhalten zählt, sind Bestandteil der Strukturen. Die Systemarchitektur ermöglicht die Erfüllung von Qualitätsanforderungen und der wirtschaftlichen Anforderungen an das IT-System und ist Voraussetzung für dessen arbeitsteilige Entwicklung.

Im IT-System können zwei grobe Bestandteile unterschieden werden, das sind das Software-System und das Trägersystem, welches das Software-System ausführt. Die Systemarchitektur umfasst also die Software-Architektur und die Architektur des Trägersystems und sie beschreibt deren Zusammenspiel. Das Zusammenspiel wird etwa in der Enterprise-JavaBeans-Spezifikation beschrieben [Sun06]. Sie stellt dar, wie Enterprise Beans (Komponenten des Software-Systems) vom Applikationsserver(Trägersystem) ausgeführt werden. Die Enterprise-JavaBeans-Spezifikation ist damit Teil der Systemarchitektur.

Im Folgenden werden allgemeine Elemente der Software- bzw. Systemarchitektur dargestellt, wie sie im Verlauf dieser Arbeit benötigt werden. Komponenten, Konnektoren, Konfigurationen und mehrere zusätzliche Konzepte werden betrachtet.

### 3.2.2 Komponenten

Architektur und Komponenten werden oft in einem Atemzug genannt. Komponenten sind darin ein Mittel zur Strukturierung eines IT-Systems. Ein IT-System besteht aus einem Software-System und einem Trägersystem. Diese können über Komponenten aufgebaut werden: Das Software-System kann über Software-Komponenten und das Trägersystem kann über Software-Komponenten (Laufzeitumgebungen, Betriebssysteme) und Hardware-Komponenten (Rechner, Netzwerke) strukturiert werden.

#### Software-Komponenten

Seit McIlroys Beitrag *Mass-produced software components* [McI68] aus dem Jahr 1968 sind ist Software-Komponenten und Komponenten-basierte Software-Entwicklung eine kaum überschaubare Zahl von Veröffentlichungen erschienen. Viele explizite oder implizite Definitionen des Begriffs Software-Komponente sind darin zu finden. Eine Übersicht findet sich etwa bei Syzperski et al. [SGM02, S. 195ff] oder bei Beneken et al. [BHB+03, S.24ff]. Eine häufig zitierte Definition wurde von Szyperski veröffentlicht und stammt aus einem Workshop am Rande der ECOOP Konferenz 1996 [SP97].

> A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

Die Definition konzentriert sich auf auslieferbare Software-Komponenten (letzthin in einem Binärformat): '*A component is what is actually deployed*' [SGM02]. Diese Sicht stimmt mit vielen technischen Komponentenmodellen wie Enterprise JavaBeans, Servlets oder Komponenten im .NET Umfeld überein, diese erzeugen in der Regel (zu) feinteilige Komponenten [BS02].

Typischerweise wird Wiederverwendbarkeit als wichtige Anforderung an Komponenten genannt [Sam97]. Gemeint ist dabei häufig nur die Wiederverwendung von übersetzter Software, dies klingt am Schluss der Definition an: *subject to composition by third parties*. Häufig wird in diesem Zusammenhang eine Metapher verwendet, die Software-Komponenten mit Lego-Steinen[10]. Eine Software wird demzufolge aus wiederverwendbaren Bausteinen zusammengesetzt.

Software-Komponenten werden jedoch bereits in frühen Phasen der Software-Entwicklung verwendet, um große Spezifikations- und Entwurfsprobleme in kleinere und damit lösbare Probleme zu zerlegen [GJM03, S.67]. Ob genau diese Software-Komponenten unabhängig voneinander ausgeliefert oder gar verkauft und wiederverwendet werden, ist fraglich. Komponenten können aus unterschiedlichen Perspektiven definiert werden. Sie sind Elemente in unterschiedliche Strukturen auf verschiedenen Detaillierungsebenen innerhalb einer Software-Architektur. Sowohl ein logischer Block, oder ein Datenbankmanagementsystem, wie auch eine Kunden-EntityBean werden als *Komponente* bezeichnet. Ein einheitlicher, d.h. vom Kontext unabhängiger, Komponentenbegriff ist schwach und für den Architekturentwurf nicht besonders hilfreich.

In der vorliegenden Arbeit soll zunächst eine allgemeine Definition für Software-Komponenten angegeben werden. Diese wird im Abschnitt 3.3 in mehrere verschiedene Komponentenbegriffe eingereiht. Allgemein anerkannt sind folgende Eigenschaften für Software-Komponenten [SGM02, OMG03, IBM04]:

- Einheit der hierarchischen (De)Komposition von Software-Systemen. Software-Komponenten können aus anderen Software-Komponenten zusammengesetzt sein.
- Logisch abgeschlossen, d.h. sie erbringt eine definierte Aufgabe für ihre Umgebung,
- Hat eine öffentliche Außensicht und geheime Innensicht[11]. Sie hat definierte Schnittstellen für die Funktionalität, die sie anbietet und ebenso definierte Schnittstellen für die Funktionalität, die sie benötigt um ihre Aufgabe zu erfüllen.
- Innerhalb eines Software-Systems austauschbar

Daraus lässt sich eine erste Definition des Begriffs Software-Komponente ableiten:

**Definition 3.3 (Software-Komponente)**
Eine Software-Komponente ist eine Einheit der Komposition von Software-Systemen. Sie bietet ihren Funktionsumfang über Schnittstellen ihrer Umgebung an und importiert Schnittstellen der Umgebung, um diesen Funktionsumfang zu erbringen. Die Umgebung besteht aus anderen Software- und Hardware-Komponenten sowie aus Benutzern. Sie hat höchstens definierte Abhängigkeiten zu ihrer Umgebung. Eine Software-Komponente ist innerhalb eines Software-Systems gegen eine andere Komponente mit denselben Schnittstellen austauschbar.

Weitere Eigenschaften wie Instanziierbarkeit, Identität und Zustand sowie Nebenläufigkeit sind abhängig vom gewählten Systemmodell oder dem verwendeten Trägersystem[12] bzw. von der zugrunde liegenden Architektur- bzw. Komponententheorie sowie von der Anwendungsdomäne. Eigenschaften wie die oft zitierte Wiederverwendbarkeit hängen von der Art und Weise ab, wie die Software-Komponenten entworfen wurden [Ran99].

[10] Eine Diskussion darüber, ob diese Metapher treffend gewählt ist, findet sich etwa bei Ran [Ran99]
[11] Nach dem Prinzip des Information Hiding [Par72]
[12] Laufzeitumgebungen wie Enterprise JavaBeans-Container haben häufig einen eigenen Komponentenbegriff

Weiterhin kann zwischen Komponententypen und Komponenteninstanzen unterschieden werden, analog der Unterscheidung zwischen Klassen und Objekten in der Objektorientierung. Der Komponententyp beschreibt die gemeinsamen Eigenschaften aller Instanzen dieses Typs. In einer Software-Architektur können daher mehrere Instanzen eines Komponententyps verwendet werden.

Der Begriff der Instanz ist problematisch: So könnte etwa ein Komponententyp *Datenverwaltung* definiert sein, damit wäre eine Komponente *Kundenverwaltung* eine Instanz dieses Typs. Die Komponente Kundenverwaltung kann auf mehreren Rechnern innerhalb eines Trägersystems installiert sein, damit wären die Installationen Instanzen der Kundenverwaltung. Zur Laufzeit können in jeder Installationen möglicherweise mehrere Instanzen der installierten Kundenverwaltung als eigene Threads gestartet werden, dies wären dann Instanzen der Instanzen der Instanzen des Komponententyps *Datenverwaltung*. Das Beispiel sollte verdeutlichen, dass sobald von Komponententypen gesprochen wird, ein entsprechender Instanzbegriff definiert werden muss. Im objektorientierten Paradigma ist dieser Begriff klarer.

#### Komponenten und Subsysteme

Analog zum Begriff der Software-Komponente kann ein allgemeiner Komponentenbegriff definiert werden, der auch auf Komponenten des IT-Systems, die aus Hardware bzw. Hardware und Software bestehen, zutrifft:

**Definition 3.4 (Komponente, Subsystem)**
Eine Komponente ist eine Einheit der Komposition von IT-Systemen und deren Bestandteilen (Software-System, Trägersystem). Sie bietet ihren Funktionsumfang über Schnittstellen ihrer Umgebung an und importiert Schnittstellen der Umgebung, um diesen Funktionsumfang zu erbringen. Die Umgebung besteht aus anderen Komponenten und Benutzern. Sie hat nur definierte Abhängigkeiten zu ihrer Umgebung. Eine Komponente ist innerhalb eines IT-Systems gegen eine andere Komponente mit denselben Schnittstellen austauschbar. Eine Komponente kann aus Hardware und/oder Software bestehen. Die Komponenten auf der ersten Dekompositionsebene eines IT-Systems werden auch Subsysteme genannt.

### 3.2.3 Schnittstellen und Konnektoren

Die Begriffe Konnektor, Schnittstelle und verwandte Konzepte wie die Ports hängen vom Komponentenbegriff ab. Allgemein kann über die Beschreibung von Schnittstellen die Außensicht einer Komponente oder eines IT-Systems dargestellt werden. Über Konnektoren werden allgemein die Interaktionsmöglichkeiten (Kommunikationskanäle) zwischen Komponenten beschrieben [Gar03]. Beide Konzepte sprechen die Zusammenarbeit von Komponenten und die Erstellung größerer Strukturen aus Komponenten an.

#### Schnittstellen

Eine Schnittstelle ist allgemein eine gedachte oder tatsächliche Grenze, über die zwei oder mehrere IT-Systeme oder Komponenten bzw. IT-Systeme und Benutzer miteinander kommunizieren[13]. Die Schnittstellen einer Komponente sind die Grenzen zwischen der Komponente und ihrer Umgebung aus anderen Komponenten, IT-Systemen oder Benutzern. Im Folgenden wird unter dem Begriff *Schnittstelle* nur noch die Schnittstelle von Komponenten verstanden. Die Schnittstellen zwischen zwei Komponenten, IT-Systemen oder zwischen den Benutzern und einem IT-System werden unter dem Begriff *Konnektor* betrachtet.

[13] *An interface is a boundary across which two independent entities meet and interact or communicate with each other*[CBB+03, S.223]

**Definition 3.5 (Schnittstellen einer Komponente)**
Die Schnittstelle einer Komponente ist eine gedachte oder tatsächliche Grenze, über die eine Komponente mit ihrer Umgebung kommuniziert. Eine Schnittstelle enthält Operationen, welche die Komponente für die Umgebung bereitstellt (exportiert) oder von der Umgebung verwendet (importiert). Eine Komponente kann mehrere Schnittstellen haben, die jeweils exportierte oder importierte Operationen zusammenfassen. Die Schnittstellen definieren das Verhalten der Komponente.

Die Beschreibung einer Schnittstelle definiert die Operationen der Schnittstelle und damit das Verhalten der Komponente[14], welche die Schnittstelle exportiert. Beschreibungen von Schnittstellen sind eigenständige Elemente des Entwurfs und der Implementierung in Software und/oder Hardware. Sie können von mehreren Komponenten als importiert oder exportiert zugeordnet werden.

Es wird zwischen Anbietern und Nutzern einer Schnittstelle unterschieden. Der Anbieter implementiert die Operationen der Schnittstelle und der Nutzer verwendet die Operationen. Die Beschreibung einer Schnittstelle definiert damit, welche Operationen der Anbieter bereitstellen muss bzw. welche Operationen der Nutzer verwenden darf. In diesem Zusammenhang wird auch von einem Vertrag zwischen Anbieter und Nutzer gesprochen, den die Schnittstellenbeschreibung definiert.

Technisch muss eine Schnittstellenbeschreibung etwa in einem Java-Quelltext nicht zwingend als Java-Interface programmiert werden: Sie kann aus mehreren Java-Interfaces und auch Java-Klassen bestehen, dabei modellieren die Java-Klassen Parameter für Operationen der Interfaces (Transportstrukturen). Ebenso kann eine Schnittstelle über Nachrichten definiert sein, indem die Inhalte und die Struktur der ausgetauschten Nachrichten definiert werden [KB06]. Eine Nachricht kann einen Operationsaufruf, ein Ereignis oder einen Datensatz (ein Dokument) darstellen [HW03].

#### Ports

Das Konzept der Ports wird verwendet, um die Interaktionspunkte einer Komponente mit ihrer Umgebung zu definieren [OMG03, S. 167f]: Komponenten kommunizieren über Ports mit ihrer Umgebung. Über Ports werden Nachrichten gesendet und empfangen, bzw. Operationen auf der Komponente werden aufgerufen oder die Komponente ruft Operationen aus ihrer Umgebung auf. In der UML 2.0 fassen Ports benötigte und exportierte Schnittstellen einer Komponente zusammen [OMG03, S. 167f].

**Definition 3.6 (Port)**
Ein Port definiert einen Interaktionspunkt einer Komponente mit ihrer Umgebung. Der Port kann sowohl Nachrichten empfangen als auch senden bzw. er kann Operationen für andere Komponenten bereitstellen oder Operationen anderer Komponenten nutzen. Einem Port können daher angebotene (exportierte) und benötigte (importierte) Schnittstellen einer Komponente zugeordnet werden. Eine Komponente kann mehrere Ports besitzen.

#### Konnektoren

Ein Konnektor repräsentiert einen Kommunikationskanal zwischen zwei oder mehreren Komponenten [Gar03, S.5f]. Über Konnektoren werden Nachrichten ausgetauscht bzw. Operationen aufgerufen. Konnektoren können auf einen Nachrichtentyp eingeschränkt sein, etwa auf Nachrichten, die auf Änderungen an Kundendaten hinweisen.

**Definition 3.7 (Konnektor)**
Ein Konnektor repräsentiert einen Kommunikationskanal, über den zwei oder mehr Komponenten interagieren können. Ein Konnektor kann zwei Komponenten verbinden, (1.) direkt oder (2.) über die

[14] Im Bereich betrieblicher Informationssysteme werden bislang nur Teile des Verhaltens einer Komponente über die Schnittstellenbeschreibung festgelegt, etwa mithilfe von Vor- und Nachbedingungen [SGM02, S.55ff]. Eigenschaften wie Antwortzeiten oder Ressourcenverbrauch werden in der Regel nicht beschrieben.

Verbindung einer exportierten Schnittstelle mit einer importierten Schnittstelle oder (3.) über zwei verbundene Ports. Konnektoren beschreiben die Protokolle[15] zur Interaktion von Komponenten, etwa zur transaktionalen oder sicheren Kommunikation. Sie legen damit einen Teil des Verhaltens des aus den verbundenen Komponenten bestehenden Teilsystems fest.

Konnektoren können etwa als einfache Prozedur- oder Methodenaufrufe umgesetzt werden, also in Quelltexten keine explizite Entsprechung haben. Sie können implementiert werden als gemeinsame Variable, die sich zwei Prozesse teilen, als Tabellen in einer Datenbank, oder als Warteschlange, über die zwei Komponenten Nachrichten austauschen. Die Tabellen oder Warteschlange sind dann softwaretechnische Entsprechungen eines Konnektors innerhalb einer Systemarchitektur.

### 3.2.4 Konfigurationen

Der Begriff Konfiguration bezeichnet eine Struktur, die aus Komponenten besteht, welche über Konnektoren verbunden sind. Eine Konfiguration kann ein IT-System, sein Software-System oder sein Trägersystem strukturieren. Der Begriff der Konfiguration ist also abhängig vom Komponentenbegriff.

Eine Konfiguration kann als gerichteter Graph verstanden werden, dessen Knoten Komponenten und dessen Kanten Konnektoren sind [MT00]. Eine Konfiguration entspricht den typischen Architekturdarstellungen als Box-and-Arrow Diagramme. Neben den Kommunikationsbeziehungen zwischen Komponenten wird auch der hierarchische Aufbau der Komponenten aus ihren Teilen in einer Konfiguration modelliert, beispielsweise als Komponentenbaum.

Wird bei Komponenten beispielsweise zwischen Komponententypen und Komponenteninstanzen unterschieden, kann die Konfiguration einerseits alle erlaubten Kommunikationsbeziehungen zwischen Komponententypen darstellen oder andererseits ein Beispiel für konkrete Interaktionsbeziehungen zur Laufzeit geben. Weiterhin müssen statische Konfigurationen, deren Struktur vor der Laufzeit festgelegt wird, von dynamischen Konfigurationen, deren Struktur sich während der Laufzeit ändern kann, unterschieden werden.

**Definition 3.8 (Konfiguration)**
Eine Konfiguration ist im Allgemeinen eine Struktur, die aus Komponenten und Konnektoren besteht. Die Struktur kann als gerichteter Graph interpretiert werden, in dem die Komponenten die Knoten und die Konnektoren die Kanten sind.

Die oben eingeführten Schnittstellen und Ports sind je nach Architektur- und Komponentenbegriff ebenfalls Bestandteile der Konfiguration.

### Architekturstile und Referenzarchitekturen

Architekturstile und Referenzarchitekturen sind Vorlagen für Architekturen. Sie definieren Typen von Komponenten und Konnektoren (und damit ein Entwurfsvokabular) sowie Regeln, wie diese zu Konfigurationen zusammengesetzt werden dürfen. Shaw und Garlan definieren Architekturstil analog [SG96]:

> An architectural style, ... defines a family of ... systems in terms of a pattern of structural organization. More specifically, an architectural style defines a vocabulary of components and connector types, and a set of constraints on how they can be combined.

[15] Etwa: Auf jede Nachricht wird mit einer Bestätigung geantwortet und vor der Kommunikation werden zunächst Schlüssel zur Verschlüsselung ausgetauscht.

Im Folgenden wird der Begriff der Referenzarchitektur weiter verwendet. Eine Referenzarchitektur ähnelt in ihren Aufgaben und Zielen einem Architekturstil. Sie ist jedoch detaillierter ausgearbeitet und auf eine Anwendungsdomäne spezialisiert. Die nachfolgende Definition ist eine Präzisierung des von Beneken in [Ben06, S.358] definierten Begriffs:

**Definition 3.9 (Referenzarchitektur)**
Eine Referenzarchitektur ist eine abstrakte Architektur für alle IT-Systeme eines bestimmten Anwendungsbereichs, etwa für alle betrieblichen Informationssysteme. Sie definiert Strukturen und Typen von Komponenten sowie deren erlaubte Interaktionen und Verantwortlichkeiten.

Eine Referenzarchitektur definiert zusätzlich einen Begriffsapparat, der die Sprache der an der Architektur interessierten Stakeholder vereinheitlicht und Missverständnisse verringert. Der Begriffsapparat umfasst beispielsweise Typen von Komponenten und grundlegende Entwurfsprinzipien. Zusätzlich ist eine Referenzarchitektur eine wiederverwendbare System-, Trägersystem oder Software-Architektur. Sie kann daher den Architekturentwurf beschleunigen und seine Qualität verbessern. Beispiele für Referenzarchitekturen sind in Quasar [Sie04] definiert. Java EE und Microsoft .NET definieren technische Referenzarchitekturen. Eine ausführliche Betrachtung und Einordnung von Referenzarchitekturen findet sich in [Ben06, S.358ff].

## Zusätzliche Konzepte

### Szenarios

Die Konfiguration definiert über die Konnektoren, welche Komponenten grundsätzlich miteinander sprechen können. Szenarios sind konkrete Beispiele für solche Interaktionen. Sie stellen dar, wie ein Anwendungsfall (Use Case) als Interaktion eines Benutzers oder eines Nachbarsystems mit den Komponenten und Konnektoren des betrachteten IT-Systems umgesetzt wird. Zur Beschreibung eines Szenarios gehört daher die Liste der an der Ausführung des Szenarios beteiligten Komponenten und Konnektoren[16].

**Definition 3.10 (Szenario)**
Ein Szenario beschreibt, welche Komponenten und Konnektoren an der Interaktion eines Benutzers oder Nachbarsystems mit dem IT-System im Rahmen eines Anwendungsfalls beteiligt sind.

### Attribute (Properties)

Komponenten, Konnektoren und Konfigurationen haben Eigenschaften (Attribute), die ihnen unabhängig von der Verhaltensbeschreibung zugeordnet werden können [Gar03, S.6], [CBB+03, S.111]. Das können etwa nichtfunktionale Eigenschaften wie Antwortzeit, Ausfallwahrscheinlichkeit oder Ressourcenbedarf, sowie Planungsinformationen, wie Fertigstellungstermin oder Kosten sein [17].

**Definition 3.11 (Attribut)**
Ein Attribut ist eine Eigenschaft, die einer Komponente, einem Konnektor oder einer Konfiguration zugeordnet werden kann. Ein Attribut hat einen Namen sowie einen Typ (etwa Datum, Wahrscheinlichkeit, Kosten oder String). Für jedes Architekturelement hat das Attribut einen konkreten Wert (eine Belegung) aus dem Wertebereich des Typs.

[16] Buhr und Casselmann dokumentieren mit ihren Use Case Maps in [Buh98] die hier definierten Szenarios.
[17] Overhage [Ove04] schlägt beispielsweise einen strukturierten Spezifikationsrahmen für Komponenten vor, der diese und weitere Attribute berücksichtigt.

### 3.2.5 Zusammenfassung der Begriffe

Die Abbildung 3.1 zeigt die Begriffe im Umfeld der Systemarchitektur. Dabei wird ein IT-System durch seine Systemarchitektur beschrieben. Das IT-System besteht aus einem Software-System und einem Trägersystem, welches das Software-System ausführt. Das Software-System wird durch eine Software-Architektur beschrieben und das Trägersystem durch eine Trägersystemarchitektur.

Die Systemarchitektur ist ein Modell des IT-Systems, die Software-Architektur ein Modell des Software-Systems und die Trägersystemarchitektur ist ein Modell des Trägersystems. Diese Sachverhalte wurden in Abbildung 3.1 zur Erhaltung der Übersichtlichkeit weggelassen. Ein Modell kann eine (ggf. unendliche oder leere) Menge von Originalen beschreiben, umgekehrt kann es zu jedem Original eine (ggf. unendliche) Menge von Modellen geben.

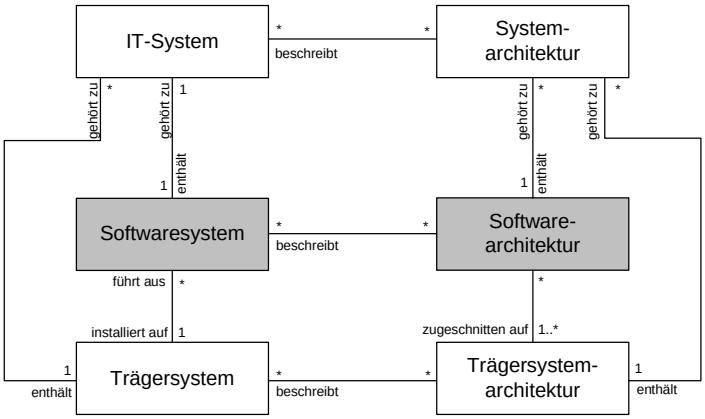


Abbildung 3.1: Begriffe der Systemarchitektur

Die Begriffe der Systemarchitektur werden in Abbildung 3.2 dargestellt. Die Relationen zwischen den Begriffen Komponente, Schnittstelle, Konnektor und Port unterscheiden sich von Architekturbeschreibungssprache zu Architekturbeschreibungssprache [MT00]. Die hier dargestellten Relationen der Begriffe stellen das im Folgenden verwendete Verständnis dar.

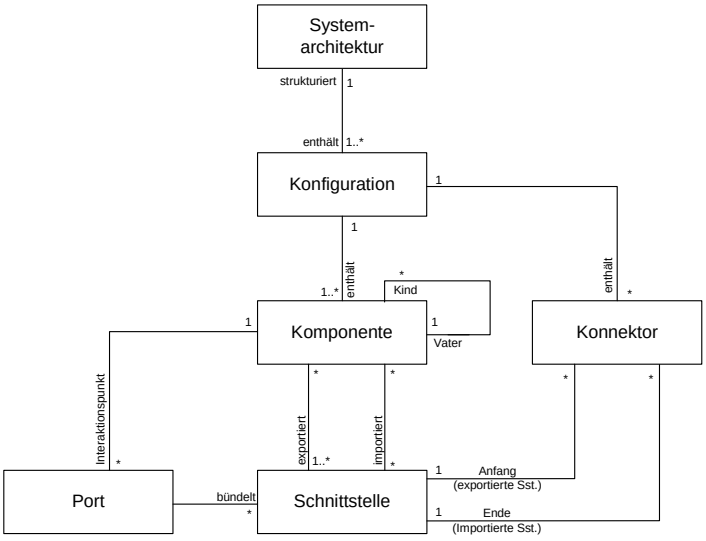


Abbildung 3.2: Begriffe der Software-Architektur

Eine Systemarchitektur wird durch eine Konfiguration strukturiert. Diese besteht aus Komponenten und Konnektoren. Eine Komponente kann aus beliebig vielen anderen Komponenten bestehen, sie

exportiert mindestens eine Schnittstelle und kann beliebig viele Schnittstellen importieren. Ports fassen die Schnittstellen zusammen und definieren so Interaktionspunkte einer Komponente mit ihrer Umgebung. Konnektoren verbinden die exportierten Schnittstellen von den Komponenten, die einen Funktionsumfang anbieten, mit den importierten Schnittstellen anderer Komponenten.

## 3.3 Arten der Architektur

### 3.3.1 Produktmodelle im Software-Entwicklungsprozess

Ein betriebliches Informationssystem wird als Netz unterschiedlicher Beschreibungen erstellt. Es hat unterschiedliche Darstellungen während der Entwicklung, die parallel oder aufeinander aufbauend erstellt werden. Unabhängig vom Vorgehensmodell, der Entwicklungsmethode und dem zugrunde liegenden Trägersystem gibt es mindestens folgende Darstellungen:

- Anforderungen (Lastenheft und Pflichtenheft, Sammlung von Anwendungsfällen etc.)
- Quelltexte des Software-Systems in einer oder mehreren Skript- und Programmiersprachen bilden das Programm, also die formale Verhaltensbeschreibung des Software-Systems.
- Binärdateien (z.B. JAR, Assembly, DLL, EXE) sind das übersetze und teilweise gebundene Programm. Die Binärdateien können direkt vom Trägersystem ausgeführt werden.
- Prozesse und Threads des IT-Systems zur Laufzeit im Hauptspeicher eines oder mehrerer Rechner (Thread, Prozess)

Mehrere Vorgehensmodelle und Entwurfsmethoden ergänzen nach der Erhebung der Anforderungen einen Architekturentwurf[18], welcher das IT-System in Subsysteme und Komponenten strukturiert, die zugekauft, wiederverwendet oder von Teilteams erstellt werden können.

Die Abbildung 3.3 gibt ein Beispiel für die Entwicklung eines betrieblichen Informationssystems über ein Netz verschiedener Darstellungen, angefangen bei ersten Anforderungen, bis hin zum laufenden IT-System, das über eine Hardware-Netzwerk-Topologie verteilt ist. Die Beschreibungen verwenden verschiedene Sprachen und Strukturierungsmittel[19], da sie auf verschiedene Stakeholder und Problemfelder ausgerichtet sind. In den frühen Phasen ist dies oft die natürliche Sprache, zusammen mit einigen Schaubildern und Tabellen, in späteren Phasen ist die Beschreibung formaler, etwa in einer Programmiersprache und in Byte- bzw. Maschinencode.

Im Allgemeinen sind die Strukturen der Darstellungen unabhängig von einander. Ein Architekt kann zwar willkürlich entscheiden, dass eine Menge von Anwendungsfällen als logische Komponente[20] dargestellt und diese nur in einem Code-Namensraum implementiert und so als Assembly ausgeliefert wird. Dies ist jedoch nicht immer so und häufig kaum möglich: Bestimmte Strukturen lösen sich in anderen auf (Feature Delocation [GSCK04, S.38ff]). Wird beispielsweise in der Spezifikation das fachliche Konzept *Kunde* definiert, findet sich dieses in der Implementierung vermischt mit technischen Konzepten in verschiedenen Komponenten der grafischen Oberfläche, des Anwendungskerns und der Datenbank wieder. Die Darstellungen haben verschiedene Entwurfstreiber: während die Anforderungen aus Nutzersicht logisch strukturiert werden, werden die Quelltexte so strukturiert, damit die physischen Abhängigkeiten kontrollierbar bleiben [Lak96, S.12f]. Strukturierungskriterium für die Bibliotheken sind dagegen Fragen der Verteilung (des Deployments) und der Einhaltung von Qualitätsmerkmalen, wie Verfügbarkeit und Durchsatz [Bri00]. Die Elemente der verschiedenen Darstellungen stehen auf Aufgrund verschiedener Entwurfsanforderungen und der Feature Delocation im Allgemeinen in n:m Beziehungen oder sie sind unabhängig.

[18] der Entwurf wird in einem DV-Konzept [Sch97], IT-Konzept [Sie02b] oder Archtectural Document [JRB99] beschrieben
[19] Parnas formuliert eine ähnliche Beobachtung [Par74], ebenso wie Ran [Ran99]
[20] vgl. Abschnitt 3.3.2

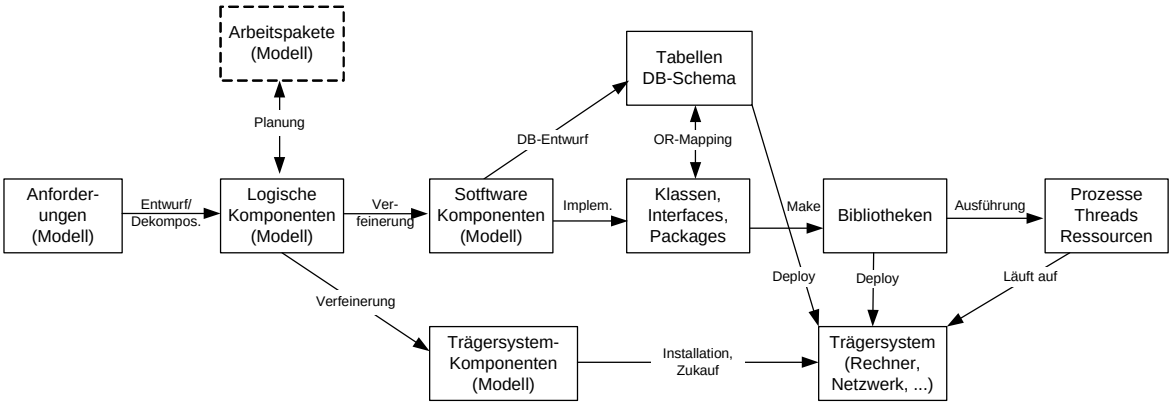


Abbildung 3.3: Zwischenergebnisse im Entwicklungsprozess

Die Abbildung 3.3 stellt die sequentielle Erstellung der verschiedenen Darstellungen dar, angefangen bei den Anforderungen bis hin zum laufenden System. Nur wenn Anforderungen definiert sind, können offenbar Quelltexte entstehen, welche diese erfüllen. Bibliotheken können offenbar erst dann erstellt werden, wenn die Quelltexte in übersetzter Form vorliegen.

Modelle der Quelltexte, der Verteilung oder der Prozesse und Threads entstehen jedoch parallel und beeinflussen sich gegenseitig: Die Hardware-Netzwerk-Topologie und die Grundzüge der Verteilung stehen beispielsweise schon früh während der Anforderungsanalyse fest, da das System häufig auf vorhandener Hardware laufen muss oder auf der Grundlage eines Architekturmusters wie der Client/Server-Architektur gebaut wird. Die Verteilung beeinflusst den Inhalt der Quelltexte, da diese die Verteilungsgrenzen implementieren müssen, in der Programmiersprache Java etwa über RMI-Schnittstellen in den Quelltexten. Abhängig von der Infrastruktur und der Programmiersprache muss auch Nebenläufigkeit in den Quelltexten explizit vorgesehen werden.

Die These, dass ein System mit einem abstrakten Modell beschrieben werden könnte, das dann sequentiell bis hin zur Verteilung verfeinert wird, trifft also in der Regel nicht zu. IT-Systeme und ihre Software werden über ein Netz von wechselseitig abhängigen Modellen erstellt.

In den folgenden Abschnitten werden daher mehrere verschiedene Arten von Architektur eingeführt. Im Einzelnen sind dies logische, technische, Implementierungs- sowie Verteilungs- und Laufzeitarchitektur. Wichtigstes Kriterium für eine Unterscheidung der Architekturarten ist der Gegenstand der Strukturierung und die damit zusammenhängenden Entwurfsziele und Architekturtreiber:

**Logische Architektur:** Die logische Architektur strukturiert das IT-System logisch in Komponenten und teilt den Funktionsumfang des IT-Systems auf diese auf. Sie lässt offen, wie/ob die Komponenten in Software oder Hardware implementiert werden. Sie wird bei betrieblichen Informationssystemen insbesondere für Grobentwürfe zu Projektbeginn verwendet, als Grundlage für eine Aufwandsschätzung, Projektorganisation und Aufgabenverteilung.

**Technische Architektur:** Die technische Architektur strukturiert das IT-System technisch in implementierbare Hardware- und Software-Komponenten. Bei betrieblichen Informationssystemen werden überwiegend Software-Komponenten entworfen und deren Trägersystem wird aus zugekauften Software- und Hardware-Komponenten zusammengestellt.

**Implementierungsarchitektur:** Die Implementierungsarchitektur strukturiert die Quelltextdateien der Software-Komponenten physisch in Verzeichnisse (Pakete, Namensräume) und definiert Einheiten des Konfigurationsmanagements. Entwurfsziel bei der Implementierungsarchitektur ist insbesondere die Änderbarkeit der Quelltexte und damit des Software-Systems.

**Verteilungsarchitektur:** Die Verteilungsarchitektur definiert, wie Bibliotheken (die Compilate der

Quelltexte die einen Teil einer oder auch mehrere Software-Komponenten implementieren) auf den verschiedenen Laufzeitumgebungen des Trägersystems installiert werden. Entwurfsziele sind unter anderem Fragestellungen des Systembetriebs wie passende Verfügbarkeit und passender (Transaktions-)Durchsatz.

**Laufzeitarchitektur:** Die Laufzeitarchitektur strukturiert das IT-System zur Laufzeit in Prozesse und Threads, die in den Laufzeitumgebungen des Trägersystems ausgeführt werden. Entwurfsziel ist insbesondere der effektive Umgang mit Ressourcen der Laufzeitumgebung (Hauptspeicher, Datenbankverbindungen, etc.) und Vermeidung bzw. Kontrolle der Probleme der Nebenläufigkeit wie etwa Verklemmungen oder das Verhungern bestimmter Threads.

Der jeweilige Komponentenbegriff ist abhängig von der Architekturart. Daher ist die nun folgende Unterteilung auch ein Vorschlag für exaktere Komponentenbegriffe im Bereich betrieblicher Informationssysteme. Mehrere Sichtenkonzepte schlagen eine vergleichbare Aufteilung in verschiedene Architekturen vor[21].

### 3.3.2 Logische Architektur

Fragestellungen der Software- und Systemarchitektur stellen sich beim Bau von IT-Systemen bereits in frühen Phasen des Entwicklungsprozesses: Zum Projektbeginn wird im Rahmen der Planungen ein erster Grobentwurf (Entwurfsskizze) erstellt. *Dieser Grobentwurf repräsentiert die erste Dekomposition des Systems in Subsysteme* [BD04, S. 612]. Der Grobentwurf konzentriert sich auf die Funktionalität der identifizierten Subsysteme, er teilt den Funktionsumfang des IT-Systems auf diese auf.

Wirtschaftliche Anforderungen sind wichtige Architekturtreiber des Grobentwurfs, da der Grobentwurf Entwicklungskosten, -zeit und -risiken wesentlich beeinflusst: Zugekaufte Subsysteme können etwa die Entwicklungszeit verkürzen oder über Wiederverwendung können Kosten eingespart werden.

Der Grobentwurf ist ein wichtiges Instrument des Projektmanagements, denn er kann im Rahmen einer Expertenklausur für die Schätzung der Erstellungsaufwände verwendet werden (s.u.) [Sie02b, Bur02]. Der Grobentwurf ist ebenso die Grundlage für die Organisation und die Planung des Projekts, denn die identifizierten Subsysteme können als Arbeitspakete an ein Teilteam vergeben werden (vgl. z.B. [HG99b, BD04]). Der Grobentwurf ist also die Grundlage für die arbeitsteilige Spezifikation und Implementierung des IT-Systems in Software und Hardware.

Der Grobentwurf eines IT-Systems ist häufig eine logische Architektur: Eine logische Architektur beschreibt die logische Strukturierung des IT-Systems in logische Komponenten unabhängig davon, mit welchen Mitteln die logischen Komponenten umgesetzt werden. Diese Eigenschaft ist für Grobentwürfe wichtig, da von dem logischen Entwurf ausgehend, verschiedene Umsetzungsalternativen durchgespielt werden müssen.

**Definition 3.12 (Logische Architektur)**
Die logische Architektur ist Teil der Systemarchitektur eines IT-Systems. Sie beschreibt die Dekomposition des IT-Systems in logische Komponenten, dabei ist dessen Funktionsumfang auf logische Komponenten aufgeteilt. Die logischen Komponenten tauschen über logische Konnektoren Nachrichten aus. Die logische Architektur macht keine Aussage darüber, auf welche Weise die logischen Komponenten und Konnektoren in Hardware oder Software implementiert werden. Die logische Architektur ist die notwendige Grundlage für die Projektplanung und die Implementierung.

Eine logische Komponente erbringt jeweils einen Teil des Funktionsumfangs des IT-Systems. Auf der Grundlage der logischen Architektur kann dann entschieden werden, wie dessen Bestandteile umgesetzt werden: Bei betrieblichen Informationssystemen werden logische Komponenten in der Regel als Software-Komponenten implementiert, wiederverwendet oder als COTS[22]-Komponenten (bzw.

[21] vgl. Abschnitt 3.4
[22] Commercial off the Shelf

COTS-Systeme) zugekauft. Eine logische Komponente kann jedoch auch als spezielle Hardware-Komponente umgesetzt werden: etwa Barcode-Leser in einem Logistiksystem oder Flugticketdrucker bei einem Abfertigungssystem.

Darstellungen der logischen Architektur sind in der Praxis typischerweise einfache Box-And-Arrow-Diagramme, die einen Überblick über die Grobarchitektur geben [Kel03, S. 86]: Die Abbildung 3.4 zeigt als Beispiel die logische Architektur eines Bringdienstsystems. Das System ist aus den drei Komponenten *Bestellverwaltung*, *Kundenverwaltung* und *Artikelverwaltung* aufgebaut. Zusätzlich sind der *Rechnungsdrucker* und das *Buchhaltungssystem* als Nachbarsysteme und die Nutzergruppen *Kunden im Internet* und *Mitarbeiter in der Filiale* dargestellt. Die *Bestellverwaltung* kann beispielsweise der *Kundenverwaltung* über einen Konnektor eine Nachricht schicken, mit der sie Kundendaten abfragt. Die Kundenverwaltung kann über einen Konnektor in Gegenrichtung eine Nachricht mit einem Datensatz für einen Kunden antworten.

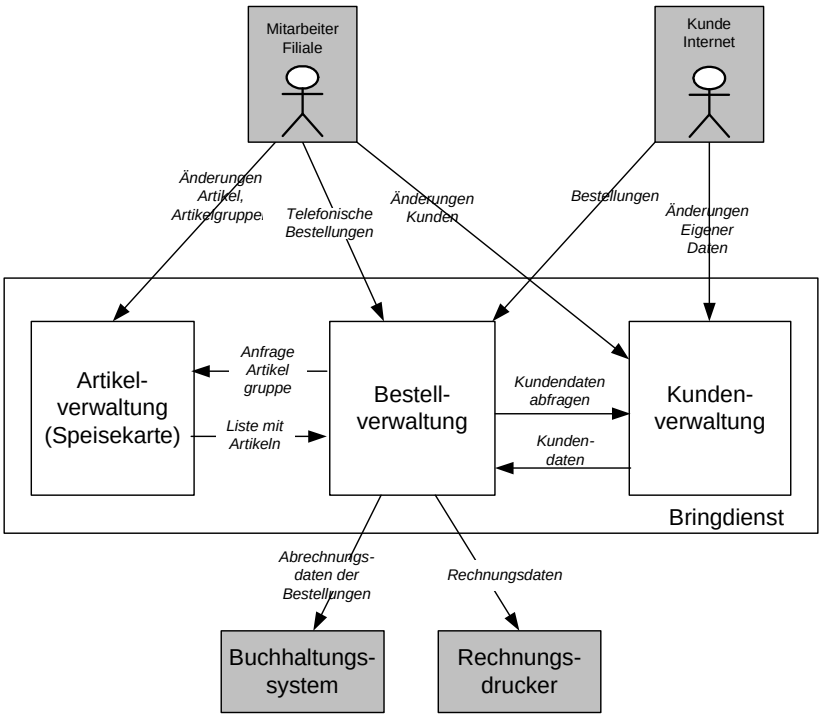


Abbildung 3.4: Logische Architektur des Bringdienstsystems

Fertigstellungstermin und Entwicklungskosten sind bei betrieblichen Informationssystemen zwei zentrale Architekturtreiber. Diese werden durch die logische Architektur und die darauf aufbauende Projektplanung beeinflusst: Kosten können beispielsweise dadurch gesenkt werden, dass die Realisierung einiger logischer Komponenten an Partner mit geringeren Kosten pro Entwicklerstunde weitergegeben werden [Tau04]. Ebenso kann über die Wiederverwendung vorhandener Software Budget gespart werden. Die Entwicklungszeit kann durch Zukauf von fertigen Fremdkomponenten verkürzt werden. Aufgaben beim Entwurf der logischen Architektur sind daher unter anderem:

- Reduktion der Komplexität: Logische Aufteilung von IT-Systemen in kleinere beherrschbare Einheiten
- Aufteilung der feineren Spezifikations-, Entwurfs- und Implementierungsarbeit auf ggf. fachlich oder technisch spezialisierte Teilteams. Die logische Architektur hat damit eine enge Verbindung zum Projektstrukturplan (Work Breakdown Structure) [Bur02, S.76ff]. Für logische Komponenten können beispielsweise ein oder mehrere Arbeitspakete definiert werden. Denn die Arbeitsaufteilung geschieht typischerweise entlang der Komponentengrenzen.

- Bereitstellung der Grundlage für die Aufwandsschätzung für die Spezifikation, den Feinentwurf und die Implementierung des Softwareanteils. Der Aufwand der zu den logischen Komponenten definierten Arbeitspakete kann im Rahmen einer Expertenschätzung geschätzt werden.
- Schaffung der Grundlage für die Planung der Auslieferung der Komponenten des IT-Systems in Iterationen bzw. Stufen. Auch eine Kosten/Nutzen/Risiko Diskussion kann geführt werden, welche den Nutzen der Funktionalität einzelner logischer Komponenten den Kosten und Risiken für die Entwicklung gegenüberstellt.
- Entscheidung über den Zukauf von fremden Software- und/oder Hardwarekomponenten, um beispielsweise Entwicklungszeit oder -kosten einzusparen,
- Entscheidung über die Fremdvergabe von Einheiten an externe Partner (Lieferanten), um beispielsweise Wissen und Arbeitskräfte einzukaufen oder um Personalkosten zu einzusparen.

Logische Komponenten sind die Einheiten der Dekomposition in logischen Architekturen. Sie sind die Elemente, aus denen das IT-System später zusammengesetzt wird. In Anlehnung an die oben genannten Aufgaben und die Definition 3.12 wird der Begriff *logische Komponente* definiert:

**Definition 3.13 (Logische Komponente)**
Eine logische Komponente ist eine Komponente. Sie ist eine Einheit des Grobentwurfs und damit auch der Projektplanung. Sie ist Strukturierungsmittel einer logischen Architektur. Dort hat sie eine definierte Aufgabe, diese ist ein Teil der Funktionalität des IT-Systems. Eine logische Komponente hat Kommunikationsbeziehungen zu anderen logischen Komponenten, mit diesen tauscht sie Nachrichten über logische Konnektoren aus. Die Inhalte der Nachrichten werden in der Regel informell spezifiziert. Das Verhalten (die Funktionalität) und Zustand einer logischen Komponente wird bei betrieblichen Informationssystemen typischerweise informell mit natürlicher Sprache beschrieben. Ihre Schnittstellen sind noch indirekt über logische Konnektoren angegeben.

Dieser Komponentenbegriff entspricht weitgehend dem von Siedersleben definierten Komponenten-Begriff [Sie02a, S. 4]: *Eine Komponente ist eine sinnvolle Einheit des Entwurfs, der Implementierung und damit der Planung*.

Schnittstellen werden in logischen Architekturen in der Regel nicht im Detail festgelegt. Dies ist erst möglich, wenn entschieden wurde, wie die logischen Komponenten umgesetzt werden: Wird etwa eine Fremdkomponente integriert, steht ihre Schnittstelle bereits fest und kann vom Architekten nicht beeinflusst werden. Außerdem muss für die Kommunikation zwischen zwei Komponenten die Entscheidung getroffen werden, ob der Nachrichtenaustausch synchron nach dem Request/Reply-Muster erfolgt oder asynchron, etwa über Warteschlangen. Weiterhin kann eine Schnittstelle Methoden oder Funktionen anbieten, aber auch Nachrichten, Ereignisse oder Dokumente verarbeiten [HW03].

Die Kommunikationsbeziehungen werden in logischen Architekturen nur abstrakt über logische Konnektoren beschrieben, diese sind Transportkanäle für die Daten, welche die Komponenten austauschen. Daten werden in Form von Nachrichten ausgetauscht. Die Inhalte der Nachrichten (Operationsaufruf/Rückgabewert, Ereignisse, Dokumente) und die Art des Austausches (synchron, asynchron) bleiben offen.

**Definition 3.14 (Logischer Konnektor)**
Ein logischer Konnektor ist ein Konnektor. Er ist eine Einheit des Grobentwurfs und damit auch der Projektplanung. Er beschreibt den Transportkanal für Nachrichten zwischen je zwei logischen Komponenten. Mehrere logische Konnektoren können dieselben zwei Komponenten verbinden.

Erst in der technischen Architektur werden die Schnittstellen soweit verfeinert, dass sie implementierbar sind. Denn die technische Architektur definiert, wie die logischen Komponenten implementiert, wiederverwendet oder zugekauft werden und trifft die grundlegenden Entscheidungen über

die Gestalt der Schnittstellen. In der technischen Architektur kann ein Konnektor beispielsweise (1) eine Warteschlange sein, über die asynchron Nachrichten versendet werden, (2) ein Paar mit exportierter und importierter Schnittstelle oder auch (3) eine Datenbank-Tabelle oder ein Shared Memory-Bereich über die zwei Komponenten Daten austauschen.

Die Verteilungsarchitektur (siehe unten) entsteht typischerweise parallel zur logischen Architektur und beeinflusst diese. Vorhandene Verteilungsgrenzen sind ein Partitionierungskriterium. Eine typische Grenze zwischen Komponenten ist beispielsweise der Client/Server-Schnitt, diese Grenze findet sich auch in der Arbeitsaufteilung wieder, weil etwa ein Client- und ein Server-Team gebildet werden.

### 3.3.3 Technische Architektur

In der technischen Architektur werden Entscheidungen über die Implementierung der logischen Architektur mit Software-Komponenten und einem Trägersystem getroffen. Das Software-System wird aufgeteilt in Software-Komponenten, die dann in einer Programmiersprache für eine bestimmte Laufzeitumgebung, etwa Enterprise JavaBeans oder Microsoft .NET, implementiert werden. Das Trägersystem wird festgelegt und aus Hardware-Komponenten (Rechner, Netzwerk, Peripheriegeräte) sowie Software-Komponenten (Betriebssystemen, Laufzeitumgebungen) zusammengestellt. Zu einer logischen Architektur sind daher mehrere technische Architekturen denkbar, d.h. verschiedene Implementierungen mit unterschiedlichen Trägersystemen. Aufgaben bei der Erstellung einer technischen Architektur sind unter anderem:

- Technische Umsetzung der logischen Komponenten in Software- und ggf. zusätzlicher Hardware-Komponenten sowie mithilfe der Komponenten des Trägersystems. Damit werden auch erste Entscheidungen über Qualitätsmerkmale des IT-Systems wie Antwortzeiten, Durchsatz oder Sicherheit getroffen.
- Spezifikation der Schnittstellen zwischen den verschiedenen Software- und Hardware-Komponenten
- Technische Integration von Fremdkomponenten und Nachbarsystemen.

Eine Software-Komponente innerhalb der technischen Architektur wird abhängig von der Ausprägung auch Modul oder Schicht[23] genannt. Sie ist typischerweise die Verfeinerung einer logischen Komponente, eine Einheit des Modultests und auch Arbeitsergebnis eines Teilteams. Der Begriff der Software-Komponente ist in Definition 3.3 ausgeführt. Die Bestandteile des Trägersystems werden nicht im Detail betrachtet.

Die Software-Komponenten und die Struktur des Trägersystems werden in der technischen Architektur in der Regel separat dargestellt. Häufig wird das Trägersystem erst in der Verteilungsarchitektur explizit modelliert. Ein Überblick über die Software-Komponenten der technischen Architektur kann in Form eines UML-Komponentendiagramms geschehen. Zusätzlich sind Schnittstellenspezifikationen erforderlich, etwa unter Verwendung der Syntax der UML oder einer Programmiersprache [CD01, Sie04].

Die Abbildung 3.5 zeigt ein Beispiel für die Beschreibung der technischen Architektur des Bringdienstsystems mit Hilfe eines UML 2.0 Komponentendiagramms. Das Diagramm stellt die Schnittstellen der jeweiligen Komponenten dar. Die Schnittstellen sind teilweise über Ports gebündelt. Die genaue Spezifikation der Schnittstellen ist in dem vorliegenden Komponentendiagramm nicht angegeben.

Technische Architektur und Verteilungsarchitektur hängen eng zusammen: Übersetzte Quelltexte der Software-Komponenten werden in Bibliotheken, etwa JARs oder Assemblies zusammengefasst.

[23] vgl. z.B. [BMR+00]

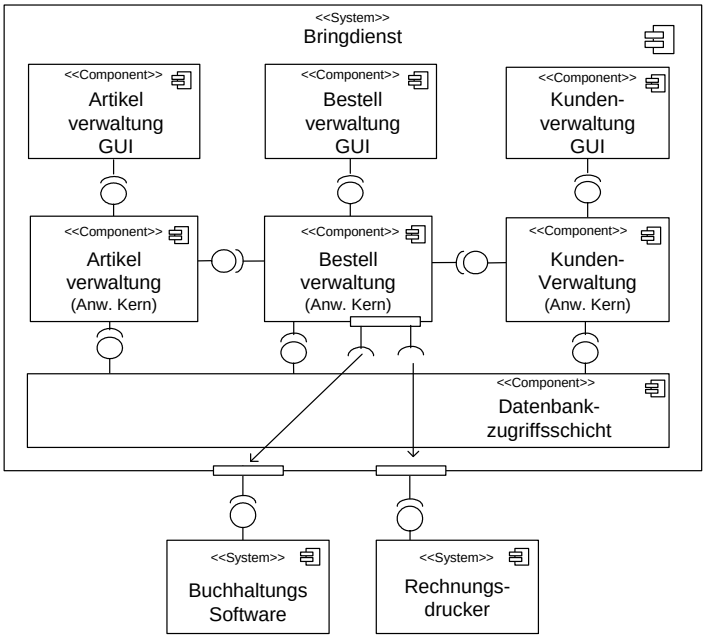


Abbildung 3.5: Technische Architektur des Bringdienstsystems

In dieser Form können sie auf einem Trägersystem installiert werden. Die Bibliotheken orientieren sich häufig an den Grenzen, wie sie durch die Struktur des Trägersystems mit seinen Rechnerknoten vorgegeben ist, etwa eine Client-Bibliothek, eine Server-Bibliothek und eine Datenbank-Bibliothek. Für die Spezifikation der Schnittstellen zu den Software-Komponenten muss die Verteilung über ein Netzwerk bereits bekannt sein, mögliche Verteilungsgrenzen müssen in Java etwa über die Verwendung bestimmter Schnittstellen (RMI) vorgesehen werden. Außerdem sind für Schnittstellen, die über ein Netzwerk aufgerufen werden, bestimmte Entwurfsrichtlinien zu beachten, etwa die Vermeidung vieler feinteiliger (get/set) Operationen [Sie04, ACM03].

### 3.3.4 Implementierungsarchitektur

Die Implementierungsarchitektur beschreibt die physische Aufteilung der Software-Komponenten auf Dateien und Verzeichnisse sowie die Abhängigkeiten zwischen den Dateien: Die Implementierung einer Software-Komponente besteht in der Regel aus mehreren Klassen und Interfaces einer objektorientierten Programmiersprache oder aus Funktionen und Strukturen einer prozeduralen Programmiersprache. Beim Entwurf der Implementierung ist daher ein entsprechender Paradigmenbruch zwischen komponentenorientiertem Entwurf und objektorientierter Implementierung zu überwinden. Die Implementierung ist zudem auf mehrere Dateien verteilt.

Den derzeit verwendeten Programmiersprachen fehlen Konzepte zur Implementierung von Komponenten und der Kontrolle der Abhängigkeiten zwischen den Quelltexten verschiedener Komponenten. Dieses Problem wird bereits 1975 von Kron und DeRemer für Programmiersprachen wie Cobol oder PL/1 beklagt [DK75]. Aufgaben bei der Erstellung einer Implementierungsarchitektur sind unter anderem:

- Physische Strukturierung der Quelltexte (und der Dateien, die Quelltexte enthalten) in IDE[24]-Projekte, Pakete, Namensräume und die Verzeichnisstruktur eines Dateisystems, mit den Mitteln der jeweiligen Programmiersprache.

[24]Integrated Development Environment

- Überwindung des Paradigmenbruchs zwischen dem komponentenorientiertem Entwurf der technischen Architektur und der objektorientierten oder prozeduralen Programmiersprache.
- Langfristige Erhaltung der Änderbarkeit des IT-Systems: Eine Übersicht über die Struktur des Codes, etwa in Form eines UML-Paketdiagramms, erleichtert das spätere Auffinden von den Stellen im Quelltext, die bei Fehlern oder Änderungswünschen anzupassen sind. Sauber in verschiedene Verzeichnisse strukturierte Quelltexte sind in der Regel besser verständlich und analysierbar als Quelltexte, die sich komplett in einem Verzeichnis auf einer Festplatte befinden.

Lakos betont die Bedeutung der Implementierungsarchitektur (physical design) für große C++ Systeme: *Logical design does not take into account physical entities such as files and libraries. Compile-time coupling, link-time dependency, and independent reuse are simply not addressed. [...] Physical design focuses on the physical entities in the system and how they are interrelated.*[Lak96, S.99f]. Lakos definiert eine Komponente: *A component is the smallest unit of physical design.*

An der Implementierungsarchitektur und ihrer Einhaltung sind Entwickler und die Qualitätssicherung interessiert, da insbesondere über die Implementierungsarchitektur die langfristige Änderbarkeit und die kurzfristige Entwickelbarkeit sichergestellt wird. Eine Komponente ist in der Implementierung:

- eine Einheit des physischen Entwurfs und der Abhängigkeitskontrolle zwischen physischen Einheiten wie Dateien,
- eine Einheit der Versionsführung: Quelltexte liegen in verschiedenen Versionen und Releases vor. Komponenten sind in der Implementierung Einheiten des Änderungs-, Release- und Versionsmanagements,
- typischerweise der Quelltext zu einer technischen Komponente sowie
- eine Einheit, die später zu einer Bibliothek zusammengebunden wird.

Ein Überblick über die Struktur der Quelltexte in Form eines UML-Paketdiagramms[Lar05, S.233] ist eine gebräuchliche Darstellung der Implementierungsarchitektur. Die Abbildung 3.6 zeigt ein Beispiel für die Aufteilung der Quelltexte des Bringdienstes in Pakete und IDE-Projekte. Die Quelltexte sind in die beiden Projekte *Bringdienst WebGUI* und *Bringdienst Core* aufgeteilt, jedes der beiden Projekte enthält mehrere Pakete (diese können Verzeichnissen in einem Dateibaum entsprechen). Die Pakete enthalten jeweils Quelltexte.

Die Abbildung 3.7 gibt ein Beispiel, wie eine Software-Komponente mithilfe der Programmiersprache Java umgesetzt werden kann. Die Komponente `KundenverwaltungAWK` hat in der Technischen Architektur die Schnittstelle `IKundenSuche`, diese verfügt über die beiden Transportstrukturen `KundenDaten` und `KundenDatenKurz`. Im Quelltext wird die Schnittstelle also durch ein Interface und zwei Klassen dargestellt.

Interface und Klassen werden im selben Java-Paket gespeichert. Dieses Paket enthält ein Unterpaket `impl` in dem sich die Implementierung der Schnittstelle befindet. Die Anforderungen an eine Software-Komponente[25] werden nur per Konvention sichergestellt. Die Programmiersprache Java verfügt über keine geeigneten Sprachmittel, um die Implementierung im `impl`-Paket zu verbergen.

### 3.3.5 Verteilungsarchitektur

Eine Komponente im Rahmen der Verteilungsarchitektur ist eine installierbare Bibliothek. Dies entspricht dem Komponentenbegriff nach Szyperski [SGM02] *'A component is what is actually deployed'*

[25] vgl. Definition 3.3

und auch dem Komponentenbegriff einiger Trägersystemteile, etwa Enterprise JavaBeans [Sun06]. Der Inhalt der Komponente kann zugekauft sein (etwa ein Datenbank-Server) oder das Compilat eines Quelltextes sein. Eine solche Komponente kann auf verschiedenen Rechnern innerhalb eines Netzwerkes installiert werden. Die Verteilungsarchitektur legt die Verteilung der Komponenten auf dem Trägersystem fest, also innerhalb eines Netzwerkes auf verschiedenen Rechnern.

Aufgabe bei der Erstellung der Verteilungsarchitektur ist insbesondere die Beeinflussung von Qualitätseigenschaften wie Durchsatz, Antwortzeiten und Verfügbarkeit. Durch die Mehrfachinstallation einer Bibliothek auf mehreren Rechnern eines Clusters kann beispielsweise die Verfügbarkeit erhöht werden (z.B. mit einer Failover-Strategie [Bri00]) und der Durchsatz kann gesteigert werden (z.B. über Loadbalancing in einem Rechner-Cluster [Bri00]). Die Quelltexte und das Trägersystem müssen dies jedoch unterstützen. Erst im Zusammenspiel von Software und Trägersystem treten die tatsächlichen Eigenschaften betrieblicher Informationssysteme hervor. Die Verteilungsarchitektur ist daher für den Systembetrieb wichtig. Eine Verteilungskomponente ist eine Einheit

- der unabhängigen Installation,
- der Verteilung in einem Netzwerk,
- der Redundanz und der Parallelisierung,
- des Managements durch ein Trägersystem sowie
- der unabhängigen Wiederverwendung.

Beschreibungen der Verteilungsarchitektur zeigen das Trägersystem mit seinen Rechnern, Laufzeitumgebungen, Netzwerkverbindungen und -protokollen. Auf dem Trägersystem werden die Verteilungskomponenten installiert. Der Installationsort jeder Komponente wird spezifiziert: Anhand von Schaubildern wird beispielsweise dargestellt, welche Komponenten redundant installiert sind.

Die Abbildung 3.8 zeigt ein Beispiel für die Darstellung des Deployments mithilfe eines UML 2.0 Verteilungsdiagramms. Zwei Bibliotheken für die oben genannten Projekte `BringdienstWebGUI.war` und `BringdienstCore.ear` werden auf einen Server mit einem Tomcat-Servlet-Container und

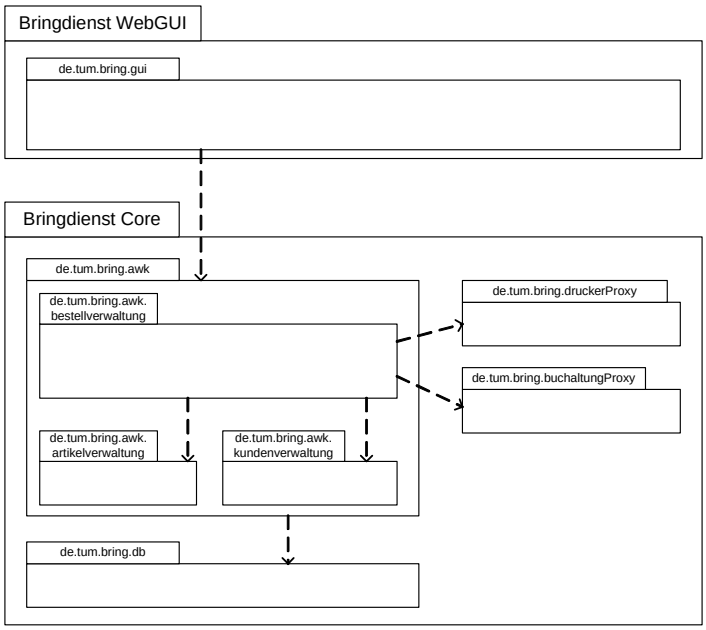


Abbildung 3.6: Implementierungsarchitektur des Bringdienstsystems

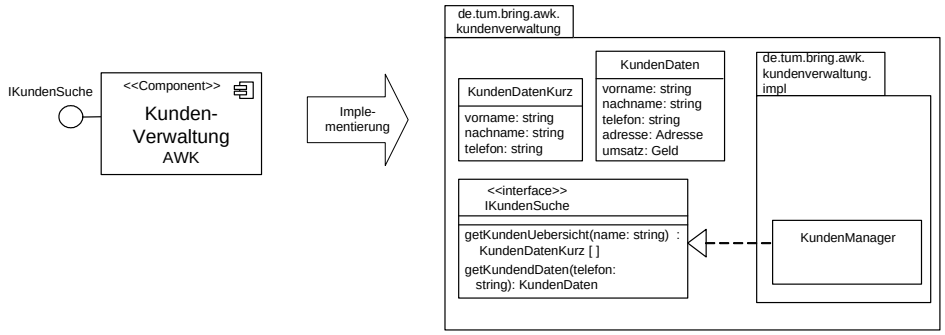


Abbildung 3.7: Abbildung einer Software-Komponente auf Code

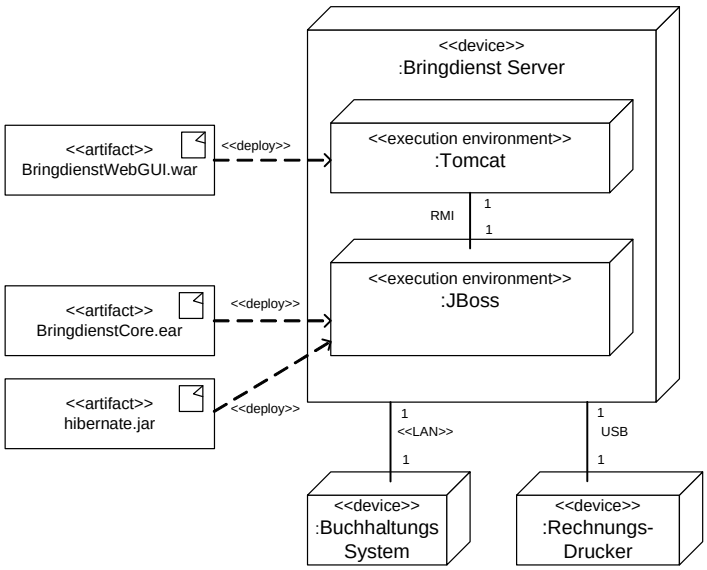


Abbildung 3.8: Verteilungsarchitektur des Bringdienstsystems

einem JBoss-Application Server installiert. Die dargestellten Bibliotheken fassen mehrere Software-Komponenten zusammen. Im vorliegenden Fall werden jeweils alle Komponenten der GUI und alle Komponenten des Anwendungskerns zu einer Bibliothek zusammengefasst.

### 3.3.6 Laufzeitarchitektur

Zur Laufzeit treten Effekte wie Verklemmungen und Ressourcen-Konflikte auf. Die Aufteilung eines IT-Systems in Prozesse und Threads beeinflusst diese Effekte und andere Eigenschaften wie Verfügbarkeit (z.B. über redundante Prozesse) und die Performance (z.B. über Parallelisierung, Sequenzialisierung). Mehrere Architekturbeschreibungssprachen, etwa Wright [All97], konzentrieren sich auf die Laufzeitarchitektur. Ziele beim Entwurf der Laufzeitarchitektur sind daher:

- Effektive Verwendung vorhandener Ressourcen wie Hauptspeicher, Rechenzeit und Netzwerkbandbreite.
- Effizienter Umgang mit Transaktionen und Zugriffskonflikten auf Daten (Sperrstrategie).
- Erfüllung von Qualitätsanforderungen wie Antwortzeiten und Durchsatz.

- Sicherer Umgang mit Nebenläufigkeit, d.h. Vermeidung von Verklemmungen, Absicherung von Fairness und Lebendigkeit.

Zur Laufzeit werden Komponenten vom Trägersystem verwaltet, etwa von einem Betriebssystem oder von einer darauf aufbauenden Laufzeitumgebung. Letzthin definiert das Trägersystem, was zur Laufzeit als Komponente betrachtet wird. In der Management-Konsole eines EJB-Containers[26] werden beispielsweise alle gerade laufenden EJB-Komponenten[27] dargestellt. Zur Laufzeit ist eine Komponente

- häufig eine Einheit der nebenläufigen Ausführung, d.h. ein Thread oder ein Prozess,
- eine Einheit des Managements durch das Trägersystem sowie
- eine Einheit der Ressourcen-Verwaltung, sie kann Ressourcen allokieren und freigeben.

Einige Ansätze setzen Komponenten und Prozesse gleich, d.h. jede Komponente wird unabhängig von anderen Komponenten parallel ausgeführt. FOKUS [BS01] ist ein Beispiel für einen solchen Ansatz.

Zwischen Laufzeitarchitektur und Implementierungsarchitektur gibt es enge Zusammenhänge. Die Verwaltung der Ressourcen, die Synchronisation des Zugriffs auf Ressourcen und das Starten und Stoppen von Threads können in den Quelltexten des Systems implementiert sein. Moderne Laufzeitumgebungen, wie etwa Applikationsserver oder Transaktionsmonitore übernehmen das Ressourcen-Management und die Kontrolle der Nebenläufigkeit vollständig [Sie02b, Sun06, S.137ff]. Sie verbieten den direkten Zugriff auf Dateien oder Datenbanken sowie das Starten und Stoppen von Threads in Quelltexten – Der Entwickler soll sich nicht mehr um Nebenläufigkeit oder Ressourcen-Management kümmern.

Die Abbildung 3.9 zeigt ein mögliches Beispiel für die Laufzeitarchitektur des Bringdienstes. Die Laufzeitarchitektur stellt die Prozesse und Threads zur Laufzeit dar und macht explizit, welcher Quelltext / welche (Instanz einer) Software-Komponente jeweils in einem eigenen Thread ausgeführt wird. Die Darstellung macht explizit, welche Teile der Software nebenläufig ausgeführt werden. Mögliche Ressourcen-Konflikte können so explizit gemacht werden: Viele Threads greifen beispielsweise gleichzeitig auf den dargestellten RECHNUNGDRUCKERPROXY zu. Die Notation ist an UML 2.0 angelehnt.

## 3.4 Sichtenkonzepte zur Architekturbeschreibung

Eine Fülle verschiedener Sichtenkonzepte[28] ist in den vergangenen Jahren entstanden [Kru95, HNS99a, HS00, CBB+03, Zac87, Sie04, Sch97, Sta05, VAC+05, RW05, GA03]. Man kann mittlerweile von einem unübersichtlichen Sichten-Zoo sprechen. Sichten stellen einerseits die oben dargestellten Architekturarten dar, andererseits können sie sich auch auf einen Sachverhalt konzentrieren, wie etwa Daten und Informationen, die in mehreren Architekturarten relevant sind. In Abschnitt 7.1 wird ein Vorschlag zur Systematisierung der Architektursichten gemacht, der es erlaubt Sichten(konzepte) einzuordnen und zu vergleichen. In den folgenden Abschnitten werden bekannte und häufig zitierte Konzepte vorgestellt.

[26] Laufzeitumgebung innerhalb der Enterprise JavaBeans Architektur, die Enterprise Beans (=Komponenten) ausführt.
[27] Enterprise Beans
[28] vgl. Abschnitt 2.3.4

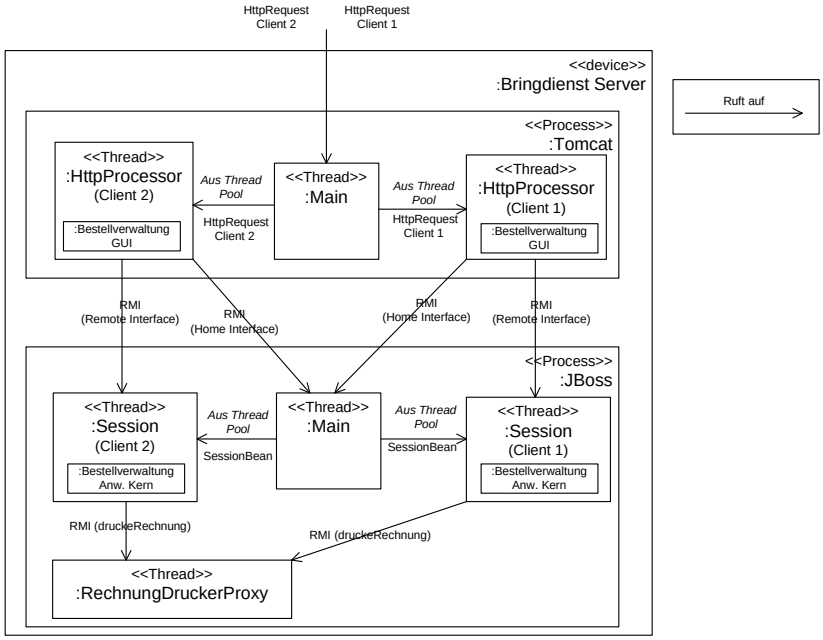


Abbildung 3.9: Laufzeitarchitektur des Bringdienstsystems

### 3.4.1 IEEE 1471

Der Standard ANSI/IEEE 1471 [IEE00] enthält Empfehlungen und Best-Practices zur Architekturbeschreibung. Er definiert neben den Begriffen Projektbeteiligter (Stakeholder), Sicht (View), Blickwinkel (Viewpoint) ein konzeptuelles Modell und setzt damit die Begriffe in Beziehung. Die Abbildung 3.10 zeigt einen Ausschnitt aus dem konzeptuellen Modell des Standards in der UML-Klassendiagramm Notation. In Abschnitt 2.3.4 wurden die Begriffe Sicht (View) und Blickwinkel (Viewpoint) eingeführt.

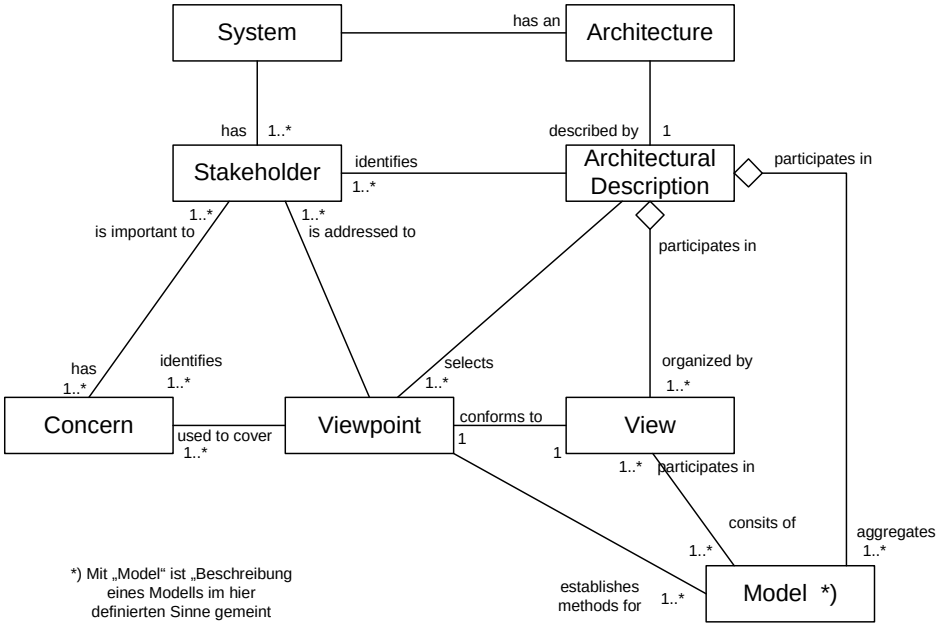


Abbildung 3.10: ANSI/IEEE 1471 Konzeptuelles Framework, Ausschnitt aus [IEE00] in UML

Wichtige Aussagen des konzeptuellen Modells sind:

- Jedes System hat genau eine Architektur und mindestens einen Stakeholder.
- Die Architektur und ihre Architekturbeschreibung sind verschiedene Dinge. Die Architektur ist eher ein gedachtes Modell des Systems. Die Architekturbeschreibung liegt in Form konkreter Modelle[29] oder Dokumente vor.
- Die Architekturbeschreibung wird über Sichten (Views) strukturiert, diese dürfen in sich tiefer strukturiert sein und weitere Modelle enthalten.
- Die Inhalte, die Methoden und die Notation jeder Sicht werden über einen Blickwinkel (Viewpoint) definiert.
- Die Blickwinkel orientieren sich an den Interessen der Stakeholder. Zur Definition eines Viewpoints gehören also neben den vorgeschlagenen Notationen, Methoden und Techniken auch die angesprochenen Stakehoder und die berücksichtigten Interessen.

Das konzeptuelle Framework verwendet den Begriff *Modell* im Sinne des hier definierten Begriffs *Beschreibung eines Modells* und unterscheidet einen Sachverhalt nicht von seiner Beschreibung.

### 3.4.2 Sichtenkategorien des Software Engineering Institute

Clements et al. [CBB+03, BCK03] versuchen, den oben erwähnten Sichten-Zoo zu systematisieren und definieren drei grundlegende Kategorien von Sichten, die *Viewtypes*. Der Module-Viewtype beschreibt dabei die Strukturierung der Implementierung, der Component-And-Connector Viewtype die Strukturierung des IT-Systems zur Laufzeit und der Allocation Viewtype beschreibt Sichten, die mit anderen Modellen zusammenhängen, etwa der Projektplanung und dem Trägersystem.

**Module Viewtype:** Module und ihre Abhängigkeiten sind die Elemente, welche im Module Viewtype dargestellt werden. Als Modul wird eine Quelltext-Einheit betrachtet: *A module is a code unit that implements a set of resposibilies. A module can be a class, a collection of classes, a layer or any decomposition of a code unit* [CBB+03, S.35]. Die Sichten des Module-Viewtypes stellen die Implementierung und ihre Strukturen dar, also die Implementierungsarchitektur. Abhängig von der Interpretation des Modulbegriffs ist auch der Software-Anteil der technischen Architektur enthalten.

**Component-and-Connector Viewtype:** Eine Komponente ist ein Verarbeitungselement eines IT-Systems zur Laufzeit: *Component-and-connector (C&C) views define models consisting of elements that have some runtime presence, such as proceses, objects, clients, servers, and data stores*[CBB+03, S.103]. Dem Component-and-Connector Viewtype können Beschreibungen der oben definierten Laufzeitarchitektur zugeordnet werden. Typische ADL wie ACME [GMW00] konzentrieren sich auf diese Sicht [CBB+03, S.145].

**Allocation Viewtype:** Der Allocation Viewtype enthält Beschreibungen der Zusammenhänge zwischen den Modulen und Komponenten der beiden anderen Viewtypes und anderen Modellen, die im Rahmen der Software-Entwicklung entstehen. Clements et al. geben drei Beispiele an: *Deployment* entspricht Beschreibungen der Verteilungsarchitektur. In *implementation* wird unter anderem das Konfigurationsmanagement der Module beschrieben sowie die grundlegende Verzeichnisstruktur des Entwicklungsprojekts[30]. Die Sichten des *work assignment* sind für das Projektmanagement relevant und werden in den Kapiteln 7 und 8 thematisiert.

[29] Mit Modell ist 'Beschreibung eines Modells' im hier definierten Sinne gemeint

[30] Diese Sachverhalte werden in der Implementierungsarchitektur beschrieben.

Das Konzept der hier vorgestellten logischen Architektur ist nicht explizit in den Viewtypes enthalten: Die logischen Komponenten sind nicht zwingend Prozesse oder andere noch zur Laufzeit sichtbare Elemente, wie dies etwa im Component-and-Connector Viewtype gefordert wird.

Die Sichtenkategorien werden über Architekturstile (Styles) detailliert [CBB+03, S.18]. Architekturstile legen die Beschreibungselemente für Sichten innerhalb einer Sichtenkategorie genauer fest. Zum Component-and-Connector Viewtype gehört beispielsweise der Architekturstil pipe-and-filter. Dieser Stil enthält Pipes (Leitungen) und Filter als Elemente, dazu passende Sichten beschreiben das IT-System unter Verwendung dieser Elemente. Die Architektursichten sind schließlich konform zu einem Architekturstil. Sie verwenden das im Stil definierte Vokabular. Der Stil definiert somit den Blickwinkel der Sichten.

### 3.4.3 Die 4+1 Views nach Kruchten

Der Kruchten strukturiert die Beschreibungen von Software-Architekturen im Architectural Document über Sichten [Kru00, S. 87], [JRB99, S. 62]. Diese wurden auf der Grundlage eigener Projekterfahrungen definiert. Kruchten hat Sichten definiert [Kru95], wobei vier Sichten durch die fünfte validiert werden:

**Logical Architecture** Die Logical Architecture gibt einen Überblick über die Strukturierung der UML-Beschreibung des Entwurfs in UML Packages. Die wichtigsten Entwurfselemente (Klassen, Packages) und -konzepte [IBM04] werden ebenso wie die Umsetzungen der wichtigsten Anwendungsfälle über Klassen bzw. Objekte (Use Case Realizations) dokumentiert. Die logische Architektur orientiert sich an den funktionalen Anforderungen: *The logical architecture primarily supports the functional requirements – what the system should provide in terms of services to its users. The system ist decomposed into a set of key abstractions, taken mostly from the problem domain* [Kru95]. Die logical Architecture ist damit verwandt zur logischen Architektur.

**Development Architecture (Implementation View)** Die Strukturierung der Software in Subsysteme, Module und Schichten ist Gegenstand der Development Architecture [Kru95] bzw. Implementation Architecture [IBM04]. Dies entspricht der Implementierungsarchitektur sowie dem Software-Teil der technischen Architektur.

**Physical Architecture (Deployment View)** Die Physical Architecture beschreibt die Verteilung der Software auf dem Trägersystem. Dies entspricht der Verteilungsarchitektur.

**Process Architecture (Process View)** Die Process Architecture thematisiert Nebenläufigkeit und Qualitätsanforderungen wie Verfügbarkeit und Performance. Die Darstellungen enthalten Prozesse und Tasks, die nebenläufig und ggf. redundant ausgeführt werden. Dies entspricht der Laufzeitarchitektur.

**Scenarios (Use Case View)** Schlüsselszenarios (allgemeine Anwendungsfälle) beschreiben, wie die anderen vier Sichten (Architekturen) zusammenarbeiten. Die Szenarios beschreiben das Verhalten des Systems, das als architekturrelevant betrachtet wird oder sie beschreiben besondere technische Risiken [IBM04].

### 3.4.4 Die Siemens-Blickwinkel

Hofmeister, Soni und Nord beschreiben in [SNH95, HNS99a, HNS99b] ein Modell mit vier verschiedenen Blickwinkeln. Die Blickwinkel stammen aus einer Untersuchung verschiedener großer und kleiner Softwareprojekte bei der Siemens AG [HNS99a, S.9].

**Conceptual View** Der Conceptual View hat das gleiche Ziel wie die logische Architektur. Sie beschreibt Komponenten, Konnektoren und Konfigurationen auf einer konzeptuellen Ebene. Den

Komponenten werden Ports als Interaktionspunkte zugeordnet, den Konnektoren werden analog Rollen zugeordnet. Der Conceptual View *describes the architecture in terms of the domain elements. The architect designs the functional features of the system* [HNS99b].

**Module View** Der Module View beschreibt die Aufteilung des Software-Systems in Module mit ihren Schnittstellen und die Aufteilung der Software in Schichten: *The module view describes the decomposition of the software and its organization into layers. An important consideration here is limiting the impact of a change in external software or hardware* [HNS99b]. Dies entspricht dem Software-Anteil der technischen Architektur.

**Code View** Die Code View beschreibt die Strukturierung der Quelltextdateien in Verzeichnisse und deren physische Abhängigkeiten über `include` und ähnliche Anweisungen. Der Code View beschreibt auch, wie die Compilate diese Dateien zu ausführbaren Programmen bzw. Bibliotheken zusammengefügt sind. Damit stellt der Code View die Implementierungsarchitektur dar.

**Execution View** Die Execution View beschreibt, wie die Module aus der Module View auf ausgeführte Einheiten (Prozesse) und Rechnerknoten abgebildet werden. Dies entspricht der oben definierten Verteilungsarchitektur zusammen mit der Laufzeitarchitektur.

### 3.4.5 Quasar Sichten

Eine Kernidee des Quasar-Projekts bei der sd&m AG [Sie02a, Sie04] beruht auf der Trennung der der eigentlichen **A**nwendung (der Fachlichkeit) von **T**echnik (dem Trägersystem). Auf dieser A/T-Trennung bauen die Konzepte von Quasar auf. So wird zwischen A-Software und T-Software unterschieden. In der Implementierung ist beispielsweise eine fachliche Klasse `Kunde` A-Software, wenn sie keine technischen Bibliotheken verwendet, während die Klasse `JFrame` als Element der technischen Bibliothek Swing als T-Software betrachtet wird. Eine Java-Klasse `KundeFrame`, die etwa zur Implementierung einer GUI-Maske A- und T-Software verwendet und implementiert, wird als AT-Software bezeichnet. Eine wichtige Entwurfsregel besagt, dass der Anteil von AT-Software so gering wie möglich gehalten werden sollte.

Im Rahmen des Quasar-Projektes wurde ein Konzept zum Entwurf und der Dokumentation von Software-Architekturen entwickelt, das drei Sichten umfasst [Sie02a]: A-Architektur, T-Architektur und TI-Architektur.

**A-Architektur** Eine fachlich orientierte Sicht beschreibt nur die Teile des IT-Systems, welche für den fachlichen Nutzen verantwortlich sind. Elemente sind beispielsweise Kundenverwaltung, Vertragsverwaltung oder Kontoverwaltung. Diese Sicht wird Anwendungsarchitektur oder auch A-Architektur genannt. Diese Sicht kann sowohl Bestandteile der logischen Architektur wie auch Bestandteile der technischen Architektur darstellen [Sie04, S.160f][31].

**T-Architektur** Eine zweite Sicht beschreibt die Teile des IT-Systems, die notwendig sind, damit die fachlichen Bestandteile ihren Nutzen erbringen können. Diese Sicht beschreibt, wie die fachlichen Bestandteile in die Software-Anteile des Trägersystems und zusätzliche technische Frameworks eingebunden werden. Sie beschreibt beispielsweise wie eine Komponente mithilfe von Enterprise Java Beans oder Servlets und Java Server Pages (JSP) implementiert wird. Diese Sicht wird technische Architektur oder T-Architektur genannt. Wie die A-Architektur kann auch die T-Architektur Bestandteile der logischen oder der technischen Architektur zeigen.

Die Abbildung 3.11 zeigt ein Beispiel für die T-Architektur des Bringdienstes. Dargestellt sind Typen technischer Komponenten wie sie in Java EE Applikationen [ACM03] vorkommen, etwa Java Server Pages, Servlets oder Session Beans. Die Abbildung deutet an, wenn es mehrere verschiedene Komponenten dieses Typs gibt: So gibt es mehrere JSPs, da sie die einzelnen Screens

[31] Siedersleben stellt die Abbildung der drei Architektursichten auf die Siemens-Sichten dar. Dabei sind die Quasar-Sichten weitgehend orthogonal zu den Siemens-Sichten.

der Software implementieren und es gibt mehrere Session Beans, da diese die einzelnen Anwendungsfälle implementieren.

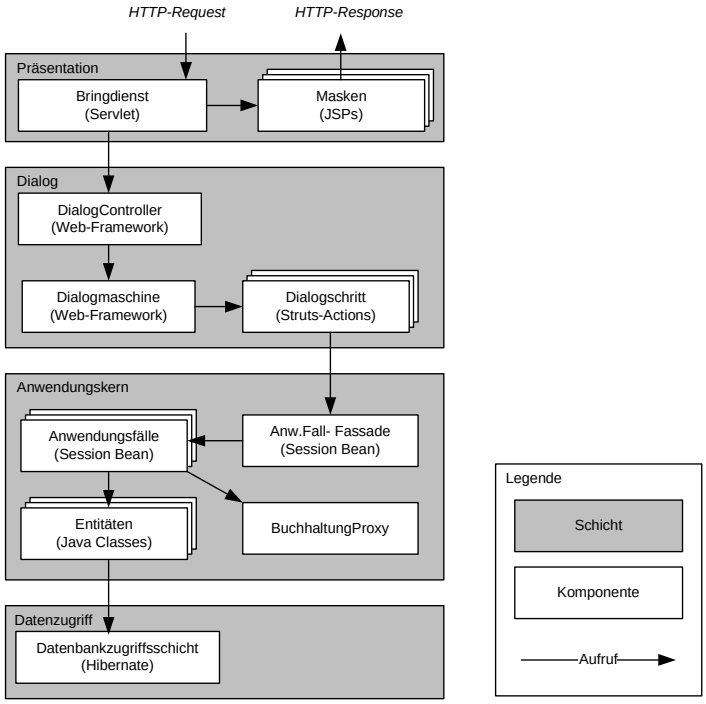


Abbildung 3.11: Beispiel für die T-Architektur des Bringdienstsystems

**TI-Architektur** Die dritte Sicht orientiert sich an den Bedürfnissen des Systembetriebs und beschreibt die Verteilung der Software auf Rechenknoten. Hier wird dargestellt, welche Systemteile redundant sind oder wie die Lastverteilung bewerkstelligt wird. Diese Sicht wird Technische- Infrastruktur- Architektur oder kurz TI-Architektur genannt. Sie entspricht der Verteilungsarchitektur.

## 3.5 Sprachen zur Architekturbeschreibung

Software- und Systemarchitekturen können in verschiedenen formellen und informellen Sprachen dargestellt werden. Wie bei den Sichtenkonzepten ist hier eine Fülle allgemeiner Sprachen und spezialisierter Architekturbeschreibungssprachen (ADL) veröffentlicht worden. Im Folgenden werden informelle Notationen, der Standard UML 2.0, AutoFOCUS2 sowie ausgewählte ADL dargestellt, soweit sie für die vorliegende Arbeit relevant sind. Die Einsatzgebiete der Sprachen werden zusätzlich herausgearbeitet.

Informelle Notationen und die UML 2.0 sind universelle Sprachen, sie sind nicht auf eine bestimmte Architektursicht bzw. Architekturart festgelegt. Verschiedene Architekturarten können in derselben Sprache dargestellt werden. Architekturbeschreibungssprachen legen dagegen häufig die Architekturart explizit fest, etwa die Laufzeitarchitektur.

### 3.5.1 Text und Ad-Hoc-Notationen

Ad-Hoc Notationen werden häufig verwendet, um Sachverhalte einer System- oder Software-Architektur zu illustrieren und zu dokumentieren. Sie verwenden einfache grafische Symbole wie Recht-

ecke, Pfeile (Box-and-Arrow) oder Piktogramme. In einer Besprechung kann ein Diagramm an ein Whiteboard gezeichnet und im Team diskutiert werden. Symbole und Diagrammelemente können ad-hoc erfunden und erklärt werden. Beliebige Sachverhalte lassen sich so visualisieren.

Die erstellten Diagramme erfordern keine tiefen Kenntnisse einer Notation, sondern nur ein grundlegendes Verständnis des dargestellten Sachverhalts. Sie sollten intuitiv verständlich sein und sind daher auch für Stakeholder ohne besonderes Hintergrundwissen geeignet. Zusätzlich können sie auch ästhetische Anforderungen in Form einer ansprechenden Darstellung erfüllen. Dies kann für eine Präsentation der Architektur vor einem Kunden oder dem Management wichtig sein.

Die Bedeutung der informellen Diagrammelemente ist nicht genau definiert, sie wird durch eine Legende und begleitenden Text beschrieben. Es ist nicht auszuschließen, dass in einem Team kein einheitliches Verständnis über die Bedeutung der Diagrammelemente besteht [Kel03, S.79] und dieses erst aufwändig hergestellt werden muss.

### 3.5.2 Unified Modeling Language, Version 2.0

Mehrere Autoren haben die UML 1.5 auf ihre Eignung zur Architekturbeschreibung untersucht, das sind beispielsweise Garlan und Kompanek [GK00], Medvidovic et al. [MRRR02] oder Hilliard [Hil99]. Garland und Anthony [GA03] sowie Hofmeister et. al [HNS99b] stellen mehrere Architektursichten auf mit den Notationsmitteln der UML 1.5 dar. Untersuchungen zur UML 2.0 wurden ebenfalls publiziert, beispielsweise [VAC+05, PBG07, Sta05]. Die vorliegende Arbeit konzentriert sich auf die UML 2.0, da sie die derzeit gültige Fassung darstellt. Mehrere Beispiele für UML 2.0 Darstellungen finden sich in Abschnitt 3.3.

Die UML 2.0 hat insgesamt 13 verschiedene Diagrammtypen, deren Syntax über ein Metamodell [OMG03] definiert ist. Diagramme können grob in drei Klassen eingeteilt werden

1. Struktur: Klassen- und Objektdiagramm, Komponentendiagramm, Kompositionsstrukturdiagramm, Verteilungsdiagramm
2. Modellstruktur: Paketdiagramm
3. Verhalten: Anwendungsfalldiagramm, Aktivitätsdiagramm, Zustandsdiagramm, Timing-Diagramm, Sequenzdiagramm, Kommunikationssdiagramm, Interaktionsübersichtsdiagramm

Die Semantik der UML 2.0 ist zwar in Teilen textuell definiert, ein Aktivitätsdiagramm kann beispielsweise als Petrinetz ausgeführt werden, jedoch ist die Übersetzung eines UML 2.0 Modells in ein lauffähiges System oder wenigstens in Quelltexte einer Programmiersprache nicht standardisiert. Letzthin ist es dem Architekten oder Modellierer überlassen, welche Inhalte dargestellt werden. Der UML-Standard legt nur bei den Verteilungsdiagrammen die Architekturart fest, lediglich einige Sachverhalte (wie Daten, Funktionen, ...) können direkt über Diagrammtypen der UML unterschieden werden.

Typischerweise wird die UML als Ausgangspunkt für spezialisierte Sprachen verwendet, die als UML-Profile definiert werden. Beneken et al. übersetzen beispielsweise das Vokabular einer Referenzarchitektur in ein UML-Profil [BSB+04]. Die Semantik dieser Erweiterungen kann unterschiedlich definiert werden entweder implizit durch einen Codegenerator [BSB+04] oder mit den Mitteln der Mathematik [Jür05]. Auch für die Modellierung von Software-Architekturen wurden UML-Profile vorgeschlagen, die beispielsweise Konzepte bekannter Architekturbeschreibungssprachen mit der UML 1.5 nachbilden [MRRR02].

Häufige Anwendung der UML ist die Darstellung der technischen Architektur und der Implementierungsarchitektur. Die in einem Klassendiagramm dargestellten Klassen werden häufig die Klassen einer objektorientierten Sprache assoziiert. Einige Modellierungswerkzeuge legen dies nahe, da Klassendiagramm und Quelltext gleichzeitig bearbeitet werden: Derselbe Sachverhalt wird mit zwei verschiedenen Sprachen dargestellt.

### Architekturelemente und Komponentenbegriff

In der UML 2.0 wurden zwei Diagrammtypen ergänzt, mit denen Architekturthemen beschrieben werden können, das sind Komponentendiagramme und Kompositionsstrukturdiagramme. Die Abbildung 3.12 zeigt einen Ausschnitt aus dem Metamodell der UML 2.0 wie es in [OMG03] dokumentiert ist: Eine Komponente kann beliebig viele Schnittstellen exportieren (provide) und beliebig viele Schnittstellen importieren (require). Sie kann aus beliebig vielen Realisierungselementen bestehen (realization). Jedes Realisierungselement verweist dabei auf einen *Classifier*. Ein Classifier ist ein Basiselement der UML 2.0 er kann beispielsweise eine Klasse oder eine Komponente sein. Das dargestellte Metamodell entspricht der Definition 3.3 für Software-Komponenten in der vorliegenden Arbeit.

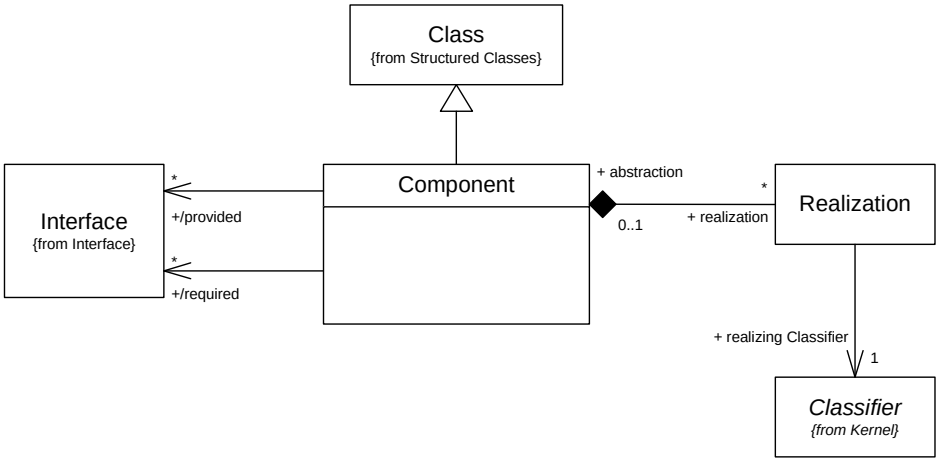


Abbildung 3.12: Metamodell der UML 2.0 zu Komponenten analog [OMG03, S.151]

### Architekturdarstellung

Ein Architekt kann die UML Notation relativ frei verwenden, um eine Architektur zu definieren. Ein Klassendiagramm kann beispielsweise ein Domänenmodell [Oes01] darstellen, das in den Bereich der Anforderungen gehört oder mehrere Java-Klassen darstellen und damit als Implementierungsmodell zählen. Ein Aktivitätsdiagramm kann einen Geschäftsprozess ebenso wie den Pageflow eines Web-Frameworks (Technisches Modell) darstellen. Die Tabelle 3.1 macht Vorschläge, mit welchen Diagrammtypen die Architekturarten beschrieben werden können.

### Kritische Würdigung der UML

Die UML 2.0 ist standardisiert, damit ist die Syntax der Diagramme und ihrer Elemente eindeutig definiert und diese Notation ist sehr weit verbreitet. Die Wahrscheinlichkeit von Missverständnissen ist geringer als bei informellen Notationen. Da die Semantik der UML 2.0 nicht präzise festgelegt ist, bleiben jedoch Interpretationsspielräume.

Eine große Menge spezialisierter Werkzeuge steht zur Erstellung und Verarbeitung der UML-Modelle zur Verfügung. Die Modelle können damit erstellt und darüber hinaus in Analyse-Werkzeugen weiterverarbeitet werden oder aus ihnen können Quelltexte generiert werden [Fra03].

Die UML ist eine umfangreiche Notation. Die beiden Spezifikationsdokumente UML-Infrastructure und UML-Superstructure umfassen zusammen mehr als 800 Seiten. Um die Notation vollständig zu beherrschen, ist einiger Einarbeitungsaufwand bzw. Schulungsaufwand erforderlich, Missverständnisse durch Unkenntnis der Notation sind nicht auszuschließen.

| Modellart | Diagrammart | Beschreibung |
|---|---|---|
| Logische Architektur | Komponenten-diagramm, Klassen-diagramm | Strukturierung des IT-Systems, ggf. zusätzlich logisches Datenmodell, Anwendungsfälle etc. |
| Technische Architektur | Komponentendiagramm | Software- und Hardware-Komponenten mit ihren Ports und Schnittstellen |
| Implementierungs-architektur | Paketdiagramm | Darstellung der Quelltextstruktur (Namespaces, Packages) |
| Verteilungsarchitektur | Verteilungsdiagramm | |
| Laufzeitarchitektur | Verteilungsdiagramm, Objektdiagramm, Interaktionsdiagramm, spezialisierte UML Profile | Kommunikation kann beispielsweise in Timing-, Kommunikations-, Sequenz- und Interaktionsübersichtsdiagrammen dargestellt werden |

Tabelle 3.1: Architekturarten mit der UML

Für eine Analyse der Eigenschaften des spezifizierten Systems genügt die in der UML 2.0 definierte Semantik in der Regel nicht. Abhängig von den zu untersuchenden Eigenschaften muss diese der UML beispielsweise über UML-Profile mit formal definierter Semantik aufgeprägt werden. Ein Beispiel für eine derartige Erweiterung ist die UMLsec nach Jürjens [Jür05].

### 3.5.3 Architecture Description Languages (ADL)

Eine Architekturbeschreibungssprache (ADL) ist eine spezialisierte grafische oder textuelle Sprache zur Beschreibung von System- und Software-Architekturen. Typische Beschreibungselemente sind Komponenten, Konnektoren und Konfigurationen. ADLs haben den Anspruch, Architekturen explizit und eindeutig darzustellen, bevor das IT-System implementiert wird. ADLs betonen grobteilige Strukturen [MT00].

Beispiele für ADLs sind Darwin [MDEK95, MK96], Rapide [Luc95] oder Wright [All97]. Mehrere Beiträge geben eine Übersicht über die Eigenschaften bestehender ADLs etwa Medvidovic et al. [MR97, MT00]. In Forschung und Praxis gibt es weder eine Standard-ADL noch ein gemeinsames Verständnis von Aufgaben und Zielen. Viele domänen- oder problemspezifische Sprachen sind bislang entstanden.

Die Sprachen wie Rapide oder Wright erlauben es, das Verhalten eines Systems formal zu beschreiben, dazu wird eine konkrete textuelle Syntax angeboten. Daraus ergeben sich die Vorteile formaler Beschreibungstechniken, wie die Prüfbarkeit auf syntaktische Korrektheit und auf Widerspruchsfreiheit. Die Exaktheit der Beschreibung und Vermeidung von Missverständnissen sind weitere Vorteile. ADL erlauben in der Regel die Analyse und/oder Simulation des Verhaltens des geplanten IT-Systems. Typischerweise stellen diese ADL nur die Laufzeitarchitektur eines IT-Systems dar, da das Verhalten des Systems analysiert oder simuliert werden soll. Die Implementierung wird normalerweise nicht betrachtet.

Zur Erstellung sowie zur Analyse oder Simulation stellen ADLs in der Regel eine Reihe von spezialisierten Werkzeugen bereit. Analyse- oder Simulationswerkzeuge erlauben abhängig von der jeweiligen Sprache eine frühe Prognose von Systemeigenschaften. Voraussetzung für die Analyse und Simulation ist eine definierte Semantik der Beschreibungselemente innerhalb der ADL. In der Regel ist eine mathematische Theorie, ein Kalkül oder ein formales Systemmodell Grundlage für die Definition der Semantik. Theorien wie beispielsweise CSP, Petrinetze oder Temporallogik finden in ADLs Verwendung [GMW97]. Die verwendete Theorie schränkt die Einsatzmöglichkeiten einer Sprache auf ein bestimmtes Anwendungsfeld ein. Die Sprache AEMILIA verwendet beispielsweise die Warteschlangentheorie und beschränkt sich damit auf die Analyse von Performance-Eigenschaften. Für

die Analyse und Simulation bestimmter Systemeigenschaften sind also jeweils Beschreibungen in entsprechend spezialisierten ADL erforderlich.

#### Architekturelemente und Komponentenbegriff

Komponenten, Konnektoren und Konfigurationen sind wesentliche Bestandteile von Architekturbeschreibungssprachen [Gar03, MT00]. Die Begriffe wurden allgemein bereits in Abschnitt 3.2 eingeführt. Die spezifische Ausprägung dieser Konzepte unterscheidet sich von Sprache zu Sprache. Im Folgenden werden die Sprachen ACME und xADL 2.0 vorgestellt, da sie die in der vorliegenden Arbeit erstellten Konzepte teilweise enthalten.

#### Second Generation ADL

Die Autoren der Sprachen ACME und xADL 2.0 bezeichnen ihre ADL als Sprachen der *'zweiten Generation'*. Sie kritisieren an anderen bekannten Sprachen der 'ersten Generation' wie Rapide oder Wright, dass diese zu eng auf eine Problemstellung ausgerichtet seien und daher nur schwer auf andere Probleme anwendbar sind. Beide Sprachen konzentrieren sich daher auf die statische Struktur eines Systems und verwenden allgemein anerkannte Abstraktionen wie Komponente und Konnektor. ACME und xADL 2.0 verfügen über Erweiterungsmechanismen, mit denen Verhaltensbeschreibungen ergänzt werden sollen. Diese Sprachen sind damit flexibler als die Sprachen der ersten Generation.

**ACME** Garlan, Monroe und Wile haben ACME [GMW97, GMW00] zum Austausch von Architekturbeschreibungen vorgeschlagen. ACME erfasst statische Informationen über die Kommunikationsstruktur. Verhaltensbeschreibungen sollen über andere ADL ergänzt werden. Dazu bietet ACME die Möglichkeit, den Komponenten und Konnektoren Attribute zuzuordnen. Eine Verhaltensbeschreibung kann der Wert eines Attributes sein. Eine Fallstudie wurde für den Austausch zwischen einer Beschreibung in Wright und in Rapide durchgeführt [GW99].

ACME bietet eine konkrete Syntax an, diese wird in [GMW00] beispielhaft dargestellt. Das nachfolgende Beispiel soll einen Eindruck von der Sprache ACME vermitteln. Der Quelltext definiert die Komponenten `bestellverwaltung` und `kundenverwaltung` mit entsprechenden Ports und einen (mit Rückgabewert arbeitenden) Konnektor `direkterAufruf`. Die beiden Komponenten werden mithilfe der `Attachments` über den Konnektor verbunden.

```
System bringdienst = {
  Component bestellverwaltung  = { Port sucheKundenDaten }
  Component kundenverwaltung   = { Port kundenInformationen }
  ...
  Connector direkterAufruf = { Roles {caller, callee} }
  Attachments : {
    bestellverwaltung.sucheKundenDaten to direkterAufruf.caller;
    kundenverwaltung.kundenInformationen to direkterAufruf.callee }
}
```

ACME konzentriert sich auf die Laufzeitarchitektur und betrachtet die Implementierung des IT-Systems nicht explizit [CBB+03, S. 163ff]. Für die Entwicklung betrieblicher Informationssysteme werden jedoch beide Darstellungen benötigt, dies wird in Abschnitt 3.3.1 verdeutlicht.

**xADL 2.0** xADL 2.0 wurde von Daschofy, van der Hoek und Taylor [DvdHT05] vorgeschlagen. xADL 2.0 basiert auf XML und ist wegen seines modularen Aufbaus erweiterbar. Wie ACME bietet

xADL 2.0 keine eigene Verhaltensbeschreibung an. Die Autoren wollen über die in xADL 2.0 vorgesehenen Erweiterungsmechanismen entsprechende Verhaltensbeschreibungen ergänzen (lassen).

xADL 2.0 besteht aus einer Kernsprache xArch[32], die grundlegende Elemente wie Komponenten und Konnektoren enthält und die hierarchische Strukturierung einer Architekturbeschreibung erlaubt. Die Syntax von xArch und xADL 2.0 ist festgelegt mit Hilfe mehrerer XML-Schemata. Die standardisierten Erweiterungsmechanismen der XML-Schemata bilden die syntaktische Grundlage für den Erweiterungsmechanismus von xADL 2.0.

Verschiedene XML-Schemata bauen xArch zu xADL 2.0 aus: Über xArch selbst werden Laufzeitarchitekturen spezifiziert, Beschreibungselemente sind Instanzen von Komponenten und Konnektoren. Auf xArch baut ein Schema zur Darstellung der Architektur zur Entwurfszeit auf. Der unten dargestellte gekürzte xADL 2.0 Quelltext zeigt eine Darstellung des Bringdienstes zur Entwurfszeit. Er soll einen Eindruck von xADL 2.0 Beschreibungen vermitteln: Komponenten, Konnektoren und Interfaces (Ports, Rollen) werden über XML-Tags dargestellt. Die Interfaces der Komponenten und Konnektoren werden über Link-Elemente miteinander verbunden.

```
<xArch>
<archStructure>
  <component id="bestellverwaltung">
    <interface id="bestellverwaltung.sucheKundenDaten">
      <direction>out</direction>
    </interface>
  </component>

  <component id="kundenverwaltung">
    <interface id="kundenverwaltung.kundenInformationen">
      <direction>in</direction>
    </interface>
  </component>

  <connector id="direkterAufruf">
    <interface id="direkterAufruf.caller">
      <direction>in</direction>
    </interface>
    <interface id="direkterAufruf.callee">
      <direction>out</direction>
    </interface>
  </connector>

  <link id="link1">
      <point> <anchor href="#bestellverwaltung.sucheKundenDaten"/> </point>
      <point> <anchor href="#direkterAufruf.caller"/> </point>
  </link>

  <link id="link2">
      <point> <anchor href="#kundenverwaltung.kundenInformationen"/> </point>
      <point> <anchor href="#direkterAufruf.callee"/> </point>
  </link>
</archStructure>
</xArch>
```

Zusätzlich bietet xADL 2.0 ein XML-Schema zur Unterstützung des Konfigurationsmanagements, außerdem gibt es XML-Schemata mit denen optionale Komponenten sowie verschiedene Varianten beschrieben werden können. Diese Schemata sind für die Entwicklung von Produktlinien-Architekturen vorgesehen. Für die Generierung von Quelltexten sind ebenfalls XML-Schemata vorhanden. Aus einer xADL 2.0 Beschreibung können so Quelltexte in einer Programmiersprache wie Java erzeugt werden.

[32] Welche zusammen mit der CMU entwickelt wurde. An der CMU wurde auch ACME definiert

### 3.5.4 AutoFOCUS 2

Zur Modellierung von System- und Software-Architekturen kann eine ganze Reihe formaler Ansätze verwendet werden. Sprachen wie FOCUS [BS01] haben eine formal definierte, mathematisch präzise Semantik und können eine ganze Reihe erfolgreich abgeschlossener Projekte vorweisen, welche diese Ansätze verwendet haben.

Hier wird das Werkzeug AutoFOCUS2 vorgestellt. Mit diesem mit dem Architekturbeschreibungen grafisch auf der Grundlage von FOCUS erstellt werden können. Die mit AutoFOCUS2 erzeugten Strukturbeschreibungen eines IT-Systems werden in der vorliegenden Arbeit verwendet.

#### Architekturelemente und Komponentenbegriff

FOCUS und AutoFOCUS2 sind streng hierarchisch sowohl in der Struktur- wie auch in der Verhaltensmodellierung. Eine Komponente ist eine stromverarbeitende Funktion, welche Nachrichten an Eingangskanälen empfängt und Nachrichten an Ausgangskanälen produziert. Komponenten kommunizieren asynchron, im Prinzip arbeitet jede Komponente nebenläufig. Komponenten haben einen Zustand. Das Verhalten einer Komponente kann beispielsweise durch einen Zustandsautomaten dargestellt werden. Eine mehrfache Verwendung identischer Komponenten (Instanziierung über Kopien) ist möglich. Über die Relation *SharedComponents* wird in AutoFOCUS 2 sichergestellt, dass die im Modell mehrfach verwendeten Komponenten identisch bleiben. Den Komponentendarstellungen liegt ein Metamodell zugrunde, von dem ein Ausschnitt in Abbildung 3.13 dargestellt wird.

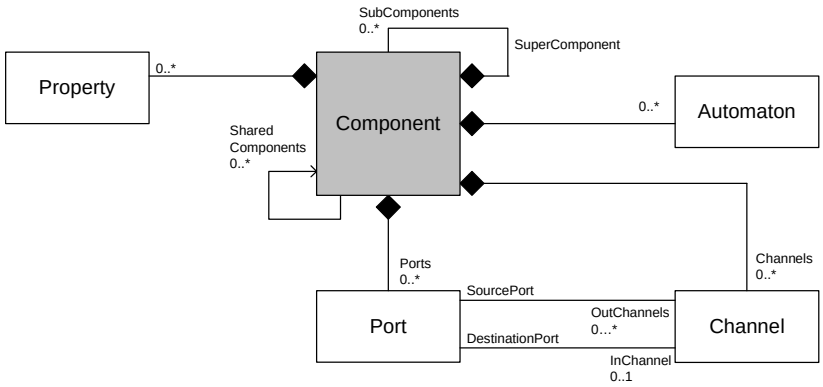


Abbildung 3.13: Auszug des AutoFOCUS2 Metamodells

Bei AutoFOCUS2 ist die Verhaltensbeschreibung streng hierarchisch, d.h. jeder Komponente kann eine eigene Beschreibung zugeordnet werden, beispielsweise in Form eines endlichen Automaten. Das Systemverhalten ergibt sich aus der Interaktion der kommunizierenden Komponenten. Ein Komponenten übergreifendes Verhalten wird durch EETs (Extended Event Traces) dargestellt, diese werden jedoch in der Regel durch Simulation der Automaten erzeugt und nicht durch einen Architekten spezifiziert. Eine Erweiterung der Informationen zu Komponenten ist über die Properties möglich.

Die Komponenten von FOCUS und AutoFOCUS2 sind Elemente einer logischen Architektur, Quelltexte oder die Verteilung werden nicht dargestellt. Die Konfiguration ist statisch, d.h. zur Laufzeit können keine Komponenten ergänzt oder entfernt werden. Zwischen Komponententypen und -instanzen wird nicht unterschieden.

#### Architekturdarstellung

In AutoFOCUS 2 wird die Systemstruktur in SSD (System Structure Diagram) dargestellt. Die Elemente des Diagramms sind im Metamodell aus Abbildung 3.13 definiert.

In Kapitel 9 wird das Werkzeug AutoARCHITECT vorgestellt, das die theoretischen Ergebnisse der vorliegenden Arbeit praktisch nutzbar macht. Zur Beschreibung der logischen Architekturen wird dabei AutoFOCUS 2 verwendet.

## 3.6 Zusammenfassung

Dieses Kapitel gibt einen Überblick über das Themenfeld System- und Software-Architektur. Dabei ist die Software-Architektur ein Modell eines IT-Systems, das die Strukturen der Software beschreibt. Die Systemarchitektur beschreibt zusätzlich Strukturen innerhalb des Trägersystems. Es wird herausgearbeitet, dass ein IT-System und seine Software mehrere grundlegend verschiedene Strukturen aufweisen. Logische, technische, Implementierungs-, Verteilungs- und Laufzeitarchitektur werden unterschieden. Diese Strukturen unterscheiden sich insbesondere in den Entwurfsproblemen und Architekturtreibern, die auf ihrer Ebene bearbeitet werden. Eigenschaften wie Verfügbarkeit und Performance werden beispielsweise vorwiegend über Verteilungs- und Laufzeitarchitektur wesentlich beeinflusst.

Zentrale Begriffe in der Software-Architektur sind Komponente, Konnektor und Konfiguration. Die Zusammenhänge dieser Begriffe wird herausgearbeitet. Für logische Architekturen werden diese Begriffe spezifisch definiert.

Bekannte Sichtenkonzepte zur Beschreibung von Software- und Systemarchitekturen werden dargestellt und diskutiert. Im Detail werden die Sichten nach Hofmeister et al. [HNS99a], Kruchten [Kru95], Siedersleben [Sie04] und nach dem Software Engineering Institute [CBB+03] dargestellt. Sprachen wie UML 2.0 und Architekturbeschreibungssprachen (ADL) sowie AutoFOCUS 2 können zur Beschreibung von System- und Software-Architekturen verwendet werden. Diese Sprachen werden dargestellt, diskutiert und verglichen.

# Kapitel 4

# Projektmanagement

In den Kapiteln 7 und 8 werden Architektursichten und Verfahren vorgeschlagenen, die Informationen des Projektmanagements auf die logischen Architektur eines IT-Systems abbilden. So werden etwa zu jeder logischen Komponente Erstellungsaufwand, Fertigstellungstermin und verantwortliches Team dargestellt. Die logische Architektur kann Instrument des Projektmanagements verwendet werden. Das ist bereits in Abschnitt 3.3.2 angeklungen. Die Sichten und darauf aufbauende Verfahren sollen durch die Zusammenführung von Architektur- und Projektmanagementdaten die Zusammenarbeit und Kommunikation zwischen Architekt und Projektleiter verbessern sowie Planung und Architektur besser auf einander abstimmen.

Die Grundbegriffe des Projektmanagements werden, soweit sie für das Verständnis der vorgeschlagenen Sichten und Verfahren notwendig sind, in diesem Kapitel eingeführt. Der Abschnitt 4.1 gibt einen Überblick über die Aufgaben des Projektmanagements und beschreibt deren Zusammenhang als Regelkreis. Die Abschnitte 4.2 bis 4.5 stellen die Initialisierung, Planung, Kontrolle und Steuerung von Projekten dar. Andere Themen des Projektmanagements wie das Änderungsmanagement, das Risikomanagement und die Mitarbeitermotivation und -führung werden nicht betrachtet.

## Übersicht

## 4.1 Projektmanagement Regelkreis

Große betriebliche Informationssysteme werden von mehreren Personen arbeitsteilig entwickelt. Die Zusammenarbeit dieser Personen muss geplant und koordiniert werden. Dies geschieht im Rahmen des Projektmanagements. Aufgabe des Projektmanagements ist es, das Projekt so zu planen und zu steuern, das es im vorgegebenen Zeit-, und Budgetrahmen das vorgegebene Ziel erreicht. Lannes [Sie02b, S.322] formuliert dies so: *Projektmanagement ist die Summe aller steuernden Schritte zur Planung, Organisation, Durchführung und Einführung eins Software-Projektes, so dass das gegebene Ziel zeit- und qualitätsgerecht erreicht wird.*.

Das Projektmanagement wird in der Literatur häufig mit einem Regelkreis verglichen [HHMS04, S.10], [Bur02, S.17]: Das Projektziel wird über die Projektplanung operationalisiert. Die Planung legt den *Soll*-Ablauf des Projektes fest, an dessen Ende das Projektziel erreicht ist. Termine von Meilensteinen werden festgelegt, der Projektumfang (die zu erbringende Leistung) wird definiert und die Arbeitspakete zur Erfüllung des Projektumfangs werden entlang der festgelegten Meilensteine geplant. Auf der Grundlage der Planung wird ein Projekt durchgeführt. Das Projekt ist jedoch Störungen ausgesetzt, etwa Änderungen in den Anforderungen, technischen Problemen oder Fehlern in der Schätzung, die dem Plan zugrunde liegt. Die Projektkontrolle erhebt den Ist-Ablauf des Projekts. Die aus den Störungen entstehenden Abweichungen zwischen Soll und Ist werden darüber festgestellt. Über die Projektsteuerung wird versucht, diese Störungen auszugleichen. Dieser Regelkreis des Projektmanagements ist in Abbildung 4.1 dargestellt:

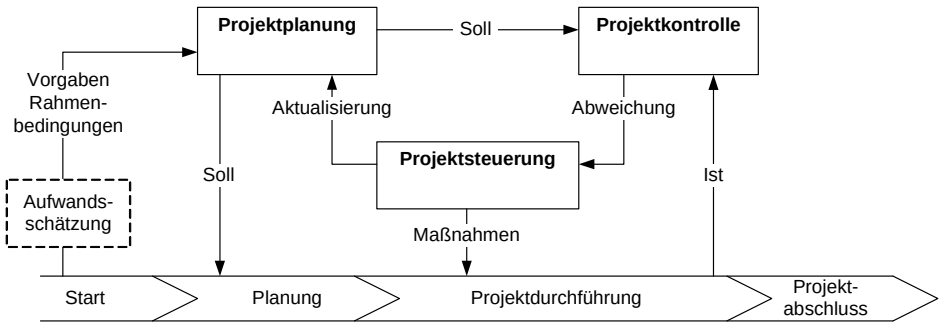


Abbildung 4.1: Regelkreis des Projektmanagements, frei nach [HHMS04, S.10]

Der Begriff Projektmanagement insgesamt ist noch weiter gefasst: In der Norm DIN 69901 [DIN87c] wird er definiert als *Gesamtheit von Führungsaufgaben, -organisation, -techniken und -mittel für die Abwicklung eines Projekts.* Die Teilaufgaben Planung, Kontrolle und Steuerung werden in den folgenden Abschnitten diskutiert. Andere Aspekte des Projektmanagements, wie das von Lannes genannte Thema Einführung von Software wird nicht weiter verfolgt, da die in den Kapiteln 7 und 8 vorgeschlagenen Lösungen sich auf den Projektmanagement Regelkreis konzentrieren.

In Abschnitt 3.1 ist es bereits angeklungen: Software-Architektur und Projektmanagement hängen offenbar eng zusammen. Häufig wird beklagt, dass dieser Zusammenhang nicht ausgenutzt wird und Projektleiter ihre Planung ohne Kenntnis der Architektur machen [Sta05, S.28]. An der Projektdefinition und -durchführung sollten jedoch Architekt und Projektleiter gleichermaßen mitwirken und gut zusammenarbeiten [Pau02]. Ein Merkmal dieses *architekturzentrischen Projektmanagements* ist die *parallele Erstellung des Grobentwurfs und des Projektplans* [BD04, S.612]. Das ist Thema von Kapitel 8.

## 4.2 Projektstart

Zum Projektstart werden die Rahmenbedingungen festgelegt, auf deren Grundlage ein Projekt durchgeführt wird. Ein Vorgehensmodell wird ausgewählt[1] und an die Bedürfnisse des Projektes angepasst (getaylored). Unter Berücksichtigung des Vorgehensmodells wird die externe und die interne Projektorganisation festgelegt. Die externe Organisation umfasst Steuergremien wie den Lenkungsausschuss [Sie02b, S.336][WM04, S.31], die interne Organisation legt Rollen wie Projektleiter und Architekt ebenso fest wie eventuelle Teilprojekte oder Teilteams.

### Vorgehensmodelle

Um ein Projekte wiederholbar durchführen zu können, werden Vorgehensmodelle wie das V-Modell XT [V-M06] oder der Rational Unified Process [IBM04] angewendet. Diese geben einen wiederverwendbaren Rahmen vor. Die Grundidee der Vorgehensmodelle ist vergleichbar mit der Idee der Referenzarchitekturen [Ben06] in der Disziplin Software-Architektur: Sie schaffen ein gemeinsames Verständnis der wichtigsten Begriffe unter den beteiligten Personen und sie bilden eine wiederverwendbare Grundlage für konkrete Projekte, etwa für die Projektplanung [Gna05].

Vorgehensmodelle definieren typischerweise drei Arten von Elementen, das sind Produkte, Rollen und Aktivitäten. Ein Vorgehensmodell definiert *Produkte*[V-M06] (Artefakte [IBM04]), die während eines Projektes zu erstellen sind. Etwa ein Projektplan, ein Pflichtenheft oder die Quelltexte. *Rollen* legen Verantwortungsbereiche und Aufgaben innerhalb des Projektes fest, so kann etwa die Rolle Projektleiter für das Produkt Projektplan verantwortlich sein. In einem konkreten Projekt werden den vorhandenen Mitarbeitern die entsprechenden Rollen zugewiesen. Die Rollen aus dem Vorgehensmodell geben damit ein Schema für die Projektorganisation vor. *Aktivitäten* beschreiben die Tätigkeiten, die zur Erstellung und Fortschreibung der Produkte erforderlich sind. Weiterhin können Workflows definiert werden [Kru00], die Aktivitäten, Produkte und Rollen in Abläufen integrieren.

Die Schnittstelle zwischen Auftraggeber und Auftragnehmer kann im Rahmen eines Vorgehensmodells geregelt werden. Das V-Modell XT regelt beispielsweise, welche Ergebnisse (Produkte = Dokumente, Software, Hardware) in welcher Phase des Projektes jeweils zwischen beiden ausgetauscht werden. Diese Stellen im Projektverlauf werden im V-Modell XT *Entscheidungspunkte* genannt, da nach dem Austausch der Ergebnisse über die Projektfortführung entschieden wird.

### Projektorganisation

Die Definition der Rollen, welche die Mitarbeiter in einem Projekt einnehmen, ist abhängig von der jeweiligen Organisation und dem gewählten Vorgehensmodell, so definiert das V-Modell XT ca. 30 verschiedene Rollen auf der Seite des Auftraggebers und Auftragnehmers. Auf der Seite des Auftragnehmers gibt es typischerweise folgende wichtige Rollen [Sie02b, S. 8ff, S.334], das sind Projektmanager, Projektleiter, Architekt (Chefdesigner), Entwickler, Qualitätsverantwortlicher und bei großen Projekten Teilprojektleiter [Sie02b, S.334]. Der Projektmanager trägt dabei die Gesamtverantwortung und teilt dem Projekt Mitarbeiter zu. Der Projektleiter plant und führt das Projekt, der Architekt (Chefdesigner) ist für den Entwurf des IT-Systems und der Software verantwortlich und der Qualitätsverantwortliche organisiert die Qualitätssicherung. Entwickler erledigen die eigentliche Arbeit und erstellen Spezifikationen, Konzepte und die Quelltexte.

In großen Projekten sind viele Personen beteiligt. Ab ca. 10 Mitarbeitern können diese nicht mehr direkt von einem Projektleiter geführt werden. Das Projekt muss in Teilprojekte zerlegt werden. Die

[1] In vielen Organisationen ist das Vorgehensmodell für die gesamte Organisation vorgeschrieben. Andere Organisationen verfügen über kein explizit aufgeschriebenes Vorgehensmodell; Projekte werden dort etwa nach dem Vorbild bereits durchgeführter Projekte abgewickelt. Die tatsächliche Auswahl eines Vorgehensmodells findet daher selten statt.

Projektorganisation ist hierarchisch aufgebaut. Die Aufteilung in Teilteams ist auch zur Steuerung des mit der Projektgröße gestiegenen Kommunikationsbedarfs [Bro03] wichtig.

Für große Projekte werden gemeinsame Berichts- und Entscheidungsgremien notwendig, an denen Auftraggeber und Auftragnehmer beteiligt sind (externe Projektorganisation). Ein Lenkungsausschuss, der mit dem Management des Auftraggebers und dem Projektleiter und Projektmanager des Auftragnehmers besetzt ist, kann beispielsweise (als Eskalationsebene) Entscheidungen über die Projektinhalte, Termine und Prioritäten treffen [Sie02b, S.336].

## 4.3 Planung

Die Planung eines Projektes ist eine wichtige Voraussetzung, um das Projekt erfolgreich durchführen zu können. Es gibt bislang noch keine in der Software-Branche etablierte Methodik der Projektplanung: Viele Unternehmen haben aus vorhandenen Teilmethoden, etwa zur Terminplanung oder zur Aufwandsschätzung, eine eigene Planungsmethodik entwickelt. Die Planungsmethodik der sd&m AG wird beispielsweise von Lannes in [Sie02b, S.320 ff] beschrieben. Hindel et al. beschreiben in [HHMS04, S.41ff] die Planungsmethodik nach dem iSQI-Standard, danach erfolgt die Projektplanung in folgenden Schritten:

- Projektumfang und Meilensteine festlegen
- Projektstrukturplan erstellen
- Größen-, Aufwands- und Kostenschätzung durchführen
- Aktivitätszeitplan aufstellen
- Kostenplanung aufstellen

Die Ergebnisse der Planung werden in einen Projektplan integriert. Die Planung erfolgt dabei iterativ. Auch während der Projektdurchführung sind noch Änderungen der Planung möglich.

Der Systematik des iSQI-Standards folgend, werden im Folgenden die Schritte der Projektplanung dargestellt und diskutiert. Die wichtigsten Aktivitäten und Begriffe der Projektplanung werden beschrieben. Ähnliche Aktivitäten und Begriffe finden sich in anderen Planungsmethoden (vgl. etwa [WM04] oder [Sie02b]).

### Projektumfang und Meilensteine

#### Abgrenzung des Projektumfangs

Wichtigstes Element der Planung ist die Definition und Abgrenzung des Projektziels[2] und damit des Projektumfangs. Das erwartete Ergebnis (Lieferumfang) wird festgelegt. Der fachliche oder technische Nutzen des Projektes (seines Ergebnisses) sollte auf dieser Grundlage bestimmt werden können. Der Projektumfang kann beispielsweise über ein Lastenheft festgelegt werden [V-M06].

#### Zerlegung großer Projekte in Stufen

Um den Bau eines großen IT-Systems besser beherrschbar zu machen, kann das Projekt in *(Leistungs-)Stufen* [Sie02b, Tau04] bzw. *Iterationen* zerlegt werden. Jede Stufe kann wie ein kleines Projekt behandelt werden, mit eigener Anforderungsanalyse bis hin zur Inbetriebnahme. Ergebnis jeder Stufe

[2] *Gesamtheit von Einzelzielen, die durch das Projekt erreicht werden sollen, bezogen auf Projektgegenstand und Projektablauf.* Ein Projektgegenstand ist ein *durch die Aufgabenstellung gefordertes materielles oder immaterielles Ergebnis der Projektarbeit* [DIN97].

ist ein nutzbares (Teil-)System[3], das an den Auftraggeber ausgeliefert wird. Jede Stufe erbringt damit einen für den Auftraggeber nutzbaren Funktionsumfang.

Für die Zerlegung eines Projekts in Stufen sollte die logische Architektur verwendet werden, denn sie teilt den Funktionsumfang eines IT-Systems auf Subsysteme und Komponenten auf. Wenn ein Subsystem einen in sich abgeschlossenen Funktionsumfang liefert, kann es als Ergebnis einer Stufe verwendet werden. Abgeschlossen bedeutet in diesem Zusammenhang, dass das Subsystem die Anwendungsfälle nur auf der Grundlage bereits ausgelieferter Subsysteme und Nachbarsysteme implementiert[4].

Häufig wird zunächst nur die erste Stufe eines Projekts im Detail geplant und durchgeführt. Die Planung der weiteren Stufen findet dann auf der Grundlage der Erfahrungen der ersten Stufe und der zu dem Zeitpunkt aktuellen Anforderungen und Prioritäten statt. Verläuft die erste Stufe erfolglos, kann das Projekt mit begrenztem Verlust frühzeitig vom Auftraggeber oder Auftragnehmer abgebrochen werden.

#### Definition der Meilensteine gemäß Vorgehensmodell

Der Verlauf eines Projektes und seiner Stufen wird grob über Meilensteine festgelegt. Ihre Definition kann auf der Grundlage des gewählten Vorgehensmodells erfolgen, beispielsweise unter Verwendung der Entscheidungspunkte des V-Modells XT [V-M06]. Die Meilensteine legen fest, welche Ergebnisse zu welchem Termin in einer definierten Qualität vorliegen müssen. Entlang der Meilensteine erfolgt die Detailplanung.

**Definition 4.1 (Meilenstein nach [HHMS04])**
Ein Meilenstein ist das Erreichen eines messbaren, bedeutenden Ereignisses im Projekt (z.B. der Abschluss eines Liefergegenstandes, das Ende einer Phase) zu einem bestimmten, geplanten Termin.

### Projektstrukturplan erstellen

Eine Voraussetzung für die Planung eines Projekts ist ein genaues Verständnis der anfallenden und damit zu planenden Aufgaben. Um dieses Verständnis zu erreichen, können ein *Produktstrukturplan* und ein *Projektstrukturplan*[5] erstellt werden [EHR00, S.22]. Der Produktstrukturplan beschreibt die Strukturierung der Liefergegenstände, während der Projektstrukturplan die Aufgaben zur Erstellung der Liefergegenstände strukturiert.

*Als Produktstruktur bezeichnet man die technische Gliederung des zu entwickelnden Produkts (bzw. Systems) in seine Einzelteile; sie ist die 'Realisierungsstruktur' des Produkts*[Bur02, S. 76]. Der Produktstrukturplan ist hierarchisch gegliedert. In der Regel werden alle Liefergegenstände (Dokumentation, Schulungsunterlagen, Installationsskripte, etc.) und die während der Entwicklung entstehenden Zwischenprodukte (etwa die Software-Entwicklungsumgebung oder die Spezifikationsdokumente) im Produktstrukturplan erfasst. Die in der vorliegenden Arbeit diskutierten logischen Architekturen sind Teilbäume von Produktstrukturplänen, in denen zusätzlich Kommunikationsbeziehungen dokumentiert werden.

**Definition 4.2 (Projektstrukturplan (PSP), Work Breakdown Structure)**
Ein Projektstrukturplan (PSP) ist die Darstellung und Strukturierung aller Aufgaben, die zur Erreichung des Projektziels während der Projektdurchführung anfallen. Er beschreibt damit den Projektumfang vollständig: Alle Aufgaben, die nicht im Projektstrukturplan stehen, gehören nicht zum Projekt.

[3] bzw. Änderungen oder Erweiterungen an produktiven IT-Systemen
[4] vgl. hierzu Kapitel 8
[5] Aufgabenplan [Sie02b, S.324], Work Breakdown Structure [PMI96, S.177]

Der PSP ist typischerweise als Baum organisiert, dessen Wurzel das zu erreichende Projektziel ist. Dieser Baum kann nach verschiedenen Aspekten weiter unterteilt werden. Die erste Dekompositionsebene können beispielsweise die Projektphasen (z.B. Analyse, Design, Implementierung, etc.) sowie Querschnittsaufgaben (Projektmanagement, Qualitätssicherung, Konfigurationsmanagement, etc.) sein, die im gewählten Vorgehensmodell festgelegt werden [PMI96, S.53]. Solche Projektstrukturpläne werden auch *ablauforientierte* Projektstrukturpläne genannt [Bur02, S. 79]. Alternativ dazu kann die erste Dekompositionsebene auch der logischen Architektur des zu liefernden Systems strukturiert werden. Diese PSP werden auch *objektorientiert* [Bur02, S.78] genannt.

Die Liefergegenstände wie Software oder Dokumente und die zu erbringenden Dienstleistungen sind Elemente des PSP, unabhängig von der gewählten Struktur der ersten Dekompositionsebene. Die Arbeitspakete bilden die Blätter des PSP [DIN87c, PMI96]. Die Arbeitspakete sind zur Erstellung und Bearbeitung der Liefergegenstände bzw. zur Erbringung der Dienstleistungen erforderlich. Die Abbildung 4.2 zeigt ein Beispiel für einen (objektorientierten) Projektstrukturplan zur Erstellung einer Software für einen Bringdienst.

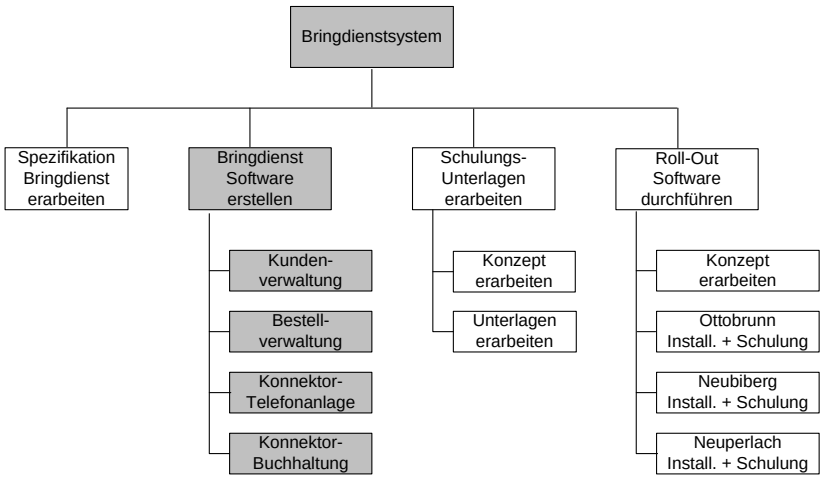


Abbildung 4.2: Beispiel für einen (objektorientierten) Projektstrukturplan

Die Elemente des Projektstrukturplans hängen offenbar von zwei Faktoren ab: dem Vorgehensmodell und der Architektur des geplanten IT-Systems. Das Vorgehensmodell definiert die zu erstellenden Produkte und die dafür notwendigen Aktivitäten. Häufig definiert ein Vorgehensmodell zusätzlich eine generische Struktur der Produkte, etwa die Kapitelstruktur der zu erstellenden Dokumente[6].

Die im Kapitel 3 vorgestellten Architekturarten definieren die Strukturen der Produkte speziell für das geplante IT-System, d.h. die Struktur hängt von den konkreten Projektinhalten ab. Über das Vorgehensmodell kann daher nur eine Liste der Top-Level Aufgaben gewonnen werden, wie etwa 'Software-Implementieren', aus der Architektur leitet sich deren Verfeinerung ab, wie etwa 'Komponente Kundenverwaltung implementieren'.

Gnatz beschäftigt sich in seiner Dissertation [Gna05] unter anderem mit der Ableitung der Aktivitäten aus einem definierten Vorgehensmodell. Er verwendet die projektspezifische Struktur der Produkte jedoch nicht für eine Detaillierung der Planung weiter.

In Kapitel 8 wird ein Verfahren zur Erzeugung eines Teils des Projektstrukturplans bzw. einer Stückliste für die Aufwandsschätzung angegeben.

[6] vgl. beispielsweise [V-M06, IBM04]

| Arbeitspaket | 'Kundenverwaltung implementieren' |
|---|---|
| ID | AP 10045 |
| Aufgabenstellung | Das Subsystem Kundenverwaltung innerhalb der Bestelldienst-Software ist gemäß der technischen Architektur zu implementieren und der Modultest ist durchzuführen |
| Ergebnis | getesteter Quelltext, technische Dokumentation als JavaDoc |
| Notwendige Fähigkeiten | Java, UML 2.0, relationale Datenbanken, Swing-Programmierung |
| Voraussetzungen | grundlegende Frameworks sind integriert (AP 9012), Testdatenbank steht zur Verfügung (AP 21778) |
| Verantwortlich | Hr. Beneken |
| Durchführend | Stuttgarter Teilteam |
| Aufwand | 7 Bearbeitermonate |

Tabelle 4.1: Arbeitspaket 'Kundenverwaltung implementieren'

## Arbeitspakete

Die Blätter des Projektstrukturplans sind Arbeitspakete. Diese sind im Grunde kleine Projekte, mit einem Starttermin, Fertigstellungstermin, geplanten Aufwänden, geplanten Einsatzmitteln und einem definierten Ergebnis.

**Definition 4.3 (Arbeitspaket, Work Package nach [pro06])**
Ein Arbeitspaket beschreibt eine in sich geschlossene Aufgabenstellung innerhalb des Projekts, die von einer einzelnen Person oder einer organisatorischen Einheit (einem Team) bis zu einem festgelegten Zeitpunkt mit definiertem Ergebnis und Aufwand vollbracht werden kann.

Die Inhalte des Arbeitspakets können über eine informelle Beschreibung des erwarteten Ergebnisses festgelegt werden. Zur Unterstützung der späteren Terminplanung werden Abhängigkeiten zu anderen Arbeitspaketen angegeben. Für die Einsatzmittelplanung ist die Definition der erforderlichen Fähigkeiten (engl. Skills) zur Umsetzung des Arbeitspaketes wichtig, um Mitarbeiter mit den richtigen Voraussetzungen auszuwählen. Zusätzlich wird jedem Arbeitspaket in der Regel eine eindeutige ID zugewiesen, damit es in anderen Planungsdokumenten referenziert werden kann. In Tabelle 4.1 ist ein Beispiel für ein solches Arbeitspaket dargestellt.

Im Rahmen der Aufwandsschätzung wird für jedes Arbeitspaket der Aufwand prognostiziert. Darauf aufbauend werden im Rahmen der Termin- und Ressourcenplanung Anfangs- und Endtermin festgelegt und Verantwortliche benannt sowie andere notwendige Ressourcen zugeordnet.

## Aufwandsschätzung

Um ein Projekt planen zu können, muss der Gesamtaufwand des Projektes und der notwendigen Arbeitspakete bekannt sein. Dafür sind verschiedene Schätzverfahren publiziert worden. Bekannte Schätzverfahren, wie die Function Point Analyse (FPA) (vgl. z.B. [BF00]) oder CoCoMo [Boe81] funktionieren nur auf der Grundlage eines vollständigen Pflichtenhefts [Sie02b, S.325]. Dieses liegt bei der Abgabe eines Angebots und in frühen Projektphasen noch nicht vor.

Typischerweise wird solchen Situationen das Verfahren der *Expertenschätzung* auf der Grundlage des Projektstrukturplans (einer Stückliste) durchgeführt. Die Delphi-Methode [HHMS04, S.50] sieht beispielsweise vor, dass mehrere Experten im Rahmen einer moderierten Sitzung den Aufwand für jedes Arbeitspaket einzeln schätzen. Die verschiedenen Schätzungen werden danach im unter den Experten diskutiert und iterativ zu einer einheitlichen Schätzung für jedes Arbeitspaket verdichtet. Die Tabelle 4.2 ist ein Beispiel für eine Stückliste.

| **Arbeitspaket** | **Expertenschätzungen: Aufwand in Personentagen** | | |
|---|---|---|---|
| | **Experte 1** | **Experte 2** | **Experte 3** |
| ... | ... | ... | ... |
| **Spezifikation Bringdienst erarbeiten: Summe** | 35 | 50 | 40 |
| ... | ... | ... | ... |
| **Feinentwurf Bringdienst erarbeiten: Summe** | 10 | 15 | 12 |
| ... | ... | ... | ... |
| Bestellverwaltung GUI erstellen | 11 | 10 | 13 |
| Bestellverwaltung Kern erstellen | 9 | 10 | 11 |
| Zu Bestellverwaltung integrieren | 3 | 4 | 5 |
| Kundenverwaltung GUI erstellen | 10 | 10 | 9 |
| Kundenverwaltung Kern erstellen | 7 | 10 | 8 |
| Zu Kundenverwaltung integrieren | 3 | 4 | 5 |
| Datenbankzugriffsschicht integrieren | | | |
| **Bringdienst Software erstellen: Summe** | 60 | 56 | 70 |
| ... | ... | ... | ... |
| **Schulungsunterlagen Erarbeiten: Summe** | 15 | 10 | 11 |
| ... | ... | ... | ... |
| **Roll Out Software durchführen: Summe** | 40 | 30 | 35 |
| **Summe Netto gesamt** | ... | ... | ... |

Tabelle 4.2: Liste der Arbeitspakete für eine Expertenschätzung

## Aktivitätszeitplan (Terminplan)

Im Rahmen der Terminplanung entsteht aus dem Projektstrukturplan ein Terminplan (Aktivitätszeitplan [HHMS04, S. 57ff]), der die zeitliche Abfolge der Arbeitspakete festlegt und jeweils Anfangs- und ein Endtermin definiert.

Eine der bekanntesten Methoden zur Terminplanung ist die *Netzplantechnik*, diese umfasst *alle Verfahren zur Analyse, Beschreibung, Planung, Steuerung, Überwachung von Abläufen auf der Grundlage der Graphentheorie, wobei Zeit, Kosten, Einsatzmittel und andere Einflußgrößen berücksichtigt werden können* [DIN87a].

Der zeitliche Projektablauf wird über Netzpläne beschrieben. Ein Netzplan ist allgemein eine *graphische oder tabellarische Darstellung von Abläufen und deren Abhängigkeiten* [DIN87a]. Eine typische Darstellung eines Netzplans ist ein gerichteter Graph, der *Vorgänge* und/oder *Ereignisse* sowie deren *Anordnungsbeziehungen* (Abhängigkeiten) darstellt [DIN87b]. Ein Vorgang ist darin ein *Ablaufelement, das ein bestimmtes Geschehen beschreibt. Hierzu gehört auch, daß Anfang und Ende definiert sind* [DIN87a]. Ein Vorgang hat damit auch eine Dauer. Ein Ereignis ist ein *Ablaufelement, das das Eintreten eines bestimmten Zustands beschreibt* [DIN87a] und eine Anordnungsbeziehung ist eine *quantifizierbare Abhängigkeit zwischen Ereignissen oder Vorgängen*.

Die Netzplantechnik umfasst eine Reihe von Methoden, das sind unter anderem CPM[7], PERT[8] oder MPM[9] [Bur02, S.113ff]. Sie unterscheiden sich in ihren Schwerpunkten und der Darstellung: Die MPM stellt Vorgangsknoten-Netzpläne dar: Im Graphen ist jeder Vorgang ein Knoten und die Anordnungsbeziehungen sind die Kanten. Ereignisse werden nicht explizit betrachtet [Bur02, S.113ff]. Im Gegensatz dazu sind etwa bei PERT Ereignisse als Knoten und Vorgänge als Kanten modelliert.

Im Folgenden wird die Terminplanung mit der Metra-Potential-Methode (MPM) betrachtet, da sie in Europa verbreitet ist [Bur02, S.113ff] und in der Literatur empfohlen wird [WM04, S.150]. In einigen deutschsprachigen Lehrbüchern, etwa [HHMS04, S.63] wird nur MPM als Netzplantechnik vorgestellt.

Bei der Terminplanung werden die Arbeitspakete durch einen oder mehrere Vorgänge (Aktivitäten

[7] Critical Path Method
[8] Program Evaluation and Review Technique
[9] Metra-Potential-Methode

[PMI96, S.171] [HHMS04]) abgebildet[10]. Vorgänge tragen zum Ergebnis eines Arbeitspakets bei oder erbringen das komplette Ergebnis. So könnte das Arbeitspaket 'Kundenverwaltung implementieren' in die Vorgänge 'Quelltext für Kundenverwaltung erstellen', 'Review des Quelltextes und der Kommentare' und 'Kundenverwaltung testen' zerlegt werden.

Die Abhängigkeiten zwischen den Vorgängen dienen als Voraussetzung für die Planung, da sie Beschränkungen für die zeitliche Abfolge darstellen. Unabhängige Vorgänge können parallel durchgeführt werden. Bestehen Abhängigkeiten, muss der Ablauf (teilweise) sequentiell sein. So sollte beispielsweise der Vorgang 'Kundenverwaltung testen' erst nach dem Vorgang 'Kundenverwaltung implementieren' begonnen werden, während 'Bestellverwaltung testen' möglicherweise parallel dazu durchgeführt werden kann. Die Netzplantechnik unterscheidet allgemein vier Arten von Abhängigkeiten, diese werden *Anordnungsbeziehungen* genannt, zwischen einem Vorgang (dem Vorgänger) und einem davon abhängigen Nachfolger: Normalfolge, Anfangsfolge, Endfolge und Sprungfolge. Die Normalfolge ist definiert als *Anordnungsbeziehung vom Ende eines Vorgangs zum Anfang seines Nachfolgers* [DIN87a, S.4]. Die Normalfolge und die anderen drei Begriffe sind in Abbildung 4.3 dargestellt:

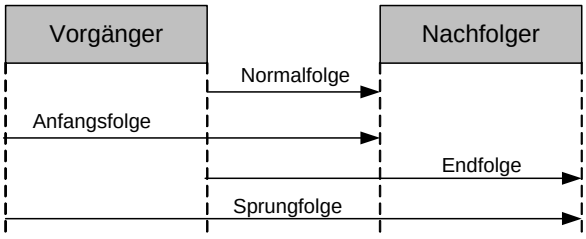


Abbildung 4.3: Darstellung der möglichen Anordnungsbeziehungen nach DIN 69900 [DIN87a, S.4]

Die MPM nutzt Informationen über Abhängigkeiten der Vorgänge und deren Dauer aus, um Anfangs- und Endtermine sowie eventuelle Puffer für die zusammenhängenden Vorgänge automatisch zu berechnen. Der früheste mögliche Endtermin der gesamten Vorgangsfolge kann so berechnet werden. Solche Berechnungsverfahren werden von typischen Werkzeugen des Projektmanagements angeboten.

Bei dieser automatischen Berechnung der Termine wird die Dauer eines Vorgangs als fest angenommen. Die Dauer eines Vorgangs hängt in der Regel vom geschätzten Aufwand und der Verfügbarkeit von Mitarbeitern und anderen Einsatzmitteln ab. Ebenso ergeben sich aus der Verfügbarkeit von Einsatzmitteln weitere Einschränkungen für die Anfangs- und Endtermine von Vorgängen.

## Beschreibung von Terminplänen

Für einen Terminplan gibt es verschiedene Beschreibungen. Gebräuchlich sind Tabellen und Balkendiagramme, welche die Vorgänge mitsamt ihrer Anordnungsbeziehungen darstellen. Seltener ist die Darstellung der Vorgänge als gerichteter Graph im Sinne der DIN 69900 Teil 2 [DIN87b]. Darstellungen als Tabelle, Balkendiagramm oder gerichteter Graph finden sich in Werkzeugen wie Microsoft Project wieder.

*Tabellen* beschreiben einen Vorgang in jeweils einer Zeile. Für jeden Vorgang werden Name sowie Anfangs- und Endtermin dargestellt. Weitere Informationen sind Vorgänger- und Nachfolger-Vorgänge sowie zugeordnete Einsatzmittel.

Ein *Balkendiagramm* (Gantt-Diagramm) beschreibt einen Terminplan grafisch. Die x-Achse des Diagramms bildet ein Zeitstrahl, der in der Regel eine Skala mit Kalendertagen als kleinster Einheit hat. Die Vorgänge sind untereinander entlang der y-Achse angeordnet. Jeder Vorgang wird durch

[10] Alternativ kann das Ergebnis eines Arbeitspakets durch ein Ereignis modelliert werden, wie in PERT.

einen Balken dargestellt, der auf der x-Achse am Starttermin des Vorgangs anfängt und am Endtermin aufhört. Anordnungsbeziehungen zwischen Vorgängen werden über Pfeile dargestellt. Die Abbildung 4.4 zeigt ein Beispiel für die Darstellungen eines Terminplans durch Tabelle und Balkendiagramm:

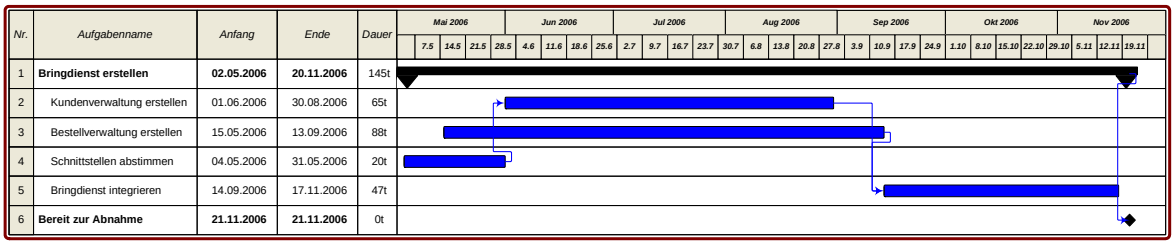


Abbildung 4.4: Beispiel für die Darstellungen eines Terminplans durch Tabelle und Balkendiagramm

## Einsatzmittelplanung (Ressourcenplanung)

Die Zuordnung der Arbeitspakete und Vorgänge zu Personen oder Teilteams geschieht im Rahmen der Einsatzmittelplanung[11]. Die Zuordnung geschieht abhängig von der Verfügbarkeit der Personen und Teilteams und anderen Faktoren, die im Folgenden dargestellt werden. Die Einsatzmittelplanung und die Terminplanung geschehen damit iterativ, denn die Verfügbarkeiten der Ressourcen bestimmen die Termine und die Zeitdauern der Vorgänge im Terminplan und die Abhängigkeiten der Vorgänge im Terminplan bestimmen die Reihenfolge, in der die Vorgänge bearbeitet werden müssen.

Eine erste Einsatzmittelplanung kann auf der Grundlage der Arbeitspakete des Projektstrukturplans geschehen. Ressourcen werden den Arbeitspaketen zugeordnet, diese werden dann im Rahmen der Terminplanung zeitlich geplant. Im Folgenden wird dieses Vorgehen unterstellt.

Für das oben dargestellte Arbeitspaket 'Kundenverwaltung implementieren' wird beispielsweise ein Aufwand von sieben Bearbeitermonaten geschätzt und es soll von einem Team aus Stuttgart bearbeitet werden. Besteht dieses Team aus einer Person, ist die Dauer des entsprechenden Vorgangs im Plan mindestens sieben Monate. Wenn das Team aus sieben Mitarbeitern besteht und alle zu 100% verfügbar sind, könnte der Vorgang theoretisch[12] in einem Monat bearbeitet werden.

### Fähigkeiten (Skills) und Erfahrung

Die Kompetenzen und die Erfahrungen der Teams und Mitarbeiter spielen eine wichtige Rolle[13]. Häufig spezialisieren sich Mitarbeiter auf bestimmte technische oder fachliche Themen, wie etwa die Implementierung grafischer Oberflächen oder den Datenbankentwurf.

Um ein Arbeitspaket zu bearbeiten, sind für das Arbeitspaket spezifische Fähigkeiten erforderlich. Den Arbeitspaketen sollten daher Teams oder Mitarbeiter zugeordnet werden, welche über die erforderlichen Fähigkeiten in ausreichendem Umfang verfügen. Bestenfalls sind Erfahrungen in Vorgängerprojekten im Team vorhanden. Fehlen den Mitarbeitern die notwendigen Kompetenzen, müssen diese vor oder während der Bearbeitung des Arbeitspakets erworben werden. Dafür müssen Aufwände für Schulungen, die Lernkurve und die Qualitätssicherung eingeplant werden.

[11] Einsatzmittel sind *Personal und Sachmittel, die zur Durchführung von Vorgängen, Arbeitspaketen oder Projekten benötigt werden.*[DIN87d]

[12] Im unwahrscheinlichen Fall, dass sich das Arbeitspaket ohne Reibungsverluste in sieben Teile zerlegen lässt.

[13] DeMarco stellt Mitarbeiter und Teams in das Zentrum des Projektmanagements. Seine Grundsätze lauten: (1) *Wähle die richtigen Leute aus*, (2)*Betrauen Sie die richtigen Mitarbeiter mit den richtigen Aufgaben*, (3) *Motivieren Sie die Mitarbeiter* und (4) *Helfen Sie den Teams, durchzustarten und abzuheben* [DeM05, S.24f]

Brügge und Dutoit empfehlen eine Fähigkeitsmatrix [BD04, S.602f] aufzustellen, welche die Aufgaben innerhalb des Projektes in ihren Zeilen, den verfügbaren Mitarbeitern (Spalten) gegenüberstellt. In die Matrix werden dann die Eignungen der Mitarbeiter für die Aufgaben beurteilt. Diese Darstellung soll es erleichtern, die Aufgaben den richtigen Mitarbeitern zuzuordnen.

#### Teamaufbau

Große Projekte sollten mit einem kleinen Team begonnen werden. Dieses Team schafft über den Architekturentwurf und die Planung die Grundlage dafür, dass ein IT-System mit einem großen Team arbeitsteilig entwickelt werden kann. DeMarco beschreibt dies so: *Wenn die Arbeit vor der Fertigstellung des Entwurfs auf ein großes Team verteilt wird, wird darauf verzichtet, die Schnittstellen zwischen den Mitarbeitern und Arbeitsgruppen zu minimieren. Die Folgen sind: größere wechselseitige Abhängigkeit, mehr Besprechungen, Redundanz und Frustration* [DeM05, S.223].

### Strategien zur Verteilung der Arbeitspakete auf Teams

Für die Verteilung der Arbeitspakete auf die vorhandenen Entwicklungsteams gibt es mehrere Strategien, bei denen der bereits erwähnte Architekturentwurf und das Vorgehensmodell eine wichtige Rolle spielen:

**Komponenten der logischen Architektur** : Die Arbeitsverteilung auf die Teilteams kann entlang der Grenzen der Komponenten innerhalb einer logischen oder technischen Architektur geschehen. Jedes Team ist für eine Komponente verantwortlich und spezifiziert, entwirft, testet und dokumentiert diese, abhängig von den Festlegungen des Vorgehensmodells. Bei einem Client/Server-System kann beispielsweise ein Client-Teilteam und ein Server-Teilteam gebildet werden. Alternativ können den Teams jeweils die fachlichen Subsysteme zugeordnet werden [BD04, S. 612]. Der Zusammenhang zwischen Architektur und Projektplanung wird in Kapitel 8 wieder aufgegriffen.

**Phasen des Vorgehensmodells** : Die Arbeitsverteilung kann entlang der Phasengrenzen des Vorgehensmodells erfolgen. Jedes Team ist für eine Phase verantwortlich. Ein Team schreibt beispielsweise die Spezifikation, ein Team implementiert und ein weiters Team ist für die Tests verantwortlich. Dieses Vorgehen ist für Offshoring-Projekte denkbar: Die Spezifikation entsteht beim Kunden in Deutschland vor Ort – dort findet auch die Qualitätssicherung statt – und die eigentliche Entwicklung geschieht beispielsweise in Indien.

Welche Strategie ausgewählt wird und wie die Arbeitspakete auf Teams und Mitarbeiter verteilt wird, hängt unter anderem von organisatorischen Gegebenheiten, den in den Teams vorhandenen Kompetenzen und vorangegangenen Projekten ab [Car99, S. 127ff]. Wichtig sind auch die Kosten für eine Arbeitsstunde, die Verfügbarkeit der Mitarbeiter und die räumliche Nähe zum Kunden. Häufig sind Mischformen anzutreffen, etwa die Aufteilung der Arbeitspakete entlang der Komponentengrenzen, der Integrations- und Systemtest wird jedoch durch ein zentrales Testteam übernommen oder die Wartung und Weiterentwicklung des IT-Systems geschieht durch einen Offshore-Partner.

### Kostenplan

Im Verlauf eines Projektes fallen Kosten an, die ebenso wie die Termine und Einsatzmittel geplant und kontrolliert werden müssen. Kosten fallen beispielsweise für eigene Mitarbeiter, Verwaltung, IT-Infrastruktur, Mieten und für Dienstreisen an. IT-Projekte sind personalintensiv, sodass die Personalkosten den größten Teil der Gesamtkosten ausmachen [EHR00, S.25]. Über die Auswahl der Mitarbeiter in einem Projekt können die Gesamtkosten daher entscheident beeinflusst werden. Die

geringeren Personalkosten sind beispielsweise eines der wichtigsten Argumente für das Offshoring von Entwicklungsprojekten [Tau04]. Bei der Kostenplanung ist die Mitarbeiterauswahl daher ein wichtiger Faktor.

Zur Verfolgung der anfallenden Kosten werden in der Regel Konten definiert, auf welche die verschiedenen anfallenden Kosten gebucht werden. Die Definition der Kontenstruktur ist ein Teil der Kostenplanung. Jedes Konto kann mit einem Budget versehen werden, um die anfallenden Kosten nach oben zu begrenzen. Dies geschieht etwa bei Reisekosten. Auf die definierten Konten können die geleisteten Arbeitsstunden der Mitarbeiter gebucht werden, so dass die angefallenen Personalkosten, direkt verfolgt werden können.

Um die Liquidität des Auftragnehmers zu erhalten und die Risiken für den Auftragnehmer kalkulierbarer zu machen, können Abschlagszahlungen vereinbart werden, die der Auftraggeber zu bestimmten Zeitpunkten zahlt. Diese Abschlagszahlungen werden in der Kostenplanung berücksichtigt.

### Projektplan

Der Projektplan fasst alle Planungsinformationen, also mindestens den Projektstrukturplan sowie die Termin- und Einsatzmittelplanung zusammen.

**Definition 4.4 (Projektplan)**
Der Projektplan ist ein Dokument. Er beschreibt ein geplantes Projekt und dessen Durchführung. Zum Projektplan gehören ein Projektstrukturplan sowie ein entsprechender Termin- und Einsatzmittelplan. Häufig ist auch ein Kostenplan enthalten.

## 4.4 Kontrolle

Im Verlauf eines Projektes muss die Umsetzung des Plans regelmäßig kontrolliert werden, um auftretende Probleme und Störungen rechtzeitig zu erkennen und Gegenmaßnahmen rechtzeitig einzuleiten. Störungen können sich auf die Formalziele, wie Termine, Kosten- oder Aufwandsrahmen auswirken oder auf die Sachziele wie die Qualität der Ergebnisse. Beispiele für Störungen sind:

- Arbeitspakete dauern länger als geplant oder verzögern sich z.B. weil eine Zulieferung (Beistellung) des Auftraggebers ausbleibt oder ein Mitarbeiter krank wird oder kündigt.
- Arbeitspakete erfordern höheren Aufwand als geplant, da Anforderungen unterschätzt wurden und damit die Schätzung zu niedrig war.
- Ergebnisse liegen nicht in der vereinbarten Qualität vor, so können etwa zu viele kritische Fehler in einer Software verblieben sein oder ein Dokument ist nicht plausibel und besteht eine Qualitätsprüfung nicht.
- Funktionsumfänge sind wegen technischer Probleme oder Abstimmungsproblemen nicht umsetzbar.
- Geplante Einsatzmittel (z.B. Räume, Testrechner und -daten) stehen nicht zur Verfügung.
- Anforderungen und ihre Prioritäten ändern sich während des Projektes. Das kann neue Arbeitspakete notwendig machen, andere werden überflüssig oder die Inhalte von Arbeitspaketen ändern sich.

Im Rahmen der Kontrolle eines Projektes werden regelmäßig Ist-Informationen zu den geplanten Arbeitspaketen erhoben, daraus können Kennzahlen wie der Fertigstellungsgrad errechnet werden.

Die Erhebung der Informationen geschieht in der Regel über Berichte [HHMS04] oder Statusmeetings [BD04, S.121]. Kontrolliert werden Formalziele wie Termine und Kostenziele bzw. Aufwandsziele [WM04, S. 188]. Für jedes Arbeitspaket oder für Gruppen von Arbeitspaketen (s.u.) werden im Rahmen der Projektkontrolle folgende Informationen erfasst:

- Tatsächlicher Anfangstermin
- Tatsächlicher Endtermin
- Verbrauchter Aufwand in Personentagen und angefallene Kosten
- Restaufwand und Fertigstellungsgrad

Termine und mindestens die Meldung des Abschlusses eines Arbeitspakets können über die Bereichte oder Statusmeetings erfolgen. Zur Erfassung des verbrauchten Aufwandes können Daten aus der Zeiterfassung der Mitarbeiter verwendet werden, hierzu ist jedoch ein Zugang zu diesen Daten erforderlich und eine zu den Arbeitspaketen passende Kontenstruktur. Da in der Regel nicht für jedes Arbeitspaket mit wenigen Tagen Aufwand ein eigenes Konto eingerichtet wird, erfolgt die Aufwandskontrolle über grobteiligere Arbeitspakete oder Mengen von Arbeitspaketen (Themen, Phasen).

Mithilfe der erhobenen Zahlen zu den jeweiligen Arbeitspaketen, soll geprüft werden, ob geplante Termine, Aufwände und andere Projektziele noch eingehalten werden können. Sind die Abhängigkeiten zwischen den Arbeitspaketen dokumentiert, können die terminlichen Konsequenzen verzögerter Arbeitspakete mit den Mitteln der Netzplantechnik direkt berechnet werden. In die Kontrolle müssen auch laufende Arbeitspakete einbezogen werden. Diese sind nur teilweise fertiggestellt.

### Fertigstellungsgrad - Formalziel

Der Fertigstellungsgrad eines Arbeitspakets kann von den Mitarbeitern im Rahmen ihrer Berichte (aus dem Bauch) geschätzt werden [WM04, S.189]. Diese Schätzungen sind jedoch häufig ungenau und es wird über die Zeit anstelle eines linearen Fortschrittsverlaufs eine langsame asymptotische Annäherung gegen 100% beobachtet. Dies ist auch als 90%-Syndrom [HHMS04, S.79] bekannt. Um den Projektfortschritt genauer beurteilen zu können, kann anstelle dieser Beurteilung eine Restaufwandsschätzung [Sie02b, S.333] durchgeführt werden, in der für jedes Arbeitspaket der noch zu erbringende Aufwand bis zur Fertigstellung geschätzt wird. Auf der Grundlage des Restaufwands kann der Fertigstellungsgrad für jeden Vorgang, jedes Arbeitspaket oder das gesamte Projekt berechnet werden [HHMS04, S.79]:

$$Fertigstellungsgrad = \frac{verbrauchter Aufwand}{verbrauchter Aufwand + Restaufwand}$$

Lannes empfiehlt, eine Restaufwandsschätzung regelmäßig *zu allen wichtigen Meilensteinen [...] oder zu bestimmten Zeiten (z.B. alle sechs Monate)* [Sie02b, S.333] durchzuführen.

### Fertigstellungsgrad - Sachziel

Besonderes Problem bei der Erhebung des Projektfortschritts ist die Beurteilbarkeit der entstandenen Dokumente und teilfertigen Software über Reviews und Tests. Wieczorrek und Mertens formulieren dies mit Bezug auf die abschließende Kontrolle des fertigen IT-Systems so [WM04, S.190]: *Alle dazwischen liegenden Kontrollstufen sind lediglich Interimskontrollen, die auf Aufzeichnungen von Gedanken, Planungen, Konzeptionen, Schätzungen usw. beruhen. Im Prinzip alles visualisierte gedankliche Modelle des Systems. Echte, einer harten Überprüfung standhaltende Ergebnisse des entwickelten Mensch-Maschine-Systems liefert erst der Abschlusstest auf einem Rechner.*

Dies unterstreicht, dass die Kontrolle des Projekts mit einer fortlaufenden Qualitätsplanung und Qualitätssicherung einhergehen muss, um fortlaufend auch die Erreichung der *Sachziele* (insbesondere der Qualitätsanforderungen) kontrollieren zu können. Die Ist-Qualität der Zwischenergebnisse ist ein Mittel, den Grad der Fertigstellung so gut es geht zu beurteilen. Bei Problemen in der Erreichung der Qualitätsanforderungen sind wie bei Terminüberschreitungen steuernde Maßnahmen erforderlich.

## 4.5 Steuerung

Steuerndes Eingreifen ist in einem Projekt spätestens dann erforderlich, wenn das durch die Kontrolle ermittelte *Ist* wesentlich von dem geplanten *Soll* abweicht. Steuerung ist notwendig, um Budget und Termine im Rahmen der vorgegebenen Qualitätsanforderungen einhalten zu können. Bei wesentlichen Abweichungen von der Planung wird eine Neuplanung des Projekts [Sie02b, S.333] bzw. eine wesentliche Anpassung der Planung [EHR00, S. 70] empfohlen.

Eine Anpassung der Planung ist insbesondere dann erforderlich, wenn sich Anforderungen oder Prioritäten ändern, die geplante oder durchgeführte Arbeitspakete betreffen. Dies geschieht im Rahmen des Änderungsmanagements. Inhalte und Methoden des Änderungsmanagements werden in der vorliegenden Arbeit nicht betrachtet.

Maßnahmen zur Steuerung ergeben sich über die Veränderung der Größen Leistungsumfang, Budget, Termin sowie Qualität:

- Geplante Arbeitspakete werden weggelassen, gekürzt oder in die nächste Stufe verschoben. Das kann einerseits bedeuten, dass der Lieferumfang verringert wird, also Anwendungsfälle nicht mehr umgesetzt werden, oder andererseits dass an der Qualität gespart wird, indem etwa Testaufwände reduziert werden.

- Neue Mitarbeiter werden in das Projekt aufgenommen. Die Einsatzmittel werden also erhöht. Dies kostet jedoch in der Regel mehr Budget und ist kann den Fertigstellungstermin gefährden[14].

- Der Fertigstellungstermin wird verschoben. Damit verschiebt sich auch die Einführung eines neuen oder geänderten IT-Systems. Die Projektergebnisse können erst später genutzt werden. Zusätzlich erhöht sich in der Regel dadurch das benötigte Budget.

Weitere Maßnahmen bei Abweichungen können auch die Umverteilung von Aufgaben, Mehrarbeit oder die Einführung neuer Werkzeuge sein. Denkbar sind auch Maßnahmen zur Motivation der Mitarbeiter und zum Teambuilding.

Werden Probleme in der Qualität der Ergebnisse erkannt, sollten Maßnahmen zur Qualitätsverbesserung ergriffen werden. Einerseits ist die Behebung der gefundenen Probleme erforderlich, anderseits können beispielsweise Schulungen für die Mitarbeiter oder Maßnahmen zur Verbesserung der Prozesse gestartet werden.

Welche dieser Maßnahmen ergriffen wird, liegt im Ermessen des Projektleiters oder der entsprechenden Steuerungsgremien, etwa dem Lenkungsausschuss [WM04, S.31]. Die Steuerungsmaßnahmen werden in der Regel vom (Teil-)Projektleiter an die Mitarbeiter in Einzelgesprächen oder regelmäßigen Meetings kommuniziert und besprochen. Projektsteuerung bedeutet dabei immer Menschen zu führen.

[14] vgl. beispielsweise das Brooks'sche Gesetz *Der Einsatz zusätzlicher Arbeitskräfte bei bereits verzögerten Software-Projekten verzögert sie nur noch mehr.* [Bro03, S.26]

## 4.6 Zusammenfassung

Die Abbildung 4.5 zeigt die Begriffe des Projektmanagements und ihre Zusammenhänge: Ein Projektplan ist ein Modell eines Projektes. Er umfasst mindestens einen Projektstrukturplan und einen Terminplan (Aktivitätszeitplan), implizit ist darin auch die Einsatzmittelplanung enthalten, da zu Arbeitspaketen bzw. Vorgängen die notwendigen Ressourcen dokumentiert sind. Der Projektstrukturplan besteht im Wesentlichen aus Arbeitspaketen und der Terminplan besteht im Wesentlichen aus Vorgängen. Im Terminplan werden die zeitlichen bzw. logischen Abhängigkeiten zwischen den Vorgängen über Anordnungsbeziehungen dokumentiert.

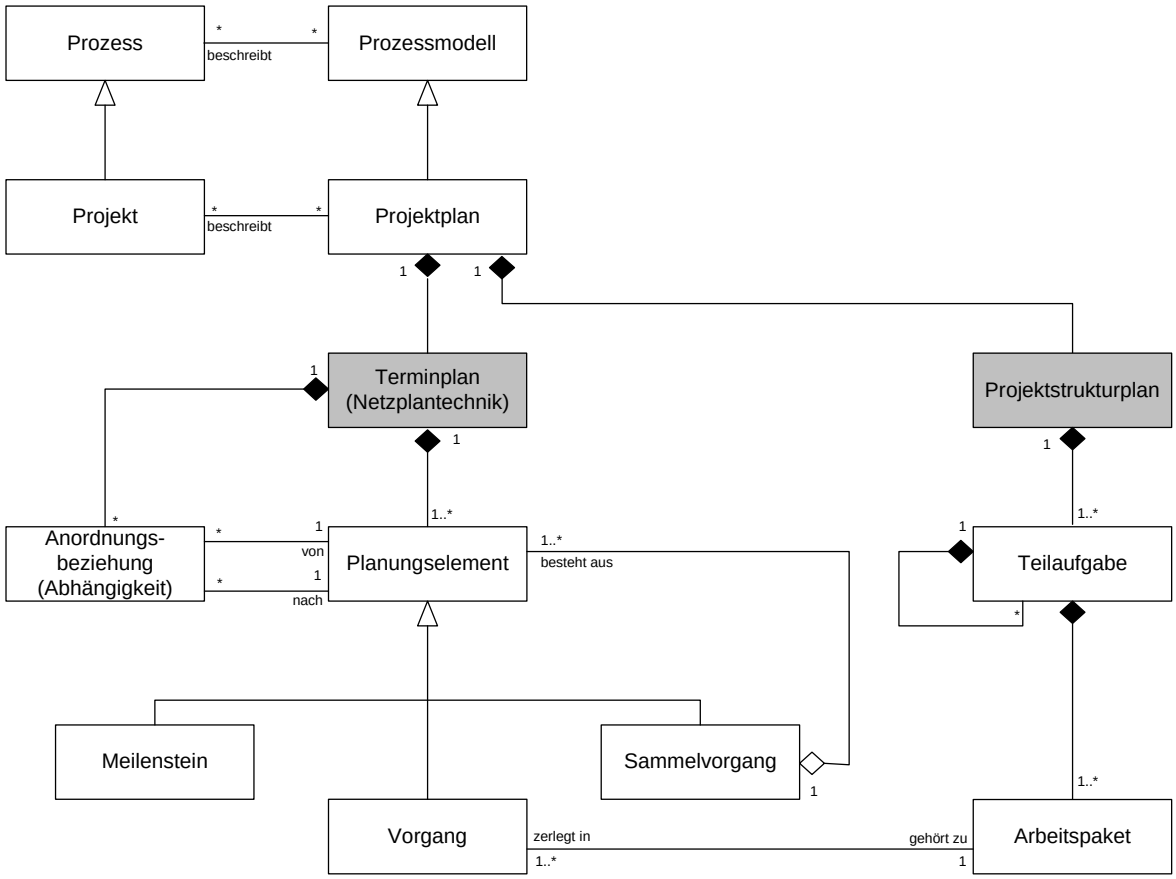


Abbildung 4.5: Projektbegriff

Vorgänge sind auf der Zeitachse verplante Arbeitspakete, also deren Umsetzung im Terminplan.

Der Kostenplan wird in der vorliegenden Arbeit nicht berücksichtigt.

Die hier definierten Begriffe bilden die Grundlage für die Kapitel 7 und 8. Diese machen Vorschläge zur Unterstützung der Projektleiter und Architekten in der Projektplanung, -kontrolle und -steuerung, um die hier identifizierten Probleme zu lindern.

# Teil II

# Architekturtheorie

# Kapitel 5

# Logische Architektur als gerichteter Multigraph

Die in der Literatur vorgeschlagenen Sichtenkonzepte[1] werden in der Regel pragmatisch begründet. Sie werden selten systematisch betrachtet. Zusatzinformationen zu den Komponenten und Konnektoren der Architektur, wie etwa die Erstellungskosten, werden bislang nicht berücksichtigt. Ebenso fehlt in der Regel eine dokumentierte Verbindung zwischen den in den Anwendungsfällen festgelegten Anforderungen und den Komponenten und Konnektoren die diese implementieren. Beides ist jedoch für die Kommunikation innerhalb des Projektes und zu den anderen Stakeholdern wünschenswert.

Dieses Kapitel schlägt eine einfache Theorie zur Beschreibung logischer Architekturen auf der Grundlage der Graphentheorie vor. Die Theorie hat das Ziel, Strukturen in System- und Software-Architekturen untersuchbar zu machen und verschiedene Varianten der Erzeugung von Architektursichten zu analysieren und zu beschreiben. Dabei arbeitet die Theorie auch mit Zusatzinformationen und stellt eine Verbindung zu den Anwendungsfällen her.

Abschnitt 5.1 gibt einen Überblick über die Formalisierung logischer Architekturen als gerichtete Multigraphen. Die Abschnitte 5.2 und 5.3 formalisieren Komponenten als Knoten und Konnektoren als Kanten innerhalb eines Multigraphen der die Kommunikationsstruktur darstellt. Der Multigraph wird auch Konfiguration genannt. Die Verbindung zu den Anforderungen wird in Abschnitt 5.4 hergestellt: Nutzungsszenarios stellen dar, welche Komponenten und Konnektoren einen Anwendungsfall umsetzen. Abschnitt 5.5 stellt dar, wie Komponenten und Konnektoren Zusatzinformationen zugeordnet werden können. Eine logische Architektur fasst Konfiguration, zugeordnete Attribute und die Nutzungsszenarios zusammen, dies ist Thema von Abschnitt 5.7. Die wichtigsten Elemente der Theorie werden schließlich in Abschnitt 5.8 zusammengefasst.

## Übersicht



[1] vgl. Abschnitt 3.4

## 5.1 Formalisierung logischer Architekturen

Wie in Abschnitt 3.3 dargestellt, entstehen im Laufe eines Software-Entwicklungsprozesses mehrere verschiedene Arten von Architekturen. Die vorliegende Arbeit betrachtet ein Modell, das hier als *logische Architektur* bezeichnet wird. Seine Elemente sind logische Komponenten, welche neu entwickelt, wiederverwendet oder zugekauft werden. Komponenten können in Hardware und/oder Software umgesetzt werden. Die Erstellung einer logischen Komponente kann einem Teilteam im Rahmen eines oder mehrerer Arbeitspakete zugewiesen werden.

Komponenten tauschen über Konnektoren Nachrichten aus. Wenn zwei Komponenten mit den Namen *Bestellverwaltung* und *Kundenverwaltung* in der logischen Architektur über einen Konnektor verbunden sind, bedeutet das: Die *Bestellverwaltung* kann der *Kundenverwaltung* auf irgendeine Weise Nachrichten schicken. Die logische Architektur lässt offen, wie die Konnektoren umgesetzt werden.

Die Beschreibung der Kommunikationsstruktur wird auch als *Konfiguration*[2] bezeichnet. Die logische Architektur ist eine *statische* Darstellung des IT-Systems. Es werden Komponenten betrachtet, jedoch nicht ihre Instanzen zur Laufzeit. Die Darstellung geht von IT-Systemen aus, denen zur Laufzeit keine fremde Komponente und kein fremder Konnektor hinzugefügt wird. Die Struktur der Konfiguration ist zur Laufzeit unveränderlich.

### Darstellung der Konfiguration als gerichteter Multigraph

Die Konfiguration der logischen Architektur kann als gerichteter Multigraph modelliert werden: Jede Komponente ist ein Knoten in dem Graphen und jeder Konnektor ist eine gerichtete Kante. Gibt es mehrere Konnektoren zwischen zwei Komponenten, werden entsprechend viele Kanten modelliert.

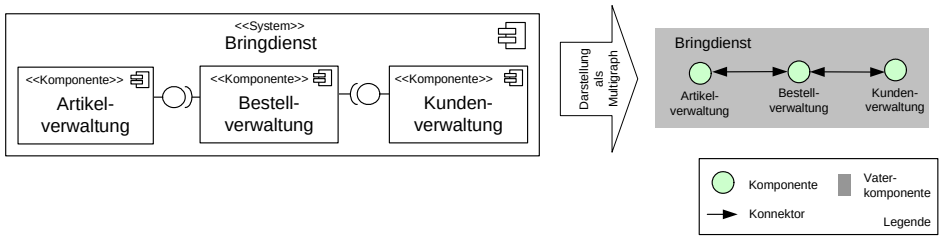


Abbildung 5.1: Darstellung der erweiterten Konfiguration als Multigraph

Die Abbildung 5.1 zeigt ein Beispiel für die Darstellung der Konfiguration eines Bringdienstsystems. Die Abbildung zeigt links und rechts das IT-System *Bringdienst*, das aus den drei Komponenten *Bestellverwaltung*, *Artikelverwaltung* und *Kundenverwaltung* besteht. Die *Bestellverwaltung* kann der *Kundenverwaltung* und der *Artikelverwaltung* Nachrichten schicken, beide können darauf antworten.

Links sind die Komponenten mit den Sprachmitteln der UML dargestellt und die Konnektoren sind als Paar von exportierten und importierten Schnittstellen modelliert. Derselbe Sachverhalt ist rechts durch einen gerichteten Multigraphen dargestellt: Zwischen der *Bestellverwaltung* und der *Artikelverwaltung* bzw. der *Kundenverwaltung* gibt es einen Konnektor in jeder Richtung. Dies ist grafisch durch einen Pfeil mit zwei Spitzen dargestellt. Die Architekturdarstellung enthält rechts insgesamt vier Konnektoren. Der hierarchische Aufbau des Multigraphen ist über die graue Fläche angedeutet.

Architekturen, bei denen an einer Kommunikation mehrere Partner beteiligt sind, etwa Publish-and-Subscribe Architekturen [HW03] werden zunächst nicht betrachtet[3].

[2] vgl. Definition 3.7, nicht zu verwechseln mit der Konfiguration im Konfigurationsmanagement

[3] Dies könnte entweder durch eine Kante für jeden möglichen Kommunikationsweg oder durch Hyperkanten, denen mehr als zwei Knoten zugeordnet werden können, modelliert werden.

Das Verhalten der logischen Komponenten und Konnektoren wird nur über Kommentare beschrieben. Beide sind Elemente der ersten Grobentwürfe zum Projektbeginn, darin wird das nur IT-System skizziert, aber noch nicht im Detail entworfen. Das Verhalten und Details zu den Schnittstellen werden erst in Verfeinerungen der logischen Architektur, nämlich der technischen Architektur, genauer betrachtet[4].

### Wozu eine Formalisierung als gerichteter Multigraph?

Logische Architekturen werden mit den Mitteln der Graphentheorie formalisiert, untersucht und verändert. Dies hat folgende Vorteile:

- Ziel der Theorie ist die Darstellung, Untersuchung und Veränderung der Strukturen in einer logischen Architektur. Die Graphentheorie ist für derartige Aufgaben besonders geeignet [Jun94].
- Die Strukturen können direkt als gerichteter Graph abgebildet werden, ohne den Aufwand, den ein umfangreiches Metamodell zur Folge hätte, wie etwa das der UML 2.0.
- Abbildungen der Strukturen auf andere Strukturen, etwa der Arbeitspaketstruktur eines Projektplans, können als einfache Gleichungen und mit den Mitteln der Mengenlehre beschrieben werden.
- Ein gerichteter Graph kommt den informellen Architekturdarstellungen als Box-and-Arrow Diagramm besonders nahe.

## 5.2 Komponenten und Konnektoren

Die unendliche Menge $COMPONENT$ sei die Menge aller denkbaren Komponenten und die unendliche Menge $CONNECTOR$ sei die Menge aller denkbaren Konnektoren. Beide Mengen seien disjunkt: $COMPONENT \cap CONNECTOR = \emptyset$. Komponenten und Konnektoren werden zusammen als *Architekturelemente* bezeichnet.

Komponenten und Konnektoren werden vereinfacht als 2-Tupel $(i, n)$ aus einer eindeutigen Id $i$ und einem Namen $n$ modelliert. Die Funktionen $id$ und $name$ liefern für jede Komponente und jeden Konnektor jeweils die Identität bzw. den Namen. Die Werte von $name$ und $id$ sind Bezeichner (Identifier). Die Menge $\mathbb{ID}$ sei die Menge aller gültigen Bezeichner mit $|\mathbb{ID}| = \infty$.

$id : COMPONENT \cup CONNECTOR \rightarrow \mathbb{ID}$.
$name : COMPONENT \cup CONNECTOR \rightarrow \mathbb{ID}$

Jeder Komponente und jedem Konnektor wird eine eindeutige Identität zugewiesen:

$\forall v, w \in COMPONENT \cup CONNECTOR : id(v) = id(w) \Leftrightarrow v = w$

Für $name$ gilt diese Eindeutigkeit nicht[5]:

$\forall v, w \in COMPONENT \cup CONNECTOR : name(v) = name(w) \Leftarrow v = w$

Da $name$ eine Abbildung ist, gilt offensichtlich:
$\forall v, w \in COMPONENT \cup CONNECTOR : id(v) = id(w) \Rightarrow name(v) = name(w)$.

[4] vgl. Abschnitt 3.3

[5] Namen können in logischen Architekturen mehrfach vorkommen, wie die nachfolgenden Beispiele zeigen werden: Dort gibt es etwa die Komponenten *Dialogschicht*, *Anwendungskern* und *Datenhaltung* in jedem Subsystem.

Auf der Menge der Identifier $\mathbb{ID}$ sei ein Konkatenationsoperator $+$ bzw. $\sum$ definiert, mit dem zwei oder mehr Identifier zusammengehängt werden können und eine vollständige Ordnung $\leq$ mit der alle Identifier verglichen werden können. Also $\forall a, b \in \mathbb{ID} : a + b \in \mathbb{ID}$, d.h. der Konkatenationsoperator ist für alle Identifyer definiert und führt nicht aus der Menge der Identifier heraus. $\forall a, b \in \mathbb{ID} : a \leq b \vee b \leq a$, d.h. jedes beliebige Paar von Identifiern kann in eine Ordnung gebracht werden.

Eine mögliche Ausprägung der Menge $\mathbb{ID}$ ist die Menge der Zeichenketten unbegrenzter Länge (mit den Elementen = 'a', 'b' ... 'z'). Der Konkatenationsoperator ist damit gegeben durch die Zeichenkettenkonkatenation (z.B. 'hallo ' + 'welt' = 'hallo welt') und die lexikographische Ordnung wird als Ordnungsrelation verwendet (z.B. 'hallo ' $\leq$ 'hallo welt' $\leq$ 'welt'). Ordnungsrelation und Konkatenationsoperator werden zur Erzeugung von Namen benötigt: Die Namen mehrerer Komponenten und/oder Konnektoren sollen eindeutig zu einem Namen zusammengesetzt werden können.

## 5.3 Konfiguration und Systemkonfiguration

### Konfiguration mit Hierarchiekanten

Die logische Architektur eines IT-Systems kann beschrieben werden durch eine Menge von logischen Komponenten, die in Kommunikationsbeziehungen stehen. Die Kommunikationsbeziehungen können als gerichteter Multigraph $C$ dargestellt werden, wobei die Komponenten die Knoten $V$ und die Konnektoren die Kanten $E$ sind.

Architekturen sind in der Regel hierarchisch aufgebaut: Komponenten können aus anderen Komponenten bestehen. Dies kann über eine eigene Kantenmenge $H \subset V \times V$ dargestellt werden, die einen gerichteten Baum oder allgemeiner einen gerichteten Wald erzeugt. Die Knoten stellen wie im Kommunikationsmultigraphen die Komponenten dar. Eine Kante befindet sich genau dann zwischen zwei Komponenten, wenn eine der beiden Teil der anderen ist[6]. Die Abbildung 5.2 gibt rechts ein Beispiel für die explizite Darstellung mit zwei verschiedenen Kantenmengen. Dabei sind die Kanten, welche den hierarchischen Aufbau darstellen, gestrichelt dargestellt. Konnektoren werden mit durchgehenden Kanten dargestellt. Auf der linken Seite der Abbildung 5.2 ist der hierarchische Aufbau angedeutet. Beide Graphen stellen dieselben Informationen dar.

Die Kommunikationsbeziehungen und der hierarchischen Aufbau werden im Folgenden gemeinsam als *Konfiguration* bezeichnet. Die Konfiguration dient als explizite Strukturbeschreibung eines IT-Systems. Das Tupel $C = (V, E, J, H)$ heißt *Konfiguration*, genau dann, wenn

- $V \subset COMPONENT$ eine endliche Menge von Komponenten ist ($V$ für Vertex),
- $E \subset CONNECTOR$ eine endliche Menge von Konnektoren ist ($E$ für Edge),
- $J : E \rightarrow V \times V$ eine Zuordnungsfunktion ist ($J$ für Junction), die jedem Konnektor ein Paar $(v, w)$ mit $v, w \in V$ von Komponenten zuordnet. Für $J$ gilt die Einschränkung: $J(e) = (v, w) \Rightarrow v \neq w$, d.h. es sind keine Konnektoren erlaubt, die eine Komponente mit sich selbst verbinden. Die Funktion $J$ ist total, d.h. ihr Definitionsbereich ist die gesamte Konnektormenge $E$. Sie ist im Allgemeinen nicht injektiv, da in einer Konfiguration einem Komponentenpaar $(v, w)$ mit $v, w \in V$ mehrere Konnektoren zugeordnet werden können.
- $H \subset V \times V$ ist eine Hierarchierelation ($H$ für Hierarchy). Für $H$ ist der Graph $D := (V, H)$ ein gerichteter Wald. Der gerichtete Graph $D$ ist also kreisfrei (auch $(v, v) \notin H$), und es gibt mindestens eine Wurzel $r \in V$, also eine Komponente, die nicht Teil einer anderen ist: $\exists r \in V$ so, dass $\forall v \in V, v \neq r$ gilt $(v, r) \notin H$.

[6] Ähnliche Vorschläge mit Hierarchie- und Kommunikationskanten finden sich bei Holt [Hol96] oder bei Millner [Mil01]

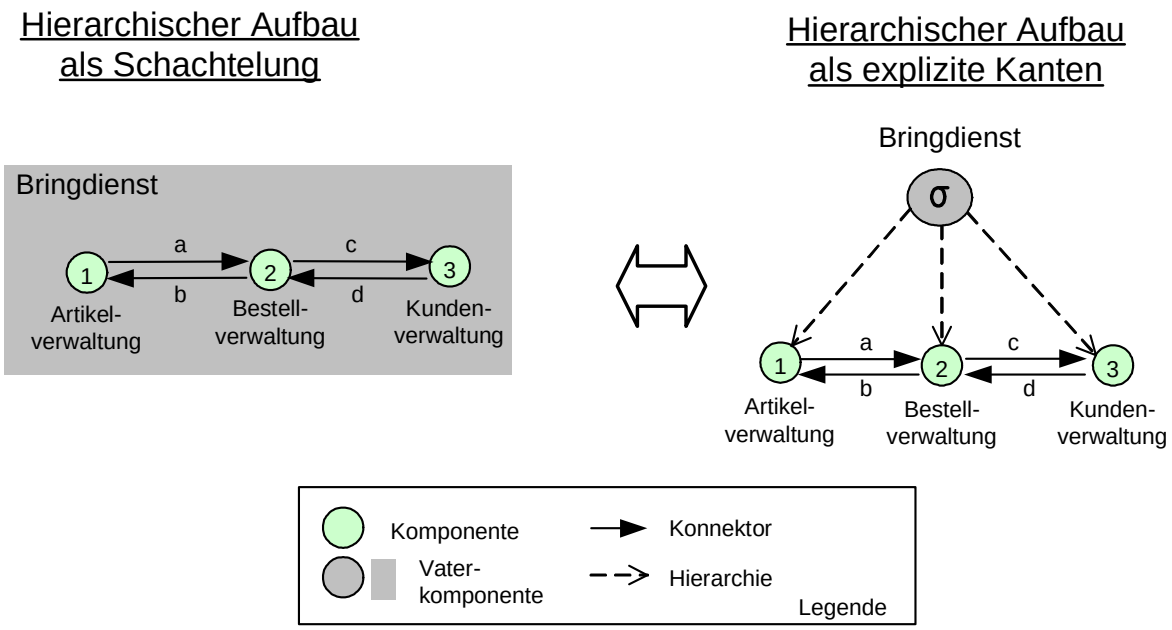


Abbildung 5.2: Zwei Varianten der Darstellung als gerichteter Multigraph

Die Konfiguration verfügt nicht über offene Konnektoren, welche die Konfiguration mit ihrer Umgebung verbinden. Im Multigraphen würde den entsprechenden Kanten der Anfangsknoten oder der Endknoten fehlen. Die Schnittstelle der Konfiguration zu ihrer Umgebung kann nicht dargestellt werden, ohne die Umgebung explizit zu modellieren. Dies wird in der unten dargestellten erweiterten Systemkonfiguration getan. In der Architekturtheorie werden nur geschlossene Systeme modelliert.

Die hier dargestellte Konfiguration ähnelt den aus der Literatur bekannten hierarchischen Graphen[7]: Ein hierarchischer Graph $HG = (G, T)$ besteht aus einem gerichteten Baum $T$, der den hierarchischen Aufbau darstellt und einem gerichteten Graphen $G$, der die Kommunikationsstruktur darstellt. Die Knoten von $G$ sind die Blätter von $T$. Die hier dargestellte Konfiguration ist allgemeiner, da über Konnektoren alle Knoten miteinander verbunden werden können, dies erlaubt eine freiere Modellierung logischer Architekturen.

Die Zuordnungsfunktion $J$ wird in den nachfolgenden Beispielen immer als Menge von Tupeln angegeben. Das Tupel $(a, (1, 2)) \Leftrightarrow J(a) = (1, 2)$ stellt dar, dass der Konnektor $a$ die Komponente 1 mit der Komponente 2 verbindet. Anders ausgedrückt wird $J$ als Tupelmenge $J \subset E \times (V \times V)$ dargestellt.

Die Abbildung 5.2 zeigt die Beispielkonfiguration $K$. Für die Komponenten $V_K$ von $K$ ist jeweils die Id und der Name angegeben. Die Komponente '2' hat beispielsweise den Namen *Bestellverwaltung*. Die Konnektoren $E_K$ werden nur durch ihre Id gekennzeichnet. $K = (V_K, E_K, J_K, H_K)$ ist damit wie folgt definiert:

- $V_K = \{\sigma, 1, 2, 3\}$, wobei die Komponente $\sigma$ das Bringdienstsystem darstellt.
- $E_K = \{a, b, c, d\}$
- $J_K = \{(a, (1, 2)), (b, (2, 1)), (c, (2, 3)), (d, (3, 2))\}$
- $H_K = \{(\sigma, 1), (\sigma, 2), (\sigma, 3)\}$

[7] Vgl. etwa [NL05]

Einen Sonderfall stellt die Konfiguration dar, die genau eine Komponente $v$ umfasst, also $C_v = (\{v\}, E_\emptyset, J_\emptyset, \emptyset)$. Die Hierarchie ist offenbar leer und es sind keine Konnektoren erlaubt (siehe Definition von $J$). Der zweite Sonderfall ist die *leere Konfiguration* $C_\emptyset = (\emptyset, \emptyset, J_\emptyset, \emptyset)$. Eine Konfiguration stellt also nicht notwendig ein vollständiges IT-System dar. Auch für eine logische Komponente können Konfigurationen angegeben werden.

## Diskussion: Hierarchie, Konnektoren und Information Hiding

Bei der Formulierung einer Konfiguration mit hierarchisch zusammengesetzten Komponenten können mehrere Varianten für die Verwendung von Konnektoren gewählt werden. Zwei Varianten werden in Abbildung 5.3 dargestellt: Links werden Konnektoren zwischen Blättern des Hierarchiebaumes erlaubt, ohne dass die Kommunikation über die entsprechenden Vaterkomponenten abgewickelt werden muss. Rechts können Konnektoren nur unter Berücksichtigung der Hierarchie definiert werden. Wenn zwei Komponenten zwischen über die Grenzen von Teilbäumen der Hierarchie hinweg kommunizieren wollen, schickt die eine Komponente die Nachricht solange über Konnektoren die Hierarchie hoch, bis eine gemeinsame Vaterkomponente beider erreicht ist und von dort aus wird die Nachricht wieder die Hierarchie hinab zur zweiten Komponente gesendet. Ports erleichtern diese Darstellung, da über sie dargestellt werden kann, welche Konnektoren von außen mit den Konnektoren innen verbunden sind. Ports stellen damit das Geheimnisprinzip sicher, da nur sie und nicht mehr die Teilkomponenten von außen adressiert werden.

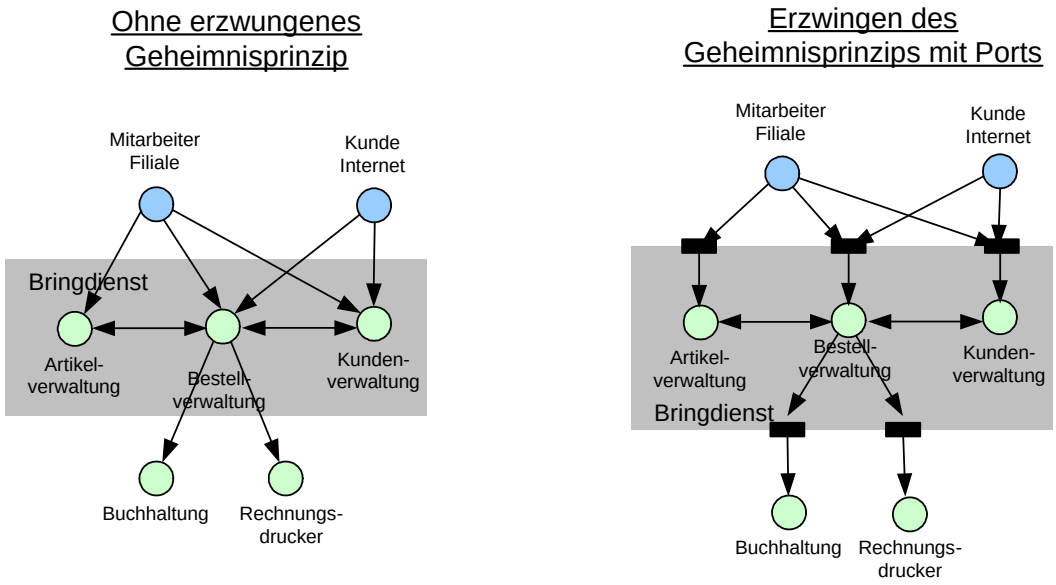


Abbildung 5.3: Hierarchie und Konnektoren

Die linke Variante kommt offenbar mit weniger Konnektoren und ohne Ports aus, da keine Konnektoren modelliert werden müssen, welche die Hierarchie überwinden müssen. Sie gibt jedoch das nach Definition 3.4 geforderte Geheimnisprinzip für Komponenten auf, denn alle Komponenten können ihre Bestandteile offen legen und die Schnittstellen der Komponenten sind nicht zwingend explizit ausgeführt.

Die vorliegende Arbeit verfolgt die linke Variante weiter, da die rechte Variante eine Einschränkung der Linken darstellt. In der praktischen Arbeit könnte diese Einschränkung durch ein Nutzungskonzept und Reviews sichergestellt werden: Ports können als spezielle Teilkomponenten modelliert werden[8].

[8] vgl. Fassaden-Muster [GHJV94]

## Sonderfall: mehrere Vaterkomponenten

Vom Hierarchiewald $H \subset V \times V$ wurde gefordert, dass er mindestens eine Wurzel hat und dass der gerichtete Graph $G := (V, H)$ kreisfrei ist. Dies erlaubt einen Sonderfall, der von strengen Hierarchien abweicht: Eine Komponente kann logischer Teil mehrerer Vaterkomponenten sein. Beispiel für eine solche Komponente ist eine Bibliothek, die von mehreren Komponenten gemeinsam verwendet wird, jedoch logisch als Teil dieser Komponenten verstanden wird.

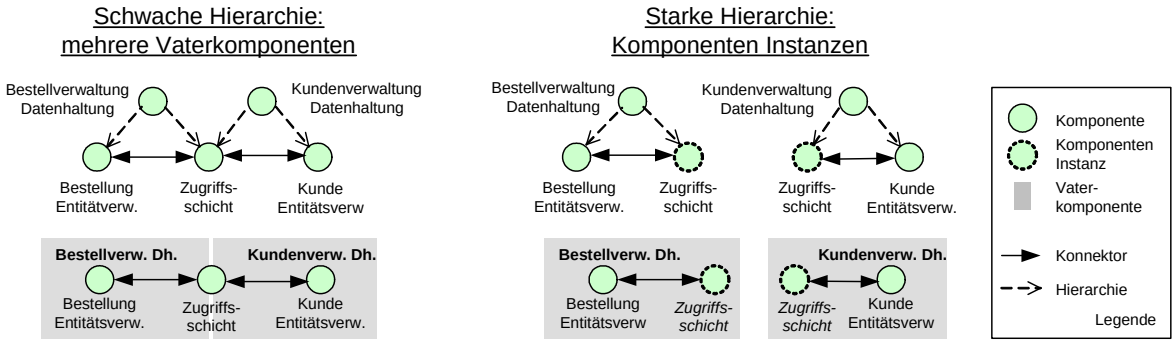


Abbildung 5.4: Komponente mit mehreren Vaterkomponenten

Die Abbildung 5.4 zeigt ein Beispiel für diese Situation: Die Komponenten *Kundenverwaltung Datenhaltung* und *Bestellverwaltung Datenhaltung* werden in ein System integriert, beide verwenden die Komponente *Zugriffsschicht*. Dies lässt sich auf zwei Arten darstellen: Entweder arbeitet jede Komponente auf einer Kopie der *Zugriffsschicht* (rechts angedeutet) oder die Komponente *Zugriffsschicht* hat die beiden Vaterkomponenten *Kundenverwaltung Datenhaltung* und *Bestellverwaltung Datenhaltung*.

Die vorliegende Arbeit kann nur die linke Variante direkt darstellen. In der rechten Variante muss mit Kopien von Komponenten gearbeitet werden. Jede Kopie einer Komponente muss eine eigene Id haben und ist somit als eigenständige Komponente mit demselben Namen zu behandeln. Die Information, dass mehrere Komponenten Kopien voneinander sind, wird nicht direkt dargestellt. Über Namenskonventionen oder eine zusätzliche $ist - Kopie - von$ Relation könnte dies ergänzt werden.

## Wohlgeformte Konfigurationen: höchstens eine Vaterkomponente

Der Begriff *wohlgeformt* bezieht sich auf den gerade dargestellten Fall einer hierarchischen Struktur, in der jede Komponente höchstens eine Vaterkomponente besitzt. Eine Konfiguration $C = (V, E, J, H)$ heißt *wohlgeformt*, genau dann, wenn jede Komponente höchstens eine Vaterkomponente hat. Das bedeutet:

$(v_1, v_2) \in H \Rightarrow \forall v \in V, v \neq v_1 : (v, v_2) \notin H.$

Dies hat auch zur Folge, dass der ungerichtete Graph $|G|$ mit $G = (V, H)$ kreisfrei (ein Wald) ist[9].

$CONFIGURATION$ sei die Menge aller wohlgeformten Konfigurationen.

## Systemkonfiguration

Die Komponente, welche das IT-System darstellt, wird mit $\sigma$ bezeichnet. Alle von $\sigma$ aus über $H$ erreichbaren Komponenten sind die Bestandteile des IT-Systems. Eine wohlgeformte Konfiguration, in

[9] vgl. etwa Jungnickel [Jun94]

welcher $\sigma \in V$ die einzige Wurzel von $H$ ist und alle Komponenten $v \in V$ von $\sigma$ aus über $H$ erreichbar sind, heißt *Systemkonfiguration*. Die Abbildung 5.2 zeigt mit der Konfiguration $K$ ein Beispiel für eine Systemkonfiguration in zwei Darstellungsvarianten.

## Erweiterte Systemkonfiguration

Für die Erstellung und Darstellung der logischen Architektur eines IT-Systems ist seine Umgebung wichtig[10]. Zur Umgebung gehören Nachbarsysteme und Nutzergruppen, die mit dem IT-System kommunizieren.

Zur Beschreibung von Anwendungsfällen werden Nutzergruppen und aktive Nachbarsysteme benötigt. Sie sind die Akteure in Anwendungsfällen, welche funktionale Anforderungen beschreiben. Die vorliegende Arbeit ordnet (unten) einer Konfiguration Anwendungsfälle zu und stellt dar, welche Komponenten und Konnektoren zu ihrer Implementierung beitragen. Auf diese Weise wird eine Verbindung zwischen den funktionalen Anforderungen und der logischen Architektur hergestellt.

Die Nachbarsysteme und menschliche Nutzer werden in der erweiterten Konfiguration vereinfachend als spezielle logische Komponenten modelliert. Die Knotenmenge der Konfiguration wird um Knoten für Nutzergruppen und Nachbarsysteme erweitert, entsprechende Kanten für Konnektoren werden ebenfalls hinzugefügt.

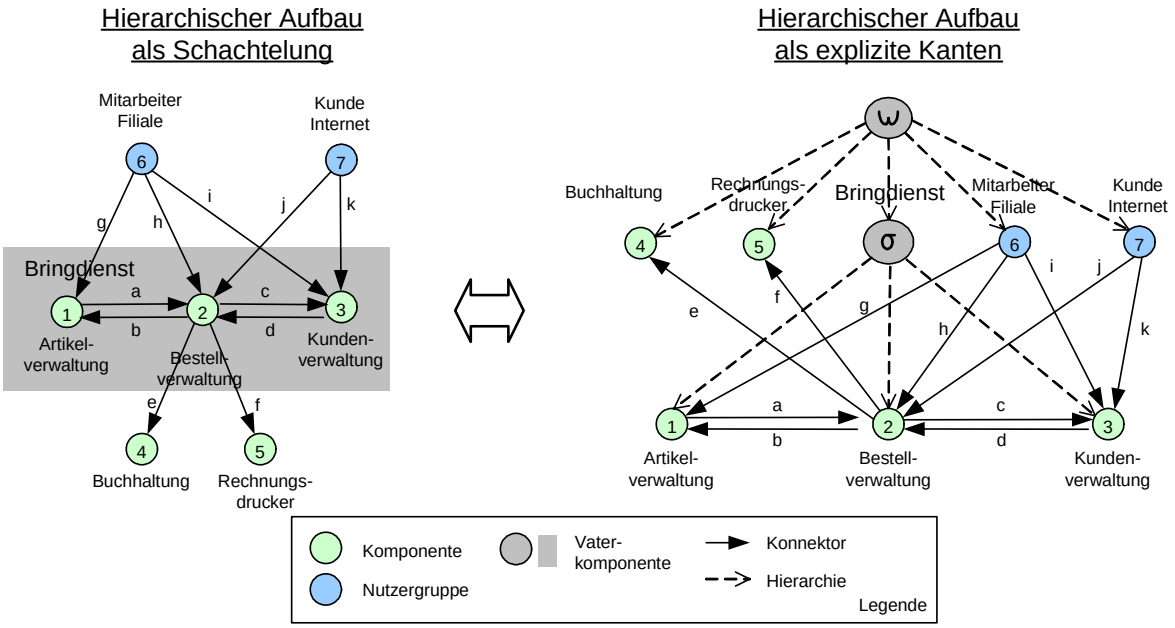


Abbildung 5.5: Darstellung der erweiterten Systemkonfiguration

Die Abbildung 5.5 zeigt links und rechts Beispiele für die Darstellung der erweiterten Konfiguration eines Bringdienstsystems. Die Abbildung zeigt das System *Bringdienst* ($\sigma$), welches aus den Komponenten *Bestellverwaltung*, *Kundenverwaltung* und *Artikelverwaltung* besteht. Mit den Komponenten von Bringdienst kommunizieren die Nachbarsysteme *Buchhaltung* und *Rechnungsdrucker*. Die Nutzergruppen *Mitarbeiter Filiale* und *Kunde Internet* greifen auf die Komponenten von Bringdienst zu.

Je nach Ausgangspunkt der Betrachtung kann es genau einen oder mehrere zusammenhängende Teilgraphen geben, die von $H$ aufgespannt werden: Einerseits kann das System $\sigma$ Wurzelknoten sein, in der Abbildung 5.2 ist der Knoten *Bringdienst* $\sigma$ der Wurzelknoten. Die Systeme der Umgebung

[10] vgl. Abschnitt 2.1.2, Clements et al. [CBB+03, S.195ff] oder Garland und Anthony [GA03, S.89ff]

sind dann ebenfalls Wurzelknoten gerichteter Bäume. Andererseits kann für das System und seine umgebenden Systeme eine Pseudokomponente $\omega$ als Wurzelknoten definiert werden. Das IT-System und seine Nachbarn bilden die erste Ebene dieses Baumes.

Eine wohlgeformte Konfiguration, in der die Hierarchie ein gerichteter Baum mit der Wurzel $\omega$ ist, wird *erweiterte Systemkonfiguration* genannt. Abbildung 5.5 zeigt rechts die Pseudokomponente $\omega$ explizit.

Die Konfiguration eines IT-Systems beschreibt die Zusammenhänge seiner Komponenten. In der erweiterten Systemkonfiguration werden auch Nachbarsysteme und Nutzergruppen berücksichtigt: Die Mengen $V$ und $E$ werden entsprechend erweitert und $J$ wird auf die Erweiterung ausgedehnt. Ebenso wird die Hierarchie $H$ um eine Pseudokomponente $\omega$ erweitert. Das System $\sigma$ und seine Nachbarn und Nutzergruppen sind Kindknoten von $\omega$.

Das Tupel $C' := (V', E', J', H')$ ist die *erweiterte Systemkonfiguration* zur Systemkonfiguration $C = (V, E, J, H)$, genau dann, wenn

- $V' := V \cup U \cup N \cup \{\omega\}$. Dabei ist $U \subset COMPONENT$ eine Menge von Nutzergruppen ($U$ für User) und $N \subset COMPONENT$ eine Menge von Nachbarsystemen ($N$ für Neighbor). Nutzergruppen werden, um die Darstellung einfach zu halten, als besondere logische Komponenten in der Umgebung eines IT-Systems aufgefasst.
- $E' := E \cup B$; wobei $B \subset CONNECTOR$ Konnektoren sind, die das IT-System mit seiner Umgebung verbinden ($B$ für Border, denn die Konnektormenge $B$ beschreibt genau die Systemgrenze),
- $J' : E' \to (V' \setminus \{\omega\}) \times (V' \setminus \{\omega\})$ und $J'|_E = J$ als Zuordnungsfunktion und
- $H' := H \cup \{(\omega, u) | u \in U\} \cup \{(\omega, n) | n \in N\} \cup \{(\omega, \sigma)\}$. Die Hierarchie $H$ der Systemkonfiguration ist ein Teilbaum der erweiterten Hierarchie $H'$.

Im Folgenden werden erweiterte Systemkonfigurationen mit der vereinfachten Schreibweise $C = (V, E, J, H)$ bezeichnet. Der umgebende Text stellt jeweils explizit dar, um welche Art von Konfiguration $C$ es sich handelt.

Die Konfiguration aus Abbildung 5.5 wird im Folgenden mit $SK$ bezeichnet. Die Elemente der erweiterten Systemkonfiguration $SK = (V_{SK}, E_{SK}, J_{SK}, H_{SK})$ zur Konfiguration $K$ (siehe oben) sind gegeben durch:

- $V_{SK} = \{\omega, \sigma, 1, 2, 3, 4, 5, 6, 7\}$, wobei $U_{SK} = \{6, 7\}$ und $N_{SK} = \{4, 5\}$ gilt.
  Damit ist $V_{SK} = V_K \cup U_{SK} \cup N_{SK} \cup \{\omega\}$
- $E_{SK} = \{a, b, c, d, e, f, g, h, j, k\}$, wobei $B_{SK} = \{e, f, g, h, i, j, k\}$ gilt.
  Damit ist $E_{SK} = E_K \cup B_{SK}$
- $J_{SK} = \{(a, (1, 2)), (b, (2, 1)), (c, (2, 3)), (d, (3, 2)), (e, (2, 4)), (f, (2, 5)), (g, (6, 1)),$
  $(h, (6, 2)), (i, (6, 3)), (j, (7, 2)), (k, (7, 3))\}$
- $H_{SK} = \{(\omega, \sigma), (\omega, 4), (\omega, 5), (\omega, 6), (\omega, 7), (\sigma, 1), (\sigma, 2), (\sigma, 3)\}$

## Einlaufende und ausgehende Konnektoren

Seien $incoming : CONFIGURATION \times COMPONENT \to \wp(CONNECTOR)$ und ebenso $outgoing : CONFIGURATION \times COMPONENT \to \wp(CONNECTOR)$ zwei Funktionen, die zur Komponente $v \in V$ jeweils eingehende und ausgehende Konnektoren innerhalb einer Konfiguration $C$ liefern:

$incoming(C, v) := \{e \in E | \exists v_{in} \in V \text{ mit } J(e) = (v_{in}, v)\}$ und

$outgoing(C, v) := \{e \in E | \exists v_{out} \in V \text{ mit } J(e) = (v, v_{out})\}$.

In der Konfiguration $SK$ aus Abbildung 5.5 gilt beispielsweise: $incoming(SK, 2) = \{a, d, h, j\}$ und $outgoing(SK, 2) = \{b, c, e, f\}$.

Abkürzend wird $incoming(v)$ bzw. $outgoing(v)$ anstatt $incoming(C, v)$ bzw. $outgoing(C, v)$ geschrieben, wenn sich $C$ aus den Kontext erschließt. Zur Bestimmung der ein- und ausgehenden Konnektoren einer Komponentenmenge $W \subseteq V$ werden die Funktionen $incoming$ und $outgoing$ auf Mengen erweitert. Interne Konnektoren der Komponentenmenge $W$ gehören weder zu den ein- noch zu den ausgehenden Konnektoren. Interne Konnektoren haben ihren Start- und Endpunkt innerhalb von $W$. Für eine beliebige Menge von Komponenten $W \subseteq V$, sind interne Konnektoren gegeben durch

$internal : CONFIGURATION \times \wp(COMPONENT) \rightarrow \wp(CONNECTOR)$:

mit

$internal(C, W) := \{e \in E | J(e) = (v_1, v_2) \text{ mit } v_1, v_2 \in W\}$.

Werden beispielsweise alle Kindkomponenten von $\sigma$ gewählt, also etwa $W = \{1, 2, 3\}$ aus der Konfiguration $SK$, dann ist $internal(SK, \{1, 2, 3\}) = \{a, b, c, d\}$ die Menge der internen Konnektoren des IT-Systems $\sigma$. Mithilfe der Funktion $internal$ können nun auch die Funktionen $incoming$ und $outgoing$ auf Mengen erweitert werden:

$$incoming(C, W) := \left(\bigcup_{w \in W} incoming(C, w)\right) \setminus internal(C, W)$$

$$outgoing(C, W) := \left(\bigcup_{w \in W} outgoing(C, w)\right) \setminus internal(C, W)$$

Für die Komponentenmenge $W = \{1, 2, 3\}$ aus der Beispielkonfiguration $SK$ können damit die ein- und ausgehenden Konnektoren des IT-Systems berechnet werden.

$$\begin{aligned} incoming(SK, \{1, 2, 3\}) &= (incoming(SK, 1) \cup incoming(SK, 2) \cup incoming(SK, 3)) \\ &\quad \setminus internal(SK, \{1, 2, 3\}) \\ &= (\{b, g\} \cup \{a, d, h, j\} \cup \{c, i, k\}) \setminus \{a, b, c, d\} \\ &= \{g, h, i, j, k\} \end{aligned}$$

## Komponentenhierarchie

### Wurzeln

Eine Wurzel des Hierarchiewaldes ist dadurch gekennzeichnet, dass sie keine Vaterkomponente hat. Das zweistellige Prädikat $isroot : CONFIGURATION \times COMPONENT \rightarrow \mathbb{B}$ liefert immer dann *true*, wenn die Komponente $v \in V$ Wurzel ist und *false*, wenn mindestens eine Komponente in der Hierarchie über $v$ steht:

$$isroot(C, v) := \begin{cases} false: & \exists u \in V, (u, v) \in H \\ true: & sonst \end{cases}$$

Die Funktion $root$ wird auf die gesamte Konfiguration $C$ angewendet
$root : CONFIGURATION \rightarrow \wp(COMPONENT)$
und liefert alle Wurzeln:

$root(C) := \{v \in V | isroot(C, v)\}$

Offenbar gilt immer: $root(C) = \emptyset \Leftrightarrow H = \emptyset$, da für $H$ Kreisfreiheit gefordert wird [Jun94]. Am Beispiel $SK$ ergibt sich für die Komponenten $\omega$, $\sigma$ und 1 folgendes Bild:

$isroot(SK, \omega) = true$, da $\forall v \in V : (v, \omega) \notin H$, nach Definition der Pseudokomponente $\omega$, weiter-

hin ist $isroot(SK, \sigma) = false$, da $(\omega, \sigma) \in H$ und $isroot(SK, 1) = false$, da $(\sigma, 1) \in H$. Für die Konfiguration $SK$ ergibt sich unter Berücksichtigung von $H_{SK}$: $root(SK) = \{\omega\}$.

#### Blätter

Blätter sind alle Komponenten, die keine Teilkomponenten haben. Zur Ermittlung der Blätter wird eine Funktion

$leafs : CONFIGURATION \rightarrow \wp(COMPONENT)$

auf Konfigurationen $C = (V, E, J, H)$ definiert mit:

$leafs(C) := \{v \in V | \forall w \in V : (v, w) \notin H\}$

Für die Konfiguration $SK$ ergibt sich: $leafs(SK) = \{1, 2, 3, 4, 5, 6, 7\}$, da für die Komponenten $\sigma$ und $\omega$ jeweils mindestens eine Kindkomponente existiert.

#### Vaterkomponenten

Für die Bestimmung der Vaterkomponente und die Navigation in der Komponentenhierarchie wird die Funktion

$parent : CONFIGURATION \times COMPONENT \rightarrow \wp(COMPONENT)$

eingeführt. Hierbei liefert $parent$ für wohlgeformte Konfigurationen $C$ entweder die Vaterkomponente als einelementige Menge, oder die leere Menge zurück. In nicht wohlgeformten Konfigurationen könnte das Ergebnis mengenwertig sein.

$$parent(C, v) := \begin{cases} \emptyset & : isroot(C, v) \\ \{u \in V | (u, v) \in H\} & : sonst \end{cases}$$

Es gilt nach Definition für jede wohlgeformte Konfiguration $C = (V, E, J, H) \in CONFIGURATION$, dass $\forall v \in V : |parent(C, v)| \leq 1$. Die komplette Menge der Eltern zu einer Komponente $v \in V$ kann mit $parent$ rekursiv über eine Funktion $allparents$ berechnet werden:

$allparents : CONFIGURATION \times COMPONENT \rightarrow \wp(COMPONENT)$

mit

$allparents(C, v) := parent(C, v) \cup \bigcup_{x \in parent(C,v)} allparents(C, x)$

Für die Konfiguration $SK$ und die Komponente 1 kann die Vaterkomponente beispielsweise und die komplette darüber stehende Hierarchie berechnet werden:

$parent(SK, 1) = \{\sigma\}$, da $(\sigma, 1) \in H$

$$\begin{aligned} allparents(SK, 1) &= parent(SK, 1) \cup \bigcup_{x \in parent(SK,1)} allparents(SK, x) \\ &= \{\sigma\} \cup allparents(SK, \sigma) \\ &= \{\sigma\} \cup parent(SK, \sigma) \cup \bigcup_{x \in parent(SK,\sigma)} allparents(C, x) \\ &= \{\sigma\} \cup \{\omega\} \cup allparents(SK, \omega) \\ &= \{\sigma, \omega\} \cup parent(SK, \omega) \\ &= \{\sigma, \omega\} \cup \emptyset \end{aligned}$$

#### Tiefe

Einer Komponente $v \in V$ kann innerhalb einer wohlgeformten Konfiguration eine Tiefe in Bezug auf eine Hierarchie $H$ zugeordnet werden. Die Tiefe ist bestimmt durch die Länge des Pfades in $H$

zwischen einer Wurzel und der Komponente $v$. Die Funktion

$depth : CONFIGURATION \times COMPONENT \rightarrow \mathbb{N}_0$

liefert die Tiefe einer Komponente $v \in V$ innerhalb einer wohlgeformten Konfiguration $C$.

$$depth(C, v) := \begin{cases} 0 & : v \in root(C, H) \\ depth(C, u) + 1 & : \{u\} = parent(C, v) \end{cases}$$

So gilt beispielsweise $depth(SK, \omega) = 0$, da $isroot(SK, \omega) = true$ und $depth(SK, \sigma) = 1$, da $\{\omega\} = parent(SK, \sigma)$ sowie $depth(SK, 1) = 2$, da $\{\sigma\} = parent(SK, 1)$.

Dem Hierarchiewald $H$ einer wohlgeformten Konfiguration kann eine Tiefe zugeordnet werden. Die Tiefe von $H$ ist das Maximum der Tiefen der Komponenten. Über $depth$ kann daher die Tiefe des Hierarchiewaldes bestimmt werden

$depth : CONFIGURATION \rightarrow \mathbb{N}_0$

mit

$depth(C) := max_{v \in V}(depth(C, v))$.

Für die Konfiguration $SK$ gilt $depth(SK) = 2 = depth(SK, 1)$.

#### Tiefe in einer nicht wohlgeformten Konfiguration

In einer nicht wohlgeformten Konfiguration $C \notin CONFIGURATION$ wird die Tiefe $depth$ hier nicht definiert: Die Tiefe von $v \in V$ kann nicht berechnet werden, wenn es mindestens zwei Knoten von $H$ gibt, von denen aus unterschiedlich lange Pfade zu $v$ existieren. Mehrere Beispiele für Hierarchien mit Angaben zu den Tiefen der jeweiligen Komponenten sind in Abbildung 5.6 dargestellt.

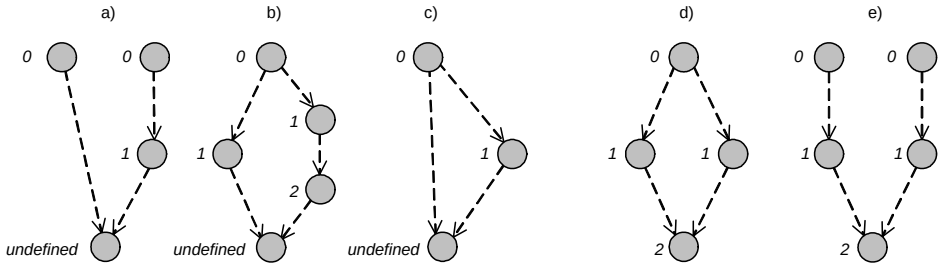


Abbildung 5.6: Beispiele für Hierarchietiefe in unterschiedlichen Konfigurationen

#### Teilkomponenten

Zur Bearbeitung der Komponentenhierarchie sind Funktionen notwendig, welche die Teilkomponenten zu einer Komponente $v \in V$ bestimmen. Die Funktion

$parts : CONFIGURATION \times COMPONENT \rightarrow \wp(COMPONENT)$

liefert zu einer Komponente $x \in V$ alle direkten Teile in Bezug auf die Hierarchie $H$:

$parts(C, x) := \{u \in V | (x, u) \in H\}$

In der Beispielkonfiguration $SK$ können so die Teilkomponenten der Komponente $\sigma$ berechnet werden.

$parts(SK, \sigma) = \{1, 2, 3\}$, da $\{(\sigma, 1), (\sigma, 2), (\sigma, 3)\} \subset H$.

Alle Teilkomponenten bis zu einer definierten Hierarchietiefe bestimmt die Funktion

$parts : CONFIGURATION \times COMPONENT \times \mathbb{N}_0 \rightarrow \wp(COMPONENT)$.

Die Funktion $parts$ wird rekursiv definiert:

$$parts(C, x, n) := \begin{cases} \emptyset & : n = 0 \\ parts(C, x) & : n = 1 \\ parts(C, x, n-1) \cup \{v \in V | (y, v) \in H, y \in parts(C, x, n-1)\} & : n > 1 \end{cases}$$

Für $SK$ gilt offenbar:

$$\begin{array}{lcl} parts(SK, \omega, 0) & = & \emptyset \\ parts(SK, \omega, 1) & = & parts(SK, \omega) = \{\sigma, 4, 5, 6, 7\} \\ parts(SK, \omega, 2) & = & parts(SK, \omega) \cup \{v \in V_{SK} | (y, v) \in H_{SK}, y \in parts(SK, \omega, 1)\} \\ & = & \{\sigma, 4, 5, 6, 7\} \cup \{v \in V_{SK} | (y, v) \in H_{SK}, y \in \{\sigma, 4, 5, 6, 7\}\} \\ & = & \{\sigma, 4, 5, 6, 7\} \cup \{1, 2, 3\} \text{ da } \{(\sigma, 1), (\sigma, 2), (\sigma, 3)\} \subset H_{SK} \\ & = & \{\sigma, 1, 2, 3, 4, 5, 6, 7\} \end{array}$$

Zusätzlich liefert

$allparts : CONFIGURATION \times COMPONENT \rightarrow \wp(COMPONENT)$

alle Teilkomponenten zu $x \in V$ über die gesamte Hierarchie hinweg. $allparts$ kann rekursiv berechnet werden:

$allparts(C, x) := parts(C, x) \cup \bigcup_{u \in parts(C,x)} allparts(C, u)$

## 5.4 Nutzungsszenarios und Kommunikationspfade

### Nutzungsszenarios als Implementierung von Anwendungsfällen

Beschreibungen von System- und Software-Architekturen beschränken sich häufig auf Darstellungen der Konfiguration. Daraus ist ablesbar, mit welchen direkten Nachbarn eine Komponente grundsätzlich kommunizieren könnte. Welche Komponenten tatsächlich miteinander kommunizieren, ist aus dieser statischen Darstellung nicht ablesbar. Hierfür sind Verhaltensbeschreibungen erforderlich. Zusätzlich fehlt bei diesen statischen Darstellungen häufig der Bezug zu den funktionalen Anforderungen:

Wenn eine Komponente beispielsweise erst in einer späteren Stufe eines Projekts umgesetzt wird, ist aus den typischen Architekturdarstellungen nicht ersichtlich, welche Anwendungsfälle damit auch erst in der nächsten Stufe ausgeliefert werden können, da sie auf diese Komponente angewiesen sind.

Der Bezug zwischen den funktionalen Anforderungen und der logischen Architektur wird in der Architekturtheorie über Nutzungsszenarios hergestellt[11]. Nutzungsszenarios zeigen, welche Komponenten und Konnektoren an der Implementierung eines Anwendungsfalls beteiligt sind. Sie werden als Teilmenge der Konnektoren einer Konfiguration dokumentiert.

Die Abbildung 5.7 zeigt drei Beispiele für Szenarios. Ein Nutzungsszenario in der oben angedeuteten Bringdienst Software könnte „Abwicklung Bestellung“ sein und zum gleichnamigen Anwendungsfall gehören. Dieses Szenario wird von den Komponenten *Bestellverwaltung*, *Kundenverwaltung*, *Artikelverwaltung*, *Buchhaltung* und *Rechnungsdrucker* sowie den entsprechenden Konnektoren implementiert. Das Szenario wird nur von der Nutzergruppe *Kunde Internet* aufgerufen. Die Nutzergruppe *Mitarbeiter Filiale* ist nicht beteiligt. Die Darstellung kann beispielsweise für einen Architekten folgende Fragen beantworten:

[11] Das ist für den Teil der funktionalen Anforderungen möglich, die als Anwendungsfälle formuliert werden können.

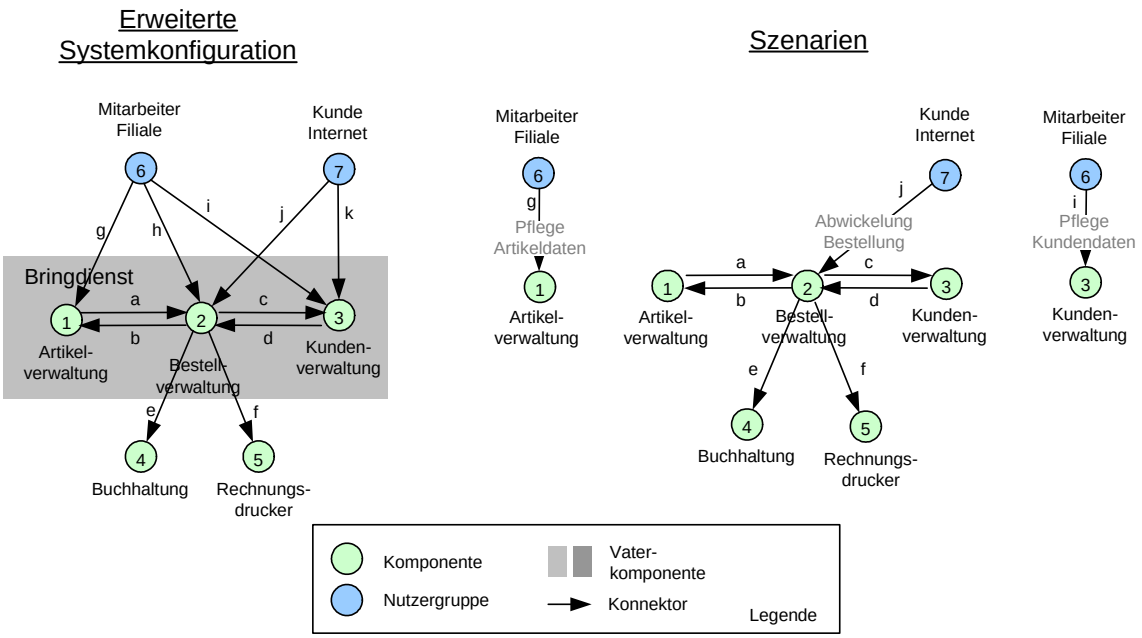


Abbildung 5.7: Szenarios als Teilmenge der Konnektoren

1. Welche Komponenten müssen implementiert sein, damit der Anwendungsfall „Abwicklung Bestellung“ getestet oder ausgeliefert werden kann?
2. Wenn die Komponente *Kundenverwaltung* geändert wird, welche Anwendungsfälle sind dann mindestens zu testen?
3. Wenn die Komponente *Kundenverwaltung* abstürzt, welche Anwendungsfälle können dann mindestens nicht mehr ausgeführt werden?
4. Gibt es Komponenten, die hohe Priorität haben, da sie an vielen Nutzungsszenarios teilnehmen?

Damit sind Nutzungsszenarios ein wichtiges Mittel der Zusammenarbeit zwischen Auftraggeber, Projektleiter und Architekt. Da diese den Zusammenhang zwischen Planung, Architektur und funktionalen Anforderungen (dem eigentlichen Nutzen des IT-Systems) herstellen und greifbar machen. In den folgenden Kapiteln wird dies bei der Erzeugung von Architektursichten und dem architekturzentrierten Projektmanagement wieder aufgegriffen.

Die Nutzungsszenarios sind keine Verhaltensbeschreibungen. Sie machen keine Aussage zur Zustandsveränderung des IT-Systems oder seiner Komponenten. Sie dokumentieren lediglich, welche Komponenten und welche Konnektoren an der Implementierung eines Anwendungsfalls beteiligt sind.

Jeder Konfiguration $C$ kann eine Menge von Anwendungsfällen $UC$ (UC für Use Case) zugeordnet werden. Diese Anwendungsfälle werden von der Konfiguration unterstützt. Die Elemente der Menge $uc \in UC$ heißen *Anwendungsfall* oder *Use Case*. Ein Anwendungsfall wird durch eine Id eindeutig identifiziert, er kann zusätzlich einen nicht eindeutigen Namen haben. Wie bei Komponenten und Konnektoren gibt es die beiden Funktionen $id$ und $name$, sie sind analog definiert:

$id : USECASE \rightarrow \mathbb{ID}$.
$name : USECASE \rightarrow \mathbb{ID}$

Die Menge aller denkbaren Anwendungsfälle wird mit $USECASE$ bezeichnet. Die Anwendungsfälle $uc \in UC$ können aus einer ersten Anforderungsanalyse stammen oder in einer späteren Pro-

jektphase aus dem Pflichtenheft entnommen sein.

Der Zusammenhang zwischen den Anwendungsfällen und einer Konfiguration $C$ wird über die Funktion $S$ (S für Scenario) hergestellt. Sie ordnet jedem Anwendungsfall ein Nutzungsszenario zu. Ein Nutzungsszenario $s$ auf einer Konfiguration $C = (V, E, J, H)$ sei eine Teilmenge der Konnektoren, also $s \subseteq E$[12]. Die Funktion $S$ ist definiert mit:

$S : USECASE \rightarrow \wp(CONNECTOR)$

Die drei Anwendungsfälle $UC_{SK} = \{uc_1, uc_2, uc_3\}$ aus Abbildung 5.7 haben über die Konnektoren aus der erweiterten Systemkonfiguration $SK$ folgende Darstellung:

$S = \{(uc_1, \{g\}), (uc_2, \{a, b, c, d, e, f, j\}), (uc_3, \{i\})\}$, bzw. $S(uc_1) = \{g\}$, $S(uc_2) = \{a, b, c, d, e, f, j\}$ und $S(uc_3) = \{i\}$.

Wie die Zuordnungsfunktion $J$ wird auch $S$ in Beispielen als Tupelmenge dargestellt mit:

$S \subset UC \times \wp(E)$.

### Kommunikationspfad

Ein Kommunikationsweg zwischen zwei Komponenten $v, w \in V$ ist eine Liste von Konnektoren $p_{v,w} = (e_1, \ldots, e_n), n \in \mathbb{N}, e_i \in E, i \leq n$, welche die beiden Komponenten verbinden. Die Konnektoren müssen zusammenhängen, damit sie einen Weg bilden können, es gilt also für $i \leq n - 1$ und $v_i, v_{i+1}, v_{i+2} \in V$:

$J(e_i) = (v_i, v_{i+1}) \Rightarrow J(e_{i+1}) = (v_{i+1}, v_{i+2})$

Der Endpunkt von $e_i$ ist der Anfangspunkt von $e_{i+1}$. Der Kommunikationsweg $p_{v,w}$ besucht Komponenten, diese werden durch die Liste $(v_1, \ldots, v_{n+1}), v_i \in V, i \leq n + 1, v_1 = v, v_{n+1} = w$ ausgedrückt. Die Konnektorfolge $p_{v,w}$ wird *Kommunikationspfad* genannt, wenn die besuchten Komponenten $v_i$ paarweise verschieden sind $v_i \neq v_j; i, j \leq n + 1; i \neq j$, wenn also in $p_{v,w}$ kein Kreis enthalten ist. Zwischen zwei Komponenten können mehrere verschiedene Kommunikationspfade existieren.

Da der Kommunikationsgraph ein Multigraph ist, kann von einer Komponentenfolge $(v_1, \ldots, v_{n+1})$ nicht eindeutig auf einen Kommunikationsweg bzw. -pfad zurück geschlossen werden.

Für die Konfiguration $SK$ kann beispielsweise ein Kommunikationspfad zwischen Komponente 6 (Mitarbeiter Filiale) und Komponente 5 (Rechnungsdrucker) angegeben werden: $p_{6,5} = (h, f)$ aber auch $p_{6,5} = (g, a, f)$.

## 5.5 Zusatzinformationen über Attribute

Derzeit enthalten Architektursichten und andere Arten der Architekturbeschreibung häufig ausschließlich Informationen über Struktur und Verhalten des IT-Systems, vgl. etwa [CBB+03, VAC+05, PBG07]. Informationen, die für die Kommunikation im Entwicklungsteam und mit dem Auftraggeber wichtig sind, sind in diesen Darstellungen typischerweise nicht enthalten:

- Ein Auftraggeber interessiert sich beispielsweise für die Aufwände zur Erstellung einer Komponente und deren Liefertermin.
- Ein Entwickler muss bei Fragen zu einer Komponente wissen, wer für diese verantwortlich ist oder wer die Anforderungen dazu definiert hat.

[12] Dies ist eine stark vereinfachte Darstellung von Szenarios, um mit diesen einfacher Rechnen zu können. Zwei weitere Forderungen an Szenarios sind eigentlich: (1) Die Konnektoren eines Szenarios müssen einen zusammenhängenden Teilgraphen aufspannen, d.h. nur eine Komponente, die eine Nachricht erhalten hat, kann selbst wieder Nachrichten versenden. (2) Ein Szenario muss in der Umgebung von $\sigma$ beginnen, also bei einem Akteur, denn $\sigma$ ist ein reaktives System.

- Die Qualitätssicherung interessiert sich für die Überdeckung der Komponente mit Testfällen oder die Zahl der existierenden Testfallspezifikationen.
- Die Betriebsführung muss möglicherweise die Ausfallwahrscheinlichkeit einer Komponente wissen [Bri00].

Werden diese und ähnliche Informationen in der logischen Architektur ergänzt, können Architekt und Projektleiter besser zusammenarbeiten, da eine Verbindung zwischen der Projektplanung und der Architektur hergestellt werden kann. Die Kommunikation mit dem Auftraggeber kann verbessert werden, da etwa Kosten und Termine transparenter werden. Weitere Entwurfsschritte können verbessert werden, da etwa Informationen zur Zuverlässigkeit einer logischen Komponente helfen, über Redundanzen in der Verteilungsarchitektur zu entscheiden.

Den Bestandteilen einer logischen Architektur werden zusätzliche Informationen mit Hilfe von *Attributen* zugewiesen. Die Informationen können aus einem Projektplan, der Implementierung oder der Analyse von Qualitätseigenschaften stammen. Die Abbildung 5.8 gibt ein Beispiel für Informationen aus einem Projektplan. Jeder Komponente und jedem Konnektor werden Termin, Aufwand und verantwortliche Teams zugewiesen. Bei der Veränderung logischer Architekturen und der Erstellung von Architektursichten werden diese Zusatzinformationen berücksichtigt.

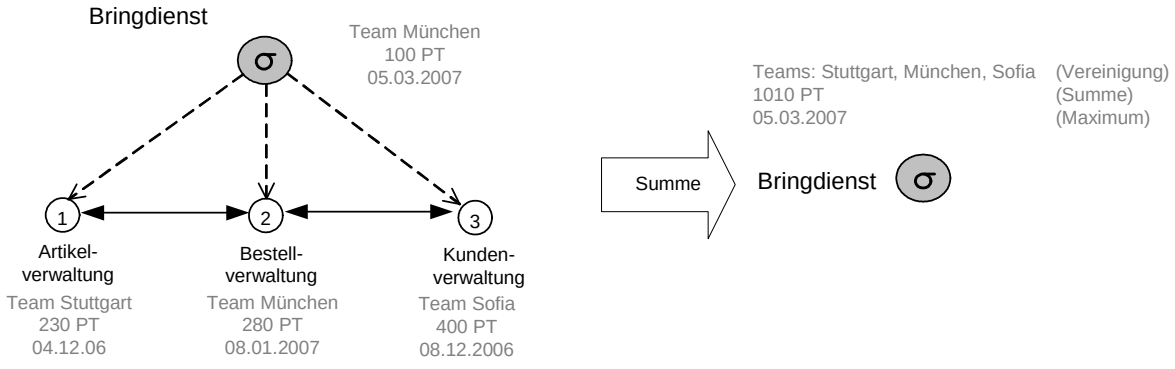


Abbildung 5.8: Zusatzinformationen

Die Vaterkomponenten können ihre Summen-Attributwerte aus den Attributwerten ihrer Teilkomponenten berechnen. Im Beispiel aus Abbildung 5.8 ergibt sich der Termin vom IT-System $\sigma$ aus dem Maximum des eigenen Fertigstellungstermins und der Fertigstellungstermine der Teile. Der Aufwand ist die Summe aus eigenem Aufwand und den Erstellungsaufwänden für die Teile und das verantwortliche Team ist die Vereinigungsmenge aller verantwortlichen Teams[13]. Jedes Attribut hat dabei eigene Summenoperationen, beispielsweise Maximum $max$, Summe $\Sigma$ oder Vereinigungsmenge $\bigcup$.

## Attribute

Jeder Komponente und jedem Konnektor können mehrere Informationen über Tupel zugewiesen werden. Ein Tupel hat die Form (*Attributname*, *Attributtyp*), etwa mit den Werten ('Aufwand', *Personentage*). Das Tupel (*Attributname*, *Attributtyp*) wird Attribut[14] genannt. Die Menge aller denkbaren Attribute wird mit $ATTRIBUTE = \mathbb{ID} \times \mathbb{AT}$ bezeichnet, dabei ist $\mathbb{AT}$ die Menge aller denkbaren Attributtypen. Die Tabelle 5.1 stellt Beispiele für mögliche Attribute dar.

Attributtypen sind Tupel der Form $(id, \mathbb{D}_{id})$. Dabei ist $id \in \mathbb{ID}$ ein eindeutiger Name für den Typ, z.B. 'Datum', 'Geld' oder 'Personenmenge'. $\mathbb{D}_{id}$ ist eine möglicherweise unendlich große Menge von

[13] Der Vaterkomponente werden dabei zunächst nur die Aufwände, Termine und Teams für die Integration zugeordnet.
[14] Dies entspricht der Idee der Properties aus ACME [GMW00]

| Themenfeld | Attributtyp | Attributname | Beispiel |
|---|---|---|---|
| Qualität | Wahrscheinlichkeit | Ausfallwahrscheinlichkeit | 5% |
| Qualität | Wahrscheinlichkeit | Angriffswahrscheinlichkeit | 10% |
| Projektplanung | Datum | geplanter Anfang | 20.09.2005 |
| Projektplanung | Datum | geplantes Ende | 18.06.2006 |
| Projektplanung | Personentage | geplanter Aufwand | 100 PT |
| Projektplanung | Geld | Kosten | 50.000 Euro |
| Architektursichten | Software-Kategorie | Software-Kategorie | AT |
| Implementierung | Quelltext | Implementierung | {Kunde.java } |

Tabelle 5.1: Beispiele für Attribute aus unterschiedlichen Themenbereichen

Werten. $\mathbb{D}_{id}$ wird Wertebereich des Attributtyps $id$ genannt. Die Menge $\mathbb{D}$ sei die Vereinigungsmenge aller denkbaren Wertebereiche. Die Menge aller Attributtypen wird, wie oben schon erwähnt, mit $\mathbb{AT}$ bezeichnet. Werden Personentage beispielsweise nur mit ganzen Zahlen dargestellt, ist die Darstellung des Attributtyps $Personentage \in \mathbb{AT}$ wie folgt:

$Personentage :=$ ('Personentage', $\mathbb{N}_0$) mit 'Personentage' $\in \mathbb{ID}$ und $\mathbb{N}_0 = \mathbb{D}_{Personentage}$

Der Wertebereich kann auch eine (endliche) Aufzählung sein, die Eigenschaft $Software{-}Kategorie \in \mathbb{AT}$ [Sie04], wie sie in Abschnitt 3.4.5 dargestellt wird, hat folgende Darstellung:

$Software - Kategorie :=$ ('Software-Kategorie', $\{0, A, T, AT\}$).

Die eigentliche Information wird einer Komponente oder einem Konnektor über die Belegung $value$ zugewiesen. Sie weist jedem Attribut in einer Konfiguration einen gültigen Wert aus dem Wertebereich des Attributtyps oder das Symbol $\langle\rangle$ für *undefiniert* zu.

$value : (COMPONENT \cup CONNECTOR) \times ATTRIBUTE \rightarrow \mathbb{D} \cup \{\langle\rangle\}$.

Die Belegung $value$ ist für jede Konfiguration spezifisch, d.h. derselben Komponente können in zwei verschiedenen Konfigurationen unterschiedliche Attributwerte zugewiesen werden. Dieser Sachverhalt ist als Metamodell in Abbildung 5.11 dargestellt.

In der Abbildung 5.8 rechts ist eine Konfiguration $C$ dargestellt, die nur aus der Komponente $\sigma$ besteht, also $C = (\{\sigma\}, \emptyset, J_\emptyset, \emptyset)$. Die Belegung $value$ liefert für diese Konfiguration:

$value(\sigma, Plan.Aufwand) = 1010PT$
$value(\sigma, Plan.Ende) =$ 05.03.2007
$value(\sigma, Plan.Verantwortliche) =$ { Team Stuttgart, Team München, Team Sofia }

Mit den zugewiesenen Werten wird im Rahmen der Projektionen in Kapitel 6 gerechnet. Daher dürfen nur Attributtypen $type \in \mathbb{AT}, type = (id_{type}, \mathbb{D}_{type})$ verwendet werden, die mindestens folgende Anforderungen erfüllen:

- Für je zwei Werte $a, b \in \mathbb{D}_{type}$ des Attributtyps $type$ muss eine Zusammenfassungsoperation $+_{type}$ definiert sein mit $a +_{type} b \in \mathbb{D}_{type}$.
- Es muss ein neutrales Element $0_{type} \in \mathbb{D}_{type}$ (Null) definiert sein, also $\forall a \in \mathbb{D}_{type} : a +_{type} 0_{type} = 0_{type} +_{type} a = a$.
- Für die Werte des Attributtyps muss mindestens eine partielle Ordnung über einen Vergleichsoperator $\leq_{type}$ definiert sein.
- Für jeden Typ gibt es zusätzlich einen gemeinsamen Wert $\langle\rangle$, der *undefiniert* darstellt. Sobald ein Attribut in einer Summe den Wert $\langle\rangle$ hat, ist der Wert der gesamten Summe undefiniert. Damit gilt $\forall a \in \mathbb{D}_{type} : a +_{type} \langle\rangle = \langle\rangle +_{type} a = \langle\rangle$.

Der Zusammenfassungsoperator sei kommutativ $\forall a, b \in \mathbb{D}_{type} : a +_{type} b = b +_{type} a$ und assoziativ

$\forall a, b, c \in \mathbb{D}_{type} : (a +_{type} b) +_{type} c = a +_{type} (b +_{type} c)$. Das hat den Vorteil, dass bei der Summation nicht auf die Reihenfolge der Summanden geachtet werden muss. Wir fordern für die partielle Ordnung und den $+_{type}$-Operator Monotonie[15], also $\forall a, b \in \mathbb{D}_{type} : a \leq_{type} a +_{type} b \wedge b \leq_{type} a +_{type} b$. Die Forderung nach Monotonie schränkt die partielle Ordnung $\leq$ noch weiter ein, da verlangt wird, dass $a$ und $a +_{type} b$ sowie $b$ und $a +_{type} b$ in Relation stehen.

Ein Attributtyp kann mehrere Summen- und Vergleichsoperatoren mit entsprechenden Null-Elementen $(+_{type}, \leq_{type}, 0_{type})$ haben, welche die oben genannten Bedingungen erfüllen. Ein Beispiel für einen solchen Attributtyp ist $Datum$, hier erfüllen beispielsweise folgende Summen- und Vergleichsoperatoren die oben genannten Anforderungen: $(min, \geq$, *MAXDATUM*) und $(max, \leq$, MINDATUM). Dafür werden zwei künstliche Konstante als neutrale Elemente verwendet. Für den Anfangstermin ist dies *MAXDATUM* und für den Endtermin ist dies *MINDATUM*. In Bezug auf die entsprechenden Summenoperatoren $min$ und $max$ verhalten sich die beiden Konstanten neutral:

Sei $MAXDATUM \in DATUM$ dann gelte $\forall d \in DATUM : min(MAXDATUM, d) = d$.

Sei $MINDATUM \in DATUM$ dann gelte $\forall d \in DATUM : max(MINDATUM, d) = d$.

Komponenten und Konnektoren können mehrere Attribute mit demselben Attributtyp haben. Wenn zu einem Attributtyp mehrere passende Summen- und Vergleichsoperatoren definiert sind, können Attributen des gleichen Typs unterschiedliche Operatoren zugeordnet werden. Die zu verwendenden Operatoren hängen damit vom Attribut selbst und seinem Attributtyp ab. Die Tabelle 5.2 gibt mehrere Beispiele für Attributtypen und entsprechende Operatoren. Die Attribute *geplanter Anfang* und *geplantes Ende* haben beispielsweise denselben Attributtyp *Datum* aber verschiedene Operatoren.

| Attributtyp | Attributname $name$ | $+_{name}$ | $\leq_{name}$ | $0_{name}$ |
|---|---|---|---|---|
| Datum | Plan.Anfang | $min$ | $\geq$ | MAXDATUM |
| Datum | Plan.Ende | $max$ | $\leq$ | MINDATUM |
| Personentage | Plan.Aufwand | $+$ | $\leq$ | 0 PT |
| Geld | Kosten | $\sum$ | $\leq$ | 0 Euro |
| Personenmenge | Plan.Verantwortlich | $\cup$ | $\subseteq$ | $\emptyset$ |
| Wahrscheinlichkeit | Überlebenswahrscheinlichkeit | $*$ | $\leq$ | 1 |
| Software-Kategorie | Software-Kategorie | $\cup$ | $\subseteq$ | 0-Software |
| Java-Quelltext | Implementierung | $\cup$ | $\subseteq$ | $\emptyset$ |

Tabelle 5.2: Beispiele für die Operatoren zu Attributen

Die Summen- und Vergleichsoperatoren werden im Folgenden über jedes Attribut $\in ATTRIBUTE$ gekennzeichnet. Die Kennzeichnung ist ein Hinweis auf die tatsächlich auszuführenden Operatoren. So ist bei zwei Operanden mit $a +_{Plan.Ende} b$ tatsächlich $max(a, b)$ gemeint und bei mehr als zwei Operanden ist mit $\sum^{Plan.Ende}(a_1, \ldots, a_n)$ tatsächlich $max(a_1, \ldots, a_n)$ gemeint.

## Beispiele: Software-Kategorie und Aufgabe

Auch Eigenschaften oder Typen von Komponenten und Konnektoren können als Attribute modelliert werden. Als Beispiel werden die beiden Attributtypen *Software-Kategorie* [Sie04] und *Aufgabe* definiert. Sie werden im Laufe der vorliegenden Arbeit an mehreren Stellen eingesetzt. Jede Komponente und jeder Konnektor kann eine oder mehrere weitere Eigenschaften haben, etwa auch Transaktionalität, Aktiv/Passiv oder Synchron/Asynchron sein. Diese Eigenschaften können beispielsweise

[15] Die Monotonie wird beispielsweise in Abschnitt 8.6 für ein Optimierungsverfahren verwendet, in dem Summanden weggelassen werden, der Wert der Summe sollte sich verringern.

aus einer Referenzarchitektur[16] stammen.

Der Attributtyp *Aufgabe* ist an die Referenzarchitekturen aus Quasar [Sie02a] angelehnt und definiert Architekturelemente eines betrieblichen Informationssystems, die sich im Laufe mehrerer Projekte herausgebildet[17] haben. Ein neutrales Element *neutral* wurde ergänzt:

$$Aufgabe \quad := \quad (\text{'Aufgabe'}, \{\ \textit{Subsystem, Präsentation, Anwendungskern, Datenhaltung, Dialog, Dialogverwalter, Fassade, A-Fall, Entitätsverwalter, Zugriffsschicht, neutral}\ \})$$

Je nach Anwendungsbereich und zugrunde liegender(n) Referenzarchitektur(en) und anderen Entwurfsentscheidungen[18] kann ein anderer Definitionsbereich $\mathbb{D}_{Aufgabe}$ von $Aufgabe$ gewählt werden.

Bei der Definition solcher Attribute, muss auch eine Summenoperation und eine Vergleichsoperation definiert sein. Für die Attributtypen Software-Kategorie und Aufgabe werden diese in den nachfolgenden Abschnitten definiert.

#### Summe und partielle Ordnung für Software-Kategorie

Die Software-Kategorien bilden mit der Kompositionsoperation einen vollständigen Verband, mit $AT - Software$ als maximalem Element und $0 - Software$ als minimalem und neutralen Element ($0 \leq A, 0 \leq T, A \leq AT, T \leq AT$) und der Rechenoperation $+$, diese Operation wird in Tabelle 5.3 vollständig dargestellt:

| + | 0 | A | T | AT |
|---|---|---|---|---|
| 0 | 0 | A | T | AT |
| A | A | A | AT | AT |
| T | T | AT | T | AT |
| AT | AT | AT | AT | AT |

Tabelle 5.3: Operation + bei Software-Kategorien

#### Summe und partielle Ordnung für Aufgabe

Für das Attribut *Aufgabe* werden mehrere Tabellen angegeben, um die Summation festzulegen, da es viele Ausprägungen gibt. Insgesamt ist der Wert *neutral* das neutrale Element und *Subsystem* das maximale Element. Die Tabelle stellt folgenden Sachverhalt dar: Wenn ein Architekturelement mit einer für die Referenzarchitektur, aus der die Aufgaben stammen, *neutralen* Aufgabe zu einer der drei angegebenen Schichten hinzugefügt wird, ändert sich die Aufgabe jeweils nicht. Werden zwei verschiedene Schichten zusammengefügt entsteht immer ein Subsystem. Umgekehrt formuliert kann ein Subsystem aus den drei genannten Schichten bestehen.

| + | neutral | Datenhaltung | Anwendungskern | Präsentation | Subsystem |
|---|---|---|---|---|---|
| neutral | neutral | Datenhaltung | Anwendungskern | Präsentation | Subsystem |
| Datenhaltung | Datenhaltung | Datenhaltung | Subsystem | Subsystem | Subsystem |
| Anwendungskern | Anwendungskern | Subsystem | Anwendungskern | Subsystem | Subsystem |
| Präsentation | Präsentation | Subsystem | Subsystem | Präsentation | Subsystem |
| Subsystem | Subsystem | Subsystem | Subsystem | Subsystem | Subsystem |

Tabelle 5.4: Operation + bei Aufgabe: Schichten einer Drei-Schichtenarchitektur

Die Tabellen 5.4 bis 5.7 zeigen die Summation für eine durchgehaltene Drei-Schichtenarchitektur, wie sie häufig in betrieblichen Informationssystemen vorkommt. Unbeantwortet ist noch die Frage: Was

[16] Vgl. Beneken in [Ben06] sowie die Ausführungen zu Texturen (recurring [uniform] microstructure of its components) bei Jazayeri et al. [JRvdL02, S.23ff]

[17] Zum Entstehungsprozess von Referenzarchitekturen vgl. etwa [Ben06].

[18] vgl. dazu etwa die Ausführungen zu Metaarchitekturen von Malan und Bredemeyer [MB02]

| + | neutral | Entitätsverwalter | Zugriffsschicht | Datenhaltung |
|---|---|---|---|---|
| neutral | neutral | Entitätsverwalter | Zugriffsschicht | Datenhaltung |
| Entitätsverwalter | Entitätsverwalter | Entitätsverwalter | Datenhaltung | Datenhaltung |
| Zugriffsschicht | Zugriffsschicht | Datenhaltung | Zugriffsschicht | Datenhaltung |
| Datenhaltung | Datenhaltung | Datenhaltung | Datenhaltung | Datenhaltung |

Tabelle 5.5: Operation + bei Aufgabe: Datenhaltungsschicht

| + | neutral | A-Fall | Fassade | Anwendungskern |
|---|---|---|---|---|
| neutral | neutral | A-Fall | Fassade | Anwendungskern |
| A-Fall | A-Fall | A-Fall | Anwendungskern | Anwendungskern |
| Fassade | Fassade | Anwendungskern | Fassade | Anwendungskern |
| Anwendungskern | Anwendungskern | Anwendungskern | Anwendungskern | Anwendungskern |

Tabelle 5.6: Operation + bei Aufgabe: Anwendungskern

| + | neutral | Dialog | Dialogverwalter | Präsentation |
|---|---|---|---|---|
| neutral | neutral | Dialog | Dialogverwalter | Präsentation |
| Dialog | Dialog | Dialog | Präsentation | Präsentation |
| Dialogverwalter | Dialogverwalter | Präsentation | Dialogverwalter | Präsentation |
| Präsentation | Präsentation | Präsentation | Präsentation | Präsentation |

Tabelle 5.7: Operation + bei Aufgabe: Präsentationsschicht

ist das Ergebnis, wenn etwa ein Dialog mit einem Entitätsverwalter kombiniert wird? Hier sind zwei Antworten möglich: (1) die Referenzarchitektur, aus der die Aufgabendefinition stammt, verbietet diese direkte Kombination, damit wäre das Ergebnis der Summation undefiniert $\langle\rangle$. (2) Die Kombination von Komponenten mit beliebigen Aufgaben wird zugelassen, wie es oben für die Summation gefordert wird. Zur Erhaltung der ebenfalls geforderten Monotonie muss die Antwort damit *Entitätsverwalter + Dialog = Subsystem* lauten. Entsprechendes gilt für die Summation aller nicht in den Tabellen aufgeführten Ausprägungen von *Aufgabe*: Ergebnis ist immer *Subsystem*.

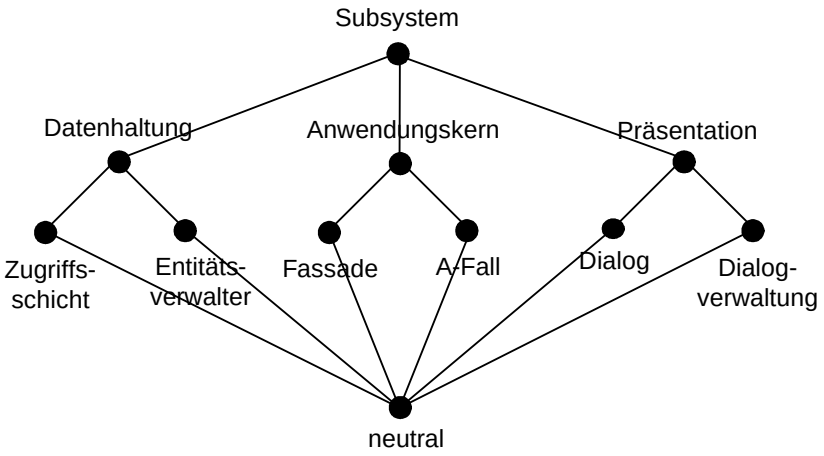


Abbildung 5.9: Ordnungsrelation für den Attributtyp Aufgabe

Die Abbildung 5.9 zeigt die Ordnungsrelation auf den Werten des Attributtyps *Aufgabe*. *Subsystem* ist darin maximales Element und *neutral* ist minimales Element. Eine Kante zwischen zwei Elementen zeigt, dass beide in Relation stehen. Das in der Abbildung oben stehende Element ist jeweils größer oder gleich dem darunter stehenden Element.

Die Ordnungsrelation beschreibt die *ist-Teil-von* Beziehung: A-Fälle und Fassaden sind beispielsweise Teile des Anwendungskerns und der Anwendungskern ist Teil eines Subsystems. Die Aufgabe *neutral* wurde künstlich hinzugefügt. Die Ordnungsrelation sagt hierzu aus, dass ein Element mit einer neutralen Aufgabe Teil von einem Element mit einer beliebigen Aufgabe sein darf.

## 5.6 Beschreibung des Verhaltens

In den vorangegangenen Abschnitten war nur von der Struktur einer logischen Architektur die Rede. Eine detaillierte Verhaltensbeschreibung ist in frühen Phasen eines Projektes, in denen die logische (Grob-)Architektur entsteht, noch nicht erforderlich. Die logische Architektur ist vielmehr als erste Entwurfsskizze zu betrachten, die mit wenig Aufwand erstellt wird und als Grundlage für die ersten Aufwandsschätzungen und die Projektplanung dient. Wenn die logische Architektur verfeinert wird, muss die Verhaltensbeschreibung ergänzt werden, denn laut Definition 3.2 ist das Verhalten Teil der Architektur(beschreibung).

Dieser Abschnitt skizziert, wie zur Verfeinerung des ersten Grobentwurfes eine Verhaltensbeschreibung ergänzt werden kann. Die Verhaltensbeschreibung verfolgt eine ähnliche Idee wie die Darstellung von Broy und Rumpe aus [BR07]: Jeder Blattkomponente in einer Konfiguration $C = (V, E, J, H)$ wird eine Moore-Maschine zugeordnet. Eine Moore-Maschine[19] ist ein endlicher Automat, der Ausgaben erzeugen kann. Die Ausgaben hängen dabei nur vom aktuellen Zustand der Maschine ab und nicht von der aktuellen Eingabe.

### Nachrichten

Die logischen Komponenten tauschen über logische Konnektoren Nachrichten aus. Die Menge der Nachrichten zu einer Konfiguration $C = (V, E, J, H)$ ist gegeben durch die Menge $\Sigma \subset MESSAGE$. Die Menge $\Sigma$ wird als Eingabe- und Ausgabealphabet der Moore-Maschinen verwendet, die den logischen Komponenten zugeordnet werden. Die Menge $MESSAGE$ ist die Menge aller denkbaren Nachrichten.

### Logische Konnektoren

Logische Konnektoren[20] sind nur noch zwischen Blättern $leafs(C)$ der Hierarchie $H$ erlaubt. Ein Konnektor $e \in E$ kann eine Teilmenge der Nachrichten transportieren[21]. Die Konnektoren leiten die Nachrichten unmittelbar weiter, sie speichern keine Nachrichten zwischen.

Es wird ein globaler Zeittakt unterstellt. Innerhalb eines Zeittakts schalten alle Moore-Maschinen gemeinsam und alle Ausgaben werden gleichzeitig erzeugt. Ein Zeittakt $t$ wird mit einer natürlichen Zahl dargestellt $t \in \mathbb{N}_0$. Damit ist $t_0 = 0$ der erste Zeittakt und $t \in \mathbb{N}$ ein folgender Zeittakt.

Innerhalb eines Zeittakts $t \in \mathbb{N}$ kann eine Nachricht, eine endliche Sequenz von Nachrichten oder auch keine Nachricht über einen Konnektor übertragen werden. Dies erlaubt es, dass die Moore-Maschinen asynchron arbeiten können[22].

Für einen Konnektor $e \in E$ kann eine Belegung $msg(e) \in \Sigma^*$ mit Nachrichtensequenzen in einem gegebenen Zeittakt formuliert werden.

$msg : E \to \Sigma^*$.

Mit $MSG(E)$ wird für eine Menge von Konnektoren $E$ die Menge aller denkbaren Belegungen bezeichnet. Einer Konnektormenge $E$ ist in einem Zeittakt also eine Belegung $msg \in MSG(E)$ zugeordnet. Abläufe können über Sequenzen von Belegungen $msg_1 \cdots msg_n$ dargestellt werden.

Zu einer Komponente $v \in V$ sind die eingehenden Konnektoren gegeben über $incoming(C, v)$ und die ausgehenden Konnektoren über $outgoing(C, v)$. Damit bezeichnet $MSG(incoming(C, v))$ alle denkbaren Belegungen der eingehenden Konnektoren mit Nachrichtensequenzen und

[19] Vgl. z.B. [HU90], hier wird eine verenfachte Fassung ohne Trennung von des Ein- und Ausgabealphabetes verwendet.
[20] Entsprechen dem Konzept der Kanäle aus [BR07].
[21] Dies kann als Datentyp des Konnektors interpretiert werden.
[22] Vgl. Grosu und Rumpe [GR95]

$MSG(outgoing(C, v))$ alle denkbaren Belegungen der ausgehenden Konnektoren mit Nachrichtensequenzen.

## Logische Komponenten

Jeder logischen Komponente $v \in leafs(C)$ einer Konfiguration $C = (V, E, J, H)$, die Blatt der Hierarchie $H$ ist, wird nun eine Moore-Maschine $M_v = (Q_v, \Sigma, \delta_v, \lambda_v, q_{0_v})$ zugeordnet. Hierbei ist:

- $Q_v \subseteq STATE$ die Menge der internen Zustände der Komponente $v \in leafs(C)$ mit $STATE$ als Menge aller denkbaren Zustände. Weiterhin sind die Zustandsmengen aller logischen Komponenten diskunkt: $\forall v, w \in leafs(C)$ mit $v \neq w : Q_v \cap Q_w = \emptyset$.
- $\Sigma$ ist das Ein- und Ausgabealphabet aller Moore-Maschinen, also die Menge der Nachrichten, welche die Komponenten senden und empfangen können.
- $\delta_v : (Q_v \times MSG(incoming(C, v)) \rightarrow \wp(Q_v)$ ist die Zustandsübergangsfunktion, die ankommende Nachrichtensequenzen verarbeitet und darüber eine Menge von Folgezuständen der Moore-Maschine festlegt. Die Übergangsfunktion ist nicht deterministisch formuliert, um einerseits Unterspezifikation zu erlauben und andererseits die Komposition von Moore-Maschinen zu vereinfachen.
- $\lambda_v : Q_v \rightarrow MSG(outgoing(C, v))$ ist die Funktion, welche die Ausgabe der Moore-Maschine erzeugt.
- $q_{0_v} \in Q_v$ ist der Startzustand der Moore-Maschine.

Die Zustandsübergangsfunktion $\delta_v$ verarbeitet in einem Zeittakt die Nachrichtensequenzen, die an den eingehenden Konnektoren ankommen. Die Folgezustände $q' \in Q_v$ hängt vom aktuellen Zustand $q \in Q_v$ und der Belegung $msg_{in} \in MSG(incoming(C, v))$ ab. Im gleichen Zeittakt erzeugt die Moore-Maschine über $\lambda_v$ (als Ergebnis vorheriger Berechnungen) eine Belegung der ausgehenden Konnektoren $msg_{out} \in MSG(outgoing(C, v))$. Diese können im nächsten Zeittakt von anderen Moore-Maschinen verarbeitet werden. Die Ausgabe hängt nur vom aktuellen Zustand $q \in Q_v$ ab.

## Komposition und Gesamtverhalten

Das Gesamtverhalten einer logischen Architektur kann durch die hierarchische Komposition der einzelnen Moore-Maschninen beschrieben werden. Hier wird die Komposition der Moore-Maschinen $M_j = (Q_j, \Sigma, \delta_j, \lambda_j, q_{0_j}), (j = 1, 2)$ von zwei Komponenten $v_j, (j = 1, 2)$ vorgeführt. Die Komposition von mehr als zwei Komponenten geschieht analog.

Eine Komponente $x$ bestehe aus zwei Teilkomponenten $v_j, (j = 1, 2)$. Der Zustandsraum der zusammengesetzten Komponente ergibt sich aus dem Kreuzprodukt der beiden Zustandsräume der Moore-Maschninen :

$Q_x := Q_1 \times Q_2$.

Bei mehr Teilkomponenten ist das Kreuzprodukt entsprechend größer. Der Startzustand der zusammengesetzten Komponente ist:

$q_{0_x} := (q_{0_1}, q_{0_2})$

Die Zustandsübergangsfunktion $\delta = \delta_1 \otimes \delta_2$ mit $\delta_j : (Q_j \times MSG(incoming(C, v_j)) \rightarrow \wp(Q_j)$ wird wie folgt berechnet: Sei $in \in MSG(incoming(C, \{v_1, v_2\}))$ die Belegung an den eingehenden Konnektoren der Komponente $x$ und $(q_1, q_2)$ der aktuelle Zustand der zusammengesetzten Moore-Maschine, dann ist:

$$\delta((q_1,q_2),in) := \{(q_1',q_2')|\exists z \in MSG(incoming(C,v_1)\cup incoming(C,v_2)) : in = z|incoming(C,\{v_1,v_2\}) \wedge q_j' \in \delta_j(q_j, z|incoming(C,v_j)) \text{ für } j=1,2\}$$

Der Teilausdruck $z|incoming(C,v_j)$ bezeichnet die Belegungen eingehenden Konnektoren der Komponente $v_j$, dies ergibt sich durch die Einschränkung der Belegungen aus $z$ auf die genannten Konnektoren.

Die Funktion $\lambda = \lambda_1 \otimes \lambda_2$ mit $\lambda_j : Q_j \rightarrow MSG(incoming(C,v_j))$ ist gegeben über:

$\lambda((q_1,q_2)) := out \in MSG(outgoing(C,\{v_1,v_2\}))$ mit
$\lambda_j(q_j)|outgoing(C,\{v_1,v_2\}) = out|outgoing(C,v_j)$ für $j=1,2$

Die vereinigte Moore-Maschine $M_x$ für die zusammengesetzte Komponente $x$ ergibt sich damit aus den Moore-Maschinen der Einzelkomponenten $v_j, (j=1,2)$ mit
$M_x = (Q_1 \times Q_2, \Sigma, \delta_1 \otimes \delta_2, \lambda_1 \otimes \lambda_2, (q_{0_1}, q_{0_2}))$.

Die Moore-Maschine, welche die gesamte logische Architektur beschreibt, muss Eingaben aus seiner Umgebung erhalten. Dies ist in der hier dargestellten Konfiguration nicht möglich, da sie nur geschlossene Systeme modelliert. Über Konnektoren zur Umwelt und Nachrichtensequenzen, welche die Eingaben aus der Umwelt modellieren, kann die Konfiguration simuliert werden.

## 5.7 Logische Architekturen

Mit den oben eingeführten Konfigurationen, Nutzungsszenarios und Attributen werden logische Architekturen formalisiert. Eine logische Architektur wird beschrieben durch das Tupel

$\mathcal{L} = (C, A, value, UC, S)$.

Darin ist $C = (V, E, J, H) \in CONFIGURATION$ eine wohlgeformte Konfiguration. $V$ ist eine Menge von Komponenten. Die Komponenten werden über Konnektoren $E$ verbunden. Eine Zuordnungsfunktion $J$ ist notwendig, da zwei Komponenten aus $V$ über mehrere Konnektoren aus $E$ verbunden sein können. Die Relation $H$ beschreibt den wohlgeformten hierarchischen Aufbau der Komponenten aus ihren jeweiligen Teilkomponenten, $H$ kann als zweite Kantenmenge aufgefasst werden, diese Kanten werden in den Abbildungen 5.2 und 5.5 jeweils explizit dargestellt. Weiterhin ist

- $A \subseteq ATTRIBUTE$ eine Menge von Attributen, welche den Elementen der Konfiguration $C$ zugeordnet werden und
- $value$ eine Belegung der Attribute für die Komponenten und Konnektoren, zusätzlich ist
- $UC$ eine Menge von Anwendungsfällen, die von der Konfiguration $C$ (teilweise) implementiert werden.
- $S$ eine Zuordnungsfunktion, die jedem Anwendungsfall genau ein Nutzungsszenario zuweist.

Die Menge aller denkbaren logischen Architekturen wird mit $LA$ bezeichnet. Die logischen Architekturen können auch den Aufbau einer Komponente oder eines Subsystems beschreiben. Damit sind logische Architekturen flexibel einsetzbar.

#### Logische Systemarchitektur

Wenn eine logische Architektur ein IT-System mit seiner Umgebung beschreibt, dann wird sie *logische Systemarchitektur* genannt. In einer logischen Systemarchitektur ist die Konfiguration $C$ eine erweiterte Systemkonfiguration. Darin ist die Pseudokomponente $\omega$ die Wurzel der Hierarchie $H$. In

der ersten Dekompositionsebene von $C$ befinden sich das System $\sigma$ sowie Nachbarsysteme und Nutzergruppen, $\sigma, \omega \in V$. Die logische Systemarchitektur ist ein Spezialfall der logischen Architektur. Sie wird mit $\mathcal{LS} = (C, A, value, UC, S)$ bezeichnet. Die Menge aller logischen Systemarchitekturen wird mit $LSA$ bezeichnet. Und es gilt

$LSA \subset LA$.

### Metamodell

Die Abbildung 5.10 zeigt das Metamodell für die bislang graphentheoretisch dargestellten logischen Architekturen:

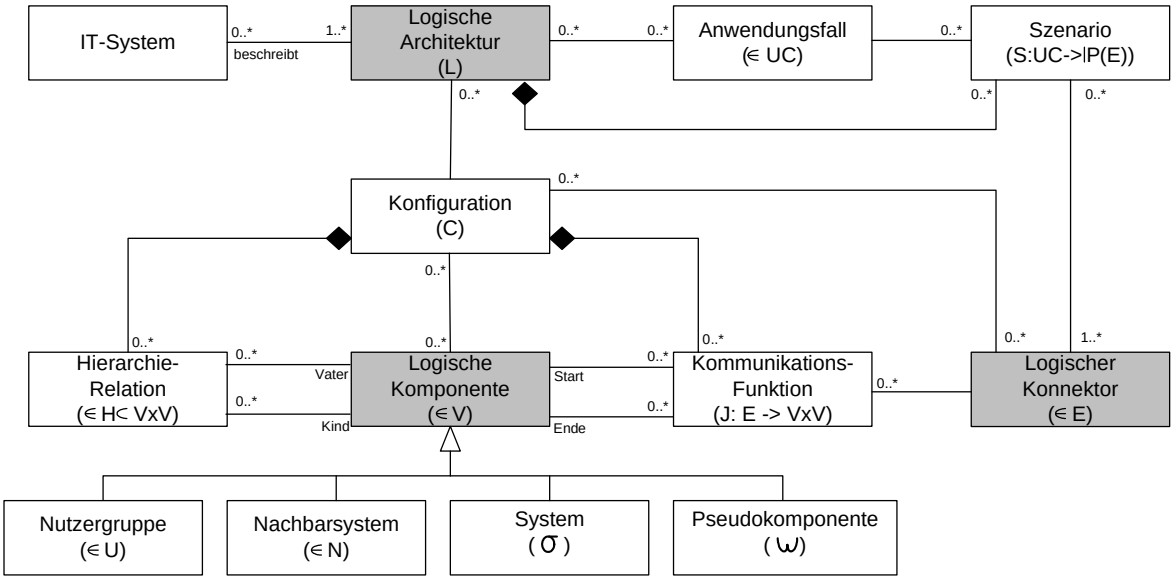


Abbildung 5.10: Metamodell für logische Architekturen

Das Metamodell weist mehrere Besonderheiten auf: Der hierarchische Aufbau und die Kommunikationsbeziehungen werden jeweils die Elemente *Hierarchierelation* und *Kommunikationsfunktion* modelliert. Die Instanzen von *Hierarchierelation* stellen die Menge $H$ dar und die Instanzen von *Kommunikationsrelation* modellieren die Zuordnungsfunktion $J$. Darüber wird erreicht, dass logische Komponenten und logische Konnektoren in mehreren Konfigurationen enthalten sein können, während Hierarchie- und Kommunikationsrelation spezifisch für eine Konfiguration sind. Daher sind diese über eine Kompositionsbeziehung mit der Konfiguration verbunden. Jeder Kommunikationsrelation ist genau ein logischer Konnektor ($E$) zugeordnet.

Eine Sonderrolle nimmt die Pseudokomponente $\omega$ ein, sie darf nicht mit anderen Komponenten in Kommunikationsrelation stehen und darf nur in der Rolle Vater mit anderen Komponenten in Hierarchierelation stehen. Zusätzlich darf sie genau wie das System $\sigma$ nur einmal in der Konfiguration vorkommen. Nutzergruppen und Nachbarsysteme werden (vereinfachend) als Spezialisierung logischer Komponenten aufgefasst. Beide dürfen nur in Hierarchierelation zu $\omega$ stehen, wobei $\omega$ die Rolle *Vater* hat und Nutzergruppen und Nachbarsysteme die Rolle *Kind*.

Einer logischen Architektur ist eine Menge von Anwendungsfällen $UC$ zugeordnet. Diese werden über die Funktion $S$ auf Teilmengen der Konnektoren abgebildet.

Logische Komponenten $V$ und logische Konnektoren $E$ sind Architekturelemente. Architekturelementen können Attribute und deren Werte zugeordnet werden. Die Attribute sind nur indirekt über die Belegung den Architekturelementen zugeordnet. Dies ist in Abbildung 5.11 dargestellt. Diesel-

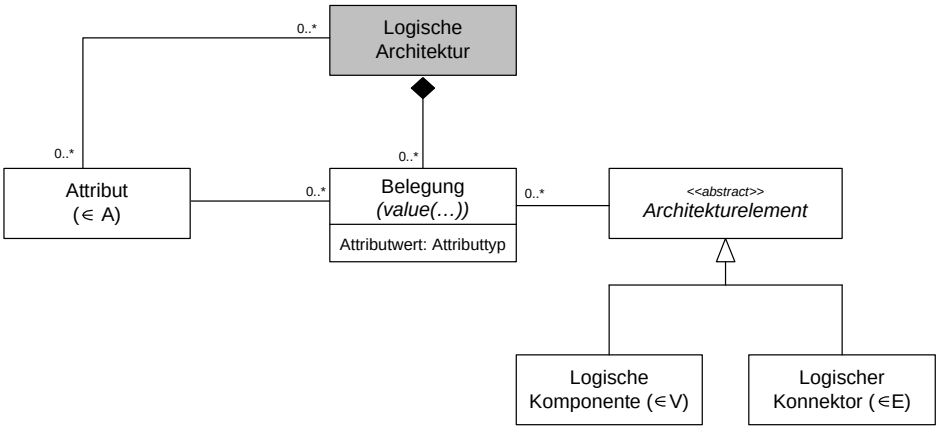


Abbildung 5.11: Metamodell für Attribute und ihre Belegung

ben Komponenten und Konnektoren können innerhalb einer anderen logischen Architektur $\mathcal{L}'$ verwendet werden und dort andere Attribute und andere Attributwerte haben. Diese Eigenschaft ist Grundlage für die in Kapitel 6 definierten Projektionen.

### Anwendungsgebiet

Eine logische (System-)Architektur wird in der vorliegenden Arbeit als Gerüst verwendet, auf das Informationen aus der Projektplanung, aus der Implementierung oder aus dem Deployment projiziert werden. Informationen über die Struktur einer Software werden mit Planungsdaten und anderen Informationen kombiniert. Abbildung 5.12 stellt die Idee der logischen Architektur als Kernmodell grafisch dar.

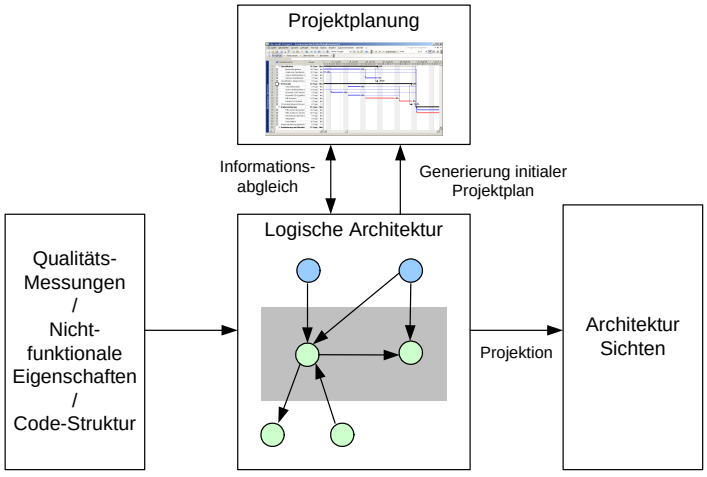


Abbildung 5.12: Logische Architektur als Kernmodell

Die Darstellung der logischen Architektur kann so die Grundlage für ein zentrales Repository sein, in dem Informationen über ein IT-System entlang der logischen Architektur abgelegt und aufgefunden werden. Vorschläge für die Abbildungen von Daten aus einem Projektplan auf die logische Architektur werden im Kapitel 8 gemacht.

## 5.8 Zusammenfassung

Die theoretischen Grundlagen zur Beschreibung logischer Architekturen als gerichtete Multigraphen werden in diesem Kapitel gelegt. Elemente einer logischen Architektur sind Komponenten, Konnektoren, Anwendungsfälle, Nutzungsszenarios sowie Attribute und ihre Belegung. Komponenten haben eine definierte Aufgabe innerhalb eines IT-Systems, sie sind Einheiten der Planung und abgrenzbare Teile der späteren Implementierung. Konnektoren stellen Kommunikationsbeziehungen zwischen je zwei Komponenten dar.

Die Kommunikationsbeziehungen innerhalb einer logischen Architektur werden als gerichteter Multigraph dargestellt, in dem die Komponenten die Knoten und die Konnektoren die Kanten sind. Die hierarchische Strukturierung der Komponenten wird über einen gerichteten Baum dargestellt, in dem eine Kante die *besteht aus* Relation modelliert. Hierarchiebaum und Kommunikationsmultigraph werden zusammen als Konfiguration bezeichnet. Anwendungsfälle werden in der Architektur als Nutzungsszenarios modelliert, diese sind Teilmengen der Konnektoren. Zu einem Nutzungsszenario gehören alle Konnektoren, die durchlaufen werden, um einen Anwendungsfall zu implementieren, der diesem Szenario zugeordnet ist. Weiterhin kann die Beschreibung einer logischen Architektur über Attribute mit zusätzlichen Informationen versehen werden. Die Attribute werden Komponenten und Konnektoren zugewiesen. Das folgende Kapitel definiert Abbildungen (Projektionen) auf Beschreibungen logischer Architekturen.

# Kapitel 6

# Projektionen

Die Architektur eines IT-Systems wird in der Regel über Sichten dargestellt. Sichten stellen bestimmte Systemaspekte, ausgerichtet auf eine Stakeholder-Gruppe oder einen Problembereich dar. Dieses Kapitel schlägt graphentheoretisch definierte Projektionen vor, um Architektursichten zu erzeugen.

Mit den vorgeschlagenen Projektionen kann die Erzeugung von Architektursichten definiert werden, die bislang nur pragmatisch begründet waren, etwa die A- und T-Architekturen nach Siedersleben [Sie04]. Die Projektionen berücksichtigen Zusatzinformationen, projizieren diese mit und fassen bei Vergröberungen Attributwerte zusammen. Damit können Architektursichten erzeugt werden, die etwa die Zusammenarbeit von Projektleiter und Architekt unterstützen, indem sie Planungsinformationen zusammen mit der Architektur zeigen.

Der Abschnitt 6.1 motiviert zunächst die Erzeugung von Architektursichten über die Projektionen, die in diesem Kapitel definiert sind. Die Abschnitte 6.2 bis 6.4 definieren drei Arten von Projektionen. Die Vergröberungsprojektion fasst Komponenten und Konnektoren zusammen und berücksichtigt dabei den hierarchischen Aufbau der Architektur, die Zusammenfassungsprojektion fasst Komponenten und Konnektoren nach frei definierbaren Kriterien zusammen und die Auswahlprojektion wählt wie eine Query oder ein Filter aus einer Architektur bestimmte Komponenten und Konnektoren aus. Der Abschnitt 6.5 fasst schließlich die wichtigsten Aspekte der Projektionen zusammen.

## Übersicht

## 6.1 Erzeugung von Sichten mit Projektionen

Die Architektur eines IT-Systems wird in der Regel über Sichten dargestellt. Sichten stellen bestimmte Systemaspekte, ausgerichtet auf eine Stakeholder-Gruppe oder einen Problembereich dar. Die vorliegende Arbeit definiert Projektionsfunktionen zur Erzeugung von Sichten. Eine logische Architektur dient dazu als Ausgangspunkt. Ergebnis einer Projektion ist immer eine logische Architektur, so dass die Projektionen verkettet werden können. Dieses Kapitel definiert drei Grundprojektionen:

**Vergröberungsprojektion $\Phi$:** Komponenten und Konnektoren werden über eine Vergröberungsprojektion[1] zusammengefasst. Als Kriterien für die Zusammenfassung können verwendet werden:

- Hierarchischer Aufbau: Damit können Sichten in verschiedenen Detaillierungsebenen[2] erzeugt werden.
- Mehrfachkonnektoren: Damit wird die Zahl der Konnektoren in einer logischen Architektur reduziert, um die Darstellung übersichtlicher zu machen.

**Zusammenfassungsprojektion $\Psi$:** Die Zusammenfassungsprojektion $\Psi$ aggregiert wie auch $\Phi$ Komponenten, dazu wird jedoch nicht der hierarchische Aufbau sondern eine beliebige Äquivalenzrelation $\mathcal{R}$ auf der Menge der Komponenten verwendet: Einer Menge äquivalenter Komponenten wird eine durch eine neue Komponente ersetzt, die diese Äquivalenzklasse repräsentiert.

**Auswahlprojektion $\Xi$:** Im Gegensatz zu $\Phi$ oder $\Psi$ wählt die Auswahl $\Xi$ nur bestimmte Komponenten, Konnektoren oder Attribute einer logischen Architektur aus. Als Kriterium für die Auswahl wird ein einstelliges Prädikat $\mathcal{Q}$ auf der Menge Komponenten $\mathcal{Q}_V : V \to \mathbb{B}$, Konnektoren $\mathcal{Q}_E : E \to \mathbb{B}$ und Attribute $\mathcal{Q}_A : A \to \mathbb{B}$ verwendet.

Die Projektionen sind im Allgemeinen mit Informationsverlust verbunden, da Komponentenmengen oder Konnektormengen zusammengefasst oder ganz weggelassen werden. Die Projektionen können daher nicht umgekehrt werden. Änderungen in einem Projektionsbild können nicht zurück auf das Urbild übertragen werden. Die über Projektionen erzeugten Sichten sind damit nur lesend.

Die Projektionen sind in verschiedenen Varianten definiert als Abbildungen $\Phi$, $\Psi$ und $\Xi$ einer logischen Architektur $\mathcal{L} \in LA$ in eine andere $\mathcal{L}' \in LA$. Diese Projektionen haben in der Regel mindestens einen weiteren Parameter, etwa eine natürliche Zahl, eine Äquivalenzrelation $\mathcal{R}$ auf der Menge der Komponenten oder ein Auswahlprädikat $\mathcal{Q}$ auf der Menge der Komponenten, Konnektoren oder Attribute:

$\Phi, \Psi, \Xi : LA \times \cdots \to LA.$

Die Wohlgeformtheit der Konfiguration spielt bei den Abbildungen eine wichtige Rolle: Die Zusammenfassungsprojektion fasst beispielsweise Blätter der Komponentenhierarchie mit derselben Vaterkomponente zusammen. Hat eine Blattkomponente mehrere Vaterkomponenten, wie dies in einer nicht wohlgeformten Konfiguration vorkommt, kann die Zusammenfassung unterhalb einer der Vaterkomponente offenbar nicht mehr sinnvoll durchgeführt werden.

#### Beispielarchitektur

Um die in diesem Kapitel vorgestellten Projektionen zu veranschaulichen, wird die Komponente *Kundenverwaltung* des *Bringdienstsystems* verwendet. Diese Konfiguration wird $KV$ genannt und ist in Abbildung 6.1 dargestellt. Sie wird auch in den nachfolgenden beiden Kapiteln als Teil der Konfiguration des Bringdienstsystems verwendet.

Die Konfiguration $KV = (V_{KV}, E_{KV}, J_{KV}, H_{KV})$ kann auch graphentheoretisch beschrieben werden. Sie ist gegeben durch:

[1] auch Aggregation genannt
[2] auch Hierarchieebenen oder Dekompositionsebenen genannt

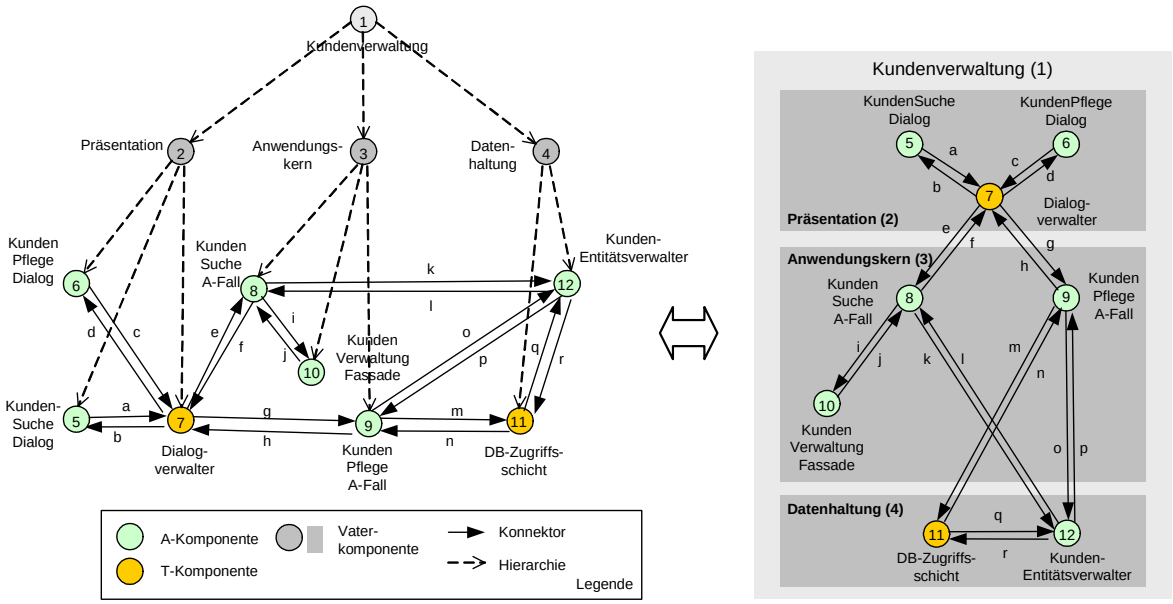


Abbildung 6.1: Beispielkomponente Kundenverwaltung

- $V_{KV} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$
- $E_{KV} = \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r\}$
- $J_{KV} = \{(a, (5, 7)), (b, (7, 5)), (c, (6, 7)), (d, (7, 6)), (e, (7, 8)), (f, (8, 7)), (g, (7, 9)), (h, (9, 7)), (i, (8, 10)), (j, (10, 8)), (k, (8, 12)), (l, (12, 8)), (m, (9, 11)), (n, (11, 9)), (o, (9, 12)), (p, (12, 9)), (q, (11, 12)), (r, (12, 11))\}$
- $H_{KV} = \{(1, 2), (1, 3), (1, 4), (2, 5), (2, 6), (2, 7), (3, 8), (3, 9), (3, 10), (4, 11), (4, 12)\}$

Die Konfiguration $KV$ sei in einer logischen Architektur $\mathcal{L}_{KV}$ enthalten. Die Anwendungsfälle $UC_{KV}$ können über die Nutzungsszenarios $S_{KV}$ die Konnektoren aus $KV$ verwenden. $S_{KV}$ sei die Zuordnungsfunktion, die einem Anwendungsfall $uc \in UC_{KV}$ ein Nutzungsszenario $\subseteq E$ zuweist. $UC_{KV} = \{uc_1, uc_2, uc_3\}$ besteht aus drei Anwendungsfällen, denen jeweils Konnektormengen zugeordnet werden:

1. Jemand sucht einen Kunden in der Kundenverwaltung über den Suchdialog und bekommt das Suchergebnis vom Bringdienstsystem mitgeteilt:

   $S_{KV}(uc_1) = \{a, e, k, r, q, l, f, b\}$

2. Ein Kunde wird direkt über die Kundenpflege angelegt:

   $S_{KV}(uc_2) = \{c, g, o, r, q, p, h, d\}$

3. Die Komponente Bestellverwaltung ruft die Kundenverwaltung Fassade auf und benötigt Daten zu einem Kunden:

   $S_{KV}(uc_3) = \{j, k, r, q, l, i\}$

Den Teilkomponenten der *Kundenverwaltung* werden *Software-Kategorie* und *Aufgabe* als Attribute mit den gleichnamigen Attributtypen zugewiesen. Das Attribut Aufgabe orientiert sich dabei an der Referenzarchitektur Quasar [Sie02a]: Die Komponenten können beispielsweise die Aufgaben *Dialog*, *A-Fall*, *Fassade* und *Entitätsverwalter* sowie *Zugriffsschicht* oder *Dialogverwaltung* haben. Die Operationen

| Id | Name | Software-Kategorie | Aufgabe |
|---|---|---|---|
| 1 | Kundenverwaltung | 0-Software | Subsystem |
| 2 | Dialogschicht | 0-Software | Präsentation |
| 3 | Anwendungskern | 0-Software | Anwendungskern |
| 4 | Datenhaltung | 0-Software | Datenhaltung |
| 5 | KundenSuche Dialog | A-Software | Dialog |
| 6 | KundenPflege Dialog | A-Software | Dialog |
| 7 | Dialogverwalter | T-Software | Dialogverwalter |
| 8 | KundenSuche A-Fall | A-Software | A-Fall |
| 9 | KundenPflege A-Fall | A-Software | A-Fall |
| 10 | KundenVerwaltung Fassade | A-Software | Fassade |
| 11 | DB-Zugriffsschicht | T-Software | Zugriffsschicht |
| 12 | Kunden Entitätsverwalter | A-Software | Entitätsverwalter |

Tabelle 6.1: Belegung von Software-Kategorie und Aufgabe für die Komponenten der Kundenverwaltung $KV$

$+$ und $\leq$ für die Attributtypen Software-Kategorie und Aufgabe werden in Abschnitt 5.5 definiert. Die Belegung $value$ wird durch die Tabelle 6.1 angegeben.

Zusätzlich gilt für alle Konnektoren aus $KV$: $\forall e \in E : value(e$,*Software-Kategorie*) = 0-Software sowie $\forall e \in E : value(e, Aufgabe) = neutral$. Den Konnektoren sind also jeweils die neutralen Elemente der Attributtypen zugewiesen. Damit haben die Konnektoren auf die unten dargestellten Summationen von Belegungen keinen Einfluss.

## 6.2 Vergröberungsprojektion (Aggregation)

Die Vergröberungsprojektion $\Phi$ fasst in einer Konfiguration $C$ bzw. in einer logischen Architektur $\mathcal{L}$ Komponenten aus $V$ und Konnektoren aus $E$ zusammen. Die Vergröberungsprojektion wird auch Aggregation genannt. Die Attributbelegung wird bei der Vergröberung berücksichtigt. Die Abbildung 6.2 zeigt eine Vergröberung von Komponenten innerhalb der Komponente *Kundenverwaltung*: alle Komponenten und Konnektoren unterhalb der ersten Dekompositionsebene werden zusammengefasst, so dass die Schichtung der Komponente *Kundenverwaltung* sichtbar wird.

### 6.2.1 Vergröberung in einer Hierarchieebene

#### Vergröberung einer Konfiguration

Zur Erzeugung einer aggregierten Sicht auf eine Konfiguration $C$ wird die Funktion $\Phi_{\mathbb{N}}$ definiert, mit

$\Phi_{\mathbb{N}} : CONFIGURATION \times \mathbb{N}_0 \rightarrow CONFIGURATION.$

Diese entfernt alle Komponenten aus einer Konfiguration, welche unterhalb einer angegebenen Hierarchieebene $n \in \mathbb{N}_0$ liegen. Ist die Konfiguration $C = (V, E, J, H)$ eine erweiterte Systemkonfiguration, gibt es zwei Sonderfälle: Ebene $n = 0$, welche nur die Pseudokomponente $\omega$ enthält und die Ebene $n = 1$, die das System $\sigma$ sowie Nachbarsysteme und Nutzergruppen enthält.

Wenn alle Komponenten bis auf die Wurzelkomponente $\omega$ entfernt werden, ergibt sich folgende Konfiguration:

$\Phi_{\mathbb{N}}(C, 0) := (\{\omega\}, \emptyset, J_\emptyset, \emptyset)$.

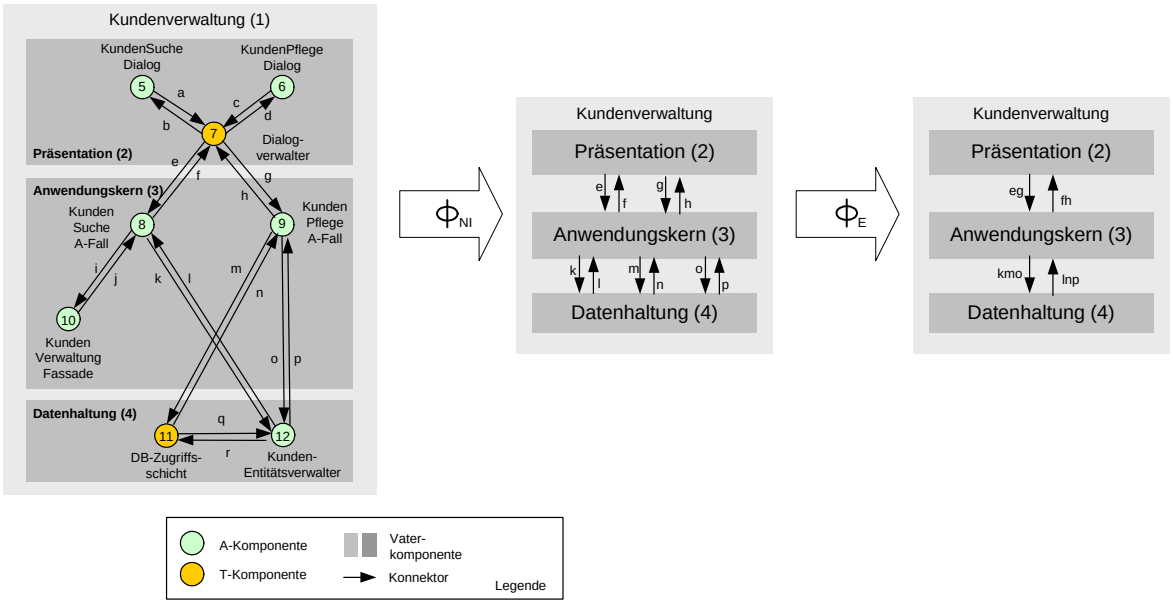


Abbildung 6.2: Komponenten unterhalb der ersten Hierarchieebene werden ausgeblendet.

Wenn die Vergröberung auf der ersten Ebene durchgeführt wird, ergibt sich eine Darstellung der Systemebene[3]. Diese Darstellung zeigt das System $\sigma$, alle Nachbarsysteme $N \subset V$ und Nutzergruppen $U \subset V$ sowie deren Kommunikationsbeziehungen $B$. Damit ist;

$\Phi_{\mathbb{N}}(C, 1) := (U \cup N \cup \{\sigma, \omega\}, B, J|_B, \bigcup_{v \in U \cup N \cup \{\sigma\}} (\omega, v))$.

$B \subseteq E$ sei die Menge der Verbindungen zwischen dem IT-System $\sigma$ und den Nutzergruppen und Nachbarsystemen, mit $\forall e \in B : J(e) \in ((N \cup U) \times (V \setminus (N \cup U))) \cup (V \setminus (N \cup U)) \times (N \cup U))$.

Zur Definition der Vergröberungsprojektion wird zunächst eine Vorgängerfunktion[4] innerhalb der Hierarchie benötigt. Diese liefert für die wohlgeformte Konfiguration $C = (V, E, J, H)$ den Vorgänger einer Komponente auf der Ebene $n \in \mathbb{N}_0$ oder die Komponente selbst, wenn sie oberhalb der genannten Ebene liegt:

$ancestor : V \times \mathbb{N}_0 \to V$. Für alle $v \in V$ und $n \in \mathbb{N}_0$:

$$ancestor(v, n) = \begin{cases} v: & depth(C, v) \leq n \\ x: & depth(C, x) = n \wedge v \in allparts(C, x) \end{cases}$$

Alle Komponenten unterhalb der angegebenen Ebene $n$ werden bei der Vergröberung entfernt und die Konnektoren werden auf ihre Vorgänger auf der angegebenen Ebene umgehängt. Eine Ausnahme bilden die Konnektoren, die bei der Vergröberung Kreise der Länge 1 bilden würden, da ihre Anfangspunkte und ihre Endpunkte denselben Vorgänger auf der Ebene $n$ haben.

Zur Berechnung dieser zu entfernenden Konnektoren wird zunächst die Hilfsfunktion *levelinternal* definiert. Konnektoren zwischen je zwei Komponenten, die beide auf einer Ebene $i > n$ liegen *und* die eine gemeinsame Vaterkomponente auf einer Ebene $n$ haben, werden durch die Funktion $levelinternal(C, n)$ ermittelt.

$levelinternal : CONFIGURATION \times \mathbb{N}_0 \to \wp(CONNECTOR)$

Zur Definition von $levelinternal$ wird die Hilfsfunktion $depth$ verwendet. Alle Komponenten der Konfiguration $C$ auf der gesuchten Dekompositionsebene $n \in \mathbb{N}_0$ sind über $\{x \in V | depth(C, x) = n\}$

[3] vgl. Abschnitt 7.1

[4] Seifert definiert in [Sei08] mithilfe einer ähnlichen ancestor-Funktion Sichten auf Implementierungsarchitekturen. Hier wird analog verfahren.

gegeben. Für jede dieser Komponenten werden alle Kindkomponenten über $allparts$ berechnet. Von dieser Komponentenmenge werden jeweils die internen Konnektoren berechnet, dies geschieht über die im vorhergehenden Kapitel eingeführte Funktion $internal$:

$levelinternal(C, n) := \bigcup_{x \in V | depth(C,x)=n}(internal(C, allparts(C, x)))$

Die Menge der internen Konnektoren der Beispielkonfiguration $KV$ der Ebene 1 ist gegeben über $levelinternal(KV, 1) = \{a, b, c, d, i, j, q, r\}$, da $\{a, b, c, d\} = internal(KV, 2)$, $\{i, j\} = internal(KV, 3)$ und $\{q, r\} = internal(KV, 4)$. Diese Konnektoren werden aus $E_{KV}$ bei der Vergröberung entfernt.

Über die beiden Hilfsfunktionen kann nun die Vergröberungsfunktion $\Phi_{\mathbb{N}}$ definiert werden. Für die Konfiguration $C = (V, E, J, H)$ und $n \in \mathbb{N}_0$ gibt eine Konfiguration $\Phi_{\mathbb{N}}(C, n) = C' = (V', E', J', H')$ mit:

- $V' := \{x \in V | depth(C, x) \leq n\}$, d.h. nur die Komponenten bis zur Ebene $n$ bleiben erhalten.
- $E' := E \setminus levelinternal(C, n)$, d.h. alle internen Konnektoren der entfernten Teilbäume entfallen.
- $J'(e) := (ancestor(v, n), ancestor(w, n))$ wenn $J(e) = (v, w)$ und $e \in E'$.
- $H' := H \cap (V' \times V')$

Die hierarchische Struktur $H'$ ergibt sich direkt durch Einschränkung von $H$ auf $V'$. Damit hat $H'$ höchstens noch die Tiefe $n$. Die Bildkonfiguration $C'$ ist wohlgeformt, denn die hierarchische Struktur $H'$ geht aus der wohlgeformten Hierarchie $H$ durch Einschränkung auf $V' \times V'$ hervor.

Für die Beispielkonfiguration $KV$ aus Abbildung 6.1 ergibt sich für die Ebene 1: $KV' = \Phi_{\mathbb{N}}(KV, 1) = (V_{KV'}, E_{KV'}, J_{KV'}, H_{KV'})$ mit

- $V_{KV'} = \{1, 2, 3, 4\}$, d.h. nur noch Komponenten auf den Ebenen 0 und 1.
- $E_{KV'} = \{e, f, g, h, k, l, m, n, o, p\}$, d.h. interne Konnektoren der Komponenten auf der Ebene 1 $levelinternal(KV, 1) = \{a, b, c, d, i, j, q, r\}$ sind entfallen.
- $J_{KV'} = \{(e, (2, 3)), (f, (3, 2)), (g, (2, 3)), (h, (3, 2)), (k, (3, 4)),$
  $(l, (4, 3)), (m, (3, 4)), (n, (4, 3)), (o, (3, 4)), (p, (4, 3))\}$
  d.h. die Zuordnungsfunktion ist gemäß Abbildungsvorschrift angepasst worden. Die in $E_{KV'}$ verbliebenden Konnektoren werden auf ihre Vorgänger auf der Ebene 1 umgehängt, etwa $J_{KV'}(e) = (2, 3)$, denn $J_{KV}(e) = (7, 8)$ und $2 = ancestor(7, 1)$ sowie $3 = ancestor(8, 1)$.
- $H_{KV'} = \{(1, 2), (1, 3), (1, 4)\}$

Diese Vergröberung ist in Abbildung 6.3 dargestellt.

#### Vergröberung logischer Architekturen

Sei $\mathcal{L} = (C, A, value, UC, S)$ eine logische Architektur und $\Phi_{\mathbb{N}} : LA \times \mathbb{N}_0 \rightarrow LA$ die Vergröberungsprojektion und $\Phi_{\mathbb{N}}(\mathcal{L}, n) := (C', A', value', UC', S') = \mathcal{L}'$.

Sei $C = (V, E, J, H)$ die Konfiguration von $\mathcal{L}$, dann ist

$C' := \Phi_{\mathbb{N}}(C, n) = (V', E', J', H')$ die Konfiguration von $\mathcal{L}'$.

Für die Beispielkonfiguration $KV$ aus Abbildung 6.1 ergibt sich für die Ebene 1: $KV' = \Phi_{\mathbb{N}}(KV, 1) = (V_{KV'}, E_{KV'}, J_{KV'}, H_{KV'})$ wie oben dargestellt.

Die Menge $UC$ stellt die Anwendungsfälle dar und die Funktion $S$ bildet diese auf Nutzungsszenarios ab. Während der Vergröberung einer wohlgeformten Konfiguration auf einer bestimmten Ebene

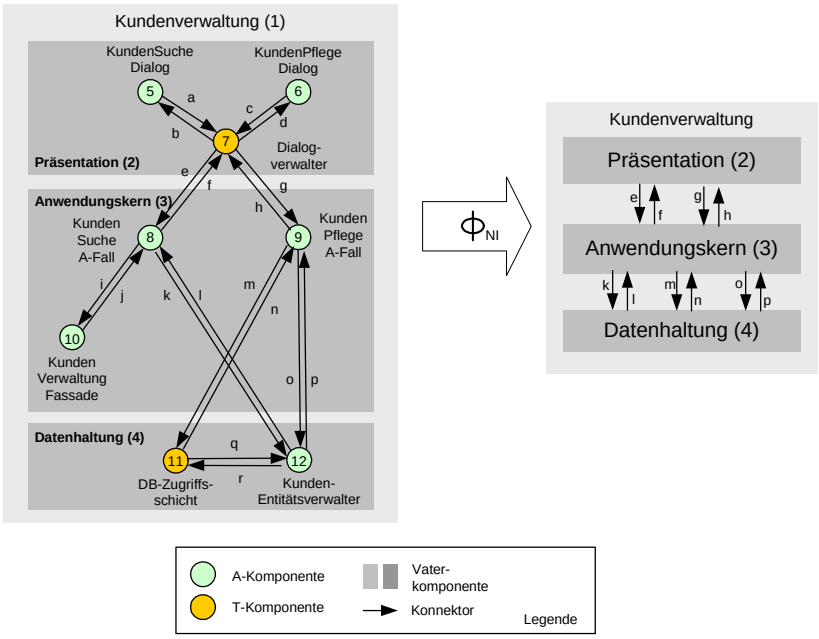


Abbildung 6.3: Komponenten unterhalb der ersten Hierarchieebene werden ausgeblendet.

$n \in \mathbb{N}_0$ werden die internen Konnektoren der beseitigten Teilbäume entfernt. Alle anderen Konnektoren bleiben erhalten. Die zu der Konfiguration $C$ passenden Nutzungsszenarios, die $S$ liefert, werden analog modifiziert. Die aus $E$ entfernten Konnektoren werden aus allen Nutzungsszenarios $S(uc)$ mit $uc \in UC$ beseitigt. Die Anwendungsfälle bleiben erhalten:

$UC' := UC$

und die Abbildung $S$ wird entsprechend angepasst, $\forall uc \in UC$:

$S'(uc) := \{S(uc) \cap E'\}$

Die Anwendungsfälle und Szenarios der Beispielkonfiguration lauten:

$S_{KV} = \{(uc_1, \{a, b, e, f, k, l, q, r\}), (uc_2, \{c, d, g, h, o, p, q, r\}), (uc_3, \{i, j, k, l, q, r\})\}$

werden um die entfallenen Konnektoren (vgl. Menge $levelinternal(KVH, 1) = \{a, b, c, d, i, j, q, r\}$) vermindert.

$S_{KV'} = \{(uc_1, \{e, f, k, l\}), (uc_2, \{g, h, o, p\}), (uc_3, \{k, l\})\}$.

Die Attributwerte aller Komponenten und Konnektoren, welche nicht von der Vergröberung betroffen sind, werden übernommen. Bei Komponenten, deren Teilkomponenten entfernt wurden, werden die Attributwerte der Teilkomponenten und der internen Konnektoren summiert. Für die Vergröberung der Attribute gilt $\forall a \in A', \forall v \in V', \forall e \in E'$:

$A' := A$

und

$value'(e, a) := value(e, a)$

$$value'(v, a) := \begin{cases} value(v, a) +_a \sum^a_{w \in allparts(C,v)} (value(w, a)) \\ +_a \sum^a_{f \in internal(C, allparts(C,v))} (value(f, a)) & : v \in leafs(C') \\ \\ value(v, a) & : sonst \end{cases}$$

Die angegebene Definition verwendet vereinfachend $v \in leafs(C')$ als Kriterium, um die aggregierten Komponenten zu finden. Unter den Blättern der Hierarchie $H'$ befinden sich möglicherweise auch nicht aggregierte Komponenten, also $parts(C, v) = \emptyset$, dies ändert jedoch nichts am Ergebnis von $value$:

$\forall v \in leafs(C') \wedge parts(C, v) = \emptyset; \forall a \in A :$

$$
\begin{aligned}
& value'(v, a) = \\
= \;& value(v, a) +_a \textstyle\sum^a_{w \in allparts(C,v)}(value(w, a)) +_a \sum^a_{f \in internal(C,allparts(C,v))}(value(f, a)) \\
= \;& value(v, a) +_a \textstyle\sum^a_{w \in \emptyset}(value(w, a)) +_a \sum^a_{f \in internal(C,\emptyset)}(value(f, a)) \\
= \;& value(v, a)
\end{aligned}
$$

Ausführliche Beispiele für die Berechnung der Belegungen zu vergröberten Komponenten werden in Abschnitt 7.3.1 gegeben. Dort werden Planungsdaten summiert.

Für die Beispielkonfiguration $KV$ ist eine Attributmenge $A_{KV} =$ {*Software-Kategorie, Aufgabe*} angegeben. Die beiden Attribute werden bei der Vergröberung übernommen: $A_{KV'} := A_{KV}$. Die Belegung muss nach den angegebenen Formeln für $value_{KV'}$ und den Rechenregeln der beiden Attribute neu berechnet werden.

| Id | Name | Software-Kategorie | Aufgabe |
|---|---|---|---|
| 1 | Kundenverwaltung | 0-Software | Subsystem |
| 2 | Dialogschicht | AT-Software | Präsentation |
| 3 | Anwendungskern | A-Software | Anwendungskern |
| 4 | Datenhaltung | AT-Software | Datenhaltung |

Tabelle 6.2: Werte von $value_{KV'}$ für die Komponenten der vergröberten Kundenverwaltung

Die Summation hat die Belegung des Attributs *Software-Kategorie* für die drei Komponenten *Dialogschicht*, *Anwendungskern* und *Datenhaltung* geändert. Die Berechnung der Summe wird beispielhaft an der Komponente *Datenhaltung (4)* vorgeführt, hierzu ist die Definition der Summe von Software-Kategorien wichtig, diese wird in Abschnitt 5.5 definiert. Die Summenoperationen des Attributtyps *Software-Kategorie* werden abkürzend mit $\sum$ statt $\sum^{SoftwareKategorie}$ sowie $+$ statt $+_{SoftwareKategorie}$ bezeichnet:

$value_{KV'}(4,$ *Software-Kategorie*$) =$

$$
\begin{aligned}
= \;& value_{KV}(4,\textit{Software-Kategorie}) + \textstyle\sum_{w \in allparts(KV,4)}(value_{KV}(w,\textit{Software-Kategorie})) \\
& + \textstyle\sum_{f \in internal(KV,allparts(KV,4))}(value_{KV}(f,\textit{Software-Kategorie})) \\
= \;& 0Software + \textstyle\sum_{w \in \{11,12\}}(value_{KV}(w,\textit{Software-Kategorie})) \\
& + \textstyle\sum_{f \in internal(KV,\{11,12\})}(value_{KV}(f,\textit{Software-Kategorie})) \\
= \;& \textit{0-Software + T-Software + A-Software} \\
& + \textstyle\sum_{f \in \{q,r\})}(value_{KV}(f,\textit{Software-Kategorie})) \\
= \;& \textit{0-Software + T-Software + A-Software} \\
& + \textit{0-Software} \\
= \;& \textit{AT-Software}
\end{aligned}
$$

### 6.2.2 Vergröberung von Mehrfachkonnektoren

#### Vergröberung eines Mehrfachkonnektors

Konnektoren, welche die gleichen zwei Komponenten $v, w \in V$ verbinden, können zu einem neuen Konnektor vergröbert werden. Dies kann beispielsweise die Übersichtlichkeit einer Architekturdarstellung erhöhen. Dafür wird die Vergröberungsfunktion $\phi_E$ innerhalb einer Konfiguration $C = (V, E, J, H) \in CONFIGURATION$ definiert:

$\phi_E : \wp(E) \rightarrow (CONNECTOR \setminus E) \cup \{\langle\rangle\}$.

Für die Konnektormenge $E_{vw} \in \wp(E)$, die $\phi_E$ übergeben wird, gilt folgende Einschränkung: Die Konnektoren $E_{vw}$ müssen dieselben zwei Konnektoren verbinden (s.u.). Wird $\phi_E$ mit einer Menge von Konnektoren verwendet, welche diese Einschränkung nicht erfüllen, ist das Ergebnis das Symbol $\langle\rangle$ (undefiniert). Da $\phi_E$ nur innerhalb der hier dargestellten Projektionen verwendet wird, tritt dieser Fall jedoch niemals auf.

Sei $C = (V, E, J, H)$ eine Konfiguration und $v, w \in V$ zwei Komponenten, die über Mehrfachkonnektoren verbunden sind, also $\exists e, f \in E, e \neq f : J(e) = J(f) = (v, w)$. Dann sei die Menge

$E_{vw} = \{e \in E | J(e) = (v, w)\}, |E_{vw}| \geq 2$

die Menge der Mehrfachkonnektoren zwischen $v$ und $w$. Zusätzlich ist $e_{vw} = \phi_E(E_{vw})$ die Vergröberung dieser Menge $E_{vw}$ genau dann, wenn es eine Konfiguration $C' = (V', E', J', H')$ gibt mit:

- $V' := V$
- $E' := (E \setminus E_{vw}) \cup \{e_{vw}\}, e_{vw} = \phi_E(E_{vw}), e_{vw} \notin E$,. Die zusammengefassten Konnektoren werden entfernt und ein neuer Konnektor $e_{vw} = \phi_E(E_{vw})$, welcher die Zusammenfassung darstellt, wird ergänzt.
- $J'(e) := \begin{cases} J(e) : & e \in E \setminus E_{vw} \\ (v, w) : & e = \phi_E(E_{vw}), \forall f \in E_{vw} : J(f) = (v, w) \end{cases}$
- $H' := H$.

Die $id(e_{vw}) \in \mathbb{ID}$ des neuen Konnektors $e_{vw}$ wird eindeutig generiert[5] und der $name(e_{vw}) \in \mathbb{ID}$ wird aus den Namen der zusammengefassten Konnektoren gemäß der durch die Ordnung in $\mathbb{ID}$ gegebenen Reihenfolge konkateniert:

$name(e_{vw}) := \sum_{i=1 \cdots n, f_i \in E_{vw}, name(f_i) \leq name(f_j), i \leq j \leq n} (name(f_i))$.

$\phi_E(E_{vw}) = e_{vw}$ liefert damit (nach Definition von $\phi_E$) immer denselben Konnektor $e_{vw}$, unabhängig vom Zeitpunkt der Anwendung.

Während der Vergröberung der Konfiguration $KV$ zu $KV'$ sind mehrere Mehrfachkonnektoren entstanden. So verbindet etwa die Menge $E_{3,4} = \{k, m, o\}$ die beiden Komponenten 3 und 4. Für diese drei Konnektoren führt die Funktion $\phi_E(E_{3,4}) = \phi_E(\{k, m, o\}) = kmo$ einen neuen Konnektor $kmo$ ein, welcher ebenfalls 3 und 4 verbindet. Die Zuordnungsfunktion $J_{KV''}$ wird entsprechend angepasst mit $J_{KV''}(kmo) = (3, 4)$.

#### Vergröberung aller Mehrfachkonnektoren einer Konfiguration

Über die gerade vorgestellte Funktion $\phi_E$ werden die Konnektoren zwischen zwei Komponenten $v, w \in V$ zusammengefasst. Diese Abbildung wird nun ausgedehnt und über $\Phi_E$ werden alle Mehr-

[5] Für die im Folgenden dargestellten Beispiele werden die Ids der zusammengefassten Konnektoren konkateniert und ergeben so die Id des neuen Konnektors.

fachkonnektoren aus einer Konfiguration entfernt, so dass das Ergebnis der Abbildung $C'$ ein einfacher gerichteter Graph ist. Sei

$\Phi_E : CONFIGURATION \rightarrow CONFIGURATION.$

Sei $C = (V, E, J, H) \in CONFIGURATION$ eine wohlgeformte Konfiguration. Dann berechnet sich die Menge $M(C)$ ($M : CONFIGURATION \rightarrow \wp(\wp(CONNECTOR))$ ), deren Elemente Mengen $E_{vw}$ von Mehrfachkonnektoren sind (also $|E_{vw}| \geq 2$), wie folgt

$M(C) := \{E_{vw} | E_{vw} \subseteq E, |E_{vw}| \geq 2 : \exists v, w \in V \text{ mit } \forall e \in E_{vw} : J(e) = (v, w)\}$

Für $\Phi_E(C) := (V', E', J', H') := C'$ gilt dann

- $V' := V$, die Komponenten bleiben unverändert,
- $E' := (E \backslash \bigcup_{E_{vw} \in M(C)}(E_{vw})) \cup \{\phi_E(E_{vw}) | E_{vw} \in M(C), \phi_E(E_{vw}) \notin E\}$. Die zusammengefassten Konnektoren $E_{vw} \in M(C)$ werden entfernt und dafür wird jeweils ein neuer Konnektor $e_{vw} = \phi_E(E_{vw})$ ergänzt.
- $J'(e) := \begin{cases} J(e) : & e \in E \setminus \bigcup_{E_{vw} \in M(C)}(E_{vw}) \\ (v, w) : & \exists E_{vw} \in M(C) : e = \phi_E(E_{vw}) \wedge \forall f \in E_{vw} : J(f) = (v, w) \end{cases}$
- $H' := H$.

Die Ergebniskonfiguration $C'$ ist wohlgeformt, wenn $C$ wohlgeformt war, da die Hierarchie $H$ unverändert übernommen wird.

Mithilfe von $\Phi_E$ kann etwa die Konfiguration $KV'$, die aus der Vergröberung von $KV$ entstanden ist, vereinfacht werden. Die Mehrfachkonnektoren werden entfernt. Die Menge $M(KV')$ in $KV'$ ist gegeben über die Mengen der Mehrfachkonnektoren in $KV'$:

$M(KV') = \{\{e, g\}, \{f, h\}, \{k, m, o\}, \{l, n, p\}\}$. In diesem Beispiel sind alle vorhandenen Konnektoren in einer Menge von $M(KV')$ enthalten.

Für $\Phi_E(KV') = KV''$ mit $KV'' = (V_{KV''}, E_{KV''}, J_{KV''}, H_{KV''})$ gilt:

- $V_{KV''} = V_{KV'} = \{1, 2, 3, 4\}$
- $E_{KV''} = \{eg, fh, kmo, lnp\}$.
  Die Konnektoren aus $KV'$ werden durch neue Konnektoren ersetzt. Für die neuen Konnektoren gilt:
  $\phi_E(\{e, g\}) = eg$, $\phi_E(\{f, h\}) = fh$, $\phi_E(\{k, m, o\}) = kmo$
  sowie $\phi_E(\{l, n, p\}) = lnp$.
- $J_{KV''} = \{(eg, (2, 3)), (fh, (3, 2)), (kmo, (3, 4)), (lnp, (4, 3))\}$
- $H_{KV''} = H'_{KV} = \{(1, 2), (1, 3), (1, 4)\}$

**Vergröberung aller Mehrfachkonnektoren einer logischen Architektur**

Sei $\mathcal{L} = (C, A, value, UC, S)$ eine logische Architektur und $\Phi_E : LA \rightarrow LA$ die Vergröberungsprojektion. $\Phi_E(\mathcal{L}) = \mathcal{L}' = (C', A', value', UC', S')$ kann dann wie folgt bestimmt werden:

Sei $C = (V, E, J, H) \in CONFIGURATION$ die Konfiguration von $\mathcal{L}$, dann ist $C' := \Phi_E(C) = (V', E', J', H')$ die Konfiguration von $\mathcal{L}'$.

Die Menge $M(C)$ der Mehrfachkonnektor-Mengen von $C$ ist gegeben durch:

$M(C) := \{E_{vw} | E_{vw} \subseteq E, |E_{vw}| \geq 2 : \exists v, w \in V \text{ mit } \forall e \in E_{vw} : J(e) = (v, w)\}$

Die Anwendungsfälle werden direkt übernommen, also:

$UC' := UC.$

Die Nutzungsszenarios $S(uc), uc \in UC$ von $\mathcal{L}$ werden so modifiziert, dass darin enthaltene Mehrfachkonnektoren $f \in S(uc) \wedge f \in E_{vw} \wedge E_{vw} \in M(C)$ durch ihre Vergröberungen $e = \phi_E(E_{vw})$ ersetzt werden:

$$\begin{aligned} S'(uc) := \quad & (S(uc) \cap E') \cup \\ & \{e \in E' | \exists f \in E, \exists E_{vw} \in M(C) : f \in S(uc) \wedge f \in E_{vw} \wedge e = \phi_E(E_{vw})\} \end{aligned}$$

Bei Konnektoren, die mehrere andere Konnektoren aggregieren, werden die Attributwerte der Teilkonnektoren summiert. Durch die Vergröberung der Mehrfachkonnektoren bleiben die Attributmengen unverändert:

$A' := A$

Haben die Konnektoren aus $E_{vw} \in M(C)$ Attributwerte, werden diese mit den entsprechenden Summenoperatoren aggregiert und dem neuen Konnektor $e_{vw}$ zugewiesen. Für die Attributwerte gilt $\forall v \in V', \forall a \in A'$:

$value'(v, a) := value(v, a)$

Für die Konnektoren und ihre Attribute gilt:$\forall a \in A', \forall e \in E'$:

$$value'(e, a) := \begin{cases} value(e, a) : & e \in E \setminus \bigcup_{E_{vw} \in M(C)} (E_{vw}) \\ \sum_{f \in E_{vw}}^{a} (value(f, a)) : & \exists E_{vw} \in M(C), e = \phi_E(E_{vw}) \end{cases}$$

Als Beispiel dient wieder die Konfiguration $KV'$, die Ergebnis der Vergröberung der Beispielkonfiguration $KV$ auf die Ebene 1 war. Die Konfiguration $KV'$ wurde im vorangegangenen Abschnitt bereits zu $KV''$ vergröbert, indem die Mehrfachkonnektoren entfernt wurden. Dies ist in Abbildung 6.4 dargestellt.

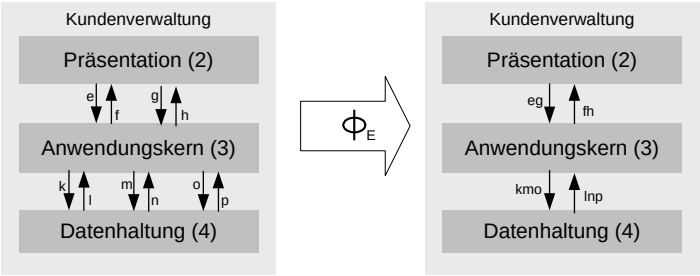


Abbildung 6.4: Vergröberung von Mehrfachkonnektoren

Um die Mehrfachkonnektoren aus der logischen Architektur zu entfernen sind nun noch die Szenarios $S_{KV'}$, sowie die Attributmenge und die Belegung zu vergröbern:

$S_{KV'} = \{(uc_1, \{e, f, k, l\}), (uc_2, \{g, h, o, p\}), (uc_3, \{k, l\})\}$

Die Szenarios werden damit ähnlicher, da Konnektoren zusammenfallen. Im vorliegenden Beispiel enthält $S_{KV''}$ mehrere identische Szenarios. Die Anwendungsfälle $uc_1$ bis $uc_3$ werden auf der gewählten (Abstraktions-)Ebene von denselben Komponenten und Konnektoren implementiert:

$S_{KV''}(uc_1) = \{eg, fh, kmo, lnp\}$, $S_{KV''}(uc_2) = \{eg, fh, kmo, lnp\}$ und $S_{KV''}(uc_3) = \{kmo, lnp\}$

$S_{KV''} = \{(uc_1, \{eg, fh, kmo, lnp\}), (uc_2, \{eg, fh, kmo, lnp\}), (uc_3, \{kmo, lnp\})\}$

Die Attributmenge bleibt unverändert: $A_{KV''} := A_{KV'} =$ {*Software-Kategorie, Aufgabe*}. Da den Konnektoren nur neutrale Belegungen zugeordnet waren, werden den neuen Konnektoren ebenfalls die

neutralen Belegungen zugeordnet. Das ist in Tabelle 6.3 zu sehen. Die Belegungen der Komponenten (vgl. Tabelle 6.2) bleiben unverändert.

| Id | Name | Software-Kategorie | Aufgabe |
|---|---|---|---|
| eg | ″ | 0-Software | neutral |
| fh | ″ | 0-Software | neutral |
| kmo | ″ | 0-Software | neutral |
| lnp | ″ | 0-Software | neutral |

Tabelle 6.3: Belegungen von Software-Kategorie und Aufgabe bei den zusammengefassten Konnektoren

## 6.3 Zusammenfassungsprojektion

Eine Zusammenfassungsprojektion (kurz Zusammenfassung) $\Psi$ fasst Komponenten zusammen, die Geschwister und Blätter in der hierarchischen Struktur sein müssen. Das Kriterium für die Zusammenfassung kann inhaltlich frei gewählt werden. Es muss eine Äquivalenzrelation $\mathcal{R}$ auf der Menge der Komponenten $V$ sein. In der Abbildung 6.5 (rechts) wird die Aufgabe der Komponenten als Zusammenfassungskriterium verwendet: Alle Komponenten mit derselben Aufgabe werden jeweils zu einer Komponente zusammengefasst.

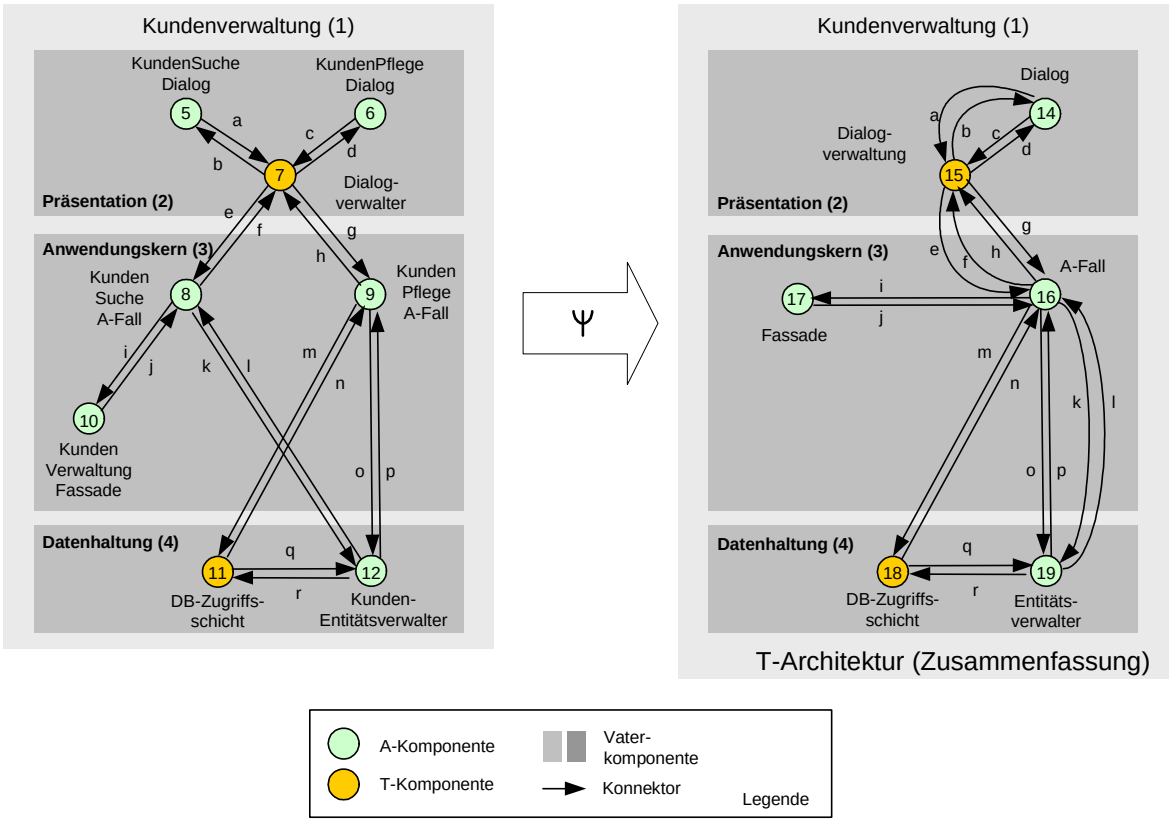


Abbildung 6.5: Zusammenfassung von Komponenten mit derselben Aufgabe

Bei der Zusammenfassung $\Psi$ werden Mengen (Äquivalenzklassen) von Teilkomponenten einer Komponente zusammengefasst, diese Teilkomponenten müssen Blätter der Komponentenhierarchie sein.

Als Kriterium für die Zusammenfassung wird eine Äquivalenzrelation (reflexiv, symmetrisch und transitiv)

$\mathcal{R} \subseteq COMPONENT \times COMPONENT$

verwendet. Zu jeder Komponente $v \in leafs(C)$ aus einer Konfiguration $C = (V, E, J, H)$ wird mithilfe von $\mathcal{R}$ eine Äquivalenzklasse gebildet. Die Äquivalenzklasse ist gegeben durch:

$$[v]_{\mathcal{R}} := \begin{cases} \{w | v\mathcal{R}w \wedge parent(C, v) = parent(C, w) \wedge w \in leafs(C))\} & : v \in leafs(C) \\ \{v\} & : sonst \end{cases}$$

Die Elemente der Äquivalenzklasse $[v]_{\mathcal{R}}$ stehen also in Relation $\mathcal{R}$, sie sind darüber hinaus Blätter der Komponentenhierarchie und haben dieselbe Vaterkomponente. Ist die Komponente $v$ keine Blattkomponente, enthält die Äquivalenzklasse nur die Komponente selbst. Die Äquivalenzklassen werden nur auf Geschwister-Komponenten und nur auf Blättern der Komponentenhierarchie gebildet, damit die Ergebniskonfiguration wohlgeformt bleibt.

Für die Beispielkonfiguration $KV$ könnte eine Äquivalenzrelation $\mathcal{R}_{Aufgabe}$ definiert werden. Diese fasst alle Komponenten mit derselben Aufgabe zusammen. Die Komponente 5 (KundenSuche Dialog) hat die Aufgabe *Dialog*. Die Komponente 6 (KundenPflege Dialog) hat dieselbe Aufgabe und gehört damit in dieselbe Äquivalenzklasse wie Komponente 5 in Bezug auf $\mathcal{R}_{Aufgabe}$.

$\mathcal{R}_{Aufgabe} := \{(v, w) | v, w \in V \wedge value_{KV}(v, Aufgabe) = value_{KV}(w, Aufgabe)\}$

Die Äquivalenzklassen auf der Menge der Teilkomponenten einer Komponente werden über die Zusammenfassungsprojektion $\Psi$ jeweils zu einer neuen Komponente zusammengefasst. Die Konnektoren werden umgehängt, sofern sie nicht innerhalb einer Äquivalenzklasse verlaufen. $\Psi$ ist definiert über:

$\Psi : CONFIGURATION \times \wp(COMPONENT \times COMPONENT) \rightarrow CONFIGURATION \cup \{\langle\rangle\}.$

Das Ergebnis von $\Psi$ lautet $\langle\rangle$, wenn die Relation $\in \wp(COMPONENT \times COMPONENT)$ keine Äquivalenzrelation auf der Menge der Komponenten der übergebenen Konfiguration ist.

#### Zusammenfassung von Teilkomponenten

Zunächst wird eine Funktion $\phi_V$ definiert, die eine Menge von Teilkomponenten $V_S \subseteq V$ durch eine neue Komponente $v_s$ ersetzt. Diese Zusammenfassung von Komponenten wird nur unterhalb derselben Vaterkomponente erlaubt und die zusammengefassten Komponenten dürfen nicht selbst unterteilt sein. Eine Zusammenfassungsfunktion $\phi_V$ ersetzt eine Menge von Teilkomponenten durch eine neue Komponente:

$\phi_V : \wp(V) \rightarrow (COMPONENT \setminus V) \cup \{\langle\rangle\}.$

Für die Komponentenmenge $V_S \in \wp(V)$, die $\phi_V$ übergeben wird, gilt folgende Einschränkung: Eie Komponenten $V_S$ müssen eine gemeinsame Vaterkomponente haben: $\exists x \in V : V_S \subseteq parts(C, x)$. Ist diese Einschränkung nicht erfüllt, ist das Ergebnis von $\phi_V$ das Symbol $\langle\rangle$ (undefiniert). Da $\phi_V$ nur innerhalb der hier dargestellten Projektionen verwendet wird, kommt dieser Fall jedoch nicht vor.

Sei $C = (V, E, J, H) \in CONFIGURATION$ eine wohlgeformte Konfiguration und $V_S \subset V$ eine Teilmenge ihrer Komponenten mit derselben Vaterkomponente: $\exists x \in V : V_S \subseteq parts(C, x)$. Alle Komponenten aus $V_S$ sind nicht weiter unterteilt: $V_S \subseteq leafs(C)$. Dann ist $v_S = \phi_V(V_S), v_S \notin V$ die Aggregation der Menge $V_S$, genau dann, wenn es eine Konfiguration $C' = (V', E', J', H')$ gibt mit:

- $V' := (V \setminus V_S) \cup \{v_S\}$, die Komponenten aus $V_S$ werden aus der Konfiguration entfernt und eine neue Komponente $v_S = \phi_V(V_S), v_S \notin V$ wird ergänzt.
- $E' := E \setminus internal(C, V_S)$. Die Konnektoren, welche zwischen den zusammengefassten Komponenten verlaufen, werden entfernt. Alle anderen werden umgehängt.

- $J'(e) := \begin{cases} J(e) & : e \in E \setminus (incoming(C, V_S) \cup outgoing(C, V_S) \cup internal(C, V_S)) \\ (v, v_S) & : e \in incoming(C, V_S); J(e) = (v, u), u \in V_S, v \in V \setminus V_S \\ (v_S, w) & : e \in outgoing(C, V_S); J(e) = (u, w), u \in V_S, w \in V \setminus V_S \end{cases}$
- $H' := (H \cap (V' \times V')) \cup (x, v_S)$ mit $(x, v) \in H, \forall v \in V_S$, aus der Hierarchie werden die zusammengefassten Komponenten entfernt und die neue Komponente $v_S$ wird eingehängt.

Die Bildkonfiguration $C'$ ist wohlgeformt, denn die hierarchische Struktur $H'$ geht aus der wohlgeformten Hierarchie $H$ durch Einschränkung hervor und durch Ergänzung einer neuen Komponente mit genau einer Vaterkomponente.

Die $id(v_S) \in \mathbb{ID}$ der hinzugefügten Komponente $v_S$ wird eindeutig generiert und der $name(v_S) \in \mathbb{ID}$ wird aus den Namen der zusammengefassten Komponenten gemäß der durch die Ordnung in $\mathbb{ID}$ gegebenen Reihenfolge konkateniert:

$name(v_S) := \sum_{i=1 \cdots n, v_i \in V_S, name(v_i) \leq name(v_j), i \leq j \leq n} (name(v_i))$. $\phi_V(C, V_S) = v_s$

liefert damit immer dieselbe Komponente $v_S$, unabhängig vom Zeitpunkt der Anwendung. Für bestimmte Projektionen wird die Funktion $name$ gegen eine fachlich oder technisch aussagekräftigere Variante ausgetauscht.

#### Zusammenfassung in einer Konfiguration mithilfe einer Äquivalenzrelation

Bei der Zusammenfassungsprojektion einer Konfiguration werden Blätter innerhalb der hierarchischen Struktur $H$ über die Äquivalenzrelation $\mathcal{R}$ zusammengefasst. Die Menge der Komponenten, deren Kindkomponenten zusammengefasst werden, da diese Blätter sind, ist gegeben durch:

$forks(C) := \{v \in V | parts(C, v) \cap leafs(C) \neq \emptyset\}$

Die Blätter werden entfernt und durch Komponenten ersetzt, welche die jeweiligen Äquivalenzklassen in Bezug zur Relation $\mathcal{R}$ repräsentieren:

- $V' := (V \setminus leafs(C)) \cup \bigcup_{v \in leafs(C)} (\phi_V([v]_\mathcal{R}))$, die Komponenten zu jeder Äquivalenzklasse $[v]_\mathcal{R}$ werden jeweils zu einer neuen Komponente $\phi_V([v]_\mathcal{R})$ zusammengefasst.
- $E' := E \setminus \bigcup_{v \in leafs(C)} (internal(C, [v]_\mathcal{R}))$. Die Konnektoren innerhalb einer Äquivalenzklasse werden entfernt, alle anderen werden bei Bedarf umgehängt.
- $J'(e) := (\phi_V([v]_\mathcal{R}), \phi_V([w]_\mathcal{R}))$
- $H' := \begin{array}{l} (H \setminus (forks(C) \times leafs(C))) \cup \\ \{(x, y) | x \in forks(C), y = \phi_V(C, [v]_\mathcal{R}), v \in (leafs(C) \cap parts(C, x))\} \end{array}$

  die Teile der gefalteten Komponenten werden aus der Hierarchie entfernt und die Zusammenfassungen werden eingehängt.

Die Bildkonfiguration $C'$ ist wohlgeformt, denn die hierarchische Struktur $H'$ geht aus der wohlgeformten Hierarchie $H$ dadurch hervor, dass von allen Komponenten $x \in forks(C)$ die Teile $leafs(C) \cap parts(C, x)$ durch neuen Komponenten $\phi_V([v]_\mathcal{R}), v \in leafs(C) \cap parts(C, x)$ ersetzt werden.

Nun soll $\Psi$ auf die Beispielkonfiguration $KV$ angewendet werden. Als Äquivalenzrelation wird die Relation

$\mathcal{R}_{Aufgabe} := \{(v, w) | v, w \in V_{KV} \wedge value_{KV}(v, Aufgabe) = value_{KV}(w, Aufgabe)\}$

verwendet. Die Namen der neuen Komponenten wird über den Wert des Attributs $Aufgabe$ berechnet, da alle Komponenten innerhalb der Äquivalenzklasse dieselbe Aufgabe haben.

$name(\phi_V([v]_{\mathcal{R}_{Aufgabe}})) := value(v, Aufgabe)$

Alle Komponenten mit derselben Aufgabe und derselben Vaterkomponente sind äquivalent und werden über $\Psi$ zusammengefasst. Dies ist eine mögliche Formulierung des Blickwinkels für T-Architekturen innerhalb der Quasar-Referenzarchitektur.

Die Relation $\mathcal{R}_{Aufgabe}$ ist auf der Konfiguration $KV$ gegeben durch die Tupelmenge:

$$\begin{array}{lll}\mathcal{R}_{Aufgabe} = \{ & (1,1),(2,2),(3,3),(4,4), & \text{zusammengesetzte Komponenten} \\ & (7,7),(10,10),(11,11),(12,12), & \text{Dialogverw., Fassade, Zugriffssch., Entitätsverw.} \\ & (5,5),(6,6),(5,6),(6,5), & \text{Dialog} \\ & (8,8),(9,9),(8,9),(9,8) & \text{A-Fall } \}\end{array}$$

Dabei sind die zusammengesetzten Komponenten nur mit sich selbst äquivalent, da die Aufgaben *Subsystem, Präsentation, Anwendungskern* und *Zugriffsschicht* in der Architektur nur jeweils einmal vergeben wurden. Das gleiche gilt für die Aufgaben *Dialogverwaltung*, *Fassade*, *Zugriffsschicht* und *Entitätsverwalter*. Nur die Aufgaben *Dialog* und *A-Fall* wurden mehr als einer Komponente zugeordnet. Die angegebene Relation ist reflexiv, symmetrisch und transitiv. Damit ist sie Äquivalenzrelation [Ste01].

Die Äquivalenzklassen zur Äquivalenzrelation $\mathcal{R}_{Aufgabe}$ werden hier als Menge von Mengen angegeben, jede Menge ist eine Äquivalenzklasse. Hier wird von den Elementen einer Äquivalenzklasse zusätzlich gefordert, dass sie Blätter und gleichzeitig Geschwister der Hierarchie $H$ sind (siehe oben). Die zusammengesetzten Komponenten 1 bis 4 kommen daher nicht in der Menge der Äquivalenzklassen vor.

$$\{[v]_{\mathcal{R}_{Aufgabe}} | v \in V_{KV}\} = \{\{5,6\},\{7\},\{8,9\},\{10\},\{11\},\{12\}\}$$

Die Komponenten jeder Äquivalenzklasse werden über die Zusammenfassung $\Psi(KV, \mathcal{R}_{Aufgabe})$ durch eine neue Komponente ersetzt, dies gilt auch für die Äquivalenzklassen mit nur einem Element. In $KV$ werden damit die 6 neuen Komponenten 14 bis 19 hinzugefügt und alle Blätter 5 bis 12, die in den Äquivalenzklassen vorkommen, werden entfernt.

Es gilt $\phi_V(\{5,6\}) = 14$, $\phi_V(\{7\}) = 15$, $\phi_V(\{8,9\}) = 16$, $\phi_V(\{10\}) = 17$, $\phi_V(\{11\}) = 18$ und $\phi_V(\{12\}) = 19$.

Sei $\Psi(KV, \mathcal{R}_{Aufgabe}) = KV' = (V_{KV'}, E_{KV'}, J_{KV'}, H_{KV'})$ mit:

- $V_{KV'} = \{1,2,3,4,14,15,16,17,18,19\}$, d.h. alle Blattkomponenten wurden durch die neuen Komponenten $\{14,15,16,17,18,19\}$ ersetzt.
- $E_{KV'} = E_{KV} = \{a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r\}$ da keine Konnektoren innerhalb einer Äquivalenzklasse verlaufen, wird die Konnektormenge übernommen.
- $J_{KV'} = \{(a,(14,15)),(b,(15,14)),(c,(14,15)),(d,(15,14)),(e,(15,16)),(f,(16,15)),$
  $(g,(15,16)),(h,(16,15)),(i,(16,17)),(j,(17,16)),(k,(16,19)),(l,(19,16)),$
  $(m,(16,18)),(n,(18,16)),(o,(16,19)),(p,(19,16)),(q,(18,19)),(r,(19,18))\}$

  Die Konnektoren werden auf die zusammengefassten Komponenten $\{14,15,16,17,18,19\}$ umgehängt. Mehrere Mehrfachkonnektoren entstehen dadurch.
- $H_{KV'} = \{(1,2),(1,3),(1,4),(2,14),(2,15),(3,16),(3,17),(4,18),(4,19)\}$

#### Zusammenfassung einer logischen Architektur mithilfe einer Äquivalenzrelation

Sei $\mathcal{L} = (C, A, value, UC, S)$ eine logische Architektur, $\mathcal{R} \subseteq COMPONENT \times COMPONENT$ eine Äquivalenzrelation auf der Komponentenmenge von $\mathcal{L}$ und

$$\Psi : LA \times \wp(COMPONENT \times COMPONENT) \rightarrow LA \cup \{\langle\rangle\}$$

eine Zusammenfassungsprojektion mit $\Psi(\mathcal{L}, \mathcal{R}) = \mathcal{L}' = (C', A', value', UC', S')$. Die Projektion wird auf die Zusammenfassungsprojektion der Konfiguration $C$ von $\mathcal{L}$ und dazu passender Szenarios $S$ zurückgeführt. Wenn $C = (V, E, J, H)$ die Konfiguration von $\mathcal{L}$ ist, dann ist

$C' := \Psi(C, \mathcal{R}) = (V', E', J', H')$

die Konfiguration von $\mathcal{L}'$. $UC$ sei die Menge der Anwendungsfälle zu $\mathcal{L}$ und die Funktion $S$ ordnet jedem Anwendungsfall ein Nutzungsszenario zu. Während der Zusammenfassungsprojektion werden alle Konnektoren, die aus $E$ entfernt wurden, auch aus allen Szenarios entfernt, die Menge der Anwendungsfälle wird nicht geändert:

$UC' := UC$ und

$S'(uc) := S(uc) \cap E'$.

Für die Szenarios $S_{KV}$ zur Beispielkonfiguration $KV$ ergeben sich keine Änderungen, da kein Konnektor durch die Zusammenfassung entfallen ist. Damit gilt $S_{KV'} = S_{KV}$

In der gefalteten Konfiguration $C' = \Psi(C, \mathcal{R})$ der logischen Architektur $\mathcal{L}'$ werden alle Attribute übernommen

$A' := A$.

Die Belegungen der für die Äquivalenzklassen hinzugekommenen Komponenten $v \in V' \setminus V$ werden aus den Belegungen der Komponenten $\{u \in V | \phi_V([u]_\mathcal{R}) = v\}$ und Konnektoren
$e \in internal(C, \{u \in V | \phi_V([u]_\mathcal{R}) = v\})$ aus der jeweiligen Äquivalenzklasse berechnet. Für die Werte der Attribute gilt $\forall a \in A', \forall v \in V'$:

$$value'(v,a) := \begin{cases} \sum^a_{w \in V | \phi_V([w]_\mathcal{R}) = v}(value(w,a)) \\ +_a \sum^a_{e \in internal(C, \{u \in V | \phi_V([u]_\mathcal{R}) = v\})}(value(e,a)) & : v \in V' \setminus V \\ \\ value(v,a) & : sonst \end{cases}$$

Die Attribute, der verbliebenen Konnektoren bleiben unverändert, also $\forall e \in E', \forall a \in A'$:

$value'(e,a) := value(e,a)$.

Für die Beispielkonfiguration $KV$ werden alle Attribute nach $KV'$ übernommen, damit gilt

$A_{KV'} = A_{KV} =$ {*Software-Kategorie, Aufgabe*}.

Nach den Summationsregeln für den Attributtyp $Aufgabe$ bleibt jeweils bei den zusammengefassten Komponenten die Aufgabe der Ursprungskomponenten enthalten (etwa: $Dialog + Dialog = Dialog$). Die Belegungen der Komponenten wird in Tabelle 6.4 dargestellt.

| Id | Name | Software-Kategorie | Aufgabe |
|---|---|---|---|
| 1 | Kundenverwaltung | 0-Software | Subsystem |
| 2 | Dialogschicht | 0-Software | Präsentation |
| 3 | Anwendungskern | 0-Software | Anwendungskern |
| 4 | Datenhaltung | 0-Software | Datenhaltung |
| 14 | Dialog | A-Software | Dialog |
| 15 | Dialogverwalter | T-Software | Dialogverwalter |
| 16 | A-Fall | A-Software | A-Fall |
| 17 | Fassade | A-Software | Fassade |
| 18 | DB Zugriffsschicht | T-Software | Zugriffsschicht |
| 19 | Entitätsverwalter | A-Software | Entitätsverwalter |

Tabelle 6.4: Belegung von Software-Kategorie und Aufgabe bei Komponenten der nach Aufgaben zusammengefassten Kundenverwaltung

Die Berechnung der Belegungen für die Komponente *Dialog* (14) wird nun als Beispiel für die in Tabelle 6.4 gezeigten Belegungen für *Software-Kategorie* und *Aufgabe* vorgerechnet, bei den Summenoperationen $\sum$ und $+$ wird der Attributtyp aus Gründen der Übersicht nicht dargestellt.

Da $14 \in V_{KV'} \setminus V_{KV}$ gilt:

$value'(14, \textit{Software-Kategorie}) =$

$= \sum_{w \in \{u \in V | \phi_V([u]_{\mathcal{R}_{Aufgabe}}) = v\}} (value(w, \textit{Software-Kategorie}))$
$+ \sum_{e \in internal(KV, \{u \in V | \phi_V([u]_{\mathcal{R}_{Aufgabe}}) = v\})} (value(e, \textit{Software-Kategorie}))$

$= \sum_{w \in \{5,6\}} (value(w, \textit{Software-Kategorie}))$ da keine internen Konnektoren

$= value(5, \textit{Software-Kategorie}) + value(6, \textit{Software-Kategorie})$

$=$ *A-Software+A-Software*

$=$ *A-Software*

Die Belegung für das Attribut *Aufgabe* wird analog berechnet:

Da $14 \in V_{KV'} \setminus V_{KV}$ gilt:

$value'(14, Aufgabe) =$

$= \sum_{w \in \{u \in V | \phi_V([u]_{\mathcal{R}_{Aufgabe}}) = v\}} (value(w, Aufgabe))$
$+ \sum_{e \in internal(KV, \{u \in V | \phi_V([u]_{\mathcal{R}_{Aufgabe}}) = v\})} (value(e, Aufgabe))$

$= \sum_{w \in \{5,6\}} (value(w, Aufgabe))$(da keine internen Konnektoren)

$= value(5, Aufgabe) + value(6, Aufgabe)$

$= Dialog + Dialog$

$= Dialog$

## 6.4 Auswahlprojektion (Query)

Eine Auswahlprojektion (kurz Auswahl, Query) $\Xi$ wählt bestimmte Elemente einer Konfiguration aus und übernimmt nur diese in die Bildkonfiguration. Ausgewählte Elemente können Komponenten $\Xi_V$, Konnektoren $\Xi_E$ oder Attribute $\Xi_A$ sein. Das Auswahlkriterium ist jeweils ein einstelliges Prädikat $\mathcal{Q}$ auf der Menge der Komponenten $\mathcal{Q}_V$, Konnektoren $\mathcal{Q}_E$ oder Attribute $\mathcal{Q}_A$. In der Abbildung 6.6 ist rechts eine Auswahl auf der Menge der Komponenten dargestellt, in der die Software-Kategorie 'A-Software' als Kriterium für die Auswahl verwendet wurde. Es wurden alle Kindkomponenten gelöscht, die keine A-Software sind.

Zusätzlich wurden in der Abbildung 6.6 rechts für alle Kommunikationspfade, die in Szenarios aus $S_{KV}$ enthalten sind, Konnektoren ergänzt. Diese Abbildung wird als $\Xi_{VS}$ eingeführt.

### 6.4.1 Auswahl von Komponenten

Die Auswahlabbildungen $\Xi_V$ und $\Xi_{VS}$ werden in den nächsten Abschnitten definiert. Beide verwenden ein Prädikat $\mathcal{Q}_V : V \to \mathbb{B}$ bzw. $\mathcal{Q}_V \in \wp(V \times \mathbb{B})$ (dabei steht $\mathcal{Q}$ für Query) erreicht. Das Prädikat wählt bestimmte Komponenten einer Konfiguration $C = (V, E, J, H)$ aus. Nur diese werden in die Ergebniskonfiguration $C' = (V', E', J', H')$ übernommen und nur dann, wenn auch alle Vaterkomponenten $\mathcal{Q}_V$ erfüllen. Das Prädikat $\mathcal{Q}_V$ ist quasi eine Query auf der Menge der Komponenten $V$ von $C$. Die Konnektoren $E$ können dabei auf drei Arten behandelt werden:

1. Nur die Konnektoren werden übernommen, welche die verbliebenen Komponenten verbin-

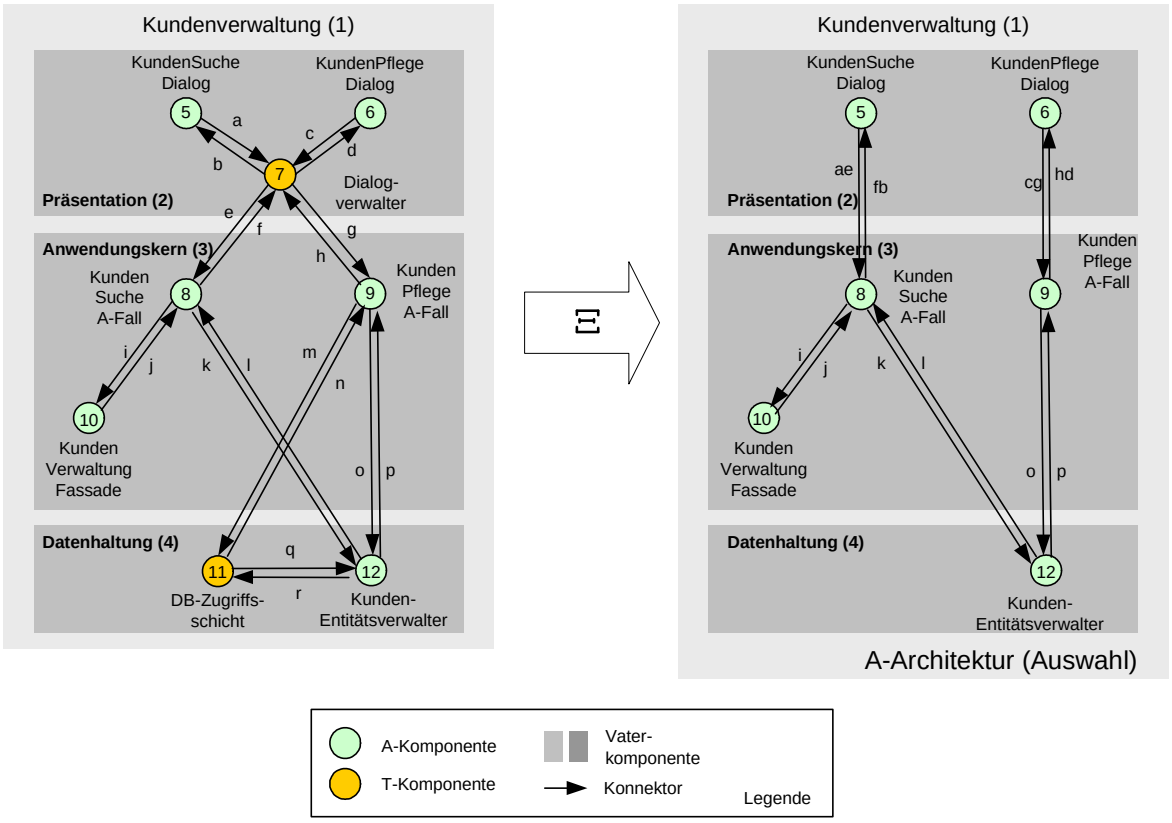


Abbildung 6.6: Komponentenauswahl der Komponente Kundenverwaltung

den. Diese Abbildung wird mit $\Xi_V$ bezeichnet.

2. Für jeden gerichteten Pfad aus der Konfiguration $C$, der zwischen je zwei verbliebenen Komponenten besteht, worin alle Zwischenkomponenten gelöscht wurden, wird ein neuer Konnektor ergänzt, welcher den gerichteten Pfad zu einem einzigen Konnektor zusammenfasst.

3. Für die unter 2. beschriebenen Pfade werden neue Konnektoren erzeugt, aber nur für die Pfade, die Teil eines spezifizierten Nutzungsszenarios $S(uc), uc \in UC$ sind. Diese Abbildung wird mit $\Xi_{VS}$ bezeichnet.

Nur für die Varianten 1 und 3 werden Projektionen definiert. Die Variante 2 erzeugt in der Regel zu viele Konnektoren, die nicht dem Zusammenspiel der Komponenten innerhalb der logischen Architektur entsprechen.

## Einfache Komponentenauswahl $\Xi_V$

Die einfache Auswahl $\Xi_V$ bildet eine Konfiguration mithilfe eines Auswahlprädikats $\mathcal{Q}_V : V \rightarrow \mathbb{B}$ auf der Menge der Komponenten auf eine Bildkonfiguration ab.

$\Xi_V : CONFIGURATION \times (COMPONENT \rightarrow \mathbb{B}) \rightarrow CONFIGURATION$

Über die Auswahlprojektion $\Xi_V(C, \mathcal{Q}_V) = C' = (V', E', J', H')$ werden nur die Komponenten $v \in V$ nach $C'$ übernommen, für deren Vaterkomponenten und sie selbst $\mathcal{Q}_V(v)$ gilt. Die verbleibenden Komponenten ergeben sich damit über $\{v \in V | \mathcal{Q}_V(v) \wedge \forall w \in allparents(C, v) : \mathcal{Q}_V(w)\}$. Nur die Konnektoren werden in die Bildkonfiguration übernommen, die zwischen den verbleibenden

Komponenten verlaufen. Die Zuordnungsfunktion muss entsprechend eingeschränkt werden. Für $\Xi_V(C, \mathcal{Q}_V) = C' = (V', E', J', H')$ gilt dann:

- $V' := \{v \in V | \mathcal{Q}_V(v) \wedge \forall x \in allparents(C, v) : \mathcal{Q}_V(x)\}$
- $E' := \{e \in E | J(e) \in V' \times V'\}$.
- $J' := J|_{E'}$
- $H' := H \cap (V' \times V')$

Die hierarchische Struktur $H'$ ist nur noch auf den verbliebenen Komponenten $V'$ definiert. Die hierarchische Struktur behält dabei ihre Eigenschaften als Baum bzw. Wald, da lediglich Blätter und Teilbäume entfernt werden.

Die Bildkonfiguration $C'$ ist unabhängig von der Ausprägung von $\Xi_V$ wohlgeformt, denn die hierarchische Struktur $H'$ geht aus der wohlgeformten hierarchischen Struktur $H$ durch Einschränkung auf $V' \times V'$ hervor.

Auf die Beispielkonfiguration $KV$ könnte etwa das Auswahlprädikat $\mathcal{Q}_{ASoftware}$ auf der Menge der Komponenten $v \in V_{KV}$ von $KV$ definiert sein. Dieses ist nur für A-Software erfüllt und dient damit zur Erzeugung der A-Architektur-Sicht nach [Sie04].

$$\mathcal{Q}_{ASoftware}(v) := \begin{cases} true: & value_{KV}(\textit{Software-Kategorie}, v) = \textit{A-Software} \\ & \vee value_{KV}(\textit{Software-Kategorie}, v) = \textit{0-Software} \\ & \vee value_{KV}(\textit{Software-Kategorie}, v) = \langle\rangle \\ false: & sonst \end{cases}$$

Unter Berücksichtigung der Belegungen aus Tabelle 6.1. Ergibt sich dann für die Komponenten und die Hierarchie von $KV' = \Xi_V(KV, \mathcal{Q}_{ASoftware}) = (V_{KV'}, E_{KV'}, J_{KV'}, H_{KV'})$:

- $V_{KV'} = \{1, 2, 3, 4, 5, 6, 8, 9, 10, 12\}$
- $E_{KV'} = \{i, j, k, l, o, p\}$
- $J_{KV'} = \{(i, (8, 10)), (j, (10, 8)), (k, (8, 12)), (l, (12, 8)), (o, (9, 12)), (p, (12, 9))\}$
- $H_{KV'} = \{(1, 2), (1, 3), (1, 4), (2, 5), (2, 6), (3, 8), (3, 9), (3, 10), (4, 12)\}$

### Komponentenauswahl mithilfe von Szenarios $\Xi_{VS}$

Bei der Auswahl $\Xi_V$ werden nur die Konnektoren übernommen, die zwischen zwei der ausgewählten Komponenten verlaufen. Die Auswahl $\Xi_{VS}$ berücksichtigt Szenarios, die auf der Konfiguration definiert sind. Wenn ein Szenario zwischen zwei Komponenten einen Kommunikationspfad enthält und die Komponenten auf diesem Pfad durch $\mathcal{Q}_V$ entfernt werden, dann wird ein zusätzlicher Konnektor erzeugt. Dieser verbindet die beiden Komponenten wieder.

Diese Variante der Auswahlprojektion kann verwendet werden, um Sichten wie die A-Architektur [Sie04] zu erzeugen. Diese Sicht stellt nur Komponenten der Kategorien 0-Software und A-Software dar. In dieser Sicht wird zusätzlich dargestellt, welche A-Komponenten miteinander kommunizieren. Häufig findet die Kommunikation jedoch über technische Komponenten (T-Software) statt. In der Beispielkonfiguration $KV$ ist das etwa die Komponente *Dialogverwalter*. Bei der Erzeugung der A-Architektur sollte diese Komponente herausgerechnet werden können und dies ist mit $\Xi_{VS}$ möglich. Die A-Architektur-Sicht wird in Abschnitt 7.6.1 noch diskutiert.

Um die Projektion $\Xi_{VS}$ definieren zu können, müssen zunächst Kommunikationspfade und Szenarios einer Konfiguration näher betrachtet werden.

**Pfade**

Um die nachfolgenden Definitionen zu vereinfachen, wird eine Hilfsmenge $PATH$ definiert, die alle möglichen Kommunikationspfade einer Konfiguration enthält:

$$PATH(C) := \{(e_1, \ldots, e_n) | \quad n \in \mathbb{N}, n > 1 \wedge e_i \in E, i \leq n \\ \wedge J(e_i) = (v_i, v_{i+1}); v_i, v_{i+1} \in V \text{ mit } v_j \neq v_k \forall j, k \leq n+1, j \neq k\}$$

Die Pfade in $PATH$ sind jeweils kreisfrei ($v_j \neq v_k \forall j, k \leq n+1, j \neq k$), d.h. jede Komponente wird höchstens einmal besucht.

Seien $v, w \in V$ zwei Komponenten für welche das Prädikat $\mathcal{Q}_V$ erfüllt ist, also $\mathcal{Q}_V(v) \wedge \mathcal{Q}_V(w)$ und alle Väter beider Komponenten erfüllen ebenfalls $\mathcal{Q}_V$, also $\forall x \in allparents(C, v) : \mathcal{Q}_V(x)$ und $\forall y \in allparents(C, w) : \mathcal{Q}_V(y)$.

Sei $p_{vw} = (e_1, \ldots, e_n), e_i \in E, i \leq n, n > 1, n \in \mathbb{N}$ ein Kommunikationspfad zwischen $v$ und $w$ in der Konfiguration $C$. Auf diesem Pfad werden die Komponenten $(v_1, \ldots, v_{n+1}), v_1 = v, v_{n+1} = w, v_i \in V, i \leq n+1$ besucht. Alle auf dem Pfad besuchten Zwischenkomponenten $v_2, \ldots, v_n$ oder deren Väter erfüllen das Prädikat nicht: $\exists x \in allparents(C, v_j) \cup \{v_j\} : \neg\mathcal{Q}_V(x), 1 < j \leq n$. Das bedeutet: In der Bildkonfiguration $C'$ entfallen alle in $p_{vw}$ enthaltenen Konnektoren. Die Auswahloperation ersetzt jeden dieser Pfade durch einen neuen Konnektor, welcher $v$ und $w$ direkt verbindet, wenn der gesamte Pfad in einem Nutzungsszenario enthalten ist.

Die Menge $PATH(C, \mathcal{Q}_V)$ schränkt die Kommunikationspfade auf die eben beschriebenen Pfade ein, so dass $\mathcal{Q}_V$ nur noch für die Anfangs- und Endkomponente jedes Pfades erfüllt ist.

$$PATH(C, \mathcal{Q}_V) \subseteq PATH(C)$$

$$PATH(C, \mathcal{Q}_V) = \quad \{(e_1, \ldots, e_n) \in PATH(C) | \quad J(e_i) = (v_i, v_{i+1}), i \leq n \wedge \\ \forall x \in allparents(v_1) \cup \{v_1\} : \mathcal{Q}(x) \wedge \\ \forall y \in allparents(v_{n+1}) \cup \{v_{n+1}\} : \mathcal{Q}(y) \wedge \\ \forall j, 1 < j \leq n : \exists z \in allparents(v_j) \cup \{v_j\} : \neg\mathcal{Q}(z)\}$$

Die Menge $PATH(C, \mathcal{Q}_V, UC, S)$ berücksichtigt schließlich auch die Anwendungsfälle, die über $S$ und $UC$ dargestellt werden:

$$PATH(C, \mathcal{Q}_V, UC, S) = \{(e_1, \ldots, e_n) \in PATH(C, \mathcal{Q}_V) | \exists uc \in UC : e_i \in S(uc) \forall i \leq n\}$$

Die Menge $PATH(C)$ wird bereits für kleine Konfigurationen groß. Daher wird hier nur die Menge $PATH(KV, \mathcal{Q}_{ASoftware})$ als Beispiel berechnet. Diese Menge enthält nur Pfade, bei denen alle Zwischenkomponenten entfallen, damit kommen nur noch Pfade in betracht, bei denen alle Zwischenkomponenten die Software-Kategorie T-Software oder AT-Software haben zusätzlich müssen Anfangs- und Endkomponente die Kategorie A-Software oder 0-Software haben.

Die Pfade in der Kundenverwaltung $KV$ müssen damit über die Komponenten 7 und 11 verlaufen, da beide zur Kategorie T-Software gehören. Die Pfade müssen bei einem direkten Nachbarn der Komponenten beginnen und bei einem anderen Nachbarn aufhören. Bei beiden Komponenten kann von jeder Nachbarkomponente jede andere Nachbarkomponente über einen Pfad der Länge 2 erreicht werden. Damit gilt:

$$PATH(KV, \mathcal{Q}_{ASoftware}) = \quad \{(a,d), (a,e), (a,g), (c,b), (c,e), (c,g), (f,b), (f,d), (f,g), \\ (h,b), (h,d), (h,e), (m,q), (r,n)\}$$

Die Menge $PATH(KV, \mathcal{Q}_{ASoftware}, UC_{KV}, S_{KV})$ schränkt $PATH(KV, \mathcal{Q}_{ASoftware})$ auf die Szenarios ein, die der Konfiguration $KV$ zugeordnet sind. Die Szenarios sind gegeben über:

$$S_{KV} = \{(uc_1, \{a, e, k, r, q, l, f, b\}), (uc_2, \{c, g, o, r, q, p, h, d\}), (uc_3, \{j, k, r, q, l, i\})\}.$$

In der Pfadmenge entfallen nun alle Pfade, die nicht in einem der Szenarios enthalten sind. Die Pfade $(a, d)$ oder $(a, g)$ kommen beispielsweise in keinem der Szenarios vor. Die Pfade werden damit auf

wahrscheinlich[6] verwendete Pfade eingeschränkt:

$PATH(KV, \mathcal{Q}_{ASoftware}, UC_{KV}, S_{KV}) = \{(a,e),(c,g),(f,b),(h,d)\}$.

### Vergröberung von Pfaden

Für die Auswahlprojektion werden Pfade von Konnektoren jeweils zu einem Konnektor vergröbert. Dazu wird die Aggregation für Pfade $\phi_{PATH}$ innerhalb einer wohlgeformten Konfiguration $C = (V, E, J, H)$ definiert mit

$\phi_{PATH} : (E \times \cdots \times E) \rightarrow CONNECTOR \cup \{\langle\rangle\}$.

Ist das übergebene Konnektoren-Tupel kein Pfad innerhalb der Konfiguration $C$, ist das Ergebnis von $\phi_{PATH}$ undefiniert $\langle\rangle$. Da $\phi_{PATH}$ nur innerhalb der hier dargestellten Projektionen verwendet wird, tritt dieser Fall jedoch niemals auf.

Sei $C = (V, E, J, H)$ eine wohlgeformte Konfiguration. Das n-Tupel $p = (e_1, \ldots, e_n), e_i \in E, i \leq n$ sei ein Pfad durch die Konfiguration $C$. Dem Pfad kann ein n+1-Tupel von Komponenten zugeordnet werden, das sind die besuchten Komponenten $(v_1, \ldots, v_{n+1})$ mit $J(e_i) = (v_i, v_{i+1})$. Die Komponenten $v, w \in V$ seien die beiden Komponenten welche der Pfad verbindet $v_1 = v$ und $v_{n+1} = w$. Die Zuordnung des Komponenten-Tupels zum Konnektor-Tupel ist nicht eindeutig, da $C$ Mehrfachkonnektoren enthalten kann.

Dann ist $e_{p_{vw}} = \phi_{PATH}((e_1, \ldots, e_n))$ die Aggregation des Pfades $(e_1, \ldots, e_n)$ mit denselben Anfangs- und Endpunkten, also $J(e_{p_{vw}}) = (v, w)$, genau dann, wenn es eine Konfiguration $C' = (V', E', J', H')$ gibt mit:

- $V' := V$
- $E' := E \cup \{e_{p_{vw}}\}, e_{p_{vw}} = \phi_{PATH}((e_1, \ldots, e_n)), e_{p_{vw}} \notin E$,. Die zusammengefassten Konnektoren verbleiben in $E$, zusätzlich wird ein neuer Konnektor $e_{p_{vw}} = \phi_{PATH}((e_1, \ldots, e_n))$, welcher die Zusammenfassung darstellt, ergänzt.
- $J'(e) := \begin{cases} J(e): & e \in E \\ (v,w): & e = \phi_{PATH}((e_1, \ldots, e_n)), \\ & J(e_1) = (v, v_1) \wedge J(e_n) = (v_n, w); v, w, v_1, v_n \in V \end{cases}$
- $H' := H$.

Die $id(e_p) \in \mathbb{ID}$ des neuen Konnektors $e_{p_{vw}}$ wird eindeutig generiert und der $name(e_{p_{vw}}) \in \mathbb{ID}$ wird aus den Namen der zusammengefassten Konnektoren gemäß der durch die Ordnung in $\mathbb{ID}$ gegebenen Reihenfolge konkateniert:

$name(e_{p_{vw}}) := \sum_{i=1\ldots n, f_j \in (e_1, \cdots, e_n), name(f_i) \leq name(f_j), i \leq j \leq n} (name(f_i))$.

$\phi_{PATH}((e_1, \ldots, e_n)) = e_{p_{vw}}$ liefert damit (nach Definition von $\phi_{PATH}$) immer denselben Konnektor $e_{p_{vw}}$, unabhängig vom Zeitpunkt der Anwendung.

### Komponentenauswahl mithilfe von Szenarios

Mithilfe der Hilfsmengen kann nun eine Bedingung für die neuen Konnektoren definiert werden: Die durch neue Konnektoren ersetzten Pfade müssen vollständig in einem Szenario $s \in S$ enthalten sein.

[6] Mit dem Umfang der Szenarios an Konnektoren steigt das Risiko, dass nicht verwendete Pfade übernommen werden. Im Beispiel ist das etwa dann der Fall, wenn etwa ein Anwendungsfall $uc_4$ zur Suche und anschließender Änderung von Kundendaten mit dem Szenario $S(uc_4) = S(uc_1) \cup S(uc_2)$ gebildet wird. Dieses Szenario umfasst fast alle Konnektoren von $KV$ und damit fast die gesamte Menge $PATH(KV, \mathcal{Q}_{ASoftware})$.

$$\Xi_{VS}: \; CONFIGURATION \times (COMPONENT \to \mathbb{B}) \times \\ \wp(USECASE) \times (USECASE \to \wp(CONNECTOR)) \to CONFIGURATION$$

Die Mengen $E'$ und $J'$ werden für $\Xi_{VS}(C, \mathcal{Q}_V, UC, S) = C' = (V', E', J', H')$ damit wie folgt bestimmt:

- $V' := \{v \in V | \mathcal{Q}_V(v) \wedge \forall x \in allparents(C, v) : \mathcal{Q}_V(x)\}$
- $E' := \{e \in E | J(e) \in V' \times V'\} \cup \{\phi_{PATH}((e_1, \ldots, e_n)) | (e_1, \ldots, e_n) \in PATH(C, \mathcal{Q}_V, UC, S)\}$.
- $J'(e) := \begin{cases} J(e) & : J(e) \in V' \times V' \\ (v_1, v_{n+1}) & : e = \phi_{PATH}((e_1, \ldots, e_n)); (e_1, \ldots, e_n) \in PATH(C, \mathcal{Q}_V, UC, S); \\ & J(e_i) = (v_i, v_{i+1}); \end{cases}$
- $H' := H \cap (V' \times V')$

Die Menge der Konnektoren wird auf die tatsächlich durch Szenarios verwendete begrenzt. Komponenten, die über die Auswahl entfallen sind, können so 'herausgerechnet' werden.

Die Menge $PATH(KV, \mathcal{Q}_{ASoftware}, UC_{KV}, S_{KV}) = \{(a, e), (c, g), (f, b), (h, d)\}$ wurde oben bereits bestimmt. Für diese vier Pfade werden in $KV' = \Xi_{VS}(KV, \mathcal{Q}_{ASoftware})$ vier neue Konnektoren ergänzt. Diese vier Konnektoren bilden den Unterschied zur einfachen Auswahl $\Xi_V(KV, \mathcal{Q}_{ASoftware})$.

Neu erzeugt werden die Konnektoren $\phi_{PATH}((a, e)) = ae, \phi_{PATH}((c, g)) = cg, \phi_{PATH}((f, b)) = fb, \phi_{PATH}((h, d)) = hd$. Sie verbinden jeweils den Anfangs- und Endpunkt der Pfade, über die sie berechnet wurden. Damit ergibt sich $KV' = \Xi_{VS}(KV, \mathcal{Q}_{ASoftware}, UC, S) = (V_{KV'}, E_{KV'}, J_{KV'}, H_{KV'})$ mit:

- $V_{KV'} = \{1, 2, 3, 4, 5, 6, 8, 9, 10, 12\}$
- $E_{KV'} = \{i, j, k, l, o, p, ae, fb, cg, hd\}$
- $J_{KV'} = \{(i, (8, 10)), (j, (10, 8)), (k, (8, 12)), (l, (12, 8)), (o, (9, 12)), (p, (12, 9))$ $(ae, (5, 8)), (fb, (8, 5)), (cg, (6, 9)), (hd, (9, 6))\}$
- $H_{KV'} = \{(1, 2), (1, 3), (1, 4), (2, 5), (2, 6), (3, 8), (3, 9), (3, 10), (4, 12)\}$

## Auswahl von Komponenten in einer logischen Architektur

Sei $\mathcal{L} = (C, A, value, UC, S)$ eine logische Architektur und $\Xi_V : LA \times (COMPONENT \times \mathbb{B}) \to LA$, sei die Auswahlprojektion, mit $\Xi_V(\mathcal{L}, \mathcal{Q}_V) = \mathcal{L}' = (C', A', value', UC', S')$ und $\mathcal{Q}_V : V \to \mathbb{B}$ sei ein Prädikat auf der Menge der Komponenten. Wenn $C = (V, E, J, H)$ die Konfiguration von $\mathcal{L}$ ist, dann ist

$C' := \Xi_V(C, \mathcal{Q}_V) = (V', E', J', H')$ die Konfiguration von $\mathcal{L}'$.

Für $\Xi_{VS}$ ist:

$C' := \Xi_{VS}(C, \mathcal{Q}_V, UC, S) = (V', E', J', H')$ die Konfiguration von $\mathcal{L}'$ mit den Anwendungsfällen $UC$ und ihrer Abbildung auf Szenarios $S$ aus $\mathcal{L}$.

Für die Szenarios $S$ von $\mathcal{L}$ findet ebenfalls die Projektion statt, dabei müssen die jeweils die Besonderheiten der Variante $\Xi_{VS}$ berücksichtigt werden:

$\Xi_V$ Die einfache Auswahlabbildung $\Xi_V$ entfernt Komponenten und die entsprechenden Konnektoren ersatzlos aus der Konfiguration $C$. Die Nutzungsszenarios $S'$ zu den Anwendungsfällen $UC' := UC$ ergebnen sich durch Einschränkung auf die verbliebenen Konnektoren $E'$.

$S'(uc) := \{S(uc) \cap E'\}, uc \in UC$.

Die Szenarios aus der Beispielkonfiguration $KV$ werden um die fehlenden Konnektoren in $KV'$ vermindert, also auf $E_{KV'}$ eingeschränkt.

$S_{KV'} := \{(uc_1, \{k, l\}), (uc_2, \{o, p\}), (uc_3, \{j, k, l, i\})\}$.

$\Xi_{VS}$ kann definiert werden, indem $PATH(C, \mathcal{Q}_V, UC, S)$ verwendet wird: Wenn im Szenario $S(uc), uc \in UC$ Pfade aus $C$ der Form $(e_1, \ldots, e_n) \in PATH(C, \mathcal{Q}_V, UC, S)$ enthalten sind, werden anstatt jedes Pfades die Konnektoren $\phi_{PATH}((e_1, \ldots, e_n)) \in E'$ zu $S'(uc)$ hinzugefügt. Alle Konnektoren des Pfades werden entfernt, da sie auch in $C' = \Xi_{VS}(C, \mathcal{Q}_V)$ fehlen. Für die Bildszenarios $S'(uc), uc \in UC$ ergibt sich damit:

$$\begin{aligned} S'(uc) := \quad & \{e \in E' | e \in S(uc)\} \cup \\ & \{e \in E' | \exists (e_1, \ldots, e_n) \in PATH(C, \mathcal{Q}_V, UC, S) \\ & \wedge e = \phi_{PATH}((e_1, \ldots, e_n)) \\ & \wedge e_i \in S(uc) \text{ für } \forall 0 \leq i \leq n\} \end{aligned}$$

Die Szenarios aus der Beispielkonfiguration $KV$ werden um die fehlenden Konnektoren in $KV'$ vermindert, also auf $E_{KV'}$ eingeschränkt.

$S_{KV'} := \{(uc_1, \{ae, k, l, fb\}), (uc_2, \{cg, o, p, hd\}), (uc_3, \{j, k, l, i\})\}$.

Die Menge der Attribute bleibt unverändert

$A' := A$

auch die Belegung bleibt erhalten, sie wird jedoch eingeschränkt auf die verbliebenen Konnektoren und Komponenten:

$value' := value|_{V' \cup E'}$.

### 6.4.2 Auswahl von Konnektoren

Die Auswahlabbildung $\Xi_E$ wird über ein Prädikat $\mathcal{Q}_E : E \to \mathbb{B}$ definiert. Das Prädikat wählt bestimmte Konnektoren aus, die in die Ergebniskonfiguration $C'$ übernommen werden.

$\Xi_E : CONFIGURATION \times (CONNECTOR \to \mathbb{B}) \to CONFIGURATION$

Das Prädikat $\mathcal{Q}_E$ ist eine Query auf der Menge der Konnektoren der Konfiguration $C$.
Für $\Xi_E(C, \mathcal{Q}_E) = C' = (V', E', J', H')$ gilt:

- $V' := V$
- $E' := \{e \in E | \mathcal{Q}_E(e)\}$. Alle Konnektoren werden übernommen, für die das Auswahlprädikat $\mathcal{Q}_E$ erfüllt ist.
- $J' := J|_{E'}$
- $H' := H$

#### Auswahl von Konnektoren in einer logischen Architektur

Die Auswahl von Konnektoren beeinflusst ausschließlich die Menge der Konnektoren $E$ der Konfiguration $C = (V, E, J, H)$ und die Szenarios $S$, die beide zu einer logischen Architektur $\mathcal{L} = (C, A, value, UC, S)$ gehören. Zur Auswahl der Konnektoren wird ein Prädikat $\mathcal{Q}_E : E \to \mathbb{B}$ verwendet. Die Konnektorauswahl $\Xi_E$ ist für logische Architekturen definiert als

$\Xi_E : LA \times (CONNECTOR \to \mathbb{B}) \to LA$ .

Wenn die logische Architektur $\mathcal{L}' = \Xi_E(\mathcal{L}, \mathcal{Q}_E) = (C', A', value', UC', S')$ die Bildarchitektur von $\mathcal{L}$ ist. Die Auswahl von Konnektoren innerhalb einer Konfiguration ist oben definiert, es gilt:

$C' := \Xi_E(C, \mathcal{Q}_E)$.

Die Szenarios $S'(uc)$ zu den Anwendungsfällen $uc \in UC' := UC$ ergeben sich aus $S$ indem jedes einzelne Szenario auf die verbliebenen Konnektoren in $E'$ eingeschränkt wird

$S'(uc) := S(uc) \cap E'$.

Die Menge der Attribute bleibt unverändert

$A' := A$

auch die Belegung bleibt erhalten, sie wird jedoch eingeschränkt auf die verbliebenen Konnektoren und Komponenten:

$value' := value|_{V' \cup E'}$.

Beispiele für die Auswahl von Konnektoren in logischen Architekturen werden in den nachfolgenden beiden Kapiteln dargestellt.

### 6.4.3 Auswahl von Attributen

Die Auswahl von Attributen $\Xi_A$ ändert weder die Konfiguration $C$ noch die Anwendungsfälle $UC$ oder die Szenarios $S$ einer logischen Architektur $\mathcal{L} = (C, A, value, UC, S)$. Die Menge der Attribute $A$ und damit auch die Belegung $value$ werden geändert. Die Abbildungsprojektion ist definiert mit:

$\Xi_A : LA \times (ATTRIBUTE \to \mathbb{B}) \to LA$.

$\Xi_A$ verwendet zur Auswahl der Attribute ein Prädikat $\mathcal{Q}_A : A \to \mathbb{B}$. Damit berechnet sich die Attributmenge $A'$ der Bildarchitektur $\mathcal{L}' = (C', A', value', UC', S')$:

$A' := \{a \in A | \mathcal{Q}_A(a)\}$

Die Belegung $value$ wird auf $A'$ eingeschränkt:

$value' := value|_{A'}$.

Für die Bildarchitektur gilt damit insgesamt $\Xi_A(\mathcal{L}, \mathcal{Q}_A) = \mathcal{L}' = (C, A', value', UC, S)$, wobei $A'$ und $value'$ wie oben dargestellt, berechnet werden. Beispiele für die Auswahl von Attributen in logischen Architekturen werden in den nachfolgenden beiden Kapiteln dargestellt.

## 6.5 Zusammenfassung

Dieses Kapitel definiert allgemeine Projektionen, die eine logische Architektur in eine andere überführen. Die Bildarchitektur enthält in der Regel weniger Informationen als die Urbildarchitektur. Die Projektionen können konkateniert werden, so dass damit komplexere Projektionen definiert werden können. Die Projektionen sind allgemein definiert, $\Phi_{\mathbb{N}}$ benötigt eine Hierarchieebene $n \in \mathbb{N}_0$, $\Psi$ erfordert eine Äquivalenzrelation $\mathcal{R}$ auf der Menge der Komponenten und $\Xi$ basiert auf Auswahlprädikaten $\mathcal{Q}$.

Die Abbildungen werden in den nun folgenden Kapiteln zur Erzeugung von Sichten auf eine logische Architektur nach bestimmten Kriterien verwendet.

# Teil III

# Anwendungen und praktische Umsetzung

# Kapitel 7

# Architektursichten

Typischerweise werden die Blickwinkel, die Architektursichten zugrunde liegen, informell über Beispiele und begleitende Texte beschrieben (vgl. z.B. [Kru95, Sie04, Sta05, VAC+05]). Die Architektursichten werden aus diesem Grund in der Regel manuell erstellt. Weiterhin werden Informationen über die Strukturen des Systems und dessen Verhalten dargestellt. Zusätzliche Aspekte wie Kosten, Termine oder etwa Ausfallwahrscheinlichkeit, die jeweils für bestimmte Stakeholder nützlich sind, werden nicht gezeigt oder diskutiert.

Dieses Kapitel schlägt zunächst ein Ordnungsschema vor, mit dem Architektursichten charakterisiert und eingeordnet werden können, dieses wird im Abschnitt 7.1 dargestellt. Entlang der darin vorgestellten Bereiche Architekturart, Auflösung und Sachverhalt können Architektursichten für ein IT-System gebildet werden. Abschnitt 7.2 stellt vor, wie mithilfe der Architekturtheorie entsprechende Blickwinkel definiert und Sichten damit generiert werden können. Die Abschnitte 7.4 bis 7.5 stellen Beispiele für Architekturblickwinkel und -sichten vor. Die Architekturtheorie stellt für jede Sicht jeweils die darzustellenden Informationen bereit, die Visualisierung erfolgt über automatisch oder manuell gelayoutete gerichtete Multigraphen.

## Übersicht

## 7.1 Ordnungsschema für Architektursichten

In diesem Abschnitt wird eine Systematik in Form eines Ordnungsschemas für Architektursichten und -blickwinkel vorgeschlagen. Das Schema hat das Ziel, Kriterien zu definieren, mit denen Architektursichten inhaltlich eingeordnet und kombiniert werden können. In der Literatur sind mehrere Vorschläge für derartige Schemata zu finden, etwa bei Zachmann und Sowa [Zac87, SZ92], Wieringa [WBFG03], Scheer [Sch97] oder Schlosser [Sch05, S.157ff]. Diese Schemata enthalten in der Regel zwei Dimensionen. Eine Dimension stellt entweder Stakeholder-Gruppe (Nutzer, Planer, Entwickler, ...) dar oder die verschiedenen Phasen innerhalb des Software-Entwicklungsprozesses. Die zweite Dimension stellt den Gegenstand der Betrachtung dar, etwa Daten, Funktionen oder Organisationsaspekte.

Das hier vorgestellte Schema hat drei Dimensionen, die inhaltlich unabhängig sind, so dass sie als orthogonale Dimensionen verstanden werden können. Die Dimensionen sind Auflösung, Art und Sachverhalt:

- Auflösung: Ein betriebliches Informationssystem kann auf verschiedenen Granularitätsebenen (in verschiedenen Auflösungen) betrachtet werden. Es kann als Teil einer IT-Landschaft, als Einzelsystem zusammen mit seinen Nachbarn oder in einer Glass-Box Sicht mit seinen Einzelteilen betrachtet werden.
- Architekturart: Ein betriebliches Informationssystem hat unterschiedliche Architekturdarstellungen, etwa die logische Architektur oder die Implementierungsarchitektur. Diese Darstellungen haben unterschiedliche Betrachtungsgegenstände, etwa logische Bestandteile des Systems oder dessen Quelltexte. Architekturarten sind für verschiedene Stakeholder-Gruppen interessant. Die Nutzer sind beispielsweise an den Anforderungen aber selten an den Details der Implementierung interessiert.
- Dargestellter Sachverhalt: Die Architektur eines betrieblichen Informationssystems kann verschiedene Aspekte darstellen, etwa Daten, Funktionen oder Sicherheitsaspekte. Sachverhalte werden auch in Kombination dargestellt, etwa Daten zusammen mit Zugriffsrechten.

Die Abbildung 7.1 zeigt die Achsen des Ordnungsschemas. Die Achsen sind zur Einordnung von Architekturbeschreibungen gedacht. Eine Architekturbeschreibung kann mehrere Punkte auf einer Achse abdecken. Es kann beispielsweise Daten und deren Verortung im Netzwerk darstellen oder Anwendungsfälle aus den Anforderungen zusammen mit der Quelltextstruktur.

### 7.1.1 Auflösung

Ein Kundenverwaltungssystem liegt offensichtlich auf einer anderen Detaillierungsebene wie eine Kundenklasse. Das Observer-Pattern [GHJV94, S.293] ist auf einer anderen Betrachtungsebene wie die Schichtenarchitektur [BMR+00, SG96]. Wird die Frage der Granularität nicht explizit betrachtet, führt das oft zu Missverständnissen, wie Maier und Rechtin bemerken [RM00, S.269]: *One person's architecture is another person's detail. One person's system is another's component*.

Eine Detaillierungsebene entspricht einem bestimmten Abstraktionsniveau bzw. einer bestimmten Auflösung. Hier werden drei Detaillierungsebenen verwendet: Auf der obersten Ebene werden viele betriebliche Informationssysteme betrachtet, beispielsweise die gesamte IT-Landschaft eines Unternehmens. Auf der nächsten Ebene wird genau ein betriebliches Informationssystem als Black-Box mit seinen Nachbarsystemen und Nutzergruppen betrachtet. Die interne Strukturierung aus Bausteinen (Komponenten) wird in einer weiteren Ebene sichtbar, in der das System als Glass-Box betrachtet wird. Weitere Verfeinerungen sind möglich, etwa wenn die Detailstruktur der Bausteine betrachtet wird. Solche oder ähnliche Unterteilungen in Ebenen werden von mehreren Autoren vorgeschlagen, etwa von Herzum und Sims [HS00, S. 36ff], Stützle [Stü02, S. 39], Malveau und Mowbray [MM00, S.

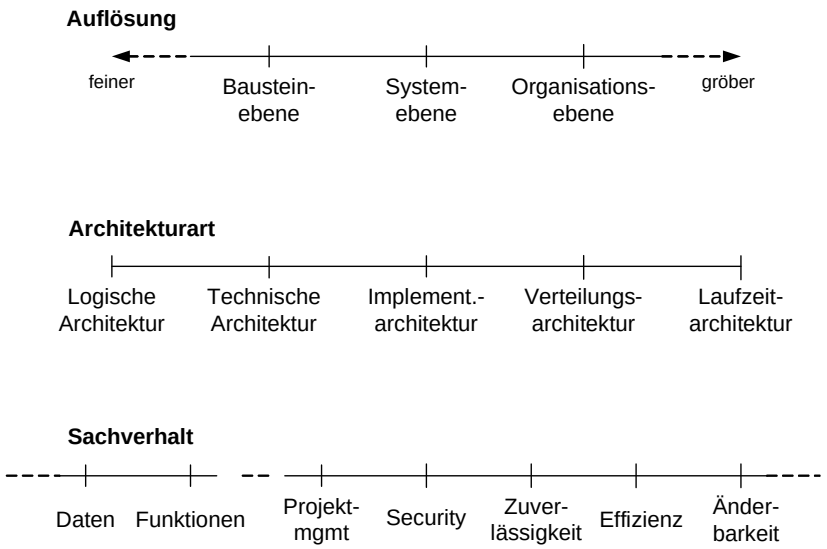


Abbildung 7.1: Dimensionen des Ordnungsschemas für Architektursichten

29] oder Vogel et al. [VAC+05, S. 66]. Schlosser verwendet in seiner Arbeit [Sch05] die von der VSI Alliance [Sys01] definierten Auflösungen und erweitert diese: Die Auflösungen werden wie hier für Strukturaspekte aber auch für die zeitliche Auflösung (beginnend bei Schaltzeiten von Gattern im Bereich von Pikosekunden bis hin zu zeitlich aufeinander folgenden Systemereignissen im Bereich einiger 10 Millisekunden), Datenauflösung (beginnend bei Bits bis hin zu Tokens) oder die funktionale Auflösung (digitale Logik bis hin zu mathematischen Formeln).

Die vorliegende Arbeit verwendet drei Detaillierungsebenen nach Vogel et al. [VAC+05]. Diese Unterteilung ist mit den drei Ebenen Organisationsebene, Systemebene und Bausteinebene einfach und für diese Arbeit ausreichend[1]. Vogel et al. verwenden diese Einteilung zur Einordnung von Architekturdarstellungen. Die Detaillierungsebenen werden dort Architekturebenen genannt.

**Organisationsebene** Die Organisationsebene beschäftigt sich mit ganzen Organisationen, das sind Unternehmen, Vereinigungen und Ähnliches. Organisationen verfügen in der Regel über viele betriebliche Informationssysteme, die miteinander interagieren. Geschäftsprozesse werden über die Organisation hinweg definiert und durch das Zusammenspiel mehrerer Systeme abgewickelt. Systeme werden generell als Black-Box betrachtet, ihr Zusammenspiel und ihr Beitrag zu den Geschäftsprozessen sind von Bedeutung. Vorgaben und IT-Standards, etwa zur technischen Infrastruktur, die ein Single-Sign-On erlaubt, gelten organisationsweit für jedes IT-System und deren Verbindungen. Auf der Organisationsebene spricht man von *Enterprise Architecture* sowie von IT-Landschaften und ihrer Bebauung [Kel06].

**Systemebene** Die Systemebene betrachtet ein betriebliches Informationssystem in seiner direkten Umgebung. Das System wird als Black-Box angesehen. Die Außensicht des Systems (d.h. die Systemgrenze) wird in dieser Ebene gezeigt. Verbindungen zu Nachbarsystemen und Nutzergruppen werden dargestellt.

**Bausteinebene** Auf der Bausteinebene wird das betriebliche Informationssystem als Glass-Box angesehen. Die wichtigsten Bestandteile des Systems und ihre Zusammenarbeit werden betrachtet. Funktionale und Qualitätseigenschaften gehören zur Außensicht einzelner Bausteine. Auch die Innensicht einzelner Bausteine kann dargestellt werden. Zusätzliche feinere Unterteilungen können ergänzt werden.

[1] Die von Schlosser verwendeten Auflösungen sind für den Entwurf betrieblicher Informationssysteme zu feinteilig.

Jede Detaillierungsebene erfordert eigene Notationen und Methoden: Auf der Organisationsebene werden beispielsweise komplette Geschäftsprozesse und Organisationsstrukturen betrachtet, während auf der Bausteinebene Teile von Anwendungsfällen umgesetzt werden.

Weitere Ebenen können je nach Schwerpunkt der Betrachtung ergänzt werden, etwa eine unternehmensübergreifende Ebene oberhalb der Organisationsebene, welche die Systeme des Unternehmens im Zusammenspiel mit Lieferanten- und Kunden-Systemen darstellt oder eine feinteilige Sicht unterhalb der Bausteinebene, beispielsweise nach dem Schema von Herzum und Sims [HS00, S. 36ff]. Sie definieren drei Arten von Komponenten:

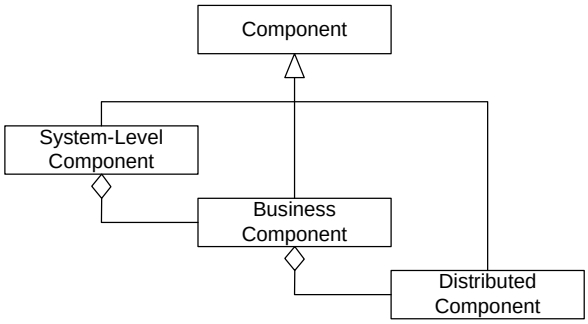


Abbildung 7.2: Komponenten in verschiedenen Detaillierungsebenen [HS00, S. 36ff]

Ein betriebliches Informationssystem entspricht einer System-Level Component (Systemebene). Dieses ist aus Bausteinen, den Business Components, zusammengesetzt (Bausteinebene). Die Business-Components bestehen ihrerseits aus Distributed Components. Während die Business Components ein in sich fachlich abgeschlossenes Teilsystem darstellen, etwa eine Vertragsverwaltung, ist eine Distributed Component eine Einheit der Verteilung auf einen Rechner (eine Tier). Nach dem Schema von Herzum und Sims kann zu der hier definierten Bausteinebene also eine weitere Unterebene definiert werden, die sich an Verteilungsaspekten orientiert.

### 7.1.2 Architekturart

Wie schon in Abschnitt 3.3 dargestellt, werden Architekturarten hier nach ihren Darstellungsgegenständen (Logische Komponenten, Code, Bibliotheken, ...) unterteilt. Die Viewtypes nach Clements et al. [CBB+03] sind konzeptionell mit den hier vorgestellten Architekturarten vergleichbar. Mehrere Autoren, etwa Kruchten [Kru95] oder Hofmeister et al. [HNS99a] schlagen für einige der Architekturarten spezifische Blickwinkel vor.

**Logische Architektur** Die logische Architektur beschreibt die logischen Elemente eines IT-Systems jedoch unabhängig von der gewählten Implementierungssprache und -technik. Die Dekomposition des Funktionsumfangs des IT-Systems in logische Komponenten oder die Aufteilung eines logisches Datenmodells auf die logischen Komponenten gehören zur logischen Architektur.

**Technische Architektur** Die technische Architektur beschreibt den technischen Entwurf des IT-Systems in Hardware und Software. Bei betrieblichen Informationssystemen beschränkt sich die technische Architektur in der Regel auf Software. Die Verfeinerungen logischer Komponenten als Software-Komponenten und die Beschreibung des Trägersystems sind in der Beschreibung der technischen Architektur enthalten.

**Implementierungsarchitektur** Die Implementierungsarchitektur beschreibt die Organisation und die Verwaltung der Quelltexte (Klassen, Interfaces, Prozeduren, Module). Sie beschreibt, wie Quelltexte physisch auf kompilierbare Dateien und Verzeichnisse verteilt sind und wie diese

zu Bibliotheken zusammengebunden werden. Das Konfigurationsmanagement findet auf und mithilfe der Implementierungsarchitektur statt.

**Verteilungsarchitektur** Die Verteilungsarchitektur (Deployment-Architektur) beschreibt die Verteilung des Software-Systems auf dem Trägersystem aus Hardware, Netzwerk, Betriebssystemen und weiteren Laufzeitumgebungen (Java Virtual Machine, Application Server, etc.). Über die Verteilungsarchitektur werden besonders Qualitätsaspekte berücksichtigt, etwa Ausfallsicherheit durch Redundanz oder hoher Transaktionsdurchsatz über Parallelisierung.

**Laufzeitarchitektur** Die Laufzeitarchitektur beschreibt das IT-System zur Laufzeit und seine Dekomposition in Prozesse und Threads. Aspekte der Nebenläufigkeit und Ressourcenverwaltung werden zusätzlich dargestellt.

Die Architekturarten werden in der Regel nicht sequentiell erstellt: Im Rahmen der Spezifikation und des Entwurfs entstehen neben der logischen Architektur normalerweise auch erste Entwürfe der technischen Architektur und der Laufzeitarchitektur. Häufig steht die Verteilungsarchitektur durch die Vorgaben des vorhandenen Trägersystems bereits in der Anforderungsanalyse fest. Die in der Verteilungs- und Laufzeitarchitektur beschriebenen Sachverhalte entscheiden über die Qualitätseigenschaften des IT-Systems, so dass diese nicht erst kurz vor der tatsächlichen Inbetriebnahme des IT-Systems definiert werden sollten.

In einem konkreten Projekt bestimmen das Vorgehensmodell und/oder die verwendete Entwicklungsmethode, in welchen Schritten ein betriebliches Informationssystem erstellt wird und mit welchen Ergebnissen jeder Schritt abgeschlossen wird.

### 7.1.3 Dargestellter Sachverhalt

Ein IT-System kann mit unterschiedlichen Schwerpunkten (Sachverhalten, Perspektiven [RW05]) betrachtet werden und zwar unabhängig von der Architekturart oder der Auflösung. Sachverhalte können Qualitätseigenschaften darstellen, wie sie in jeder Architekturart und Auflösung vorkommen, etwa Security oder Änderbarkeit. Sachverhalte können auch das Projektmanagement betreffen und etwa Planungs- oder Projektverfolgungsinformationen darstellen. Typische Sachverhalte sind weiterhin Daten und Funktionen. Für jede Architekturart wird etwa dargestellt, welche Komponente für welche Daten verantwortlich ist oder welchen Teil der funktionalen Anforderungen umsetzt.

Die Sachverhalte bilden die dritte 'Dimension' des hier vorgestellten Ordnungsschemas, wobei eine Architektursicht mehrere verschiedene Sachverhalte gleichzeitig darstellen kann. Scheer [Sch01, S. 1] schlägt in seinem ARIS Modell vier Sachverhalte vor, das sind Daten, Funktionen, Organisation und Leistung. Er beschreibt diese jeweils isoliert und verknüpft sie schließlich in der Steuerungssicht. Zachman und Sowa unterscheiden sechs Sachverhalte [Zac87, SZ92], das sind Daten (What), Funktionen (How), Netzwerk und Ort (Where), Personen und Organisation (Who), Zeit (When) sowie Motivation (Why) und Rozanski und Woods unterscheiden die Sachverhalte mit Qualitätseigenschaften wie Security, Performance and Scalability, Availability and Resilience, Evolution und weitere wie etwa Development Ressource [RW05]. Mehrere häufig beschriebene Sachverhalte sollen im Folgenden beispielhaft vorgestellt werden:

**Daten[2]:** Betriebliche Informationssysteme sind datenzentriert. Gespeicherte Stamm- und Bewegungsdaten machen einen wichtigen Mehrwert aus. Die logische Architektur kann beispielsweise das logische Datenmodell auf Komponenten aufteilen [HS00, Oes01, GA03]. In der technischen Architektur wird das Datenmodell in ein Datenbankschema umgesetzt [Sie02b] und die Verteilungsarchitektur definiert, in welcher Datenbank das Schema angelegt wird. Zur Laufzeit sind der nebenläufige Zugriff auf Daten über Transaktionen und dazu passende Sperrkonzepte wichtig. Die Betrachtung der Daten schließt häufig auch den Fluss der Daten durch das IT-System mit ein [Wie03], insbesondere für Untersuchungen im Zusammenhang zum Zugriffsschutz.

**Funktionen**[3]**:** Der Funktionsumfang wird in den verschiedenen Architekturarten dargestellt: Die logische Architektur zeigt, wie sich der Funktionsumfang eines IT-Systems auf seine logischen Komponenten aufteilt. Auch in anderen Architekturarten ist es wichtig zu wissen, welche Bibliothek oder welcher Prozess die jeweiligen funktionalen Anforderungen erfüllt, etwa um durch Redundanz dieser Elemente die Zuverlässigkeit oder den Durchsatz des IT-Systems zu erhöhen.

**Security**[4]**:** Daten und Funktionen eines IT-Systems sollen nicht für jeden Benutzer sichtbar, verwendbar und manipulierbar sein. Während der Anforderungsanalyse wird beispielsweise ein erstes Rechtemodell für Benutzer festgelegt. In der Verteilungssicht kann ein IT-System in mehrere Bereiche in Bezug zum Zugriffsschutz eingeteilt werden, etwa einen *trusted*- und einen *untrusted*-Bereich [Jür05]. Über die Analyse des Nachrichtenflusses in der logischen Architektur können etwa Angriffspunkte identifiziert werden [BCK03] oder für die Schnittstellen der technischen Architektur können Zugriffsrechte vergeben werden [Sun06]. Die Verteilungsarchitektur stellt etwa verschlüsselnde Netzwerkprotokolle oder Netzwerkbereiche wie eine demilitarisierte Zone dar.

**Verfügbarkeit und Ausfallsicherheit**[5]**:** Die logische Architektur kann bereits Informationen zur Verfügbarkeit des IT-Systems darstellen, etwa die erwarteten Verfügbarkeiten der einzelnen Systembestandteile. Komponenten, welche die Verfügbarkeit unterstützen, können hervorgehoben werden: etwa eine Komponente, die den Offline-Betrieb von Clients ermöglicht oder Komponenten für Backup und Recovery. In der Verteilungsarchitektur werden Redundanzen im IT-System dargestellt, welche die Verfügbarkeit und die Ausfallsicherheit erhöhen können, etwa ein Cluster von Rechnern. Werden die Komponenten in der jeweiligen Architekturart mit ihren Ausfallwahrscheinlichkeiten dargestellt, werden Analysetechniken wie die Fehlerbaumanalyse möglich.

**Antwortzeiten, Durchsatz und Ressourcenverbrauch**[6]**:** Anforderungen und Informationen zu Antwortzeiten, Durchsatz und dem Ressourcen-Verbrauch können in allen Architekturarten relevant sein: So kann schon in der logischen Architektur dargestellt werden, welcher Transaktionsdurchsatz von einer Komponente geleistet werden muss oder welche Latenzzeiten Konnektoren haben (dürfen). Wichtige Information für den Architekturentwurf ist die Häufigkeit in der ein Anwendungsfall in einer Zeiteinheit aufgerufen wird. In der Verteilungsarchitektur kann die verfügbare Netzwerkbandbreite oder die zu erwartenden Latenzzeiten des verwendeten Netzwerks dargestellt werden. Ebenso können Messergebnisse für Antwortzeiten oder Durchsatz aus dem technischen Durchstich [HT99] eingetragen werden.

**Planung: Kosten, Plan- und Ist-Termine, Risiken**[7]**:** Dieser Sachverhalt ist Thema von Kapitel 8. Für den Architekten und den Auftraggeber ist eine Kosten und Terminsicht auf das IT-System wichtig, damit kann beispielsweise eine Übersicht über den Fertigstellungsgrad des IT-Systems gewonnen werden, ebenso ist die Diskussion über Kosten und Nutzen einiger Bestandteile des IT-Systems möglich.

**Motivation und Begründung**[8]**:** Der wirtschaftliche Nutzen ist die Daseinsberechtigung eines betrieblichen Informationssystems. Dieser Nutzen kann auf die Teile des IT-Systems herunter gebrochen werden, darüber können beim Bau, beim Betrieb und bei der Weiterentwicklung Wirtschaftlichkeitsbetrachtungen durchgeführt werden. So könnte jeder logischen Komponente der Nutzen quantitativ in Euro oder qualitativ über einen Beitrag zu den Zielen oder Strategien des Unternehmens zugeordnet werden. Logische Komponenten, deren Daten und Funktionen keinen oder nur geringen Nutzen erbringen, können möglicherweise abgeschaltet oder gar nicht erst gebaut werden.

Diese Liste ist nicht vollständig: Sachverhalte können nach Projektkontext ergänzt oder entfernt werden: Ist das betriebliche Informationssystem vielen Änderungen ausgesetzt, ist etwa die Darstellung eines Sachverhalts zur *Änderbarkeit*, Änderungswahrscheinlichkeit bzw. Stabilität nützlich. Soll Qualitätssicherung betrieben werden können Sichten hilfreich sein, die Qualitätsmetriken visualisieren (vgl. etwa Termeer et al. [TLTC05]).

Zachman und Sowa schlagen vor, die Sachverhalte isoliert zu betrachten, um die Modelle nicht zu überfrachten. Beispielsweise wird nur das Datenmodell oder nur der Geschäftsprozess betrachtet. Die Konsistenz der Sachverhalte wird über Konsistenzregeln gewährleistet. Die gleichzeitige Darstellung mehrerer Sachverhalte ist jedoch häufig erforderlich. Etwa die Darstellung, welcher Organisationseinheit welche Daten gehören. Dies ist in dem hier vorgeschlagenen Ordnungsschema explizit erlaubt.

## Vektordarstellung für Architektursichten

Um eine Architektursicht zu kennzeichnen wird eine Tupel-Schreibweise[9] für Architektursichten eingeführt. Sie enthält für Auflösung, Architekturart und Sachverhalt jeweils einen Eintrag. Die Einträge Sachverhalt und Architekturart dürfen mengenwertig sein, da eine Sicht mehrere Sachverhalte oder Architekturarten darstellen kann.

Hierzu werden zunächst drei Grundmengen definiert. In einem konkreten Projekt könnten diese Mengen erweitert werden, etwa durch feinere oder gröbere Auflösungen sowie andere Sachverhalte:

*Auflösung* := {*Organisation, System, Baustein*}

*Architekturart* := {*Logisch, Technisch, Implementierung, Verteilung, Laufzeit*}

*Sachverhalt* := {Daten, Funktionen, Security, Verfügbarkeit, Antwortzeiten und Durchsatz, Planung, Begründung }

Damit wird eine Architektursicht $\mathcal{AS}$ charakterisiert durch das Dreitupel $\mathcal{AS} = (AL, AA, SV)$ mit $AL \in$ *Auflösung*, $AA \subseteq$ *Architekturart*, $AA \neq \emptyset$ und $SV \subseteq$ *Sachverhalt*. Anders ausgedrückt:

$\mathcal{AS} \in$*Auflösung*$\times \wp(Architekturart) \times \wp(Sachverhalt)$

Eine Architektursicht $\mathcal{AS} = (AL, AA, SV)$, für die gilt $|AL| = |AA| = |SV| = 1$, wird als *rein* bezeichnet.

Zusätzlich kann formuliert werden, wann zwei Architektursichten kompatibel sind und daher vereinigt werden können. Zwei Architektursichten $\mathcal{AS}_1 = (AL_1, AA_1, SV_1)$ und $\mathcal{AS}_2 = (AL_2, AA_2, SV_2)$ heißen *kompatibel* genau dann, wenn gilt:

$AL_1 = AL_2 \wedge AA_1 = AA_2$

Das bedeutet, dass beide Sichten in denselben Auflösungen formuliert sein müssen und dieselben Architekturarten darstellen müssen. Die dargestellten Sachverhalte dürfen sich unterscheiden.

## Beispiel für die Verwendung des Schemas

Eine Architektursicht könnte etwa für eine logische Architektur in der Bausteinebene die Datenhaltung gemeinsam mit Informationen zum Zugriffsschutz darstellen, um beides aufeinander abzustimmen und nicht etwa geheime Daten in leicht zugänglichen Systemteilen zu verwalten. Grafisch kann dies, analog zu den Darstellungen von Schlosser [Sch05], über Markierungen in der Abbildung 7.1 geschehen. Die Abbildung 7.3 zeigt ein Beispiel dafür. In der Tupelschreibweise wäre die Darstellung:

$\mathcal{AS}_{Datensicherheit} = (\{Baustein\}, \{Logisch\}, \{Daten, Sicherheit\})$.

[9] Eine ähnliche Darstellung findet sich bei Schlosser [Sch05]

## 7.2 Erzeugung von Architektursichten

Als Ausgangspunkt für die Erzeugung von Sichten wird eine logische Architektur verwendet. Dieser werden Informationen zugeordnet, die in einer Sicht dargestellt werden sollen oder die Erzeugung der Sicht beeinflussen. Die Informationen werden als Attribute den Komponenten und Konnektoren zugeordnet. Die logische Architektur, die als Ausgangspunkt für die Sichtenerzeugung fungiert, wird im Folgenden auch *Kernmodell* genannt.

Sichten entstehen aus dem Kernmodell durch die Projektion. Die Projektionsvorschrift kann als Blickwinkel aufgefasst werden. Die dazu notwendigen grundlegenden Projektionen werden in Kapitel 6 auf der Grundlage der Graphentheorie eingeführt. Komplexe Projektionen können aus den grundlegenden Projektionen zusammengesetzt werden. Die (verknüpften) Projektionen dienen als formale Beschreibung eines Blickwinkels[10]. Vier grundlegende Typen von Sichten werden erzeugt:

**Architektursichten mit Zusatzinformationen (Sachverhalten)** stellen die logische Architektur vollständig dar, sie zeigen zusätzlich eine Teilmenge der Attribute, die den Komponenten und den Konnektoren zugeordnet sind. Über die Attribute werden den Architekturelementen Sachverhalte zugewiesen. Beispielsweise kann die Struktur des IT-Systems zusammen mit Planungsinformationen visualisiert werden, oder für jede Komponente werden die zuständigen Teilteams dargestellt und der Architekt erhält damit eine *Karte der Zuständigkeiten*. Diese Sichten gehören zum Allocation Viewtype nach Clements et al. [CBB+03].

**Vergröbernde Sichten** nutzen Informationen über den hierarchischen Aufbau des IT-Systems. Sie stellen verschiedene Detaillierungsebenen (Auflösungen) dar. In Abschnitt 7.1 werden beispielsweise drei grundlegende Auflösungen betrachtet, das sind Organisationsebene, Systemebene und Bausteinebene. Generiert werden Sichten der System- und der Bausteinebene. Diese werden in der Literatur gefordert, vgl. etwa [CBB+03, S.195ff].

**Zusammenfassende Sichten** fassen Komponenten und Konnektoren einer Architektur nach vorgegebenen Kriterien zusammen. Die von Siedersleben beschriebene T-Architektur-Sicht[11] [Sie04, S. 151ff] fasst beispielsweise Komponenten mit derselben Aufgabe zusammen und stellt die Kommunikationsbeziehungen dieser Komponententypen dar.

**Auswählende Sichten** zeigen einen Ausschnitt aus der Architektur. Dieser Ausschnitt wird über Auswahloperationen (vgl. Abschnitt 6.4) erzeugt. Die A-Architektur-Sicht [Sie04, S. 151ff] kann den auswählenden Sichten zugeordnet werden. Sie stellt nur fachliche Komponenten dar und blendet technische Komponenten aus. Für die Test- und Integrationsplanung kann es beispielsweise wichtig sein, nur die Komponenten einer Architektur darzustellen, die zu einem gegebenen Zeitpunkt fertiggestellt sind, diese Zeitpunktsbetrachtung ist ebenfalls eine auswählende Sicht.

Die folgenden Abschnitte demonstrieren die Anwendbarkeit der Architekturtheorie und stellen dar, wie die verschiedenen Typen von Sichten über Projektionen erzeugt werden. Die vorgestellten Sichten sind keineswegs eine umfassende Darstellung aller denkbaren Architektursichten, es handelt sich um Vorschläge und um häufig verwendete Sichten aus der Literatur. Die Projektionen werden jeweils durch die Formelschreibweise der Projektionen definiert und durch ein Beispiel illustriert.

### Einordnung der Sichten und Blickwinkel

Die im Folgenden präsentierten Blickwinkel werden aus der Beschreibung einer logischen Architektur erzeugt. Den Strukturinformationen der logischen Architektur können beliebige Sachverhalte über Attribute zugewiesen werden. Einer logischen Komponente oder einem Konnektor können etwa folgende Sachverhalte zugeordnet werden:

[10] vgl. Abschnitt 2.3.4
[11] vgl. Abschnitt 3.4.5

- Die Entitäten eines logischen Datenmodells werden den Komponenten zugeordnet, die für diese zuständig sind[12] (Sachverhalt *Daten*).
- Nutzergruppen aus einer Organisation werden den Komponenten und Konnektoren zugewiesen, wenn sie diese nutzen dürfen (Sachverhalt *Security, Zugriffsschutz*).
- Daten aus der Projektplanung wie zuständiges Team, Fertigstellungstermin und geplanter Aufwand werden Komponenten und Konnektoren zugeordnet (Sachverhalt *Planung*).

Die vorgestellten Blickwinkel und die damit erzeugten Sichten befinden sich unterhalb der Systemebene. Die gröbste mögliche Sicht stellt genau die Systemebene dar, also das System als Black-Box zusammen mit Nutzergruppen und Nachbarsystemen. Die Ursache für diese Beschränkung liegt in der Definition der logischen Systemarchitekturen. In der ersten Ebene der Komponenten-Hierarchie befindet sich eine Komponente $\sigma$, die das IT-System darstellt und nur die Nachbarsysteme der direkten Umgebung werden in der logischen Systemarchitektur modelliert.

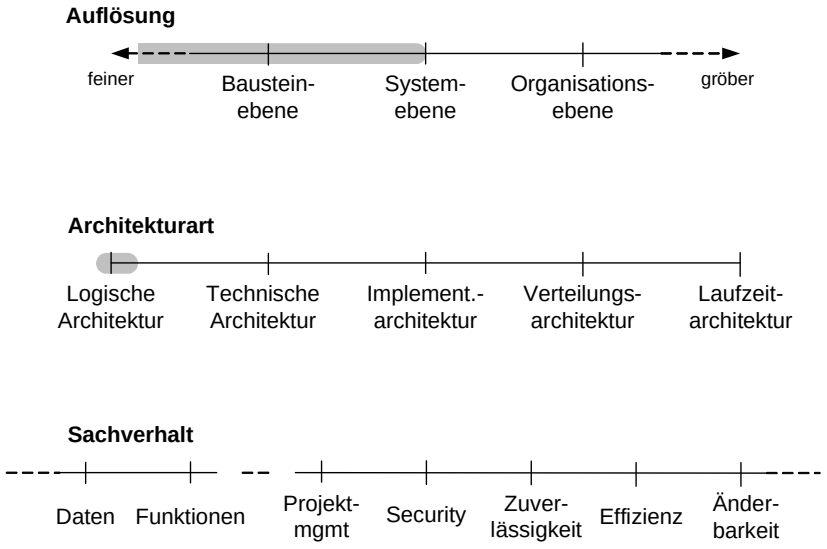


Abbildung 7.3: Einordnung der Sichten logischer Architekturen

Feinteiligere Sichten auf der Bausteinebene (erste Dekompositionsebene des IT-Systems) und darunter sind im Rahmen der Theorie bis zu einer beliebigen Detaillierung möglich.

Darstellungen auf der Organisationsebene würden alle IT-Systeme eines Unternehmens darstellen. Dies geht über das Thema Software- und Systemarchitekturen hinaus und fällt in den Bereich des Themenfeldes Anwendungslandschaften [MW04].

Die im Folgenden präsentierten Sichten unterscheiden sich in ihrer Ausrichtung von den Sichtenmodellen, wie sie in Kapitel 3.4 dargestellt wurden: Während sich die Sichtenkonzepte typischerweise entlang der Architekturarten bewegen und etwa logische Architekturen von Implementierungsarchitekturen unterscheiden, bewegen sich die hier vorgestellten Sichten innerhalb einer Architekturart – der logischen Architektur – und unterscheiden sich in Auflösung und Sachverhalten. Die hier vorgestellten Sichten werden in Abbildung 7.3 grafisch eingeordnet. Die vorgestellten Sichten dienen zur Qualitätssicherung sowie dem Projektmanagement und der Verbesserung der Kommunikation im Entwicklungsteam und mit den Auftraggebern.

Die Abbildungen in diesem Kapitel sind manuell erstellt. Das Werkzeug AutoARCHITECT kann die angegebenen Sichten ebenfalls erzeugen. Jedoch ist durch den allgemeinen Layout-Algorithmus des

[12] vgl. z.B. [HS00, Oes01]

Visualisierungswerkzeugs die grafische Positionierung der Komponenten noch verbesserungsbedürftig. In Kapitel 9 werden Beispiele für Architektursichten gegeben, die AutoARCHITECT aus AutoFOCUS 2 Beschreibungen generiert hat. Typischerweise werden derartige Darstellungen vom Architekten nachbearbeitet, sodass bereits der vorhandene Funktionsumfang zur automatischen Layoutung hilfreich ist.

## Bringdienstsystem als Beispiel

Als Ausgangspunkt für die Beispiele in den folgenden Abschnitten, dient die schon mehrfach in der vorliegenden Arbeit gezeigte logische Systemarchitektur des Bringdienstsystems $\mathcal{LS} = (BD, A_{BD}, value_{BD}, UC_{BD}, S_{BD})$. Diese wird in Abbildung 7.4 dargestellt. Formal wird sie als Konfiguration $SK$ in Abschnitt 5.3 beschrieben.

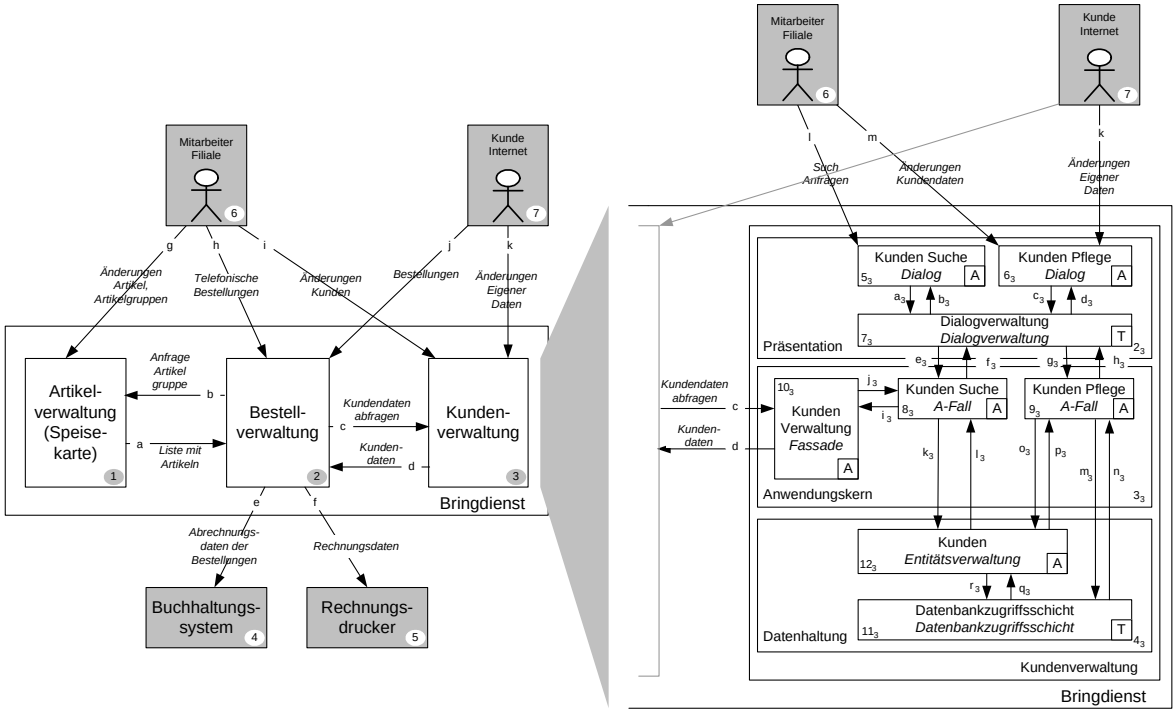


Abbildung 7.4: Logische Architektur des Bringdienstsystems

Die Kundenverwaltung (Abbildung 3.4, rechts) dient als Grundlage für verschiedene Sichten, bereits im vorangegangenen Kapitel. Sie wird in Abschnitt 6.1 graphentheoretisch definiert.

### Konfiguration

Die Konfiguration aus Abbildung 7.4 des Bringdienstes $BD$ mit der verfeinerten Komponente *Kundenverwaltung* kann beschrieben werden als $BD = (V_{BD}, E_{BD}, J_{BD}, H_{BD})$ (BD für Bringdienst):

- $V_{BD} = \{\omega, \sigma, 1, 2, 3(= 1_3), 4, 5, 6, 7, 2_3, 3_3, 4_3, 5_3, 6_3, 7_3, 8_3, 9_3, 10_3, 11_3, 12_3\}$
- $E_{BD} = \{a, b, c, d, e, f, g, h, j, l, m, a_3, b_3, c_3, d_3, e_3, f_3, g_3, h_3, i_3, j_3, k_3, l_3, m_3, n_3, o_3, p_3, q_3, r_3\}$

- $J_{BD} = \{(a,(1,2)),(b,(2,1)),(c,(2,10_3)),(d,(10_3,2)),(e,(2,4)),(f,(2,5)),(g,(6,1)),$
  $(h,(6,2)),(l,(6,5_3)),(m,(6,6_3)),(j,(7,2)),(k,(7,6_3)),(a_3,(5_3,7_3)),(b_3,(7_3,5_3)),$
  $(c_3,(6_3,7_3)),(d_3,(7_3,6_3)),(e_3,(7_3,8_3)),(f_3,(8_3,7_3)),(g_3,(7_3,9_3)),(h_3,(9_3,7_3)),$
  $(i_3,(8_3,10_3)),(j_3,(10_3,8_3)),(k_3,(8_3,12_3)),(l_3,(12_3,8_3)),(m_3,(9_3,11_3)),(n_3,(11_3,9_3)),$
  $(o_3,(9_3,12_3)),(p_3,(12_3,9_3)),(q_3,(11_3,12_3)),(r_3,(12_3,11_3))\}$

- $H_{BD} = \{(\omega,4),(\omega,5),(\omega,6),(\omega,7),(\omega,\sigma),(\sigma,1),(\sigma,2),(\sigma,3),(1_3,2_3),(1_3,3_3),(1_3,4_3),(2_3,5_3),$
  $(2_3,6_3),(2_3,7_3),(3_3,8_3),(3_3,9_3),(3_3,10_3),(4_3,11_3),(4_3,12_3)\}$

Die Komponenten und Konnektoren der Komponente *Kundenverwaltung* haben den Index 3, da sie Teile der Komponente Kundenverwaltung mit der ID 3 sind.

#### Anwendungsfälle und Szenarios

Die folgenden drei Anwendungsfälle $UC_{BD} := \{uc_1, uc_2, uc_3\}$ werden der Konfiguration $BD$ zugeordnet. Es sind dieselben Anwendungsfälle wie bei der Kundenverwaltung in Abschnitt 6.1. Die zugeordneten Nutzungsszenarios wurden um die Nutzergruppen und Nachbarsysteme erweitert.

1. Ein Mitarbeiter in der Filiale sucht einen Kunden in der Kundenverwaltung über einen Suchdialog, über den mit verschiedenen Parametern (Telefonnummer, Adresse, Name, etc.) gesucht werden kann:

   $S_{BD}(uc_1) := \{l, a_3, e_3, k_3, r_3, q_3, l_3, f_3, b_3\}$

2. Ein Kunde im Internet ändert seine eigenen Daten, er gibt dazu seine Telefonnummer ein, danach ändert er seinen Datensatz, hierzu wird zweimal folgendes Szenario ausgeführt (1. Datensatz holen, 2. geänderten Datensatz zurück schreiben):

   $S_{BD}(uc_2) := \{k, c_3, g_3, o_3, r_3, q_3, p_3, h_3, d_3\}$

3. Die Komponente *Bestellverwaltung* benötigt die Daten zu einem Kunden:

   $S_{BD}(uc_3) := \{c, j_3, k_3, r_3, q_3, l_3, i_3\}$

#### Attribute

Die Attributmenge der logischen Systemarchitektur des Bringdienstes lautet:

$A_{BD} := \{\ Software-Kategorie, Aufgabe, Plan.Anfang, Plan.Ende, Plan.Aufwand,$
$Plan.Verantwortlich\}$

Den Komponenten des Bringdienstes werden die Software-Kategorien [Sie04] A- und T-Software zugeordnet. Die Software-Kategorie A-Software wird allen Komponenten zugeordnet, die direkt zur fachlichen Funktionalität des Bringdienstes beitragen, also zur Erfüllung der funktionalen Anforderungen. Das sind beispielsweise die Komponenten *Kunden-Pflege-Dialog* und *Kunden-Pflege-A-Fall*. Die Software-Kategorie T-Software wird allen Komponenten zugeordnet, die (allgemeine) technische Aufgaben erfüllen, wie etwa die *Dialogsteuerung* oder die *Datenbankzugriffsschicht*. Den Komponenten ist zusätzlich eine Aufgabe zugeordnet, welche die Komponenten der jeweiligen Kategorien spezialisiert. So ist ein Dialog beispielsweise für die grafische Darstellung und die Interaktionen mit einem Benutzer zuständig, während ein Entitätsverwalter für die Verwaltung von Entitäten (Kunden, Artikel, Bestellungen)zuständig ist.

Die Belegung $value_{BD}$ der Attribute innerhalb des Bringdienstes ist gegeben durch die Tabelle 7.1.

Allen Konnektoren wird als Software-Kategorie *0-Software* und als Aufgabe *neutral* zugeordnet. Dies sind die Null-Werte der entsprechenden Attributtypen.

| Id | Name | Software-Kategorie | Aufgabe |
|---|---|---|---|
| $\omega$ | Umgebung | $\langle\rangle$ | $\langle\rangle$ |
| 4 | Buchhaltungssystem | $\langle\rangle$ | $\langle\rangle$ |
| 5 | Rechnungsdrucker | $\langle\rangle$ | $\langle\rangle$ |
| 6 | Mitarbeiter Filiale | $\langle\rangle$ | $\langle\rangle$ |
| 7 | Kunde Internet | $\langle\rangle$ | $\langle\rangle$ |
| $\sigma$ | Bringdienst | 0-Software | $\langle\rangle$ |
| $1(1_1)$ | Artikelverwaltung | 0-Software | Subsystem |
| $2(1_2)$ | Bestellverwaltung | 0-Software | Subsystem |
| $3(1_3)$ | Kundenverwaltung | 0-Software | Subsystem |
| $2_3$ | Dialogschicht | 0-Software | Präsentation |
| $3_3$ | Anwendungskern | 0-Software | Anwendungskern |
| $4_3$ | Datenhaltung | 0-Software | Datenhaltung |
| $5_3$ | KundenSuche Dialog | A-Software | Dialog |
| $6_3$ | KundenPflege Dialog | A-Software | Dialog |
| $7_3$ | Dialogverwalter | T-Software | Dialogverwalter |
| $8_3$ | KundenSuche A-Fall | A-Software | A-Fall |
| $9_3$ | KundenPflege A-Fall | A-Software | A-Fall |
| $10_3$ | KundenVerwaltung Fassade | A-Software | Fassade |
| $11_3$ | DB Zugriffsschicht | T-Software | Zugriffsschicht |
| $12_3$ | Kunden Entitätsverwalter | A-Software | Entitätsverwalter |

Tabelle 7.1: Belegungen für Software-Kategorie und Aufgabe zu den Komponenten des Bringdienstsystems

Der logischen Architektur ist ein Terminplan zugeordnet, der für jede Komponente und mehrere Konnektoren Termine, verantwortliche Teams und den Entwicklungsaufwand enthält. Der Terminplan wird in der Tabelle 7.2 dargestellt. Mit Projektmanagementwerkzeugen kann aus der Tabelle leicht eine grafische Darstellung, etwa ein Netzplan oder ein Gantt-Diagramm gewonnen werden.

Wie die Daten Terminplans auf die Elemente einer logischen Architektur abgebildet werden, wird in Kapitel 8 dargestellt. Die Tabellen 7.2 und 7.3 stellen das Ergebnis der Abbildung dar.

| Id | Name | Plan Anfang | Plan Ende | Plan Aufwand | Plan verantw. |
|---|---|---|---|---|---|
| $\omega$ | Umgebung | 15.03.07 | 19.04.07 | 25 PT | Team München |
| 4 | Buchhaltungssystem | 16.10.06 | 27.10.06 | 10 PT | Firma CO Ltd. |
| 5 | Rechnungsdrucker | MAXDATUM | MINDATUM | 0 PT | $\emptyset$ |
| 6 | Mitarbeiter Filiale | MAXDATUM | MINDATUM | 0 PT | $\emptyset$ |
| 7 | Kunde internet | MAXDATUM | MINDATUM | 0 PT | $\emptyset$ |
| $\sigma$ | Bringdienst | 29.01.07 | 15.03.07 | 100 PT | Team München |
| $1(1_1)$ | Artikelverwaltung | 02.10.06 | 04.12.06 | 230 PT | Team Stuttgart |
| $2(1_2)$ | Bestellverwaltung | 23.10.06 | 08.01.07 | 280 PT | Team München |
| $3(1_3)$ | Kundenverwaltung | 04.12.06 | 08.12.06 | 8 PT | Team Sofia |
| $2_3$ | Dialogschicht | MAXDATUM | MINDATUM | 0 PT | $\emptyset$ |
| $3_3$ | Anwendungskern | MAXDATUM | MINDATUM | 0 PT | $\emptyset$ |
| $4_3$ | Datenhaltung | MAXDATUM | MINDATUM | 0 PT | $\emptyset$ |
| $5_3$ | KundenSuche Dialog | 09.10.06 | 17.11.06 | 90 PT | Team Sofia |
| $6_3$ | KundenPflege Dialog | 09.10.06 | 01.12.06 | 130 PT | Team Sofia |
| $7_3$ | Dialogverwalter | 02.10.06 | 13.10.06 | 20 PT | Team Sofia |
| $8_3$ | KundenSuche A-Fall | 30.10.06 | 10.11.06 | 30 PT | Team Sofia |
| $9_3$ | KundenPflege A-Fall | 13.11.06 | 01.12.06 | 50 PT | Team Sofia |
| $10_3$ | KundenVerwaltung Fassade | 27.11.06 | 01.12.06 | 12 PT | Team Sofia |
| $11_3$ | DB Zugriffsschicht | 02.10.06 | 13.10.06 | 20 PT | Team Sofia |
| $12_3$ | Kunden Entitätsverwalter | 09.10.06 | 27.10.06 | 40 PT | Team Sofia |

Tabelle 7.2: Belegung der Planungsattribute für die Komponenten des Bringdienstsystems

Zusätzlich werden zwei Konnektoren Daten aus der Planung zugewiesen. Die Arbeitsaufwände und Zeitpunkte sind Aufwände zur Abstimmung der entsprechenden Schnittstellen zwischen dem Bringdienst und den beiden Nachbarsystemen.

| Id | Name | Plan Anfang | Plan Ende | Plan Aufwand | Plan verantw. |
|---|---|---|---|---|---|
| e | Schnittstelle zur Buchhaltung | 30.10.06 | 08.12.06 | 15 PT | Team München |
| f | Schnittstelle zum Rechnungsdrucker | 08.01.07 | 23.02.07 | 35 PT | Team München |
| sonst | ″ | MAXDATUM | MINDATUM | 0 PT | ∅ |

Tabelle 7.3: Belegung der Planungsattribute bei den Konnektoren des Bringdienstsystems

## 7.3 Architektursichten mit Zusatzinformationen

Architektursichten mit Zusatzinformationen zeigen die Strukturinformationen aus der Konfiguration $C$ einer logischen Architektur $\mathcal{L}$ zusammen mit zusätzlichen Sachverhalten. Die Konfiguration $C$ fungiert als Grundlage der Darstellung, darauf werden verschiedene Informationen projiziert, beispielsweise aus der Planung oder andere in Abbildung 7.3 dargestellte Informationen wie Entitäten des logischen Datenmodells, für welche die Komponenten zuständig sind oder Organisationseinheiten, die Komponenten nutzen dürfen.

Zur Definition einer attributbasierten Sicht werden Prädikate $\mathcal{Q}_A : A \rightarrow \mathbb{B}$ auf der Menge der Attribute $A$ der logischen Architektur $\mathcal{L} = (C, A, value, UC, S)$ verwendet mit $\mathcal{Q}_A : A \rightarrow \mathbb{B}$. Diese bestimmen die Attribute, welche für die jeweilige Sicht relevant sind.

$\mathcal{L}_{Attributbasiert} := \Xi_A(\mathcal{L}, \mathcal{Q}_A)$

Bei der Abbildung bleibt die Konfiguration der logischen Architektur erhalten, lediglich Attribute und ihre Werte werden entfernt. Das Prädikat $\mathcal{Q}_A$ wird auf die Attributmenge $A$ der logischen Architektur $\mathcal{L}$ angewendet.

$$\mathcal{Q}_A(a) := \begin{cases} true & : a \in NochZuBestimmendeAttributmenge \\ false & : sonst \end{cases}$$

Die Sicht $\mathcal{L}_{Attributbasiert}$ (= Bild) hat weniger Attribute als die logische Architektur $\mathcal{L}$ (= Urbild), von der ausgegangen wird. Darstellungen der logischen Architektur $\mathcal{L}_{Attributbasiert}$ sind damit übersichtlicher und fokussierter, da sie nicht mit allen (denkbaren) Informationen ihres Urbildes überfrachtet sind.

### 7.3.1 Planungssicht

In der Regel wird die Aufteilung eines Projekts in Arbeitspakete in Projektstrukturplänen [DIN87c, pro06, Bur02] oder in Varianten von Netzplänen [Bur02] festgehalten. Projektstrukturpläne stellen die hierarchische Strukturierung eines Projekts in Arbeitspakete dar, jedoch ohne deren Abhängigkeiten auszuführen. Netzpläne stellen die zeitlichen Abhängigkeiten dar. In beiden Darstellungsvarianten fehlt der direkte Bezug zur (logischen) Architektur eines IT-Systems. Wird ein solcher Bezug hergestellt, hat das positiven Einfluss auf die Kommunikation mit dem Auftraggeber und im Entwicklungsteam: Dem Auftraggeber können Aufwände und Termine für einzelne Systembestandteile transparent gemacht werden und die Entwickler sehen grafisch, welches Teilteam für welche Komponente verantwortlich ist. Weiterhin können aus der Architektur aus technischen Abhängigkeiten[13] oder zeitliche Abhängigkeiten abgeleitet werden. Diese Zusammenhänge und Vorteile werden im nachfolgenden Kapitel diskutiert.

Die hier dargestellte Planungssicht $\mathcal{AS}$ = {{Baustein},{Logisch}, {Projektmanagement}) ist ein Vorschlag, wie Informationen aus Projektplänen und der Projektverfolgung auf die logische Architektur $\mathcal{L} = (C, A, value, UC, S)$ abgebildet werden könnten. Die Tabelle 7.4 gibt einen Überblick über die Attribute, welche für die Darstellung der Planung notwendig sind und die Rechenoperationen, die für spätere Vergröberungs- und Zusammenfassungsprojektionen benötigt werden. Dies sind dieselben Attribute, welche auch Vorgänge eines Terminplans oder die Arbeitspakete eines Projektstruk-

[13] Wenn zwei Komponenten miteinander kommunizieren, sind sie möglicherweise von einander abhängig.

| Attributtyp | Attributname $name$ | $+_{name}$ | $\leq_{name}$ | $0_{name}$ |
|---|---|---|---|---|
| Datum | Plan.Anfang | $min$ | $\geq$ | MAXDATUM |
| Datum | Plan.Ende | $max$ | $\leq$ | MINDATUM |
| Personentage | Plan.Aufwand | $+$ | $\leq$ | 0 PT |
| Personenmenge | Plan.Verantwortlich | $\cup$ | $\subseteq$ | $\emptyset$ |

Tabelle 7.4: Planungsdaten, die Komponenten und Konnektoren zugeordnet werden können (Ausschnitt aus Tabelle 8.1).

turplans haben. Ein Vorschlag, wie diese Attribute aus der Planung übernommen werden können, ist in Abschnitt 8.3.2 dargestellt.

Die Planungssicht $\mathcal{L}_{Planung}$ der logischen Architektur zeigt ausschließlich Planungsdaten, wie sie im Projekt erfasst werden. Dazu wird ein Auswahlprädikat $\mathcal{Q}_{APlan} : A \rightarrow \mathbb{B}$ auf der Menge der Attribute $A$ einer logischen Architektur definiert, $\forall a \in A$:

$$\mathcal{Q}_{APlan}(a) := \begin{cases} true & : a \in \{Plan.Anfang, Plan.Ende, Plan.Aufwand, Plan.Verantwortlich\} \\ false & : sonst \end{cases}$$

Das Auswahlprädikat $\mathcal{Q}_{APlan}$ filtert nur die Attribute heraus, die Planungsdaten darstellen. Die Architekturdarstellung wird damit erreicht über eine Auswahlprojektion:

$\mathcal{L}_{Planung} := \Xi_A(\mathcal{L}, \mathcal{Q}_{APlan})$.

In Abbildung 7.5 ist ein Beispiel für die Erzeugung der Planungssicht dargestellt. Die Projektion hat die Attribute *Software-Kategorie* und *Aufgabe* aus der logischen Architektur entfernt. Für jede Komponente ist direkt ablesbar, wann sie fertiggestellt ist, wie viel Aufwand die Erstellung kostet und welches Teilteam für die Erstellung verantwortlich ist.

Die Planungssicht kann über Vergröberungs- oder Auswahlprojektionen vereinfacht und modifiziert werden. Die dazu notwendigen Rechenoperationen sind in der Tabelle 7.4 dargestellt.

Die zur Kundenverwaltung dargestellten Belegungen der Attribute sind durch die Summation der Belegungen der Teilkomponenten entstanden. Um diese vergröberte Darstellung zu gewinnen, muss zusätzlich eine Vergröberungsprojektion durchgeführt werden, welche die logische Architektur auf die zweite Ebene vergröbert und so alle Teilkomponenten der Kundenverwaltung ausblendet. Die Abbildung 7.5 zeigt damit eigentlich die Projektion:

$\mathcal{L}_{PlanungBausteinebene} := \Phi_{\mathbb{N}}(\Xi_A(\mathcal{L}, \mathcal{Q}_{APlan}), 2)$.

Die Summation der Belegung für die Komponente Kundenverwaltung wird als Beispiel vorgeführt. Die Berechnungsvorschrift dazu findet sich in Abschnitt 6.2. Das Ergebnis der Summation ist bereits in Abbildung 7.5 dargestellt. Die nachfolgenden Gleichungen stellen die Summationsoperatoren $\sum^a$ und $+_a$ für den Attributtyp $a$ vereinfachend mit $\sum$ und $+$ dar.

Für das Attribut $Plan.Anfang$ ist $min$ der Summenoperator und $MAXDATUM$ das neutrale Element. Damit wird der Beginn für die vergröberte Kundenverwaltung aus dem Minimum aller Anfangstermine berechnet. Das Attribut $Plan.Anfang$, das der Komponente Kundenverwaltung $(1_3)$in der Tabelle 7.2 zugeordnet ist, ist der Anfangstermin für die Integration der Kundenverwaltung. Eine Begründung ist in Abschnitt 8.3.1 zu finden. Da $1_3 \in leafs(C_{PlanungBausteinebene})$, gilt:

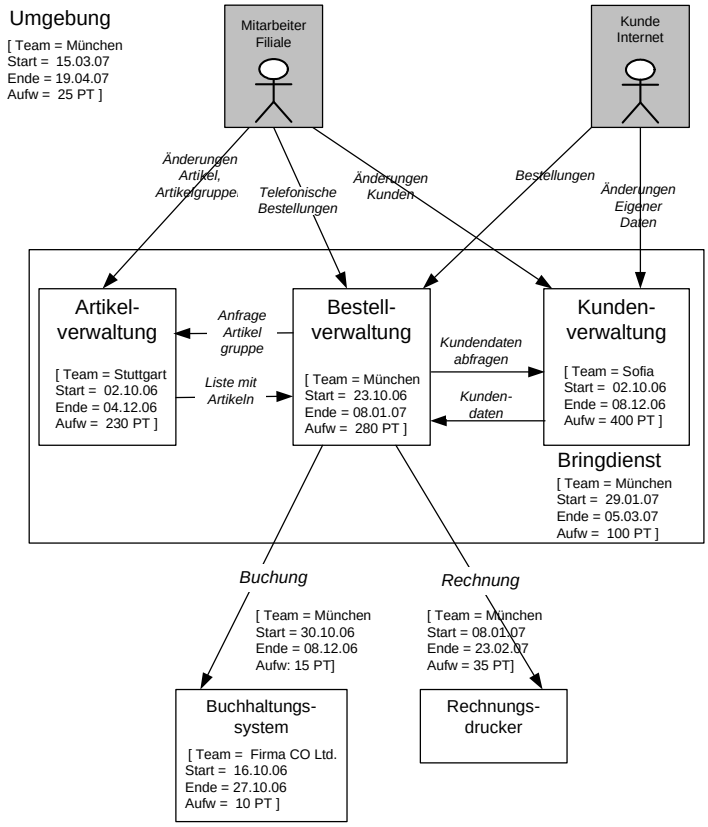


Abbildung 7.5: Darstellung der Planungssicht des Bringdienstsystems

$value_{PlanungBausteinebene}(1_3, Plan.Anfang)$

$= value(1_3, Plan.Anfang) +$
$\sum_{w \in allparts(BD,1_3)}(value(w, Plan.Anfang)) +$
$\sum_{f \in internal(BD,allparts(BD,1_3))}(value(f, Plan.Anfang))$

$= min(value(1_3, Plan.Anfang), min($
$min_{w \in \{2_3,3_3,4_3,5_3,6_3,7_3,8_3,9_3,10_3,11_3,12_3\}}(value(w, Plan.Anfang)),$
$min_{f \in internal(BD,\{2_3,3_3,4_3,5_3,6_3,7_3,8_3,9_3,10_3,11_3,12_3\})}(value(f, Plan.Anfang))))$

$= min(04.12.06, min($
$min(MAXDATUM, MAXDATUM, MAXDATUM, 09.10.06, 09.10.06, 02.10.06,$
$30.10.06, 13.11.06, 27.11.06, 02.10.06, 09.10.06),$
$min(MAXDATUM, \ldots, MAXDATUM)))$

$= min(04.12.06, min(02.10.06, MAXDATUM)) = min(04.12.06, 02.10.06)$

$= 02.10.06$

Für das Attribut $Plan.Ende$ ist $max$ der Summenoperator und $MINDATUM$ das neutrale Element. Der Endtermin für die Kundenverwaltung ist das Maximum der Endtermine aller Teilkomponenten. Die Kundenverwaltung ist erst dann fertiggestellt, wenn alle Teilkomponenten fertig sind und die Kundenverwaltung aus diesen integriert wurde.

$value_{PlanungBausteinebene}(1_3, Plan.Ende) =$

$$
\begin{aligned}
= \quad & value(1_3, Plan.Ende)+ \\
& \textstyle\sum_{w \in allparts(BD,1_3)}(value(w, Plan.Ende))+ \\
& \textstyle\sum_{f \in internal(BD,allparts(BD,1_3))}(value(f, Plan.Ende))
\end{aligned}
$$

$$
\begin{aligned}
= \quad & max(value(1_3, Plan.Ende), max( \\
& max_{w \in \{2_3,3_3,4_3,5_3,6_3,7_3,8_3,9_3,10_3,11_3,12_3\}}(value(w, Plan.Ende)), \\
& max_{f \in internal(BD,\{2_3,3_3,4_3,5_3,6_3,7_3,8_3,9_3,10_3,11_3,12_3\})}(value(f, Plan.Ende))))
\end{aligned}
$$

$$
\begin{aligned}
= \quad & max(08.12.06, max( \\
& max(MINDATUM, MINDATUM, MINDATUM, 17.11.06, 01.12.06, 13.10.06, \\
& 10.11.06, 01.12.06, 01.12.06, 13.10.06, 27.10.06), \\
& min(MINDATUM, \ldots, MINDATUM)))
\end{aligned}
$$

$$= \quad max(08.12.06, max(01.12.06, MINDATUM)) = max(08.12.06, 01.12.06)$$

$$= \quad 08.12.06$$

Die Aufwände für alle Teilkomponenten und Konnektoren werden summiert, daraus ergibt sich der Gesamtaufwand für die Kundenverwaltung.

$value_{PlanungBausteinebene}(1_3, Plan.Aufwand) =$

$$
\begin{aligned}
= \quad & value(1_3, Plan.Aufwand)+ \\
& \textstyle\sum_{w \in allparts(BD,1_3)}(value(w, Plan.Aufwand))+ \\
& \textstyle\sum_{f \in internal(BD,allparts(BD,1_3))}(value(f, Plan.Aufwand))
\end{aligned}
$$

$$
\begin{aligned}
= \quad & value(1_3, Plan.Aufwand)+ \\
& \textstyle\sum_{w \in \{2_3,3_3,4_3,5_3,6_3,7_3,8_3,9_3,10_3,11_3,12_3\}}(value(w, Plan.Aufwand))+ \\
& \textstyle\sum_{f \in internal(BD,\{2_3,3_3,4_3,5_3,6_3,7_3,8_3,9_3,10_3,11_3,12_3\})}(value(f, Plan.Aufwand))))
\end{aligned}
$$

$$
\begin{aligned}
= \quad & 8PT+ \\
& 90PT + 130PT + 20PT + 30PT + 50PT + 12PT + 20PT + 40PT+ \\
& 0PT
\end{aligned}
$$

$$= \quad 400PT$$

Analog zur Planungssicht kann eine ähnliche Sicht für das Projekt-Controlling erstellt werden. Diese zeigt die Ist-Daten im Vergleich zu den Daten aus der Planung. Attribute wie *Ist.Anfang, Ist.Ende, Ist.Aufwand, Ist.Verantwortlich* sowie *Ist.Restaufwand* [Sie02b] sind dazu erforderlich.

### 7.3.2 Karte der Zuständigkeiten

Herbsleb und Grinter führen aus: *Developers often reported great difficulty in deciding who to contact at the other site with questions* [HG99a]. Die hier vorgeschlagene Projektplanungssicht kann einen Beitrag leisten, dieses Problem zu lindern, da am Architekturübersichtsbild die Zuständigkeiten für Komponenten und Konnektoren abgelesen werden können. Werden nur die Zuständigkeiten dargestellt, wird dies im Folgenden die *Karte der Zuständigkeiten* genannt[14]. Die Karte der Zuständigkeiten wird über ein Prädikat definiert, das nur für das Attribut $Plan.Verantwortlich$ zutrifft:

$$\mathcal{Q}_{AZustaendigkeit}(a) := \begin{cases} true & : a \in \{Plan.Verantworltich\} \\ false & : sonst \end{cases}$$

[14]Diese Architektursicht könnte etwa in der Projektdokumentation (etwa Projekt-Wiki oder Projekthandbuch) enthalten sein.

In Abbildung 7.6 ist ein Beispiel für eine Karte der Zuständigkeiten dargestellt. Darin wurden die Komponenten der Bausteinebene auf verschiedene Teilteams in Stuttgart, München und Sofia verteilt. Bei Bedarf könnte diese Darstellung in Verfeinerungen der logischen Architektur bis zu einzelnen Entwicklern heruntergebrochen werden.

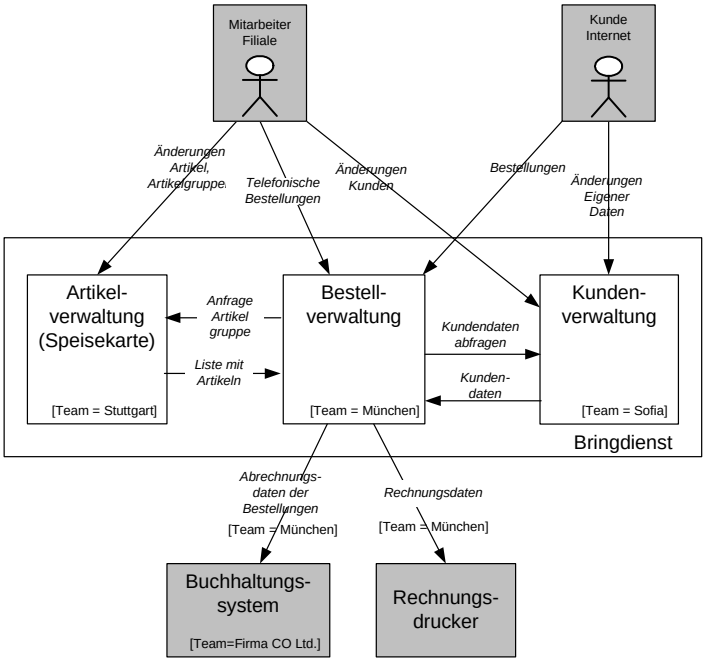


Abbildung 7.6: Darstellung der Zuständigkeiten während der Entwicklung der Bringdienst-Software

Die Karte der Zuständigkeiten kann über differenziertere Verantwortungsbereiche verfeinert werden, etwa durch eine Unterteilung des Attributs *Plan.Verantwortlich* in die aus dem Rollenkonzept eines Vorgehensmodells oder einer Entwicklungsmethodik stammenden Teilverantwortlichkeiten, etwa *Plan.Verantwortlich.Spezifikation*, *Plan.Verantwortlich.Implementierung* oder *Plan.Verantwortlich.QS*. Um die genannten Informationen aus einem Projektplan zu übernehmen, ist jedoch eine differenziertere Abbildung erforderlich, als sie in Kapitel 8 vorgestellt wird.

Zusätzlich kann über die Karte der Zuständigkeiten die Projektorganisation überprüft werden: Wenn für eine (alte) Komponente oder ein Nachbarsystem kein Team und keine externe Firma benannt ist, die dafür zuständig ist, liegt möglicherweise ein Fehler in der Planung oder der Organisation vor. Die Abbildung 7.6 zeigt beispielsweise, dass für das Nachbarsystem Rechnungsdrucker niemand verantwortlich ist.

## 7.4 Vergröbernde Architektursichten

Vergröbernde Sichten vereinfachen die Beschreibung einer logischen Architektur, indem sie Teilkomponenten aus der Beschreibung entfernen und nur noch eine oder wenige Hierarchieebenen gleichzeitig darstellen. Derartige Überblickssichten werden vielfach in der Literatur gefordert und sind in vielen Sichtenkonzepten enthalten.

Zur Steuerung der Vergröberungsprojektion werden Informationen über den hierarchischen Aufbau der logischen Architektur verwendet. Von den Detaillierungsebenen, wie sie in Abschnitt 7.1 vorgestellt werden, können Sichten der Systemebene und der Bausteinebene erzeugt werden. Dazu wird die in Kapitel 6 definierte Vergröberungsprojektion $\Phi_{\mathbb{N}}(\mathcal{L}, n), n \in \mathbb{N}_0$ auf eine logische Architektur $\mathcal{L} = (C, A, value, UC, S)$ angewendet. Über eine natürliche Zahl $n$ wird die Detailebene festgelegt,

bis zu der die Komponenten dargestellt werden sollen, alle anderen Komponenten werden ausgeblendet, die Belegungen ihrer Attribute wird dabei berücksichtigt.

### 7.4.1 Sicht der Systemebene (Kontextdiagramm)

Ein Umgebungsdiagramm stellt dar, wie sich das IT-System $\sigma$ in seine Umgebung aus Nachbarsystemen einbettet und mit diesen kommuniziert. Weiterhin werden die Nutzergruppen dargestellt, die auf das IT-System zugreifen. Diese Diagrammform wird beispielsweise bei Clements et al. als Kontextdiagramm [CBB+03, S195ff] oder bei Starke als Kontextsicht [Sta05, S.97ff] beschrieben.

Sei $\mathcal{LS} = (C, A, value, UC, S)$ eine logische Systemarchitektur, d.h. auf der Ebene 0 des Hierarchiebaumes befindet sich die Pseudokomponente $\omega$ und in der ersten Ebene sind das IT-System $\sigma$, sowie Nachbarsysteme und Nutzergruppen, dann berechnet sich das Umgebungsdiagramm wie folgt:

$\mathcal{L}_{Systemebene} := \Phi_{\mathbb{N}}(\mathcal{L}, 1)$

Die Details zu dieser Abbildung sind in Abschnitt 6.2.1 beschrieben, dort wird sie vorgerechnet. Das IT-System wird mit der Vergröberung auf dieser ersten Ebene als Blackbox betrachtet. Durch eine Verknüpfung mit der Vergröberung von Konnektoren kann die Darstellung vereinfacht werden $\Phi_E \circ \Phi_{\mathbb{N}}$:

$\mathcal{L}_{SystemebeneEinfach} := \Phi_E(\Phi_{\mathbb{N}}(\mathcal{L}, 1))$

Die Abbildung 7.7 zeigt das Umgebungsdiagramm zu der in Abbildung 7.4 dargestellten logischen Architektur. Es wird in der Vektorschreibweise charakterisiert mit $\mathcal{AS} = (\{System\}, \{Logisch\}, \{\})$, d.h. eine logische Architektur wird auf der Systemebene dargestellt, ohne zusätzliche Sachverhalte.

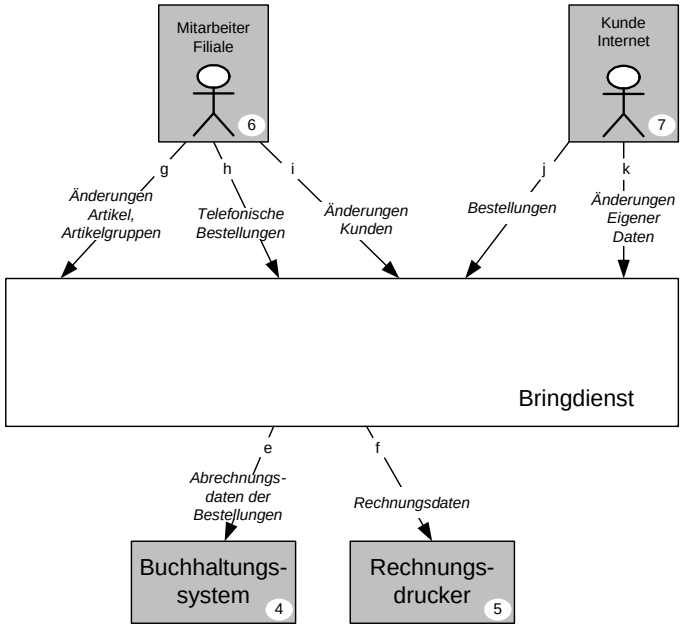


Abbildung 7.7: Darstellung der Systemebene des Bringdienstsystems

### 7.4.2 Sicht der Bausteinebene

Eine Sicht der Bausteinebene stellt die erste Dekompositionsebene des IT-Systems dar. Dies ist eine Glass-Box-Sicht auf das IT-System. Typischerweise werden auf der Bausteinebene die fachlichen Sub-

systeme [BD04, S.612] oder die (technischen) Schichten dargestellt, in die das IT-System strukturiert wird.

$\mathcal{L}_{Bausteinebene} := \Phi_{\mathbb{N}}(\mathcal{L}, 2)$

Nachbarsysteme und Nutzergruppen sind in der Regel in der Glass-Box-Sicht ausgeblendet. Die Ausblendung der Nutzergruppen und Nachbarsysteme findet sich in der nachfolgenden Gleichung. Sei $\mathcal{Q}_V : V \rightarrow \mathbb{B}$ ein einstelliges Prädikat auf der Menge der Komponenten der Konfiguration $C = (V, E, J, H)$ von $\mathcal{L}$, das nur für die Komponenten und Konnektoren des IT-Systems $\sigma$ gültig ist.

$$\mathcal{Q}_V(v) := \begin{cases} true & : v \in (allparts(C, \sigma) \cup \{\omega, \sigma\}) \\ false & : sonst \end{cases}$$

Das Prädikat $\mathcal{Q}_V$ wird für eine Auswahlprojektion $\Xi_V$ verwendet, die nur die Komponenten übernimmt, welche $\mathcal{Q}_V$ erfüllen, also als Komponenten zum Bringdienstsystem gehören. Durch das Entfernen der Nachbarsysteme und Nutzergruppen entfallen auch die dazu gehörenden Konnektoren.

$\mathcal{L}_{BausteinebeneOhneUmgebung} := \Xi_V(\Phi_{\mathbb{N}}(\mathcal{L}, 2), \mathcal{Q}_V)$

Die Abbildung 7.8 zeigt ein Beispiel für die erste Dekompositionsebene des Bringdienstsystems. Die dargestellte Sicht wird im Ordnungsschema über $\mathcal{AS} = (\{Baustein\}, \{Logisch\}, \{\})$ charakterisiert.

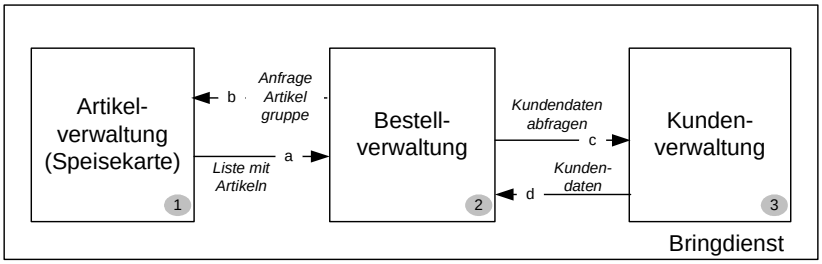


Abbildung 7.8: Darstellung der Bausteinebene des Bringdienstes

## 7.5 Zusammenfassende Architektursichten

### 7.5.1 T-Architektur

In Abschnitt 3.4.5 werden die drei im Quasar-Projekt definierte Sichten A-, T- und TI-Architektur vorgestellt. Die zugehörigen Blickwinkel werden in [Sie04, S.145ff] nur informell anhand von Beispielen beschrieben. Im Folgenden werden die A- und die T-Architektur formal über Projektionen definiert und damit konkretisiert. Bei beiden kann ein Zusammenhang zur logischen Architektur hergestellt werden [Sie04, S.161]. Die TI-Architektur nach [Sie04, S.145ff] entspricht einer Verteilungsarchitektur.

Die vorliegende Arbeit definiert damit erstmalig formal, was eine A-Architektur oder eine T-Architektur tatsächlich ist. Diese Sichten werden aus einer logischen Architektur generiert. Die so erzeugten Sichten können im Rahmen der Qualitätssicherung der logischen Architektur dazu dienen, die vorhandenen Kommunikationsbeziehungen gegen die Vorgaben einer Referenzarchitektur abzugleichen.

Grundlage für die Projektionen sind die Attribute *Software-Kategorie* und *Aufgabe*. Auf diesen werden Auswahlprädikate und Äquivalenzrelationen für die Auswahlprojektion und die Zusammenfassungsprojektion definiert. Die Details zu beiden Attributen und ihren Attributtypen finden sich in Abschnitt 5.5. Die Tabelle 7.5 gibt einen Überblick über die für Quasar-Sichten notwendigen Attribute.

Das Attribut *Software-Kategorie* ist definiert mit den vier Elementen $0$, $A$, $T$ und $AT$. Komponen-

| Attributtyp | Attributname $name$ | $+_{name}$ | $\leq_{name}$ | $0_{name}$ |
|---|---|---|---|---|
| Software-Kategorie | Software-Kategorie | $+$ | $\leq$ | $0 - Software$ |
| Aufgabe | Aufgabe | $+$ | $\leq$ | $neutral$ |

Tabelle 7.5: Attribute zur Erzeugung der A- und T-Architektur

ten, die nur zur Erfüllung der fachlichen (funktionalen) Anforderungen beitragen, haben dabei die Kategorie $A$ (Anwendung), Komponenten, die zur technischen Infrastruktur beitragen, haben die Kategorie $T$ (Technik). Komponenten, die beide Aufgaben erfüllen, haben die Kategorie $AT$. Alle anderen Komponenten haben die Kategorie $0$. Wie sich die Kategorie der Komponenten bei ihrer Komposition verändert, wird in Abschnitt 5.5 dargestellt, dort ist die '+' (Kompositions-)Operation definiert.

$Software - Kategorie :=$ ('Software-Kategorie', $\{0, A, T, AT\}$).

Komponenten haben unterschiedliche Aufgaben, etwa die Verwaltung von Daten oder die grafische Darstellung der Oberfläche inklusive der Interaktion mit den Benutzern. Die möglichen Aufgaben werden durch das Attribut $Aufgabe$ dargestellt.

$Aufgabe :=$ ('Aufgabe', {*Dialog, A-Fall, Entitätsverwalter, Fassade, Zugriffsschicht*, ...}

Die T-Architektur fasst alle Komponenten mit derselben Aufgabe zusammen, insbesondere Komponenten der Kategorie A-Software. Die T-Architektur entsteht daher aus der logischen Architektur durch eine Zusammenfassungsprojektion, bei welcher die Aufgabe die Grundlage für die Definition der Äquivalenzrelation $\mathcal{R} \subseteq V \times V$ auf der Menge der Komponenten einer logischen Architektur darstellt.

$\mathcal{R}_{TArchitektur} := \{(v_1, v_2) | v_1, v_2 \in V \wedge value(v_1, Aufgabe) = value(v_2, Aufgabe)\}$

Damit ergibt sich die T-Architektur über die Zusammenfassung $\Psi$. Diese fasst alle Geschwister-Komponenten zusammen, die nach $\mathcal{R}$ äquivalent sind.

$\mathcal{L}_{TArchitektur} := \Psi(\mathcal{L}, \mathcal{R}_{TArchitektur})$

Die Abbildung 7.9 gibt ein Beispiel für die T-Architektur der Kundenverwaltung aus dem Bringdienstsystem. Dieses Beispiel wird in Abschnitt 6.3 graphentheoretisch dargestellt und die Projektion wird dort vorgeführt. Aus der Abbildung lässt sich der Nutzen der T-Architektur erkennen: Sie stellt dar, welche Arten von Komponenten grundsätzlich miteinander kommunizieren, damit wird die Anbindung der fachlichen A-Komponenten an die T-Komponenten der technischen Infrastruktur und des Trägersystems sichtbar gemacht.

Im Ordnungsschema für Architektursichten muss für die T-Architektur ein neuer Sachverhalt eingeführt werden. Dieser wird hier *Quasar* genannt und enthält alle Informationen, die Bezug zu den Referenzarchitekturen aus dem Quasar-Projekt haben, etwa Software-Kategorie oder Aufgabe. Damit wird die hier beschriebene T-Architektur wie auch die unten dargestellte A-Architektur charakterisiert über: $\mathcal{AS}_{TArchitektur} = (\{Baustein\}, \{Logisch\}, \{Quasar\})$.

Die T-Architektur kann auch in der technischen Architektur oder der Implementierungsarchitektur dargestellt werden [Sie04, S.161]. Eine vollständige Charakterisierung der T-Architektur ist damit: $\mathcal{AS}_{TArchitektur} = (\{Baustein\}, \{Logisch, Technisch, Implementierung\}, \{Quasar\})$.

## 7.6 Auswählende Architektursichten

Die auswahlbasierten Sichten zeigten Ausschnitte aus der Architektur und fassen Komponenten und Konnektoren nach vorgegebenen Regeln zusammen.

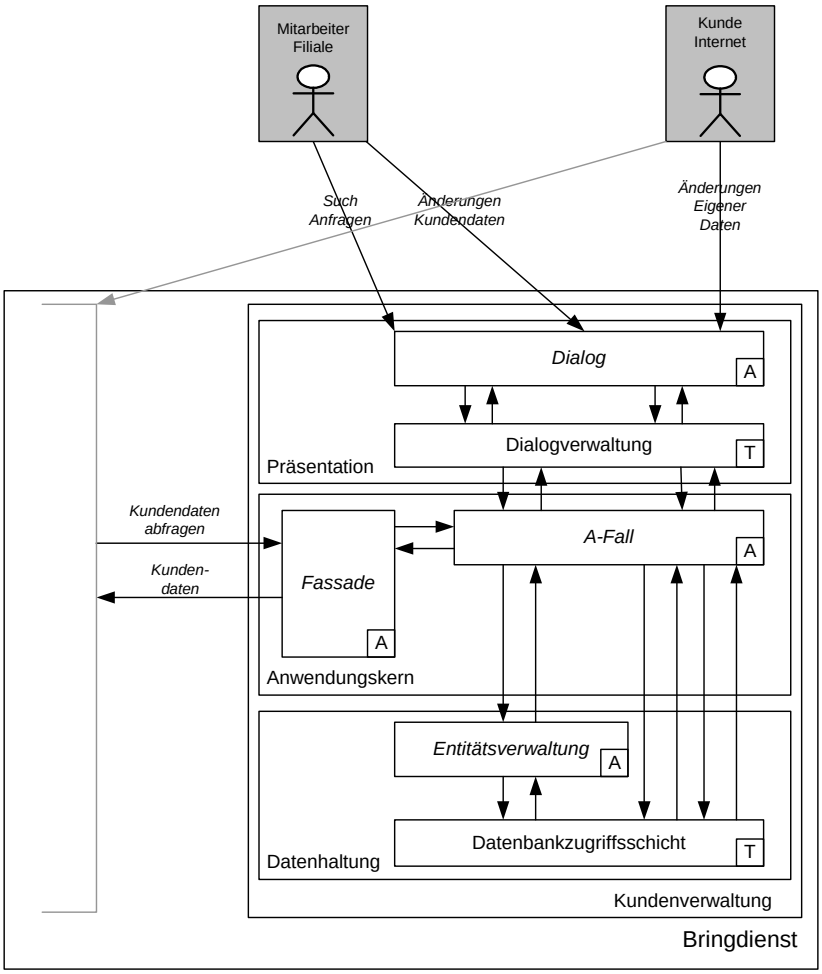


Abbildung 7.9: Darstellung der T-Architektur des Bringdienstsystems

### 7.6.1 A- Architektur

Für die Erzeugung einer aussagekräftigen A-Architektur sind Nutzungsszenarios wichtig, da sie es erlauben, technische Komponenten wie etwa die Dialogverwaltung *heraus zu rechnen*. Darüber kann in der A-Architektur dargestellt werden, welche A-Komponenten tatsächlich über technische Komponenten miteinander kommunizieren.

Die A-Architektur stellt nur A-Software dar. Damit werden nur die Komponenten und Konnektoren ausgewählt, bei denen das Attribut Software-Kategorie den Wert 'A'(-Software) bzw. '0'(-Software) hat. Das Auswahlprädikat $\mathcal{Q}_{A-Software}(v) : V \rightarrow \mathbb{B}$ ist definiert über:

$$\mathcal{Q}_{ASoftware}(v) := \begin{cases} true & : value(v, Software - Kategorie) \in \{A, 0\} \\ false & : sonst \end{cases}$$

Für die Erzeugung der A-Architektur aus der logischen Architektur wird die Auswahloperation $\Xi_{VS}$ verwendet. Darüber soll sichtbar gemacht werden, welche A-Komponenten in der Architektur tatsächlich miteinander kommunizieren. In der logischen Architektur müssen dazu Nutzungsszenarios $S(uc)$ für Anwendungsfälle $uc \in UC$ definiert sein.

$\mathcal{L}_{AArchitektur} := \Xi_{VS}(\mathcal{L}, \mathcal{Q}_{ASoftware})$

Die Abbildung 7.10 gibt ein Beispiel für die A-Architektur der Kundenverwaltung aus dem Bring-

dienstssystem. Für die Erstellung dieser Sicht werden die Szenarios aus dem einführenden Beispiel verwendet. Diese Projektion wird in Abschnitt 6.4 graphentheoretisch vorgeführt.

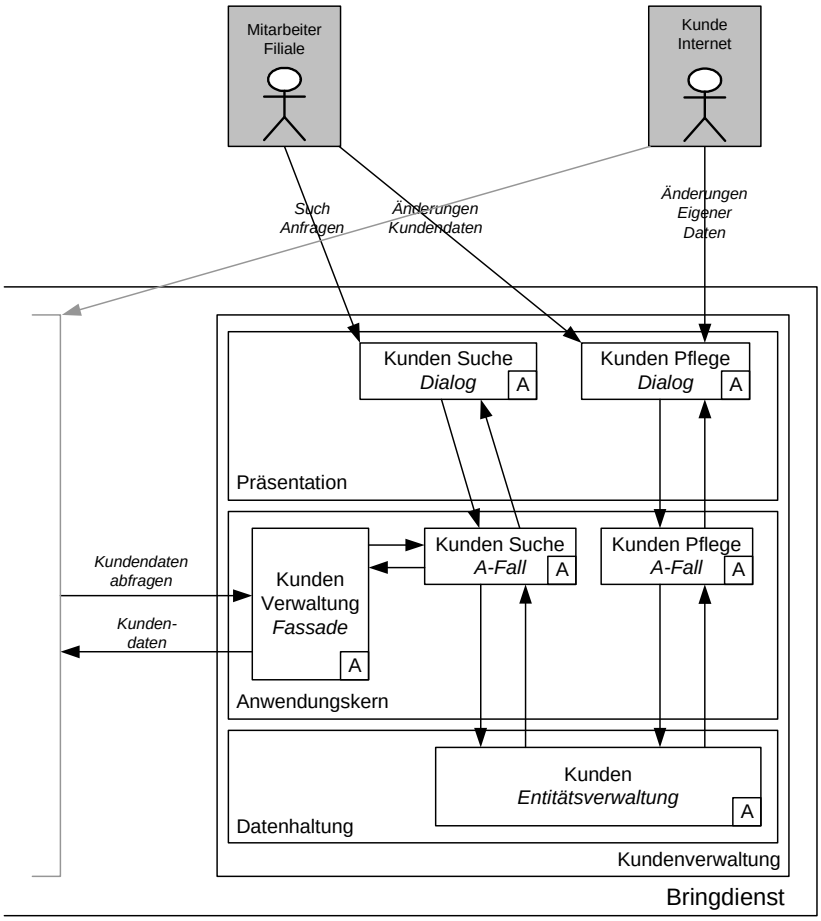


Abbildung 7.10: Darstellung der A-Architektur des Bringdienstsystems

### 7.6.2 Zeitpunktsbetrachtung

Für die Planung der (stufenweisen) Integration komplexer IT-Systeme aus Subsystemen und Komponenten und für die Planung des Integrationstests ist eine Zeitpunktsbetrachtung $\mathcal{AS}_{Zeitpunkt} = (\{Baustein\}, \{Logisch\}, \{Planung\})$ hilfreich: Eine Zeitpunktssicht zeigt alle Komponenten und Konnektoren, die zu einem gegebenen Zeitpunkt während eines Projektes fertiggestellt sind. Mit dieser Darstellung kann sichtbar gemacht werden, wann etwa welche Anwendungsfälle testbar werden: Hierfür müssen alle beteiligten Komponenten und Konnektoren fertiggestellt sein oder für die fehlenden Bestandteile müssen Dummy-Komponenten und -Konnektoren implementiert werden. Für die Planung von Iterationen oder Stufen eines Projekts kann diese Darstellung ebenfalls eingesetzt werden, indem als Zeitpunkt das jeweilige Ende der Stufe verwendet wird und damit jeweiligen Lieferumfang darstellt.

Die Zeitpunktsbetrachtung über das Prädikat $\mathcal{Q}_{VZeitpunkt}$ ist eine Hilfe bei der Planung und der Kontrolle eines Projekts: Sie verwendet für die Auswahl der anzuzeigenden Komponenten den jeweiligen Fertigstellungstermin, so wie er bereits bei der Planungssicht definiert wurde. Zur Auswahl der Komponenten wird ein Auswahlprädikat $\mathcal{Q}_{VZeitpunkt} : V \rightarrow \mathbb{B}$ auf der Menge der Komponenten einer logischen Architektur definiert:

$$\mathcal{Q}_{VZeitpunkt}(v) := \begin{cases} true & : value(v, Plan.Ende) \leq t, t \in Datum \vee v \notin leafs(C) \\ false & : sonst \end{cases}$$

Der gesuchte Zeitpunkt wird in das Prädikat einbezogen. Alle zusammengesetzten Komponenten werden ohne Prüfung auf den Fertigstellungszeitpunkt übernommen, sonst würde die Zeitpunktsbetrachtung kein Ergebnis liefern. Zusammengesetzte Komponenten sind erst dann fertiggestellt, wenn alle ihre Teilkomponenten fertig sind.

$\mathcal{L}_{Zeitpunkt} := \Xi_V(\mathcal{L}, \mathcal{Q}_{VZeitpunkt})$

Die Abbildung 7.11 zeigt ein Beispiel für eine Zeitpunktssicht auf die Kundenverwaltung. Dargestellt ist die Architektur zu einem Zeitpunkt nach dem 17.11.2006 und vor dem 01.12.2006. Die Komponenten, die zu diesem Zeitpunkt noch nicht fertig gestellt sind, sind ausgegraut.

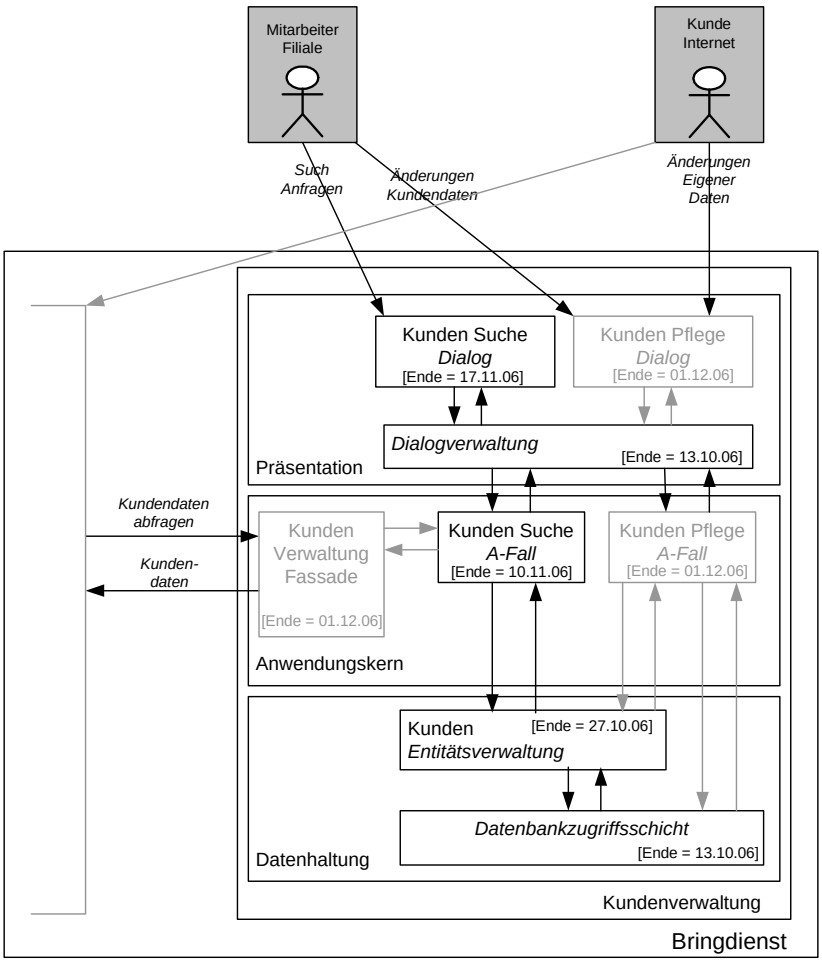


Abbildung 7.11: Konfiguration der Bringdienst-Software zu verschiedenen Zeitpunkten

Mithilfe der Projektion kann auf die ausführbaren Nutzungsszenarios geschlossen werden. Das sind die Nutzungsszenarios, die unverändert von $\mathcal{L}$ in die Bildarchitektur $\mathcal{L}_{Zeitpunkt}$ übernommen wurden:

$UC_{testbar} := \{uc \in UC | S(uc) = S_{Zeitpunkt}(uc)\}$

Aus der grafischen Darstellung ist abzulesen, dass bei vorzeitiger Integration alle Anwendungsfälle zur Kundensuche (im Beispiel ist dies $uc_1$) ab dem 17.11.06 testbar sind.

## 7.7 Zusammenfassung

Am Beispiel der Generierung verschiedener Sichten wird die Anwendbarkeit der Architekturtheorie praktisch demonstriert. Die Generierungsvorschrift für die Sichten, also der Blickwinkel, wird mit den Mitteln der Architekturtheorie beschrieben. Damit wird erstmals eine formale Darstellung der sonst nur informell definierten Blickwinkel angegeben.

Software-Architekten und Projektleiter werden mit diesen generierbaren Sichten in ihrer täglichen Arbeit unterstützt. Teile der sonst manuell zu erstellenden Architekturgrafiken können nun mit einem Werkzeug aus einer zentralen Beschreibung generiert werden. Für den Projektleiter und seine Mitarbeiter wird ein Projekt durch die Möglichkeit, übersichtliche Schaubilder zu generieren, transparenter. Beispiele für solche Darstellungen wurden im vorliegenden Kapitel demonstriert, etwa die Darstellung der Planungsdaten auf der Architektur, die Karte der Zuständigkeiten oder die T-Architektur.

# Kapitel 8

# Architekturzentriertes Projektmanagement

Offenbar kann in der Praxis ein Zusammenhang zwischen der logischen Architektur eines IT-Systems und der Planung und der Organisation seiner Erstellung hergestellt werden. Empirische Hinweise sind beispielsweise bei Herbsleb und Grinter [HG99a, GHP99] oder bei Pizka und Bauer [BP03] zu finden. Conway hat ähnliche Beobachtungen 1968 formuliert [Con68]: *Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations.*

Dieses Kapitel charakterisiert in Abschnitt 8.1 den Zusammenhang zwischen System- bzw. Software-Architektur und Projektmanagement. Um den Zusammenhang formal zu beschreiben, wird in Abschnitt 8.2 eine Formalisierung von Projektplänen als gerichtete Graphen vorgeschlagen. Die Abbildungen zwischen Architektur und Planung werden darauf aufbauend in Abschnitt 8.3 definiert. Das iterative architekturzentrierte Projektmanagement verwendet diese Abbildungen. Es wird in Abschnitt 8.4 dargestellt. Die Abschnitte 8.5 und 8.6 stellen schließlich zwei Verfahren zur Optimierung der Planung und Aufgabenverteilung in verteilten Teams vor.

## Übersicht

# 8.1 Zusammenhang zwischen Arbeitspaketen und Komponenten

## Zusammenarbeit von Projektleiter und Architekt

In Software-Entwicklungsprojekten werden typischerweise die Rollen Architekt (Chef-Designer) und Projektleiter unterschieden [Sie02b, S. 8f], [V-M06, Teil 4]. Der Projektleiter ist verantwortlich für die Planung und Durchführung des Projektes und der Architekt ist verantwortlich für den Entwurf des IT-Systems in Software und Hardware und die Umsetzung des Entwurfs im Verlauf des Projekts[1].

Die Planung sollte unter Berücksichtigung der Architektur entwickelt werden[2] [Sta05]. Eine grobe logische Architektur muss daher möglichst früh im Projekt verfügbar sein, damit sich die Planung daran orientieren kann. Projektleiter und Architekt sollten daher von Anfang an eng zusammenarbeiten.

**Definition 8.1 (Architekturzentriertes Projektmanagement)**
Architekturzentriertes Projektmanagement ist eine besondere Form des Projektmanagements, in der die (logische) Architektur eines IT-Systems eine Grundlage der Planung, Steuerung und Kontrolle des Entwicklungsprojektes darstellt. Architekt und Projektleiter arbeiten eng bei der Gestaltung der Architektur und der Planung zusammen.

Architekturzentriertes Projektmanagement bedeutet nicht, dass auf Ergebnisse der Planung oder des Architekturentwurfs verzichtet werden könnte. Die entstehenden Modelle und Dokumente werden besser aufeinander abgestimmt.

## Komponenten und Arbeitspakete

In Abschnitt 3.3.2 werden logische Architekturen als Grundlage der Projektplanung eingeführt. Die logische Architektur beschreibt die logischen Bestandteile des IT-Systems, die logischen Komponenten. Diese müssen entwickelt, wiederverwendet oder zugekauft werden. Eine logische Komponente bietet sich daher als Ergebnis eines Arbeitspaketes innerhalb eines Projektstrukturplans oder eines Vorgangs innerhalb eines Terminplans an.

Zwischen dem Projektplan als Modell des Erstellungsprozesses und der Architektur als Modell des zu erstellenden Produktes wird so eine Verbindung hergestellt. In der Literatur[3] wird diese Verbindung fortwährend erwähnt, jedoch nicht explizit ausgeführt oder verwendet.

## Aufteilung der Arbeit in großen Entwicklungsteams

Die Arbeitspakete aus dem Projektstrukturplan bzw. die Vorgänge aus dem Terminplan können im Rahmen der Einsatzmittelplanung nach verschiedenen Strategien auf die Teams bzw. Projektmitarbeiter verteilt werden: Einerseits können Subsysteme (bzw. logische Komponenten) als Gegenstand der Verteilung gewählt werden, andererseits können die Phasen des Vorgehensmodells benutzt werden. Beide Strategien werden in Abschnitt 4.3 vorgestellt. Mischformen der Strategien sind möglich. Welche der beiden Strategien ausgewählt wird, ist im Einzelfall vom Projektleiter zusammen mit dem Architekten zu entscheiden[4].

Unabhängig von der Wahl der Strategie ist bei großen Software-Entwicklungsprojekten eine feinere Aufteilung der Spezifikations-, Entwurfs-, Implementierungs- und Testaktivitäten notwendig. Die Verwendung logischer Komponenten als Verantwortungsbereiche für Teilteams oder Mitarbeiter ist daher sinnvoll.

[1] zur Rolle der Architektur vgl. Abschnitt 3.1
[2] Daher wirkt etwa der Systemarchitekt an der Erstellung des Projektplans im V-Modell XT mit [V-M06, Teil 4].
[3] vgl. etwa [HNS99a, Pau02, BCK03, VAC+05]
[4] Carmel diskutiert die Vor- und Nachteile beider Strategien für verteilte Software-Entwicklung [Car99, S.125ff].

## 8.2 Formalisierung der Arbeitspaketstruktur

Analog zu den logischen Architekturen werden auch erweiterte Projektstrukturpläne als gerichtete Graphen formalisiert. Diese enthalten Informationen über alle Arbeitspakete und deren hierarchische Strukturierung und auch Informationen aus der Terminplanung wie Anfangs- und Endtermin sowie Anordnungsbeziehungen zur Modellierung von zeitlichen Abhängigkeiten zwischen den Arbeitspaketen.

### 8.2.1 Arbeitspakete und ihre Abhängigkeiten

Ein Plan $\mathcal{P}$ wird formalisiert als gerichteter Graph. Die Arbeitspakete bilden die Knotenmenge $WP$ (für Work Package) des Graphen. Die Arbeitspakete können hierarchisch strukturiert werden, dies wird über die Kantenmenge $WH \subset WP \times WP$ (für Work Package Hierarchy) dargestellt. Die Abhängigkeiten zwischen den Arbeitspaketen werden mit einer weiteren Kantenmenge $WD \subset WP \times WP$ dargestellt, diese Kanten modellieren zur Vereinfachung nur Anordnungsbeziehungen des Typs Normalfolge.

Den Arbeitspaketen $wp \in WP$ werden, wie schon den Komponenten und Konnektoren, Name und Id zugeordnet, dazu werden zwei Funktionen $id$ und $name$ definiert.

$id : WORKPACKAGE \rightarrow \mathbb{ID}$.
$name : WORKPACKAGE \rightarrow \mathbb{ID}$

Jedem Arbeitspaket wird eine eindeutige Identität zugewiesen:

$\forall wp_1, wp_2 \in WORKPACKAGE : id(wp_1) = id(wp_2) \Leftrightarrow wp_1 = wp_2$

Über Teilaufgaben (Sammelvorgänge) werden Projektstrukturpläne hierarchisch strukturiert. Eine Teilaufgabe umfasst dabei ein oder mehrere Arbeitspakete sowie ggf. weitere Teilaufgaben[5]. Eine Teilaufgabe hat die gleichen Attribute wie ein Arbeitspaket. Die Belegungen der Attribute einer Teilaufgabe können jedoch nicht von Außen festgelegt werden, sondern sie werden aus den Belegungen der zugeordneten Arbeitspakete berechnet. Die Struktur eines Projektstrukturplans wird über eine Kantenmenge $WH \subset WP \times WP$ dargestellt.

Zwischen zwei Arbeitspaketen können zeitliche Abhängigkeiten (Anordnungsbeziehungen) festgelegt werden. In der Norm DIN 69900 [DIN87a] werden vier Arten von Beziehungen definiert, das sind: Normalfolge, Anfangsfolge, Endfolge und Sprungfolge[6]. Eine Anordnungsbeziehung legt die zeitliche Reihenfolge von je zwei Arbeitspaketen fest, zusätzlich kann ein Zeitabstand zwischen Anfang bzw. Ende des einen Arbeitspakets und Anfang bzw. Ende seines Nachfolgers angegeben werden. Die Abhängigkeiten in der Planung werden über eine Kantenmenge $WD \subset WP \times WP$ dargestellt. Der Zeitabstand wird noch nicht modelliert.

### 8.2.2 Attribute und ihre Belegung

Zentrales Element eines Projektstrukturplans ist das Arbeitspaket. Ihm werden Planungsdaten und später auch Istdaten aus dem laufenden Projekt zugeordnet. Zu den Planungsdaten gehören Anfangstermin, Endtermin, Dauer in Tagen, Aufwand in Personentagen und zugeordnete Ressourcen. Im Zeitraum zwischen dem Anfangstermin und dem Endtermin wird das Arbeitspaket bearbeitet. Die Dauer ist die Differenz zwischen Endtermin und Anfangstermin in Tagen abzüglich arbeitsfreier Tage (Wochenenden, Feiertage, etc.). Der Aufwand ist der geschätzte Aufwand, der notwendig ist, um das Arbeitspaket zu bearbeiten. Im Rahmen der Einsatzmittelplanung werden jedem Arbeitspaket Ressourcen zugeordnet, in der vorliegenden Arbeit sind dies nur verantwortliche Teams bzw.

[5] entspricht Composite-Muster [GHJV94]
[6] vgl. Abschnitt 4.3

| Attributtyp | Attributname $name$ | $+_{name}$ | $\leq_{name}$ | $0_{name}$ |
|---|---|---|---|---|
| Datum | Plan.Anfang | $min$ | $\geq$ | $MAXDATUM$ |
| Datum | Plan.Ende | $max$ | $\leq$ | $MINDATUM$ |
| Tage | Plan.Dauer | $+$ | $\leq$ | 0 Tage |
| Personentage | Plan.Aufwand | $+$ | $\leq$ | 0 Personentage |
| Personenmenge | Plan.Verantwortlich | $\cup$ | $\subseteq$ | $\emptyset$ |
| Datum | Ist.Anfang | $min$ | $\geq$ | $MAXDATUM$ |
| Datum | Ist.Ende | $max$ | $\leq$ | $MINDATUM$ |
| Tage | Ist.Dauer | $+$ | $\leq$ | 0 Tage |
| Personentage | Ist.ErbrachterAufwand | $+$ | $\leq$ | 0 Personentage |
| Personentage | Ist.Restaufwand | $+$ | $\leq$ | 0 Personentage |
| Personenmenge | Ist.Verantwortlich | $\cup$ | $\subseteq$ | $\emptyset$ |

Tabelle 8.1: Attribute aus einem Projektplan, die Komponenten und Konnektoren zugeordnet werden

Mitarbeiter.

Neben den Planungsdaten werden den Arbeitspaketen im Rahmen des Projekt-Controllings die Istdaten zugeordnet. Dies sind der tatsächliche Anfangstermin, der tatsächliche Endtermin, die tatsächliche Dauer in Tagen und das oder die Teams, die tatsächlich für das Arbeitspaket verantwortlich sind. Die Aufwände werden aufgeteilt in den bereits erbrachten Aufwand und den geschätzten Restaufwand. Hat der Restaufwand den Wert 0 Personentage, ist das Arbeitspaket fertig gestellt.

Die Tabelle 8.1 fasst alle Attribute zusammen, welche den Arbeitspaketen $WP$ aus einem Plan $\mathcal{P}$ zugeordnet werden. Wie in den vorangegangenen Kapiteln werden auch die von der Architekturtheorie verlangten Summenoperatoren, Vergleichsoperatoren und die Neutralen Elemente dargestellt. Diese sind auch für Attribute eines Plans erforderlich, da die Attributbelegungen für Arbeitspakete, die nicht Blätter der Hierarchie sind, als Summen berechnet werden müssen.

Einem Plan $\mathcal{P}$ wird immer die Attributmenge $A_P :=$ {*Plan.Anfang, Plan.Ende, Plan.Aufwand, Plan.Dauer, Plan.Verantwortlich* } fest zugeordnet. Diese Menge wird im Rahmen des Projekt-Controllings um Attribute für die Ist-Daten ergänzt, $A_C :=$ { *Ist.Anfang, Ist.Ende, Ist.ErbrachterAufwand, Ist.Restaufwand, Ist.Dauer, Ist.Verantwortlich*}. Analog zu den logischen Architekturen werden den Arbeitspaketen über eine Belegung $wvalue$ die Attributwerte zugewiesen.

$wvalue : WORKPACKAGE \times A_P \cup A_C \rightarrow \mathbb{D} \cup \{\langle\rangle\}$.

Dabei ist $\mathbb{D}$ die Vereinigungsmenge aller Definitionsbereiche der Planungs- und Controlling-Attribute aus $A_P$ und $A_C$. Ist ein Attribut für ein Arbeitspaket nicht belegt, wird der Wert $\langle\rangle$ (*undefiniert*) zugewiesen.

### 8.2.3 Projektplan als gerichteter Graph

Mit den definierten Knoten- und Kantenmengen sowie der Belegung kann ein Plan als gerichteter Graph beschrieben werden. Das Tupel $\mathcal{P} = (WP, WH, WD, wvalue)$ wird Plan genannt, genau dann wenn

- $WP \subset WORKPACKAGE$ eine endliche Menge von Arbeitspaketen ist,
- $WH \subset WP \times WP$ den hierarchischen Aufbau des Plans darstellt. Der Graph $G = (WP, WH)$ ist dabei ein gerichteter Baum. Der gerichtete Graph $G$ ist also kreisfrei (auch $(wp, wp) \notin WH$), und es gibt genau eine Wurzel $wr \in W$, also ein Arbeitspaket, das nicht Teil eines anderen ist: $\exists wr \in WP : \forall wp \in WP, wp \neq wr (wp, wr) \notin WH$.
- $WD \subset WP \times WP$ die Abhängigkeiten (Normalfolge) zwischen den Arbeitspaketen darstellt.

- $wvalue$ eine Belegung ist, die jedem Arbeitspaket $wp \in WP$ und jedem Attribut $a \in A_P$ einen Wert aus dem Wertebereich des jeweiligen Attributs zuordnet. Dies kann auch das neutrale Element aus dem Wertebereich des Attributes sein.

Die Menge aller denkbaren Pläne wird mit $PLAN$ bezeichnet. Allen Plänen aus $PLAN$ wird die Attributmenge $A_P$ explizit zugeordnet, ohne dass diese Attributmenge in der Tupel-Schreibweise von $\mathcal{P}$ angegeben werden muss

Die Abbildung 8.1 zeigt ein Beispiel für die Darstellung eines groben Projektstrukturplans zur Erstellung des Bringdienstsystems. Dargestellt sind vier Arbeitspakete und eine Teilaufgabe (Sammelvorgang). Der Plan ist nicht vollständig, da Arbeitspakete zur Integration in die Umgebung (Buchungssystem, Rechnungsdrucker) noch fehlen. Der Graph wird im Folgenden $BP \in PLAN$ genannt.

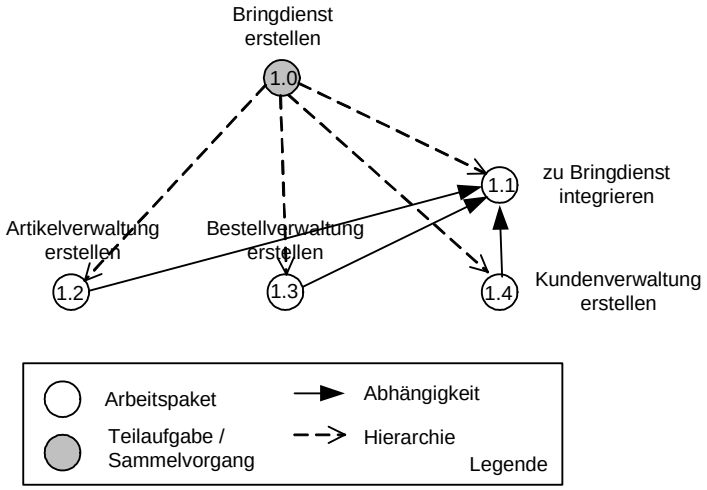


Abbildung 8.1: Darstellung eines Projektstrukturplans als Graph

Mit der oben dargestellten Formalisierung wird der Graph beschrieben über das Tupel

$BP = (WP_{BP}, WH_{BP}, WD_{BP}, wvalue_{BP})$ mit:

- $WP_{BP} = \{1.0, 1.1, 1.2, 1.3, 1.4\}$, die Namen der Arbeitspakete sind in der Abbildung 8.1 zu finden.
- $WH_{BP} = \{(1.0, 1.1), (1.0, 1.2), (1.0, 1.3), (1.0, 1.4)\}$, d.h. die Arbeitspakete '1.1' bis '1.4' sind Teile der Teilaufgabe '1.0'.
- $WD_{BP} = \{(1.2, 1.1), (1.3, 1.1), (1.4, 1.1)\}$, d.h. die Arbeitspakete '1.2' bis '1.4' müssen fertiggestellt sein, bevor das Paket '1.1' begonnen werden kann. Das IT-System kann also erst integriert werden, wenn seine Komponenten fertiggestellt sind.
- $wvalue_{BP}$ wird in der Tabelle 8.2 dargestellt. Die Belegung des Arbeitspakets '1.0' ist berechnet aus den Belegungen ihrer Teilarbeitspakete, etwa

  $value(1.0, Plan.Anfang) = min(\ value(1.1, Plan.Anfang), value(1.2, Plan.Anfang),$
  $value(1.3, Plan.Anfang), value(1.4, Plan.Anfang))$ und
  $value(1.0, Plan.Ende) = max(\ value(1.1, Plan.Ende), value(1.2, Plan.Ende),$
  $value(1.3, Plan.Ende), value(1.4, Plan.Ende))$

  siehe Tabelle 8.1.

Um mit Plänen umzugehen, werden analog zu den logischen Architekturen zwei Hilfsfunktionen definiert, welche den hierarchischen Aufbau $WB$ auswerten. Die Funktion $wleafs$ liefert alle Blatt-

| Arbeitspaket | Plan Anfang | Plan Ende | Plan Dauer | Plan Aufwand | Plan.Verantwortlich |
|---|---|---|---|---|---|
| 1.0 | *02.10.06* | *05.03.07* | ⟨⟩ | *1010 PT* | *Team München, Team Stuttgart, Team Sofia* |
| 1.1 | 29.01.07 | 05.03.07 | ⟨⟩ | 100 PT | Team München |
| 1.2 | 02.10.06 | 04.12.06 | ⟨⟩ | 230 PT | Team Stuttgart |
| 1.3 | 23.10.06 | 08.01.07 | ⟨⟩ | 280 PT | Team München |
| 1.4 | 02.10.06 | 08.12.06 | ⟨⟩ | 400 PT | Team Sofia |

Tabelle 8.2: Arbeitspakete eines Projekt(struktur)plans

Arbeitspakete eines Planes, also Arbeitspakete, die nicht weiter unterteilt sind. Die Funktion ist definiert als:

$wleafs : PLAN \rightarrow \wp(WORKPACKAGE)$

auf Plänen der Form $\mathcal{P} = (WP, WH, WD, wvalue)$ und ist definiert mit:

$wleafs(\mathcal{P}) := \{wp \in WP | \forall wx \in WP : (wp, wx) \notin WH\}$

Zur Berechnung von Summen von Attributwerten bei zusammengesetzten Arbeitspaketen wird eine Funktion $wparts$ benötigt, die alle direkten Teile eines Arbeitspaketes liefert.

$wparts : PLAN \times WORKPACKAGE \rightarrow \wp(WORKPACKAGE)$

liefert zu einer Teilaufgabe bzw. einem Arbeitspaket $wp$ alle direkten Teile in Bezug auf die Hierarchie $WH$:

$wparts(\mathcal{P}, wp) := \{wx \in WP | (wp, wx) \in WH\}$

Für alle Teilaufgaben (zusammengesetzte Arbeitspakete) $wp \notin wleafs(\mathcal{P})$ werden die Attributwerte aus den Attributwerten der direkten Kinder berechnet und können nicht durch den Projektleiter oder den Architekten geändert werden. Dies entspricht dem typischen Verhalten von Projektmanagementwerkzeugen bei Sammelvorgängen.

$\forall wp \notin wleafs(\mathcal{P}), \forall a \in A_P : wvalue(wp, a) = \sum^{a}_{wx \in wparts(\mathcal{P}, wp)} (wvalue(wx, a))$.

## 8.3 Synchronisation von Architektur und Arbeitspaketstruktur

### 8.3.1 Generierung eines initialen Projektplans

Einer logischen Komponente kann mindestens ein Arbeitspaket in einem Projekt(struktur)plan zugeordnet werden. Ergebnis des Arbeitspakets ist dabei die spezifizierte, implementierte, getestete oder integrierte Komponente: Aus der Architekturbeschreibung kann ein erster Projektstrukturplan, der passende Arbeitspakete jedoch ohne definierten Anfangs- und Endtermin enthält, generiert werden. Dieser kann auch als Stückliste für die Aufwandsschätzung verwendet werden.

Die Abbildung 8.2 gibt ein Beispiel für diesen Zusammenhang. Für jede Komponente und jeden Konnektor ist im Projektplan, der als Balkendiagramm (Gantt-Diagramm) dargestellt ist, ein Arbeitspaket definiert. Wird eine Komponente aus mehreren Teilkomponenten und Konnektoren zusammengesetzt, kann ein Arbeitspaket für die Integration der Teile zum Ganzen definiert werden, sowie eine Teilaufgabe, die alle Arbeitspakete der Teilkomponenten zusammenfasst.

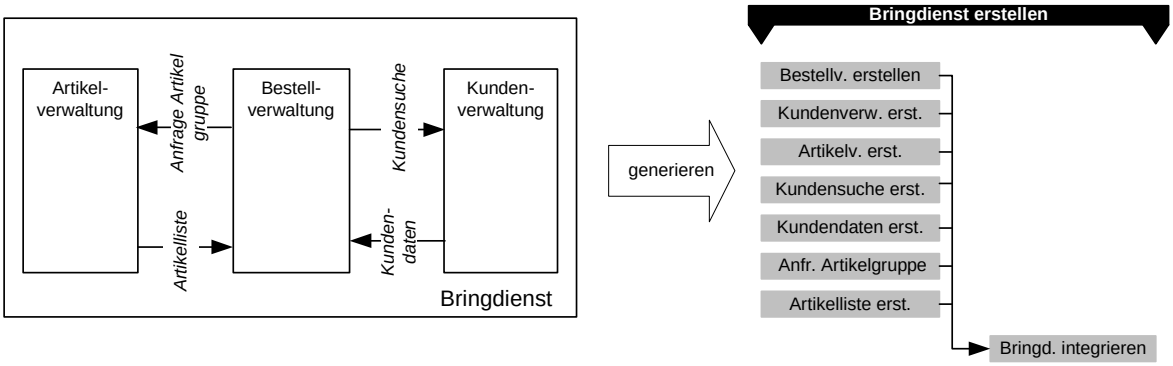


Abbildung 8.2: Generierung des initialen Projektplans aus der Architekturbeschreibung

### Zusammenhang zwischen Arbeitspaketen und Komponenten

Jedem Arbeitspaket innerhalb des Projektstrukturplans ist ein Ergebnis zugeordnet. Dies kann beispielsweise die Spezifikation, die Implementierung oder das Ergebnis eines Tests einer Komponente oder eines Konnektors sein. Damit können mehrere Arbeitspakete zur Erstellung einer Komponente oder eines Konnektors beitragen. Umgekehrt kann im allgemeinen Fall ein Arbeitspaket mehrere Komponenten oder Konnektoren als Ergebnis haben. Es muss daher davon ausgegangen werden, dass im Allgemeinen eine m:n Beziehung zwischen Arbeitspaketen und Komponenten bzw. Konnektoren besteht.

Die Generierungsvorschrift für einen Projektplan muss zudem im Allgemeinen das verwendete Vorgehensmodell kennen: Das Vorgehensmodell bestimmt die von der Idee bis zur Inbetriebnahme zu erstellenden Produkte und die Aktivitäten, welche die Produkte erzeugen oder modifizieren. Das Vorgehensmodell legt damit die Aktivitäten und Zwischenprodukte fest, welche zur Fertigstellung einer Komponente wichtig sind. Der Zusammenhang zwischen Vorgehensmodell und Projektplanung wird von Gnatz [Gna05] diskutiert. Er erzeugt aus der Beschreibung eines Vorgehensmodells einen Projektplan für Projekte, die konform zu dem Vorgehensmodell durchgeführt werden[7].

Die Eigenarten und Erfahrungen einer Organisation bestimmten die Methodik der Projektplanung und der Projektdurchführung. Die Detaillierung der Planung und damit auch die Größe der Arbeitspakete unterscheiden sich im Allgemeinen von (Teil-)Organisation zu (Teil-)Organisation.

Auf der Grundlage der logischen Architektur kann also kein vollständiger Projektplan generiert werden. Ein Teil eines Plans[8] kann jedoch aus der logischen Architektur erzeugt werden, dies kann dem Projektleiter als erste Grundlage für die Aufwandsschätzung und die Planung dienen.

Für den initialen Plan $\mathcal{P}$ wird eine Hierarchie von Arbeitspaketen erzeugt. Die Zuordnung wird über vier Funktionen beschrieben, das sind $\gamma_V$, $\gamma_E$, $\gamma_I$ und $\Gamma$. Sie ordnen den Elementen einer logischen Architektur $\mathcal{L} = (C, A, value, UC, S)$ mit der Konfiguration $C = (V, E, J, H)$ die Elemente eines Plans $\mathcal{P} = (WP, WH, WD, wvalue)$ zu.

### Komponenten

Die Funktion $\gamma_V$ ordnet jeder Komponente $v \in V$ einer Konfiguration $C = (V, E, J, H)$ genau ein Arbeitspaket $wp \in WORKPACKAGE$ zu. Dieses Arbeitspaket $wp$ beinhaltet alle Tätigkeiten, die zur Erstellung der Software oder Hardware für $v$ notwendig sind, also Spezifikations-, Entwurfs-,

[7] Für das V-Modell XT steht mit dem Projektassistenten ein Werkzeug zur Generierung erster Projektpläne zur Verfügung [Koo06]

[8] Der Plan enthält nur Arbeitspakete, die eindeutig einer oder mehreren Komponenten und Konnektoren zugeordnet werden können.

Implementierungs- und Modultest-Tätigkeiten.

$\gamma_V : V \rightarrow WORKPACKAGE$

$\gamma_V(v) := wp \in WORKPACKAGE, v \in V$

Der Name des Arbeitspakets ergibt sich aus dem Namen der Komponente und dem Zusatz 'erstellen', also $name(\gamma_V(v)) := name(v)+$ ' erstellen'. Die Id des Arbeitspaketes $id(\gamma_V(v)) \in \mathbb{ID}$ wird generiert. Arbeitspakete und Komponenten sind sich eindeutig zugeordnet: $\gamma_V(v_1) = \gamma_V(v_2) \Leftrightarrow v_1 = v_2$ mit $v_1, v_2 \in V$.

#### Konnektoren

Jedem Konnektor $e \in E$ der Konfiguration $C = (V, E, J, H)$ wird mit $\gamma_E$ ein Arbeitspaket $wp \in WORKPACKAGE$ zugeordnet.

$\gamma_E : E \rightarrow WORKPACKAGE$

$\gamma_E(e) := wp \in WORKPACKAGE, e \in E.$

Der Name des Arbeitspakets ergibt sich aus dem Namen des Konnektors analog zu der Erzeugung der Namen der Komponenten: $name(\gamma_E(e)) :=$ 'Konnektor '$+name(e)+$ ' erstellen'. Die Id des Arbeitspaketes $id(\gamma_E(e)) \in \mathbb{ID}$ wird generiert. Arbeitspakete und Konnektoren sind sich eindeutig zugeordnet: $\gamma_E(e_1) = \gamma_E(e_2) \Leftrightarrow e_1 = e_2; e_1, e_2 \in E$.

Die Funktionen $\gamma_V$ und $\gamma_E$ erzeugen jeweils verschiedene Arbeitspakete:

$\forall v \in V, \forall e \in E : \gamma_E(e) \neq \gamma_V(v)$.

#### Zusammengesetzte Komponenten

Die Funktion $\gamma_I$ ordnet jeder zusammengesetzten Komponente der Konfiguration $C = (V, E, J, H)$ ein Arbeitspaket $wp \in WORKPACKAGE$ zu. Da dieselben Komponenten in verschiedenen Konfigurationen vorkommen können und in der einen zusammengesetzt sein können, in der anderen aber möglicherweise nicht, kann die Funktion $\gamma_I$ nur die Hierarchie einer Konfiguration zur Zuordnung zu $wp$ verwenden.

$\gamma_I : \wp(H) \rightarrow WORKPACKAGE \cup \{\langle\rangle\}$

Parameter für $\gamma_I$ ist demnach eine Menge $h \subseteq H$ von 2-Tupeln von Komponenten. Für diese 2-Tupel gilt eine Einschränkung: Die erste Komponente jedes Tupels muss identisch sein, damit stellt die Tupelmenge die hierarchische Beziehung zwischen einer Komponente und ihren Kindkomponenten dar. Ist diese Voraussetzung nicht erfüllt, ist das Ergebnis von $\gamma_I$ undefiniert $\langle\rangle$. Da $\gamma_I$ nur innerhalb der hier vorgestellten Definitionen verwendet wird, tritt $\langle\rangle$ niemals als Ergebnis auf.

$\gamma_I(h(x)) := wp \in WORKPACKAGE,$
$h(x) = \{(x, v) | x, v \in V; (x, v) \in H\} \subseteq H$

Der Name des Arbeitspakets ergibt sich aus dem Namen der Komponente, die in allen 2-Tupeln vorkommt $name(\gamma_I(h(x))) :=$'zu '$+name(x)+$ ' integrieren', mit dem oben definierten $h(x)$. Die Id des Arbeitspaketes $id(\gamma_I(h)) \in \mathbb{ID}$ wird generiert. Arbeitspakete und Komponenten-Tupel sind sich eindeutig zugeordnet: $\gamma_I(h_1) = \gamma_I(h_2) \Leftrightarrow h_1 = h_2$.

$\gamma_I$, $\gamma_V$ und $\gamma_E$ können nicht dasselbe Arbeitspaket erzeugen: $\forall v \in V, \forall h \in \wp(H), \forall e \in E : \gamma_I(h) \neq \gamma_V(v) \neq \gamma_E(e)$.

**Pläne**

Die Funktion $\Gamma$ verwendet $\gamma_V$, $\gamma_I$ und $\gamma_E$ und erzeugt für eine logische Architektur $\mathcal{L} \in LA$ einen Plan $\mathcal{P} \in PLAN$.

$\Gamma : LA \rightarrow PLAN$

$\Gamma$ übersetzt den gerichteten Multigraphen, welcher die logische Architektur darstellt, in den gerichteten Graphen, der den Plan modelliert. Sei $\mathcal{L} = (C, A, value, UC, S)$ eine logische Architektur mit der Konfiguration $C = (V, E, J, H)$, dann kann daraus ein Plan $\mathcal{P} = (WP, WH, WD, wvalue)$ erzeugt werden $\Gamma(\mathcal{L}) := \mathcal{P}$ mit:

- $WP := \{\gamma_V(v)|v \in V\} \cup \{\gamma_E(e)|e \in E\} \cup \{\gamma_I(\{(x,v)|v \in V \wedge (x,v) \in H\})|x \in V, x \notin leafs(C)\}$
- $WH := \{(\gamma_V(x), \gamma_V(v))|x, v \in V, (x,v) \in H\} \cup$
  $\{(\gamma_V(x), \gamma_I(\{(x,v) \in H|v \in V\}))|x \in V, x \notin leafs(C)\} \cup$
  $\{(\gamma_V(x), \gamma_E(e))|J(e) = (v_1, v_2); v_1, v_2 \in V; x \in allparents(v_1) \cap allparents(v_2)$
  $\wedge depth(C,x) > depth(C,y) \forall y \in allparents(v_1) \cap allparents(v_2), x \neq y\}$
  Die Komponentenhierarchie $H$ wird in die Strukturierung der Arbeitspakete direkt übernommen ( also $\{(\gamma_V(x), \gamma_V(v))|x, v \in V, (x,v) \in H\}$), zusätzlich werden die Arbeitspakete, die zur Integration der Teilkomponenten erzeugt wurden, unter das Arbeitspaket für die zusammengesetzte Komponente gehängt, daher $\{(\gamma_V(x), \gamma_I(\{(x,v)|v \in V; (x,v) \in H\}))|x \in V, x \notin leafs(C)\}$. Die Arbeitspakete für die Konnektoren, werden unter das Arbeitspaket für den gemeinsamen Vater des Anfangs- und des Endpunktes gehängt.
- $WD := \{(\gamma_V(y), \gamma_I(\{(x,v) \in H|v \in V\}))|x, y \in V, x \notin leafs(C), (x,y) \in H\} \cup$
  $\{(\gamma_E(e), \gamma_I(\{(x,v) \in H|v \in V\}))|J(e) = (v_1, v_2), v_1, v_2 \in V,$
  $x \in allparents(v_1) \cap allparents(v_2) \wedge$
  $depth(C,x) > depth(C,y) \forall y \in allparents(v_1) \cap allparents(v_2), x \neq y\}.$
  Die Integration einer Komponente aus ihren Teilkomponenten und den internen Konnektoren ist erst möglich, wenn die Teilkomponenten und die internen Konnektoren fertig gestellt sind. Daher werden zwischen den entsprechenden Arbeitspaketen Abhängigkeiten definiert.
- Wenn der logischen Architektur bereits Attribute zugeordnet sind $a \in A_P \cap A$, die auch in der Planung vorkommen, werden die entsprechenden Belegungen übernommen (siehe dazu die Zusammensetzung von $WP$).
  $a \in A_P \cap A, wp \in WP$
  $$wvalue(wp,a) := \begin{cases} value(v,a) & v \in V, wp = \gamma_V(v), v \in leafs(C) \\ \sum^{a}_{wp' \in wparts(\mathcal{P},wp)} value(wp',a) & v \in V, wp = \gamma_V(v), \\ & v \notin leafs(C) \\ value(x,a) & x \in V, wp = \gamma_I(\{(x,v) \in H|v \in V\}), \\ & x \notin leafs(C) \\ value(e,a) & e \in E, wp = \gamma_E(e) \end{cases}$$

Die definierten Abbildungen sollen nun auf eine logische Architektur angewendet werden, dafür wird die Konfiguration $K = (V_K, E_K, J_K, H_K)$ aus Abschnitt 5.3 verwendet. Sie stellt die Bausteinsicht des Bringdienstsystems mit den drei Komponenten Artikelverwaltung (1), Bestellverwaltung (2) und Kundenverwaltung (3) dar.

- $V_K = \{\sigma, 1, 2, 3\}$, wobei die Komponente $\sigma$ das Bringdienstsystem darstellt.
- $E_K = \{a, b, c, d\}$
- $J_K = \{(a, (1,2)), (b, (2,1)), (c, (2,3)), (d, (3,2))\}$
- $H_K = \{(\sigma, 1), (\sigma, 2), (\sigma, 3)\}$

Sei $\mathcal{P}_K = (WP_K, WH_K, WD_K, wvalue_K)$ der initiale Plan, der über die Abbildung $\Gamma$ aus $K$ erzeugt wird, also $\Gamma(K) = \mathcal{P}_K$. Die Abbildung 8.3 stellt rechts das Ergebnis der Abbildung in Form des gerichteten Graphen zu $\mathcal{P}_K$ grafisch dar. Die Tabelle 8.3 zeigt die Belegung der Arbeitspakete, so wie sie auch später in ein Projektmanagementwerkzeug übernommen werden können.

- $WP_K = \{1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8\}$, die Namen der Arbeitspakete sind in der Abbildung 8.3 und in der Tabelle 8.3 zu finden. Für jede Komponente und jeden Konnektor aus $K$ wurde ein Arbeitspaket generiert. Das bedeutet etwa für die Komponenten $\sigma$ und 1 $\gamma_V(\sigma) = 1.0$, $\gamma_V(1) = 1.2$ oder für die Konnektoren a und b $\gamma_V(a) = 1.5$ sowie $\gamma_V(b) = 1.6$. Zusätzlich wurde das Arbeitspaket 1.1 zur Integration erzeugt, also $\gamma_I(\{(\sigma, 1), (\sigma, 2), (\sigma, 3)\}) = 1.1$
- $WH_K = \{(1.0, 1.1), (1.0, 1.2), (1.0, 1.3), (1.0, 1.4), (1.0, 1.5), (1.0, 1.6), (1.0, 1.7), (1.0, 1.8)\}$, d.h. alle Arbeitspakete sind Teile von dem Arbeitspaket 1.0.
- $WD_K = \{(1.2, 1.1), (1.3, 1.1), (1.4, 1.1), (1.5, 1.1), (1.6, 1.1), (1.7, 1.1), (1.8, 1.1)\}$, d.h. die Arbeitspakete '1.2' bis '1.8' müssen fertiggestellt sein, bevor das Paket '1.1' begonnen werden kann.
- $wvalue_K$ als Belegung der einzelnen Arbeitspakete, sie ist in Tabelle 8.3 dargestellt.

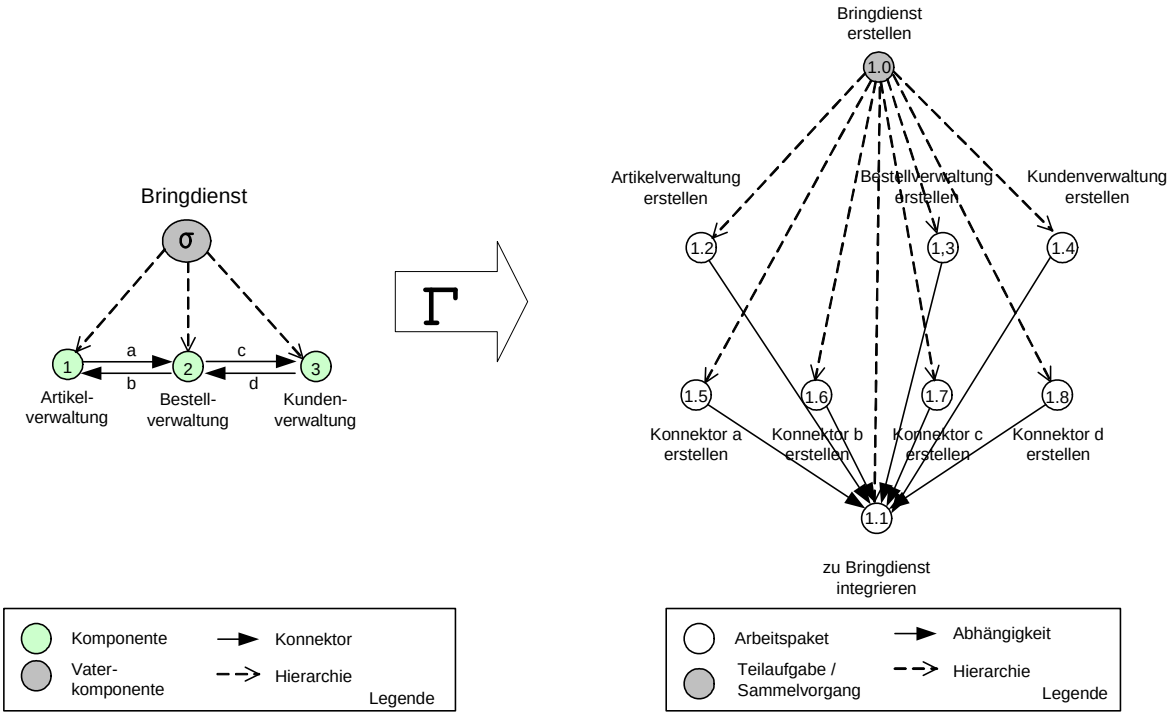


Abbildung 8.3: Generierung eines initialen Graphen für die Planung

Die Tabelle 8.3 zeigt ein Beispiel des generierten Projektplans in tabellarischer Darstellung. Um die Zuordnung zwischen der Planung und der Architektur zu erhalten, ist in der letzten Spalte eine Id der Komponente oder des Konnektors enthalten, dem das Arbeitspaket zugeordnet ist.

Die gezeigte Tabelle kann aus einem Plan $\mathcal{P}$ darüber gewonnen werden, dass der Baum $G = (WP, WH)$ mithilfe der Tiefensuche durchlaufen wird. Die Spalten der Tabelle ergeben sich aus den Belegungen der Attribute der besuchten Arbeitspakete.

Im Rahmen der Aufgaben-/Termin- bzw. Einsatzmittelplanung werden jedem Arbeitspaket Termine, Aufwände und verantwortliche Teams zugeordnet, dabei müssen nicht alle der generierten Arbeitspakete berücksichtigt werden: Projektleiter und/oder Architekt sollten alle Arbeitspakete streichen oder zusammenfassen, bei denen kein oder zu wenig Aufwand anfällt.

| Id | Name des Arbeitspakets | Anfang | Ende | Aufwand | Verantwortlich | $\mathcal{Z}$ | Nachfolger |
|---|---|---|---|---|---|---|---|
| **1.0** | **System Bringdienst erstellen** | 29.7.06 | 31.7.06 | 0 PT | | $\sigma$ | |
| 1.1 | Zu Bringdienst integrieren | 01.8.06 | 02.8.06 | 0 PT | | $\sigma$ | |
| 1.2 | Artikelverwaltung erstellen | 29.7.06 | 31.7.06 | 0 PT | | 1 | 1.1 |
| 1.3 | Bestellverwaltung erstellen | 29.7.06 | 31.7.06 | 0 PT | | 2 | 1.1 |
| 1.4 | Kundenverwaltung erstellen | 29.7.06 | 31.7.06 | 0 PT | | 3 | 1.1 |
| 1.5 | Konnektor ArtikelListe erstellen | 29.7.06 | 31.7.06 | 0 PT | | a | 1.1 |
| 1.6 | Konnektor AnfrageArtikelGruppe erstellen | 29.7.06 | 31.7.06 | 0 PT | | b | 1.1 |
| 1.7 | Konnektor KundenAnfrage erstellen | 29.7.06 | 31.7.06 | 0 PT | | c | 1.1 |
| 1.8 | Konnektor KundenDatensatz erstellen | 29.7.06 | 31.7.06 | 0 PT | | d | 1.1 |

Tabelle 8.3: Beispiel für die Generierung des ersten Projektplans

#### Expertenschätzung

Die Tabelle 8.3 mit ihren generierten Arbeitspaketen kann einerseits als Grundlage für eine Aufwandsschätzung über Expertenbefragungen verwendet werden. Burghardt stellt hierzu mehrere Verfahren vor [Bur02, S.107ff]. Grundlage aller vorgestellten Verfahren ist eine solche Tabelle mit Arbeitspaketen.

Die Expertenschätzung kann über ein zusätzliches Attribut *Schwierigkeit* := ('Schwierigkeit', {*einfach*, *mittel*, *schwer*}) erleichtert werden. Dieses Attribut wird Komponenten und Konnektoren zugeordnet und während des Entwurfs wird zu jeder Komponente angegeben, wie schwierig deren Spezifikation bzw. Implementierung ist. Ebenso können bei Konnektoren einfach zu realisierende Protokolle von komplexen Protokollen unterschieden werden. Das Attribut *Schwierigkeit* unterstützt die Experten bei ihrer Arbeit.

#### Grafische Planung

Über die Abbildung der in der logischen Architektur bereits definierten Planungsattribute ist eine grafische Expertenschätzung möglich: Architekt, Projektleiter und ggf. zusätzliche Experten schätzen gemeinsam anhand der Architekturdarstellung die Aufwände zur Umsetzung von Komponenten und Konnektoren. Die Schätzwerte werden direkt in die Architekturdarstellung eingetragen[9]. Am Ende des Verfahrens tragen die wichtigsten Komponenten und Konnektoren der logischen Architektur Schätzzahlen für den Aufwand. Eine erste Termin- und Einsatzmittelplanung ist so ebenfalls möglich.

### 8.3.2 Abbildung der Planinformationen auf die Architektur

Eine Verbindung zwischen der logischen Architektur und einem Projektstrukturplan kann über eine Relation *ist zugeordnet* manuell hergestellt werden. Eine vollautomatische Synchronisierung beider ist wegen der oben dargestellten m:n Relation zwischen Arbeitspaketen und Architekturelementen nicht möglich. Ein Arbeitspaket und eine Komponente bzw. ein Konnektor stehen genau dann in der *ist zugeordnet* Relation, wenn das Arbeitspaket zur Fertigstellung oder Veränderung der Komponente bzw. des Konnektors beiträgt. Diese allgemeine Verbindung erlaubt es, Plan und Architektur unabhängig von einander zu modifizieren und etwa im Projektstrukturplan zusätzliche Arbeitspakete zu definieren.

[9] AutoARCHITECT ist dazu in der Lage, die Kommentare zu den AutoFOCUS 2 Komponenten und Kanälen (Konnektoren) auszuwerten

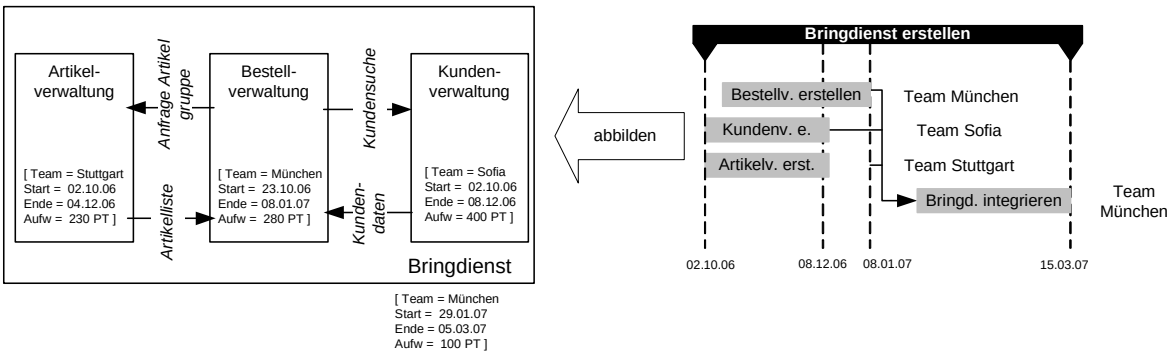


Abbildung 8.4: Abbildung von Planungsinformationen auf die Architektur

Die Relation *ist zugeordnet* ($\mathcal{Z}$) wird zwischen einer logischen Architektur $\mathcal{L} = (C, A, value, UC, S)$ und einem Plan $\mathcal{P} = (WP, WH, WD, wvalue)$ formuliert mit:

$\mathcal{Z} \subseteq (V \cup E) \times WP$, mit den folgenden Einschränkungen:

$wp \in WP, v \in V : v\mathcal{Z}wp \Rightarrow \forall x \in V, x \neq v : \neg x\mathcal{Z}wp \wedge \forall e \in E : \neg e\mathcal{Z}wp$ und

$wp \in WP, e \in E : e\mathcal{Z}wp \Rightarrow \forall f \in E : \neg f\mathcal{Z}wp \wedge \forall v \in V : \neg v\mathcal{Z}wp.$

Die Relation ist so gewählt, dass jedem Arbeitspaket höchstens ein Architekturelement zugewiesen werden kann. Jedem Architekturelement können dagegen mehrere Arbeitspakete zugeordnet werden. Damit ist die Abbildung der Planungsinformationen auf die Architekturelemente nicht umkehrbar eindeutig.

Mithilfe der Relation wird nun eine Abbildung $\bar{\Gamma}$ definiert, welche die Attribute und deren Belegung aus der Planung in die logische Architektur überträgt.

$\bar{\Gamma} : LA \times PLAN \rightarrow LA$

Sei $\mathcal{L} = (C, A, value, UC, S) \in LA$ eine logische Architektur mit der Konfiguration $C = (V, E, J, H)$ und $\mathcal{P} = (WP, WH, WD, wvalue) \in PLAN$ ein auf die logische Architektur abgestimmter Plan. $\mathcal{Z} \subseteq (V \cup E) \times WP$ sei eine Relation, die Komponenten und Konnektoren der Konfiguration $C$ mit den Arbeitspaketen des Plans $\mathcal{P}$ in Beziehung setzt. Dann ist die Abbildung der Planungsdaten auf die Architektur wie folgt definiert:

$\bar{\Gamma}(\mathcal{L}, \mathcal{P}) = \mathcal{L}'$ mit $\mathcal{L}' = (C, A', value', UC, S)$.

Die Konfiguration $C$ sowie die Anwendungsfälle $UC$ und deren Abbildung auf die Konnektoren $S$ werden durch $\bar{\Gamma}$ nicht geändert. $\bar{\Gamma}$ betrifft nur Attribute und deren Belegung.

Die Aufwände, Termine und verantwortliche Teams eines Arbeitspakets können eindeutig einer Komponente oder einem Konnektor zugeordnet werden. Sind einer Komponente oder einem Konnektor mehrere Arbeitspakete zugeordnet, werden die Informationen entsprechend der Rechenregeln der einzelnen Attribute summiert. So wird das Minimum der Anfangstermine, das Maximum der Endtermine, die Summe der Aufwände und die Vereinigungsmenge der Ressourcen gebildet:

Die Attributmenge $A$ von $\mathcal{L}$ umfasst damit mindestens die Attribute $A_P$ der Planung:

$A' := A \cup A_P$

Für die Belegung der Attribute der Komponenten und Konnektoren werden die dort eventuell zugeordneten (alten) Belegungen überschrieben und durch die aktuelleren Daten aus der Planung ersetzt $\forall a \in A_P, \forall v \in V, \forall e \in E$:

$value'(v, a) := \sum^{a}_{wp \in WP | v\mathcal{Z}wp, wp \in wleafs(\mathcal{P})} (wvalue(wp, a))$

| Id | Name des Arbeitspakets | Anfang | Ende | Aufwand | Verantwortlich | $\mathcal{Z}$ | Nachfolger |
|---|---|---|---|---|---|---|---|
| **1.0** | **Bringdienst erstellen** | *02.10.06* | *05.03.07* | *1010 PT* | *Team München, Team Stuttgart, Team Sofia* | $\sigma$ | |
| 1.1 | Zu Bringdienst integrieren | 29.01.07 | 05.03.07 | 100 PT | Team München | $\sigma$ | |
| 1.2 | Artikelverwaltung erstellen | 02.10.06 | 04.12.06 | 230 PT | Team Stuttgart | 1 | 1.1 |
| 1.3 | Bestellverwaltung erstellen | 23.10.06 | 08.01.07 | 280 PT | Team München | 2 | 1.1 |
| 1.4 | Kundenverwaltung erstellen | 02.10.06 | 08.12.06 | 400 PT | Team Sofia | 3 | 1.1 |

Tabelle 8.4: Beispiel für die Modifikation des generierten Projektplans

| Id | Name des Architekturelements | Plan Anfang | Plan Ende | Plan Aufwand | Plan Verantwortlich |
|---|---|---|---|---|---|
| $\sigma$ | **Bringdienst** | 29.01.07 | 05.03.07 | 100 PT | Team München |
| 1 | Artikelverwaltung erstellen | 02.10.06 | 04.12.06 | 230 PT | Team Stuttgart |
| 2 | Bestellverwaltung erstellen | 23.10.06 | 08.01.07 | 280 PT | Team München |
| 3 | Kundenverwaltung erstellen | 02.10.06 | 08.12.06 | 400 PT | Team Sofia |

Tabelle 8.5: Beispiel für die aus dem modifizierten Plan übertragenen Daten

$value'(e,a) := \sum_{wp \in WP | e \mathcal{Z} wp}^{a}(wvalue(wp,a))$

Die Belegungen zusammengesetzter Arbeitspakete werden nicht in die logische Architektur übertragen, da ihre Belegungen nur die Summen der Belegungen ihrer direkten Kinder darstellen. Den Komponenten und Konnektoren, zu denen es kein Arbeitspaket gibt, werden jeweils die neutralen Elemente des Attributs zugeordnet, etwa *MINDATUM* bei dem Attribut *Plan.Ende* oder $\emptyset$ beim Attribut *Plan.Verantwortlich*.

Planungs- und Istdaten zum Projektverlauf können aus gängigen Projektmanagementwerkzeugen exportiert werden. Details dazu werden in Kapitel 9 beschrieben. Um die Zuordnung über ein Werkzeug durchführen zu können, muss entweder in der Planung die Information zu den zugeordneten Architekturelementen hinterlegt werden[10] oder umgekehrt muss bei jeder Komponente und jedem Konnektor eine Liste mit IDs der zugeordneten Arbeitspakete gehalten werden. Das Werkzeug AutoARCHITECT setzt die erste Variante mit den angereicherten Plänen voraus.

Als Beispiel für diese Abbildung kann die in Abbildung 8.1 dargestellte Planung dienen. Sie ist durch Streichung von Arbeitspaketen und die Modifikation der generierten Arbeitspakete aus der Tabelle 8.3 zur logischen Architektur mit der Konfiguration $K = (V_K, E_K, J_K, H_K)$ entstanden.

Die Tabelle 8.4 zeigt zunächst die geänderte Planung zur Tabelle 8.3, die Daten entsprechen der Planung aus Tabelle 8.2. Diese wird auf die logische Architektur $K$ mithilfe der *ist-zugeordnet* Relation zurück übertragen.

Die Tabelle 8.5 zeigt schließlich die Belegung $value_K$ der Konfiguration $K$ für die planungsrelevanten Attribute. Die Belegung wurde über die Relation *ist-zugeordnet* vom Plan $\mathcal{P}_K$ auf $K$ abgebildet. Der Komponente $\sigma$ wird als Belegung nur die Belegung des Arbeitspakets $1.1$ zur Integration zugeordnet. Der Summen aus Arbeitspaket (Sammelvorgang) $1.0$ (Bringdienst erstellen), können über die Vergröberung der Architekturdarstellung auf Ebene 0 ($\Phi(\mathcal{L}_K, 0)$) berechnet werden.

Die hier vorgestellte Abbildung der Planungsdaten auf die Architektur ist die Grundlage für mehrere der in Kapitel 7 vorgestellten Architektursichten. Über Schaubilder wie die Karte der Zuständigkeiten oder eine Planungssicht kann die Transparenz eines Projektes alle Projektbeteiligten erhöht werden.

[10] z.B. zusätzliche Spalte in der tabellarischen Darstellung der Planung, welche die ID der jeweils zugeordneten Komponente oder des zugeordneten Konnektors enthält. Ein Beispiel dafür ist in Tabelle 8.3 in der Spalte *ist zugeordnet* dargestellt.

### 8.3.3 Synchronisation von Planung und Architektur im Projektverlauf

Nach der Erzeugung der initialen Planung laufen die logische Architektur und die Planung auseinander, da sie unabhängig voneinander geändert werden. Für die im Folgenden vorgestellten Verfahren zur iterativen Verbesserung der Planung und der Architektur muss ein Abgleich zwischen beiden erfolgen.

Ausgangspunkt für die Synchronisation sind die Änderungen in einer logischen Architektur. Einfache Änderungen, in denen nur Komponenten oder Konnektoren hinzugefügt oder entfernt wurden, können über die Differenzen von Komponenten- und Konnektor-Mengen ermittelt werden:

Seien $\mathcal{L}_1 = (C_1, A_1, value_1, UC_1, S_1)$ und $\mathcal{L}_2 = (C_2, A_2, value_2, UC_2, S_2)$ mit zwei logische Architekturen mit den zugehörigen Konfigurationen $C_1 = (V_1, E_1, J_1, H_1)$ und $C_2 = (V_2, E_2, J_2, H_2)$. $\mathcal{L}_2$ ist die Weiterentwicklung von $\mathcal{L}_1$.

Die Komponenten und Konnektoren, die in $\mathcal{L}_2$ ergänzt wurden, berechnen sich aus der Differenz der Architekturelemente aus $C_2$ und $C_1$. Die ergänzten und entfernten Komponenten berechnen sich damit jeweils aus den Differenzen der beiden Komponentenmengen:

- $V_+ := V_2 \setminus V_1$ sind die hinzugefügten Komponenten und
- $V_- := V_1 \setminus V_2$ sind die entfernten Komponenten.

Die Änderungen bei den Konnektoren berechnen sich analog. Zusätzlich können Konnektoren umgehängt worden sein, diese werden mit $E_\sim$ bezeichnet. Diese können über die Änderungen in der Abbildung $J$ nach $J'$ berechnet werden:

- $E_+ := E_2 \setminus E_1$ sind die hinzugefügten Konnektoren und
- $E_- := E_1 \setminus E_2$ sind die entfernten Konnektoren.
- $E_\sim := \{e \in E_1 \cap E_2 | J(e) \neq J'(e)\}$.

Auch im hierarchischen Aufbau der Architektur können Änderungen vorgekommen sein:

- $H_+ := H_2 \setminus H_1$ sind die hinzugefügten Hierarchiebeziehungen und
- $H_- := H_1 \setminus H_2$ sind die entfernten Hierarchiebeziehungen.

Die berechneten Änderungen wirken sich auch auf die Planung aus: Für neue Komponenten $V_+$ und Konnektoren $E_+$ können neue Arbeitspakete ergänzt werden, für entfernte Komponenten $V_-$ und Konnektoren $E_-$ sollten die entsprechenden Arbeitspakete gelöscht werden. Über die Mengen $E_\sim$ sowie $H_+$ und $H_-$ kann auf zu ändernde Arbeitspakete geschlossen werden.

Das Hinzufügen und Löschen kann jedoch nur durch den Architekten und den Projektleiter entschieden werden. Ein Verfahren kann hierzu nur einen Vorschlag machen.

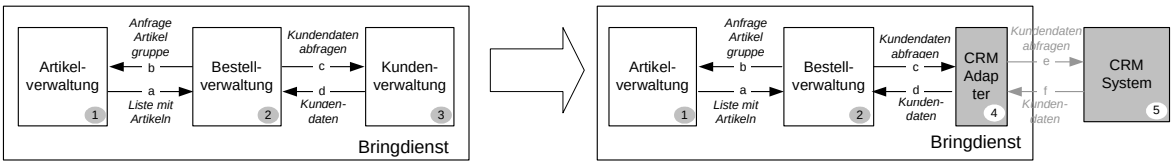


Abbildung 8.5: Ein CRM-System wird an das Bringdienstsystem angeschlossen.

Als Beispiel wird in der Konfiguration $K = (V_K, E_K, J_K, H_K)$ aus Abschnitt 5.3 (siehe oben) die Kundenverwaltung durch ein CRM-System[11] ersetzt. Diese Konfiguration wird $K' = (V_{K'}, E_{K'}, J_{K'}, H_{K'})$ genannt. Sie ist in Abbildung 8.5 in der Box-And-Arrow Notation und in Abbildung 8.5 als gerichteter Graph dargestellt. Die Konfiguration $K'$ ist gegeben durch:

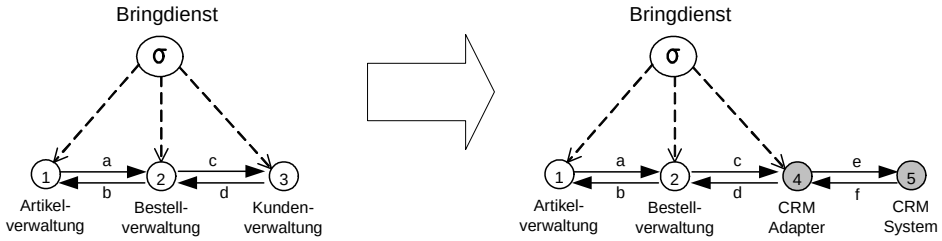


Abbildung 8.6: Ergänzung des CRM-Systems dargestellt als Multigraph

- $V_{K'} = \{\sigma, 1, 2, 4, 5\}$, wobei die Komponente $\sigma$ das Bringdienstsystem darstellt und die Komponente 5 das neue CRM-System. Die Komponente 4 ist ein Adapter, der die vorhandenen Konnektoren an das CRM-System anschließt.
- $E_{K'} = \{a, b, c, d, e, f\}$
- $J_{K'} = \{(a, (1, 2)), (b, (2, 1)), (c, (2, 4)), (d, (4, 2)), (e, (4, 5)), (f, (5, 4))\}$
- $H_{K'} = \{(\sigma, 1), (\sigma, 2), (\sigma, 4)\}$

Nun können die Änderungen in der Architektur berechnet werden. Die beiden Komponenten CRM-System (5) und CRM-Adapter (4) wurden hinzugefügt und die Komponente Kundenverwaltung (3) wurde entfernt:

- $V_+ = V_{K'} \setminus V_K = \{\sigma, 1, 2, 4, 5\} \setminus \{\sigma, 1, 2, 3\} = \{4, 5\}$
- $V_- = V_K \setminus V_{K'} = \{\sigma, 1, 2, 3\} \setminus \{\sigma, 1, 2, 4, 5\} = \{3\}$
- $E_+ = E_{K'} \setminus E_K = \{a, b, c, d, e, f\} \setminus \{a, b, c, d\} = \{e, f\}$ sind die hinzugefügten Konnektoren und
- $E_- = E_K \setminus E_{K'} = \{a, b, c, d\} \setminus \{a, b, c, d, e, f\} = \emptyset$ sind die entfernten Konnektoren.
- $E_\sim := \{c, d\}$, da beide Konnektoren von der Kundenverwaltung auf den Adapter umgehängt wurden.
- $H_+ = H_{K'} \setminus H_K = \{(\sigma, 1), (\sigma, 2), (\sigma, 4)\} \setminus \{(\sigma, 1), (\sigma, 2), (\sigma, 3)\} = \{(\sigma, 4)\}$.
- $H_- = H_K \setminus H_{K'} = \{(\sigma, 1), (\sigma, 2), (\sigma, 3)\} \setminus \{(\sigma, 1), (\sigma, 2), (\sigma, 4)\} = \{(\sigma, 3)\}$.

Aus den entfernten Komponenten und Konnektoren wird eine Liste von Arbeitspaketen berechnet, die möglicherweise gelöscht werden können. Da auch die Arbeitspakete zur Integration ('zu XXX integrieren') in der Relation $\mathcal{Z}$ zu den entsprechenden zusammengesetzten Komponenten stehen, werden auch diese identifiziert:

$WP_- := \{wp \in WP | \exists v \in V_- : v \mathcal{Z} wp\} \cup \{wp \in WP | \exists e \in E_- : e \mathcal{Z} wp\}$

Umgekehrt wird aus den ergänzten Komponenten und Konnektoren eine Liste mit Arbeitspaketen erzeugt, die möglicherweise ergänzt werden können, hierbei sind für hinzugefügte zusammengesetzte Komponenten und für Blattkomponenten, die in Teilkomponenten dekomponiert wurden, Arbeitspakete für die Integration zu ergänzen.

[11] CRM=Customer Relationship Management

$$
\begin{array}{lll}
WP_+ := & \{\gamma_V(v)|v \in V_+\}\cup & \text{für neue Komponenten} \\
& \{\gamma_E(e)|e \in E_+\}\cup & \text{für neue Konnektoren} \\
& \{\gamma_I(\{(x,v)|v \in V_+;(x,v) \in H_2\})|x \in V_+, x \notin leafs(C_2)\}\cup & \text{für neue zus. Komponenten.} \\
& \{\gamma_I(\{(x,v)|v \in V_+;(x,v) \in H_2\})|x \in V_1, x \in leafs(C_1)\} & \text{dekomp. Blätter aus } V_1
\end{array}
$$

Wenn Konnektoren umgehängt wurden oder bei zusammengesetzten Komponenten Teilkomponenten ergänzt oder entfernt wurden, kann dies den Inhalt der entsprechenden Arbeitspakete beeinflussen. Diese Arbeitspakete müssen möglicherweise angepasst werden. Sie werden berechnet durch:

$$
\begin{array}{lll}
WP_\sim := & \{wp \in WP| & \exists e \in E_\sim : e\mathcal{Z}wp\}\cup \\
& \{wp \in WP \setminus WP_-| & \exists x \in V_1 \cap V_2 : x\mathcal{Z}wp\wedge \\
& & \wedge(\exists v \in V_1 : (x,v) \in H_- \vee \exists w \in V_2 : (x,w) \in H_+)\}
\end{array}
\quad .
$$

Für die Konfiguration $K$ ist ein Projektplan in Tabelle 8.4 mit einer passenden Zuordnungsrelation $\mathcal{Z}$ angegeben. Der Plan enthält für die entfernte Komponente Kundenverwaltung (3) das Arbeitspaket 1.4 (*Kundenverwaltung erstellen*). Dieses Arbeitspaket kann aus der Planung entfernt werden:

$WP_- = \{1.4\}$ da $1.4 \in WP \wedge 3 \in V_- \wedge 3\mathcal{Z}1.4$

Zwischen den Konnektoren und den Arbeitspaketen besteht in Tabelle 8.4 keine Relation, außerdem wurde kein Konnektor entfernt.

Für die beiden neuen Komponenten 4 und 5 und die neuen Konnektoren $e$ und $f$ sind jeweils Arbeitspakete zu ergänzen. Eine zusammengesetzte Komponente (Summand mit $\gamma_I$) wurde nicht hinzugefügt:

$WP_+ = \{\gamma_V(4), \gamma_V(5)\} \cup \{\gamma_E(e), \gamma_E(f)\}$

Für die umgehängten Konnektoren waren keine Arbeitspakete eingeplant. Bei der Komponente Bringdienst wurde eine Komponente entfernt und eine andere hinzugefügt.

$WP_\sim = \{1.1\}$

## 8.4 Iteratives architekturzentriertes Projektmanagement

Die Abbildung 8.7 zeigt ein Aktivitätsdiagramm in der UML 2.0 Notation. Das Diagramm zeigt die Zusammenarbeit von Architekt und Projektleiter mithilfe der Architekturtheorie[12]. Dargestellt ist die iterative Erstellung der Grobarchitektur zusammen mit der Projektplanung.

Der Architekt beginnt damit, einen ersten Grobentwurf der logischen Architektur zu erstellen, daraus wird ein initialer Projektstrukturplan erzeugt. Architekt und Projektleiter führen mit diesem Plan Aufwandschätzungen und erste Termin- und Einsatzmittelplanungen durch. Nach der ersten Planung werden die Planungsdaten auf die Architektur abgebildet. Die verschiedenen Schaubilder aus Kapitel 7 und diesem Kapitel machen dabei eventuelle Probleme sichtbar. Daraufhin kann die Architektur angepasst werden. Mithilfe der Architekturtheorie können für die Änderungen der Architektur Änderungsvorschläge für die Planung erzeugt werden.

Bei der Definition der ersten Grobentwürfe können neben der Projektplanung auch wirtschaftliche Anforderungen und Qualitätsanforderungen Architekturtreiber sein. Diese Anforderungen können im Konflikt stehen, so kann ein hoher Transaktionsdurchsatz in der Regel nur mit hohen Kosten errecht werden. Architekt und Projektleiter müssen hier einen Kompromiss finden und die Anforderungen und verschiedene Architekturvarianten gegeneinander abwägen.

[12] Diese ist im Werkzeug AutoARCHITECT umgesetzt.

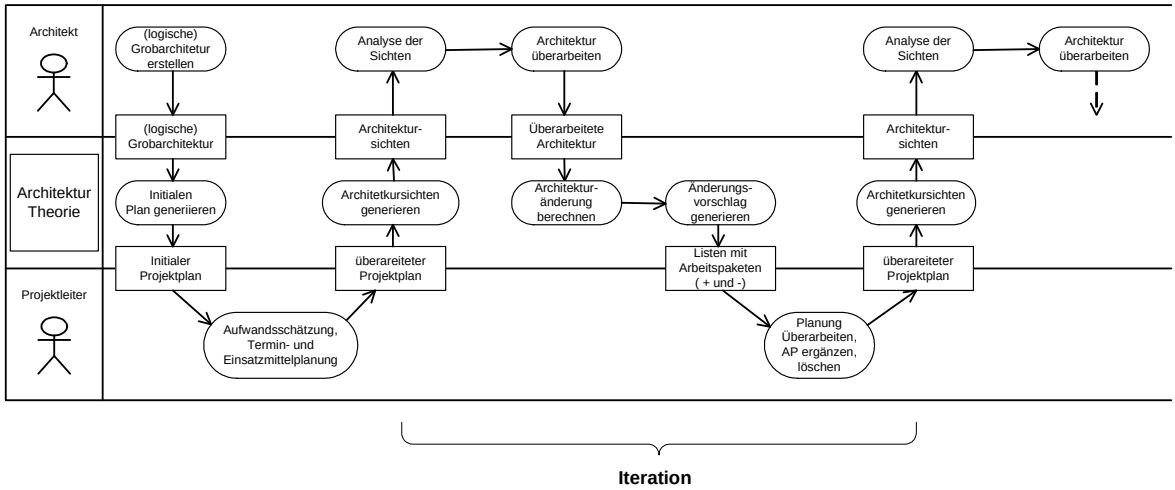


Abbildung 8.7: Gemeinsame Planung von Projektleiter und Architekt

## 8.5 Optimierung der Aufgabenverteilung in verteilten Projekten

### 8.5.1 Schnittstellen: Ursache für Abstimmungsbedarf

Die Schnittstellen der Komponenten werden in der Regel während des Entwurfs der technischen Architektur im Rahmen der Verfeinerung der Konnektoren aus der logischen Architektur spezifiziert. Schnittstellen haben eine wichtige Bedeutung: Wird die Schnittstelle einer Komponente geändert, betrifft dies die Implementierungen aller Komponenten, die diese Schnittstelle direkt nutzen.

Ist die Schnittstelle ungenau oder widersprüchlich spezifiziert, machen die Implementierungen der Komponenten, die diese Schnittstelle exportieren, Annahmen. Die Annahmen dienen dazu, mit den Ungenauigkeiten und Widersprüchen umzugehen. Die Implementierungen der Komponenten, die diese Schnittstellen nutzen, müssen (möglicherweise falsche) Annahmen über die Implementierung der exportierenden Komponente machen. Die Integration beider Komponenten führt bei unterschiedlichen Annahmen zu Fehlern.

Werden die Komponenten, welche die Schnittstelle anbieten und nutzen, von demselben Team entwickelt, werden Probleme in der Schnittstellenspezifikation schnell erkannt und können leicht behoben werden, da im Entwicklungsteam beide Implementierungen bekannt sind. Darüber hinaus kann die Schnittstelle geändert werden, ohne andere Teams zu beeinflussen.

Werden die Komponenten von verschiedenen Teams entwickelt, die an verteilten Standorten arbeiten, sind die Schnittstellen zwischen den Komponenten kritischer: Auftretende Probleme werden möglicherweise erst beim Integrationstest erkannt. Abstimmungen zwischen den Teams sind wegen der Distanz aufwändig [Car99]. Änderungen der Schnittstelle müssen zwischen allen beteiligten Teams abgestimmt werden. Die Abstimmung erfolgt bilateral zwischen den Teams, damit ist die Schnittstelle auch Kommunikationsschnittstelle zwischen den Teams. Oder die Abstimmung erfolgt über den Architekten, der die Schnittstelle definiert hat. Die Schnittstellen von Komponenten, die von verschiedenen Teams entwickelt werden, müssen daher eine höhere Stabilität und Qualität aufweisen, als die Schnittstellen zwischen Komponenten, die von demselben Team lokal entwickelt werden.

Für den Zusammenhang zwischen Architektur und Kommunikationsstruktur (und -aufkommen) gibt es empirische Hinweise: Im Global Studio Projekt [Küd05, SBM+06] gab es Untersuchungen zu diesem Thema: Dort wurde das Management global verteilter Projekte im Rahmen des Spiel-Projektes *MSLite* untersucht. Im Projekt wurden auch das Mail-Aufkommen und die Telefonate zwi-

schen den Teams gezählt. Dabei wurde festgestellt, dass zwischen dem zentralen Architekturteam, das die Schnittstellen spezifiziert hatte und den verteilt arbeitenden Entwicklungsteams wegen sich ändernder Schnittstellenspezifikationen und allgemeinen Problemen in der Architektur sehr viel kommuniziert wurde[13]. Das zentrale Architekturteam war dabei insbesondere Mediator in der Abstimmung zwischen den verschiedenen verteilten Teams, selbst wenn die Teams aus derselben Stadt kamen. Diese Beobachtungen stützen die Bedeutung der kritischen Konnektoren in der Teamkommunikation: Die Probleme bei der Abstimmung der Schnittstellen führten zu einer Überlastung des Architekturteams.

Küderli schlägt aufgrund dieser Beobachtung vor, Komponenten, die mit vielen kritischen Konnektoren verbunden sind, möglichst von räumlich benachbarten Teams entwickeln zu lassen und diesen die direkte Kommunikation zu erlauben.

#### Kritische Konnektoren in logischen Architekturen

Die Konnektoren innerhalb der logischen Architektur stellen dar, welche Komponenten miteinander über Nachrichten kommunizieren können. Die später in der technischen Architektur definierten Schnittstellen können diesen Konnektoren zugeordnet werden. Mit den oben dargestellten Beobachtungen über den Zusammenhang zwischen Schnittstellen in der Architektur und der Aufgabenverteilung kann nun eine besondere Art von Konnektoren definiert werden:

**Definition 8.2 (Kritische Konnektoren)**
Konnektoren werden *kritisch* genannt, wenn die Komponenten, die sie verbinden, von zwei verschiedenen Teams entwickelt werden. Änderungen und Fehlerbehebungen in den später zugeordneten Schnittstellen sind aufwändiger als bei Konnektoren zwischen Komponenten die von demselben Team entwickelt werden.

Küderli schlägt in [Küd05, S. 80f] ein Distanzmaß zwischen Teams vor, das räumliche, sprachliche und kulturelle Unterschiede berücksichtigt: Konnektoren können so mit unterschiedlichen Kritikalitätsmaßen bewertet werden. Die Distanz zwischen Teams in München und Stuttgart kann etwa geringer sein, als die Distanz zwischen den Teams aus München und etwa Peking. Denn in München und Stuttgart wird dieselbe Sprache gesprochen, beide Städte sind in derselben Zeitzone und im selben Kulturkreis, während diese Eigenschaften zwischen München und Peking unterschiedlich sind.

#### Optimierung der Aufgabenverteilung

Zur Reduktion der Abstimmungs- und Änderungsaufwände bei Änderungen der Schnittstellen, sollte entweder die Zahl der kritischen Konnektoren möglichst gering sein oder kritische Konnektoren sollten nur zwischen solchen Komponenten verlaufen, bei denen Änderungen der Schnittstellen unwahrscheinlich sind. Auf zwei Wegen können Anzahl und Gestalt der kritischen Konnektoren beeinflusst werden:

1. Die logische Architektur wird so entworfen, dass es möglichst wenige Konnektoren gibt. Dies entspricht der Entwurfsregel nach Yourdon und Constantine [YC79], die eine lose Koppelung zwischen den Komponenten fordern. Zusätzlich kann, Conway's Law [Con68, ER03] folgend, die Architektur an die Verteilung der Teams angepasst werden. Planungsaspekte werden zu Architekturtreibern. Diese Anpassung sollte jedoch nur unter Berücksichtigung der anderen Architekturtreiber, insbesondere der Qualitätsanforderungen, entstehen. Es könnte beispielsweise sonst die Optimierung der Aufgabenverteilung eine zu breite Client/ Server Schnittstelle und damit Nachteile in der Performance und Änderbarkeit des Systems zur Folge haben.

[13] vgl. dazu Küderli [Küd05, S. 45ff]

2. Die Entwicklung der Komponenten wird so auf die Teams aufgeteilt, dass es möglichst wenige kritische Konnektoren gibt und dass wahrscheinliche Änderungen nicht diese Schnittstellen betreffen. Komponenten, die über mehrere kritische Konnektoren verbunden sind, sollten von Teams mit geringer (räumlicher, zeitlicher, kultureller) Distanz entwickelt werden. Dabei müssen die in den Teams vorhandenen Kenntnisse und Erfahrungen zu den für die Komponenten notwendigen Fähigkeiten passen.

### 8.5.2 Kritische Konnektoren Sicht

Die *Kritische Konnektoren* Sicht ist ein erster Schritt zur Reduktion der Abstimmungsaufwände im Team und zur iterativen Verbesserung der Aufgabenverteilung in verteilten Teams: Diese Sicht macht auf das Problem der kritischen Konnektoren aufmerksam.

Die kritischen Konnektoren werden über eine Auswahl $\Xi_E$ ermittelt. Diese Sicht gehört daher zu den auswahlbasierten Sichten. Die Auswahl wählt nur die Konnektoren aus, deren Komponenten von zwei verschiedenen Teams erstellt werden. Dazu wird ein Prädikat $\mathcal{Q}_{critical} : E \rightarrow \mathbb{B}$ auf der Menge der Konnektoren $E$ der logischen Architektur $\mathcal{L}$ definiert:

$$\mathcal{Q}_{critical}(e) := \begin{cases} true & : J(e) = (v_1, v_2) \wedge \\ & value(v_1, Plan.Verantwortlich) \cap value(v_2, Plan.Verantwortlich) = \emptyset \\ false & : sonst \end{cases}$$

Das Prädikat $\mathcal{Q}_E$ wird in der Auswahl $\Xi_E$ verwendet:

$\mathcal{L}_{critical} := \Xi_E(\mathcal{L}, \mathcal{Q}_{critical})$

#### Beispiel

Die Abbildung 8.8 gibt ein Beispiel für die Darstellung der kritischen Konnektoren auf der zweiten Dekompositionsebene der logischen Architektur. Die drei Subsysteme *Artikelverwaltung*, *Bestellverwaltung* und *Kundenverwaltung* sind jeweils an Teams in München, Stuttgart und Sofia vergeben worden. Damit sind nur die Konnektoren zwischen den Subsystemen kritisch, die Konnektoren zwischen den Schichten der Subsysteme jedoch nicht.

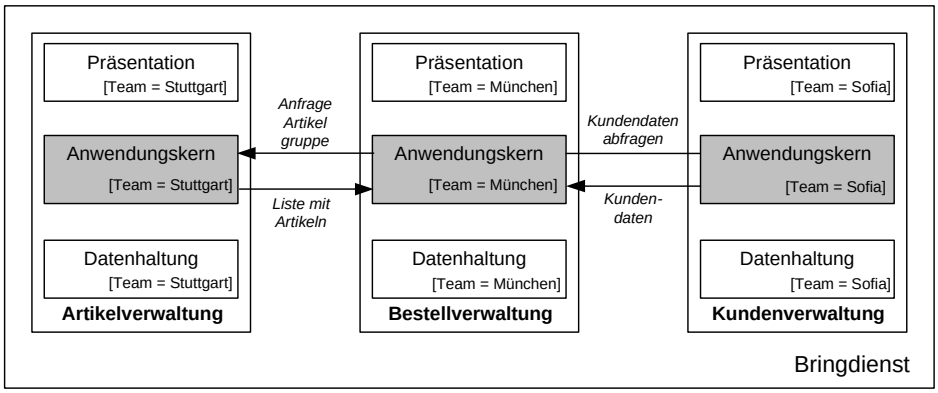


Abbildung 8.8: Darstellung der kritischen Konnektoren des Bringdienstsystems

Die Abbildung 8.9 zeigt eine zweite Variante, wie die Arbeit zwischen den drei Teams aufgeteilt werden kann. Diesmal werden die drei Schichten *Präsentation*, *Anwendungskern* und *Datenhaltung* jeweils nach Sofia, München und Stuttgart vergeben. Optisch wird deutlich, dass bei der Arbeitsaufteilung nach Schichten mehr kritische Konnektoren existieren.

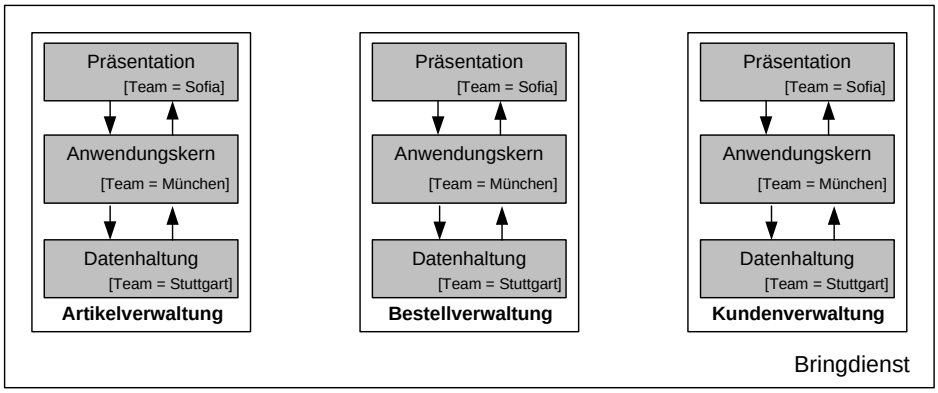


Abbildung 8.9: Beispiel für kritische Konnektoren bei Arbeitsaufteilung nach Schichten

Die Anzahl der kritischen Konnektoren ist alleine kein Optimierungskriterium für Architektur und Planung, da Kriterien wie die Verteilung des Wissens [BD04, S.602f] und die Kosten pro Arbeitsstunde eine wichtige Rolle spielen: Möglicherweise sind die Fähigkeiten der Mitarbeiter so verteilt, dass beispielsweise alle GUI-Experten in Sofia sitzen. Und die Arbeitskosten in Sofia sind möglicherweise so gering, dass zugunsten der preiswertren GUI-Entwicklung[14] die aufwändigere Schnittstellenabstimmung in Kauf genommen wird. Die kritischen Konnektoren weisen Architekt und Projektleiter auf mögliche Abstimmungsprobleme hin und auf Schnittstellen, die aufwändiger abzustimmen sind.

### 8.5.3 Verfahren zur Reduktion des Abstimmungsbedarfs

Die Identifikation kritischer Konnektoren innerhalb einer logischen Architektur macht mögliche Probleme in Architektur und Planung sichtbar. Diese Information dient Architekten und Projektleitern als Hinweis. Wie und ob die Architektur oder die Aufgabenverteilung angepasst wird, liegt im Ermessen des Projektleiters und des Architekten. Das im Folgenden vorgestellte Verfahren soll daher den Entwurfsprozess unterstützten, ihn jedoch nicht automatisieren.

**Berechnung und Darstellung der kritischen Konnektoren:** Die Darstellung der kritischen Konnektoren macht mögliche Probleme in der Abbildung zwischen Architektur und Planung sichtbar.

**Bewertung der Konnektoren:** Architekt und Projektleiter bewerten diese möglichen Schwachstellen gemeinsam und führen ggf. Änderungen der Architektur oder der Planung durch. Eine pauschale Zählung der kritischen Konnektoren erscheint wegen der oben diskutierten Einflussfaktoren nicht sinnvoll.

Das Verfahren zur Reduktion des Abstimmungsbedarfs wird abgebrochen, sobald Architekt und Projektleiter dies entscheiden: etwa wenn keine weitere Verbesserung der Planung oder der Architektur mehr möglich scheint.

**Entweder Änderung der Ressourcenzuordnung in der Planung:** Abhängig von den anderen Einflussgrößen auf Termin- und Einsatzmittelplanung, wie Wissensverteilung, Stundensatz und Team-Verfügbarkeit, kann die Planung die Zahl der kritischen Konnektoren reduzieren oder kritische Konnektoren verlagern. Im Rahmen der Einsatzmittelplanung werden dazu die Aufgaben neu über die Teams verteilt.

**Oder Änderung der Architektur:** Über Änderungen der Architektur kann die Zahl der kritischen Konnektoren reduziert werden, etwa durch allgemeine Reduktion von Konnektoren über Restrukturierungen, denkbar ist es beispielsweise, den Zugang zu einer Komponente über eine Fassaden-Komponente [GHJV94] zu bündeln.

[14] Die Oberflächen-Entwicklung umfasst häufig mehr als 50% des Gesamtaufwands [Sie02b]

**Zurück zur Berechnung der kritischen Konnektoren**

## 8.6 Architekturbasierte Verringerung des Lieferumfangs

### 8.6.1 Weglassen oder verschieben unwichtiger Anwendungsfälle

Idee der architekturbasierten Umfangsverringerung ist es, die Anwendungsfälle $uc \in UC$, welche die logische Architektur $\mathcal{L} = (C, A, value, UC, S)$ mit der Konfiguration $C = (V, E, J, H)$ umsetzt, mit Prioritäten zu versehen. Anwendungsfälle $uc_d$ mit geringer Priorität werden weggelassen oder als Risikopuffer an das Ende des Plans verschoben, wenn der geschätzte Aufwand für das Projekt zu groß ist oder der Fertigstellungstermin zu weit in der Zukunft liegt. Das Weglassen eines Anwendungsfalls hat aber nur dann Einfluss auf Termin und Budget, wenn deswegen Arbeitspakete aus der Planung entfallen und diese Arbeitspakete noch nicht bearbeitet wurden.

Zu jeder logischen Architektur werden zu Anwendungsfällen $uc \in UC$ passende Szenarios $S(uc)$ beschrieben. Jedes Szenario ist definiert über die Menge der Konnektoren, welche den Anwendungsfall $uc$ umsetzen $S(uc) \subseteq E$. Die architekturbasierte Umfangsverringerung berechnet die Komponenten und Konnektoren, die nur die gestrichenen Anwendungsfälle implementieren. Diese können aus der Konfiguration $C$ entfernt werden. Die zu den entfallenen Architekturelementen gehörenden Arbeitspakete können aus dem Projektplan entfernt werden und so das benötigte Budget verringern und zu einer Umplanung führen, die den tatsächlichen Fertigstellungstermin positiv beeinflusst.

Diese Art der Umfangsverringerung geschieht iterativ: Wenn beispielsweise beim Streichen der Anwendungsfälle keine Komponenten entfallen, kann dies zu einer Modifikation der Architektur oder zu einer Modifikation der Abbildung der Anwendungsfälle auf die Architektur führen. So ändern sich die Funktionen, welche die logischen Komponenten umsetzen und damit können auch die zugeordneten Arbeitspakete geändert werden.

Die Berechnung der Komponenten und Konnektoren, die wegen der Streichung eines Arbeitspakets entfallen, geschieht in zwei Schritten: Im ersten Schritt werden die überflüssigen Konnektoren entfernt. Das sind alle Konnektoren, die in keinem der Szenarios zu den verbliebenen Arbeitspaketen vorkommen. Im zweiten Schritt werden alle Komponenten entfernt, bei denen weder sie selbst noch eines ihrer Kinder über Konnektoren mit anderen Komponenten verbunden sind.

**Entfernung der überflüssigen Konnektoren**

Sei $\mathcal{L} := (C, A, value, UC, S)$ eine logische Architektur und $UC' \subset UC$ die reduzierte Menge der Anwendungsfälle $UC$. Dann kann die Streichung der Konnektoren über die Auswahlprojektion $\Xi_E$ für Konnektoren und ein entsprechendes Prädikat $\mathcal{Q}_E$ formuliert werden. Das Prädikat trifft nur für die Konnektoren zu, welche in mindestens einem Szenario $S(uc'), uc \in UC'$ aus der reduzierten Anwendungsfallmenge $UC'$ vorkommen:

$$\mathcal{Q}_{Ereduce}(e) := \begin{cases} true & : \exists uc \in UC' : e \in S(uc) \\ false & : sonst \end{cases}$$

Damit ergibt sich die um die überflüssigen Konnektoren bereinigte logische Architektur $\mathcal{L}_{Ereduced}$ über:

$$\mathcal{L}_{Ereduced} := \Xi_E(\mathcal{L}, \mathcal{Q}_{Ereduce})$$

#### Entfernung der überflüssigen Komponenten

Im zweiten Schritt werden alle Komponenten entfernt, die nicht mehr selbst oder durch eines ihrer Kinder mit anderen Komponenten über Konnektoren kommunizieren.

$$\mathcal{Q}_{Vreduce}(v) := \begin{cases} true & : \exists e \in E, \exists x \in V : J(e) = (v,x) \vee J(e) = (x,v) \vee \\ & (\exists w \in allparts(C,v) \wedge \exists e \in E, \exists x \in V : J(e) = (w,x) \vee J(e) = (x,w)) \\ false & : sonst \end{cases}$$

Damit ergibt sich die um die überflüssigen Komponenten bereinigte logische Architektur $\mathcal{L}_{Vreduced}$ über:

$\mathcal{L}_{Vreduced} := \Xi_V(\mathcal{L}_{Ereduced}, \mathcal{Q}_{Vreduce})$

Durch Verkettung der beiden Projektionen können überflüssige Konnektoren und Komponenten gemeinsam entfernt werden:

$\mathcal{L}_{reduced-effort} := \Xi_V(\Xi_E(\mathcal{L}, \mathcal{Q}_{Ereduce}), \mathcal{Q}_{Vreduce})$

#### Beispiel

Die Abbildung 8.10 zeigt ein Beispiel, für ein Zwischenergebnis der architekturbasierten Umfangsreduktion: Auftraggeber, Architekt und Projektleiter haben gemeinsam entschieden, dass der Anwendungsfall zur komfortablen Suche nach Kunden aus der Komponente *Kundenverwaltung* gestrichen wird. Das ist der Anwendungsfall $uc_1$ aus der im vorangegangenen Kapitel definierten Kundenkomponente[15]. Demzufolge entfallen alle Komponenten und Konnektoren, die nur diesen Anwendungsfall implementieren. In der Abbildung 8.10 sind die entfallenen Architekturelemente ausgegraut.

Die zu den Architekturelementen gehörenden Arbeitspakete können im Plan gestrichen werden. Der Gesamtaufwand reduziert sich um den Aufwand, der diesen Arbeitspaketen (Architekturelementen) zugeordnet ist. Im Beispiel entfallen die Aufwände zur Erstellung der Komponente *Kunden Suche Dialog* und der entsprechenden Konnektoren.

In der Abbildung 8.10 ist nach der Streichung weiteres Sparpotential sichtbar, wenn eine Architekturänderung vorgenommen wird. Die Komponente *Kundenverwaltung Fassade* verwendet als einzige noch die Komponente *Kunden Suche A-Fall*. Wenn die von der Fassade benötigten Funktionen in die Komponente *Kunden Pflege A-Fall* integriert werden, kann auch diese Komponente entfallen, dies ist in Abbildung 8.11 dargestellt.

Nach der Architekturänderung und der Streichung der Komponente sind die von der Änderung betroffenen Komponenten und Konnektoren neu zu schätzen, da etwa die Ergänzung von Funktionen in der Komponente *Kunden Pflege A-Fall* zusätzliche Arbeiten notwendig macht.

### 8.6.2 Das Verfahren zur Umfangsreduktion

Die architekturbasierte Umfangsreduktion ist kein Verfahren, das automatisch angewendet werden kann. Auftraggeber, Projektleiter und Architekt müssen sich auf die streichbaren Anwendungsfälle einigen. Nach der Streichung der entsprechenden Komponenten und Konnektoren ist eine Neuplanung erforderlich, da das Verfahren die freigesetzten Ressourcen und die Terminverschiebungen der anderen Arbeitspakete nicht automatisch berechnet. Das Verfahren zur Umfangsreduktion kann in folgenden Schritten ablaufen:

**Ermittlung des Gesamtaufwands:** Für die Ermittlung des Gesamtaufwands wird über den Summenoperator $\sum$ des Attributs *Plan.Aufwand* (bzw. *Ist.Restaufwand*, vgl. Tabelle 8.1 ) verwendet:

[15] vgl. Abschnitt 7.2

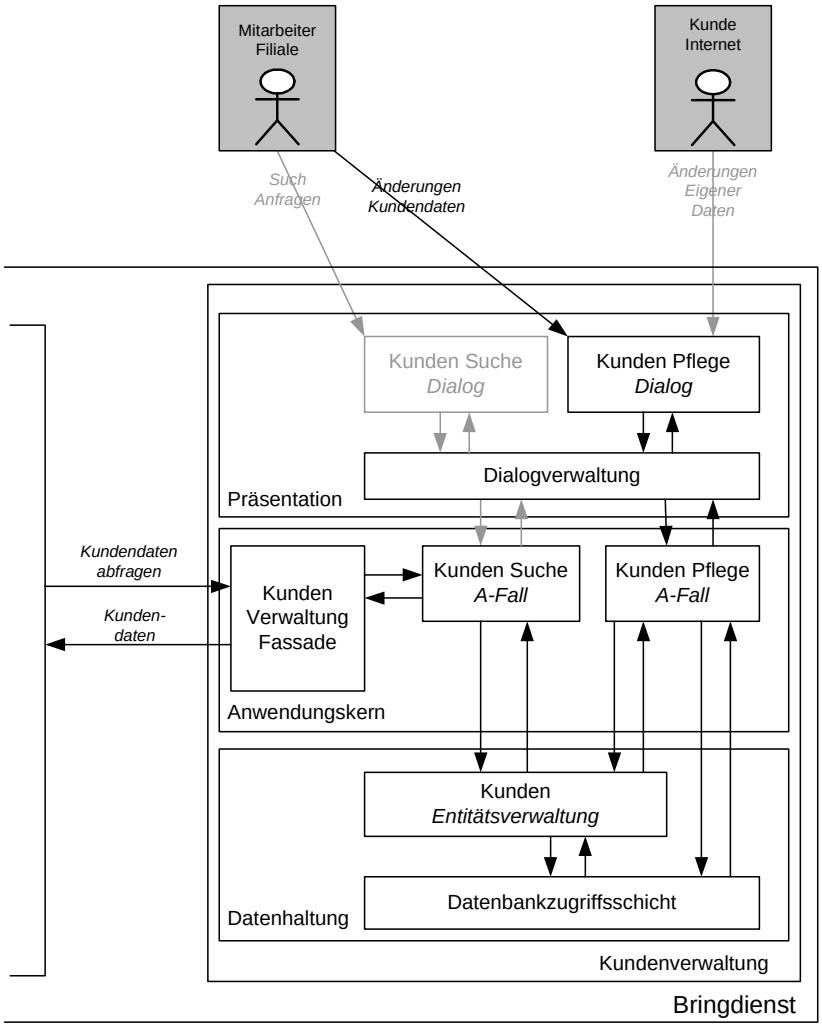


Abbildung 8.10: Beispiel für ein Ergebnis der Umfangsreduktion

Alle Werte werden summiert, die den Komponenten und Konnektoren der logischen Architektur $\mathcal{L}$ zugeordnet sind:

$Effort := \sum_{v \in V}(value(v, Plan.Aufwand)) + \sum_{e \in E}(value(e, Plan.Aufwand))$ für Projekte in der Planung.

**Entscheidung über Umfangsreduktion:** Ist der Gesamtaufwand (immer noch) oberhalb des veranschlagten Budgets, wird das Verfahren zur Umfangsreduktion fortgesetzt. Ist der Gesamtaufwand gering genug, kann die Anpassung der Planung stattfinden.

**Festlegung der zu streichenden Anwendungsfälle:** Architekt und Projektleiter entscheiden zusammen mit dem Auftraggeber, welche der Anwendungsfälle $uc_d \in UC$ gestrichen oder in die nächste Stufe verschoben werden.

**Entfernen überflüssiger Komponenten und Konnektoren:** Konnten Anwendungsfälle gestrichen werden, werden nun die dadurch überflüssig gewordenen Komponenten und Konnektoren aus der Konfiguration entfernt.

**Entweder Anpassung der logischen Architektur:** Konnte durch die Streichung von Anwendungsfällen keine Komponente und kein Konnektor entfernt werden, kann eine Anpassung der logischen Architektur zum Erfolg führen: Dabei wird die Architektur derart geändert oder verfei-

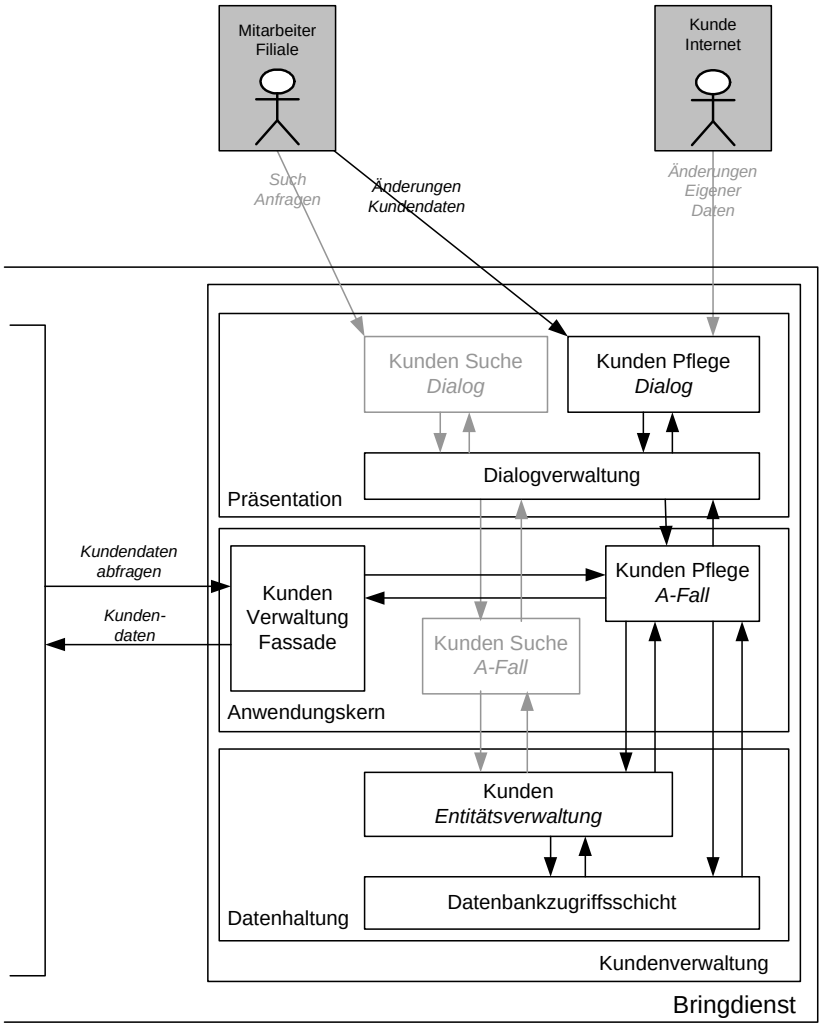


Abbildung 8.11: Beispiel für das Ergebnis einer Architekturänderung zusammen mit der Umfangsreduktion

nert, dass Komponenten und/oder Konnektoren eindeutig den zu streichenden Anwendungsfällen über Szenarios zugeordnet werden können.

**Oder Anpassung der Szenarios:** Begleitend zur Modifikation der logischen Architektur, kann die Abbildung $S$ der Anwendungsfälle auf die Konnektormengen angepasst werden. Das entspricht der Verschiebung von Funktionen zwischen logischen Komponenten oder der Streichung von Teilfunktionen innerhalb eines Anwendungsfalls.

**Anpassung der Planung:** Alle Arbeitspakete, die in Relation zu den gestrichenen Komponenten und Konnektoren stehen, können aus dem Projektplan entfernt oder in die nächste Stufe verschoben werden. Wenn die Architektur geändert wurde und Komponenten oder Konnektoren angepasst worden sind, müssen entsprechend viele neue Arbeitspakete nach dem in Abschnitt 8.4 beschriebenen Synchronisationsverfahren erzeugt werden. Die Erstellungsaufwände für diese Arbeitspakete sind zu schätzen. Dann kann eine Umplanung stattfinden: Die Arbeitspakete können nun neu eingeplant werden und die freigewordenen Ressourcen können neu zugeteilt werden. Erst nach der Umplanung steht daher der Einfluss der Streichungen auf den Endtermin fest.

**Zurück zur Ermittlung des Gesamtaufwands**

Ist das Verfahren erfolgreich abgeschlossen, hat sich der Gesamtaufwand durch die Streichung von Anwendungsfällen und in der Folge durch die Streichung von Komponenten und Konnektoren reduziert.

### 8.6.3 Definition eines Risikopuffers über streichbare Anwendungsfälle

Das Verfahren kann auch zur Optimierung der Planung verwendet werden, ohne dass dabei Arbeitspakete gestrichen werden: Über das Optimierungsverfahren werden Komponenten und Konnektoren identifiziert, die gestrichen werden können. Die zu diesen Komponenten und Konnektoren gehörenden Arbeitspakete werden am Ende einer Stufe oder am Ende des Projektes eingeplant. Sie dienen damit als Risikopuffer, falls Budget und/oder Zeit knapp werden.

Im Verlauf der Optimierung wird möglicherweise auch die logische Architektur so angepasst, dass streichbare Komponenten und Konnektoren vorhanden sind. Damit werden mögliche Risiken, die Budget oder Termine betreffen, zum Architekturtreiber.

## 8.7 Zusammenfassung

Zwischen der logischen Architektur eines IT-Systems und der Planung zu seiner Erstellung bestehen enge Zusammenhänge. Der Zusammenhang zwischen Architektur und Planung wird in diesem Kapitel charakterisiert und Vorschläge für die Generierung eines ersten Projektplans aus einer logischen Architektur und umgekehrt die Abbildung eines Projekt(struktur)plans auf eine vorhandene logische Architektur werden dargestellt. Zusätzlich wird ein pragmatischer Vorschlag gemacht, wie Architektur und Planung fortlaufend synchronisiert werden können.

Zwei Verfahren zur Anwendung des Zusammenhangs zwischen Architektur und Planung werden präsentiert: Das sind die architekturbasierte Umfangsreduktion und die architekturbasierte Reduktion des Abstimmungsbedarfs. Grundlage für die Anwendungen ist jeweils die Architekturtheorie aus Teil II der vorliegenden Arbeit.

# Kapitel 9

# Werkzeugprototyp

Das Werkzeug AutoARCHITECT wird in diesem Kapitel vorgestellt. Die aktuelle Version wird über die Kommandozeile gesteuert. Sie liest Architekturbeschreibungen im AutoFOCUS 2 Format und kann diese Informationen mit Planungsdaten mischen, die aus einem Werkzeug zum Projektmanagement stammen. Daraus erzeugt AutoARCHITECT Architekturschaubilder. Das sind jeweils annotierte gerichtete Graphen. Die Schaubilder werden mit dem Werkzeug GraphViz aus der Beschreibung der gerichteten Graphen erzeugt.

Dieses Kapitel gibt eine Überblick über die Architektur, die Außensicht und die Bedienung von AutoARCHITECT. Zunächst wird ein Überblick mit Hilfe der logischen Architektur in Abschnitt 9.2 gegeben. Die wichtigsten logischen Komponenten und Anwendungsfälle werden vorgestellt. Die Schnittstellen von AutoARCHITECT zu Modellierungswerkzeugen, Projektmanagementwerkzeugen und einem Visualisierungswerkzeug werden in den Abschnitten 9.3 bis 9.5 erläutert. Abschließend erklärt Abschnitt 9.6 die Bedienung des Werkzeugs über die Kommandozeile.

## Übersicht

## 9.1 Das Werkzeug AutoARCHITECT

Das Werkzeug AutoARCHITECT hat das Ziel, Projektleiter und Architekten in ihrer täglichen Zusammenarbeit zu unterstützen: Aus einer ersten Darstellung der Grobarchitektur wird initialer Projektplan generiert. Die Informationen aus der Planung werden anschließend auf die Grobarchitektur zurück projiziert. Auf der Grundlage dieser Informationen werden verschiedene Architektursichten erzeugt. Beispiele für die erzeugten Sichten werden in den Kapiteln 7 und 8 beschrieben. Die fortlaufende Synchronisation der Architektur und der Planung wird ebenfalls unterstützt.

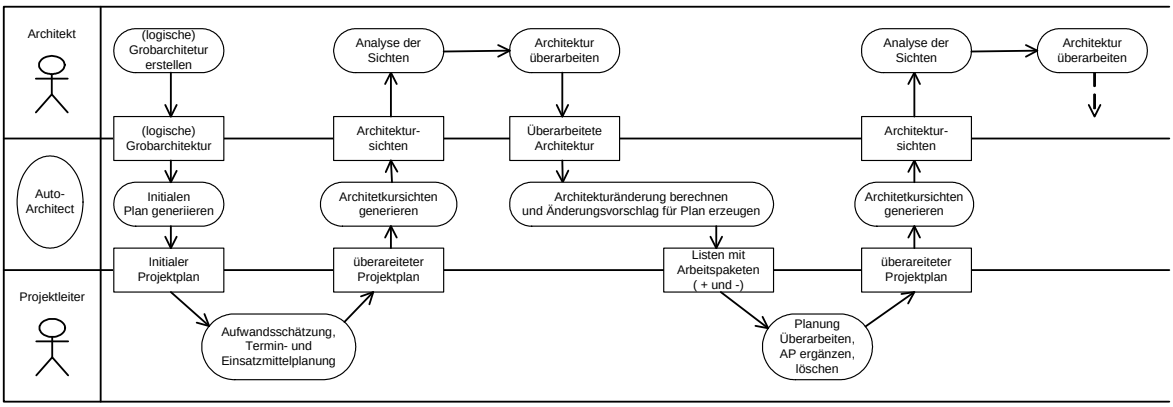


Abbildung 9.1: Zusammenarbeit von Projektleiter und Architekt mithilfe von AutoArchitect

Die Abbildung 9.1 zeigt, wie das Werkzeug in der Zusammenarbeit zwischen Architekt und Projektleiter verwendet werden kann. Die iterative Planung und der iterative Architekturentwurf sind dort im Wechselspiel dargestellt.

## 9.2 Logische Architektur

Die Abbildung 9.2 zeigt die logische Architektur des Werkzeugs AutoARCHITECT. Das Werkzeug fungiert als Datendrehscheibe: Es importiert Architekturbeschreibungen aus AutoFOCUS2 und Projektpläne aus Werkzeugen des Projektmanagements. Es exportiert generierte oder geänderte Projektpläne im CSV[1]-Format und Beschreibungen von Architektursichten in Form gerichteter, annotierter Multigraphen, die mit einem Graph-Visualisierer (hier GraphViz [gra07]) dargestellt werden.

### 9.2.1 Komponenten

AutoARCHITECT hat eine Komponente `Import`, die aus den beiden Komponenten `AF2-Import` und `CSV-Import` besteht. Aus einer Architekturbeschreibung in AutoFOCUS2 wird mithilfe von `AF2-Import` eine interne Darstellung der über AutoFOCUS 2 beschriebenen logischen Architektur erzeugt. Aus einem Projektplan im CSV-Format erzeugt die Komponente `CSV-Import` eine interne Darstellung von Projektplänen. Die Funktionsweise der Import-Komponenten wird in den Abschnitten 9.3 und 9.4 beschrieben.

Die internen Darstellungen der Pläne und der Architektur werden in der Komponente `Model` mit ihren Teilkomponenten `Architecture Models` und `Plan Models` verwaltet.

Über die Komponente `Plan Generator` wird aus der internen Darstellung einer logischen Architektur die interne Darstellung eines initialen Projektplans erzeugt. Die Komponente `View Gene-`

[1]Comma Separated Values

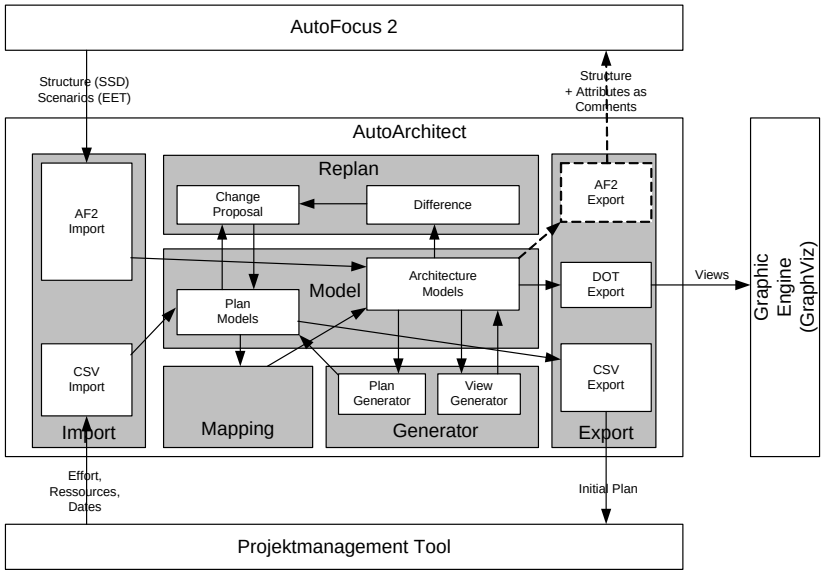


Abbildung 9.2: Logische Architektur von AutoARCHITECT

`rator` erzeugt aus der internen Darstellung einer logischen Architektur gemäß einer vorgegebenen Abbildungsvorschrift (vgl. Kapitel 6 und 7) die Darstellung des Abbildungsergebnisses, die ebenfalls eine logische Architektur beschreibt. Die Komponente `Mapping` bildet die Attribute eines Projektplans auf eine logische Architektur ab. Die Differenz zwischen zwei logischen Architekturen wird mithilfe der Komponente `Difference` berechnet und die Komponente `Change Proposal` ergänzt in einem vorhandenen Projektplan Änderungsvorschläge.

Die Komponente `Export` übersetzt die internen Darstellungen der Architektur und der Planung wieder in externe Darstellungen in Formaten wie CSV, AutoFOCUS 2[2] oder DOT.

### 9.2.2 Anwendungsfälle

AutoARCHITECT ist ein System, das eine Reihe von Eingabedaten ohne Interaktion mit Benutzern in Ausgabedaten transformiert. Es unterstützt zurzeit vier Anwendungsfälle:

**Erzeugung eines initialen Plans:** Aus einer Architekturbeschreibung im AutoFOCUS 2 Format wird ein Projektplan im CSV-Format generiert, dieser kann mit Werkzeugen zum Projektmanagement weiterverarbeitet werden. Dieser Anwendungsfall $uc_1 \in UC$ kann über folgende Konnektormenge beschrieben werden:

$S(uc_1) = \{$ (*Auto*FOCUS2 → *AF2-Import*),(*AF2-Import → Architecture Models*), (*Architecture Models → Plan Generator*), (*Plan Generator → Plan Models*) ,(*Plan Models → CSV-Export*), (*CSV-Export → PM-Tool*) $\}$.

**Erzeugung von Architektursichten:** Eine Architekturbeschreibung im AutoFOCUS 2 Format wird in einen gerichteten Multigraphen im Dot-Format des Werkzeugs GraphViz überführt, dabei können eine oder mehrere Projektionen durchgeführt werden. Ein Anwendungsfall in dem mindestens eine Projektionen durchgeführt wird, kann über die folgende Konnektormenge beschrieben werden:

$S(uc_2) = \{$ (*Auto*FOCUS2 → *AF2-Import*),(*AF2-Import → Architecture Models*), (*Architecture Models → View Generator*), (*View Generator → Architecture Models*), (*Architecture Models → DOT-Export*), (*DOT-Export → GraphViz*) $\}$

[2]Bislang nur geplant

**Erzeugung von Planungs- und Architektursichten:** Eine Architekturbeschreibung im AutoFOCUS 2 Format und die Beschreibung eines Plans im CSV-Format werden zusammengeführt. Mehrere Projektionen können darauf ausgeführt werden, schließlich wird ein gerichteter Multigraphen im Dot-Format erzeugt. Die Zusammenführung nach der zusätzlich mindestens eine Projektionen durchgeführt wird, kann über die folgende Konnektormenge beschrieben werden:

$S(uc_3) = \{$ (*Auto*FOCUS2 → *AF2-Import*), (*AF2-Import* → *Architecture Models*), (*PM-Tool* → *CSV-Import*), (*CSV-Import* → *Plan Models*), (*Plan Models* → *Mapping*), (*Architecture Models* → *Mapping*), (*Mapping* → *Architecture Models*), (*Architecture Models* → *DOT-Export*), (*DOT-Export* → *GraphViz*) $\}$

**Erzeugung von Änderungsvorschlägen für einen Plan:** Zwei Architekturbeschreibungen im AutoFOCUS 2 Format werden gelesen und die Differenz zwischen beiden wird gebildet. Aus der Differenz beider Architekturen werden Änderungsvorschläge für einen Plan gemacht. Dies kann über die folgende Konnektormenge beschrieben werden:

$S(uc_4) := \{$ (*Auto*FOCUS2 → *AF2-Import*), (*AF2-Import* → *Architecture Models*), (*PM-Tool* → *CSV-Import*), (*CSV-Import* → *Plan Models*), (*Plan Models* → *Mapping*), (*Architecture Models* → *Mapping*), (*Mapping* → *Architecture Models*), (*Architecture Models* → *Difference*), (*Difference* → *Change Proposal*), (*Plan Models* → *Change Proposal*), (*Change Proposal* → *Plan Models*), (*Plan Models* → *CSV-Export*), (*CSV-Export* → *PM-Tool*) $\}$.

## 9.3 Schnittstelle zu Modellierungswerkzeugen

AutoFOKUS2 speichert Architekturbeschreibungen im XML-Format. Komponenten, Ports, Kanäle und Kommentare sind als XML Elemente abgelegt. AutoARCHITECT kann dieses Format lesen und die AutoFOCUS2 Darstellung in eine interne Darstellung überführen. Die Tabelle 9.1 gibt an, wie die verschiedenen Elemente aus AutoFOCUS2 auf die Elemente der Architekturtheorie abgebildet werden. Das Metamodell, das den Zusammenhang der AutoFOCUS2 Elemente darstellt, findet sich in Abbildung 3.13.

Anwendungsfälle bzw. Szenarios können mit dem aktuellen Stand der AutoFOCUS2 Implementierung nicht importiert werden. Die Szenarios können zwar über EETs (Extended Event Traces) modelliert werden, zurzeit werden diese Szenarios noch nicht in der AutoFOCUS2 XML-Datei gespeichert. Sichten, die auf Szenarien angewiesen sind (nur A-Architektur) konnten daher nur über Testtreiber in AutoARCHITECT erzeugt werden.

## 9.4 Schnittstelle zu Projektmanagementwerkzeugen

### 9.4.1 Dateiformat

Projektpläne werden häufig mit Projektmanagementwerkzeugen erstellt und gepflegt. Diese Werkzeuge können Pläne (als Gantt-Diagramme) und die Ressourcenauslastung (als Balkendiagramm) grafisch darstellen. Diese Werkzeuge verfügen häufig über eine Dateischnittstelle. Ein Plan wird als Tabelle in eine Textdatei gespeichert. Die Spalten der Tabelle werden jeweils über Kommata getrennt. Dieses Format wird CSV oder *Comma Separated Values* genannt. AutoARCHITECT liest CSV-Dateien ein und generiert neue CSV Dateien. Eine CSV-Datei mit einem Projektplan wird für AutoARCHITECT mit den in Tabelle 9.2 dargestellten Spalten gespeichert:

| AutoFOCUS2-Element | Theorie-Element | Beschreibung |
|---|---|---|
| Komponente (Component) | Komponente | Komponenten werden 1:1 von AutoFOCUS2 in die interne Darstellung übernommen. Als Komponenten IDs werden dabei die Namen der Komponenten zusammen mit den Namen aller Vaterkomponenten verwendet. Ein Beispiel ist in Tabelle 8.3 zu finden. |
| Subcomponents Kompositionsbeziehung | Hierarchierelation | Komponenten können hierarchisch aus anderen Komponenten aufgebaut sein. Eine Komponente kennt ihre Teilkomponenten über die Subcomponents-Aggregationsbeziehung. Diese Aggregationsbeziehung wird mit der Hierarchierelation dargestellt. Die Komponenten kennen jedoch selbst nicht mehr ihre Teilkomponenten. |
| Kanal (Channel) | Konnektor | Kanäle werden auf Konnektoren abgebildet. Die aktuelle Implementierung von AutoARCHITECT entfernt bei Pfaden von Kanälen, die über die Hierarchie hinauf oder hinab verlaufen, die Zwischenkomponenten. Dies soll die Darstellung durch die Reduktion der Zahl der Konnektoren einfach halten. |
| Port | nicht abgebildet | Die Architekturtheorie enthält (noch) nicht das Konzept von Ports, diese werden nicht abgebildet. Der mögliche Nachrichtenaustausch wird nur über die Kanäle dargestellt. |
| Properties | nicht abgebildet | Properties hätten sich für die Darstellung der Attribute aus der Architekturtheorie geeignet. Um in AutoFOCUS2 entsprechende Eingabemöglichkeiten zu schaffen, hätte jedoch die Implementierung von AutoFOCUS2 angepasst werden müssen. |
| Kommentar | Attribut-Belegungen | Um Attribute in einer Architekturbeschreibung eingeben zu können, werden die Kommentare von Komponenten und Kanälen verwendet. Die Belegung wird im Kommentar-String spezifiziert: `[Attributname]=[Attributwert];`, beispielsweise `SoftwareCategory=A;`. Mehrere Belegungen von Attributen werden jeweils über ein Semikolon getrennt. |

Tabelle 9.1: Abbildung von Architekturbeschreibungen in AutoFOCUS2 nach AutoARCHITECT

### 9.4.2 Abbildung zwischen Komponenten, Konnektoren und Arbeitspaketen

Jeder Komponente und jedem Konnektor ist eine eindeutige Id zugeordnet, die sie/ihn identifiziert. Beide haben zusätzlich einen Namen, der mehrfach vorkommen darf. Die Relation *'ist zugeordnet'* zwischen Arbeitspaketen und Komponenten bzw. Konnektoren wird über die zusätzliche Spalte `Architekturelement` gepflegt. Diese Spalte findet sich in der Dateischnittstelle.

Bei der Generierung eines initialen Projektplans wird diese zusätzliche Spalte generiert. Sie enthält die Id der Komponente oder des Konnektors, zu der bzw. dem das Arbeitspaket etwas beiträgt. Bei der Abbildung der (geänderten) Planungsinformationen auf eine bestehende logische Architektur wird die Id aus der zusätzlichen Spalte verwendet, um die Komponente oder den Konnektor zu identifizieren, der bzw. dem die Informationen aus der Planung zugeordnet werden können.

| Name | Datentyp | Beschreibung |
|---|---|---|
| Nr | natürliche Zahl | Nummer der Vorgänge im Projektplan |
| Name | String | Name des Vorgangs |
| Gliederungs-ebene | natürliche Zahl | Die Gliederungsebene dient dazu, die hierarchische Struktur eines Plans darzustellen. Hat etwa ein Arbeitspaket die Gliederungsebene '2' und das Arbeitspaket in der folgenden Zeile die Ebene '3', so ist das erste Arbeitspaket eine Teilaufgabe(Sammelvorgang), die das zweite Arbeitspaket enthält. |
| Arbeit | Positive Fixpunktzahl, zwei Nachkommastellen, Einheit Tage | Aufwand, der für das Arbeitspaket geplant ist. |
| Dauer | Positive Fixpunktzahl, zwei Nachkommastellen, Einheit Tage | Dauer des Arbeitspakets in Tagen, d.h. die Differenz zwischen Ende und Anfang abzüglich der Wochenenden und Feiertage |
| Anfang | Datum | Geplanter Beginn des Arbeitspakets |
| Ende | Datum | Geplantes Ende des Arbeitspakets |
| Vorgänger | Liste natürlicher Zahlen, Trennsymbol: Semikolon | Die Nummer (siehe Spalte Nr) des vorangehenden Arbeitspakets. Die Spalte Vorgänger spezifiziert damit Anordnungsbeziehungen des Typs Normalfolge[3]. |
| Architektur-element | String | Id der Komponenten, des Konnektors oder des Szenarios, zu welcher bzw. welchem das Arbeitspaket einen Beitrag leistet. |
| Ressource-namen | Liste von Strings, Trennsymbol: Semikolon | Liste mit den Namen der Ressourcen, die diesem Arbeitspaket zugeordnet werden. Die Ressourcen werden ausschließlich als (Entwicklungs-)Teams interpretiert. |

Tabelle 9.2: Spalten der CSV-Datei zum Datenaustausch mit Projektmanagementwerkzeugen

Bei der Generierung des Änderungsvorschlags zu einem existierenden Plan, werden Arbeitspakete, die gelöscht werden können, mit '[Delete]' gekennzeichnet. Arbeitspakete, die neu eingeplant werden können, werden an den Plan angehängt.

## 9.5 Schnittstelle zu Visualisierungswerkzeugen

Das Werkzeug GraphViz [gra07] liest eine formatierte Textdatei ein und erzeugt daraus eine Grafik, die einen gerichteten oder ungerichteten Graphen darstellt. Hierarchische Strukturen können ebenfalls visualisiert werden. Die erstellten Grafiken können beispielsweise in den Formaten `svg`, `gif` oder `jpg` gespeichert werden.

Die folgenden Zeilen sollen einen Eindruck von dem dot-Dateiformat vermitteln. Die Zeilen beschreiben die Bausteinsicht des für den Bringdienst erzeugten Graphen.

```
digraph "" {
label="[Omega]\n Team = München\n Start = 2.10.2006\n End = 19.4.2007\n Effort = 25"
"_17398493"
    [label="[Buchhaltungssystem]\n Team = CO Ltd.\n Start = 16.10.2006\n
    End = 27.10.2006\n Effort = 10"shape=box,color=black,style=filled,fillcolor=lightgray];
"_32741206"
    [label="[KundeInternet]"shape=ellipse,color=black,style=filled,fillcolor=lightgray];
"_22323857"
```

```
    [label="[MitarbeiterFiliale]"shape=ellipse,color=black,style=filled,fillcolor=lightgray];
"_21745561"
    [label="[Bringdienst]\n Team = [Stuttgart, München, Sofia]\n Start = 2.10.2006\n
    End = 8.12.2006\n Effort = 1010"shape=box,color=black,style=solid];
"_23162831"
    [label="[Rechnungsdrucker]"shape=box,color=black,style=filled,fillcolor=lightgray];

"_21745561" -> "_23162831"
   [label="[Rechnung]\n Team = München\n Start = 8.1.2007\n End = 23.2.2007\n
   Effort = 35"color=black,style=solid];
"_32741206" -> "_21745561" [label="[BestellungenInternet]"color=black,style=solid];
"_22323857" -> "_21745561" [label="[BestellungFiliale]"color=black,style=solid];
"_22323857" -> "_21745561" [label="[AenderungArtikel]"color=black,style=solid];
"_32741206" -> "_21745561" [label="[KundenDatenInternet]"color=black,style=solid];
"_22323857" -> "_21745561" [label="[AenderungKundenDatenFiliale]"color=black,style=solid];
"_21745561" -> "_17398493"
    [label="[Abrechnung]\n Team = München\n Start = 30.10.2006\n End = 8.12.2006\n
    Effort = 8"color=black,style=solid];
}
```

Die beschriebenen Textdateien werden von AutoARCHITECT erzeugt. Hierarchische Strukturen werden über Subgraphen dargestellt. Die Attribute der Komponenten und Konnektoren werden als Zusätze zu den Namen der erzeugten Knoten und Kanten dargestellt. Die Abbildung 9.3 zeigt eine mit GraphViz erzeugte Grafik zum Bringdienstsystem.

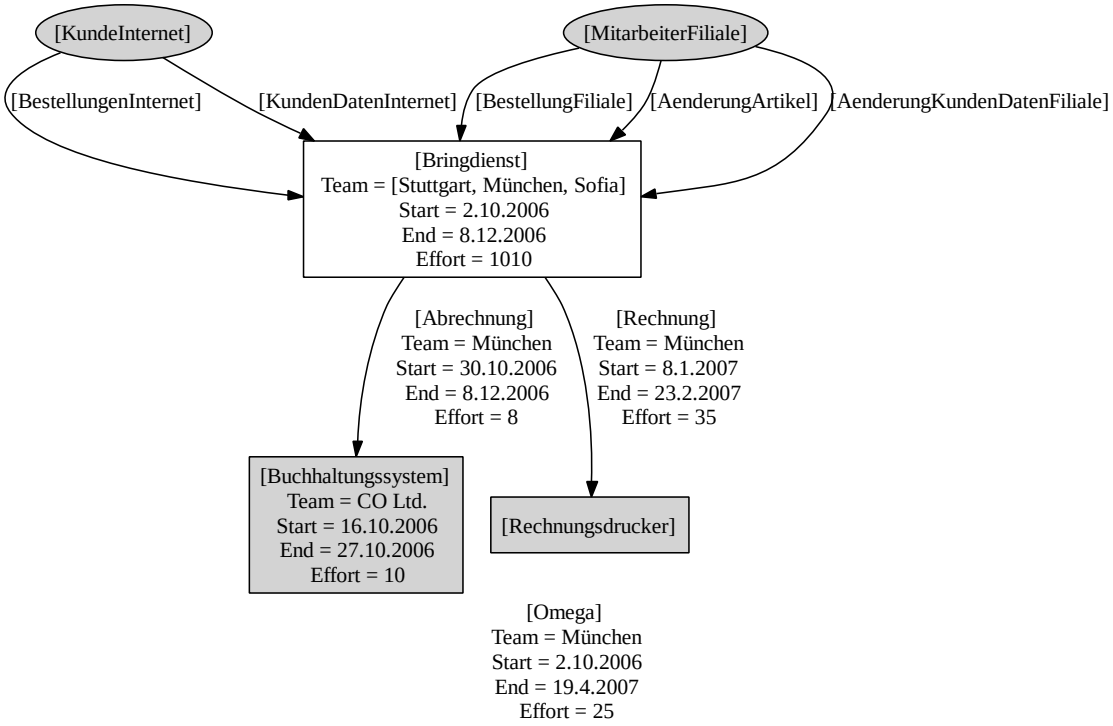


Abbildung 9.3: Von GraphViz erzeugte Architekturgrafik

## 9.6 Bedienung

AutoARCHITECT wird über die Kommandozeile bedient. Mit der Option -h werden alle erlaubten Kommandozeilenparameter dargestellt. AutoARCHITECT bietet drei verschiedene Operationen an mit entsprechenden Parametern:

**-initialPlan:** Aus einer AutoFOCUS2 Architekturbeschreibung wird ein initialier Projektplan erzeugt. Die folgenden Optionen dienen zur Spezifikation von Dateinamen, die für die Generierung eines initialen Projektplans notwendig sind:

- **-a** Architekturbeschreibung in AutoFOCUS2, aus welcher der initiale Plan erzeugt werden soll,
- **-i** CSV - Datei, in welcher der initiale Projektplan gespeichert werden soll.

**-architecturalView:** Aus einer mit AutoFOCUS2 beschriebenen Architektur werden die Architektursichten erzeugt, die in den vorangegangenen Kapiteln beschrieben wurden. Als Generierungsvorschrift können verschiedene Projektionen verwendet werden. Die Architekturbeschreibung kann durch Informationen aus einem Projektplan angereichert werden. Die folgenden Optionen dienen zur Spezifikation von Dateinamen, die für die Generierung von Architektursichten notwendig sind:

- **-a** Architekturbeschreibung in AutoFOCUS2, aus welcher die Sichten erzeugt werden sollen,
- **-p** CSV - Datei, die Daten zur Projektplanung enthält, welche der Architekturbeschreibung zugeordnet werden sollen,
- **-t** Projektionen zur Erzeugung einer Architektursicht (Für die Architektursichten der Kapitel 7 und 8 sind Projektionen auf der Grundlage der Architekturtheorie implementiert. Diese Projektionen werden in Tabelle 9.3 zusammengefasst),
- **-o** Grafikdatei (gif-Format), in der die Sicht gespeichert werden soll.

**-difference:** Aus zwei mit AutoFOCUS2 beschriebenen Architekturen wird die Differenz gebildet. Diese Differenz wird verwendet, um Änderungsvorschläge zu einem Plan zu erzeugen. Mithilfe von dieser Operation soll die Übernahme der Architekturänderungen in die Planung unterstützt werden. Zur Spezifikation der Dateinamen gibt es folgende Optionen:

- **-a** die Architekturbeschreibung in AutoFOCUS2
- **-d** die Architekturbeschreibung in AutoFOCUS2, welche die Änderungen gegenüber der mit -a angegebenen Architekturbeschreibung enthält.
- **-p** CSV - Datei, die Daten zur Projektplanung enthält, welche der mit -a spezifizierten Architekturbeschreibung zugeordnet werden sollen,
- **-e** CSV - Datei, in welcher der Projektplan mit Änderungsvorschlägen gespeichert werden soll.

Der folgende Aufruf erzeugt aus dem AutoFOCUS2 Architekturbeschreibung, die in der Datei `bringdienst.xml` gespeichert ist und dem Projektplan, der in `bringdienst.csv` gespeichert ist, eine Karte der Zuständigkeiten. Die Karte zeigt, welche Teams innerhalb der ersten Dekompositionsebene der logischen Architektur für die einzelnen Komponenten zuständig sind. In der Datei `zustaendig.gif` wird die Karte gespeichert.

```
autoarchitect -architecturalView -a bringdienst.xml -p bringdienst.csv
   -t GLASSBOX RESPONSIBILITY -o zustaendig
```

Das Beispiel zeigt auch, wie die verschiedenen Projektionen kombiniert werden können. Die Projektionen in der Reihenfolge von rechts nach links abgearbeitet. Es wird also zuerst die Karte der Zuständigkeiten erzeugt, diese wird dann zur Bausteinebene hin vergröbert.

## 9.7 Zusammenfassung

Das Werkzeug AutoARCHITECT macht die Architekturtheorie anwendbar, die in der vorliegenden Arbeit vorgeschlagen wird. Dieses Kapitel gibt eine Übersicht über den Leistungsumfang des Werkzeugs und beschreibt die wichtigsten logischen Komponenten. Die Komponenten zum Import der

| Name | Art der Projektion | Beschreibung |
|---|---|---|
| BLACKBOX | vergröbernd | Erstellt ein Umgebungsdiagramm, welches das IT-System zusammen mit Nachbarsystemen und Nutzergruppen zeigt |
| GLASSBOX | vergröbernd | Erstellt ein Bausteindiagramm, welches die erste Dekompositionsebene des IT-Systems darstellt. |
| RESPONSIBILITY | Zusatzinformationen | Erzeugt die Karte der Zuständigkeiten. Die Projektion filtert alle Attribute einer logischen Architektur bis auf das Entwicklungsteam, das die Komponente oder den Konnektor erstellt. Voraussetzung für die Erzeugung der Karte der Zuständigkeiten ist, dass Planungsinformationen in der AutoFOCUS2-Architekturbeschreibung oder aus einem Projektplan verfügbar sind. |
| PLANNING | Zusatzinformationen | Eine Planungssicht wird über diese Projektion erzeugt. Die Planungssicht stellt den Anfangstermin, den Endtermin, den geplanten Aufwand und das zuständige Team bei Komponenten und Konnektoren dar. |
| UNUSED | auswählend | Entfernt alle Komponenten aus der Konfiguration, die nicht über Konnektoren mit anderen Komponenten verbunden sind |
| AARCHITECTIRE | auswählend | Generiert die A-Architektur. Zur Erzeugung dieser Sicht muss das Attribut `SoftwareCategory` in der AutoFOCUS2-Architekturbeschreibung als Kommentar ('SoftwareCategory=A', siehe oben) zu Komponenten und Kanälen hinterlegt sein. |
| TARCHITECTIRE | auswählend | Erzeugt die T-Architektur. Neben dem Attribut `SoftwareCategory` muss auch das Attribut `ComponentType` den Komponenten und Kanälen über Kommentare zugewiesen werden. |
| TIMEMACHINE <Datum> | auswählend | Erzeugt die Zeitpunktsbetrachtung: Komponenten und Konnektoren werden schwarz dargestellt, deren Fertigstellungstermin vor dem angegebenen Datum liegt. Alle anderen Komponenten und Konnektoren werden ausgegraut. |
| CRITICAL | auswählend | Erzeugt die Sicht der kritischen Konnektoren: Es werden alle Komponenten ausgewählt, jedoch nur die kritischen Konnektoren. |

Tabelle 9.3: Vordefinierte Projektionen in AutoARCHITECT

AutoFOCUS2 Architekturbeschreibungen und der Projektpläne bzw. zum Export der Pläne und der Architektursichten spielen dabei eine wichtige Rolle. Ein Überblick über die Bedienung des Werkzeugs bildet den Abschluss dieses Kapitels.

# Kapitel 10

# Zusammenfassung und Ausblick

## 10.1 Ergebnisse

Die vorliegende Arbeit hat drei wesentliche Ergebnisse:

1. Eine Architekturtheorie, mit der logische Architekturen beschrieben und untersucht werden können:

   Die Theorie hat zwei wesentliche Merkmale, die sie von bislang veröffentlichten Ansätzen zur Beschreibung von Architekturen unterscheidet: (1.) Architekturelementen werden Zusatzinformationen zugeordnet, etwa aus der Qualitätssicherung, dem Konfigurationsmanagement oder dem Projektmanagement. Die Zusatzinformationen werden bei der Erzeugung von Architektursichten berücksichtigt. (2.) Zusätzlich beschreiben Nutzungsszenarios, welche Architekturelemente die Anwendungsfälle aus der Anforderungserhebung implementieren (sollen). Über die Zusatzinformationen und die Nutzungsszenarios wird eine Verbindung der logischen Architektur eines IT-Systems zu den funktionalen Anforderungen und zu anderen Disziplinen der Projektdurchführung hergestellt, etwa dem Projektmanagement.

   Die Architekturtheorie stellt Projektionen bereit, mit denen eine logische Architektur in eine andere transformiert wird. Zusatzinformationen, Nutzungsszenarios und Projektionen werden zur Erzeugung von Architektursichten und für das architekturzentrische Projektmanagement verwendet.

2. Eine neuartige Systematik für Architektursichten und Vorschriften, wie diese mithilfe der Architekturtheorie zu erzeugen sind.

   In diesem Zusammenhang werden mehrere neuartige Architektursichten zur Unterstützung des Projektmanagements vorgeschlagen.

3. Eine formal beschriebene Abbildung zwischen Architekturbeschreibungen und der Arbeitspaketstruktur aus dem Projektmanagement.

   Mehrere Verfahren werden vorgeschlagen, die diesen Zusammenhang verwenden und so die Zusammenarbeit zwischen Architekt und Projektleiter verbessern.

Die Anwendbarkeit der Ergebnisse wird über eine durchgehende Fallstudie -ein betriebliches Informationssystem- illustriert und über eine Werkzeugunterstützung nachgewiesen.

### 10.1.1 Architekturtheorie

Die vorliegende Arbeit erarbeitet eine Architekturtheorie. Logische Architekturen von IT-Systemen werden als gerichtete Multigraphen formalisiert: Komponenten sind die Knoten und die Konnektoren sind die Kanten des Multigraphen. Die hierarchische Struktur der Komponenten wird über eine spezielle Kantenmenge beschrieben. Den Komponenten und Konnektoren werden Zusatzinformationen mithilfe von Attributen zugeordnet. Diese Zusatzinformationen stammen beispielsweise aus der Projektplanung oder dem Projekt-Controlling (etwa Fertigstellungstermine oder Aufwände), beschreiben Qualitätseigenschaften (etwa Ausfall- oder Angriffswahrscheinlichkeiten) oder sind Informationen zur Qualitätssicherung (etwa Qualitätsampel oder -status, Testüberdeckung). Diese Informationen werden zur Definition und Erstellung von Architektursichten genutzt. Eine Architektursicht könnte beispielsweise zu jeder Komponente das zuständige Team anzeigen.

Zusätzlich werden zur logischen Architektur Anwendungsfälle als Kantenmenge (Konnektormenge) modelliert, die Kantenmenge wird Szenario genannt. Szenarios stellen dar, welche Komponenten und Konnektoren an der Erbringung der wichtigsten Anwendungsfälle des IT-Systems beteiligt sind. Dies schafft eine Verbindung zu den funktionalen Anforderungen. Diese Verbindung ist die Grundlage für Verfahren des architekturzentrierten Projektmanagements.

Ziel beim Entwurf der Architekturtheorie ist es, Strukturen von IT-Systemen mit möglichst einfachen Mitteln darstellen und untersuchen zu können. Daher wird die logische Architektur als gerichteter Multigraph modelliert, auf die Darstellung von Ports oder Schnittstellen wurde bewusst verzichtet, diese haben in der technischen Architektur größere Bedeutung.

Zu den logischen Architekturen werden Projektionsfunktionen definiert. Diese dienen zur Erzeugung von Sichten aus einer logischen Architektur. Die Projektionen sind eine formale Beschreibung der Blickwinkel (Viewpoints), welche zur Definition von Architektursichten (Views) verwendet werden. Die definierten Grundprojektionen bilden eine logische Architektur auf eine andere ab. Daher können sie zu komplexeren Projektionen verkettet werden. Die Projektionen berücksichtigen auch die Attribute, welche den Komponenten und Konnektoren zugeordnet sind und ebenso die Szenarios.

### 10.1.2 Anwendungen der Architekturtheorie

#### Architektursichten

Die vorliegende Arbeit schlägt eine Systematik für Architektursichten vor, darin werden die Sichten nach den drei Kriterien Architekturart (Logisch, Technisch, Implementierung, Verteilung, Laufzeit), Auflösung (Organisation, System, Bausteine) und dargestellter Sachverhalt (Daten, Funktionen, Projektmanagement, Qualitätssicherung, etc.) eingeordnet. Die Systematik verbessert die Einordnung und den Vergleich von Architekturbeschreibungen. Dies erlaubt insgesamt gegenüber der aktuellen Literatur eine deutlich verbesserte Einordnung und Beschreibung von Architektursichten und -blickwinkeln und zeigt zusätzlich eine Reihe pragmatisch nützlicher Sichten auf, die bislang nicht veröffentlicht wurden.

Die automatische Erzeugung verschiedener Architektursichten ist eine Anwendung der Architekturtheorie. Die Projektionen aus der Theorie sind eine formale Beschreibungen von Blickwinkeln (Viewpoints). Mehrere neue Architektursichten werden beispielhaft definiert, das sind unter anderem:

- Eine Planungssicht, die zu den Elementen einer logischen Architektur auch Planungsdaten darstellt,
- Eine Karte der Zuständigkeiten, die zu jeder Komponente und jedem Konnektor aufzeigt, welches Team dafür verantwortlich ist,

- Eine Darstellung kritischer Konnektoren: Kritische Konnektoren verbinden Komponenten, die von zwei verschiedenen Teams entwickelt werden. Die kritischen Konnektoren müssen mit besonderer Sorgfalt definiert werden
- Eine Zeitpunktsbetrachtung: Die Zeitpunktsbetrachtung erleichtert die Planung der Tests und der Integration. Für einen beliebigen Zeitpunkt während der Entwicklung kann dargestellt werden, welche Komponenten und Konnektoren bis dahin laut Plan fertig gestellt und damit integrierbar sind.

Mehrere Blickwinkel aus der Literatur werden mithilfe der Architekturtheorie formalisiert, etwa die von Siedersleben beschriebenen A- und T-Architekturen [Sie04, Sie02a].

#### Architekturzentriertes Projektmanagement

Das architekturzentriertes Projektmanagement ist eine Anwendung der Architekturtheorie. Es nutzt den Zusammenhang zwischen der logischen Architektur und der Arbeitspaketstruktur aus dem Projektmanagement aus: Die Zuordnung von Komponenten der Architektur zu Arbeitspaketen in der Projektplanung (bzw. dem Projekt-Controlling) erlaubt es, den Komponenten die entsprechenden Planungs- und Controlling-Daten zuzuordnen. Dieser Zusammenhang wird über eine Formalisierung der Arbeitspaketstruktur als gerichteter Graph und Abbildungsvorschriften zwischen Arbeitspaket- und Architekturgraph beschrieben. Projektplanung und -durchführung können darüber mit der Architektur abgestimmt und iterativ optimiert werden. Zwei Verfahren zur Optimierung der Planung und der Architektur werden vorgestellt, das sind:

- Architekturbasierte Reduktion des Lieferumfangs: Anwendungsfälle mit geringer Priorität werden identifiziert. Die Theorie berechnet die Komponenten und Konnektoren, welche ausschließlich zu diesen Anwendungsfällen gehören. In der Planung können die zu diesen Komponenten und Konnektoren gehörenden Arbeitspakete als Handlungsspielraum an das Ende einer Stufe gelegt oder ganz weggelassen werden. Zusätzlich berechnet die Theorie die Aufwände, die durch das Weglassen der Anwendungsfälle entfallen.
- Architekturbasierte Optimierung der Aufgabenverteilung in verteilten Projekten: Aufgaben werden so innerhalb eines Projektteams verteilt, dass die Zahl der kritischen Konnektoren möglichst gering wird.

#### Werkzeugunterstützung

Das Werkzeug AutoARCHITECT macht die Architekturtheorie mit ihren beiden Anwendungen in der Generierung von Sichten und der Unterstützung des Projektmanagements anwendbar. Projektleiter und Architekt können dieses Werkzeug nutzen, um Planung und Architektur iterativ aufeinander abzustimmen und zu optimieren. AutoARCHITECT hat folgende Features:

- Generierung eines Projektplans aus einer Architekturbeschreibung
- Generierung von Architektursichten aus einer annotierten Architekturbeschreibung
- Generierung eines Änderungsvorschlags für einen Projektplan, welcher die Änderungen zwischen zwei nacheinander erstellten logischen Architekturen im Rahmen des iterativen Architekturentwurfs verwendet.

Zur Umsetzung seines Funktionsumfangs verfügt AutoARCHITECT über eine Import-Schnittstelle zu einem Modellierungswerkzeug, eine Import/Export-Schnittstelle für Planungs- und Controlling-Daten, diese kann von Projektmanagementwerkzeugen genutzt werden, sowie eine Export-Schnittstelle für ein Visualisierungswerkzeug, zur Darstellung annotierter, gerichteter Graphen.

## 10.2 Übertragbarkeit

### 10.2.1 Auf andere Domänen

Wie in Kapitel 2 ausgeführt, beschäftigt sich die vorliegende Arbeit vorwiegend mit betrieblichen Informationssystemen und Projekten, die solche IT-Systeme erstellen und weiterentwickeln. Die vorgestellte Architekturtheorie macht jedoch keine spezifischen Annahmen über die entwickelten IT-Systeme.

Die Übertragbarkeit der Theorie etwa auf eingebettete Systeme ist wünschenswert: Die Erstellung großer eingebetteter Systeme ist auch eine Herausforderung an die Organisation und das Management der entsprechenden Projekte. Beispielsweise bei der Entwicklung der Netzwerke aus Steuergeräten, wie sie in Oberklasseautomobilen implementiert sind, müssen viele Mitarbeiter und viele Zulieferer koordiniert werden. Eine Karte der Zuständigkeiten, welche die verantwortlichen Zulieferer für jedes Steuergerät anzeigt oder eine Terminsicht, welche die Integrationstermine für die gelieferte Teilsysteme darstellt, kann in diesen komplexen Projekten für alle Beteiligten hilfreich sein.

Beim Entwurf eingebetteter IT-Systeme im Automobil wird auch von logischen Architekturen gesprochen [Sal06]. Logische Architekturen werden dort über Funktionsnetze modelliert. Der Begriff logische Architektur hat in den Domänen Automotive und betriebliche Informationssysteme dieselbe Bedeutung. Ein Unterschied zwischen den Domänen ist jedoch, dass die logische Architektur in der Automotive Industrie in Form der Funktionsnetze deutlich feinteiliger ausgearbeitet wird.

### 10.2.2 Auf andere Architekturarten

Ein IT-System kann über eine Reihe verschiedener Architekturarten beschrieben werden. Neben der logischen Architektur werden häufig eine technische Architektur sowie Implementierungs-, Verteilungs- und Laufzeitarchitektur verwendet. Die Architekturtheorie kann grundsätzlich auf Architekturarten übertragen werden, in denen sich die Konfiguration als gerichteter Multigraph darstellen lässt.

Die technische Architektur ist beispielsweise in der Regel eine Verfeinerung der logischen Architektur, daher kann diese ebenfalls als gerichteter Multigraph dargestellt werden. Jede Software-Komponente wird als Knoten dargestellt und jeder Konnektor bzw. jedes aus einer exportierten Schnittstelle und dazu passenden importierten Schnittstelle bestehende Paar wird als Kante dargestellt. Diese Darstellung ist jedoch aufwändig, da die Schnittstellen(spezifikationen) und eventuell vorhandene Ports nur als Attribute den Komponenten und/oder den Konnektoren zugewiesen werden können. Dasselbe gilt für Beschreibungen des Verhaltens der Komponenten und Konnektoren.

Die Annotation anderer Architekturarten mit Zusatzinformationen unterstützt den Architekten bei seiner Arbeit: So könnten den Quelltexte der Implementierungsarchitektur Informationen wie die Testüberdeckung zugewiesen werden oder den Rechnerknoten in der Verteilungsarchitektur Wahrscheinlichkeiten für Hacker-Angriffe oder Ausfallwahrscheinlichkeiten. Sichten wie die Karte der Zuständigkeiten (vgl. Kapitel 7) können auch für diese Architekturen nützlich sein.

## 10.3 Ausblick

### 10.3.1 Konkrete Syntax zur grafischen Darstellung

Architekturbeschreibungen auf der Grundlage der Architekturtheorie, stellen Informationen bereit, die in einer Architektursicht dargestellt werden sollen. Eine konkrete Syntax, wie sie etwa UML 2 definiert, wird nicht geliefert.

Das Werkzeug AutoARCHITECT visualisiert jede Komponente als Rechteck und jeden Konnektor als Pfeil. Rechtecke und Pfeile werden über einen Layoutalgorithmus automatisch positioniert. Zusatzinformationen werden als Kommentar an Komponenten und Konnektoren geheftet.

Die Definition einer konkreten Syntax, die auch die Zusatzinformationen berücksichtigt, ist ein wichtiger nächster Schritt, um die hier vorgestellten Ergebnisse besser in Software-Entwicklungsprojekten einsetzen zu können. Die konkrete Syntax kann auf der Basis eines UML 2 Projektmanagement-Profils oder auf der Grundlage einfacher grafischer Symbole (Rechteck, Pfeil, Piktogramm) definiert werden. Als Grundlage dafür kann ein Metamodell für logische Architekturen verwendet werden. Kapitel 5 stellt Grundzüge davon vor.

### 10.3.2 Empirische Fundierung des architekturzentrischen Projektmanagements

Das Kapitel 7 macht Vorschläge für Architektursichten, die voraussichtlich für das Management von Projekten nützlich sind. Mehrere Autoren schlagen beispielsweise die Karte der Zuständigkeiten als Architektursicht vor [Kel03, Pau02]. Der Nutzen der Sichten für konkrete Projekte wird in der vorliegenden Arbeit pragmatisch begründet. Er wird jedoch nicht empirisch belegt, etwa durch die Anwendung der beschriebenen Sichten in einem oder mehreren großen Industrieprojekten oder etwa durch eine Befragung erfahrener Projektleiter oder Architekten.

Die beiden vorgestellten Verfahren zur Optimierung der Aufgabenverteilung und zur architekturbasierten Reduktion des Lieferumfangs werden anhand der durchgehenden Fallstudie *Bringdienst* getestet. Der Nutzen beider Verfahren ist offensichtlich. Die Validierung des Nutzens in einem oder mehreren realen Projekten hat noch nicht stattgefunden.

### 10.3.3 Architekturtheorie als Basis für ein Projekt-Repository

Den Architekturelementen einer logischen Architektur, die als gerichteter Multigraph beschrieben wird, können grundsätzlich alle Artefakte zugeordnet werden, die im Rahmen eines Entwicklungsprojektes entstehen. Damit kann die logische Architektur zur Strukturierung des Projekt-Repositories verwendet werden, das die Zusammenarbeit in Entwicklungsteams fördert:

Brügge et al. setzen eine solche Idee im Werkzeug Sysiphus um [BDW06]. Sysiphus repräsentiert in seinem Repository Modelle eines IT-Systems, organisatorische Modelle und Modelle, die aus der Teamarbeit entstehen, in einem einzigen gerichteten Graphen. Damit können beispielsweise der Beschreibung eines Anwendungsfalls, offene Fragen und Probleme ebenso wie Informationen zu dem bearbeitenden Team zugewiesen werden. Anfrageschnittstellen können verschiedene Sichten der Daten im Repository erzeugen.

### 10.3.4 Projektmanagement

In den Kapiteln 7 und 8 werden Sichten und zwei Verfahren zur Unterstützung des Projektmanagements vorgeschlagen. Die Sichten und Verfahren berücksichtigen Termine, Ressourcen und Aufwände (Kosten) über Attribute der Architekturelemente. Die beiden wichtigen Themen *Risiko* und *Qualität* werden nicht betrachtet. Für das Projekt-Controlling werden ebenfalls keine Vorschläge für Sichten und Verfahren gemacht.

#### Risiko

Den Komponenten einer logischen Architektur können im Rahmen des Risikomanagements auch benennbare Risiken aus einer Risikoliste als Attribute zugeordnet werden oder verdichtete Infor-

mationen etwa in Form einer Risikoampel. Mit diesen Informationen können Risiko-Sichten erzeugt werden, die einen Überblick etwa über den Risikostatus des Projekts und über gefährdete Komponenten geben.

#### Qualitätsstatus

Ähnlich kann auch beim Qualitätsstatus verfahren werden, indem den Komponenten der logischen Architektur ein verdichteter Qualitätsstatus, etwa eine Qualitätsampel, als Attribut zugeordnet wird. Denkbar ist auch die Zuordnung von anderen Maßen in Zusammenhang mit dem Qualitätsstatus, etwa der Testüberdeckung der Implementierung, die den logischen Komponenten zugeordnet ist.

#### Konformität zu Referenzarchitekturen

Die Prüfung auf bestimmte Vorgaben einer Referenzarchitektur ist möglich: Strukturen der Referenzarchitektur können als logische Architektur formuliert werden: Etwa die Definition der erlaubten Aufrufbeziehungen innerhalb einer T-Architektur. Über die Faltungs- sowie die Auswahlprojektionen der Theorie werden Zusammenfassungen der logischen Architektur erzeugt, von diesen wird die über die Referenzarchitektur definierte Architektur, wie in Abschnitt 8.3.3 gezeigt, abgezogen. Ergebnis sind alle Konnektoren, welche gegen die Architekturregeln verstoßen.

### 10.3.5 Qualitätsanforderungen

Den Komponenten und Konnektoren innerhalb der logischen Architekturen können für die ganze Komponente oder den ganzen Konnektor geltende Qualitätseigenschaften zugewiesen werden, wie etwa die geschätzte Zuverlässigkeit oder die Angriffswahrscheinlichkeit. Damit sind mithilfe der Architekturtheorie auch Betrachtungen der Sicherheitseigenschaften eines Systems oder seiner Zuverlässigkeit auf der Grundlage logischer Architekturen möglich. Die Anwendung der Architekturtheorie zur Untersuchung von Qualitätseigenschaften eines IT-Systems ist noch offen.

# Teil IV

# Anhang

# Anhang A

# Hinweise zur Notation

Die in dieser Arbeit verwendete Notation entspricht der von Steger [Ste01]:

## A.1 Mengen

Eine Menge ist die Zusammenfassung von paarweise verschiedenen Objekten. Folgende Notation wird verwendet:

- Mengen werden in der vorliegenden Arbeit immer mit Großbuchstaben bezeichnet. $V$,$E$, oder $H$ sind Beispiele.
- $|V|$ bezeichnet die Anzahl der Elemente einer endlichen Menge $V$.
- $\subseteq$ bezeichnet die Teilmengenbeziehung zwischen zwei Mengen, und $\subset$ ist die strikte Teilmengenbeziehung.
- $A \times B$ ist das kartesische Produkt zweier Mengen $A$ und $B$. Es definiert Tupel aus Elementen beider Mengen. Für alle $a \in A$ und alle $b \in B$ ist $(a, b) \in A \times B$.
- Die Potenzmenge $\wp(A)$ ist Menge aller Teilmengen der Menge $A$. Es gilt $\wp(A) := \{B|B \subseteq A\}$.
- Die Menge $\mathbb{N}$ ist die Menge der natürlichen Zahlen ohne 0, $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$.
- Die Menge $\mathbb{B} = \{true, false\}$ die Menge der Booleschen Werte.

## A.2 Relationen

Eine Relation $\mathcal{R}$ zwischen zwei Mengen $A$ und $B$ ist eine Teilmenge des kartesischen Produkts beider Mengen $\mathcal{R} \subseteq A \times B$. Eine Relation $\mathcal{R} \subseteq A \times A$ auf der Menge $A$ kann unter anderem folgende Eigenschaften haben:

- $\mathcal{R}$ ist reflexiv: $\forall a \in A : a\mathcal{R}a$
- $\mathcal{R}$ ist symmetrisch: $\forall a, b \in A : a\mathcal{R}b \Leftrightarrow b\mathcal{R}a$
- $\mathcal{R}$ ist transitiv: $\forall a, b, c \in A : a\mathcal{R}b \wedge b\mathcal{R}c \Rightarrow a\mathcal{R}c$

Die Relation $\mathcal{R}$ heißt Äquivalenzrelation, genau dann wenn sie reflexiv, symmetrisch und transitiv ist.

Eine Äquivalenzrelation $\mathcal{R}$ teilt eine Menge, auf der sie definiert ist, in disjunkte Äquivalenzklassen auf. Eine Äquivalenzklasse ist durch eines ihrer Elemente eindeutig bestimmt. Sei $x \in V$ eine Komponente und $\mathcal{R} \subset V \times V$ eine Äquivalenzrelation. Die Menge der zu $x$ über $\mathcal{R}$ äquivalenten Komponenten wird geschrieben mit:

$[x]_{\mathcal{R}} := \{v \in V | x\mathcal{R}v\}$.

## A.3 Funktion und Prädikate

Eine Relation $\mathcal{R} \subseteq A \times B$ wird Funktion oder Abbildung genannt, wenn jedes Element der Menge $A$ mit genau einem Element der Menge $B$ in Relation steht:

$\forall a \in A : |\{b \in B | (a, b) \in \mathcal{R}\}| = 1$.

Um eine Funktion $f$ zu beschreiben, ist folgende Schreibweise üblich:

$f : A \to B$ und

$a \mapsto f(a)$

Die Menge $f(a) := \{b \in B | b = f(a)\} \subset B$ heißt Bild von $a$. Die Menge $f^{-1}(b) := \{a \in A | f(a) = b\}$ heißt Urbild von $b$.

Ein einstelliges Prädikat $\mathcal{Q}$ auf einer Menge $A$ weist jedem Element der Menge einen booleschen Wert zu, also *true* oder *false*. Ein Prädikat ist in der vorliegenden Arbeit also als Funktion $\mathcal{Q} : A \to \mathbb{B}$. Über Prädikate werden bestimmte Elemente aus einer Menge $A$ ausgewählt: $A' = \{a \in A | \mathcal{Q}(a)\} \subseteq A$.

## A.4 Graphen

In der vorliegenden Arbeit werden zwei Arten von Graphen verwendet. Der hierarchische Aufbau einer logischen Architektur wird über einen Baum dargestellt und die Kommunikationsbeziehungen zwischen den logischen Komponenten über einen Multigraphen. Im Folgenden sollen die wichtigsten Begriffe der Graphentheorie dargestellt werden.

Ein gerichteter Graph $D$ (D für Directed) ist ein Tupel $D = (V, E)$, wobei $V$ eine endliche, nichtleere Menge von Knoten (V für Vertex) und $E$ eine endliche Menge von gerichteten Kanten (E für Edge) ist. Kanten können als 2-Tupel von Knoten ausgedrückt werden: $E \subseteq V \times V$. Die Abbildung A.1 zeigt einen gerichteten Graphen mit $V = \{1, 2, 3, 4, 5, 6, 7\}$ und $E = \{(1, 2), (1, 3), (2, 4), (3, 5), (3, 6), (3, 7)\}$.

In dem gerichteten Graphen aus der Abbildung A.1 gibt es genau einen Knoten $x \in V$, von dem aus alle anderen Knoten über eine Kantenfolge erreichbar sind. Ein solcher gerichteter Graph heißt auch gerichteter Baum. Der Knoten $x$ heißt auch Wurzel des Baumes. In einem gerichteten Baum heisst der Startknoten einer Kante auch Vater(knoten) und der Endknoten heißt auch Kind(knoten). Kindknoten, die denselben Vater haben heißen auch Geschwister. In der Abbildung ist beispielsweise 2 der Vater von 4 und 3 ist der Vater von 5,6 und 7. Die Knoten 5,6 und 7 sind Geschwister.

Ein gerichteter Graph, in dem zwischen zwei Knoten mehrere Kanten in derselben Richtung verlaufen, heißt gerichteter Multigraph. Die Kanten eines Multigraphen können daher nicht mehr über eine Menge von 2-Tupeln von Knoten beschrieben werden. In Multigraphen kann beispielsweise eine Zuordnungsfunktion $J$ (J für Junction) definiert werden, die Kanten die 2-Tupel von Knoten zuordnet, also $J : E \to V \times V$. Ein gerichteter Multigraph wird damit beschrieben durch das Tupel $M = (V, E, J)$. Die Abbildung A.2 stellt ein Beispiel für den Multigraphen

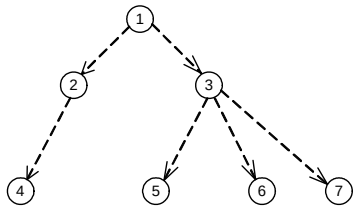


Abbildung A.1: Beispiel für einen gerichteten Baum

$M = (\{5, 6, 7\}, \{a, b, c\}, \{(a, (5, 6)), (b, (6, 7)), (c, (6, 7))\})$

dar.

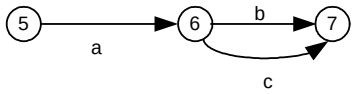


Abbildung A.2: Beispiel für einen gerichteten Multigraphen

In der vorliegenden Arbeit werden Konfigurationen als Kombination gerichteter Bäume und gerichteter Multigraphen dargestellt. Dafür werden zwei Mengen von Kanten definiert. Eine Konfiguration ist ein Tupel $C = (V, E, J, H)$, wobei $V$ die Knotenmenge darstellt, $E$ die Kanten eines Multigraphen modelliert und $J$ die zusätzlich erforderliche Zuordnungsfunktion. Die Menge $H \subset V \times V$ ist die zweite Kantenmenge des kombinierten Graphen. Um die Kantenmengen $E$ und $H$ in Darstellungen unterscheiden zu können, werden die Kanten des gerichteten Baumes $H$ gestrichelt dargestellt. Abbildung A.3 gibt ein Beispiel dafür.

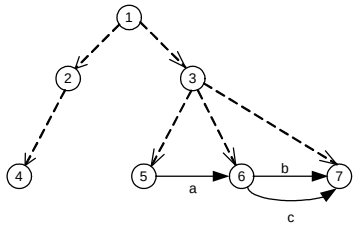


Abbildung A.3: Beispiel für einen Graphen mit zwei Kantenmengen

## A.5 Namen von Variablen

In der vorliegenden Arbeit wird folgende Systematik bei der Benennung von Variablen verwendet, diese wird soweit möglich durchgehalten:

- Komponenten aus der Komponentenmenge $V$ werden benannt mit $v, w, x, y$ und $z$, wobei $v$ und $w$ in der Regel Blattkomponenten sind und $x, y$ und $z$ Vaterkomponenten.
- Konnektoren aus der Konnektormenge $E$ werden benannt mit $e$ und $f$.
- Attribute, welche den Komponenten und Konnektoren zugeordnet werden, werden mit $a$, $b$ oder $c$ benannt.
- Natürliche Zahlen $\in \mathbb{N}_0$ werden benannt mit $i, j, k$ und $n$.

- Relationen werden immer mit $\mathcal{R}$ oder mit $\mathcal{Z}$ und einer tiefer gestellten Beschreibung gekennzeichnet, beispielsweise $\mathcal{R}_{Software-Kategorie}$.
- Prädikate werden immer mit $\mathcal{Q}$ und einer tiefer gestellten Beschreibung gekennzeichnet, beispielsweise $\mathcal{Q}_{Planungsattribute}$.
- $\sigma$ bezeichnet eine Komponente, die das zu erstellende IT-System in einer logischen Architektur $\mathcal{L}$ darstellt.
- $\omega$ bezeichnet eine Pseudokomponente, welche die Umgebung eines IT-Systems mit der logischen Architektur $\mathcal{L}$ darstellt.
- Arbeitspakete werden immer mit einem vorangestellten $w$ bezeichnet, etwa $wp$ oder $wr$.
- $\langle\rangle$ ist ein Symbol für *undefiniert*.

## A.6 Namen von Mengen und Funktionen

In der Tupelschreibweise für Pläne und Architekturen bezeichnen die folgenden Großbuchstaben:

- $C := (V, E, J, H) \in CONFIGURATION$ eine wohlgeformte Konfiguration
- $V \subset COMPONENT$ die Menge von Komponenten einer Konfiguration $C$
- $N \subset COMPONENT, N \subset V$ die Menge von Nachbarsystemen einer Konfiguration $C$
- $U \subset COMPONENT, U \subset V$ die Menge von Nutzergruppen, die ein IT-System verwenden, Nutzergruppen werden zur Vereinfachung der Darstellung als logische Komponenten aufgefasst.
- $E \subset CONNECTOR$ die Menge von Konnektoren einer Konfiguration $C$.
- $B \subset CONNECTOR, U \subset E$ die Menge von Konnektoren in einer Konfiguration $C$, welche die Komponenten des Systems $\sigma$ mit seiner Umgebung aus Nachbarsystemen $N$ und Nutzergruppen $U$ verbindet.
- $J : E \rightarrow V \times V$ eine Zuordnungsfunktion, die Konnektoren einem Komponenten-Paar in einer Konfiguration $C$ zuordnet.
- $H \subset V \times V$ eine Hierarchierelation (zweite Kantenmenge) in einer Konfiguration $C$.
- $A \subseteq ATTRIBUTE$ eine Menge von Attributen, die einer logischen Architektur $\mathcal{L}$ zugeordnet sind.
- $value$ eine Belegungsfunktion, die Komponenten und Konnektoren mit einem Attribut einen Wert zuordnet.
- $UC \subseteq USECASE$ eine Menge von Anwendungsfällen, die einer logischen Architektur $\mathcal{L}$ zugeordnet sind.
- $S : UC \rightarrow \wp(E)$ eine Funktion, die jedem Anwendungsfall eine Menge von Konnektoren zuordnet. Diese Mengen werden auch als Nutzungsszenarios bezeichnet.
- $\mathcal{L} := (C, A, value, UC, S) \in LA$ eine logische Architektur.
- $\mathcal{P} := (WP, WH, WD, wvalue) \in PLAN$ einen Plan.
- $A_P :=$ {*Plan.Anfang, Plan.Ende, Plan.Aufwand, Plan.Dauer, Plan.Verantwortlich* } eine feste Menge von Attributen, die einem Plan $\mathcal{P}$ zugeordnet sind.

- $A_C :=$ { *Ist.Anfang, Ist.Ende, Ist.ErbrachterAufwand, Ist.Restaufwand, Ist.Dauer, Ist.Verantwortlich* }, Attributmenge, die einem Plan im Rahmen des Projekt-Controllings zugeordnet wird.
- $WP \subset WORKPACKAGE$ eine Menge von Arbeitspaketen, die einem Plan $P$ zugeordnet sind.
- $WH \subset WP \times WP$ eine Hierarchierelation eines Planes $P$.
- $WD \subset WP \times WP$ eine Menge von Abhängigkeiten zwischen Arbeitspaketen eines Plans $P$.
- $wvalue$ eine Belegungsfunktion, die Arbeitspaketen eines Plans Belegungen für die Attribute $A_P$ zuordnet.
- $M = (Q, \Sigma, \delta, \lambda, q_0)$ eine Moore-Maschnine
- $Q$ eine Menge von Zuständen einer Moore-Maschine
- $\delta$ eine Funktion, welche den Zustandsübergang einer Moore-Maschine beschreibt
- $\lambda$ eine Funktion, welche die Ausgabe einer Moore-Maschine erzeugt
- $\Sigma$ Ein- und Ausgabealphabet einer Moore-Maschine

Im Rahmen von Erläuterungen wird ein gerichteter Graph mit $G$ bezeichnet.

## A.7 Namen der Grundmengen

Die Elemente von Plänen und logischen Architekturen stammen aus Grundmengen. In der vorliegenden Arbeit werden folgende Grundmengen verwendet:

- $\mathbb{AT}$ Menge aller denkbaren Attributtypen
- $ATTRIBUTE$ Menge aller denkbaren Attribute
- $COMPONENT$ Menge aller denkbaren Komponenten
- $CONFIGURATION$ Menge aller denkbaren wohlgeformten Konfigurationen
- $CONNECTOR$ Menge aller denkbaren Konnektoren
- $\mathbb{ID}$ Menge aller denkbaren Identifier
- $LA$ Menge aller denkbaren logischen Architekturen
- $MESSAGE$ Menge aller denkbaren Nachrichten
- $PLAN$ Menge aller denkbaren Pläne
- $STATE$ Menge aller denkbaren Zustände der logischen Komponenten bzw. Moore-Maschinen
- $USECASE$ Menge aller denkbaren Anwendungsfälle
- $WORKPACKAGE$ Menge aller denkbaren Arbeitspakete

## A.8 Namen von Projektionen

Die nachfolgenden Projektionen sind auf Konfigurationen, logischen Architekturen und Plänen definiert.

Mehrere Projektionen setzen Eigenschaften ihrer Parameter voraus, die über die Grundmengen nicht gewährleistet werden können. Die Projektion $\Psi$ erwartet etwa eine Äquivalenzrelation $\mathcal{R} \subset V \times V$ auf der Menge der Komponenten der übergebenen Konfiguration. Dies wird in der Beschreibung der Projektion nur unzureichend über $\Psi : LA \times \wp(COMPONENT \times COMPONENT) \rightarrow LA \cup \{\langle\rangle\}$ mithilfe von $\wp(COMPONENT \times COMPONENT)$ ausgedrückt. Nach dieser Festlegung könnte auch irgendeine andere Relation $\in \wp(COMPONENT \times COMPONENT)$ übergeben werden und nicht zwingend die geforderte Äquivalenzrelation $\mathcal{R}$. Für diese Fälle ist allen Festlegungen in der Bildmenge das Symbol $\langle\rangle$ (undefiniert) ergänzt worden. Undefiniert ist das Projektionsergebnis immer dann, wenn einer der Parameter die geforderten Voraussetzungen nicht erfüllt.

- $\Phi$ bezeichnet die Vergröberungsprojektion (Aggregation) in der Architekturtheorie
- $\Phi_{\mathbb{N}} : CONFIGURATION \times \mathbb{N}_0 \rightarrow CONFIGURATION$ bezeichnet die Vergröberungsprojektion in der Architekturtheorie, die eine Konfiguration bis auf eine Tiefe von $n \in \mathbb{N}_0$ reduziert.
- $\Phi_{\mathbb{N}} : LA \times \mathbb{N}_0 \rightarrow LA$, wie oben, erweitert auf logische Architekturen.
- $\phi_V : \wp(V) \rightarrow COMPONENT \cup \{\langle\rangle\}$ fasst eine Menge von Komponenten $V_S \subset V$ aus einer Konfiguration $C = (V, E, J, H)$ zu einer neuen Komponente zusammen, die zusammengefassten Komponenten müssen dieselbe Vaterkomponente haben.
- $\phi_E : \wp(E) \rightarrow CONNECTOR \cup \{\langle\rangle\}$ ist eine Zusammenfassungsprojektion für Konnektoren $E_{vw} \subseteq E$, welche dieselben Komponenten innerhalb einer Konfiguration $C = (V, E, J, H)$ verbinden.
- $\phi_{PATH} : (E \times \cdots \times E) \rightarrow CONNECTOR \cup \{\langle\rangle\}$ fasst Pfade aus Konnektoren innerhalb einer Konfiguration $C = (V, E, J, H)$ zu einem neuen Konnektor zusammen, der Anfangs- und Endpunkt des Pfades verbindet.
- $\Psi$ bezeichnet die Zusammenfassungsprojektion in der Architekturtheorie
- $\Psi : CONFIGURATION \times \wp(COMPONENT \times COMPONENT) \rightarrow CONFIGURATION \cup \{\langle\rangle\}$ fasst Komponenten zusammen, die sich in derselben Äquivalenzklasse in Bezug auf die Äquivalenzrelation $\mathcal{R} \subseteq \wp(COMPONENT \times COMPONENT)$ befinden.
- $\Psi : LA \times (COMPONENT \times COMPONENT) \rightarrow LA \cup \{\langle\rangle\}$ wie oben, erweitert auf logische Architekturen.
- $\Xi$ bezeichnet die Auswahlprojektion in der Architekturtheorie.
- $\Xi_V : CONFIGURATION \times (COMPONENT \rightarrow \mathbb{B}) \rightarrow CONFIGURATION$ wählt aus einer Konfiguration über ein einstelliges Prädikat $\mathcal{Q} : COMPONENT \rightarrow \mathbb{B}$ Komponenten aus.
- $\Xi_V : LA \times (COMPONENT \rightarrow \mathbb{B}) \rightarrow LA$ wie oben, erweitert auf logische Architekturen.
- $\Xi_{VS} : CONFIGURATION \times (COMPONENT \rightarrow \mathbb{B}) \times \wp(USECASE) \times (USECASE \rightarrow \wp(CONNECTOR)) \rightarrow CONFIGURATION$ wählt aus einer Konfiguration über ein einstelliges Prädikat $\mathcal{Q}$ Komponenten aus und ersetzt gelöschte Pfade von Konnektoren, die in Szenarios vorkommen durch einen neuen Konnektor.
- $\Xi_{VS} : LA \times (COMPONENT \rightarrow \mathbb{B}) \rightarrow LA$, wie oben, erweitert auf logische Architekturen.
- $\Xi_E : CONFIGURATION \times (CONNECTOR \rightarrow \mathbb{B}) \rightarrow CONFIGURATION$ wählt über ein einstelliges Prädikat auf der Menge der Konnektoren, bestimmte Konnektoren aus.

- $\Xi_E : LA \times (CONNECTOR \to \mathbb{B}) \to LA$, wie oben, erweitert auf logische Architekturen.
- $\Xi_A : LA \times (ATTRIBUTE \to \mathbb{B}) \to LA$ wählt über ein einstelliges Prädikat auf der Menge der Attribute, die Attribute aus, welche in der Bild-Architektur enthalten sein sollen.
- $\Gamma : LA \to PLAN$ bildet logische Architekturen auf Projektpläne ab.
- $\bar{\Gamma} : LA \times PLAN \to LA$ bildet die Informationen aus einem Projektplan auf eine vorhandene logische Architektur ab.
- $\gamma_V : V \to WORKPACKAGE$ ordnet jeder Komponente $v \in V$ aus einer Konfiguration $C = (V, E, J, H)$ ein Arbeitspaket zu.
- $\gamma_I : \wp(H) \to WORKPACKAGE \cup \{\langle\rangle\}$ ordnet bestimmten Hierarchiebeziehungen $h \in H$ zwischen Komponenten einer Konfiguration $C = (V, E, J, H)$ ein Arbeitspaket zu.
- $\gamma_E : E \to WORKPACKAGE$ ordnet jedem Konnektor $e \in E$ aus einer Konfiguration $C = (V, E, J, H)$ ein Arbeitspaket zu.

## A.9 Notation für Begriffsübersichten

Für die Übersichten über die Zusammenhänge zwischen den definierten Begriffen werden Klassendiagramme der UML verwendet. Dieses Verfahren ist üblich zur Darstellung konzeptueller Modelle. Es wird beispielsweise in der Norm ANSI/IEEE 1471 [IEE00] verwendet.

Jeder Begriff wird als Klasse modelliert. Assoziationen und Generalisierungsbeziehungen verdeutlichen die Zusammenhänge zwischen den Begriffen. Im Folgenden wird die Notation an Beispielen erläutert. Die Klassendiagramm-Notation ist im Detail in [OMG03] definiert.

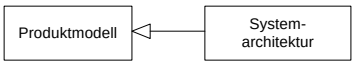


Abbildung A.4: Beispiel für eine Spezialisierung von Begriffen

Die Spezialisierung eines Begriffs wird durch die Vererbung dargestellt. Die Abbildung A.4 zeigt etwa, dass ein eine Systemarchitektur ein Produktmodell ist. Damit hat eine Systemarchitektur alle Eigenschaften eines Produktmodells, diese Eigenschaften kann die Systemarchitektur weiter spezialisieren und erweitern.


Abbildung A.5: Beispiel für eine Assoziation von Begriffen

Mit der Assoziation wird ausgedrückt, dass zwei Begriffe in Beziehung zueinander stehen. Die Beziehung ist über Äritäten und die Benennungen der Assoziationsenden weiter definiert. Die Arität 1 wird in der Regel in Diagrammen weggelassen. Abbildung A.5 zeigt etwa die Beziehung zwischen *Sicht* und *Blickwinkel*: Eine Sicht ist zu nur **einem** Blickwinkel **konform**: Das Assoziationsende hat beim Begriff Blickwinkel die Arität 1 und ist mit *konform zu* bezeichnet. Die Assoziation kann auch in umgekehrter Richtung gelesen werden: Ein Blickwinkel definiert beliebig viele (0 eingeschlossen) Sichten, dies wird durch den Stern ausgedrückt.

# Anhang B

# Abkürzungen

**ACID** Atomicity Consistency Isolation Durability

**ADL** Architecture Description Language

**ARIS** Architektur integrierter Informationssysteme

**ATAM** Architecture Tradeoff Analysis Method

**COTS** Commercial Off The Shelf

**CPM** Critical Path Method

**CRM** Customer Relationship Management

**CRUD** Create Read Update Delete

**CSV** Comma Separated Values

**DIN** Deutsche Industrie Norm

**DMZ** DeMilitarisierte Zone

**EAI** Enterprise Application Integration

**EDI** Electronic Data Exchange

**EET** Extended Event Traces

**EJB** Enterprise Java Beans

**EPK** Ereignisgesteuerte Prozesskette

**ERP** Enterprise Ressource Planning

**GUI** Graphical User Interface

**IDE** Integrated Development Environment

**ISO** International Standards Organization

**IT** Informationstechnologie

**J2EE** Java 2 Enterprise Edition

**JAR** Java Archive

**Java EE** Java Enterprise Edition

**JSP** Java Server Page

**MDA** Model Driven Architecture

**MOF** Meta Object Facility

**MPM** Metra Potential Methode

**MTBF** Mean Time Between Failure

**MTTR** Mean Time To Repair

**OEM** Original Equipment Manufacturer

**OLAP** OnLine Analytical Processing

**OLTP** OnLine Transaction Processing

**OMG** Object Management Group

**OSI** Open Systems Interconnection

**QUASAR** Qualitäts-Software-Architektur

**PERT** Program Evaluation and Review Technique

**PM** Personenmonat

**PSP** Projektstrukturplan

**PT** Personentag

**RMI** Remote Method Invocation

**RTE** Runtime Environment

**RUP** Rational Unified Process

**SSD** System Structure Diagram

**UCM** Use Case Map

**UML** Unified Modelling Language

# Anhang C

# Glossar

**Anwendungsfall:** Ein Anwendungsfall ist eine Folge von Interaktionen zwischen Akteuren und einem IT-System. Er beschreibt das sichtbare Verhalten des IT-Systems aus der Sicht und in der Sprache der Akteure. Akteure können menschliche Benutzer oder Nachbarsysteme sein. Mit einem Anwendungsfall wird durch das IT-System ein für den Anwender sinnvoller Dienst erbracht oder ein benutzbares Ergebnis erzielt. Ein Anwendungsfall ist Teil funktionalen Anforderungen an ein IT-System. (Seite 31)

**Arbeitspaket:** Ein Arbeitspaket beschreibt eine in sich geschlossene Aufgabenstellung innerhalb des Projekts, die von einer einzelnen Person oder einer organisatorischen Einheit (einem Team) bis zu einem festgelegten Zeitpunkt mit definiertem Ergebnis und Aufwand vollbracht werden kann. (Seite 81)

**Architekt (Chefdesigner):** Architekt ist eine Rolle in einem Software-Entwicklungsprojekt. Aufgabe des Architekten ist der Entwurf der System- und der Software-Architektur. Er stimmt die Architektur mit allen Projektbeteiligten ab. Der Architekt ist verantwortlich dafür, dass das fertige IT-System alle funktionalen, wirtschaftlichen und Qualitätsanforderungen erfüllt. Der Architekt wirkt an der Projektplanung, -kontrolle und -steuerung mit. (Seite 25)

**Architekturtreiber:** Anforderungen, die Auswirkungen auf die Gestalt der System- und Software-Architektur haben, werden als Architekturtreiber bezeichnet. (Seite 30)

**Architekturzentriertes Projektmanagement:** Architekturzentriertes Projektmanagement ist eine besondere Form des Projektmanagements, in der die (logische) Architektur eines IT-Systems eine Grundlage der Planung, Steuerung und Kontrolle des Entwicklungsprojektes darstellt. Architekt und Projektleiter arbeiten eng bei der Gestaltung der Architektur und der Planung zusammen. (Seite 170)

**Attribut:** Ein Attribut ist eine Eigenschaft, die einer Komponente, einem Konnektor oder einer Konfiguration zugeordnet werden kann. Ein Attribut hat einen Namen sowie einen Typ (etwa Datum, Wahrscheinlichkeit, Kosten oder String). Für jedes Architekturelement hat das Attribut einen konkreten Wert (eine Belegung) aus dem Wertebereich des Typs. (Seite 49)

**Beschreibung eines Modells:** Die Beschreibung eines Modells ist die Formulierung des Modells oder eines Ausschnitts in einer oder in mehreren Sprachen. Die Beschreibung kann beispielsweise ein Diagramm, ein Text, eine mathematische Formel oder einige Zeilen Quelltext enthalten. Auch bei der physischen Speicherung der Beschreibung in einer oder mehreren Dateien wird hier von der Beschreibung eines Modells gesprochen. (Seite 27)

**Beschreibungselement:** Ein Beschreibungselement ist Teil einer Beschreibung. Es ist in nur einer Sprache abgefasst. Eine Textpassage, ein UML-Diagramm oder eine Java-Klasse sind Beispiele für Beschreibungselemente. (Seite 27)

**Betriebliches Informationssystem:** Ein betriebliches Informationssystem ist ein IT-System. Die Unterstützung und Steuerung betrieblicher Prozesse sind sein Zweck. Das sind primäre Prozesse der Wertschöpfungskette eines Unternehmens wie Beschaffung oder Produktion und auch sekundäre Prozesse wie Personalabrechnung oder Rechnungswesen. Alternative Begriffe sind betriebliches Anwendungssystem und betriebswirtschaftliches Informationssystem. (Seite 21)

**Blickwinkel, Viewpoint:** Ein Blickwinkel legt Verfahren zur Modellierung und Sprachen zur Beschreibung von Modellen fest. Darüber hinaus verkörpert er eine Menge verwandter Interessen von Erstellern und Nutzern der Modelle. Dies grenzt auch die Inhalte der Beschreibung ein, etwa auf Verteilungsaspekte für die Betriebsführung oder die Strukturierung der Quelltexte für die Entwickler. Eine Sicht heißt *konform* zu einem Blickwinkel, wenn sie die Beschreibungssprachen verwendet, über die Modellierungsverfahren entstanden ist und denselben Zwecken dient. (Seite 29)

**Funktionale Anforderung:** Eine funktionale Anforderung ist eine Bedingung oder Eigenschaft, die ein IT-System benötigt, um sein fachliches oder technisches Ziel zu erreichen. Sie bezieht sich nur auf die Außensicht des IT-Systems, wie sie sich Benutzern und Nachbarsystemen darstellt, insbesondere das Verhalten bei korrekten und fehlerhaften Eingaben. Eine Beschreibung der Bedingung oder Eingenschaft wird ebenfalls als funktionale Anforderung aufgefasst. (Seite 30)

**IT-System:** Ein IT-System ist eine Einheit, die aus interagierenden Software- und Hardware-Komponenten besteht. Es existiert zur Erfüllung eines fachlichen oder technischen Ziels. Das IT-System kommuniziert zur Erreichung seines Ziels mit seiner Umgebung und muss den durch die Umgebung vorgegebenen Rahmenbedingungen Rechnung tragen. (Seite 20)

**Komponente, Subsystem:** Eine Komponente ist eine Einheit der Komposition von IT-Systemen und deren Bestandteilen (Software-System, Trägersystem). Sie bietet ihren Funktionsumfang über Schnittstellen ihrer Umgebung an und importiert Schnittstellen der Umgebung, um diesen Funktionsumfang zu erbringen. Die Umgebung besteht aus anderen Komponenten und Benutzern. Sie hat nur definierte Abhängigkeiten zu ihrer Umgebung. Eine Komponente ist innerhalb eines IT-Systems gegen eine andere Komponente mit denselben Schnittstellen austauschbar. Eine Komponente kann aus Hardware und/oder Software bestehen. Die Komponenten auf der ersten Dekompositionsebene eines IT-Systems werden auch Subsysteme genannt. (Seite 46)

**Konfiguration:** Eine Konfiguration ist im Allgemeinen eine Struktur, die aus Komponenten und Konnektoren besteht. Die Struktur kann als gerichteter Graph interpretiert werden, in dem die Komponenten die Knoten und die Konnektoren die Kanten sind. (Seite 48)

**Konnektor:** Ein Konnektor repräsentiert einen Kommunikationskanal, über den zwei oder mehr Komponenten interagieren können. Ein Konnektor kann zwei Komponenten verbinden, (1.) direkt oder (2.) über die Verbindung einer exportierten Schnittstelle mit einer importierten Schnittstelle oder (3.) über zwei verbundene Ports. Konnektoren können beschreiben die Protokolle zur Interaktion von Komponenten, etwa zur transaktionalen oder sicheren Kommunikation. Sie legen damit einen Teil des Verhaltens des aus den verbundenen Komponenten bestehenden Teilsystems fest. (Seite 47)

**Kritischer Konnektor:** Konnektoren werden *kritisch* genannt, wenn die Komponenten, die sie verbinden, von zwei verschiedenen Teams entwickelt werden. Änderungen und Fehlerbehebungen in den später zugeordneten Schnittstellen sind aufwändiger als bei Konnektoren zwischen Komponenten die vom selben Team entwickelt werden. (Seite 186)

**Logische Architektur:** Die logische Architektur ist Teil der Systemarchitektur eines IT-Systems. Sie beschreibt die Dekomposition des IT-Systems in logische Komponenten, dabei ist dessen Funktionsumfang auf logische Komponenten aufgeteilt. Die logischen Komponenten tauschen über logische Konnektoren Nachrichten aus. Die logische Architektur macht keine Aussage darüber, auf welche Weise die logischen Komponenten und Konnektoren in Hardware oder Software implementiert werden. Die logische Architektur ist die notwendige Grundlage für die Projektplanung und die Implementierung. (Seite 53)

**Logische Komponente:** Eine logische Komponente ist eine Komponente. Sie ist eine Einheit des Grobentwurfs und damit auch der Projektplanung. Sie ist Strukturierungsmittel einer logischen Architektur. Dort hat sie eine definierte Aufgabe, diese ist ein Teil der Funktionalität des IT-Systems. Eine logische Komponente hat Kommunikationsbeziehungen zu anderen logischen Komponenten, mit diesen tauscht sie Nachrichten über logische Konnektoren aus. Die Inhalte der Nachrichten werden in der Regel informell spezifiziert. Das Verhalten (die Funktionalität) und Zustand einer logischen Komponente wird bei betrieblichen Informationssystemen typischerweise informell mit natürlicher Sprache beschrieben. Ihre Schnittstellen sind noch indirekt über logische Konnektoren angegeben. (Seite 55)

**Logischer Konnektor:** Ein logischer Konnektor ist ein Konnektor. Er ist eine Einheit des Grobentwurfs und damit auch der Projektplanung. Er beschreibt den Transportkanal für Nachrichten zwischen je zwei logischen Komponenten. Mehrere logische Konnektoren können dieselben zwei Komponenten verbinden. (Seite 55)

**Meilenstein:** Ein Meilenstein ist das Erreichen eines messbaren, bedeutenden Ereignisses im Projekt (z.B. der Abschluss eines Liefergegenstandes, das Ende einer Phase) zu einem bestimmten, geplanten Termin. (Seite 79)

**Modell:** Ein Modell ist eine Beschreibung von Eigenschaften eines Originals. Die Form der Beschreibung ist hier nicht definiert, die Aussage kann beispielsweise ein Idee des Fachbereichs, ein Quadrat auf einem Blatt Papier oder mehrere Sätze in einem Dokument sein. Ein betriebliches Informationssystem oder ein Projekt sind Beispiele für Originale. Das Modell ist eine Abstraktion des Originals: Es stellt einen bestimmten Ausschnitt der Eigenschaften des Originals dar, dieser erfüllt für eine Menge von natürlichen Benutzern und/oder IT-Systemen einen definierten Zweck. Weiterhin muss eine Abbildung zwischen Modell und Original existieren, welche das Modell auf feststellbare Eigenschaften des Originals abbildet. Das Modell kann auch ein gedachtes (mentales) Modell sein. (Seite 26)

**Port:** Ein Port definiert einen Interaktionspunkt einer Komponente mit ihrer Umgebung. Der Port kann sowohl Nachrichten empfangen als auch senden bzw. er kann Operationen für andere Komponenten bereitstellen oder Operationen anderer Komponenten nutzen. Einem Port können daher angebotene (exportierte) und benötigte (importierte) Schnittstellen einer Komponente zugeordnet werden. Eine Komponente kann mehrere Ports besitzen. (Seite 47)

**Produktmodell:** Ein Produktmodell ist ein Modell, dessen Original ein Produkt oder das Modell eines Produktes ist. Ein IT-System ist ein Beispiel für ein Produkt; Seine Software-Architektur ist ein Beispiel für ein Produktmodell. (Seite 26)

**Projekt:** Ein Projekt ist ein Vorhaben, das im Wesentlichen durch die Einmaligkeit seiner Bedingungen in ihrer Gesamtheit gekennzeichnet ist. Ein Projekt ist definiert durch die Zielvorgabe (die geforderte Leistung) und die zeitlichen, finanziellen und personellen Rahmenbedingungen (frei nach DIN 69901 [DIN87c]). (Seite 23)

**Projektleiter** Projektleiter ist eine Rolle in einem Software-Entwicklungsprojekt. Aufgabe des Projektleiters ist die Planung, die Organisation sowie die operative Kontrolle und Steuerung eines Projektes. Er stimmt die Planung mit allen Projektbeteiligten ab. Der Projektleiter ist dafür verantwortlich, dass die Ergebnisse des Projekts zum geplanten Termin, innerhalb des geplanten Budgets und in angemessener Qualität vorliegen. (Seite 24)

**Projektplan:** Der Projektplan ist ein Dokument. Er beschreibt ein geplantes Projekt und dessen Durchführung. Zum Projektplan gehören ein Projektstrukturplan sowie ein entsprechender Termin- und Einsatzmittelplan. Häufig ist auch ein Kostenplan enthalten. (Seite 86)

**Projektstrukturplan (PSP):** Ein Projektstrukturplan (PSP) ist die Darstellung und Strukturierung aller Aufgaben, die zur Erreichung des Projektziels während der Projektdurchführung anfallen. Er beschreibt damit den Projektumfang vollständig: Alle Aufgaben, die nicht im Projektstrukturplan stehen, gehören nicht zum Projekt. (Seite 79)

**Prozessmodell:** Ein Prozessmodell ist ein Modell, dessen Original ein Prozess ist. Beispiel für Prozesse sind Software-Entwicklungsprojekte oder Geschäftsprozesse. (Seite 26)

**Qualität:** *Software-Qualität ist die Gesamtheit der Merkmale und Merkmalswerte eines Softwareprodukts, die sich auf dessen Eignung beziehen, festgelegte oder vorausgesetzte Erfordernisse zu erfüllen* [DIN94].

**Qualitätsanforderung:** Qualitätsanforderungen definieren gewüschte Qualitätsmerkmale eines IT-Systems. Beispiele für Merkmale, zu denen häufig Anforderungen formuliert werden, sind Sicherheit, Verfügbarkeit, Änderbarkeit oder Effizienz. (Seite 32)

**Referenzarchitektur:** Eine Referenzarchitektur ist eine abstrakte Architektur für alle IT-Systeme eines bestimmten Anwendungsbereichs, etwa für alle betrieblichen Informationssysteme. Sie definiert Strukturen und Typen von Komponenten sowie deren erlaubte Interaktionen und Verantwortlichkeiten. (Seite 49)

**Schnittstelle:** Die Schnittstelle einer Komponente ist eine gedachte oder tatsächliche Grenze, über die eine Komponente mit ihrer Umgebung kommuniziert. Eine Schnittstelle enthält Operationen, welche die Komponente für die Umgebung bereitstellt (exportiert) oder von der Umgebung verwendet (importiert). Eine Komponente kann mehrere Schnittstellen haben, die jeweils exportierte oder importierte Operationen zusammenfassen. Die Schnittstellen definieren das Verhalten der Komponente. (Seite 46)

**Semantik der Beschreibung eines Modells:** Um die Aussagen zu verstehen, die in einer Modellbeschreibung enthalten sind, interpretiert ein natürlicher oder maschineller Beobachter die Beschreibung. Das ist beispielsweise ein Compiler, ein Analysewerkzeug, der Projektleiter oder der Architekt. Die Interpretationsvorschrift wird auch als Semantik bezeichnet.

**Sicht, View:** Eine Sicht (View) ist Teil einer Beschreibung eines Modells. Sichten strukturieren Beschreibungen. Eine Sicht bezieht sich in der Regel auf einen Teil eines größeren Modells und zeigt einen zweckgebundenen Ausschnitt. Der Zweck wird durch die Interessen der Ersteller und Nutzer der Sicht bestimmt. (Seite 28)

**Software-Architektur eines IT-Systems:** Die Software-Architektur ist ein Modell eines Software-Systems und beschreibt dessen grundlegende Strukturen. Die Strukturen bestehen aus interagierenden Software-Komponenten, sowie deren sichtbaren Eigenschaften und den Beziehungen zwischen den Software-Komponenten. Die Aufteilung in Software-Komponenten ist eine Voraussetzung für die arbeitsteilige Entwicklung der Software. Die Software-Architektur ermöglicht die Erfüllung von Qualitätsanforderungen und der wirtschaftlichen Anforderungen an das IT-System, dessen Bestandteil das Software-System ist. (Seite 43)

**Software-Komponente** Eine Software-Komponente ist eine Einheit der Komposition von Software-Systemen. Sie bietet ihren Funktionsumfang über Schnittstellen ihrer Umgebung an und importiert Schnittstellen der Umgebung, um diesen Funktionsumfang zu erbringen. Die Umgebung besteht aus anderen Software- und Hardware-Komponenten sowie aus Benutzern. Sie hat höchstens definierte Abhängigkeiten zu ihrer Umgebung. Eine Software-Komponente ist innerhalb eines Software-Systems gegen eine andere Komponente mit denselben Schnittstellen austauschbar. (Seite 45)

**Software-System:** Ein Software-System ist Bestandteil eines IT-Systems. Es implementiert die fachlichen oder technischen Ziele des IT-Systems. Dem Software-System kann eine formale Beschreibung seines Verhaltens zugeordnet werden. Diese wird als Programm bezeichnet. (Seite 20)

**Systemarchitektur eines IT-Systems:** Die Systemarchitektur ist ein Modell des IT-Systems. Sie beschreibt die grundlegenden Strukturen des IT-Systems. Die Strukturen bestehen aus Elementen wie logischen Komponenten, Software- und Hardware-Komponenten, Prozessen, Bibliotheken oder Quelltexten. Die Beziehungen zwischen den Elementen und deren sichtbare Eigenschaften, zu denen auch ihr Verhalten zählt, sind Bestandteil der Strukturen. Die Systemarchitektur ermöglicht die Erfüllung von Qualitätsanforderungen und der wirtschaftlichen Anforderungen an das IT-System und ist Voraussetzung für dessen arbeitsteilige Entwicklung. (Seite 44)

**Szenario:** Ein Szenario beschreibt, welche Komponenten und Konnektoren an der Interaktion eines Benutzers oder Nachbarsystems mit dem IT-System im Rahmen eines Anwendungsfalls beteiligt sind. (Seite 49)

**Trägersystem:** Ein Trägersystem ist Bestandteil eines oder mehrerer IT-Systeme. Es ist die Laufzeitumgebung für Software-Systeme. Das Trägersystem besteht einer oder mehreren Hardwarekomponenten, die über ein Netzwerk verbunden sind und darauf aufbauenden Betriebssystemen und weiteren Laufzeitumgebungen wie virtuellen Maschinen, Datenbanksystemen oder Applikationsservern. Das Trägersystem verwaltet die Daten des IT-Systems, etwa in einem Dateisystem oder einem Datenbanksystem. (Seite 21)

**Umgebung eines IT-Systems:** Zur Umgebung eines IT-Systems zählen andere IT-Systeme (Nachbarsysteme) und Benutzer, die jeweils mit dem IT-System direkt kommunizieren. Ein IT-System und seine Nachbarsysteme können sich ein Trägersystem teilen. (Seite 21)

**Wirtschaftliche Anforderung:** Anforderungen an Kosten, Zeitrahmen und Risiken zum Bau, Betrieb und der Weiterentwicklung von IT-Systemen werden als wirtschaftliche Anforderungen bezeichnet. (Seite 35)

# Tabellenverzeichnis

# Abbildungsverzeichnis

# Literaturverzeichnis


[ACM03] ALUR, DEEPAK, JOHN CRUPI und DAN MALKS: *Core J2EE Patterns*. Prentice Hall, 2003.

[AGM04] AVGERIOU, P., N. GUELFI und N. MEDVIDOVIC: *Software Architecture Description and UML*. In: *UML Modeling Languages and Applications, UML 2004*, Band 3297 der Reihe *LNCS*. Springer Verlag, 2004.

[All97] ALLEN, ROBERT J.: *A Formal Approach to Software Architecture, Ph.D. Thesis*. Technischer Bericht CMU-CS-97-144, Carnegie Mellon University, 1997.

[BBS03] BALSAMO, S., M. BERNARDO und M. SIMEONI: *Performance evaluation at the software architecture level*. In: BERNADO, M. und P. INVERADI (Herausgeber): *Formal Methods for Software Architecture*, Band 2804 der Reihe *LNCS*, Seiten 207–258. Springer Verlag, 2003.

[BCK03] BASS, LEN, PAUL CLEMENTS und RICK KAZMAN: *Software Architecture in Practice*. Addison-Wesley, 2. Auflage, 2003.

[BD97] BRÜGGE, BERND und ALLEN H. DUTOIT: *Communication Metrics for Software Development*. In: *ICSE '97: Proceedings of the 19th International Conference on Software Engineering*, Seiten 271–281. ACM Press, 1997.

[BD04] BRÜGGE, BERND und ALLEN H. DUTOIT: *Objektorientierte Softwaretechnik. Mit Entwurfsmustern, UML und Java*. Pearson Studium, 2004.

[BDW06] BRÜGGE, BERND, ALLEN H. DUTOIT und TIMO WOLF: *Sysiphus: Enabling Informal Collaboration in Global Software Development*. In: *Proceedings of the First International Conference on Global Software Engineering*, Oktober 2006.

[Bec99] BECK, KENT: *Extreme Programming Explained: Embrace Change*. Addison-Wesely, 1999.

[Ben06] BENEKEN, GERD: *Referenzarchitekturen*, In [RH06] Handbuch Software-Architektur, Seiten 357–370. dpunkt.verlag, 2006.

[BF00] BUNDSCHUH, MANFRED und AXEL FABRY: *Aufwandschätzung von IT-Projekten*. MITP, 2000.

[BHB+03] BENEKEN, GERD, ULRIKE HAMMERSCHALL, MANFRED BROY, MARIA VICTORIA CENGARLE, JAN JÜRJENS, BERNHARD RUMPE und MAURCE SCHOENMAKERS: *Componentware: State of the Art 2003, Background Paper for the Understanding Components Workshop of the CUE Initiative*. http://www.cue-initiative.org/workshops/ 2003venice/ComponentWare.pdf, Oktober 2003.

[BKPS04] BENEKEN, GERD, MARCO KUHRMANN, MARKUS PIZKA und TILMAN SEIFERT: *Workshop Hot Spots der Software-Entwicklung, Referenzarchitekturen*. Bericht VSEK-013/D, VSEK Projekt, Technische Universität München, Juli 2004.

[BM03] BRAUN, PETER und FRANK MARSCHALL: *BOTL - The Bidirectional Object Oriented Transformation Language*. Technischer Bericht TUM-I0307, TU München, 2003.

[BMR+00] BUSCHMANN, F., R. MEUNIER, H. ROHNERT, M. STAL und P. SOMMERLAD: *Pattern-orientierte Software-Architektur*. Addison-Wesley, 1. korrigierter Nachdruck, 2000.

[Boe81] BOEHM, BARRY W.: *Software Engineering Economics*. Prentice Hall, 1981.

[BP03] BAUER, ANDREAS und MARKUS PIZKA: *The Contribution of Free Software to Software Evolution*. In: MIKKONEN, TOMMI, MICHAEL W. GODFREY und MOTOSHI SAEKI (Herausgeber): *Proc. of the Int. Workshop on Principles of Software Evolution (IWPSE)*, Helsinki, Finland, 2003. IEEE Computer Society.

[BR07] BROY, MANFRED und BERNHARD RUMPE: *Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung*. Informatik Spektrum, 30:3–18(16), 2007.

[Bri00] BRITTON, CHRIS: *IT Architectures and Middleware: Strategies for Building Large, Integrated Systems*. Addison-Wesely, 2000.

[Bro87] BROOKS, F.P.: *No Silver Bullet: Essence and Accidents of Software Engineering*. IEEE Computer, 20(4):10–19, 1987.

[Bro03] BROOKS, F.P.: *Vom Mythos des Mann-Monats: Esseys über Software-Engineering (Deutsche Übersetzung des Mythical Man Month-Ausgabe zum 20. Jubiläum der Originalausgabe)*. MITP, 2003.

[BS01] BROY, MANFRED und KETIL STØLEN: *Specification and Development of Interactive Systems*. Springer, 2001.

[BS02] BENEKEN, GERD und MANFRED SCHAMPER: *Komponenten mit J2EE Patterns*. Java Spektrum, (2), 2002.

[BSB+04] BENEKEN, GERD, TILMAN SEIFERT, NIKO BAEHR, INGE HANSCHKE und OLAF RAUCH: *Referenzarchitekturen und MDA*. In: DADAM, PETER und MANFRED REICHERT (Herausgeber): *INFORMATIK 2004 - Informatik verbindet, Band 2, Beiträge der 34. Jahrestagung der Gesellschaft für Informatik e.V. (GI), Ulm, 20.-24. September 2004*, Band 51 der Reihe *LNI*, Seiten 101–105. GI, 2004.

[BT06] BYELAS, H. und A. TELEA: *Visualization of Areas of Interest in Software Architecture Diagrams*. In: *SoftVis '06: Proceedings of the 2006 ACM symposium on Software visualization*, Seiten 105–114, 2006.

[Buh98] BUHR, R.J.A.: *Use Case Maps as Architectural Entities for Complex Systems*. IEEE Transactions on Software Engineering, 24(12), 1998.

[Bur02] BURGHARDT, MANFRED: *Einführung in Projektmanagement. Definition, Planung, Kontrolle, Abschluss*. Publicis Corporate Publishing, 2002.

[Car99] CARMEL, ERRAN: *Global Software Teams: Collaborating Across Borders and Time Zones*. Prentice Hall, 1999.

[CBB+03] CLEMENTS, PAUL, FELIX BACHMANN, LEN BASS, DAVID GARLAN, JAMES IVERS, REED LITTLE, ROBERT NORD und JUDITH STAFFORD: *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2003.

[CD94] COOK, STEVE und JOHN DANIELS: *Designing Object Systems: Object-Oriented Modelling With Syntropy*. Prentice-Hall, 1994.

[CD01] CHEESMAN, JOHN und JOHN DANIELS: *UML Components, A Simple Process for Specifying Component-Based Systems*. Addison-Wesley, 2001.

[CHA99] CHAOS: *Chaos: A Recipe for Success*. Technischer Bericht, The Standish Group Interantional, Inc., 1999.

[Che76] CHEN, PETER: *The Entity-Relationship Model–Toward a Unified View of Data*. ACM Transactions on Database Systems, 1(1):9–36, 1976.

[Con68] CONWAY, MELVIN: *How Do Committees Invent?* Datamation Magazine, F.D. Thomson Publication, 1968.

[DeM05] DEMARCO, TOM: *Der Termin*. Hanser Verlag, 2005.

[Den91] DENERT, ERNST: *Software-Engineering*. Springer, erster korrigierter Nachdruck, 1991.

[Den93] DENERT, ERNST: *Dokumentorientierte Softwareentwicklung*. Informatik Spektrum, 16:159–164, 1993.

[DIN87a] DIN69900-TEIL1: *Projektwirtschaft: Netzplantechnik, Begriffe*. Technischer Bericht, Beuth-Verlag, Deutsches Institut für Normung e. V., 1987.

[DIN87b] DIN69900-TEIL2: *Projektwirtschaft: Netzplantechnik, Darstellungstechnik*. Technischer Bericht, Beuth-Verlag, Deutsches Institut für Normung e. V., 1987.

[DIN87c] DIN69901: *Projektwirtschaft: Projektmanagement, Begriffe*. Technischer Bericht, Beuth-Verlag, Deutsches Institut für Normung e. V., 1987.

[DIN87d] DIN69902: *Projektwirtschaft: Einsatzmittel, Begriffe*. Technischer Bericht, Beuth-Verlag, Deutsches Institut für Normung e. V., 1987.

[DIN94] DIN66272: *Bewerten von Softwareprodukten - Qualitätsmerkmale und Leitfaden zu ihrer Verwendung, Identisch mit ISO/IEC 9126:1991*. Technischer Bericht, Beuth-Verlag, Deutsches Institut für Normung e. V., 1994.

[DIN97] DIN69905: *Projektwirtschaft: Projektabwicklung, Begriffe*. Technischer Bericht, Beuth-Verlag, Deutsches Institut für Normung e. V., 1997.

[DK75] DEREMER, FRANK und HANS KRON: *Programming-in-the-Large versus Programming-in-the-Small*. In: *Proceedings of the International Conference on Reliable Software*, Seiten 114 – 121, 1975.

[DRC+04] DESOUZA, CLEIDSON R. B., DAVID REDMILES, LI-TE CHENG, DAVID MILLEN und JOHN PATTERSON: *Sometimes you need to see through walls: a field study of application programming interfaces*. In: *CSCW '04: Proceedings of the 2004 ACM conference on Computer supported cooperative work*, Seiten 63–71. ACM Press, 2004.

[DvdHT01] DASHOFY, ERIC M., ANDRÉ VAN DER HOEK und RICHARD N. TAYLOR: *A Highly-Extensible, XML-Based Architecture Description Language*. In: *Proceedings of the Working IEEE/IFIP Conference on Software Architectures (WICSA 2001)*, 2001.

[DvdHT05] DASHOFY, ERIC M., ANDRE VAN DER HOEK und RICHARD N. TAYLOR: *A comprehensive approach for the development of modular software architecture description languages*. ACM Trans. Softw. Eng. Methodol., 14(2):199–245, 2005.

[EHR00] ETZEL, HANS-JOACHIM, HEIDI HEILMANN und REINHARD RICHTER (Herausgeber): *IT-Projektmanagement, Fallstricke und Erfolgsfaktoren*. dpunkt.verlag, 2000.

[ELSW06] ERNST, A., J. LANKES, C. SCHWEDA und A WITTENBURG: *Using Model Transformation for Generating Visualizations from Repository Contents - An Application to Software Cartography*. Technischer Bericht TB0601, Technische Universität München, Institut für Informatik, Lehrstuhl für Informatik 19, 2006.

[ER03] ENDRES, ALBERT und DIETER ROMBACH: *A Handbook of Software and Systems Engineering: Empirical Observations, Laws and Theories*. Addison-Wesely, 2003.

[Erl05] ERL, THOMAS: *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice-Hall, 2005.

[FH00] FAHMY, H. und R. HOLT: *Using Graph Rewriting to Specify Software Architectural Transformations*. In: *Proc. of Automated Software Engineering (ASE 2000)*, Seiten 187–196, 2000.

[FMP99] FRADET, PASCAL, DANIEL LE MÉTAYER und MICHAËL PÉRIN: *Consistency Checking for Multiple View Software Architectures*. In: *Proceedings of the 7th ACM ESEC / FSE'99 conference*, Band 1687 der Reihe *LNCS*, Seiten 410–428. Springer Verlag, 1999.

[Fra03] FRANKEL, DAVID S.: *Model Driven Architecture*. Wiley, 2003.

[FY97] FOOTE, BRIAN und JOSEPH YODER: *Big Ball of Mud*. In: *The 4th Pattern Languages of Programming Conference*, 1997.

[GA03] GARLAND, JEFF und RICHARD ANTHONY: *Large-scale Software Architecture. A Practical Guide Using UML.* Wiley, 2003.

[Gar03] GARLAN, DAVID: *Formal Methods for Software Architectures*, Band 2804 der Reihe *LNCS*, Kapitel Formal Modeling and Analysis of Software Architecture: Components, Connectors, and Events, Seiten 1 – 24. Springer Verlag, 2003.

[GHJV94] GAMMA, E., R. HELM, R. JOHNSON und J. VLISSIDES: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA, 1994.

[GHP99] GRINTER, REBECCA E., JAMES D. HERBSLEB und DEWAYNE E. PERRY: *The Geography of Coordination: Dealing with Distance in R&D Work*. In: *GROUP '99: Proceedings of the international ACM SIGGROUP conference on Supporting group work*, Seiten 306–315. ACM Press, 1999.

[GJM03] GHEZZI, CARLO, MEHDI JAZAYERI und DINO MANDRIOLI: *Fundamentals of Software Engineering*. Prentice Hall, 2. Auflage Auflage, 2003.

[GK00] GARLAN, DAVID und ANDREW KOMPANEK: *Reconciling the Needs of Architectural Description with Object-Modeling Notations.* In: EVANS, A., S. KENT und B. SELIC (Herausgeber): *UML 2000 - The Unified Modeling Language. Advancing the Standard: Third International Conference*, Band 1939 der Reihe *LNCS*, Seiten 498–512, 2000.

[GMW97] GARLAN, DAVID, ROBERT T. MONROE und DAVID WILE: *ACME: An Architecture Description Interchange Language*. In: *Proceedings of CASCON'97*, Seiten 169–183, Toronto, Ontario, November 1997.

[GMW00] GARLAN, DAVID, ROBERT T. MONROE und DAVID WILE: *ACME: Architectural Description of Component-Based Systems*, Kapitel Foundations of Component-Based Systems, Seiten 47–68. Cambridge University Press, 2000.

[Gna05] GNATZ, MICHAEL: *Vom Vorgehensmodell zum Projektplan*. Doktorarbeit, Technischen Universität München, 2005.

[GR95] GROSU, RADU und BERNHARD RUMPE: *Concurrent Timed Port Automata*. Technischer Bericht TUM-I9533, Technische Univerität München, 1995.

[gra07] *Graphviz - Graph Visualization Software*. http://www.graphviz.org/, September 2007.

[GSCK04] GREENFIELD, JACK, KEITH SHORT, STEVE COOK und STUART KENT: *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004.

[GW99] GARLAN, DAVID und Z. WANG: *A case study in software architecture interchange*. In: *Proceedings of Coordination '99*. Springer Verlag, 1999.

[Her02] HERRMANN, STEPHAN: *Views and Concerns and Interrelationships: Lessons Learned from Developing the Multi View Software Engineering Environment PIROL*. Doktorarbeit, Technischen Universität Berlin, 2002.

[HG99a] HERBSLEB, JAMES D. und REBECCA E. GRINTER: *Architectures, Coordination, and Distance: Conway's Law and Beyond.* IEEE Software, 16(5):63–70, 1999.

[HG99b] HERBSLEB, JAMES D. und REBECCA E. GRINTER: *Splitting the Organization and Integrating the Code: Conway's Law Revisited.* In: *Proceedings of the International Conference on Software Engineering*, Seiten 85–95. ACM Press, 1999.

[HHMS04] HINDEL, BERND, KLAUS HÖRMANN, MARKUS MÜLLER und JÜRGEN SCHMIED: *Basiswissen Software-Projektmanagement*. dpunkt.verlag, 2004.

[Hil99] HILLIARD, RICH: *Using the UML for Architectural Description*. In: FRANCE, ROBERT und BERNHARD RUMPE (Herausgeber): *The Unified Modeling Language, Second International Conference, Fort Collins, Proceedings*, Band 1723 der Reihe *LNCS*, Seiten 32–48. Springer, 1999.

[HNS99a] HOFMEISTER, CHRISTINE, ROBERT NORD und DILIP SONI: *Applied Software Architecture*. Addison-Wesley, 1999.

[HNS99b] HOFMEISTER, CHRISTINE, ROBERT L. NORD und DILIP SONI: *Describing Software Architecture with UML.* In: *WICSA1: Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1)*, Seiten 145–160, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V.

[Hol96] HOLT, RIC: *Binary Relational Algebra Applied to Software Architecture.* Technischer Bericht CSRI 345, University of Toronto, 1996.

[HS00] HERZUM, PETER und OLIVER SIMS: *Business Component Factory: A Comprehensive Overview of Component-Based Development for the Enterprise*. Wiley, 2000.

[HT99] HUNT, ANDREW und DAVID THOMAS: *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, 1999.

[HU90] HOPCROFT, JOHN E. und JEFFREY D. ULLMAN: *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*. Addison-Wesely, 1990.

[HW03] HOHPE, GREGOR und BOBBY WOOLF: *Enterprise Integration Patterns*. Addison-Wesely, 2003.

[IBM04] IBM-RATIONAL: *IBM Rational Unified Process Evaluation V6.13*. http://www-306.ibm.com/software/awdtools/rup/, Stand 15.02.2005, 2004.

[IEE00] IEEE: *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*. http://standards.ieee.org/, 2000.

[ISO91] ISO: *Software Engineering : Product Quality :Quality Model. ISO Standard 9126-1991*, 1991.

[Jür05] JÜRJENS, JAN: *Secure Systems Development with UML*. Springer Verlag, 2005.

[JRB99] JACOBSON, IVAR, JAMES RUMBAUGH und GRADY BOOCH: *The Unified Software Development Process*. Object Technology Series. Addison-Wesley, Reading, MA, 1999.

[JRvdL02] JAZAYERI, MEHDI, ALEXANDER RAN und FRANK VAN DER LINDEN: *Software Architecture for Product Families: Principles and Patterns*. Addison-Wesely, 2002.

[Jun94] JUNGNICKEL, DIETER: *Graphen, Netzwerke und Algorithmen*. B-I-Wissenschaftsverlag, 3. Auflage, 1994.

[KB06] KUHRMANN, MARCO und GERD BENEKEN: *Windows Communication Foundation: Konzepte - Programmierung - Migration*. Spektrum Akademischer Verlag, 2006.

[Küd05] KÜDERLI, LUTZ: *Software Architecture Design for Globally Distributed Development Teams*. Diplomarbeit, Technische Universität München, 2005.

[Kel02] KELLER, WOLFGANG: *Enterprise Application Integration. Erfahrungen aus der Praxis.* dpunkt.verlag, 2002.

[Kel03] KELLER, FRANK: *Über die Rolle von Architekturbeschreibungen im Software-Entwicklungsprozess*. Doktorarbeit, Hasso-Plattner-Institut für Softwaresystemtechnik an der Universität Potsdam, 2003.

[Kel06] KELLER, WOLFGANG: *IT-Unternehmensarchitektur – Von der Geschäftsstrategie zur optimalen IT-Unterstützung*. dpunkt.verlag, 2006.

[Koo06] KOORDINIERUNGS- UND BERATUNGSSTELLE DER BUNDESREGIERUNG FÜR INFORMATIONSTECHNIK IN DER BUNDESVERWALTUNG: *V-Modell XT Projektassistent*. http://www.kbst.bund.de, 2006.

[Kru95] KRUCHTEN, PHILIPPE: *Architectural Blueprints - the 4+1 View Model of Software Architecture*. IEEE Software, 12(6):42–50, 1995.

[Kru00] KRUCHTEN, PHILIPPE: *The Rational Unified Process, An Introduction*. Addison-Wesley, 2. Auflage, 2000.

[LA02] LEVI, KEITH und ALI ARSANJANI: *A Goal-driven Approach to Enterprise Component Identification and Specification*. Communications of the ACM, 45(10):45–52, 2002.

[Lak96] LAKOS, JOHN: *Large-Scale C++ Software Design*. Addison-Wesely, 1996.

[Lar05] LARMAN, CRAIG: *UML 2 und Patterns angewendet - Objektorientierte Softwareentwicklung*. Mitp-Verlag, 2005.

[LMW05] LANKES, JOSEF, FLORIAN MATTHES und ANDRÉ WITTENBURG: *Softwarekartographie: Systematische Darstellung von Anwendungslandschaften.* In: FERSTL, OTTO K., ELMAR J. SINZ, SVEN ECKERT und TILMAN ISSELHORST (Herausgeber): *Wirtschaftsinformatik 2005,*, Seiten 1443–1462. Physica-Verlag, 2005.

[Luc95] LUCKHAM, DAVID C. ET AL.: *Specification and Analysis of System Architecture Using Rapide*. IEEE Transactions on Software Engineering, 21:336–355, 1995.

[LWC07] LANGE, CHRISTIAN F. J., MARTIJN A. M. WIJNS und MICHEL R. V. CHAUDRON: *A Visualization Framework for Task-Oriented Modeling Using UML*. In: *HICSS '07: Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, 2007.

[MB02] MALAN, RUTH und DANA BREDEMEYER: *Software Architecture: Central Concerns, Key Decisions*. http://www.bredemeyer.com/pdf_files/ArchitectureDefinition.PDF (21.01.06), 2002.

[McI68] MCILROY, D.: *Mass-produced Software Components*. In: *Software Engineering, NATO Science Committe Report*, Seiten 138–155, 1968.

[MDEK95] MAGEE, JEFF, NARANKER DULAY, SUSAN EISENBACH und JEFF KRAMER: *Specifying Distributed Software Architectures*. In: *Proceedings of the 5th European Software Engineering Conference*, Band 989 der Reihe *LNCS*, Seiten 137 – 153. Springer Verlag, 1995.

[Mer04] MERTENS, PETER: *Integrierte Informationsverarbeitung 1: Operative Systeme in der Industrie*. Gabler, 2004.

[Mil01] MILNER, ROBIN: *Bigraphical Reactive Systems*. In: *International Conference on Concurrency Theory, CONCUR'01*, Band 2154 der Reihe *LNCS*, Seiten 16–35. Springer Verlag, 2001.

[MK96] MAGEE, JEFF und JEFF KRAMER: *Dynamic Structure in Software Architectures*. In: *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineerin*, Seiten 3 – 14, 1996.

[MM00] MALVEAU, RAPHAEL C. und THOMAS J. MOWBRAY: *Software Architect Bootcamp*. Prentice Hall, 2000.

[MR97] MEDVIDOVIC, NENAD und DAVID S. ROSENBLUM: *Domains of Concern in Software Architectures and Architecture Description Languages*. In: *Proceedings of the 1997 USENIX Conference on Domain-Specific Languages*, 1997.

[MRRR02] MEDVIDOVIC, NENAD, DAVID S. ROSENBLUM, DAVID F. REDMILES und JASON E. ROBBINS: *Modeling software architectures in the Unified Modeling Language*. ACM Transactions on Software Engineering and Methodology, 11(1):2–57, 2002.

[MT00] MEDVIDOVIC, NENAD und RICHARD N. TAYLOR: *A Classification and Comparison Framework for Software Architecture Description Languages*. Software Engineering, 26(1):70–93, 2000.

[MW04] MATTHES, F. und A. WITTENBURG: *Softwarekarten zur Visualisierung von Anwendungslandschaften und ihrer Aspekte*. Technischer Bericht, Technische Universität München, Institut für Informatik, Lehrstuhl für Informatik 19, 2004.

[NL05] NOACK, ANDREAS und CLAUS LEWERENTZ: *A space of layout styles for hierarchical graph models of software systems*. In: *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, Seiten 155–164. ACM, 2005.

[Oes01] OESTEREICH, BERND: *Objektorientierte Softwareentwicklung - Analyse und Design mit der UML*. Oldenbourg Verlag, 5. Auflage Auflage, 2001.

[OMG03] OMG: *Unified Modeling Language: Superstructure*. http://www.omg.org, 2003.

[OMG05] OMG: *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*. Technischer Bericht, Object Management Group, 2005.

[Ove04] OVERHAGE, SVEN: *UnSCom: A Standardized Framework for the Specification of Software Components*. In: WESKE, MATHIAS und PETER LIGGESMEYER (Herausgeber): *Object-Oriented and Internet-Based Technologies, 5th Annual International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a NetworkedWorld, Net.ObjectDays 2004, Erfurt, Germany, September 27-30, Proceedings*, Band 3263 der Reihe *LNCS*, Seiten 169–184. Springer Verlag, 2004.

[Par72] PARNAS, DAVID L.: *On the Criteria To Be Used in Decomposing Systems into Modules*. Communications of the ACM, 15(12):1053–1058, Dezember 1972.

[Par74] PARNAS, DAVID L.: *On a 'Buzzword': Hierarchical Structure.* In: *IFIP Congress*, Seiten 336–339, 1974.

[Par94] PARNAS, DAVID L.: *Software Aging*. In: FADINI, BRUNO (Herausgeber): *Proceedings of the 16th International Conference on Software Engineering*, Seiten 279–290, Sorrento, Italy, may 1994. IEEE Computer Society Press.

[Pau02] PAULISH, DANIEL J.: *Architecture-Centric Software Project Management: A Practical Guide*. Addison-Wesely, 2002.

[PBG07] POSCH, TORSTEN, KLAUS BIRKEN und MICHAEL GERDOM: *Basiswissen Softwarearchitektur*. dpunkt.verlag, zweite Auflage Auflage, 2007.

[PMI96] PMI STANDARDS COMMITTEE: *A Guide to the Project Management Body of Knowledge*. Technischer Bericht, Project Management Institute, 1996.

[pro06] *Projekt Magazin, Glossar*. http://www.projektmagazin.de/glossar, April 2006.

[Ran99] RAN, ALEXANDER: *Software isn't Built from Lego Blocks*. In: *SSR '99: Proceedings of the 1999 Symposium on Software Reusability*, Seiten 164–169. ACM Press, 1999.

[RH06] REUSSNER, RALF und WILHELM HASSELBRING (Herausgeber): *Handbuch Software-Architektur*. dpunkt.verlag, 2006.

[RKB95] RUMPE, BERNHARD, CORNEL KLEIN und MANFRED BROY: *Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme – SysLab Systemmodell –*. Technischer Bericht, Technische Universität München, 1995.

[RM00] RECHTIN, EBERHARDT und MARK MAIER: *The Art of Systems Architecting*. CRC Press, 2. Auflage, 2000.

[Roz97] ROZENBERG, GRZEGORZ (Herausgeber): *Handbook of Grammars and Computing by Graph Transformation*. World Scientific Publishing, 1997.

[RW05] ROZANSKI, NICK und EOIN WOODS: *Software Systems Architecture*. Addison-Wesely, 2005.

[SAFW99] SIEDERSLEBEN, JOHANNES, GERHARD ALBERS, PETER FUCHS und JOHANNES WEIGEND: *Segregating the Layers of Business Information Systems*. In: DONOHOE, PATRICK (Herausgeber): *Software Architecture, TC2 First Working IFIP Conference on Software Architecture (WICSA1), 22-24 February 1999, San Antonio, Texas, USA*, 1999.

[Sal06] SALZMANN, CHRISTIAN: *Automotive Software - Methoden und Technologien*. http://www4.in.tum.de/lehre/vorlesungen/ase/ss06/, Sommersemester 2006.

[Sam97] SAMETINGER, JOHANNES: *Software Engineering with Reusable Components*. Springer Verlag, 1997.

[SB05] SEIFERT, TILMAN und GERD BENEKEN: *Evolution and Maintenance of MDA Applications*. In: BEYDEDA, SAMI, MATTHIAS BOOK und VOLKER GRUHN (Herausgeber): *Model-Driven Software Development*, Seiten 269–286. Springer, Berlin, Heidelberg, New York, Juli 2005.

[SBM+06] SANGWAN, RAGHVINDER, MATTHEW BASS, NEEL MULLICK, DANIEL J. PAULISH und JÜRGEN KAZMEIER: *Global Software Development Handbook*. Auerbach Publications, 2006.

[Sch97] SCHEER, AUGUST-WILHELM: *Wirtschaftsinformatik. Referenzmodelle für industrielle Geschäftsprozesse*. Springer, Zweite, durchgesehene Auflage Auflage, 1997.

[Sch01] SCHEER, AUGUST-WILHELM: *ARIS. Modellierungsmethoden, Metamodelle, Anwendungen*. Springer Verlag, vierte Auflage Auflage, 2001.

[Sch04] SCHWERIN, WOLFGANG: *Ein Produktmodell für die Entwicklung verteilter Informationssysteme*. Doktorarbeit, Technische Universität München, 2004.

[Sch05] SCHLOSSER, JOACHIM: *Architektursimulation von verteilten Steuergerätesystemen*. Doktorarbeit, Technische Universität München, 2005.

[Sei08] SEIFERT, TILMAN: *Abhängigkeitsmanagement für Software mit besonderer Betonung von Wartung und Weiterentwicklung (unveröffentlicht)*. Doktorarbeit, Technische Universität München, 2008.

[SG96] SHAW, MARY und DAVID GARLAN: *Software Architecture, Perspectives of an Emerging Discipline*. Prentice Hall, Upper Saddle River, NJ, 1996.

[SGM02] SZYPERSKI, CLEMENS, DOMINIK GRUNTZ und STEPHAN MURER: *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley, 2. Auflage, 2002.

[SH02] STAHLKNECHT, PETER und ULRICH HASENKAMP: *Einführung in die Wirtschaftsinformatik*. Springer, 10. Auflage, 2002.

[Sie02a] SIEDERSLEBEN, JOHANNES (HRSG.): *Quasar: Die sd&m Standardarchitektur – Teil 1 und 2*. Technischer Bericht, sd&m AG, 2002.

[Sie02b] SIEDERSLEBEN, JOHANNES (HRSG.): *Softwaretechnik. Praxiswissen für Softwareingenieure*. Hanser, 2002.

[Sie04] SIEDERSLEBEN, JOHANNES: *Moderne Software-Architektur - Umsichtig planen, robust bauen mit Quasar*. dpunkt.verlag, 2004.

[SNH95] SONI, DILIP, ROBERT L. NORD und CHRISTINE HOFMEISTER: *Software Architecture in Industrial Applications*. In: *International Conference on Software Engineering*, Seiten 196–207, 1995.

[SP97] SZYPERSKI, C. und C. PFISTER: *Workshop on Component-Oriented Programming, Summary*. In: MÜHLHÄUSER, M. (Herausgeber): *Special Issues in Object-Oriented Programming - ECOOP 96, Workshop Reader*, Heidelberg, 1997. dpunkt.verlag.

[SP02] SMOLANDER, KARI und TERO PÄIVÄRINTA: *Describing and Communicating Software Architectures in Practice: Observations ond Stakeholders and Rationale*. In: PIDDUCK, ANNE BANKS, JOHN MYLOPOULOS, CARSON C. WOO und M. TAMER ÖZSU (Herausgeber): *CAiSE 2002 – The Fourteenth International Conference on Advanced Information Systems Engineering*, Band 2348 der Reihe *LNCS*, Seiten 117–133. Springer Verlag, 2002.

[Sta73] STACHOWIAK, HERBERT: *Modelltheorie*. Springer, 1973.

[Sta05] STARKE, GERNOT: *Effektive Software-Architekturen – Ein praktischer Leitfaden*. Hanser, 2. Auflage, 2005.

[Ste01] STEGER, ANGELIKA: *Diskrete Strukturen 1. Kombinatorik, Graphentheorie, Algebra*. Springer Verlag, 2001.

[Stü02] STÜETZLE, RUPERT: *Wiederverwendung ohne Mythos: Empirisch fundierte Leitlinien für die Entwicklung wiederverwendbarer Software*. Doktorarbeit, Technische Universität München, 2002.

[Sun06] SUN MICROSYSTEMS: *JSR 220: Enterprise JavaBeans,Version 3.0, EJB Core Contracts and Requirements*. http://java.sun.com, 02.05.2006.

[Sys01] SYSTEM-LEVEL DESIGN DEVELOPMENT WORKING GROUP: *Model Taxonomy Version 2.1*. Technischer Bericht, VSI Alliance, 2001.

[SZ92] SOWA, JOHN F. und JOHN A. ZACHMAN: *Extending and Formalizing the Framework for Information Systems Architecture*. IBM Systems Journal, September 1992.

[Tau04] TAUBNER, DIRK: *Effizientes Software-Engineering: Vorgehen für wirtschaftliche Projekte*. Sonderausgabe Fachzeitschrift für Information Management & Consulting, Seiten 14–18, Oktober 2004.

[TLTC05] TERMEER, MAURICE, CHRISTIAN F. J. LANGE, ALEXANDRU TELEA und MICHEL R. V. CHAUDRON: *Visual Exploration of Combined Architectural and Metric Information*. In: DUCASSE, STÉPHANE, MICHELE LANZA, ANDRIAN MARCUS, JONATHAN I. MALETIC und MARGARET-ANNE D. STOREY (Herausgeber): *Proceedings of the 3rd International Workshop on Visualizing Software for Understanding and Analysis, VISSOFT 2005, September 25, 2005, Budapest, Hungary*, Seiten 21–26, 2005.

[V-M06] V-MODELL XT: *Das neue V-Modell XT Release 1.2 - Der Entwicklungsstandard für IT-Systeme des Bundes*. Technischer Bericht, Bundesministerium des Inneren, Koordinierungs- und Beratungsstelle der Bundesregierung für Informationstechnik in der Bundesverwaltung, 2006.

[VAC+05] VOGEL, OLIVER, INGO ARNOLD, ARIF CHUGHTAI, EDMUND IHLER, UWE MEHLIG, THOMAS NEUMANN, MARKUS VÖLTER und UWE ZDUN: *Software-Architektur - Grundlagen, Konzepte, Praxis*. Spektrum Akademischer Verlag, 2005.

[WBFG03] WIERINGA, ROEL J., H.M. BLANKEN, M.M. FOKKINGA und P.W.P.J. GREFEN: *Aligning Application Architecture to the Business Context*. In: EDER, JOHANN und MICHELE MISSIKOFF (Herausgeber): *Advanced Information System Engineering, 15th International Conference, CAiSE 2003, Klagenfurt, Austria*, Band 2681 der Reihe *Lecture Notes in Computer Science*, Seattle, Washington, 2003.

[WF02] WERMELINGER, MICHEL und JOSÉ LUIZ FIADEIRO: *A Graph Transformation Approach to Software Architecture Reconfiguration.* Sci. Comput. Program., 44(2):133–155, 2002.

[Wie03] WIERINGA, ROEL: *Design Methods for Reactive Systems: Yourdon, Statemate and the UML*. Morgan Kaufmann Publishers, 2003.

[WM04] WIECZORREK, HANS W. und PETER MERTENS: *Management von IT-Projekten. Von der Planung zur Realisierung*. Springer Verlag, 2004.

[YC79] YOURDON, EDWARD und LARRY CONSTANTINE: *Structured Design*. Prentice-Hall, Englewood Cliffs, NJ, 1979.

[Zac87] ZACHMAN, JOHN A.: *A Framework for Information Systems Architecture*. IBM Systems Journal, IBM Publication G321-5298.38, 26(3):0, 1987.