Diplomarbeit

Adaptive Architekturen für verteilte Systeme

bearbeitet von

Sebastian Klamar

geboren am 17. März 1980 in Dresden

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Lehrstuhl Rechnernetze

DaimlerChrysler AG
Forschungszentrum Ulm
RIC/SA, Software-Architekturen

Hochschullehrer:	Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill
Betreuer:	Dr.-Ing. habil. Thomas Flor (DaimlerChrysler AG) Dipl.-Inf. Thomas Hamann (TU Dresden)
Eingereicht am:	30. September 2004

Für die zwei Frauen in meinem Leben —
meine Mary und meine Mamma

Aufgabenstellung

Nicht nur im Fahrzeug, sondern auch in industriellen Entwicklungs- und Produktionsprozessen bietet die Vernetzung aktiver (z. B. PDAs, Smart Phones, Smart Devices) und passiver (z. B. RFID-Tags) mobiler Komponenten neue Möglichkeiten. So werden dadurch Identifikation, Transparenz, Verfolgbarkeit, Auswertbarkeit und Verteilung von Prozessfunktionen verbessert. Der aktuellen Tendenz in der Erforschung von selbstorganisierenden Software-Architekturen (z. B. Peer-to-Peer-Computing) folgend, soll die Diplomarbeit folgende Untersuchungen zum Gegenstand haben:

- Definition einer Software-Architektur unter den Aspekten Adaptivität und Adaptierbarkeit mit Anforderungsanalyse, Modellen, sowie Komponenten- und Schnittstellenbeschreibung.
- Bewertung einer vorhandenen Lösung anhand der zuvor durchgeführten Anforderungsanalyse (ggf. neues Konzept erstellen bzw. vorhandenes verbessern).
- Nutzbarkeit von Architekturgeneratoren/-konfiguratoren in verteilten Architekturen prüfen und im Konzept berücksichtigen.
- Analyse der Integrationsfähigkeit aktiver und passiver Architekturkomponenten in eine DV-Infrastruktur bzw. ein Fahrzeug.

Als praktischer Teil dieser Arbeit sollen die Ergebnisse der Untersuchungen in eine prototypische Implementierung der entwickelten Architektur einfließen (z. B. basierend auf JXTA-Framework). Anhand eines praktischen Beispiels soll die Fähigkeit zur Selbstorganisation der Peer-to-Peer-Architektur evaluiert werden.

Inhaltsverzeichnis

Aufgabenstellung v

Abkürzungsverzeichnis xi

Abbildungsverzeichnis xvii

Tabellenverzeichnis xxi

1. Einleitung 1

2. Begrifflichkeit 3
- 2.1. Software-Architektur . . . 3
  - 2.1.1. Was ist Software-Architektur? . . . 3
  - 2.1.2. Sichten . . . 4
  - 2.1.3. Architekturbeschreibungssprachen . . . 8
- 2.2. Verteilte Systeme . . . 12
- 2.3. Selbstorganisation . . . 12
- 2.4. Verwandte Themen . . . 13
  - 2.4.1. Autonomic Computing . . . 13
  - 2.4.2. Organic Computing . . . 15
- 2.5. Zusammenfassung . . . 17

3. Adaptive Software-Systeme — eine Begriffsbestimmung 19
- 3.1. Enzyklopädieeinträge und Übersetzungen . . . 19
- 3.2. Beispiele . . . 24
- 3.3. Zwischenergebnis: Bestandteile eines Adaptionsprozesses 25
- 3.4. Definitionen . . . 27
  - 3.4.1. Besonderheit 5. Definition . . . 29
  - 3.4.2. Analyse der übrigen Definitionen . . . 32
- 3.5. Analyse: Auslöser für die Adaption . . . 35
  - 3.5.1. Kontext . . . 35
  - 3.5.2. Betriebsumgebung . . . 39
  - 3.5.3. Einsatzumgebung . . . 43
  - 3.5.4. Zusammenhang . . . 44
- 3.6. Fazit . . . 47
  - 3.6.1. Kontext + X — Informationen für eine Adaption . 47
  - 3.6.2. Adaptierbarkeit und adaptierbare Software . . . 53
  - 3.6.3. Adaptivität und adaptive Software . . . 55
  - 3.6.4. Adaptionsobjekt/-subjekt Software . . . 60

3.6.5. Zusammenhang Adaptivität und Adaptierbarkeit 64
3.6.6. Wer, was, wann, warum Adaption 67
3.6.7. Adaption 69
3.6.8. Thema dieser Arbeit 77
3.7. Adaptionsarten 79
3.8. Zusammenfassung 83

4. Adaptionsmechanismen 87
4.1. Adaption der Software-Architektur 91
4.1.1. Vorbetrachtung 91
4.1.2. Adaption der Anwendungsarchitektur 96
4.1.3. Adaption der Systemarchitektur 105
4.1.4. Anpassung der Adaptionsarchitektur 108
4.1.5. Zusammenfassung bezüglich Adaptivität / Adaptierbarkeit 109
4.2. Adaption der Anwendungsdaten 110
4.2.1. Reduzierung 111
4.2.2. Ersetzung 111
4.2.3. Transformation 112
4.2.4. Anreicherung 115
4.3. Adaption der Kommunikation 115
4.3.1. Übertragung 115
4.3.2. Zwischenspeicherung 117
4.3.3. Verlagerung des Zugriffs 118
4.4. Zusammenfassung 120

5. Ansätze in der Literatur 123
5.1. Spezifikationsmethoden 123
5.1.1. Externer, expliziter Ansatz 124
5.1.2. Selbstorganisation der Architekturkomponenten 130
5.1.3. Zusammenfassung 131
5.2. Kontroll-Theorie 132
5.2.1. Optimalwertsteuerung / offener Regelkreis 134
5.2.2. Rückkoppelungssteuerung / geschlossener Regelkreis 135
5.2.3. Vergleich Rückkoppelungs- und Optimalwertsteuerung 136
5.2.4. Ansatz: Kombination Rückkoppelungs- und Optimalwertsteuerung 136
5.2.5. Ansatz einer selbst-kontrollierenden Architektur 137
5.2.6. Fazit 141
5.3. Reflection 143
5.3.1. Meta-Level-Architektur 145
5.3.2. Reflective Middleware 147
5.3.3. Fazit 153
5.4. Muster für die Adaptivität in Software 156
5.4.1. Blackboards für die Verschmelzung von Signalen 157

5.4.2. Indirekter Aufruf . . . 159
5.4.3. Aufruf per Entscheidungstheorie . . . 160
5.4.4. Beobachten, Diagnose und Wiederherstellung . . 160
5.5. Weitere Ansätze . . . 161
5.5.1. Cactus . . . 162
5.5.2. Adaptionsframework Rainbow . . . 167
5.5.3. Autonomic Computing . . . 170
5.5.4. Fazit . . . 173
5.6. Zusammenfassung . . . 175

6. Empfehlungen für die Software-Architektur 179
6.1. Dynamische Aspekte — Ablaufplan . . . 180
6.1.1. Laufendes System . . . 183
6.1.2. System und Umgebung beobachten . . . 184
6.1.3. Beobachtungen auswerten . . . 184
6.1.4. Änderungen planen und formulieren . . . 184
6.1.5. Änderungen analysieren . . . 186
6.1.6. Änderungsskripte in systemspezifische Operationen übersetzen . . . 187
6.1.7. Herbeischaffen der an der Adaption beteiligten Elemente . . . 187
6.1.8. Deployen der Änderungsskripte in dem verteilten System . . . 188
6.1.9. Koordinierte Ausführung der Änderungen im System (Adaption) . . . 188
6.1.10. Adaptionsregeln anpassen . . . 189
6.2. Statische Aspekte — Infrastruktur . . . 189
6.2.1. Informationslieferanten — Monitore . . . 189
6.2.2. Gauges . . . 194
6.2.3. Komponentendatenbank für die Anwendungsarchitektur . . . 195
6.2.4. Komponentendatenbank für die Adaptionsarchitektur . . . 198
6.2.5. Alternative Verarbeitungsknoten . . . 198
6.2.6. Spezifikationsdatenbank für Komponenten . . . 199
6.2.7. Planer . . . 201
6.2.8. Analysierer . . . 202
6.2.9. Datenbank für die Adaptionsregeln . . . 202
6.2.10. Spezifikationsdatenbank für die Adaptionsziele . 202
6.2.11. Konfigurationsmanager . . . 203
6.2.12. Koordinierer einer Adaption . . . 205
6.2.13. Worklets . . . 209
6.2.14. Effektoren . . . 209
6.2.15. Übersetzer für die Spezifikationssprachen und Mapping-Datenbank . . . 210
6.3. Evaluation der Empfehlungen . . . 211
6.3.1. Anwendungsfall 1 — beliebiges, verteiltes System 211

6.3.2. Anwendungsfall 2 — Cactus 212
6.3.3. Anwendungsfall 3 — Cache-Policy 214
6.3.4. Anwendungsfall 4 — frei verfügbare Währungsrechner 217
6.4. Fazit 219

7. Zusammenfassung **223**

A. Mindmap Diplom **229**

B. Mindmap Adaptionsmechanismen **231**

C. Inhalt der CD **233**

Literaturverzeichnis **235**

Selbstständigkeitserklärung **267**

Abkürzungsverzeichnis

4GL	Fourth Generation Language
AAM	Adaptation Aware Module
AC	Adaptive Component (nur innerhalb der Abschnitte 2.4 und 2.5 auch Autonomic Computing)
ACAP	Application Configuration Access Protocol
ACDL	Adaptation Contract Definition Language
ACL	Architecture Constraint Language
ACM	Association for Computing Machinery
ADL	Architecture Description Language
AEM	Architecture Evolution Manager
AI	Artifical Intelligence
AML	Architecture Modification Language
ANSI	American National Standards Institute
AOP	Aspect Oriented Programming
AOSD	Aspect Oriented Sofware Development
API	Application Program(ming) Interface
ASCII	American Standard Code for Information Interchange
BAA	Broad Agency Announcement
BMP	Bitmap
CAM	Component Adaptor Module
CD	Compact Disc
CHAM	Chemical Abstract Machine
cHTML	Compact HTML

CIM	Computation Independent Model
CLOS	Common Lisp Object System
CMYK	Cyan–Magenta–Yellow–Key/blacK (4-Farben-Modell, das sich besonders für das Drucken eignet)
CORBA	Common Object Request Broker Architecture
COTS	Commercial Off-The-Shelf
CP	Configuration Programming
CPU	Central Processing Unit
DAC	Distributed Adaptive Component
DARPA	Defense Advanced Research Projects Agency
DASADA	Dynamic Assembly for Systems Adaptability, Dependability, and Assurance
DB	Datenbank
DDL	Dialog Description Language
DLL	Dynamic Link Library
DOI	Digital Object Identifier
DOM	Document Object Model
DV	Datenverarbeitung
ECA	Event–Condition–Action
EDM	Enterprise Deployment Model
EJB	Enterprise Java Bean(s)
EPS	Encapsulated Postscript
FTP	File Transfer Protocol
GI	Gesellschaft für Informatik
GIF	Graphic Interchange Format
GPRS	General Packet Radio Service
GSM	Global System for Mobile Communications
GUI	Graphical User Interface
HTML	HyperText Markup Language

I18N	Internationalization
IBM	International Business Machines Corporation
IDL	Interface Definition Language
IEEE	Institute of Electrical & Electronics Engineers
IIOP	Internet Inter-ORB Protocol
IMAP	Internet Message Access Protocol
IMSP	Internet Message Support Protocol
IT	Informationstechnologie
JPEG	Joint Photographic Experts Group
JXTA	Juxtapose (von »to juxtapose«, deutsch »nebeneinander stellen«)
KI	Künstliche Intelligenz
KM	Konfigurationsmanager
L10N	Localization
LAN	Local Area Network
LDAP	Lightweight Directory Access Protocol
LEO	Link Everything Online
LFU	Least Frequently Used
LRU	Least Recently Used
MDA	Model Driven Architecture
MIT	Massachusetts Institute of Technology
MLP	Meta-Level Programming
MOP	Metaobjektprotokoll
MPEG	Moving Picture Experts Group
MRU	Most Recently Used
MVC	Model-View-Controller
OC	Organic Computing
OGSA	Open Grid Services Architecture
OSI	Open Systems Interconnection

PDA	Personal Digital Assistent
PDF	Portable Document Format
PIM	Platform Independent Model
PNG	Portable Network Graphic
PSM	Platform Specific Model
QoS	Quality of Service
RD	Resource Discovery
RFID	Radio Frequency Identification
RGB	Rot–Grün–Blau (additives Farbmodell)
RMI	Remote Method Invocation
RPC	Remote Procedure Call
RTF	Rich Text Format
SA	Software-Architektur
SAS	Self-Adaptive Software
SIP	Session Initiation Protocol
SLUB	Sächsische Landesbibliothek — Staats- und Universitätsbibliothek Dresden
SMIL	Synchronized Multimedia Integration Language
SNMP	Simple Network Management Protocol
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SoC	Separation of Concern
SOL	Solicitation
TCP	Transmission Control Protocol
UDDI	Universal Description, Discovery and Integration
UIML	User Interface Markup Language
UML	Unified Modeling Language
UMTS	Universal Mobile Telecommunications System
UPS	Universal Profiling Schema

URL	Uniform Resource Locator
VM	Virtual Machine
WAN	Wide Area Network
WAP	Wireless Application Protocol
WAV	Windows Wave
WBMP	Wireless Bitmap
WLAN	Wireless LAN
WML	Wireless Markup Language
WWW	World Wide Web
XML	eXtensible Markup Language
XSL	eXtensible Stylesheet Language

Abbildungsverzeichnis

2.1. Empfohlene Sichten nach [Bass *et al.* 2003, S. 206] . . 6
2.2. Elemente einer ACME-Beschreibung 11
2.3. Repräsentationen und Properties einer Komponente in ACME 11

3.1. Bestandteile eines Adaptionsprozesses 26
3.2. Aktivitätsdiagramm für die Bearbeitung einer Client-Anfrage in OPERA; vergleichbar mit dem Transcoding-Framework der TU Dresden 31
3.3. Überschneidungen der einzelnen Begriffe Kontext, Betriebs- und Einsatzumgebung, die Auslöser für eine Adaption der Software sind 46
3.4. Für adaptive Anwendungen werden nicht nur Kontextinformationen benötigt, sondern auch alles andere Beobachtbare (z. B. die Beschaffenheit von Schnittstellen, eigene Performance, Fehler) 52
3.5. Spektrum von adaptiver Software 55
3.6. (selbst-) adaptive Software muss (sich) wegen vielerlei Faktoren anpassen 56
3.7. Unterschiedlicher Begriff »Software« (SW); hier für Vermarktung, als Gegenstand adaptierbarer SW und für adaptive SW 62
3.8. Zusammenhang zwischen den Thematiken Adaptivität und Adaptierbarkeit 64
3.9. Erklärung der Symbole für die Definition einer Adaption 71
3.10. Bei vielen Beispielen (hier Transcoding) erfolgt nach der Adaption keine Evaluation, sondern die Adaptionsmethode wird durch einen Determinismus bestimmt (Auswertung der Bedürfnisse) und anschließend nur ausgeführt. Nach der Ausführung wird das Resultat nicht dahingehend überprüft, ob es den Bedürfnissen tatsächlich entspricht. 74
3.11. Einordnung des Untersuchungsgegenstandes der vorliegenden Arbeit 77
3.12. Kategorisierung von Adaption 79

4.1. Adaption einer verteilten Anwendung, bestehend aus der Software-Architektur (Komponenten und Konnektoren), der Kommunikation zwischen den Komponenten und den Daten 88
4.2. C&C-Sicht der Benutzeroberfläche eines Cargo-Routing-Systems im C2-Style 92
4.3. Deployment-Architektur (a) und C&C-Sicht (b) eines Beispiel-Systems in [Cheng *et al.* 2002b] 93
4.4. Unterschiedliche Interpretationen für das Krankenwagenbeispiel 95
4.5. Möglichkeiten zur Anpassung einer Software-Architektur 97
4.6. Hinzufügen einer Komponente 97
4.7. Entfernen einer Komponente 98
4.8. Austauschen der Komponente A durch A' 99
4.9. Adaption in Hadas 101
4.10. Migration einer Komponente von X nach Y 106
4.11. Verteilte Anwendung, bei der der Austausch einer Komponente in der System-Architektur nur im Binden des Clients (C) zu einer anderen Instanz des Anbieters (A) resultiert 108
4.12. Mechanismen zur Adaption von Anwendungsdaten . . . 110
4.13. Möglichkeiten zur Anpassung der Kommunikation . . . 116

5.1. (Architektur-) Modell-basierte Adaption 125
5.2. Optimalwertsteuerung/Feedforward 135
5.3. Rückkoppelungssteuerung/Feedback 135
5.4. Integrierte Architektur für selbst-adaptive Systeme . . . 137
5.5. Indirekt adaptives Regelungssystem 138
5.6. Rekonfigurierbarer Controller 139
5.7. Modell einer selbst-kontrollierenden Software 140
5.8. Struktur einer Meta-Level-Architektur 145
5.9. Meta- und Basisebene in Open ORB 148
5.10. Aufbau von K-Components nach dem Reflection-Muster 151
5.11. Dynamische Rekonfiguration einer 3-in-1-Telefonanwendung 152
5.12. Beispiel-Blackboard für die Verschmelzung von Signalen 158
5.13. Modell für das Blackboard 158
5.14. Entwurfsmuster für Beobachten, Diagnose und Wiederherstellen 161
5.15. Architektur des Cactus-Ansatzes für einen Host 162
5.16. Komponententyp *Distributed Adaptive Component* . . . 162
5.17. Struktur einer *Adaptive Component* 163
5.18. Rainbow-Framework für die externe Adaption eines Systems 167
5.19. Beispiel für die Adaption in einem Videokonferenzsystem 169
5.20. Struktur eines autonomen Elements 171

6.1. Aktivitätsdiagramm für ein adaptives System 182
6.2. Infrastruktursicht für Adaption an einem beliebigen, verteilten System 190
6.3. Läuft die verteilte Anwendung in einem Container, ist eine Komponentendatenbank für alternative Komponenten notwendig. 196
6.4. Verteilte Anwendung, bei der der Austausch einer Komponente in der Architektur nur im Binden zu einem anderen Anbieter resultiert, der nicht deployt werden muss 197
6.5. Verteiltes System mit einem zentralen und einem lokalen Konfigurationsmanager (KM) und Agenten für die Unterstützung von Legacy Systemen 204
6.6. Verteiltes System ohne einen zentralen Konfigurationsmanager (KM), jedoch mit einem lokalen KM auf jedem Host. Die Rekonfiguration in dem autonomen System erfolgt selbstorganisierend. 205
6.7. Verteilte Koordinierung einer Adaption mit Workflakes und Worklets 207

A.1. Mindmap zum Diplom 230

B.1. Mindmap für die betrachteten Adaptionsmechanismen . 232

Tabellenverzeichnis

3.1. Verfügbare Übersetzungen 20
3.2. Übersetzungen der englischen Begriffe 23
3.3. Einteilung von Kontext in vier Klassen 38
3.4. Gegenüberstellung von unterschiedlicher Adaptivität je nach Verständnis von »Kontext« 40
3.5. Mögliche Zeitpunkte für eine Adaption 69

4.1. Adaptierbarkeit (Mensch) und Adaptivität (Software passt sich selbst an) in einer Software-Architektur 109

5.1. Vergleich von Open ORB und K-Components 154
5.2. Übereinstimmung der Abstraktion von MLP und ADL/CP 156
5.3. Fitnessfunktion eines AAMs 164

6.1. Verwendete Architektur-Begriffe in Kapitel 6 180
6.2. Vergleich der Anwendungsfälle 220

It's not the strongest of the species that survives, nor the most intelligent; but the most responsive to change.

Charles Darwin

1 Einleitung

Ein immer Anforderung an Software-Systeme ist die Fähigkeit, sich zur Laufzeit selbst anzupassen, um auf variierende Ressourcen, aufgetretene Fehler im System oder sich ändernde Anforderungen zu reagieren. In der Vergangenheit waren Systeme, die sich selbst anpassen, rar; sie waren meist beschränkt auf Domänen wie Software zur Raumfahrtkontrolle oder für Telekommunikationsswitches, wo ein Herunterfahren des Systems für ein Upgrade aus ökonomischen oder sicherheitstechnischen Gründen nicht in Erwägung gezogen werden kann und der Eingriff durch den Menschen nicht immer machbar ist.

Mit zunehmender Technisierung unseres Alltags und der Expansion des Internets — Ubiquitous und Pervasive Computing sind hier die Schlagwörter — ist auch für »normale« die Notwendigket zur dynamischen Rekonfiguration gegeben. Solche Systeme müssen auf variierende Ressourcen (Bandbreite, Erreichbarkeit der Server) und Systemfehler (Server und Netzwerk werden unerreichbar, Komponenten haben Fehler) reagieren. Die Nutzer verlangen QoS-Garantien, Sicherheit und Flexibilität der Plattformen. Hinzukommen sich ändernde Anforderungen und eine heterogene Infrastruktur für heutige verteilte Software-Systeme.

Darüber hinaus besteht neuerdings auch die Erfordernis nach autonom arbeitenden Systemen. Der Drang nach Autonomie der Systeme, bei der der Mensch weitestgehend keinen Eingriff an dem System vornimmt, ist gegeben durch die zunehmende Komplexität der Software, die die Software durch den Menschen nicht mehr beherrschbar macht, sowie der durch die Unternehmen vorgegebenen Forderung zur Senkung der Administrationskosten und auch der allgemeinen Notwendigkeit, wenn beispielsweise der Internetzugang im Flugzeug durch das Bordpersonal nicht gewartet werden kann.

Inhalt und Aufbau der Arbeit

In der vorliegenden Arbeit sollen das Paradigma von adaptiver Software, die sich zur Erfüllung der angeführten Anforderungen selbst anpasst, untersucht und Ansätze für die Software-Architektur aufgezeigt werden. Dazu wird in Kapitel 2 in die Begrifflichkeit eingeführt und verwandte Arbeiten werden genannt. Wenn auch zur Begrifflichkeit gehörend, wird adaptiver Software im Besonderen ein eigenes Kapitel 3 gewidmet. Das sich daran anschließende Kapitel 4 betrachtet Adaptionsmechanismen in einer Software-Architektur sowie die der Daten und der Übertragung in einem verteilten System. In Kapitel 5 wird eine Literaturrecherche vorgenommen und es wird eine Übersicht über Unterstützungsmechanismen für adaptive Software-Systeme, in denen eine Adaption auf der Ebene der Software-Architektur vorgenommen wird, gegeben. Als Ergebnis der Recherche wird in Kapitel 6 ein eigener Software-Architektur-Ansatz formuliert und mithilfe einiger Anwendungsfälle für adaptive Software evaluiert. Die Ergebnisse der vorliegenden Arbeit finden eine Zusammenfassung schließlich im letzten Kapitel 7.

Anhang A beinhaltet auf Seite 230 mit einer Mindmap eine grobe Übersicht über die Diplomarbeit.

Architect, *n. One who drafts a plan of your house,*
and plans a draft of your money.

Ambrose Bierce in »The Enlarged Devil's Dictionary« [Bierce 1989]

2 Begrifflichkeit

In diesem Kapitel wird in die Begrifflichkeit, wie sie für die vorliegende Arbeit notwendig ist, eingeführt. Als erstes wird der wichtige Begriff Software-Architektur beschrieben und die Bedeutung verteilter Systeme charakterisiert. Danach erfolgt die Klärung, was Selbstorganisation bedeutet, und die Vorstellung der verwandten Themen Autonomic Computing und Organic Computing. Die Erläuterung der Begriffe Adaptivität und Adaptierbarkeit wird im Anschluss an dieses Kapitel vorgenommen.

2.1. Software-Architektur

2.1.1. Was ist Software-Architektur?

Es gibt zahlreiche Definitionen des Begriffs Software-Architektur[1] in der Fachliteratur. Das renommierte Software Engineering Institut (SEI) der Carnegie Mellon Universität, USA, hat auf einer Webseite ca. 150 Definitionen sammeln können [SEI 2004b]. Motiviert durch das starke Interesse von Industrie und Wissenschaft hat sich sogar das Standardisierungskomitee der IEEE[2]-Organisation damit befasst und den ANSI[3]/IEEE Standard 1471-2000, »Empfohlene Praktiken für Architekturbeschreibungen von Sofware-intensiven Systemen«, herausgearbeitet [IEEE 1471-2000], der unter anderem wichtige Schlüsselbegriffe wie System, Stakeholder, Architekt, Architektur, Architektur-Sicht (view) und Architektur-Sichtweise (viewpoint) definiert.

[1] In den folgenden Ausführungen auch nur kurz Architektur genannt und ggf. mit SA abgekürzt.
[2] Institute of Electrical & Electronics Engineers
[3] American National Standards Institute

Vertretend seien hier zwei moderne Definitionen für Software-Architektur aufgeführt:

> »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.* 2003]

> »Architecture: the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution.« [IEEE 1471-2000]

Software-Architektur befasst sich demnach mit der Spezifizierung der Strukturen und den Interaktionen eines Software-Systems. Architektur muss die Elemente, die Komponenten, eines Systems definieren, deren wesentliche (extern sichtbaren)[4] Merkmale beschreiben sowie die Beziehungen zwischen diesen Komponenten charakterisieren. Damit beschreibt eine Architektur sowohl *statische* als auch *dynamische* Aspekte. Sie erfüllt sowohl die Aufgabe eines *Bauplans* als auch die eines *Ablaufplans* für Software (vgl. [Bass *et al.* 2003, Kap. 1.3] und [Starke 2002, Kap. 2.1]).

Bei der Aufteilung eines Projektes in seine Komponenten muss der Begriff Komponente grobgranularer gesehen werden. Die in einer Architektur(beschreibung) modellierte Komponente entspricht nicht unbedingt einer physikalischen Komponente, wie sie später als kleinster Softwarebaustein auf Implementierungsebene zum Einsatz kommt, sondern kann auch — je nach Abstraktionsstufe und mit wachsender Größe — Business-Komponenten, Business-Systeme und Business-Domains[5] umfassen. In diesem Fall sind Komponenten hierarchisch aufgebaut, sie sind ineinander gekapselt.

Neben den Strukturen beschäftigt sich Software-Architektur auch mit den Prinzipien und Entscheidungen, die hinter der Entwicklung der Architektur stehen. Perry und Wolf [1992] nennen dies »rationale«, d. h. die logische Grundlage einer Software-Architektur. In der IEEE-Definition wird dieser Sachverhalt in »the principles guiding its design and evolution« deutlich.

2.1.2. Sichten

Ähnlich wie beim Hausbau sind bei einer Architektur verschiedene Sichten auf das System wichtig (z. B. bei einem Haus sind dies Grundriss-, Elektro- und Heizungsplan). Statt alle Aspekte einer Architektur in einer einzigen Grafik festzuhalten, werden diverse Sichten

[4] Private Details der Elemente, also Implementierungsdetails, gehören nicht zu den Belangen einer Architektur.

[5] Entsprechend der Einteilung für logische Komponenten nach [Andresen 2003].

für die verschiedenen Zwecke und die verschiedenen Stakeholder erstellt.[6]

Bass *et al.* [2003] heben diesen wichtigen Sachverhalt schon in ihrer Definition für Software-Architektur hervor, indem sie nicht von einer singularen Struktur, sondern von Struktur*en* reden (»the structure or structures of the system ...«). In der IEEE-1471-Definition ist die Multiplizität der Strukturen auf den ersten Blick nicht gleich zu erkennen, jedoch wird sie bei näherer Betrachtung des Wortes »fundamental« (»the fundamental organization ...«) deutlich: »Fundamental« muss im Kontext der Stakeholder und der Umgebung interpretiert werden, denn es ist nicht genau bestimmbar, was das Fundamentale an einem System ist, ohne zu wissen »fundamental für wen?«.

Jede der Sichten dokumentiert einzelne Aspekte des Gesamtsystems. Das seit jeher in der Informatik verwendete Mittel, die immer stärkere Abstraktion, findet auch hier Eingang. Eine essenzielle Aufgabe von Architekten besteht darin, die für eine bestimmte Aufgabe nicht benötigten Informationen gezielt wegzulassen, zu abstrahieren. Mittels spezifischer Abstraktionen und Filter können Software-Artefakte eines Systems so dargestellt werden, dass diese Strukturen in Form von architektonischen Sichten auf die jeweiligen Stakeholder zugeschnitten sind [Vrandecic 2001].

Die Fachliteratur bietet für die Definition von Sichten viele, verschiedene Ansätze. Am meisten sind sicherlich die 4+1 Sichten von Kruchten bzw. der Firma Rational bekannt (logical view, development view, process view, physical view, + scenarios) [Kruchten 1995]. Hofmeister bietet ein System aus 4 Sichten (konzeptionelle Sicht, Modulsicht, Codesicht, Ausführungssicht) [Hofmeister *et al.* 2000] und mit [Clements *et al.* 2002] ist gar ein gesamtes Buch zu Sichten gegeben. Die Sichten haben sich bei den Autoren während ihrer langjährigen Erfahrung als besonders nützlich erwiesen, wobei zwischen den einzelnen Systemen viele Parallelen zu erkennen sind (vgl. [Garland und Anthony 2003]). Es gibt nicht *den* Ansatz. Wichtig ist, dass der Architekt für einen gegebenen Stakeholder eine passende Sicht heraussucht, um den Aspekt des Systems, der für den Stakeholder von Bedeutung ist, entsprechend zu dokumentieren.

[6] An dieser Stelle wird auch deutlich, dass Software-Architektur nur bei komplexen, großen Systemen Sinn macht. Eine Hunde- oder Vogelhütte zu entwickeln ist einfach, um es mit den Worten von Hochmüller *et al.* [2003] auszudrücken. Mit Größe des Gegenstandes nehmen jedoch die Randbedingungen zu, die zu beachten sind. Die Entwicklung des Systems wird damit komplexer. Ebenso gibt es mehr Stakeholder, die an der Entwicklung des Systems mitwirken oder an dem Projekt interessiert sind. Genau hier setzt Software-Architektur an, da es die Komplexität von Sytemen beherrschbar und verständlich macht, indem komplexe Anforderungen in geordnete Strukturen übersetzt werden.

	Module Views				C&C Views	Allocation Views	
Stakeholder	**Decomposition**	**Uses**	**Class**	**Layer**	**Various**	**Deployment**	**Implementation**
Project Manager	s	s		s		d	
Member of Development Team	d	d	d	d	d	s	s
Testers and Integrators		d	d		s	s	s
Maintainers	d	d	d	d	d	s	s
Product Line Application Builder		d	s	o	s	s	s
Customer					s	o	
End User					s	s	
Analyst	d	d	s	d	s	d	
Infrastructure Support	s	s		s		s	d
New Stakeholder	x	x	x	x	x	x	x
Current and Future Architect	d	d	d	d	d	d	s

Key: d = detailed information, s = some details, o = overview information, x = anything

Abbildung 2.1.: Empfohlene Sichten nach [Bass *et al.* 2003, S. 206]

Beispiele

Die Sichten können sehr unterschiedlich sein. Nachfolgend werden zwei Beispiele für Sichtenempfehlungen gegeben.

Beispiel 1. Das erste Beispiel entstammt [Bass *et al.* 2003] und ist in Abbildung 2.1 reproduziert.[7] Dieser Empfehlung liegt die Kategorisierung zugrunde,

1. wie die Software in Implementierungseinheiten strukturiert ist,
2. wie die Software in Elemente mit Laufzeitverhalten und Interaktionen strukturiert ist und
3. wie die Software zu nicht-Software-Einheiten in ihrer Umgebung in Beziehung steht.

Beispiel 2. Ein zweites Beispiel ist [Starke 2002, Kapitel 4] entnommen. Der Autor des vorgenannten Buchs empfiehlt diese Sichten für »effektive Software-Architekturen«. Starke beschreibt seine fünf Sichten wie folgt in Kurzform:

[7] In der gegebenen Form fehlen die genaue Beschreibung, Anliegen sowie die Konventionen für die Modellierung und die Analyse der Sicht — Details sind in [Clements *et al.* 2002] zu finden.

Konzeptionelle Sicht Wie funktioniert das System? Die konzeptionelle Sicht liefert einen Überblick über das System und stellt es in seiner logischen Systemumgebung dar. Dazu gehören auch die Systemgrenzen, seine Benutzer und angrenzende Fremdsysteme. Diese Sicht kann als »Vision« etwas abstrakter betrachtet werden als die übrigen Sichten.

Implementierungssicht Wie ist das System intern aufgebaut? Die Implementierungssicht zeigt die Softwarebestandteile des Systems. Sie zerlegt das System in Subsysteme und Komponenten und spezifiziert deren Schnittstellen. Die letzte Stufe der Zerlegung bildet der Quellcode. Sie dient der Aufteilung von Arbeitspaketen an Teams und Mitarbeiter, unterstützt Projektleiter und Auftraggeber bei der Projektüberwachung und fungiert darüber hinaus als Referenz für Software-Entwickler.

Infrastruktursicht In welcher Umgebung läuft das System ab? Die Infrastruktursicht beschreibt die Hardwarekomponenten, auf denen das System abläuft. Sie dokumentiert Rechner, Prozessoren, Netztopologien und -protokolle sowie sonstige Bestandteile der physischen Systemumgebung. Die Infrastruktursicht zeigt eine Betreibersicht auf das System.

Laufzeitsicht Die Laufzeitsicht beschreibt, welche Bestandteile des Systems zur Laufzeit existieren und wie sie zusammenwirken.

Datensicht Bei hochgradig datengetriebenen Anwendungen kann es nützlich sein, Daten- oder Informationsflüsse innerhalb des Systems explizit zu beschreiben. Die Datensicht ist bei der Modellierung von Geschäftsprozessen nützlich und kommt daher bei der Entwicklung übergreifender Systeme zum Einsatz.

Resümee. Sichten sind Mittel, um der Komplexität Herr zu werden, indem einzelne Aspekte eines Software-Systemes gefiltert und abstrahiert in den jeweiligen Sichten für die Stakeholder dokumentiert werden. Die zwei Beispiele zeigen, dass die Empfehlungen für Sichten sehr unterschiedlich sein können. Dies ist jedoch nicht als Meinungsverschiedenheit und Unklarheit in der »Software-Architektur-Gemeinde« zu deuten, sondern als Bestätigung dafür, dass die (Dokumentation einer) Software-Architektur vielseitig sein kann und das Verständnis von Strukturen und dem Wesen einer entwickelten Software-Architektur sich für jeden Stakeholder anders erschließt und andere Foki/Abstraktionen erfordert.

Sicht/view und Sichtweise/viewpoint

In der Literatur und auch in der vorliegenden Arbeit wird vorrangig der Begriff Sicht (view) verwendet. Dies ist streng genommen falsch, da es

sich eher um eine Sichtweise (viewpoint) handelt. In IEEE 1471 werden die beiden Konzepte konkretisiert [IEEE 1471-2000]: In Analogie zu Typ und Instanz ist eine Sichtweise (viewpoint) die *Vorlage* für die Erstellung und die Benutzung einer Sicht. Eine Sichtweise beinhaltet den Namen, die Stakeholder und das Anliegen sowie die Konventionen für die Modellierung und die Analyse der Sichtweise. Dahingegen ist eine Sicht (view) die Repräsentation eines *bestimmten* Systems oder eines Teiles dieses — unter der Betrachtung einer bestimmten Perspektive (Sichtweise/viewpoint). Deshalb sind beispielsweise die 4+1 »Sichten« [Kruchten 1995] eher »Sichtweisen«, weil sie Schablonen und Bauanleitungen für die Schaffung von Sichten darstellen. Erst die Modellierung eines bestimmten Systems beispielsweise im »Physical View« (hier »physische Sicht«, eigentlich »physische Sichtweise« richtig) beschreibt eine Sicht.[8]

2.1.3. Architekturbeschreibungssprachen

Einen nicht unerheblichen Anteil in der vorliegenden Arbeit haben Architekturbeschreibungssprachen (ADL[9]s). Ein weitere Motivation für die Vorstellung von ADLs ist die Verdeutlichung des »programming in the large«[10] mit der den ADLs eigenen Abstraktion, die noch höher und ausgefeilter als die der UML ist.[11]

ADLs sind Formalismen für die Spezifikation der architektonischen Struktur und ihrer operationalen Semantik, d. h. wie die einzelnen Elemente (Komponenten) miteinander zusammenhängen. ADLs können auch andere Aspekte einer Architektur beschreiben, beispielsweise Kommunikationsprotokolle, das zugrunde liegende Designprinzip und die Zuordnung von Komponenten in der Architektur zu Quellcodemodulen, die die Komponenten implementieren [Medvidovic 1996]. Die verschiedenen ADLs sind meist entstanden, um die Entwicklung von Produktfamilien zu unterstützen.

Es gibt kein einheitliches Verstehen, was eine ADL ist und was nicht bzw. welche Anforderungen an eine ADL bestehen. Ebenso wie kein Konsens darüber besteht, welche Architekturelemente in einer ADL

[8] Dass die Benennung hier auseinander geht, liegt einfach daran, dass die 4+1 Sichten im Jahre 1995 publiziert wurden, der ANSI/IEEE Standard 1471-2000 jedoch erst danach im Jahre 2000.

[9] Architecture Description Language

[10] Als »programming in the large« wird ein komponentenbasiertes Entwickeln bezeichnet. Ein Programm soll nicht mehr zeilenweise (»programming in the small«), sondern komponentenweise entstehen. Programme wandeln sich daher zunehmend von der monolithischen Aneinanderreihung einzelner Prozeduraufrufe hin zu Collagen interagierender Softwarekomponenten. Bedingt durch den höheren Abstraktionsgrad und die Modellierungsmächtigkeit ist es möglich, komplexe Softwaresysteme zu realisieren [Griffel 1998, S. 21].

[11] Dennoch ist die UML den ADLs nicht unterlegen, da sie durch ihre Universalität ein »modeling in breadth« erlaubt, ADLs dahingehen fokussieren mit speziellen Sichten, z. B. der C&C-Sicht (Komponenten-und-Konnektoren-Sicht), auf »depth over breadth« (vgl. [Medvidovic *et al.* 2002]).

modelliert werden, herrscht auch keine Einigung darüber, was die Knoten und Kanten in einem Graph der Architektur darstellen. Zum Beispiel stellen die Knoten (vertices) und Kanten (edges) eines Software-Architektur-Graphen

- Komponenten und Konnektoren in C2 [Medvidovic *et al.* 1996],
- Schnittstellen und Konnektoren in Rapide [Luckham und Vera 1995],
- Schnittstellen (versehen mit einer implementierenden Komponente) und Konnektoren in [Wermelinger 1999] dar.

Manche ADLs bieten Toolsupport für Analyse, Design und Simulation der Architektur, andere erlauben nur die Modellierung, manche mit besonderem Fokus der formalen Spezifikation. Beispiele für sehr formal gehaltene ADLs sind COMMUNITY [Fiadeiro und Maibaum 1995] und CHAM[12] [Le Métayer 1998], welche auch in den Abschnitten 5.1.1 und 5.1.2 behandelt werden. Nach Meinung von Medvidovic und Taylor [2000] ist CHAM jedoch keine ADL.

Für einen guten Überblick sowie Vergleich der Vielzahl an ADLs wird auf [Medvidovic und Taylor 2000] verwiesen. Für das Verständnis in der vorliegenden Arbeit soll im Folgenden ACME als ADL vorgestellt werden.

ACME

Mit ACME [ACME 1998; Garlan *et al.* 1997] wurde eine Sprache entwickelt, die den Austausch der verschiedenen Architekturbeschreibungen untereinander gestatten soll. Damit ist es möglich, die ADL zu nutzen, die für den gerade zu betrachtenden Problembereich die beste Spezifikation und Toolunterstützung bei der Architektur-Entwicklung bietet. Für das Vorstellen einer ADL wurde ACME gewählt, da sie als Austauschsprache typische Eigenschaften einer ADL aufzeigt. Außerdem kann ACME als weitestgehender Konsens für die Architekturmodellierung angesehen werden [Medvidovic und Taylor 2000].

Natürlich kann eine Austauschsprache nur den kleinsten gemeinsamen Nenner aller ADLs unterstützen. Jedoch versucht ACME, auch die Speicherung von Informationen, die nicht in das Vokabular fallen, das alle ADLs unterstützen, in sog. *Properties* zu ermöglichen. Diese Properties werden bei der Umwandlung von ACME in die ADL, die diese Informationen verstehen und modellieren kann, in die ADL-spezifische Repräsentation umgewandelt. Bei Umwandlung in andere ADLs werden die Properties, die die ADL nicht versteht, ausgeklammert.

Nachfolgend wird die Ontologie von ACME näher beschrieben.

[12] Chemical Abstract Machine

Ontologie. Für die Beschreibung von Architekturen definiert ACME sieben[13] Element-Typen [Garlan *et al.* 1997]:

1. *Komponenten* repräsentieren die primären Elemente eines Systems, die Daten verarbeiten und speichern. Typische Beispiele sind Clients, Server, Filter, Objekte und Datenbanken.

2. *Konnektoren* sind für die Interaktion zwischen Komponenten verantwortlich. Erst durch sie wird es möglich, dass durch Koppelung mehrerer Komponenten das Gesamtsystem seine Arbeit erfüllt, weshalb man sie auch als »glue« (Klebstoff) beim architekturellen Design bezeichnet. Für die Interaktion gibt es einfache Formen wie Pipes, Procedure Call und Ereignis-Broadcast. Aber auch kompliziertere Formen wie ein Client-Server-Protokoll oder die SQL-Kommunikation zwischen Applikation und Datenbank kann durch Konnektoren dargestellt werden.

   Konnektoren spielen in der Software-Architektur eine große Rolle. Für weitere Informationen zu Konnektoren wird auf [Mehta *et al.* 2000] verwiesen.

3. Das *System* repräsentiert die spezielle Konfiguration der Komponenten und Konnektoren. Für den Begriff »System« wird deshalb in anderen ADLs sehr oft die Bezeichnung »Konfiguration« verwendet (vgl. [Medvidovic und Taylor 2000]).

4. *Ports* definieren die Schnittstelle einer Komponente. Dabei entspricht jeder Port einem Interaktionspunkt zwischen der Komponente und seiner Umgebung. Je nachdem, wie viele Schnittstellen eine Komponente hat, so viele Ports sind auch vorhanden. Das Besondere an den Ports ist — verglichen mit der einfachen Schnittstellenspezifikation in UML[14] —, dass sie zum einen sehr einfach sein können, indem sie nur die Signatur einer Prozedur deklarieren, zum anderen wiederum die Komplexität von mehreren Prozeduraufrufen, die in einer bestimmten Reihenfolge erfolgen müssen, darstellen können.

5. *Rollen* definieren ebenso wie Ports eine Schnittstelle, jedoch für den Konnektor. Für jeden Teilnehmer eines Konnektors wird eine Rolle definiert. So gibt es bei binären Konnektoren den Aufrufer und den Aufgerufenen, im Falle einer Pipe den Sender und den Empfänger, und, um ein Beispiel für einen Konnektor mit mehr als zwei Rollen zu nennen, bei einem Broadcast-Konnektor gibt es eine Rolle für den Nachrichten-Sender sowie mehrere Rollen für die Nachrichten-Empfänger.

[13] Die ersten drei Elemente bilden den Kern von ACME. Verwendet man für »System« gleichwertig die Bezeichnung »Configuration«, dann erhält man die Hauptbestandteile, die eine, d. h. jede, ADL ausmachen und in der Klassifikation nach [Medvidovic und Taylor 2000] ebenso postuliert werden.

[14] Unified Modeling Language

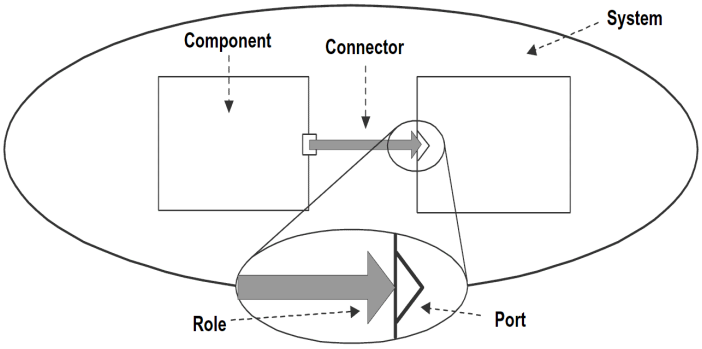

Abbildung 2.2.: Elemente einer ACME-Beschreibung (nach [Garlan *et al.* 1997, S. 7])

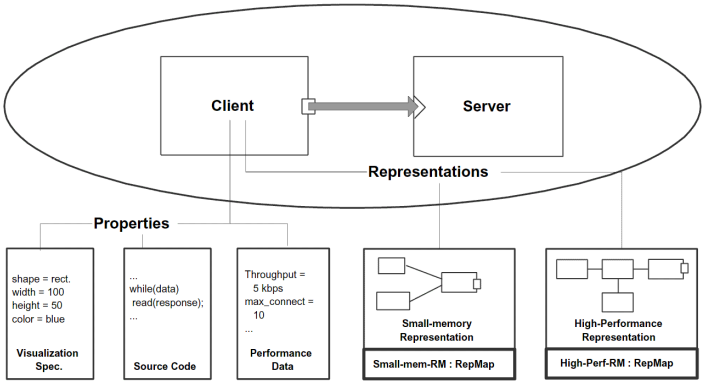

Abbildung 2.3.: Repräsentationen und Properties einer Komponente in ACME (nach [Garlan *et al.* 1997, S. 7])

6. *Repräsentationen.* In einer Architekturbeschreibung ist es wichtig, mehrere, unterschiedlich detaillierte Darstellungen für die Komponenten und Konnektoren zu spezifizieren. Diese Aufgabe wird von den Repräsentationen erfüllt. Sie entsprechen mehreren Sichten auf ein System, wobei auch die hierarchische Repräsentation möglich ist.

7. *Rep-Maps* (»representation map«). Da es für eine Komponente oder einen Konnektor mehrere Repräsentationen gibt, muss es einen Weg geben, um die Übereinstimmung zwischen der internen System-Repräsentation und der externen Schnittstelle der Komponente respektive des Konnektors zu kennzeichnen. Eine rep-map definiert diesen Zusammenhang.

Putting the pieces together. In den Abbildungen 2.2 und 2.3 ist der Zusammenhang zwischen den einzelnen Elementen einer Architekturbeschreibung, wie ACME sie ermöglicht, abschließend dargestellt.

2.2. Verteilte Systeme

Definition

Die vorliegende Arbeit orientiert sich an der Definition von Coulouris *et al.* [2002]:

> »Bei einem verteilten System arbeiten Komponenten zusammen, die sich auf vernetzten Computern befinden und die ihre Aktionen durch den Austausch von Nachrichten koordinieren.«

Was bedeutet Verteilung?

Diese Definition ist für die vorliegende Arbeit noch etwas unreichend. Mehr von Interesse ist, was überhaupt verteilt vorliegen kann. Mülle [2001] unterscheidet hierbei

- Hardware,
- Last (z. B. durch Betriebssystem),
- Daten,
- Kontrolle und
- Verarbeitung.

Im Interesse der vorliegenden Arbeit stehen vorrangig die Verteilung der Verarbeitung (in Form der Verteilung der Funktionalität auf die Softwarekomponenten, im Gegensatz zu einer monolithisch aufgebauten Software) und die Verteilung der Last im Vordergrund. Damit ergibt sich eine große Gemeinsamkeit zur Betrachtungsweise der Software-Architektur: Das System ist aufgeteilt auf verschiedene Komponenten, die auf verschiedenen Rechnerknoten arbeiten. In der Tat stammen viele Arbeiten zu dem vorliegenden Thema der Forschungsgemeinde für verteilte Systeme.

2.3. Selbstorganisation

Zusätzlich zu Software-Architektur und verteilten System wird als letztes noch der Begriff Selbstorganisation definiert.

Selbstorganisation bezeichnet das spontane Entstehen neuer Strukturen in dynamischen Systemen, das auf das kooperative Wirken von Teilsystemen zurückgeht [Brockhaus19].

Ultsch definiert Selbstorganisation als die Fähigkeit eines Systems, sich ohne gerichtete Vorgaben von außen zu ordnen. Überträgt man die Idee in den Bereich der Datenverarbeitung, so bedeutet dies, dass

Daten nicht durch Vorgaben organisiert werden, sondern sich durch eine geeignete Computertechnik selbstständig organisieren. Datensätze sollen sich selbstständig zu zusammengehörigen Gruppen zusammenfinden. Weiterhin sollten solche Gruppen eine die wesentlichen Merkmale der Gruppe charakterisierende Beschreibung entwickeln [Ultsch 2004].

2.4. Verwandte Themen

2.4.1. Autonomic Computing (AC)

Motivation

In den letzten Jahrzenten hat sich die IT[15]-Landschaft rasant geändert. Verteilte Computersysteme werden immer komplizierter und komplexer. Der Trend geht eindeutig in Richtung immer leistungsfähigerer Komponenten. Als Folge ist die zunehmde Komplexität von Netzwerken und Computersystemen eine für Menschen immer schwieriger zu bewältigende Aufgabe.

IBM hat dieses Problem 2001 erkannt und eine Lösung, eine Vision dafür proklamiert: *Autonomic Computing* [Horn 2001]. Die Vision des Autonomic Computing hat dabei den Organismus des Menschen als Vorbild. Namensgeber ist das vegetative Nervensystem (engl. *autonomic* nervous system):

Das vegetative Nervensystem des Menschen kann das, wovon die IT-Industrie noch träumt. Abhängig von der aktuellen Umgebung und Tätigkeit reguliert das vegetative Nervensystem mandatorische Körperfunktionen wie Herzfrequenz und Atmung. Diese Funktionen geschehen vollautomatisch und unbewusst im Hintergrund. Infolgedessen braucht man sich nur darauf konzentrieren, was man machen will, und nicht wie: Beispielsweise braucht man nicht für eine schnellere Atmung und eine erhöhte Pumpaktivität des Herzens sorgen, wenn man nach einem Zug rennt.

Nach Meinung von IBM besitzen IT-Systeme diese Eigenschaften nicht. Sie werden zwar immer leistungsfähiger und schneller, stellen zugleich aber auch immer höhere Anforderungen an die Administration. Dies ist auch verbunden mit höheren Kosten. Derzeit wenden Unternehmen immerhin zwei Drittel ihrer Investitionen für IT-Systeme allein für die Wartung der IT-Infrastruktur auf [IBM 2003a]. Dieser Aufwand soll durch autonome Funktionen auf ein Minimum reduziert werden.

Selbstmanagement — Self-X-Eigenschaften

Grundgedanke des Autonomic Computing ist das Selbstmanagement eines Systems. Dadurch soll die Verwaltung des Systems aus den Hän-

[15] Informationstechnologie

den des Administrators genommen werden, das System soll sich selbst verwalten. Dazu fasst IBM die Anforderung an ein autonomes System in den so genannten CHOP-Funktionen zusammen: self-**c**onfiguring, self-**h**ealing, self-**o**ptimizing und self-**p**rotecting [Keffart und Chess 2003], [Steinberg 2003a]:

selbst-konfigurierend Selbst-konfigurierende Komponenten passen sich an die sich ändernde Umgebung an. Solche Systeme können neue Features, zusätzliche Server und neue Softwareversionen hinzufügen und konfigurieren, ohne dass das System gestoppt werden muss. Wichtig hierbei ist der autonome Charakter, d. h. dass die Eingriffe des Menschen in das System minimiert werden.

selbst-heilend Selbst-heilende Systeme erkennen, analysieren und reagieren auf Ausfälle und erhöhen damit die Verfügbarkeit des Systems. Es werden fehlerhafte Komponenten identifiziert und isoliert. Diese Komponenten werden aus dem System ausgegliedert, repariert oder ersetzt und dann wieder in das System eingefügt. Autonome Systeme benötigen daher ein Minimum an Redundanz, damit die Heilungsverfahren transparent gegenüber den Nutzern geschehen können.

Der Charakter eines sich selbst-heilenden Systems, das fehlerhafte Komponenten isoliert und (ggf. durch Eingriff des Menschen) repariert, ist nicht wortwörtlich zu nehmen. So existieren in der Literatur auch andere Beispiele für »selbst-heilende« Systeme. Beispielsweise wird in [Blair *et al.* 2002] als selbst-heilend auch bezeichnet, wenn ein mobiles Endgerät für die Führung eines Touristen je nach Verfügbarkeit SOAP[16], IIOP[17] oder Publish-Subcribe für die Kommunikation mit dem Touristeninformationssystem benutzt. In diesem Fall wird nicht auf einen Ausfall reagiert, sondern das System passt sich an die Systemumgebung an (verfügbare Kommunikationsprotokolle) und *bleibt damit operabel*.

de Lemos und Fiadeiro [2002] merken an, dass das allgemeine »self-adaptation« auch manchmal als »self-healing« bezeichnet wird.

selbst-optimierend Selbst-optimierende Systeme suchen kontinuierlich nach Wegen, ihre Operationen zu verbessern, indem sie die zur Verfügung stehenden Ressourcen hinsichtlich Performance und Kosten effizient ausnutzen. Autonome Systeme beobachten und experimentieren mit ihren Parametern und lernen daraus.

selbst-schützend Selbst-schützende Systeme schützen sich selbst vor Eindringlingen und verhindern Datenkorruption. Sie verteidigen

[16] Simple Object Access Protocol
[17] Internet Inter-ORB Protocol

das System gegen Attacken oder sich auswirkende Fehler, die durch den selbst-heilenden Mechanismus nicht korrigiert werden konnten. Außerdem ahnen sie Probleme voraus, indem sie frühzeitig generierte Reports von Sensoren analysieren und den Problemen entgegenwirken oder sie abmildern.

Diese Eigenschaften werden in der Literatur auch allgemein als Self-X [Rademacher 2004], [Schmeck 2004] oder Self-* [Self-∗ 2004] referenziert. Mittlerweile gibt es eine Fülle an sog. Self-X-Eigenschaften, die die vier von IBM aufgestellten erweitern (siehe auch Organic Computing) und gar Thema eines ganzen Workshops sind [Self-∗ 2004]. Zwei der weiteren Self-X-Eigenschaften sind die folgenden:

selbst-organisierend Eine Eigenschaft, die nicht primär im Fokus von Autonomic Computing steht (vgl. Manifest [Horn 2001] und »die Vision« [Keffart und Chess 2003]), jedoch durch verwandte Forschungsaktivität, beispielsweise Organic Computing, als Eigenschaft von zukünftigen Systemen proklamiert wird, ist die Selbstorganisation (vgl. Kapitel 2.3 auf Seite 12).

self-awareness Ein autonomes System muss Wissen über sich selbst besitzen, ähnlich dem eigenen Bewusstsein des Menschen.[18] Das schließt Wissen über seine Komponenten, den eigenen Status, Verbindungen zu anderen Systemen und ebenso Ressourcen ein, über die das System verfügen kann [Horn 2001].

2.4.2. Organic Computing (OC)

Wie so oft in Wissenschaft und Forschung werden neue Verfahren und Technologien bei der Natur abgeschaut. Nicht anders ist es bei der Organic-Computing-Initiative (OC) [OC 2003], [Rademacher 2004]. »Ein „organischer Computer“ist definiert als ein selbstorganisierendes System, das sich den jeweiligen Umgebungsbedürfnissen dynamisch anpasst.« Organische Computer sind selbst-konfigurierend, selbst-optimierend, selbst-heilend und selbst-schützend. Sie sind in der Lage, »sich selbständig an die Wünsche und Bedürfnisse des Anwenders des technischen Systems anzupassen« [Rademacher 2004]. Diese Eigenschaft hat Organic Computing seinen Namen gegeben, da sich OC-Systeme lebensähnlich, also organisch verhalten. Durch das Nachempfinden der Natur sollen zwei Kernziele erreicht werden: Beherrschung der Komplexität und Erhöhung der Robustheit mächtiger dynamischer Systeme [OC 2003].

[18] In dem Zusammenhang eine interessante Anmerkung: Menschen werden nicht mit einem Modell von sich selbst bereits geboren, sondern erlernen und entwickeln dieses, wenn sie aufwachsen. Es wurde beispielsweise das Phänomen beobachtet, dass Kleinkinder unter dem Alter von 15 Monaten nicht imstande sind, sich selbst in einem Spiegel zu erkennen (vgl. [Amsterdam 1972]; in Anlehnung an [Czarnecki und Eisenecker 2000]).

Die Organic-Computing-Initiative ist ein starker Zusammenschluss aus Vertretern der Industrie und der Universitäten, wobei Spezialisten aus verschiedenen Bereichen ihr Wissen beisteuern: Spezialisten für Systemarchitekturen, Prozessoren, Ubiquitous- und Wearable-Computing, Neuroinformatiker und Biologen. Durch diese Interdisziplinarität der Initiative ist der Ansatz breiter als das auf IT-Infrastrukturen beschränkte Autonomic Computing von IBM, Sun oder HP.[19] Auch Sensornetze, Steuereinheiten in Anlagen und Fahrzeugen oder domotische[20] Systeme sollen profitieren — das Autonomic Computing ist vormerklich auf große Serverfarmen beschränkt.

Gemeinsamkeiten AC/OC. Gemeinsam mit Autonomic Computing hat Organic Computing die Eigenschaft, dass es den Fokus auf die sog. Self-X-Eigenschaften richtet: *self*-configuring, *self*-optimizing, *self*-healing, *self*-protecting, self-organizing[21]. Diese bisher nur in lebendigen Systemen anzutreffenden Eigenschaften werden zunehmend auch für technische Systeme benötigt, da man mit herkömmlichen Methoden komplexe Systeme nicht mehr in Griff bekommt.

Unterschiede AC/OC. Im Unterschied zu Autonomic Computing ist die Organic Computing Initiative interdisziplinär angelegt und fokussiert sich nicht nur auf IT-Systeme oder Grid Computing, sondern auf eingebettete Systeme wie Autos oder eine Ansammlung von Smartphones. Neben der Selbstorganisation wird zudem ein Phänomen komplexer Systeme einbezogen, das sich Emergenz nennt. Emergenz liegt vor, wenn das globale Verhalten von anderer Art ist als das der einzelnen Komponenten [Schmeck 2004]. Emergenz bezeichnet das Entstehen neuer Strukturen und Eigenschaften aus dem Zusammenwirken der Elemente, also Eigenschaften, die ein Gesamtsystem hat, aber keins seiner Einzelteile.[22]

Die Interdisziplinarität der OC-Initiative macht sich besonders auch in einem Aspekt bemerkbar, der von Autonomic Computing gar nicht verfolgt wird: energiesparende Hardware. Der zunehmenden Integrierung von Rechnersystemen beispielsweise im Fahrzeugbereich wird

[19] Eine Übersicht über die Ansätze von Sun's »N1 Computing«, HP's »Adaptive Enterprise« und IBM's »Autonomic Computing« findet der Leser in [Dingler 2004].

[20] Domotisch leitet sich aus dem Wort »Domus« (lat. für Haus) und dem Wort »automatisch« ab. http://1367.rapidforum.com/topic=111383341714

[21] Streng genommen kann die Eigenschaft der Selbstorganisation nicht zum Autonomic Computing gezählt werden (vgl. Abschnitt 2.4.1). Literatur, die explizit Autonomic Computing behandelt, zählt nur die vier erstgenannten Self-X-Eigenschaften auf. In Unterlagen zu Organic Computing wird im Zusammenhang mit den Self-X-Eigenschaften auf Autonomic Computing verwiesen, das diese ebenso verfolgt, und dann wird auch *selbstorganisierend* genannt.

[22] Ein populäres Beispiel für eine solche emergente Struktur ist das Auftauchen einer La-Ola-Welle, die aus der Selbstorganisation vieler Menschen entstehen kann [Ultsch 2004].

Einhalt durch die begrenzte Energieversorgung gegeben. Banale Batterieprobleme haben sich zum wichtigsten Pannen-Verursacher im Bereich Elektrik/Elektronik entwickelt [Süddeutsche 2004]. Deshalb stellt die Reduktion des Energiebedarfs von Mikroprozessoren und -controllern eine zentrale Anforderung an zukünftige Rechnersysteme dar, sei es im Fahrzeug oder im ubiquitären Bereich.

»Das OC ist der große Rahmen, der Trends wie Context-Awareness, nichtlineare und Mulitagentensysteme oder Ubiquitous-Computing umfasst«, umreißt Paul Lukowicz Organic Computing in [Rademacher 2004].

2.5. Zusammenfassung

Dieses Kapitel hat in die Begriffswelt der Software-Architektur eingeführt und die Relevanz verteilter Systeme gekennzeichnet. Mit verteilten System wird in der vorliegenden Arbeit nur die Komponentisierung von Software betrachtet. Damit ergibt sich ein starker Zusammenhang zu Software-Architektur, die ebenso die Aufteilung eines Projektes in seine Komponenten vornimmt, hier jedoch in mehrere Strukturen/Sichten. Diese Strukturen sind die Abstraktion eines komplexen Systems für jeden Stakeholder. Zudem deklariert eine Software-Architektur auch die Prinzipien und Entscheidungen, die hinter dem Entwurf der Software stehen.

Im zweiten Teil wurden die deklarierten verwandten Themen kurz angerissen und die Selbstorganisation eines Systems definiert. Die Forschungsthemen Autonomic Computing (AC) und Organic Computing (OC) können zusammengefasst werden, dass sie folgende Motivationen besitzen (SAS ist das von der DARPA[23] initiierte Projekt, auf das in den Abschnitten 3.4 und 3.6.3 zurückgekommen wird):

- Beherrschung der Komplexität von Software (AC + OC + SAS),
- autonomes Operieren eines Systems auf Basis eines vorgegebenen Ziels (AC + OC + SAS),
- Robustheit der Software (OC + SAS),
- Stabilität (OC + SAS) und
- ggf. selbstorganisierend (OC).

Damit ist mit selbst-managenden Systemen und organischen Systemen ein breiter Themenrahmen gesteckt. Im weiteren Verlauf wird untersucht, welche Rolle dieser für »adaptive Software-Systeme« besitzt.

[23] Defense Advanced Research Projects Agency

Der weitere Verlauf

Neben den in diesem Kapitel geklärten Begriffen fehlt noch ein Begriff, das Hauptthema dieser Arbeit. Im nächsten Kapitel wird ausführlich auf Adaptivität und Adaptierbarkeit in Software, im Anschluss daran insbesondere für Software-Architekturen, eingegangen und eine Begriffsbestimmung für adaptive Software vorgenommen.

Der Deutung für den Begriff »adaptiv« ist anpassungsfähig,
die für »Kontext« vom Kontext abhängig.

Autor

3 Adaptive Software-Systeme — eine Begriffsbestimmung

Für die Thematisierung der vorliegenden Arbeit und die Erfüllung des ersten Schwerpunktes (»Definition einer Software-Architektur bzgl./unter den Aspekten Adaptivität und Adaptierbarkeit«) ist es wichtig, zuerst die Begriffe Adaption, adaptierbar/Adaptierbarkeit und adaptiv/Adaptivität — und aufgrund des hohen Prozentsatzes an genutzter englischer Literatur auch die englischen Formen adaptation, adaptive/adaptivity/adaptiveness und adaptable/adaptability — zu klären. Das vorliegende Kapitel erfüllt diese Aufgabe.

3.1. Enzyklopädieeinträge und Übersetzungen

Die Übersetzung für das englische Wort »adaptive« ist mehrdeutig. Das PONS Großwörterbuch [PONS 2002] liefert die Übersetzung *anpassungsfähig*, die bekannte Übersetzungsmaschine `http://dict.leo.org` des LEO[1]-Dienstes der TU München zusätzlich *adaptiv* (dt.), *anwendbar/verwendbar* sowie *lernend/lernfähig*. Für »adaptable« liefert LEO *anpassbar*, *anpassungsfähig* und *verwendbar*. Duden-Oxford Großwörterbuch Englisch [Duden-Oxford 1990] und PONS Großwörterbuch [PONS 2002] halten für »adaptable« die Übersetzungen *anpassungsfähig*, *vielseitig* und *flexibel* bereit. Eine komplette Auflistung der verfügbaren Übersetzungen ist in Tabelle 3.1 auf der nächsten Seite zusammengefasst.

Schon an dieser Stelle fällt auf, dass sowohl »adaptive« als auch »adaptable« mit *anpassungsfähig* übersetzt werden können, d. h., dass eine mit diesen Attributen versehene Software die Fähigkeiten besitzt,

[1] Link Everything Online

englische und deutsche Übersetzungen	
adaptiv	adaptive^{LOP}
Adaptivität	∅
adaptierbar	∅
Adaptierbarkeit	∅
anpassungsfähig	adaptable^{LOP}, adaptive^{L}, adjustable^{L}, flexible^{L}, matchable^{L}
Anpassungsfähigkeit	adaptability^{LOP}, adaptiveness^{L}, flexibility^{L}
anpassbar	adaptable^{L}, customizable^{L}, fittable^{L}
Anpassbarkeit	adaptability^{L}, adaptivity^{L}
adaptive	adaptiv^{L}, anpassungsfähig^{LP}, anwendbar^{L}, verwendbar^{L}, lernend^{L}, lernfähig^{L}
adaptivity	Anpassbarkeit^{L}
adaptiveness	Anpassungsfähigkeit^{L}
adaptable	anpassbar^{L}, anpassungsfähig^{LOP}, vielseitig^{OP}, flexibel^{OP}, verwendbar^{L}
adaptability	Anpassbarkeit^{L}, -fähigkeit^{L}, -möglichkeit^{L}, -vermögen^{L}, Anwendbarkeit^{L}

Tabelle 3.1.: Verfügbare Übersetzungen (L = LEO, O = Oxford, P = PONS)

Anpassungen vorzunehmen. Es besteht demnach kein Unterschied zwischen »adaptive« und »adaptable«? Das scheint seine Richtigkeit zu haben, denn in der Tat bedeutet »(to be) able«, das ein Wortbestandteil in »adapt·able« ist, dass jemand *fähig*/imstande ist, etwas zu tun.[2] Deshalb scheint es legitim zu sein, »adaptable« mit *anpassungsfähig* zu übersetzen (»adapt·able« = »able to adapt«).

Werden jedoch die Einträge für »adaptable« im Duden-Oxford- und im PONS-Wörterbuch weitergehend analysiert, ...

> **adaptable** *adj.* anpassungsfähig; vielseitig <*Maschine*>; flexibel <*Planung*>; **be ~ to** *or* **for sth.** an etw. (*Akk.*) angepasst werden können [Duden-Oxford 1990]
>
> **adaptable** *adj plant, animal, person* anpassungsfähig; *vehicle, hair style* vielseitig; *schedule* flexibel; *book* zur Adaption *or* Bearbeitung geeignet. **to be ~ to sth** (*person, animal, plant*) sich an etw (*acc*) anpassen können; (*vehicle*) sich in etwas (*dat*) verwenden lassen [PONS 2002]

... dann fallen mehrere Ungereimtheiten auf:

1. Der Oxford-Eintrag beschreibt einerseits für *adaptable* → *anpassungsfähig* eine Aktiv-Form (da nach obiger These »able« fähig heißt, d. h., jemand ist fähig, etwas zu tun), aber für die

[2] Hier lässt sich eine Ausnahme finden: einsatzfähig. Für das Beispiel eines Fußballers lässt sich einerseits aussagen, dass er einsatzfähig ist, wenn er fit ist (die Fähigkeit zum guten Spielen besitzt). Andererseits kann er zwar fit, aber nicht einsatzfähig, d. h. nicht einsetzbar sein, wenn er eine rote Karte hat. Im Deutschen scheinen beide Verwendungen plausibel zu sein.

Formulierung »our software is adaptable to/for sth.« wird eine Passiv-Form beschrieben (»unsere Software kann an etwas angepasst werden«).

2. Der PONS-Eintrag beschreibt für *adaptable* → *anpassungsfähig* auch eine Aktiv-Form, für »our software is adaptable to sth.«, also für den gleichen Wortlaut, wie ihn auch der Duden-Oxford verwendet, *jedoch* eine Aktiv-Form.

→ Resultat: Je nachdem, ob der Leser Duden-Oxford oder PONS für eine Übersetzung von »our software is adaptable to sth.« heranzieht, erschließen sich für ihn zwei verschiedene Sachverhalte. Bei dem einen ist die Software so gut entwickelt worden, dass sie leicht an neue Anforderungen angepasst werden kann (Passiv), bei dem anderen wird die Aussage getroffen, dass die Software gar selbst Anpassungen vornehmen kann (Aktiv).

Es ist ein Unterschied, ob die Beleuchtungsstärke eines Bildschirms durch den Nutzer nur angepasst werden kann oder ob der Bildschirm selbst dazu fähig ist, diese Justierung in Abhängigkeit von der Umgebung vorzunehmen.

Als Ergebnis an dieser Stelle bleibt festzuhalten, dass mithilfe der PONS- und Duden-Oxford-Übersetzungen überhaupt nicht klar unterschieden werden kann, ob »an adaptable software« eine Software ist, die nur angepasst werden kann, oder ob sie eine Software ist, die selbst Anpassungen vornehmen kann.

Hilfreicher ist für »adaptable« die Übersetzung *anpassbar* von LEO, die eindeutig eine Eigenschaft der Passivität der Software ausdrückt. Sie bedeutet, dass eine Software angepasst werden kann. Diese Variante ist auch konform dazu, wenn das deutsche *anpassbar* in das Englische übersetzt werden soll. Da das eigentliche Thema »adaptive Architekturen« lautet, wird im Folgenden wieder der Fokus auf »adaptiv« und »Adaptivität« zurückgenommen — denoch dürfen adaptierbar/anpassbar und Adaptierbarkeit/Anpassbarkeit nicht ungeklärt bleiben, da sie in der Aufgabenstellung Verwendung finden.

Das deutsche »adaptiv« hat im Englischen eine gleichlautende Übersetzung. Doch für eine Übersetzung in der Rückrichtung werden neben dem bisher noch bedeutungslosen adaptiv und dem problematischen anpassungsfähig außerdem *anwendbar/verwendbar* und *lernend/lernfähig* genannt. Diese Attribute beschreiben unterschiedliche Eigenschaften. Für einen Menschen kann zwar in einem Satz formuliert werden, dass dieser für eine schwierige Arbeit *verwendbar* ist, weil er *lernfähig* und *lernend* ist. Aber eine Software stellt je nach zu berücksichtigender Eigenschaft unterschiedliche Herausforderungen an den Software-Entwickler: Muss die Software nur *halbwegs verwendbar* oder gar in ihrem Verhalten *lernfähig* sein? Die verfügbaren Übersetzungen für »adaptive« sind zu breit gefächert.

Als kleines Ergebnis am Rande: Mancher Leser besitzt vielleicht ein Vorurteil gegenüber einer (kostenlosen) Übersetzungsmaschine im Internet, da diese vermutlich kein mit PONS und Duden-Oxford vergleichbares Redaktionsteam besitzt, um Qualität zu liefern.[3] Für die Übersetzungen, wie sie in Tabelle 3.1 auf Seite 20 zusammengefasst sind, lässt sich jedoch das (nicht repräsentative) Ergebnis ziehen, dass LEO quantitiv mehr Übersetzungen liefert als die Wörterbücher PONS und Duden-Oxford. Qualitativ bewegen sich die Übersetzungslieferer jedoch auf fast gleichem (hier schlechtem) Niveau, weil die Einträge mehrdeutig übersetzt werden können.

Alle verfügbaren Übersetzungen sind für eine klare Begriffsbestimmung/Definition, wie sie in einer wissenschaftlichen Arbeit notwendig sind, noch zu allgemein und vieldeutig.

Doch auch die Brockhaus-Enzyklopädie [Brockhaus19] hilft nicht weiter. Der Begriff »adaptiv« ist als »auf Adaption beruhend, durch Anpassung erworben« definiert, wobei Adap(ta)tion[4] im Allgemeinen als eine Aktivität (teils als Reaktion, teils als Aktion (hier zufällige Mutation)) für Veränderungen verstanden werden kann. Adaption kann dabei eine kurzzeitige Veränderung des Organismus (Hell-Dunkel-Anpassung des Auges) als auch eine langzeitliche Entwicklung im Rahmen der Evolution bedeuten (siehe Fußnote 4).

Sowohl die Brockhaus-Definition für *adaptiv* und die Übersetzungen/der Begriff *adaptable/anpassungsfähig* suggerieren, dass etwas adaptiv/anpassungsfähig ist, wenn es sowohl Adaptionen vornimmt (aktive Rolle) als auch adaptiert wird/werden kann (passive Rolle).

Resümee. Für eine klares Verständnis von Adaption, adaptiv/Adaptivität und adaptierbar/Adaptierbarkeit sind die allgemeinen Definitionen in Enzyklopädien und Wörterbüchern noch zu ungenau. Es scheint angebrachter zu sein, Adaptivität nicht im allgemeinen Sinne, sondern nur im Rahmen von Software zu betrachten — zumal bei

[3] Vgl. auch [Hasenbein und Schreiber 2002].

[4] Nicht nur im Englischen heißt es »adaptation«. Im Deutschen gibt es sowohl »Adaption« als auch »Adadap**ta**tion«.

Der Brockhaus [Brockhaus19] unterscheidet diese zwei Begriffe. Unter *Adap**ta**tion* im physiologischen Sinne versteht der Brockhaus sowohl die *langzeitliche Entwicklung* der Organismen, die sich unter Einfluss von Evolutionsfaktoren wie beispielsweise Selektion und Mutation unterschiedlich entwickeln — was sich z. B. in der unterschiedlichen Pigmentierung der Haut von Menschen widerspiegelt —, als auch *kurzzeitige Anpassungen* an die Umgebung — so beispielsweise die Hell-Dunkel-Anpassung des Auges. *Adaption* sieht die Enzyklopädie bei Literatur gegeben, als »formale, gattungsverändernde Umarbeitung eines Werks, bes. die Umarbeitung eines Erzählwerks zum Drama oder zum Filmdrehbuch« [Brockhaus19].

Langenscheidts Fremdwörterbuch merkt auch beide Formen an, nimmt jedoch keine Trennung vor: »*Adaptation*: [lateinisch] auch Adaption, 1. Anpassung, Anpassungsvermögen 2. (biol., med.) Anpassung an Umweltbedingungen (Organe) 3. Bearbeitung, Umarbeitung eines Kunstwerks, Neubearbeitung in einer anderen Gattung« [Langenscheidt 1989].

Software, auch in Hinblick auf Evolution/Evolutionsfaktoren, die Frage besteht, wer oder was die Software anpasst.[5]

Ist eine Software adaptiv,

1. wenn sie sich selbst anpasst,
2. wenn sie lediglich Anpassungen vornimmt (d. h. nicht nur an sich selbst, sondern z. B. auch an den Anwendungsdaten) oder
3. wenn die Software durch den Softwareentwickler angepasst wird bzw. (leicht) angepasst werden kann?

Bei allen drei Prozessen findet eine Adaption statt (Brockhaus-Kriterium für »adaptiv«), fraglich sind jedoch Adaptionssubjekt (wer passt an) und Adaptionsobjekt (was wird angepasst).

Die bisherigen Ausführungen waren noch sehr abstrakt. Sie versuchten, den Begriff linguisitisch und mithilfe der deutschen Grammatik anzugehen, um entweder die aktive oder die passive Rolle von Software zu bestimmen. Außerdem wurde bisher noch nicht der Begriff Adaption geklärt. Für den Leser ist es deshalb sicher schwierig, die Thematik zu verstehen.

Bevor verfügbare Definitionen für die Begriffe Adaption, Adaptivität, Adaptierbarkeit in Software gesucht und angegeben werden (Deduktion), soll deshalb zunächst der induktive Weg gewählt und einige Beispiele für Adaption in Software gefunden werden, um aus diesen eine Verallgemeinerung ableiten zu können.[6]

Übersetzungen in der vorliegenden Arbeit. Ehe dieser Abschnitt abgeschlossen wird und im nächsten Abschnitt verfügbare Definitionen und Beispiele recherchiert werden, muss eine Regelung für die Übersetzungen getroffen werden. Tabelle 3.2 fasst die im weiteren Verlauf genutzten Übersetzungen für die englischen Begriffe zusammen.

englisch	*deutsch*
adaptation	Adaption, Anpassung
adaptive	adaptiv
adaptivity, adaptiveness	Adaptivität, Anpassungsfähigkeit, A.-Vermögen
adaptable	adaptierbar, anpassbar
adaptability	Adaptierbarkeit, Anpassbarkeit

Tabelle 3.2.: Übersetzungen der englischen Begriffe

[5] Das »was« und die »Software« können in dem letzten Nebensatz sowohl als Objekt als auch als Subjekt angesehen werden.

[6] [Balzert 1996, S. 12] merkt für eine Vorgehensweise in der Didaktik an: Bei der induktiven Vorgehensweise wird vom Speziellen zum Allgemeinen hingeführt. Ausgangspunkt sind mehrere Beispiele, aus denen auf eine allgemeine Regel geschlossen wird. Diese Art ist für Lernende oft einfacher zu verstehen. Dahingegen wird beim deduktiven Weg der Einzelfall bzw. das Besondere aus dem Allgemeinen hergeleitet. Diese Art wird in wissenschaftlichen Abhandlungen i. d. R. angewandt.

3.2. Beispiele

Schon Subramanian und Chung [2002a] mussten feststellen, dass in der Literatur zahlreiche Beispiele für Softwareanpassung (software adaptation) zu finden sind, wobei für das Verständnis kein gemeinsamer Konsens erkennbar ist.

Einige der Beispiele aus [Subramanian und Chung 2002a] werden nachfolgend aufgelistet. Nach dem Gedankenstrich wird vom Autor der vorliegenden Arbeit ein Versuch unternommen, die gegebene Anwendung in Beziehung zu *Anpassung* (Adaption) zu setzen:

1. Ein Dual-Band-Handy, das automatisch zwischen den zwei Systemen wechselt, je nachdem, welcher Service aktuell verfügbar ist — Das Handy passt sich an die dynamische Umgebung an.

2. Dynamisches Hochladen der Firmware, ohne das System rebooten zu müssen — Softwareanpassung (Update) ist während des laufenden Betriebs möglich.

3. Ein Kommandoverarbeitungssystem (command-processing system), das Kommandos verschiedener Versionen unterstützt — Die Schnittstelle passt sich an die verschiedenen (heterogenen) Versionen der Anwendung an.

4. Eine Software, die auf verschiedenen Betriebssystemen lauffähig ist — Hier sind verschiedene Deutungen möglich: (1) Die Software kann unangepasst (ohne Neukompilation für Zielplattform) auf verschiedenen Plattformen laufen, z. B. durch die Verwendung einer VM[7], die im Endeffekt die Anpassung vornimmt (Zielsprachenübersetzung). (2) Software muss neu kompiliert werden und wird durch Compiler oder durch Präprozessor-`#ifdefs` an die Zielplattform angepasst. (3) Binary ist durch sein besonderes Mischformat (von a.out, ELF und .EXE), falls es so was (irgendwann mal/bereits?) gibt, auf allen Plattformen lauffähig.

5. Ein System, das Wartungsarbeiten wie Backup und Speicherbereinigung (garbage collection) nur ausführt, wenn das System am wenigsten beschäftigt ist — Die Ausführungszeit für die Wartungsarbeiten wird an die Systemauslastung angepasst.

6. Ein dynamisch änderbares Format, z. B. für das Datum 2- oder 4-stellig oder als Einheit Kilometer vs. Meilen (das Problem, das den Mars Climate Orbiter zum Absturz brachte) — Das Format von Maßeinheiten kann an verschiedene Repräsentationen angepasst werden.

7. Das Mars Pathfinder Projekt konnte erfolgreich durchgeführt werden, da die Software während des Einsatzes geändert werden

[7] Virtual Machine

konnte — Softwareanpassung (Update) ist während des laufenden Betriebs aus der Ferne möglich.

8. eLiza-Projekt bei IBM, in dem selbst-verwaltende Systeme entwickelt werden — Vorgänger der »Autonomic Computing«-Initiative; »eLiza is taking the ultimate step in dynamically adapting to changes in business policies«, d. h. das IT-System passt sich den Vorgaben der Unternehmensleitung an.[8]

Diese Liste ließe sich noch beliebig fortsetzen. Beispielsweise bezeichnen Amberg *et al.* [2003] die Internetsoftware Amazon als adaptiv, weil sie aufgrund von Protokollierung eine Personalisierung der Oberfläche durchführt und dem Nutzer von ihm kürzlich angesehene Objekte als Favoriten anzeigt, Kommentare und Rezensionen für einen Artikel sowie das bekannte »Kunden, die dieses Buch gekauft haben, haben auch folgende Bücher gekauft« anbietet. Auch genannt werden das Startmenü von Windows und die Menüstruktur der Microsoft-Office-Familie, bei denen nur häufig genutzte oder zuletzt verwendetete Menüpunkte aufgelistet werden. Diese Anwendungsbeispiele können als Anpassung der Oberfläche und Daten (Produktlinks) an den Nutzer verallgemeinert werden (Personalisierung).

Ohne noch weitere Beispiele in der Literatur aufzusuchen, soll nun ein erster Schritt unternommen werden, Adaption in Software zu verallgemeinern.

3.3. Zwischenergebnis: Bestandteile eines Adaptionsprozesses

Die vorgenannten Beispiele sind vielfältig und in ihrem Verständnis von Adaption und Adaptivität noch weiter differierend als es die Definitionen aus den Wörterbüchern und Enzyklopädien vermuten lassen haben. Alle Beispiele behandeln Anpassung/Adaption, jedoch unterscheiden sie sich in folgenden Dingen:

[8] Ein Beispiel hierfür ist in [Steinberg 2003a] zu finden, die Übersetzung wird [Dingler 2004] entnommen: »... die angebotenen Dienste und wirtschaftlichen Regeln des Unternehmens [sind] miteinander gekoppelt. Es werden nun nicht mehr IT Regeln definiert für bestimmte Komponenten und Optimierungen. Hier werden Strategien auf Grund von Business Präferenzen erstellt und das IT Personal hat die Aufgabe das System und die Regeln zu optimieren um bestmögliche wirtschaftliche Ergebnisse für das Unternehmen zu erzielen. Offeriert ein Unternehmen zum Beispiel auf der einen Seite freie Inhalte und auf der anderen Seite Inhalte welche nur Bezahlern zugänglich sind, so könnte eine Regel die Bevorzugung der Bezahler definieren, was zur Folge hätte, dass die Ressourcen immer für die bezahlenden Kunden zur Verfügung gestellt werden, auch wenn dies eine Verschlechterung der Versorgung der freien Inhalte zur Folge hätte. Mittels dieser Technologien sind Unternehmen in der Lage ihre IT Architektur in Echtzeit auf Marktgegebenheiten und Produktstrategien anzupassen. Ausserdem werden mögliche Probleme automatisch erkannt und gelöst um die wirtschaftlichen Vorgaben am besten zu erfüllen.« [Dingler 2004, S. 14]

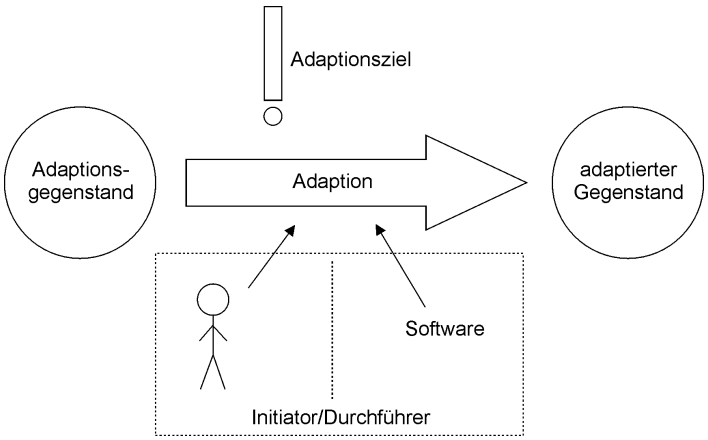

Abbildung 3.1.: Bestandteile eines Adaptionsprozesses

Es gibt

- einen unterschiedlichen Grund/Anlass für eine Anpassung, das Adaptionsziel,
- den Anpassungsgegenstand (Adaptionsobjekt; mal Software in allen verschiedenen Facetten, mal andere Aspekte, wie z. B. die Ausführungszeit) und
- einen Initiator/Durchführer, der den Adaptionsbedarf erkennt und der die Adaption durchführt (Adaptionssubjekt; hier Mensch oder Software).

Diese Bestandteile eines Adaptionsprozesses sind in Abbildung 3.1 schematisch zusammengefasst und sollen in der weiteren Begriffsbestimmung bei der Einordnung von Adaptionsbeispielen helfen.

Eine weitere Gemeinsamkeit und Verallgemeinerung kann, so unterschiedlich die Beispiele in ihrer Art auch sind, gezogen werden: Adaption verbessert Dinge, sei es, dass das Handy nicht nur in einem Netz, sondern in mehreren funktioniert, oder dass Software an geänderte Anforderungen angepasst wird (womöglich zur Laufzeit), sie nicht nur auf einer, sondern auf mehreren Plattformen läuft, ihr Verhalten an die Systemumgebung angepasst ist (Systemauslastung)... Und auch der in der Software-Technologie bekannte Adapter [GoF 1996], der Inkompatibilitäten zwischen zwei Schnittstellen anpasst, sorgt für eine Verbesserung, da die Schnittstellen (Komponenten) nach Einsetzen des Adapters miteinander interagieren können.

Auf eine weitere Analyse der oben aufgeführten Beispiele wird verzichtet. Als nächstes werden verfügbare Definitionen in der Literatur diskutiert und versucht, mithilfe dieser eine Begriffsbestimmung für »adaptive Software« vorzunehmen.

3.4. Definitionen

Oft werden in der Informatik Begriffe unklar definiert. Das Gleiche trifft auch auf die Termini *Adaptivität* und *Adaptierbarkeit* zu. Die folgenden Definitionen geben einen Überblick über das unterschiedliche Verständnis für Adaptivität/adaptivity und Adaptierbarkeit/adaptability in Software:

1. »*Adaptability* is defined as the ease with which a system or parts of the system may be adapted to the changing *requirements*.« [Tekinerdogan und Aksit 1996]

   Die Einfachheit, mit der ein System oder ein Teil des Systems an sich ändernde *Anforderungen* angepasst werden kann, wird als *Adaptierbarkeit* bezeichnet.

2. »*Adaptive* object-oriented software is software that adapts automatically to changing *contexts*. Contexts may be behavior, implementation class structures, synchronization structures, object migration structures, etc.« [Lieberherr 1995, S. 1]

   Adaptive objektorientierte Software ist Software, die sich automatisch an sich ändernde *Kontexte* anpasst. Kontexte können sein das Verhalten, Klassenstrukturen der Implementierung, Synchronisationsstrukturen und Strukturen für die Objektmigration.

3. »A program is called *adaptable* if it can be easily changed. A program is called *adaptive* if it changes its behavior automatically according to its *context*.« [Lieberherr und Lopes 1995]

   Ein Programm ist *adaptierbar*, wenn es einfach geändert werden kann. Ein Programm wird *adaptiv* genannt, wenn es sein Verhalten automatisch entsprechend des *Kontextes* ändert.

4. »*Adaptable* systems can be adapted to a particular *deployment environment*, whereas *adaptive* systems adapt themselves to a *deployment enviroment*.« [Czarnecki und Eisenecker 2000, S. 397]

   Adaptierbare Systeme können an eine bestimmte *Einsatzumgebung* angepasst werden, wohingegen *adaptive* Systeme sich selbst an die *Einsatzumgebung* anpassen.

5. »*Adaptiv* ist ein Softwaresystem, das eigenes Verhalten selbständig evaluiert und ändert. Wenn die Evaluation zeigt, dass die Ziele nicht erreicht werden oder die Funktionalität des Systems suboptimal ist [Laddaga 2001]. Der Begriff „Funktionalität“ wird sehr breit definiert und umfasst sowohl die Anwendungslogik als auch die Datengrundlage, die Darstellung der Daten und die Interaktion mit anderen Systemen.« [Amberg *et al.* 2003, S. 13]

6. »*Self-adaptive* software evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible.« [Laddaga 1997] (SAS)

   Selbst-adaptive Software wertet ihr eigenes Verhalten aus und ändert dieses, wenn die Evaluierung anzeigt, dass die Software nicht ihrem Ziel folgt, oder wenn bessere Funktionalität oder Performance möglich ist.

7. »*Self-adaptive* software modifies its own behaviour in response to changes in its *operating environment*. By operating environment, we mean anything observable by the software system, such as end-user input, external hardware devices and sensors, or program instrumentation.« [Oreizy *et al.* 1999]

   Selbst-adaptive Software ändert ihr eigenes Verhalten in Reaktion auf Änderungen in der Umgebung, in der die Software betrieben wird (Betriebsumgebung). Unter der *Betriebsumgebung* verstehen wir alles, was die Software beobachten kann. Beispiele dafür sind Eingaben des Benutzers, externe Hardware-Geräte, Sensoren oder Programmanweisungen.

8. »*Adaptive* Anwendungen werden durch die Fähigkeit zur dynamischen Adaption von Anwendungsdaten, der Kommunikation sowie der Anwendungsstruktur in Reaktion auf Änderungen des *Kontextes* zur Laufzeit charakterisiert. Kontext bezeichnet die verfügbaren Ressourcen der Zugangsnetzwerke und Endgeräte sowie Benutzereinstellung und die jeweilige Anwendungssituation.« [Springer 2004]

Auch wenn die Definitionen in ihrer Formulierung und in der Verwendung von Begriffen sehr unterschiedlich sind, kann das positive Ergebnis festgestellt werden, dass eine (selbst-) adaptive Software Adaptionen vornimmt (also eine aktive Rolle hat), wohingegen die Begriffe einer adaptierbaren Software bzw. Adaptierbarkeit (adaptable/adaptability) klar die passive Rolle einer Software beinhalten. Damit wird durch die Software-spezifischen Definitionen mehr Klarheit geschaffen als durch die verfügbaren Übersetzungen und Enzyklopädieeinträge eingangs des Kapitels. (Dennoch ist in der Literatur [Seth und Kokar 2003], [Subramanian und Chung 2002a, b], [Soto und Khosla 2003] die Verwendung von »adaptability« im Sinne von Anpassungsvermögen/Adaptivität zu finden statt wie hier im Sinne von Adaptierbarkeit.)

Die Frage 3 auf Seite 23 ist demzufolge geklärt: Adaptivität bezeichnet nur Änderungen, die die Software vornimmt, und Punkt 3 ist der Adaptierbarkeit zuzuordnen (und »adapt·able« := »able to be adapted«; *nicht* »able to adapt«). Offen bleiben die Punkte 1 und 2 von Seite 23.

Die Definitionen werden nachfolgend näher analysiert und in Beziehung zueinander gebracht.

3.4.1. Besonderheit 5. Definition

Besondere Aufmerksamkeit soll innerhalb dieses Abschnittes der 5. Definition, die in [Amberg *et al.* 2003, S. 13] für »adaptive Software« gegeben wird, zugewendet werden. Es fällt auf, dass diese Definition im ersten Teil[9] der 6. Definition von [Laddaga 1997] gleicht. Wird die Definition genauer untersucht, so fallen zwei bedenkliche Dinge auf:

1. Bei der Übernahme der Definition von [Laddaga 2001] wird »self-adaptive« um das »self« auf »adaptive« gekürzt und

2. Prof. Amberg und Kollegen erweitern die Definition von [Laddaga 2001] um eine Deutbarkeit der Funktionalität, die nicht in der Intention der originalen Definition liegt — und nach Meinung des Autors außerdem in sich widersprüchlich ist.

Zur Abschwächung der ersten Kritik sei angemerkt, dass Prof. Amberg und Kollegen ihre Definition für *adaptive* und nicht für *selbst-adaptive* Software formuliert haben, weil das Projekt so hieß.[10] Doch der zweite Punkt bleibt bedenklich.

Die Definition, wie sie in [Laddaga 2001] gegeben ist und von [Amberg *et al.* 2003, S. 13] herangezogen wird, besitzt eine andere Intention. Sie lautet im Original folgendermaßen:

> »Self-adaptive software evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible.
>
> . . . This implies that the software has multiple ways of accomplishing its purpose, and has enough knowledge of its construction to make effective changes at runtime. Such software should include functionality for evaluating its behavior and performance, as well as the ability to replan and reconfigure its operations in order to improve its operation. Self adaptive software should also include a set of components for each major function, along with descriptions of the components, so that components of systems can be selected and scheduled at runtime, in response to the evaluators. It also requires the ability to impedance match input/output of sequenced components, and the ability to generate some of this code from specifications. In addition, DARPA seek this new basis of adaptation to be applied at runtime, as opposed to development/design time, or as a maintenance activity.« [Laddaga 2001]

Nebenbei: In der Tat entstammt diese Definition der DARPA, die 1998 in einer Ankündigung für das Forschungsprojekt SAS (SOL BAA SAS 98-12) propagiert wurde [Laddaga 1997] und sich in der 6. Definition wiederfindet.

[9] Bis zu der Literaturreferenz »[Laddaga 2001]«, die in dieser Form auch als Kennzeichnung in [Amberg *et al.* 2003, S. 13] zu finden ist.

[10] Telefonat mit Hrn. Wehrmann am 22. April 2004.

Widerspruch in der Definition. Es sei Prof. Amberg und Kollegen gestattet, eine Definition (oder Teile davon) zu nehmen und mit einer Umformulierung und Ergänzung für das eigene Verständnis in ihrer Arbeit zu verwenden. Jedoch dürfen sie in dem Fall nicht ein Beispiel für »adaptive Software« nennen, wie es auf Seite 20 in [Amberg *et al.* 2003] zu finden ist, weil es dann unweigerlich zu einem Widerspruch kommt. Ein Widerspruch besteht außerdem in der gesamten Definition. Zunächst das Beispiel:

> »Ein interessantes Beispiel der adaptiven Software wurde im Projekt OPERA umgesetzt [Lemlouma und Layaïda 2002]. Es wurde ein Framework aufgestellt, das es ermöglicht, die Webseiten für unterschiedliche Clientgeräte und Softwaresysteme zu transformieren, so dass jeder Client an seine Darstellungs- und Rechenmöglichkeiten angepasste Inhalte und Grafiken erhält. Das Framework ermöglicht damit die Erstellung der gerät- und systemunabhängigen Webseiten. Die Textinhalte für den Client werden durch XSL-Transformationen angepasst und die Grafiken abhängig von der durch den Client unterstützten Auflösung, Größe und Farbtiefe konvertiert.« [Amberg *et al.* 2003, S. 20]

Das Projekt OPERA [Lemlouma und Layaïda 2002] ähnelt sehr stark dem Transcoding Framework der TU Dresden [Hübsch *et al.* 2003], das Webseiten oder Medieninhalte in einer geräteunabhängigen Sprache — beim OPERA-Projekt ist dies SMIL[11] [Ayars *et al.* 2001], an der TU Dresden wurde dafür die DDL[12] entwickelt — kodiert und sie für Anfragen in die gerätespezifischen Sprachen wie z. B. HTML[13], WML[14] und cHTML[15] umwandelt. Dieser Vorgang heißt Transcoding, die Verwendung *einer* Ausgangssprache für Seiten in vielen Endgerätesprachen »Single-Source-Publishing«. Erkenntnisse zu Transcoding konnten auch bei DaimlerChrysler in [Vögler *et al.* 2004] gewonnen werden. Für ein weiteres Verstehen von Transcoding muss der Leser auf die referenzierte Literatur verwiesen werden, da ein genaueres Eingehen den Fokus der vorliegenden Arbeit nehmen würde.[16]

Die Kritik zur Definition von Amberg *et al.* [2003] im Einzelnen:

1. Im ersten Teil der Definition steht, dass das Verhalten (behavior) angepasst wird, aber im vorgenannten Beispiel OPERA werden Änderungen/Anpassungen an den Inhalten vorgenommen. Amberg und Kollegen nennen zwar Inhalte (Datengrundlage/Darstellung der Daten) als Definition von »Funktionalität«,

[11] Synchronized Multimedia Integration Language
[12] Dialog Description Language
[13] HyperText Markup Language
[14] Wireless Markup Language
[15] Compact HTML
[16] Auch herrscht bei den Gutachtern, die erster Adressat der vorliegenden Arbeit sind, Kompetenz auf diesem Sachgebiet, da vorgenannte Arbeiten im Rahmen ihres Bereiches entstanden sind.

aber an keiner Stelle in ihrer Definition wird die Anpassung (Adaption) der Funktionalität als Eigenschaft »adaptiver Software« angegeben, sondern nur die Anpassung des Verhaltens.

2. Im ersten Teil der Definition steht, dass das Verhalten evaluiert wird, aber im vorgenannten Beispiel OPERA findet keine Evaluation des Verhaltens der Software statt.

   Fakt ist, dass in dem Papier zum OPERA-Projekt [Lemlouma und Layaïda 2002] nur Inhaltsadaption mittels SMIL und ein Aushandeln/in-Erfahrung-bringen des Client-Profils (UPS) beschrieben wird (vgl. Abschnitt 4.2 in vorgenannter Literatur und siehe Abbildung 3.2 unten). Dabei findet nur ein Aushandeln der zu sendenden (ggf. adaptierten) Variante einer angeforderten Ressource zwischen Client und Server statt, indem beispielsweise das Client-Profil ausgewertet wird. Keineswegs wird das Verhalten der Software evaluiert/ausgewertet, indem geprüft wird, ob die adaptierten Inhalte das gewünschte Ergebnis auf dem Client erbracht haben. Eine Evaluation würde sich in dem Aktivitätsdiagramm derart äußern, dass entweder vom Client eine Bewertung zu der empfangenen Ressource an den Server gesandt wird oder der Server die generierte Ressource bewertet.

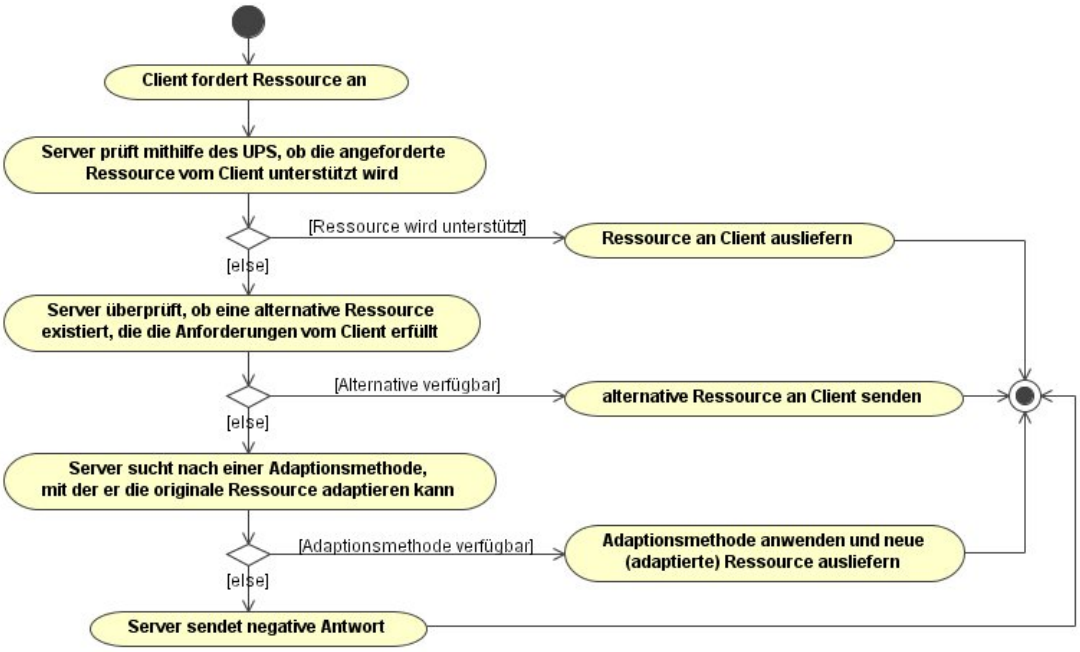

Abbildung 3.2.: Aktivitätsdiagramm für die Bearbeitung einer Client-Anfrage in OPERA; vergleichbar mit dem Transcoding-Framework der TU Dresden

Resümee. Die Definition von Amberg *et al.* [2003] ist in sich widersinnig. Zum einen wird eine andere Definition für den ersten Bestandteil herangezogen, die nichts mit der Sache zu tun hat, zum anderen ist die gesamte Definition widersprüchlich und ungenau, weil Verhalten und Funktionalität — definiert unter anderem als Datengrundlage — implizit gleichgesetzt werden. Nach Meinung des Autors zeigt dieses Beispiel einer sehr nachlässige Vorgehensweise für eine Begriffsbestimmung. Im restlichen Teil der Arbeit von Amberg *et al.* werden jedoch interessante Konzepte zu innovativen Benutzungsschnittstellen, für Softwareanalyse und -pflege sowie für Application Service Providing vorgestellt.

Im Rahmen dieses Kapitels werden zunächst die übrigen Definitionen analysiert.

3.4.2. Analyse der übrigen Definitionen

Die 1. Definition [Tekinerdogan und Aksit 1996] bezeichnet die Anpassbarkeit von Software in Reaktion auf sich geänderte Anforderungen an die Software. Diese Definition fällt in die Kategorie *Evolutions-Adaptierbarkeit* (im Gegensatz zur »Möglichkeit-Anpassbarkeit«, wie sie in Kapitel 4 betrachtet wird) und meint die nicht-funktionale Eigenschaft *Anpassbarkeit* (Änderbarkeit) eines Software-Systems, d. h. mit welchem Aufwand eine Software aufgrund anderer Anforderungen geändert werden kann. Ziel bei der Entwicklung sollte es bereits sein, die Software offen für Änderungen zu konstruieren.[17]

Die 2. Definition enstammt dem Buch »Adaptive Object-Oriented Software: The Demeter Method« [Lieberherr 1995]. Lieberherr versteht »adaptive Software« als Erweiterung von objektorientierter Software, bei der die Beziehungen zwischen den Funktionen und den Daten flexibel gehalten werden, d. h. Funktionen und Daten sind locker miteinander gekoppelt. Die Adaption/Anpassung ist dabei derart gegeben, dass die Software Änderungen in den Anforderungen — in diesem Fall sind dies geänderte Objektstrukturen — berücksichtigen kann. In diesem Buch propagiert der Autor dafür eine neue Softwareentwicklungsmethode, die er *Demeter* nennt. Die Demeter-Methode folgt dem »Polya's Inventor's Paradox«[18] (PIP, formuliert im »kleinen Polya« [Polya

[17] Neben der Anpassbarkeit (Änderbarkeit) gibt es noch weitere nicht-funktionale Eigenschaften einer Software wie beispielsweise Wiederverwendbarkeit, Interoperabilität, Effizienz, Testbarkeit, Wiederverwendbarkeit und Zuverlässigkeit (vgl. [Buschmann *et al.* 1998, Kap. 6.4]). Lewandowski [2003] nennt zusätzlich Performance, Time-to-Market, Gesamtkosten der Lösung, Verfügbarkeit, Robustheit und Einfachheit. Allgemein werden diese Eigenschaften bzw. Anforderungen an Software auch als »ilities« bezeichnet, da die Wörter im Englischen die Endung »-ility« besitzen [Filman *et al.* 2002].

[18] Der Mathematiker George Polya formulierte 1949 [Polya 1949], dass es oft leichter für die Problemlösung ist, ein spezielles Problem zu generalisieren und die Lösung des generellen Problems auf das spezielle Problem anzuwenden. Das »Paradoxe«

1949, 1995]), das für die Lösung eines konkreten Problems empfiehlt, dieses zu generalisieren. Für adaptive Software, die mit der Demeter-Methode entwickelt wird, bedeutet dies, dass nicht ein bestimmtes Programm geschrieben wird, sondern das Programm beschreibt eine Kategorie/Familie/Klasse von objektorientierten Programmen, die alle eine bestimmte Form besitzen, deren Details allerdings offen bleiben.

Die 3. Definition [Lieberherr und Lopes 1995] ist die erste, die eine gute Begriffsunterscheidung für adaptive und adaptierbare Software gibt. Lieberherr und Lopes verstehen unter adaptierbarer Software das Gleiche wie Tekinerdogan und Aksit [1996], die dafür eine Art Kennzahl Adaptierbarkeit formulieren (vgl. 1. Definition). Als adaptiv wird ein Programm angesehen, dass sein Verhalten an den Kontext anpasst. Kontext wird nicht genau definiert. Eine erste Deutung lässt den Kontext-Begriff vermuten, wie er in der Forschungsrichtung Context-Awareness vorzufinden ist, z. B. Orts- und Temperaturkontext, Display- und Speichergröße des Endgerätes. Dem ist jedoch nicht so! Dies wird bereits allein daran deutlich, dass der Mitautor der Definition, Lieberherr, in der 2. Definition (vorhergehende Definition) Kontext als Eigenschaften der Software deklariert (Klassenstrukturen, Synchronisationsstrukturen). Diesem Aspekt soll nähere Aufmerksamkeit im nächsten Abschnitt (3.5.1) zugewendet werden.

Einen ähnlichen Wortlaut in ihrer Begriffsbestimmung wie bei der Definition zuvor schlagen Czarnecki und Eisenecker [2000] in der 4. Definition ein. Als Anpassungsziel wird hier jedoch nicht der »Kontext« benannt, sondern die »Einsatzumgebung« (deployment environment). Einsatzumgebung suggeriert aufgrund des Wortes »deployment« eine Anpassung/Optimierung der Parameter der Software an die Systemumgebung, in der die Software deployt wird. Dies lässt viele Interpretationen zu. Bei einem Webserver wären das beispielsweise die Konfiguration der Anzahl der vorinstanzierten Threads oder gar die Auswahl des Multi-Processing-Modules des Apache Webservers [ASF 2004], im Rahmen des Software-Deployments von Web-Anwendungen in einem Application-Server z. B. auch Ressourcendefinitionen wie die der benutzten Datenbank. Diese Vermutung trifft nur teilweise zu. Czarnecki und Eisenecker sehen ihre Definition im Kontext von *Metaprogramming*, das sie als Technik für adaptive Systeme verstehen. Hinter dem Begriff *Metaprogramming* verbirgt sich das Schreiben von Programmen, die sich oder andere Programme repräsentieren und verändern können [Czarnecki und Eisenecker 2000]. Als Beispiele für Metaprogramme, die andere Programme verändern, werden Compiler, Interpreter, Programmgeneratoren (z. B. Makros), aspektorientierte Programmierung und Optimierer genannt, da diese ein Programm als Eingabe nehmen und ein anderes Programm als Ausgabe produzieren.

---

daran ist, dass normalerweise davon ausgegangen wird, dass das Lösen eines generellen Problems schwieriger ist als das Lösen des gerade vorhandenen, speziellen Problems. Laut Polya trifft jedoch oft genau das Gegenteil zu.

Als Beispiele für adaptive Systeme nennen Czarnecki und Eisenecker automatische Prozesse, die verschiedene Aspekte von Komponenten modifizieren, Komponenten konfigurieren und diese zu einem System zusammenbauen. Neben dieser automatischen Konfiguration von Systemen für eine spezifische Einsatzumgebung wird auch die Anpassung an eine sich ändernde Umgebung für adaptive Software erwähnt; so beispielsweise die Änderung der Regeln für das Caching und die Zeitplanung (scheduling) in Abhängigkeit der gegenwärtigen Auslastung des Systems und die Generierung von Adaptern und Wrappern, bevor eine neue Umgebung kontaktiert wird.

Die 6. Definition [Laddaga 1997], in voller Länge, wurde bereits auf Seite 29 näher beschrieben. Ihre weitere Analyse erfolgt zusammen mit der nachfolgenden Definition.

Die 7. Definition [Oreizy *et al.* 1999] (nachfolgend kurz Oreizy-Definition genannt) erschien in der Mai/Juni-Ausgabe 1999 von *IEEE Intelligent Systems* in einem Schwerpunktthema für »selbst-adaptive Software«. Das ist insofern wichtig, dass dieses Schwerpunktthema im Zeichen der DARPA-Definition für »self-adaptive Software« (SAS) [Laddaga 1997] steht (vgl. den Einleitungsartikel [Laddaga 1999]). Demnach müssten alle im Rahmen des Schwerpunktthemas erschienenen Artikel das gleiche Verständnis für selbst-adaptive Software besitzen. Jedoch weisen DARPA-Definition (Nr. 6) und Oreizy-Definition (Nr. 7) eine Unstimmigkeit auf. Bei beiden steht zwar die Veränderung des Verhaltens im Vordergrund, in [Oreizy *et al.* 1999] allerdings allgemein als Reaktion auf Änderungen in der Betriebsumgebung, bei der DARPA-Definition andererseits explizit nach Evaluation des Verhaltens oder wenn allgemein bessere Performance verfügbar ist. Die Oreizy-Definition zielt nur auf das Beobachten (und Evaluieren) der *Umgebung* ab, der DARPA-Definition folgend muss das *Verhalten* der Software beobachtet und evaluiert werden. Damit können aus den zwei Definitionen nicht die gleichen Eigenschaften für selbst-adaptive Software abgeleitet werden, obwohl die Definitionen im gleichen Kontext (dem Schwerpunktthema in *IEEE Intelligent Systems*) stehen.

Anderer Art ist die 8. Definition von Springer [2004] — sowohl bezüglich Klarheit als auch in der Einschränkung, welche Anwendungsbereiche der Begriff umfasst. Und die Domäne ist eine ganz andere. Während die vorangegangenen Definitionen lediglich postulierten, dass sich die Software anpasst, nennt Springer klar als Adaptionsgegenstände die Anwendungsdaten, die Kommunikation und die Anwendungsstruktur. Ebenso klar ist der Begriff des Kontextes, der ausschlaggebend für eine Adaption ist; Springer beschränkt diesen auf die Ressourcenverfügbarkeit, die Benutzereinstellungen und die Anwendungssituation. Als solche umfasst dieser Typ einer »adaptiven Software« ubiquitäre Anwendungen, die aufgrund von Mobilität und einer heterogenen sowie ressourcenbeschränkten Laufzeitumgebung Anpassungen an der Anwendungsstruktur, der Kommunikation sowie den Anwendungsdaten vornehmen (müssen). Anzumerken ist, dass

diese Definition als Einschränkung des Themas in einer Dissertation benutzt wurde. Keinesfalls wird damit eine allgemeine Definierung für »adaptive Software« beabsichtigt.

Resümee. Wie auch die in Kapitel 3.2 auf Seite 24 aufgeführten Beispiele zeigen die vorliegenden Definitionen ein breites Spektrum an Adaptionen in Software und damit einhergehend ein breites Verständnis für »adaptive Software«. Aufgrund des allgemeinen Charakters einer Definition ist hier zudem das Spektrum breit gefächerter und unklarer, da die Definitionen zu viel Interpretation zulassen. Einige dieser werden nachfolgend noch näher untersucht.

3.5. Analyse: Auslöser für die Adaption

In den zuvor aufgeführten und analysierten Definitionen konnten vielerorts die Begriffe Kontext sowie Einsatz- und Betriebsumgebung als verantwortlich für eine Adaption erkannt werden. Eine Änderung oder das Vorhandensein dieser (gemeint ist, dass eine Anpassung *an* den Kontext/Betriebsumgebung und nicht wegen einer *Änderung* stattfindet), führt zu einer Anpassung des Systems.

Für den Begriff Kontext wurde bereits zuvor festgestellt, dass dieser in einem unterschiedlichen Zusammenhang verwendet wird. Diese Untersuchung soll hier fortgeführt werden. Außerdem stellt sich die Frage, welchen Zusammenhang es zwischen Kontext, Einsatz- sowie Betriebsumgebung gibt.

3.5.1. Kontext

Bei den Definitionen 2, 3 und 8 wird der Kontext als ausschlaggebend für eine Adaption angeführt. Würden nur die 3. und 8. Definition allein betrachtet werden, so könnte der Eindruck entstehen, die zwei Definitionen meinen das Gleiche, wobei die 8. Definition etwas eingeschränkten Charakter bezüglich des Kontextes besitzt (Springer beschränkt den Kontext auf die Ressourcenverfügbarkeit, die Benutzereinstellungen und die Anwendungssituation.). Erst die Analyse des genaueren Umfelds der 3. Definition und eine weitere Recherche in [Lieberherr 1998] — und auch die Definition 2, die ebenso von Lieberherr stammt — zeigen, dass vorgenannte zwei Definition nicht vergleichbar sind.

Beide Definitionen folgen einer anderen Intention und haben einen unterschiedlichen Adaptionsgegenstand. Springer nennt zwar neben Daten und Kommunikation auch die Anwendungsstruktur als Adaptionsgegenstand, diese ist jedoch keinesfalls mit dem Adaptionsgegenstand zu vergleichen, wie ihn Lieberherr verwendet (»software that adapts to« und »changes its behavior«). Mit dem Gegenstand Software meint Lieberherr objektorientierte Software, die generisch genug ist,

um für verschiedene Kontexte einsetzbar zu sein. Lieberherr möchte flexible Software schaffen, bei der keine feste Kopplung zwischen Funktionen und den Daten besteht. Für ein bestimmtes Einsatzgebiet können beispielsweise Softwarebibliotheken parametrisiert werden, Algorithmen setzen keine bestimmte Datenstruktur als Eingabe voraus, sondern können auf einer Vielzahl an Strukturen operieren (vgl. [Lieberherr 1995]).

Lieberherrs adaptive Software hat als Zielgruppe Programmierer, die die Änderungen, die aufgrund der Evolution einer Software (bzw. einer iterativen/evolutionären Vorgehensweise bei der Softwareentwicklung, wie beispielsweise im Spiralmodell [Boehm 1986]) vorzunehmen sind, gering halten wollen. Adaptive Software umfasst eine Softwareentwicklungsmethode, mit der sich Software für verschiedene Einsatzzwecke entwerfen lässt. Die Demeter-Methode führt dazu, dass sich die Software selbst anpasst und nur wenige Änderungen an ihr durch den Entwickler durchzuführen sind. Dies führt zu einer höheren Adaptierbarkeit der Software.

Dahingegen stellt Springer Mechanismen zusammen, die die Daten, Kommunikation und Anwendungsstruktur an die beschränkten Eigenschaften der Ausführungsumgebung anpassen (z. B. Transcoding, Caching, abgekoppeltes Arbeiten). Die Software passt sich nicht selbst an, sondern nimmt Anpassung unter anderem an den Daten und an der Kommunikation vor (diese Mechanismen werden in der vorliegenden Arbeit in den Kapiteln 4.2 und 4.3 beschrieben).

Damit nimmt die Software in beiden Definitionen eine Anpassung an den Kontext vor, jedoch auf nicht vergleichbare Weise. Der Kontext-Begriff ist weit fassbar, er ist gar kontextabhängig.

Lieberherrs Kontext-Begriff. Lieberherrs Definition für adaptive Software ist eine Definition auf hoher Ebene, die mit klarem Verstand erfasst werden muss. Denn je nachdem, was unter Kontext verstanden wird, führt diese Definition zu verschiedenen Arten von Adaptivität. In [Lieberherr 1998] weist der gleiche Autor auch auf diesen Umstand hin und nennt Kontext-Beispiele, die in verschiedenen Arten für Adaptivität resultieren. So sind neben Daten- und Klassenstrukturen, wie sie im Rahmen der Demeter-Methode als Kontext verwendet werden, als Kontext auch denkbar [Lieberherr 1998]:

- *Laufzeitumgebung:* Abhängig von anderen Prozessen, mit denen das Programm läuft, optimiert das Programm Parameter für eine bessere Performance.

- *Verteilungseigenschaften:* Anstatt die Objektmigration fest in das Programm zu integrieren, wird diese als Kontext gesehen. Auf diese Weise kann das Programm mit verschiedenen Objektmigrationsstrategien betrieben werden.

- *Software-Architektur:* Anstatt die Verbindungen zwischen den Softwarekomponenten fest in dem Programm zu verdrahten, werden diese als Kontext gesehen. Damit kann dasselbe Programm mit einer Klasse von Software-Architekturen arbeiten.

- *Ressourcen:* Ressourcen wie Drucker, Displays, Name Server, etc., werden nicht in dem Programm fest verdrahtet, sondern dienen ebenso als Kontext. Damit funktioniert dasselbe Programm mit einer Reihe von Ressourcen.

- *Ausnahmen und Fehler:* Auch Ausnahmen und Fehler können als Kontext behandelt werden. Wenn die Ausnahmebehandlung nicht im Programm fest verankert wird, kann dasselbe Programm mit einer Familie von Ausnahmebehandlungsschemata arbeiten.

Die Spanne von Adaptivität, die sich durch die Aufzählung von Kontextarten durch Lieberherr ergeben, ist sehr breit. Sie reicht von *eigenständiger* Optimierung von Parametern über Ressourceneinstellungen, die *durch den Benutzer* vorgenommen werden können (*keine* Eigenständigkeit der Software), bis hin zu Ausnahmebehandlungsschemata, die (während der *Entwicklung*?) variabel in das Programm eingefügt werden können (zielt wahrscheinlich auf aspektorientierte Programmierung [Kiczales *et al.* 1997] ab).

Soweit einige Beispiele zu Kontext, wie Lieberherr sie für adaptive Software im Rahmen seiner Definition anführt.

Dieses Verständnis von Kontext ist ein anderes als es die in den letzten Jahren aktuelle Forschungsrichtung für Context-Awareness besitzt.

Springers und auch üblicher Kontext-Begriff. Das Verständnis, wie es Springer und auch die Forschungsgemeinde für Kontext besitzt, ist ein anderes. In der Vergangenheit wurde vor allem der Ortskontext (Person A befindet sich an Ort Y) im Rahmen von Mobile Computing für lokationsbasierte Dienste als Kontext betrachtet. Wie jedoch bereits 1999 Schmidt *et al.* hervorgehoben haben, gibt es zudem andere wichtige Kontextarten wie sozialen Kontext und Aufgabenkontext des Nutzers [Schmidt *et al.* 1999b]. In der Literatur und auch an der TU Dresden wurde der Kontext-Begriff mittlerweile umfassend untersucht. So konnten beispielsweise in der angefertigten Arbeit [Goslar 2001] die vier Kontextklassen physikalischer, technischer, persönlicher und situationsbezogener Kontext identifiziert werden (vgl. Tabelle 3.3 auf der nächsten Seite).

Ist diese Einteilung in die Kontextklassen womöglich unvollständig, da sie Lieberherrs Kontexte nicht berücksichtigt? Offenbar nicht. Denn wird die weit verbreitete Definition von [Dey 2000] betrachtet, dann wird deutlich, worauf das Verständnis von Kontext und damit kontextsensitiven Anwendungen in der Wissenschaft hinausläuft:

Kontextklasse	*Beispiele*
physikalischer Kontext	Ort, Zeit, Temperatur, Lichtintensität
technischer Kontext	Display- und Speichergröße des Endgerätes, Datenrate und Verzögerung des genutzten Kommunikationskanals, verfügbare Energie
persönlicher Kontext	Adresse, Telefonnummer, Termine, bekannte Personen, Voreinstellungen für Anwendungen
situationsbezogener Kontext	Aktivität, Rolle und Aufgabe des Benutzers

Tabelle 3.3.: Einteilung von Kontext in vier Klassen [Goslar 2001] (zitiert nach [Springer 2004])

> »Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and application themselves.« [Dey 2000]
>
> Kontext ist irgendeine Information, die für die Charakterisierung der Situation einer Entität genutzt werden kann. Dabei bezeichnet der Begriff Entität eine Person, einen Ort oder ein Objekt, das für die Interaktion zwischen einem Nutzer und einer Anwendung von Bedeutung ist, eingeschlossen den Nutzer und die Anwendung.

Kontext kann in der Tat irgendeine Information sein. Die Informationen können zum einen in Form von Ein- und Ausgabedaten von Anwendungen explizit vorliegen. Jedoch wird der überwiegende Teil der Informationen in irgendeiner Weise gewonnen, d. h. Kontext bezeichnet Informationen, die implizit vorliegen (». . . all but the explicit input and output of an application« [Lieberman und Selker 2004]). Kontext wird gewonnen durch die Messdaten physikalischer Sensoren oder die Werte logischer Sensoren (z. B. durch Extraktion, Benutzereingaben und -beobachtung oder aus Datenbanken) sowie durch Abstraktion und Verknüpfung dieser Basisinformationen zu höherwertigen Kontextinformationen.

Für ein besseres Verstehen des Kontext-Begriffs helfen außerdem die Ausführungen, die in [Kadner 2004a, S. 3] und [Springer 2004, S. 9] zu finden sind:

Entität[19] Eine Entität ist eine Person, ein Ort, abstrakt gesagt jedes Objekt, das für den Nutzer von Interesse ist. Dies schließt den Nutzer selbst mit ein. Es muss sich dabei allerdings nicht um ein physisches Objekt handeln, sondern kann auch ein »virtuelles« Objekt sein, etwa die logische Repräsentation einer Präsentation.

[19] In Abwandlung der originalen Formulierung »Entity« wird in der vorliegenden Arbeit ein deutscher Begriff verwendet.

Kontext-Variable Eine Kontext-Variable hat einen bestimmten Kontexttyp und besitzt einen entsprechenden Kontextwert. Der Typ einer Kontext-Variablen ist z. B. die Zeit, ein Ort oder eine Aktivität. Als Wert kommen einfache oder zusammengesetzte Informationen in Frage.

Kontextinformation Der semantische Zusammenhang zwischen Entitäten und Kontext-Variable wird als Kontextinformation bezeichnet. Die einfache Angabe einer Temperatur beispielsweise ist noch keine Kontextinformation; erst durch den Zusammenhang zwischen der Temperatur und einem Ort (an dem diese Temperatur herrscht) wird daraus eine Kontextinformation.

Kontextsensitivität Eine Verarbeitung ist kontextabhängig oder kontextsensitiv, wenn sie die Fähigkeit zur Wahrnehmung von Kontext besitzt und ein definierter Zusammenhang zwischen Kontextwerten und Verarbeitungsergebnissen besteht, d. h. sich in Reaktion auf Änderungen des Kontextes die Verarbeitungsergebnisse in definierter Weise ändern.

Ein weiterer Aspekt ist der, dass Kontext vor allem im Rahmen der Interaktion zwischen Anwendung und Nutzer gewonnen wird. Kontextsensitive Anwendungen verwenden diesen in irgendeiner Weise weiter, um den Nutzer zu unterstützen. Anwendungen, die Kontext nutzen, reichen dabei von einfachen Anfragen (z. B. »zeige mir meine geographische Position an«) bis hin zu vollautomatisierten »intelligenten Umgebungen« (vgl. [Dey 2000] und [Löschau 2002]).

Resümee. Im Zusammenhang mit »adaptiver Software« besteht ein vollkommen unterschiedliches Verständnis für Kontext. Mit Kontext wird nicht nur die Nutzungssituation bezeichnet, wie sie der Forschungsbereich Context-Awareness für kontextsensitive Anwendungen versteht, sondern auch Dinge, die im Kontext (Zusammenhang) von Programmen/Software (bei deren Entwicklung?) stehen. Lieberherr [1998] nannte hierfür unter anderem Fehlerbehandlungsschemata und Ressourcendefinitionen als Beispiele.

Tabelle 3.4 auf der nächsten Seite stellt für zwei ausgewählte »Kontexte« das gegensätzliche Verständnis für Adaptivität einander gegenüber.

3.5.2. Betriebsumgebung

Neben Kontext wurde auch die Betriebsumgebung (operating environment) als verursachend für eine Adaption in der 7. Definition genannt. Die Verfasser der Definition, Oreizy *et al.* [1999], verstehen unter Betriebsumgebung der Software alles, was die Software beobachten kann. Als Beispiele nennen sie Eingaben des Nutzers, externe Hardware-Geräte, Sensoren und Programmanweisungen.

»Anpassung an Kontext« bedeutet bei	*Lieberherr*	*Allgemeinheit*
für Kontext *Fehler und Ausnahmen*	Ausnahmebehandlung wird nicht im Programm fest verdrahtet, sondern das Programm kann mit einer Vielzahl an Fehlerbehandlungs-schemata arbeiten	im Sinne einer Selbst-Heilung werden die fehlerhaften Komponenten des Systems identifiziert und ausgetauscht
für Kontext *Verteilungseigenschaft*	die Objektmigration wird nicht fest im Programm integriert, sondern als Kontext (Variablität) angesehen, sodass das Programm mit verschiedenen Objektmigrationsstra-tegien betrieben werden kann	Adaptivität im Sinne von Verteilungseigen-schaften / Objektmigration bedeutet lediglich, dass das Programm selbst eine Migration vornimmt, um bessere Performance zu liefern (vgl. Seite 105)

Tabelle 3.4.: Gegenüberstellung von unterschiedlicher Adaptivität je nach Verständnis von »Kontext«

Im Folgenden wird versucht, die vorgenannten Beispiele für Betriebsumgebung zu deuten:

Benutzereingaben Die Benutzereingaben stellen keine Kontextinformation dar. Für das Beispiel eines Logins des Nutzers sind Benutzername und Passwort normale Daten, die im Rahmen der Funktion der Anwendung verarbeitet werden. Erst die Tatsache, dass sich der Nutzer einloggt, stellt eine Kontextinformation dar. Wenn alle Daten, die vom Nutzer während der Benutzung der Anwendung eingegeben werden, als Kontextinformation angesehen werden und das Verarbeitungsergbnis von diesen abhängig ist, dann wäre jede Anwendung kontextsensitiv (einfaches Beispiel Taschenrechner, der zwei Zahlen korrekt addiert).

Allerdings wird Kontext unter anderem aus Eingabedaten gewonnen. Dabei bilden jedoch nicht die Eingabedaten selbst eine Kontextinformation, sondern die Metainformationen, die aus den Daten hervorgehen, z. B. dass das Passwort dreimal falsch eingegeben wurde. Deshalb auch ist die Grenze zu gewöhnlichen Benutzereingaben für diese Kontextgewinnung fließend (das gibt auch Dey [2000] zu). Beispielswiese kann für den Login-Prozess der Nutzername doch als Kontextinformation verstanden werden, weil er den Nutzer für eine an ihn angepasste Nutzeroberfläche notwendigerweise identifiziert.

externe Hardware-Geräte Es ist nicht klar verständlich, welche Infor-

mationen bezüglich der externen Hardware-Geräte gemeint sind, da von den o. g. Autoren für diese Begriffe keine weiteren Ausführungen erfolgen. Wird eine Anwendung betrachtet, die auf einer bestimmten Hardware läuft oder die eine gegebene Hardware in der Umgebung hat, dann können das Vorhandensein dieser oder die Eigenschaften, die die Hardware näher beschreiben (nicht unbedingt *Attribute*, wie der Fabrikationsname oder die Seriennummer, sondern eher Werte, die sich auch ändern, z. B. die Auslastung oder die verbleibende Kapazität eines Energielieferanten), als Kontext verstanden werden.

Sensoren Die Angabe, dass die Anwendung Sensoren beobachten kann, ist von ihrer Formulierung her in der 7. Definition unsinnig. Wahrscheinlich meinten Oreizy *et al.*, dass die Betriebsumgebung vor allem über Sensoren beobachtet werden kann.

Für die Sensoren ergibt sich eine große Gemeinsamkeit mit dem Kontext. So kann physikalischer Kontext durch Sensoren gemessen werden. Wenn eine Software ihre Umgebung mittels Sensoren beobachtet, dann entspricht alles Beobachtbare Kontextinformationen; in erster Linie Umgebungskontext des Sensors, der der Entität Anwendung zugeordnet werden kann. Auch wenn der Sensor in der Anwendung platziert ist, kann das Beobachtete dem Nutzer als Kontext zugeschrieben werden.[20]

Programmanweisungen Wie auch für die Hardware-Geräte ist hier nicht klar ersichtlich, was Oreizy *et al.* mit Programmanweisungen im Zusammenhang mit »was Software beobachten kann« meinen. Programmanweisungen, wie sie vom Programmierer im Quellcode platziert werden, sind hierbei bedenklich. Vielleicht meinen die Autoren der Definition mit Programmanweisungen Kommandos des Nutzers. Diese können nicht unbedingt als Kontextinformationen verstanden werden; aus ihnen kann jedoch Kontext gewonnen werden (vgl. Punkt Benutzereingaben).

Bis auf einige kleine Ausnahmen entspricht die Betriebsumgebung allem Anschein nach Kontext. Dennoch ist eine Gleichsetzung von Kontext und Betriebsumgebung nicht ohne Bedenken möglich. Informationen zum Kontext und zur Betriebsumgebung werden zwar beide größtenteils über Sensoren gewonnen. Aus dieser Gemeinsamkeit kann jedoch nicht die übergeordnete Gemeinsamkeit geschlussfolgert werden. Zum einen ist die Kontextgewinnung nicht nur auf Sensoren

[20] In dem Zeitschriftenartikel [Rademacher 2004] zu Organic Computing wird ein interessantes Beispiel gegeben: »Wenn sich die Scheiben in einem Mietwagen beschlagen und der Fahrer am Armaturenbrett herumfingert, bietet ein OC-System Hilfe an«, meint Prof. Peter E. Hofmann, Leiter des Competence Center Elektrik/Elektronikarchitektur von DaimlerChrysler. Hierbei werden aus den Sensorinformationen die höherwertigen Informationen, dass der Nutzer anscheinend Hilfe benötigt, ermittelt.

beschränkt, sondern schließt auch andere Quellen ein (beispielsweise Browsertyp und verarbeitbare Datenformate des Clients bei einer Anfrage an einen Webserver; diese Angaben sind als Metainformationen einer Anfrage beigefügt). Zum anderen werden Kontextinformationen im Rahmen der Mensch-Maschine-Kommunikation ermittelt, Oreizy *et al.* [1999] betrachten dahingegen alles, was durch die Software beobachtbar ist. Deshalb ist ihr Typus Anwendung zum einen nicht auf die Mensch-Maschine-Kommunikation beschränkt, bei der die Software durch Nutzung impliziter Informationen ihr Verhalten für den Nutzer verbessert oder kontextsensitive Dienste anbietet, zum anderen werden nicht nur *Kontext*informationen betrachtet.

In [Oreizy *et al.* 1999] skizzieren die Autoren ein Beispiel (es ist eher eine Vision, die sie als Aufhänger für ihren Artikel verwenden) für eine selbst-adaptive Software, die durch Berücksichtigung der Betriebsumgebung ein besseres Verhalten zeigt. Das skizzierte Beispiel umfasst unbemannte Flugzeuge im Einsatz, die selbstständig andere Software nachladen, wenn sie erkennen, dass sich ihre Mission geändert hat. Die ursprüngliche Mission der unbemannten Flugzeuge besteht darin, einen Flugplatz zu zerstören, der nicht verteidigt wird. Während der Ausführung der Mission erkennen spezielle Sensoren, dass der Flugplatz doch Boden-Luft-Raketen für die Abwehr besitzt. In Folge dieser neuen Erkenntnis teilt sich die Flugzeugtstaffel auf und plant die Mission neu, wobei der eine Teil der unbemannten Flugzeuge zusätzliche Komponenten nachlädt, die spezielle Algorithmen zur Erkennung von Abschussstationen für Boden-Luft-Raketen enthalten. In diesem Beispiel stellen die zusätzlich vorhandenen Boden-Luft-Raketen die (noch entfernte) geänderte Betriebsumgebung dar.

Ist diese Anwendung kontextsensitiv, d. h. benutzt sie Kontext zur Verarbeitung? Die Abschussstationen für die Boden-Luft-Raketen sind kein Kontext, sondern Entitäten. Ebenso ist der Flugplatz eine Entität. Erst die Tatsache, dass sich die Abschussstationen auf dem Flugplatz befinden, also die Beziehung zwischen zwei Entitäten, ist eine Kontextinformation, die die Situation des Flugplatzes näher beschreibt.[21] In diesem Anwendungsbeispiel werden also Kontextinformationen genutzt und ein davon abhängiges Verarbeitungsergebnis erzielt. Das Beispiel der umbemannten Flugzeuge von Oreizy *et al.* [1999] beschreibt eine kontextsensitive Anwendung.

Resümee. Der Begriff Betriebsumgebung, wie ihn Oreizy *et al.* als ausschlaggebend für eine Adaption benutzen, kann in eingeschränk-

[21] Die Überlegungen, dass nicht nur der semantische Zusammenhang zwischen Entität und Kontext-Variable eine Kontextinformation beschreibt, sondern auch die Beziehung, die zwischen zwei Entitäten aufgestellt werden kann, wurden in Gesprächen und E-Mail-Verkehr mit Kay Kadner ersichtlich. Diese essentielle Erkenntnis ist in keiner dem Autor bekannten Arbeit zu Kontext/Context-Awareness formuliert (vgl. Seite 38 und [Kadner 2004a], [Löschau 2002], [Goslar 2001], [Dey 2000], [Springer 2004]).

ter Form mit Kontext gleichgesetzt werden. Im Rahmen der Deutung der Beispiele für die Betriebsumgebung mussten zwar einige Bedenklichkeiten eingeräumt werden, jedoch entstammen diese der Tatsache, dass die Beispiele nicht näher umschrieben sind und so schwer richtige Deutungen möglich sind.

Neben den Bedenklichkeiten gibt es einen kleinen Unterschied bei einem Vergleich von Kontext und Betriebsumgebung. Kontext-Verarbeitung wird im Rahmen der Mensch-Maschine-Kommunikation definiert, die Betriebsumgebung umfasst jedoch alle Informationen, die von der Anwendung beobachtbar sind. Besonders deutlich wird dieser Unterschied in dem Beispiel der unbemannten Flugstaffel, die einen Flugplatz zerstören soll. Es gibt keine Mensch-Maschine-Kommunikation, sondern lediglich die »destruktive Interaktion« zwischen dem sich verteidigendem Flugfeld und der unbemannten Angreifer-Flugstaffel.

Wenn Kontext und Betriebsumgebung wirklich gleichgesetzt werden können, dann muss der Kontext-Begriff besser definiert werden, sodass er nicht nur auf die Interaktion eines Nutzers mit der Anwendung beschränkt ist. Ebenso müssten die theoretischen Grundlagen für Kontext, insbesondere dass eine Kontextinformation nicht nur durch den semantischen Zusammenhang zwischen Entität und Kontextvariable, sondern auch durch die Beziehungen zwischen Entitäten gegeben ist, einer besseren wissenschaftlichen Untersuchung unterzogen werden.

3.5.3. Einsatzumgebung

Czarnecki und Eisenecker geben als ausschlaggebend für eine Adaption die Einsatzumgebung (deployment environment) in der 4. Definition an. Anders als bei Kontext und Betriebsumgebung bei den Definitionen zuvor werden weder der Begriff der Einsatzumgebung näher definiert noch Beispiele dafür genannt. Deshalb muss eine Deutung mithilfe der Beispiele, die Czarnecki und Eisenecker für adaptive Software angeben, erfolgen.

Werden die Beispiele für adaptive Software analysiert, dann kann für die Einsatzumgebung, an die sich die Software selbst anpasst, eine breite Ausprägung bezüglich der Informationsarten und der -herkunft geschlussfolgert werden. Es hat den Anschein, dass die Einsatzumgebung eine Mischung der zuvor analysierten Begriffe Kontext und Betriebsumgebung ist! So stellt beispielsweise die Auslastung des Systems eine *Kontext*information dar, auf Basis derer eine Adaption des Caching- und Scheduling-Verhaltens stattfindet. Dahingegen ist für die Generierung von entsprechenden Adaptern und Wrappern für eine neue Umgebung ein ganz anderer Informationstyp, der nicht konform zu dem Kontext-Begriff von Springer und der aktuellen Forschung um Context-Awareness ist, für den Adaptionsprozess notwendig. Eher ist dieser Informationstyp mit den Kontexten von Lieberherr vergleichbar,

denn auch Lieberherr benötigt für seine »adaptive Software« Informationen zu den Daten- und Klassenstrukturen. Für die automatischen Prozesse, die verschiedene Aspekte von Komponenten modifizieren, Komponenten konfigurieren und diese zu einem System zusammenbauen, kann an dieser Stelle keine Analyse bezüglich der notwendigen Informationen erfolgen, da für eine Deutung durch den Autor die vorgenannten Szenarien von Czarnecki und Eisenecker noch mehr hätten umschrieben werden müssen. Nichtsdestotrotz kann ein Fazit gezogen werden:

Resümee. Durch die Analyse der von Czarnecki und Eisenecker für die 4. Definition genannten Beispiele kann eine breite Deutung des Begriffs Einsatzumgebung vorgenommen werden. Einsatzumgebung umfasst zum einen Teile des Kontext-Begriffs aus dem Forschungsbereich Context-Awareness (Abschnitt 3.5.1), ist aber auch vergleichbar mit Lieberherrs Kontextarten (Abschnitt 3.5.1), und der Betriebsumgebung von Oreizy *et al.* [1999] (Abschnitt 3.5.2). So betrachtet Lieberherr Klassen- und Datenstrukturen als Kontext, der Forschungsbereich Context-Awareness ganz sicher nicht. Für Czarnecki und Eisenecker stellen Klassen- und Datenstrukturen (bei der Generierung von Adaptern und Wrappern) die Betriebsumgebung dar, für Lieberherr ist es »Kontext«.

3.5.4. Zusammenhang

Im Rahmen dieses Unterkapitels wurden die Begriffe Kontext, Betriebs- und Einsatzumgebung untersucht, da diese als ausschlaggebend für eine Adaption in den Definitionen genannt werden. Dabei konnte festgestellt werden, dass zum einen Parallelen existieren oder die Begriffe fast gleichbedeutend sind (Kontext und Betriebsumgebung).

Zum anderen existieren für den Kontext-Begriff mit dem Verständnis von Lieberherr einerseits und dem der Forschungsgemeinde um Context-Awareness andererseits zwei vollkommen verschiedene Verwendungen dieses Begriffs im Rahmen »adaptiver Software«. Kontext, wie ihn die Forschungsrichtung Context-Awareness (CA) versteht, sind Informationen, die implizit oder explizit gewonnen werden und der Anwendung zu einer besseren Verarbeitung helfen. Dahingegen bezeichnet Lieberherrs Kontext nur zum kleinen Teil Informationen, die durch die Anwendung zur Laufzeit gewonnen werden. Viel mehr steht Kontext für das Variable, mit dem eine Software programmiert werden kann. Beispiele hierfür sind die Behandlung von Ausnahmen und Fehler, sodass verschiedene Fehlerbehandlungsschemata variabel einsetzbar sind, oder Verteilungseigenschaften werden als Kontext angesehen, sodass die Anwendung mit verschiedenen Objektmigrationsstrategien eingesetzt werden kann. Bei diesem Verständnis besteht ein großer Unterschied zu den anderen Vertreter adaptiver Software (vgl. Tabelle 3.4 auf Seite 40).

Lieberherr nicht falsch, sondern breit gefächert

In den Ausführungen zu dem Kontext-Begriff wurde bereits ausgedrückt, dass sich dem Kontext-Verständnis von Lieberherr nicht angeschlossen wird. Der Grund hierfür liegt nicht in einer Abneigung für sein Verständnis oder darin, dass die Definition für adaptive Software falsch oder widersprüchlich wäre.[22] Eher ist es der Umstand, dass das Kontext-Verständnis zu breit, zu ungenau für eine klare Begriffsbestimmung ist.

Das genau ist allerdings der Vorteil von Lieberherrs Auslegung. Er will damit aufzeigen, dass Kontext unterschiedlich verstanden werden kann, Kontext ist kontextabhängig. Und jede Interpretation für sich führt zu einem anderen Typ von »adaptiver Software«: Für den Programmierer oder den Algorithmus, der die Daten verarbeiten muss, kann Kontext in den Datenstrukturen gegeben sein. Ebenso im Rahmen der Softwareentwicklung können die Ausnahmebehandlungsschemata Kontext darstellen. Wird der Betrachtungswinkel weg von der Softwareentwicklung hin zur Laufzeit einer Anwendung genommen, kann Kontext und damit »adaptive Software« etwas ganz anderes umfassen. Ein Programm kann die »Betriebsumgebung« [Oreizy *et al.* 1998], die »Einsatzumgebung« [Czarnecki und Eisenecker 2000] oder den »Kontext« [Springer 2004] beobachten und eine Adaption vornehmen.

[22] Allerdings ist die von Lieberherr vorgenommene Interpretation und Auslegung von Kontext in Verbindung mit seiner Definition unklar: Wenn beispielsweise die für ein Programm notwendigen Ressourcen wie Drucker, Displays, Name Server nicht im Programm fest verdrahtet sind, sondern als Kontext angesehen werden, wann genau ist ein Programm adaptiv? Wird die Auslegung von Lieberherr genommen, dann ist das Programm adaptiv, weil die Ressourcen (vom Nutzer) *eingestellt werden können* (Passivität der Software). Nach Meinung des Autors zeigt dieses Beispiel aber eine *adaptierbare* oder »customizable« Software, die an die Rahmenbedingungen angepasst werden kann. Lieberherrs Ausführungen zu einer Interpretation für Kontext schaffen keine Klarheit, da sie adaptive Software (Software hat aktive Rolle) und adaptierbare Software (Software wird angepasst; passive Rolle) vermischen, obwohl die Ausführungen für adaptive Software formuliert sind.

Die vorgenannte Software *wäre adaptiv*, wenn sie die Ressourcen *selbst* konfigurieren würde (Software hätte aktive Rolle). Hierfür ein Beispiel aus dem Bereich Ubiquitous Computing, mit dem sich das MIT AI Lab im Rahmen des Projektes »Intelligent Room« beschäftigt [Gajos *et al.* 2003]: Die adaptive Software bzw. der »intelligente Raum« sollte fähig sein, eine abstrakte Dienstanfrage, z. B. »zeige mir diese Information«, einer Vielfalt an Lösungen zuordnen zu können (»projeziere die Information«, »zeige sie auf einem PDA«, »drucke sie auf einem Drucker aus«). Einhergehend damit muss die Software von den Möglichkeiten die beste Lösung finden, basierend darauf, wie gut die Lösung sowohl für den Nutzer ist als auch den Gebrauch von kostenintensiven oder beschränkten Ressourcen minimiert.

Die abstrakte Dienstanfrage ist optional zu sehen. Es reicht auch, wenn die Software verfügbare Drucker selbst konfiguriert oder für einen Druckauftrag den besten Drucker auswählt (z. B. bei Schwarz-Weiß-Druck den Laser benutzen, bei Bunt-Druck den Tintenstrahler, weil die schwarzen Farbpatronen teuer sind oder die Qualität des Lasers besser ist).

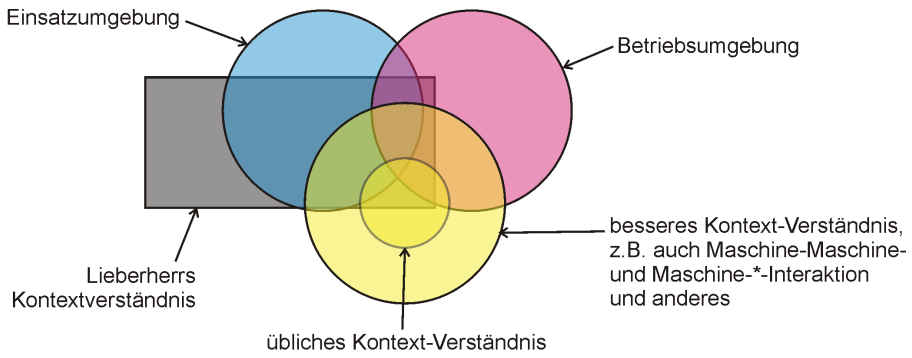

Abbildung 3.3.: Überschneidungen der einzelnen Begriffe Kontext, Betriebs- und Einsatzumgebung, die Auslöser für eine Adaption der Software sind. Für adaptive Software nach Verständnis der vorliegenden Arbeit wird die Vereinigung aller drei Kreismengen benötigt, d. h. ohne Lieberherr und auch nicht ein Kreis allein.

Schnittmengen und Differenzen

Bei einer Teilmenge dieser Umgebungen spielt dann auch der Kontext-Begriff aus der Forschungsrichtung Context-Awareness eine Rolle; Teilmenge nur deshalb, weil beispielsweise die Beschaffenheit einer Komponente (Schnittstelle), zu der Kontakt aufgenommen werden soll, nicht zum Kontext-Verständnis der gleichnamigen Forschungsrichtung passt. (Im vorgenannten Beispiel für ein adaptives System ist folgende Adaption denkbar: Das System erkennt die fehlende Interoperabilität (die Schnittstellen der zu kommunizierenden Teilsysteme sind inkompatibel zueinander) und generiert einen Adapter, der zwischen die zu kommunizierenden Komponenten platziert wird und damit für Interoperabilität sorgt; vgl. Seite 102.)

In Abbildung 3.3 ist der Zusammenhang zwischen den einzelnen Auslösern für eine Adaption einfach dargestellt. Das Bild zeigt, dass es Überschneidungsbereiche der einzelnen Begriffe gibt (wobei die Größenverhältnisse der Schnittmengen ohne Bedeutung sind), aber die Begriffe nicht ganz deckungsgleich zu bringen sind. Am deutlichsten abzugrenzen ist das Kontext-Verständnis von Lieberherr, wie die bisherigen Ausführungen bereits zeigten. Auch die Einsatzumgebung wird, sofern deutbar, von Czarnecki und Eisenecker vielfältig verstanden. Für sie und die zugehörige Definition für »adaptive Software« werden gar Beispiele von Czarnecki und Eisenecker genannt, die diesen Begriff als eine Mischung von Betriebsumgebung und den zwei Kontext-Verständnissen erscheinen lassen. Nur allen Begriffen gemein ist, dass sie ausschlaggebend für eine Adaption sind.

Der Begriff »∗-Umgebung« ist außerdem etwas verwirrend. Eine Deutung lässt vermuten, dass nur die Beschaffenheit der Umgebung ausschlaggebend ist (Temperatur, verfügbare Bandbreite etc.). Wird je-

doch die Kommunikation zwischen zwei Komponenten betrachtet, genauer, der Kontrakt zwischen den zwei Komponenten, dann muss der Begriff Umgebung relativiert werden. Im Rahmen des Forschungsprojektes COMQUAD wird beispielsweise die Unterstützung von zusagbaren nicht-funktionalen Eigenschaften untersucht (hier quantitive Eigenschaften wie Durchsatz und Verweilzeit oder auf höheren Abstraktionsebenen etwa auch Bildraten, Bildqualität, Transaktionsanzahl pro Sekunde) [Schill *et al.* 2000], [Aigner *et al.* 2003]. Auch die Informationen aus dem Kontrakt sind notwendige Informationen für eine Adaption (z. B. wenn der Kontrakt geändert wird). Sie der »Umgebung« zuzurechnen, ist nach Meinung des Autors etwas verwirrend.

An dieser Stelle soll die Analyse für diesen Abschnitt beendet und die gesammelten Erkenntnisse in einem Fazit formuliert werden.

3.6. Fazit

In diesem Unterkapitel werden Begriffe und Fakten verfasst, wie sie für die Thematik Adaptivität und Adaptierbarkeit von Bedeutung sind. Als erstes werden, das vorhergehende Unterkapitel abschließend, die notwendigen Informationen für adaptive Software charakterisiert und der Querbezug zur Forschungsrichtung Context-Awareness aufgezeigt. Danach werden Adaptierbarkeit, Adaptivität und Adaption definiert sowie hinsichtlich Software als Adaptionsgegenstand erörtert. Für eine Hilfestellung, das umfangreiche Thema Adaption zu erfassen, werden zum Schluss die vier W-Fragen formuliert und mit diesen eine Einschränkung der zu vertiefenden Thematik für die vorliegende Arbeit vorgenommen.

3.6.1. Kontext + X — Informationen für eine Adaption

In Abschnitt 3.5.1 auf Seite 38 wurde die Definition für Kontext von Dey [2000] sowie eine Übersetzung für den englischen Wortlaut bereits angegeben. Die Bedeutung von Kontext wurde jedoch noch nicht abschließend geklärt. Im ersten Teil dieses Abschnittes soll deshalb die weit verbreitete Definition für Kontext einer Überarbeitung unterzogen werden. Daran anschließend wird die Relevanz von Kontext als Informationsquelle für adaptive Software aufgezeigt.

Motivation für eine bessere Kontext-Definition

An dieser Stelle noch einmal die weit verbreitete Definition für Kontext von Dey:

> »Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an

> application, including the user and application themselves.« [Dey 2000]
>
> Kontext ist irgendeine Information, die für die Charakterisierung der Situation einer Entität genutzt werden kann. Dabei bezeichnet der Begriff Entität eine Person, einen Ort oder ein Objekt, das für die Interaktion zwischen einem Nutzer und einer Anwendung von Bedeutung ist, eingeschlossen den Nutzer und die Anwendung.

Zwei Dinge sind an dieser Definition zu bemängeln:

1. Die Definition kann falsch verstanden werden.
2. Auch bei richtiger Interpretation ist die Definition einschränkend.

Falsches Verstehen der Definition. Ein kleiner Test im Umfeld des Autors[23] ergab verschiedene Interpretationen für den letzten Nebensatz »... including the user and application themselves« in o. g. Definition. Ein Teil der Befragten meinte, das Dazuzählen von Nutzer und Anwendung beziehe sich auf die Aufzählung für eine Entität. Eine Entität sei eine Person, ein Ort, ein Objekt und auch der Nutzer und die Anwendung selbst. Der andere Teil der Befragten bezog das »including« auf das Wort »interaction«. Als Entitäten kämen damit neben einer Person und einem Ort auch alle Objekte in Frage, die relevant für die Interaktion zwischen Nutzer und Anwendung *sowie* zwischen verschiedenen Nutzern und verschiedenen Anwendungen relevant sind. Damit standen sich zwei verschiedene Interpretation gegenüber, wobei jede für sich überzeugend war.

Eine Definition ist keine Definition, wenn sie nicht klar verstanden werden kann. Selbst wenn der Anteil der richtigen Interpretationen der minimal größere ist, ist dies trotzdem Anzeichen für eine schlecht formulierte Definition. Drei der Befragten gaben die richtige Interpretation an, die zwei anderen die falsche (darunter der Kiwi).

Die Arbeit [Dey 2000], in der die Definition formuliert ist, sowie die an der TU Dresden angefertigten Arbeiten zu Kontext/Context-Awareness [Goslar 2001], [Löschau 2002], [Kadner 2003] und [Kadner 2004a] wurden dahingehend analysiert, welche Interaktionsformen für Context-Awareness sie berücksichtigen. Bei allen vorgenannten Arbeiten wurden lediglich nutzerorientierte Szenarien genannt, eine Maschine-Maschine-Kommunikation konnte selbst in [Dey 2000] nicht vorgefunden werden. Mit diesen Informationen konnte nur eine Deutung des Kontext-Begriffs vorgenommen werden, wie sie in Abschnitt 3.5.1 ab Seite 37 beschrieben ist. Erst die Nachfrage beim Verfasser brachte Klarheit.

In einer E-Mail von Anind K. Dey [Dey 2004a] an den Autor gibt der Verfasser der Definition eine klare Aussage, wie Kontext zu verstehen

[23] Das Umfeld bestand aus vier deutschen Studenten, die mindestens ein Jahr im englischsprachigen Ausland (USA, England, Australien) gelebt haben, und einem Kiwi (Neuseeländer).

ist: »Context-awareness considers interaction between the user and the application, the user and the environment, the application and other relevant applications, between users, and between the application and the environment. When I say environment, I refer to other devices (computers, etc.) and conditions (temperature, lighting, etc.)« [Dey 2004a]. Das »including« bezieht sich demnach auf das Wörtchen »interaction«.

Einschränkender Charakter Interaktion. Auch wenn die Definition von Dey für Kontext richtig verstanden wird, ist diese einschränkend. So wird durch die Fixierung auf eine Interaktion, hier zwischen Nutzer und Anwendung, einem Nutzer mit einem anderen Nutzer sowie einer Anwendung mit einer anderen Anwendung, der Fall außer Acht gelassen, wenn keine Interaktion stattfindet:

Es ist auch das Beispiel denkbar, dass ein Roboter (oder »Anwendung«, allg. Software) sich durch eine herunterrollende Steinlawine bedroht sieht und deshalb den Fortgang seiner Arbeit oder die ganze Mission anpassen muss. Die Steine entsprechen nicht der Umgebung, sondern können als einzelne Entitäten aufgefasst werden, die durch den Kinetik-Kontext (Geschwindigkeit, Richtung, dahinterstehende Masse) bestimmt werden. Bei diesem Beispiel findet keine Interaktion statt, sondern nur ein einseitiges (bestimmtes) Handeln. Der Roboter muss seine Mission an die neue Lage anpassen oder zumindest vor den daherrollenden Steinen ausweichen. Die Steine interagieren nicht mit dem Roboter, sondern rollen nur durch ihre Kinetik getrieben den Hang herunter.

Neue Definition für Kontext

Aus diesen Überlegungen heraus kann eine neue Definition für Kontext formuliert werden:

> Context is any information that can be used to characterize an entity or the relationship between several entities. An entity may be a real (physical, e. g. person, place) or abstract object (non-physical; e. g. application, components, presentation).
>
> Kontext ist irgendeine Information, die für die Charakterisierung einer Entität oder für die Beziehung von Entitäten zueinander genutzt werden kann. Eine Entität kann ein reales (physisch; z. B. Person, Ort) aber auch ein abstraktes Objekt (nicht-physisch; z. B. Anwendung, Komponenten, Präsentation) sein.

Diese Definition hat den Vorteil, dass sie keine Beschränkung bezüglich der Interaktion vornimmt, nicht fordert, dass die »Situation« einer Entität durch Kontext beschrieben wird,[24] und ebenso nicht einschränkt, wofür der Kontext genutzt wird.

[24] In E-Mail-Verkehr mit Kay Kadner flammte die Diskussion auf, welche Bedeutung dieser Situationsbegriff habe. Kadner meinte, dass Kontext nur *variable* Informatio-

Der neue Kontext-Begriff ist kompatibel zu allen notwendigen Informationen, die auszugsweise in Abschnitt 3.5 als Auslöser einer Adaption identifiziert wurden. Auch der für selbst-heilende Systeme relevante *Fehler* wird dadurch berücksichtigt: Ein Fehler ist eine Entität, der zu einer bestimmten Zeit an einer bestimmten Stelle (im System, Programmcode) auftritt.

Die neue Definition für Kontext ist nicht als Versuch zu sehen, die in Abschnitt 3.5 genannten Begriffe Kontext, Betriebsumgebung und Einsatzumgebung zu »verheiraten«. Stattdessen ist die neue Begriffsbestimmung der Erkenntnis entstanden, dass Kontext allumfassender ist als das Verständnis in den oben aufgeführten Arbeiten zeigt und es die formulierten Begriffe vermuten lassen.

Relevanz von Kontext für eine Adaption — Kontext + X

In den bisherigen Ausführungen wurde versucht, die Begriffe Betriebsumgebung, Einsatzumgebung und Kontext zu vereinheitlichen. Die zu klärende Frage war, welche Informationen für adaptive Software notwendig ist. Kann gesagt werden, dass für adaptive Software die Verfügbarkeit von Kontext essentiell ist, oder ist es mehr als Kontext (Kontext + X)?

Springer hat in seiner Arbeit postuliert, dass adaptive Anwendungen eine Adaption in Reaktion auf Änderungen des Kontextes vornehmen [Springer 2004, Kap. 1.4]. Diese Aussage stimmt für seine betrachtete Klasse von Anwendungen,[25] für adaptive Software im breiteren Rahmen ist die Beschränkung auf Kontext allerdings einschränkend (auch dafür, dass eine Reaktion stattfindet; vgl. Seite 72 »Adaption durch Probieren«).

Kontext wird bisher unzureichend verstanden. Dieser Begriff ist zudem schwer zu verstehen, da keine klare Abgrenzung dafür angegeben werden kann, was Kontext ist und was nicht. Mit der Motivation,

nen (z. B. den Ort einer Entität, die Umgebung etc., also die gegenwärtige Situation) beinhalte. Der Autor erwiderte auf diese Meinung, dass auch die Beschaffenheit der Schnittstelle einer Komponente eine Kontextinformation darstelle, wobei diese auf lange Sicht *konstant* bleibt. Variabilität entsteht hier erst, wenn ein Dienstnehmer einen Dienstgeber mit einer anderen Schnittstelle (andere Interoperabilität, vgl. Seite 102) kontaktieren muss. In dem Fall ändert sich die Entität und nicht der Kontext der Entität. Auch wenn die Beschaffenheit einer Schnittstelle anscheinend nicht als Kontext angesehen wird (siehe die Ausführungen auf dieser Seite), besteht dennoch ein Widerspruch für die Forderung der Variabilität als Situationsbegriff. So sind für adaptive ubiquitäre Anwendungen die Informationen zur Display- und Speichergröße des Endgerätes notwendig. In [Hitz *et al.* 2002] und [Springer 2004] werden diese verständlicherweise als Kontext der Ausführungsumgebung betrachtet. Damit besteht jedoch ein Widerspruch zur vermeintlich geforderten Variabilität, der spezifischen Situation einer Entität, denn Display- und Speichergröße des Endgerätes bleiben konstant.

[25] Klassische Internetanwendungen wie E-Mail, Instant Messaging, FTP und WWW, die die Heterogenitäten und beschränkten Eigenschaften der Endgeräte, verschiedene Zuverlässigkeit und Qualität der Netzwerkverbindungen sowie Mobilität und heterogene Benutzeranforderungen und -kontexte berücksichtigen müssen.

den Überschneidungsbereich von Adaption und Context-Awareness genauer zu untersuchen, wurde vom Autor erörtert, ob in Anwendungen aufgetretene Fehler (Selbst-Heilung) und eine unterschiedliche Schnittstelle (an die eine Anpassung erfolgt; vgl. Seite 102) als Kontext deutbar sind. Kadner, der sich ein Jahr mit Context-Awareness beschäftigt hat, meinte, dass ein Fehler keinen Kontext darstellt [Kadner 2004b]. Für die Schnittstelle wurde die Frage bejaht, jedoch unter der Voraussetzung, dass sich die Schnittstelle der (gleichen) Komponente auch ändert [Kadner 2004c]. Die gleiche Frage an Dey gestellt, einem im Forschungsbereich Context-Awareness bekannten Wissenschaftler, zeigte in seinen Antworten ein etwas anderes Bild. Die Schnittstelle einer Komponente sei kein Kontext, da eine Schnittstelle keine Informationen enthält, sondern nur Informationen repräsentiert. Fehler sind allerdings mit Gewissheit Kontext [Dey 2004b]. Somit stehen zwei unterschiedliche Meinungen gegenüber. Dey konnte kein Rezept geben, wie Kontext von Nicht-Kontext unterschieden werden kann, meinte jedoch, dass nicht alle Informationen relevant sind, weil andernfalls alles Kontext sei [Dey 2004c].

Informationen = alles, was beobachtbar ist. Es kann, ohne noch weitere notwendige Informationen, die für adaptive Software benötigt werden, hinsichtlich Kontext zu deuten, formuliert werden, dass für adaptive Software mehr als Kontext notwendig ist. Adaptive Software aggregiert die notwendigen Informationen nicht nur mithilfe eines Kontextdienstes, sondern gewinnt die Informationen aus allem, was für sie beobachtbar ist.

Diese Formulierung ist sehr flexibel. Sie umfasst sowohl Betriebs- und Einsatzumgebung als auch Kontext, da ebenso für die vorgenannten Begriffe eine adaptive Software ihre Umgebung / den Kontext selbst in Erfahrung bringen muss. Der Begriff der Anforderungen, der in einigen Definitionen genannt wird, erhält hier auch elegante Berücksichtigung: Eine Software kann sich an andere Anforderungen anpassen, sofern sie diese erkennen kann. Normalerweise kann eine Software keine anderen Anforderungen erkennen (vgl. die Ausführungen auf Seite 66 zu dem diskutablen Überschneidungspunkt). Der Autor ist jedoch der Meinung, dass in der Zukunft die in Software realisierte Intelligenz mit der des Menschen insofern gleichziehen wird, dass Software auch geänderte Anforderungen erkennen sowie diese umsetzen und eine Adaption an geänderte Anforderungen vornehmen kann. Dieser Gedanke wird durch die Formulierung der Beobacht*barkeit* einkalkuliert.

Ergebnis: Eine notwendige Voraussetzung für die Adaption ist es, dass Informationen über das Adaptionsziel verfügbar sind bzw. ermittelt werden können, um eine Adaption durchführen zu können. Die Gewinnung, Verarbeitung und Speicherung von Informationen über

die Ausführungsumgebung sind wesentliche Ziele des Forschungsbereiches Context-Awareness. Die betrachteten Informationen werden unter dem Begriff Kontext zusammengefasst. Adaptive Software benötigt darüber hinaus weitere Informationen, die nicht durch eine Kontextgewinnung abgedeckt werden. Eine adaptive Software erhält und gewinnt ihre Informationen aus *allem Beobachtbaren*.

Dennoch können große Parallelen zwischen »adaptiven« und »kontextabhängigen/kontextsensitiven« Anwendungen gezogen werden. Beide Anwendungsarten verwenden die Informationen aus der Umgebung, in der sie laufen, und bieten eine darauf angepasste Verarbeitung. Dey [2000] konnte sogar noch andere Bezeichnungen in der Literatur finden, die synonym für »adaptiv« und »kontextsensitiv« Verwendung finden: »reactive«, »responsive«, »situated« und »environment-directed« [Dey 2000, S. 5]. Ben-Shaul *et al.* [2001a] bringen für Systeme, die sich anpassen, außerdem den Begriff Flexibilität ins Spiel, um auszudrücken, dass eine Software eben nur flexibel statt anpassungsfähig/(selbst-) adaptiv ist, wenn die Regeln für eine Adaption sowie die verfügbaren Kode-Alternativen, zwischen denen die Komponente wählen kann, statisch sind.

In Abbildung 3.4 ist der in diesem Abschnitt diskutierte Zusammenhang dargestellt. Für adaptive Software sind die Informationen relevant, die sich innerhalb des dick umrandeten Kreises befinden. Sie sind alles, was die Software beobachten kann (einschließlich Kontext). Derzeit nicht von Software feststellbar sind geänderte funktionale Anforderungen (außerhalb des dick umrandeten Kreises). Geänderte funktionale Anforderungen sind relevant für die Adaptierbarkeit von Software, da Software einer ständigen Evolution unterliegt (siehe nächster Abschnitt).

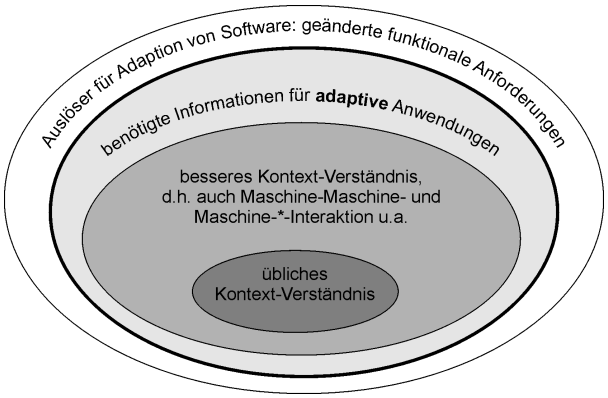

Abbildung 3.4.: Für adaptive Anwendungen werden nicht nur Kontextinformationen benötigt, sondern auch alles andere Beobachtbare (z. B. die Beschaffenheit von Schnittstellen, eigene Performance, Fehler)

3.6.2. Adaptierbarkeit und adaptierbare Software

Adaptierbarkeit bezeichnet, gemäß den Definitionen 1, 3 und 4 von Seite 27 die Einfachheit, mit der ein System an sich geänderte Anforderungen/$*$-Umgebung angepasst werden kann. Gegenstand der Anpassung ist die Software, d. h. die Software wird an die anderen Rahmenbedingungen/Anforderungen angepasst.

Tekinerdogan und Aksit [1996] nennen zwei Gründe, die für die Änderung von Anforderungen verantwortlich sind: Zum einen sind komplexe Systeme schwer zu verstehen, weshalb deren initiale Anforderungen oft fehlerbehaftet sind. Zum anderen unterliegt komplexe Software einer Evolution, d. h. nach deren ursprünglichen Spezifikation entwickelt sie sich weiter.

Deshalb formulieren Buschmann *et al.* [1998], dass es eine wesentliche Aufgabe beim Entwurf der Architektur eines Software-Systems ist, diesen für Veränderung zu entwickeln. Eine Software sollte ihre eigene Modifikation und Erweiterung von vornherein unterstützen. Veränderungen sollten die Kernfunktionalität und die wesentlichen Entwurfsabstraktionen nicht beeinflussen, sonst wird das System schwierig zu warten und teuer in der Anpassung.

Viele Softwareentwickler neigen daher dazu, ihre Software derart zu entwickeln, dass sie hochflexibel ist, indem sie beispielsweise frei konfigurierbare Schnittstellen entwerfen. Wie jedoch auch Dittert [2004] betont, ist eine derartige Vorgehensweise nicht anzuraten. Frei konfigurierbare Schnittstellen erlauben zwar die Änderbarkeit/Anpassbarkeit der Software, diese wird jedoch mit einer niedrigeren Wartungsfreundlichkeit und höheren Komplexität des Systems erkauft, da Methodendeklarationen und zulässige Parameter sorgfältig dokumentiert werden müssen. Außerdem gibt es keine Schnittstellenbeschreibung, wie beispielsweise die IDL[26], die auf Verträglichkeit bei der Integration von Komponenten geprüft werden kann.

Eine adaptierbare Software-(Architektur) ist eine Software-(Architektur), die leicht angepasst werden kann. Software, die sich mit dieser nicht-funktionalen Eigenschaft schmückt (für andere nicht-funktionale Eigenschaften siehe auch Fußnote 17 auf Seite 32), muss Adaptierbarkeit unterstützen. Adaptierbarkeit kann auf vielfältige Weise erbracht werden. Sie kann durch die sie berücksichtigende Architektur des Systems erbracht werden (indem mögliche Änderungen berücksichtigt werden oder die Architektur offen für Veränderung entwickelt wird, zum Beispiel mit den Architekturmustern Microkernel und Reflection [Buschmann *et al.* 1998]), aber auch durch entsprechende Softwareentwicklungsmethoden, wie beispielsweise der modellgetriebenen Softwareentwicklung, die mit der aktuell diskutierten MDA[27] wieder eine Renaissance erlebt — sie kann gar als

[26] Interface Definition Language
[27] Model Driven Architecture

Hype bezeichnet werden.[28] Die modellgetriebene Softwareentwicklung erlaubt es — ähnlich dem Single-Source-Publishing mit einer endgeräteunabhängigen (Seitenbeschreibungs-) Sprache [Göbel *et al.* 2001] (z. B. DDL [Hübsch *et al.* 2003], SMIL [Ayars *et al.* 2001] und UIML[29] [UIML]) — die Software in einem Modell zu beschreiben und an verschiedene Plattformen mittels einer Transformation (CIM[30] zu PIM[31] zu PSM[32] zu EDM[33]; vgl. [Miller und Mukerji 2003], [Andresen 2003]) anzupassen.

Software, die zur Laufzeit angepasst werden kann, ohne sie herunterfahren zu müssen, wird als *dynamisch* adaptierbar bezeichnet [Wermelinger 1999].

Andere Interpretationen. Der Begriff Adaptierbarkeit wurde auf Seite 32 extra als »Evolutions-Adaptierbarkeit« bezeichnet, weil er die Adaption im Rahmen der Evolution einer Software ausdrückt. Auf Seite 87 kann und wird er jedoch derart interpretiert, dass untersucht wird, was adaptierbar ist (»Möglichkeit-Adaptierbarkeit«).

In der Literatur sind auch Verwendungen für den Begriff »adaptability« zu finden, die eher auf Adaptivität/»adaptivity« abzielen. Wie der Abschnitt 3.1 gezeigt hat, liegt dies daran, dass die Wörterbücher für eine einheitliche Übersetzung nicht behilflich sind. Beispielsweise wird in [Seth und Kokar 2003] der Begriff Adaptierbarkeit/»adaptability« als Maßzahl dafür benutzt, in welchem Maße, verglichen mit nichtadaptiven Systemen, ein System Adaptionen vornimmt. Das heißt, die Maßzahl gibt an, inwiefern ein System auf die andere Systemumgebung reagiert.[34] Nach Meinung des Autors müsste die Maßzahl in die-

[28] Michael Stal fühlt sich bei Betrachtung der MDA an »gute alte Zeiten« erinnert [Stal 2003]. Wie alle genialen Konzepte lassen sich ähnliche Ideen in der Vergangenheit finden, wenn »der interessierte Informatiker nur intensiv genug in den vergilbten Annalen der Software-Entwicklung« blättert. Stal erinnert an Programmgeneratoren und 4GL-Konzepte der achtziger Jahre und Wizards in modernen Programmierumgebungen. Oder zum Beispiel das Unternehmen Taligent, das Anfang der Neunziger versucht hat, auf Basis generischer Frameworks ein neues Betriebssystem zu entwickeln. Allen diesen Ansätzen ist eines gemeinsam: Entweder sind sie gescheitert oder konnten lediglich in engen Nischen Einzug halten, da generative und generische Ansätze nur in begrenzten und gut verstandenen Domänen funktionieren. Mit größerem Fokus der Domäne ergibt sich eine größere Komplexität der Modellierung. Knackpunkt hier: Komplexität lässt sich niemals beseitigen, sondern nur verbergen. Für ein genügend breites Anwendungsgebiet ist die Modellierung und automatische Generierung mittels MDA faktisch genauso aufwendig wie eine manuelle Implementierung (vgl. [Stal 2003]).

[29] User Interface Markup Language

[30] Computation Independent Model

[31] Platform Independent Model

[32] Platform Specific Model

[33] Enterprise Deployment Model

[34] In vorgenannter Quelle ist die Anwendung eine Rechtschreibprüfung, die die Korrekturvorschläge nicht in simpler alphabetischer Reihenfolge dem Nutzer präsentiert, sondern die Liste von Alternativen an den jeweiligen Nutzer anpasst. Die adaptive Rechtschreibung kommt in dem Maße zum Tragen, dass sie Fehler(arten) (bei-

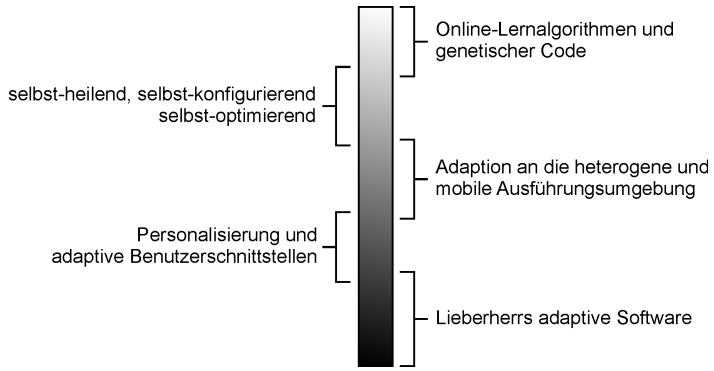

Abbildung 3.5.: Spektrum von adaptiver Software

sem Fall Adaptivität/»adaptiveness« lauten, da gemessen/ausgedrückt wird, in welchem Maß sich ein System adaptiv verhält, d. h. (sich) anpasst (Aktiv). Es wird nicht gemessen, in welchem Umfang ein System angepasst werden kann.

3.6.3. Adaptivität und adaptive Software

Anders als Adaptierbarkeit, bei der Software angepasst wird (passive Rolle der Software), bezeichnet Adaptivität von Software Änderungen, die *die Software selbst* vornimmt (aktive Rolle). Eine Software ist adaptiv.

Allgemein kann Adaptivität in Software überall und jederzeit in einer Phase, die Software durchlebt, vorkommen — je nachdem, wo und warum eine Anpassung an sich ändernde Umstände erwünscht ist. Wie die vorangegangenen Ausführungen zeigten, ist für adaptive Software ein sehr großes Spektrum möglich.

Breites Spektrum für adaptive Software

In Abbildung 3.5 wird versucht, das Spektrum für adaptive Software zu verdeutlichen.[35] Am unteren Ende der Skala für adaptive Software steht sicherlich Software nach dem Verständnis von Lieberherr [1998]. Hier nimmt die Software nicht unbedingt Anpassungen vor, sondern die Software (der Quellcode) wird z. B. mithilfe der Demeter-Methode

spielsweise Buchstabendreher, doppelter Anschlag für einen Buchstaben (Buchstabendoppler) und umgekehrt, benachbarte Tasten auf Tastatur erwischt etc.) des Nutzers beobachtet und basierend darauf, wie oft ein bestimmter Fehler vom Nutzer gemacht wird, Korrekturvorschläge in einer *an den Nutzer angepassten Reihenfolge* anzeigt. Dadurch hat der Nutzer die wahrscheinlich korrekte Version des von ihm falsch eingetippten Wortes immer weit oben stehen.

[35] Eigentlich kann eine solche Abbildung nicht nur für eine Skale erfolgen. Stattdessen ergibt sich Adaptivität in einem mehrdimensionalen Raum (vgl. die vier W's auf Seite 47). Deshalb kann die Zuordnung der Beispiele nicht als sortiert und abgegrenzt verstanden werden.

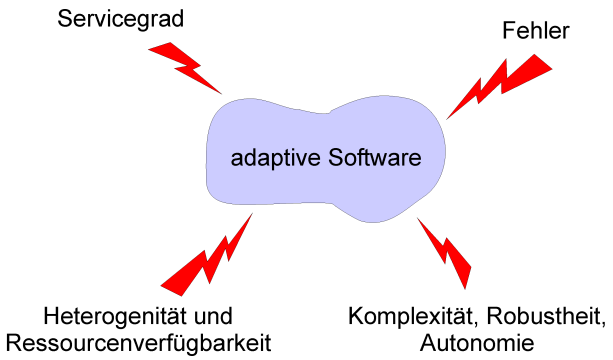

Abbildung 3.6.: (selbst-) adaptive Software muss (sich) wegen vielerlei Faktoren anpassen

so entwickelt, dass sie generisch genug ist, um für verschiedene »Kontexte« einsetzbar zu sein (vgl. [Lieberherr 1995]). Andere Beispiele wurden ebenso von Lieberherr genannt (siehe Seite 36). Das andere Ende bildet adaptive Software, die sich während der Ausführungszeit selbst anpasst, indem sie die Umgebung beobachtet und alternative(s) »Verhalten« oder Implementierungen mit einbezieht, gegebenfalls sogar selbst neuen Code erzeugen kann [Langdon und Poli 2002]. Dazwischen kann adaptive Software eingeordnet werden, bei der nicht die Software, sondern Daten, Kommunikation, die Interaktion mit dem Nutzer (Personalisierung und innovative Benutzerschnittstellen) oder anderes angepasst werden (in den bisherigen sowie in den folgenden Ausführungen wurden/werden noch viele Beispiele deutlich).

Flexibilität und Lernfähigkeit. Höhere Adaptivität ergibt sich zudem durch eine Lernfähigkeit der Software, d. h. die Software lernt aus eigener Erfahrung, die sie durch Adaption erreicht hat, zukünftige Adaptionsentscheidungen zu treffen. Daher kann auch argumentiert werden, dass die andere adaptive Software nicht wirklich adaptiv, sondern eher flexibel ist.[36]

Motivation für (selbst-) adaptive Software

Eine Anpassung/Adaption erfolgt, wie bereits an mehreren Stellen deutlich geworden ist, aus vielfältigen Gründen. Für ein besseres Verständnis werden im Folgenden einige Beispiele gegeben (siehe auch Abbildung 3.6):

Software darf nicht starr sein, sondern muss Anpassungen an sich vornehmen können aufgrund von

[36] Womit am Ende dieses Kapitels die Bestätigung kommt, dass die Übersetzung *lernfähig* für das englische »adaptive« ihre Berechtigung hat (vgl. Seite 19) — nichtsdestotrotz hat die Software andere, erweiterte Eigenschaften.

- Heterogenität (Verschiedenartigkeit) der Ausführungsumgebung (Netzwerk, Geräte, Benutzer[37], Software selbst (z. B. Schnittstellen; siehe Beispiel 3 auf Seite 24 und Erörterung zu Schnittstellenanpassung auf Seite 102)),
- unterschiedlicher Ressourcenverfügbarkeit in der Ausführungsumgebung,
- Wünschen für einen besseren Servicegrad (sei es, Beispiel Amazon, um zum Absatz von mehr Waren anzuregen (Vorteil Serviceanbieter), oder um dem Servicenehmer eine bessere Performance zu bieten)
- auftretenden Fehlern,
- Erfordernis nach Autonomie und nach erhöhter Robustheit der Software,
- erhöhter Komplexität der Anwendungen, die für den Menschen nur noch schwer beherrschbar ist.

Triviale Kandidaten. Am intuitivsten zu verstehen ist sicher der erste Punkt, da der Begriff Anpassung am leichtesten mit der Überwindung von *Heterogenitäten* (Verschiedenartigkeiten) in Verbindung zu setzen ist. Denn ein Adapter schafft einen Ausgleich zwischen nicht zusammenpassenden Systemen. Nicht nur die heterogenen Eigenschaften von miteinander kommunizierenden Systemen sind Auschlag für eine Adaption, auch die einem System zur Verfügung stehenden *Ressourcen* können maßgebend für eine Anpassung sein.

Wenn die Betriebsmittel wie vorhandene Kommunikationstechnologie, Bandbreite und Verarbeitungskapazitäten schwanken — was vor allem in dynamischen und mobilen Systemen vorkommt —, dann muss sich das System an die gegebene Verfügbarkeit entsprechend anpassen. Eine gleichbleibende Dienstgüte durch eine Ressourcenreservierung ist meist nicht mehr möglich. Stattdessen erfolgt eine Skalierung (Adaption) des zu erbringenden Dienstes an die verfügbaren Ressourcen, wodurch nur eine variable Dienstgüte erbracht werden kann (vgl. [Springer 2004, Abschnitt 1.3]).

Mit der Wahrung einer vorgegebenen oder minimalen Dienstgüte in gewisser Weise vergleichbar ist der dritte Punkt. Hier jedoch ist es Ziel, den *Servicegrad* einer Anwendung zu erhöhen. Der durch die Anwendung erbrachte Mehrnutzen kann gesteigert werden, wenn diese adaptiv ist. Einfaches Beispiel hierfür sind personalisierte Benutzeroberflächen oder die dazugehörigen Daten (wobei eine Personalisierbarkeit auch für die Bewältigung von heterogenen Nutzeranforderungen verwendet wird). Der bessere Servicegrad muss nicht unbedingt nur

[37] Sowohl Benutzeranforderungen können heterogen sein als auch Benutzerfähigkeiten (Beispiel Farbsehfähigkeit auf Seite 88); Personalisierung.

zum Vorteil des Service-/Dienstnehmers sein, sondern kann auch dem Diensterbringer Vorteile erschaffen (z. B. Produktmarketing bei Amazon; vgl. Seite 25).

Auch Fehler, z. B. die Unerreichbarkeit von Servern und Netzwerken oder die Fehler in Komponenten, können Grund für eine Anpassung des Systems sein. Das System muss alternative Wege suchen, das gewünschte Ziel zu erreichen. Dazu muss der ursprünglich gewählte Weg an die neue Situation angepasst werden. Diese Fähigkeit erhöht auch die Robustheit von Anwendungen gegenüber unerwarteten Situationen, wodurch sich — wie sicher auch bei den anderen Punkten untereinander — Querbezüge zu dem nächsten Punkt herstellen lässt.

Besonderheit Autonomie, Robustheit und Beherrschung Komplexität. Die letzten zwei Punkte für eine Motivation sind bestimmt weniger intuitiv mit adaptiven Software-Systeme in Verbindung zu bringen. Eine gute Erklärung für die Relevanz liefert [Laddaga 1999, 2001]. Laddaga sieht drei große Probleme, denen sich die Software-Industrie in den nächsten zwei Dekaden stellen muss: die steigende Komplexität von Anwendungen, die Erfordernis nach Autonomie und Robustheit der Anwendungen. Die Gründe für diese Probleme sind vielfältig. Auf eine Erklärung, warum sie existieren, muss an dieser Stelle verzichtet werden. Der Leser wird auf die vorgenannte Literatur verwiesen. Wichtig für ein Verstehen ist nur ihre Relevanz für (selbst-) adaptive Software.

Laddaga sieht selbst-adaptive Software als Lösung für diese Probleme.[38] Die Software muss aktive Verantwortung für ihre eigene Robustheit und das Management ihrer eigenen Komplexität übernehmen. Für die Erreichung dieser Ziele ist es notwendig, dass die Software mit Repräsentationen ihrer Ziele, Methoden, Alternativen und der Umgebung ausgestattet ist. Hierfür hat die DARPA bereits 1997 in einem »Broad Agency Announcement« [Laddaga 1997] für selbst-adaptive Software (SAS) Forschung ausgerufen.[39] Die Definition der DARPA für selbst-adaptive Software (siehe Seite 29) gibt bereits Auskunft, wie derartige Software beschaffen ist. Die Schlüsselaspekte der Definition sind, dass das Code-Verhalten zur Laufzeit evaluiert und getestet wird, wobei ein negatives Testresultat zu einer Änderung im Verhalten führt. Damit kann die Software derart charakterisiert werden, dass sie folgende Dinge enthält, die normale Software nicht aufweist:

- Beschreibung der Absichten der Software (Ziele und Entwurf) und der Programmstrukturen sowie
- eine Ansammlung von alternativen Implementationen und Algorithmen (manchmal genannt »reuse asset base«).

[38] Daneben werden noch andere Technologien genannt: »tolerant software, physically grounded software and negotiated coordination.«

[39] Etwas bekannter sind die Initiativen Autonomic Computing und Organic Computing, die sich auch diesen Problemen annehmen wollen; vgl. Abschnitt 2.4 und 2.5.

Selbst-adaptive Software evaluiert ihr eigenes Verhalten und die Umgebung entsprechend der Zielerfüllung und korrigiert das Verhalten in Reaktion auf die Auswertung. Das heißt, die Software versteht, was sie macht, wie sie es macht, wie sie ihre eigene Performance evaluiert und wie sie auf die geänderten Bedingungen reagiert.

adaptiv vs. selbst-adaptiv

In der vorliegenden Arbeit wird keine Unterscheidung zwischen »adaptiver« und »selbst-adaptiver« Software getroffen. Dies ist damit zu begründen, dass viele der Recherchen Quellen entstammen, die beide Begriffe verwenden (sie werden bei der Übernahme so belassen) und auch — das ist der springende Punkt — dasselbe Verständnis für adaptive Software besitzen, wie es in den nachfolgenden Ausführungen besteht. Im weiteren Verlauf der Arbeit finden daher beide Begriffe im Text Verwendung.

Verwunderlich ist auch, warum das »adaptive« in der Literatur den Präfix self/selbst bekommen hat. Steht es dafür, dass die Software etwas selbst macht (wie, adverbiale Modalbestimmung), oder dafür, dass die Software sich selbst als Anpassungsgegenstand hat (was, das Objekt)? Zumal beispielsweise in [Karsai und Sztipanovits 1999] und [Czarnecki und Eisenecker 2000] beide Begriffe parallel verwendet werden, ohne dass ersichtlich wird, ob und welcher Unterschied besteht.

Definition

Nachdem viele Beispiele und Definitionen für »adaptive/adaptierbare Software« aufgeführt und diskutiert wurden, stellt sich die Frage, wie adaptive Software in der vorliegenden Arbeit definiert wird.

Es ist schwierig, eine klare Definition für adaptive Software zu geben. Wie die innerhalb dieses Kapitels erfolgten Analysen zeigen, können Definitionen zum einen schwer oder falsch verstanden werden, weil Begriffe anders oder kaum interpretierbar sind (z. B. der Kontext-Begriff, was ist Verhalten?, Umgebung), zum anderen läuft der Verfasser einer Definition Gefahr, diese in ungenügender Weise aufzustellen. Wenn außerdem verschiedene oder alle Adaptionsgegenstände berücksichtigt werden müssen und kein Anlass für eine Adaption lieferbar ist, dann wird die Formulierung außerdem unklar, weil sie entweder zu umfangreich oder zu unreichend ist.

Das Verständnis für adaptive Software in der vorliegenden Arbeit ist deshalb eine Mischung aller Adaptionen mit Ausnahme der Definitionen 2 und 3 von Lieberherr. Der wichtigste Punkt ist folgender: Adaptive Software ist dadurch gekennzeichnet, dass sie selbst die Adaptionen vornimmt, wohingegen bei adaptierbarer Software die Software adaptiert wird. Bei adaptierbarer Software hat die Software eine passive Rolle und zugleich ist der Adaptionsgegenstand (das Adaptionsobjekt)

auf die Software beschränkt — bei adaptiver Software kann »alles« adaptiert werden.

Adaptive Software, wie sie in der vorliegenden Arbeit (und auch mit allen Beispielen und Definitionen, außer denen von Lieberherr, konform ist) verstanden wird, kann folgendermaßen definiert werden:

> *Eine adaptive Software nimmt Adaptionen an sich vor.* Oder:
> *Eine adaptive Software passt sich an.*

Diese Definition ist eben sehr einfach. Damit ist sie vergleichbar mit der Brockhaus-Definition (»auf Adaption beruhend«, vgl. Seite 22). Im Vergleich zu dieser nennt sie jedoch ein Adaptionssubjekt, womit klar ist, ob das Adaptionssubjekt (hier Software) eine aktive oder eine passive Rolle übernimmt.

Es fällt auf, dass die Definition nicht heißt: »Eine adaptive Software nimmt Adaptionen vor.« Eine Interpretation dieser Formulierung würde bedeuten, dass eine Software auch adaptiv wäre, wenn sie eine Adaption an einem Gedichtsband vornimmt. Um dem zuvorzukommen, wurde die Definition um das Adaptionsobjekt »sich« erweitert. Damit wird allerdings das Adaptionsobjekt Software festgelegt.

3.6.4. Adaptionsobjekt/-subjekt Software

Die soeben formulierte Definition für adaptive Software ist bezüglich des Adaptionsobjekts noch unklar. Was genau ist Software? Was bedeutet es, wenn eine Software *sich* anpasst?

Je nach Stakeholder wird der Software-Begriff sicherlich unterschiedlich aufgefasst. Der Programmierer sieht nur seinen geschriebenen Quellcode, der Anwender alles, was auf dem Bildschirm zu sehen ist und langsam reagiert oder abstürzt, also einschließlich Verhalten der Software. Eine formalere Begriffsumfassung bietet der Arbeitskreis 5.10.3 der GI[40]: Turowski [2002] nennt im Zusammenhang mit vermarktbaren Fachkomponenten als *Softwareartefakte* unter anderem den ausführbaren Code, darin verankerte Grafiken, Textkonstanten usw., Daten wie zum Beispiel Voreinstellungen oder Parameter, die einen initialen Zustand der Komponente beschreiben, sowie Spezifikation, Dokumentation und (automatisierte) Tests [Turowski 2002].

Aufteilung der Software-Artefakte. Die für einen funktionierenden Komponentenmarkt sonst unerlässlichen Artefakte Spezifikation, Dokumentation und (automatisierte) Tests scheiden als Adaptionsgegenstand aus.

Textkonstanten fallen für eine Adaption auf dem ersten Blick auch weg, da sie eben konstant sind. Allerdings fallen sie nicht wirklich

[40] Gesellschaft für Informatik

weg, denn bei einer Anwendung, die sich im Rahmen einer Lokalisierung[41] an den Nutzer anpasst, finden die Textkonstanten bei der Adaption Verwendung. Denn wenn die Benutzeroberfläche in einer für den Nutzer verständlichen Sprache gehalten sein soll, müssen aus vielen Alternativen die richtigen Textkonstanten (Textbausteine des Resource Bundles) ausgewählt werden, die Textkonstanten liegen redundant vor. Selbst ihren konstanten Zustand können sie verlieren, wenn lange Texte im Rahmen der Fragmentierung für kleine Seiten aufgeteilt werden müssen.

Die Voreinstellungen und Parameter einer Komponente können auch einer Adaption unterliegen. So ist es vor allem für selbst-managende Systeme ein Ziel, dass die Systeme sich selbst optimieren, indem sie an den Parametern drehen und versuchen, die zu finden, die das System im gewollten Sinne beinflussen (vgl. [Rademacher 2004]).

In der Software verankerte Grafiken können auch einer Adaption unterliegen, beispielsweise einer Verkleinerung oder Umwandlung in ein anderes Bild-, Farb- oder Dateiformat (vgl. Transformation in Abschnitt 4.2.3). Diesen Punkt berücksichtigend schließt sich auch die Frage an, ob nur in der Software verankerte Grafiken oder auch vom Anwender bereitgestellte Daten (Anwendungsdaten) als Adaptionsgegenstand zählen. Zur eigentlichen Software zählen die Anwendungsdaten sicher nicht, jedoch wäre es nach Meinung des Autors unsinnig, eine Anwendung nur adaptiv zu nennen, wenn sie ihre eigenen, verankerten Daten adaptiert, nicht aber die vom Nutzer geschaffenen und verwendeten Anwendungsdaten. Beispiel: Adaption (Farbtransformation, Verkleinerung, Ersetzung) von Icons des GUI[42]s oder von Logos in Dialogen versus Adaption von Bildern in E-Mails. Im weiteren Verlauf werden die verankerten Grafiken allgemein als (Anwendungs-) Daten bezeichnet, um sie von der eigentlichen Software, die die Funktionalität bereitstellt, abgrenzen zu können.

Damit ergibt sich eine Aufteilung des Software-Begriffs von Turowski [2002]. Dieser Zusammenhang (und der nachfolgend formulierte) ist in Abbildung 3.7 auf der nächsten Seite bildlich dargestellt. Die graue Ellipse ist in zwei Bereiche eingeteilt: Der linke Bereich fällt für eine Adaption heraus, der rechte Bereich kann als direkter oder indirekter Gegenstand (z. B. in Form von Redundanz oder als Alternative) für eine Adaption dienen (Adaptionsobjekt).

Erweiterung der Software-Ellipse für adaptive Software? Im vorangegangenen Text, bei dem der Software-Begriff von Turowski [2002] unter die Lupe genommen wurde, sind bereits einige Zweifel und Schlussfolgerungen getroffen worden: Wenn verankerte Grafiken als Adaptions-

[41] Mit Lokalisierung ist hier nicht eine Positionsbestimmung (Ortskontext) gemeint, sondern die Lokalisierung (L10N) einer Anwendung nach der erfolgten Internationalisierung (I18N). Die Begriffe in den Klammern geben die Fachkürzel an.

[42] Graphical User Interface

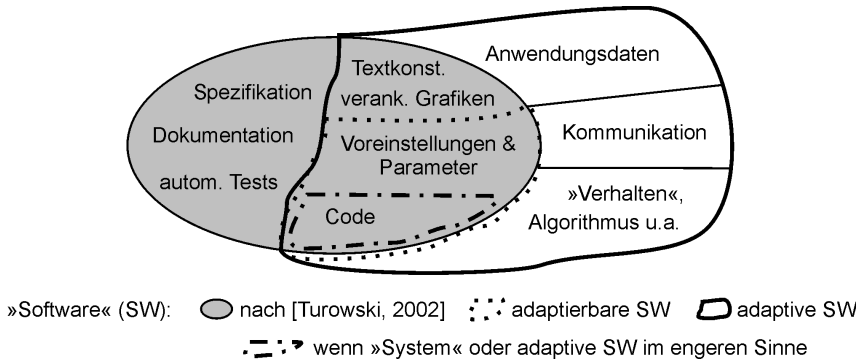

Abbildung 3.7.: Unterschiedlicher Begriff »Software« (SW); hier für Vermarktung, als Gegenstand adaptierbarer SW und für adaptive SW

objekt dienen können, warum dann nicht auch »richtige Anwendungsdaten«, also Daten, die erst durch den Nutzer durch Anwendung der Software entstehen (z. B. Bild in einer Webseite oder im Anhang einer E-Mail)? Wenn sich dieser kausale Zusammenhang in einer Erweiterung des Software-Begriffs für adaptive Software für Anwendungsdaten ergibt, dann muss er auch für die Kommunikation sowie für alles andere, das eine Software/Anwendung ausmacht, getroffen werden.

Dieser Betrachtung folgend müsste der Adaptionsgegenstand »Software« sehr weit gefasst werden. Er wird dann nicht mehr durch die rechte Hälfte der Ellipse bestimmt, sondern ist alles, was durch die dicke, durchgezogene Linie in Abbildung 3.7 begrenzt wird. Damit ist neben der vorgenannten Adaption der Daten auch die der Kommunikation und des — nicht klar bestimmbaren — Verhaltens und anderem (z. B. Routingalgorithmus oder die Änderung der Transaktions- und Sicherheitsregeln in einem verteilten System) möglich.

Adaptive Software im engeren Sinne. Allerdings kann auch argumentiert werden, das zu »richtiger adaptiver Software« keinesfalls die Adaption der Kommunikation und der Daten gehört. Eher ist eine »adaptive Software« nur eine Software, die reine Funktion, die durch die Komponenten des Systems (den ausführbaren Code) vorgegeben ist, ändert (dargestellt als gestrichpunktete Linie in Abbildung 3.7). Denn wenn die Adaptions- [Brockhaus19] und Anpassungsbeispiele [Brockhaus17] in der Brockhaus-Enzyklopädie analysiert werden, dann fällt auf, dass die Organismen keine »Daten« (z. B. Sprache), sondern nur z. B. Organe oder Kreislauf-. Atem- und Stoffwechselleistungen anpassen.

Dieser Sachverhalt findet auch Einklang, wenn die Definition einer Adaption von Subramanian und Chung [2002a, b] (vgl. auch Abschnitt 3.6.7) herangezogen wird. Die Verfasser dieser Definition definieren zwar ein »System« nicht näher, aber da sie Adaptierbarkeit

untersuchen, kann davon ausgegangen werden,[43] dass ein System in Einklang mit der IEEE-1471 Definition verstanden wird: »System is a set of components that accomplishes a specific function or set of functions« [IEEE 1471-2000] (zitiert nach [Garland und Anthony 2003]). Software ist nur die Funktionalität, die in den Komponenten vorgehalten wird.

Gegenstand für Adaptierbarkeit. Wird der Adaptionsgegenstand »Software« betrachtet, der bei einer Untersuchung der Adaptierbarkeit bzw. für adaptierbare Software relevant ist, so können gewisse Überschneidungen mit dem von adaptiver Software festgestellt werden. Je nach Stakeholder ergibt sich ein anderer Begriff von Software, der als Adaptionsgegenstand dient. Wird nur der Anwender betrachtet, der die Software an seine Präferenzen anpassen möchte, kann dieser lediglich unter einer begrenzten Anzahl an Optionen und Parametern wählen (Heineman [1998] bezeichnet dies als Customization). Für die Anwendungsentwickler bietet eine parametrisierbare Software auch Adaptierbarkeit, doch wirklich Anpassbarkeit an neue oder geänderte Anforderungen wird nur auf Ebene des Entwurfs, der hinter der entworfenen Software steht, erreicht.

Es kann auch argumentiert werden, dass zu einer Software, die Adaptierbarkeit unterstützen soll, auch die Dokumentation gehört. Denn wenn die Dokumentation in einem proprietären Format geschrieben ist, kann sie viel schwerer an ein anderes Ausgabeformat angepasst werden als wenn sie beispielsweise im Docbook-Format [Walsh 2004] verfasst ist. Ähnliche Überlegungen können für die Grafikdateien angestellt werden.

Deshalb kann die gestrichelte Linie in Abbildung 3.7 auf der vorherigen Seite, die den Begriff »Software« für adaptierbare Software begrenzt, je nach Auffassung deckungsgleich mit der gestrichpunkteten oder noch weitumfassender als derzeit eingezeichnet gesehen werden. Damit kann eine Deckungsgleichheit mit adaptiver Software zustande kommen (wenn Code nicht unbedingt als ausführbarer Code, sondern auch als Quellcode betrachtet werden kann).

Ergebnis. Eine genaue Aufteilung für den Gegenstand Software bezüglich Adaptivität und Adaptierbarkeit mit richtig und falsch kann hier nicht angegeben werden, da jeder Autor einer adaptiven Software für sich berechtigt behaupten kann, dass seine Software adaptiv genannt werden kann, obwohl beispielsweise die Anwendungsdaten, die angepasst werden, nicht zu (einer Funktionalität) einer Software gehören.

[43] Dies ist natürlich eine wage Vermutung, denn nirgendwo in [Subramanian und Chung 2002a, b] steht konkret geschrieben, dass unter System das Gleiche verstanden wird wie es in IEEE 1471-2000 formuliert ist. Wird allerdings der gesamte Text der zwei Paper gelesen, so kann mit einiger Sicherheit behauptet werden, dass eine Gleichheit vorherrscht. Eine solche Herleitung von Aussagen entbehrt jedoch jeglicher Wissenschaftlichkeit.

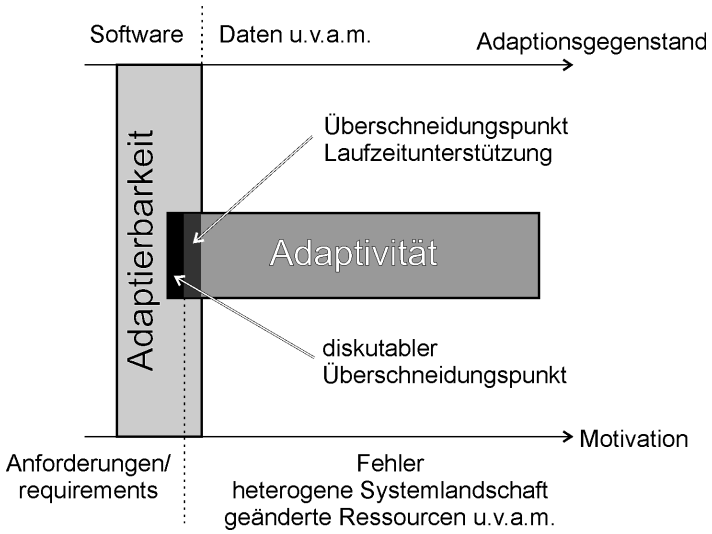

Abbildung 3.8.: Zusammenhang zwischen den Thematiken Adaptivität und Adaptierbarkeit

Deshalb kann die noch ausstehende Beantwortung der Fragen 1 und 2 von Seite 23 nicht beantwortet werden (kann auch eine Software als adaptiv bezeichnet werden, wenn sie nicht wirklich sich selbst, sondern die Anwendungsdaten anpasst). Die Literatur bescheinigt beiden Softwarearten Adaptivität. Ein Beurteilung nach richtig oder falsch kann und soll nicht vorgenommen werden. Eine solche einschränkende Aufteilung läge auch nicht im Interesse der Arbeit, zumal mit der Software-Architektur ein Adaptionsgegenstand vorgegeben ist.

Adaptive Software-Architektur

Eine »adaptive Software-Architektur« in der vorliegenden Arbeit zeichnet sich dadurch aus, dass sich die Software-Architektur selbst zur Laufzeit adaptieren kann. Damit findet die Adaption auf Software-Architektur-Ebene statt. In Unterkapitel 4.1 wird untersucht, welche Adaptionsmechanismen für das Adaptionsobjekt Software-Architektur möglich sind.

3.6.5. Zusammenhang Adaptivität und Adaptierbarkeit

Der Zusammenhang zwischen Adaptivität und Adaptierbarkeit von Software ist in Abbildung 3.8 dargestellt. Wie zu sehen ist, bewegen sich adaptive und adaptierbare Software auf zwei verschiedenen Ebenen. Über die Horizontale breitet sich die Thematik von adaptiver Software aus. Sie wird charakterisiert durch einen unterschiedlichen Adaptionsgegenstand und durch eine unterschiedliche Motivation für eine Adaption durch Software, wobei die Achsen, obwohl beide hori-

zontal angeordnet, unabhängig voneinander sind. Die Vertikale füllt adaptierbare Software aus.[44]

Beide Thematiken haben zwar etwas mit Adaption zu tun, aber sie haben nicht viel gemeinsam. Ein erster Unterschied ist, dass bei adaptierbarer Software der Gegenstand nur die Software selbst ist, wohingegen adaptive Software alle möglichen Facetten umfasst. Einander gegenüberstellend kann folgende Definition gegeben werden:

> *Adaptierbare Software* ist Software, die (leicht) adaptiert werden kann. Dabei bezeichnet *Adaptierbarkeit* das Maß, wie leicht die Software angepasst werden kann. Dahingegen ist *adaptive Software* Software, die sich anpasst. Der Begriff *Adaptivität* kann als Maßzahl dafür verwendet werden, inwiefern eine Software Adaptionseigenschaften gegenüber einer nicht-adaptiven oder anderen adaptiven Software besitzt.

Abgrenzbar sind adaptive und adaptierbare Software auch durch eine Motivation, nämlich geänderte Anforderungen, die nur bei adaptierbarer Software vorkommt — wobei dieser Punkt diskutabel ist (dazu später mehr). Außerdem sind bei einer Recherche hinsichtlich adaptiver respektive adaptierbarer Software unterschiedliche Probleme zu lösen:

Für die Problemstellung bei *adaptiver* Software können folgende Beispiele gegeben werden (kurzgefasst, die Anwendungslogik, die hinter einer Adaption steht, ist hier entscheidend):

- Wie kann Adaptionsbedarf festgestellt werden?
- Wie wird die richtige Adaptionsmaßnahme ausgewählt?
- Wie werden Adaptionsmechanismen in der Software implementiert?
- Wie wird festgestellt, dass die Software nach der Adaption besser ist, d. h. eine bessere Anpassung aufzeigt?

Die Problemstellung, die bei *adaptierbarer* Software zu lösen ist, lautet dahingegen wie folgt: Wie kann Software entworfen werden, die leicht angepasst werden kann? Dieser Frage nachgehend müssen Ansätze von Softwareentwicklungsmethoden (z. B. modellgetriebene Softwareentwicklung) und Architekturansätze aufgezeigt werden, die eine leichte Änderbarkeit/Anpassbarkeit der Software zur Folge haben.

In zwei kleinen Bereichen sind Überschneidungspunkte für adaptive und adaptierbare Software festzustellen. Der erste Bereich ist der Überschneidungspunkt für die Laufzeitunterstützung, der andere Bereich ist der eben schon genannte diskutable Überschneidungspunkt bezüglich geänderter Anforderungen.

[44] Würden in einer Abbildung noch mehr Dimensionen zur Verfügung stehen, dann müsste hier unter anderem noch die Zeit berücksichtigt werden.

Überschneidungspunkt Laufzeitunterstützung

Der in Abbildung 3.8 auf Seite 64 mit 70 % Schwarz gekennzeichnete Bereich umfasst Problemstellungen, die gleichsam durch adaptierbare und auch adaptive Software gelöst werden müssen. Normalerweise zielt Adaptierbarkeit auf Anpassungen der Software im Rahmen der Entwicklung dieser ab (Evolution). Wird Software betrachtet, die *zur Laufzeit* angepasst wird (engl. »dynamic updatable«, z. B. Komponenten hinzufügen oder entfernen im laufenden Betrieb), dann ist dieser Fall ebenso für adaptive Software von Bedeutung. Denn ob die Software durch den Menschen angepasst wird oder die Software die Adaption vornimmt, die unterliegende Middleware muss die Adaption im laufenden Betrieb auch unterstützen. Folglich besteht bei adaptiver Software der Problembereich einerseits aus der Unterstützung der Adaptionslogik, andererseits aber auch aus der Unterstützung durch die Laufzeitumgebung, die die Adpationsmechanismen zur Laufzeit ermöglicht.

Diskutabler Überschneidungspunkt — geänderte Anforderungen

Ein weiterer Überschneidungspunkt (schwarz gekennzeichnet) ergibt sich, wenn sich Software »an geänderte Anforderungen« anpassen soll. Grob betrachtet erscheint dies widersinnig: Woher soll beispielsweise ein Mailclient wissen, dass der Nutzer sein Adressbuch nicht mehr über LDAP[45] verwalten will, sondern nun die Unterstützung von ACAP[46]- oder IMSP[47]-Funktionalität fordert.[48] Wie soll eine Software diese geänderten Anforderungen erkennen und sie umsetzen?

Gegenbeispiel. Jedoch konnte beispielhaft eine Literaturquelle gefunden werden, die ihrer Architektur die Eigenschaft zuweist, sich an ändernde Anforderungen anpassen zu können [Eracar und Kokar 2000].

Die Autoren vorgenannter Arbeit definieren Anforderungen als die Funktionalität der Software, d. h. wie die Software auf bestimmte Eingaben aus ihrer »Umgebung« reagieren soll.[49] Gemäß Zave und Jackson [1997], die sagen, dass *alle* Aussagen, die im Rahmen einer Anforderungsermittlung/Systemanalyse (requirements engineering) gemacht worden, Aussagen über die Umgebung sind, argumentieren Eracar und Kokar folgendermaßen. Eingaben und Ausgaben eines Programms sind Teile der Umgebung des Problembereichs und jeder

[45] Lightweight Directory Access Protocol
[46] Application Configuration Access Protocol
[47] Internet Message Support Protocol
[48] Anders als LDAP ermöglichen ACAP und IMSP auch das Recherchieren und Bearbeiten eines Adressbuchs im Offline-Modus, d. h. ohne mit dem Server verbunden zu sein. Der IMAP-Client Mulberry ist der dem Autor einzig bekannte Mailclient, der vorgenannte Protokolle unterstützt.
[49] Im Original: »The most typical meaning of the term “software requirements” is the functionality of the software, i. e., how it should respond to particular inputs from its “environment”.« [Eracar und Kokar 2000, S. 3]

Änderung dieser steht im Verhältnis mit der Änderung von Anforderungen des Programms. Diese Sichtweise wird auch berücksichtigt im formalen Ansatz für die Softwarespezifikation, wo die Spezifikation der Funktionalität durch Vor- und Nachbedingungen ausgedrückt wird. Die Vorbedingungen spezifizieren, welche Beziehungen durch die Programmeingaben (program inputs) erfüllt werden sollten, damit das Programm korrekt arbeitet. Deshalb muss jede Änderung in den Eingaben an/für das Programm, die in einer Verletzung der Vorbedingungen resultiert, als Änderung der Anforderungen an die Software betrachtet werden. Deshalb werden Änderungen in den Eingaben an das Programm als Änderungen in der Spezifikation behandelt.

In ihrer Arbeit untersuchen Eracar und Kokar Adaptivität am Beispiel von Anwendungen für die Bild-/Objekt-/Zielerkennung. Bei dieser Art Software besteht die Schwierigkeit, alle möglichen Szenarien vorherzusehen, mit denen sich die Sensoren später befassen müssen. Zusätzlich zu diesem kommt ein weiteres Problem hinzu: Durch die Komplexität bei der Verarbeitung schleicht sich jede denkbare Zufälligkeit in die Sensordaten ein. Folglich wird die Software ohne eine komplette Spezifikation der Eingaben, der Funktionalität und der Randbedingungen (constraints) gebaut.

Die Änderungen in den Eingaben an das Programm aus der Umgebung bestehen in der Fallstudie von Eracar und Kokar [2000] im Folgendem: Bilder mit binärer vs. Kodierung für Graustufen, rauschfreie vs. rauschbehaftete Bilder (noisy-free/noisy), Objekttypen in den Bildern, Ausrichtung von Kanten in Einklang mit den Bildbegrenzungen vs. unbeschränkte Orientierung der Kanten. Da diese spezifischen Änderungen in der Umgebung nicht in den initialen Spezifikationen vertreten sind und während des Entwurfs des Bilderkennungsprogramms von Eracar und Kokar nicht berücksichtigt wurden, stellen sie unerwartete Änderungen in der Anforderungsspezifikation dar.

Für dieses Problem stellen die Autoren in ihrer Forschungsarbeit einen Lösungsansatz vor — der Ansatz ist außerdem ein generischer, der nicht abhängig von der Domäne der Bildverarbeitung ist. Ihr Lösungsansatz basiert darauf, die Anwendung als Regelungssystem zu bauen und Rückkoppelung einzubeziehen.

3.6.6. Wer, was, wann, warum Adaption

In den vorangegangenen Ausführungen wurde bei der Analyse von Übersetzungen, allgemeinen Begriffsdefinitionen in Enzyklopädien, Beispielen in der Literatur und schließlich Definitionen für »adaptive Software« ein breites Spektrum ersichtlich. Dieses Ergebnis allein resultiert natürlich nur in der Erkenntnis, dass Adaptivität und Adaptierbarkeit für Software eine breite Anwendung finden. Für eine Diplombearbeitung, in der Ansätze und Empfehlungen für »adaptive Software-Architekturen« aufgezeigt werden sollen, ist ein solches Ergebnis selbstverständlich wenig wert. Aus den vielen Varianten von ad-

aptiver und/oder adaptierbarer Software muss eine Teilmenge gewählt werden, für die eine Bearbeitung im weiteren Verlauf der Ausführungen erfolgt. Wie kann eine solche Einschränkung erfolgen?

Eine Einschränkung für adaptive und für adaptierbare Software kann bezüglich der Fragestellungen *Wer*, *Was*, *Wann* und *Warum* erfolgen:

Wer nimmt Adaptionen vor (Adaptionssubjekt)? Für diese Arbeit können hierfür der Mensch, d. h. ein Wesen mit höherer Intelligenz, und Software unterschieden werden. Einfach formuliert nimmt bei adaptiver Software die Software selbst die Adaptionen vor, bei adaptierbarer Software ist der Mensch das Adaptionssubjekt. Er muss es aber nicht sein, da sicher in der Zukunft Softwareprozesse genügend Intelligenz besitzen werden, um Anpassungen an anderer Software vorzunehmen (Stichwort: Wer steuert und verändert »die Matrix«?). Zu beachten ist auch, dass Adaptierbarkeit, also die substantive Form von »adaptierbar«, die Einfachheit bezeichnet, mit der eine Software angepasst werden kann; vgl. Abschnitt 3.6.2.

Was wird adaptiert (Adaptionsobjekt)? Sind es die Daten, ist es die Kommunikation, ist es die Interaktion mit dem Benutzer, ist es die Software selbst, ist es das Verhalten etc. (vgl. Abschnitt 3.6.4)? Für Adaptionsobjekte gibt es viele Möglichkeiten. Die Wahl des Gegenstandes ist meist durch eine Motivation für eine Adaption bedingt, d. h. ausgehend vom Problem werden Lösungsmöglichkeiten gesucht (hier Adaption).

Wann wird die Adaption vorgenommen? Erfolgt die Adaption während der Entwicklungszeit, Compile-Zeit, Link-Zeit, Inbetriebnahme, Ausführungszeit, Laufzeit, Wartungszeit der Software (vgl. Tabelle 3.5 auf der nächsten Seite)?

Warum wird etwas adaptiert (Motivation)? Die Gründe hierfür können sehr, sehr vielfältig sein. Je nachdem, ob adaptierbare oder adaptive Software betrachtet wird, können sie grundverschieden sein. Am signifikantesten ist sicherlich, dass bei adaptierbarer Software geänderte Anforderungen berücksichtigt werden müssen. Eine Motivation für adaptive Software kann in vielerlei Hinsicht gegeben sein. Abschnitt 3.6.3 nennt hierfür einige Beispiele, die im Rahmen des Literaturstudiums ersichtlich wurden.

Je nachdem, welche Werte die vier W's annehmen, ergeben sich verschiedene Konstellationen für adaptive oder eben nur adaptierbare Software.

Auch wenn Adaption nicht für Software betrachtet wird, dann helfen diese W-Fragen weiter. Ein Adaptionssubjekt kann dann ein Tier

Entwicklungszeit	Compile/Linking	Inbetriebnahme	Betriebs-/Einsatzzeit	
			Ruhezeit	Laufzeit
Software wird spezifiziert und implementiert		Assemblieren und Deployen	System wird/ist heruntergefahren	System wird ausgeführt und erfüllt Dienstanfragen

Tabelle 3.5.: Mögliche Zeitpunkte für eine Adaption. Die Zeiten sind nicht immer disjunkt voneinander. Für das Beispiel der Anpassung einer Software an neue Anforderungern wird deren Spezifikation angeglichen, betroffene Komponenten müssen neu kompiliert oder assembliert werden und das Update geschieht entweder, indem das System heruntergefahren wird (Ruhezeit), oder im laufenden Betrieb. Eine Software, die sich selbst anpasst, erledigt dies ausschließlich während der Betriebszeit, da sie für diese Anpassungen spezifiziert worden ist. Lieberherrs Beispiele für »adaptive Software« sind dagegen größtenteils in der Entwicklungszeit anzusiedeln.

(eben jeder Organismus) oder beispielsweise der sich in manchen, teuren Allround-Brillen befindende Werkstoff sein, der sich bei Sonneneinstrahlung selbstständig verdunkelt (an die Lichtverhätnisse anpasst). Diese W-Fragen sind nicht nur für Software-Anpassung gültig.

Doch es soll nicht versucht werden, in eine Allgemeingültigkeit abzuschweifen. Stattdessen muss endlich der Begriff Adaption geklärt werden — mittlerweile verschlingt die Begriffsbestimmung in der vorliegenden Arbeit immerhin 69 Seiten (abzüglich Einleitung). Nach Klärung des Adaption-Begriffs wird eine Fokussierung des genauen Themas mithilfe der W-Fragen vorgenommen.

3.6.7. Adaption

Was genau ist eine Adaption? Welche Kriterien müssen erfüllt sein, die einen Adaptionsprozess von einem »normalen« Prozess unterscheiden? In Abschnitt 3.3 auf Seite 26 wurde hierzu lediglich formuliert, dass Adaption »Dinge verbessert«. Das ist für eine wissenschaftliche Definition unzureichend. Außerdem müsste dann jeder Adapter oder jeder Adaptionsprozess »Verbesserer/Verbesserungsprozess« lauten.

Ziel dieses Abschnittes ist es, eine Art Lackmustest zu entwickeln, der einen Adaptionsprozess von einem »normalen« Prozess und damit eine adaptive von einer nicht-adaptiven Anwendung unterscheiden kann.

Bevor Adaption genau definiert werden soll, wird in der Literatur *eine* verfügbare Definition gesucht.

Verfügbarer formaler Ansatz für Adaption

Ein Ansatz, Adaption mehr formal zu beschreiben, ist in [Subramanian und Chung 2002a, b] zu finden. Dieser Ansatz wird im Folgenden geschildert.

Adaption wird definiert als Änderung in einem System in Reaktion auf die Änderung der Umgebung (environment). Spezifischer ausgedrückt wird die Adaption eines Software-Systems (S) ausgelöst von einer Änderung (δ_E) von einer alten Umgebung (E) zu einer neuen Umgebung (E'). Die Adaption resultiert in einem neuen System (S'), das idealerweise die Bedürfnisse seiner neuen Umgebung (E') erfüllt. Damit kann Adaption formal definiert werden als folgende Funktion:

$$E \times E' \times S \rightarrow S', \text{ wobei } \mathsf{erfüllt}(S', \mathsf{bedürfnisseVon}(E')) \tag{3.1}$$

Des Weiteren gilt, dass ein System adaptiv ist, wenn eine Adaptionsfunktion existiert. Das Anpassungsvermögen bezieht sich dann auf die Fähigkeit des Systems, eine Adaption vorzunehmen.[50]

Bei einer Adaption sind demnach drei Aufgaben zu erfüllen. Diese sind

1. die Fähigkeit, δ_E zu erkennen,

$$E' - E \rightarrow \delta_E \tag{3.2}$$

2. die Fähigkeit zur Bestimmung, welche Änderung δ_S für ein System S in Berücksichtigung von δ_E vorzunehmen ist,

$$\delta_E \times S \rightarrow \delta_S \tag{3.3}$$

   und

3. die Fähigkeit, die Änderung herbeizuführen, um das neue System S' zu generieren.

$$\delta_S \times S \rightarrow S', \text{ wobei } \mathsf{erfüllt}(S', \mathsf{bedürfnisseVon}(E')) \tag{3.4}$$

Die erfüllt-Funktion muss dabei zwei Anforderungen erfüllen: Sie muss sowohl validieren als auch verifizieren, dass das geänderte System (S) tatsächlich den Bedürfnissen der geänderten Umgebung (E') entspricht.

Dieser Sachverhalt ist vereinfacht in Abbildung 3.9 auf der nächsten Seite dargestellt.

[50] Anmerkung zur Übersetzung: Im Original lautet dieser Absatz folgendermaßen: »A system is adaptable if an adaptation function exists. Adaptability then refers to the ability of the system to make adaptation.« Da hier eine aktive Eigenschaft beschrieben wird (»ability of a system to make adaptation«), muss »adaptable« mit *adaptiv* übersetzt werden (»adaptability« mit *Anpassungsvermögen/-fähigkeit* ebenso; vgl. Abschnitt 3.1 und Resultat zur Unterscheidung adaptiv/adaptierbar auf Seite 28).

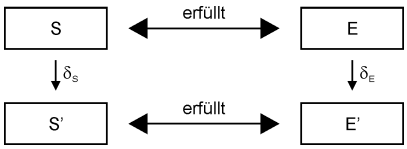

Abbildung 3.9.: Erklärung der Symbole für die Definition einer Adaption [Subramanian und Chung 2002b]

Analyse des Ansatzes

Die Definition für eine Adaption, wie sie Subramanian und Chung [2002b] vornehmen, erscheint mit einer mathematischen Unterlegung klar formuliert. Für ein klares Verständnis ergeben sich für den Autor allerdings noch vier Fragen:

1. Was ist als Umgebung (environment) definiert?
2. Was wird angepasst, d. h. was ist das »System«? Was genau gehört zu einem System und was nicht?
3. Eine Anpassung durch reines Probieren (z. B. Mutation und anschließende Selektion) wird nicht berücksichtigt?
4. Ist es unbedingt notwendig, dass *nach* der Änderung des Systems eine Validierung und Verifizierung durch die erfüllt()-Funktion durchgeführt wird?

Umgebung. Die Autoren der Definition einer Adaption geben für die Umgebung einige Beispiele an: Keyboard, Maus, Front-Panel-Schalter, eine andere Anwendungen u. a. Es ist anzunehmen, dass sich diese Umgebung mit der Umgebung deckt, die bereits in Unterkapitel 3.5 teilweise erörtert wurde.

System. Das »System«, an dem die Anpassungen vorgenommen werden, wird von den Autoren der Definition nicht definiert. Im Rahmen des Abschnittes 3.6.4 in Abbildung 3.7 auf Seite 62 wurde ein System entsprechend der IEEE-1471-2000-Definition mit einer gestrichpunkteten Linie als Code oder Menge von Komponenten, die die Funktionalität erbringen, charakterisiert. Es kann angenommen werden, dass dieses Verständnis im Sinne von Subramanian und Chung ist (vgl. Anmerkung zu einer solchen Interpretation in Fußnote 43 auf Seite 63).

Damit wäre diese Definition einer Adaption nicht anwendbar für die Daten, die Kommunikation oder andere Bestandteile eines von Software betriebenen Systems. Der Schluss, dass Adaption nur das System umfasst, wäre ein falscher, denn Adaption ist ein allgegenwärtiger Prozess sowohl für Organismen als auch für andere Objekt als Software

(vgl. Fußnote 4 auf Seite 22). Außerdem kann in dieser Arbeit nicht beurteilt werden, welche Beispiele aufgrund des Adaptionsgegenstandes eine adaptive Software umfassen und welche nicht (eine solche Einteilung wird lediglich hinsichtlich des Adaptionssubjektes vorgenommen).

Adaption durch Probieren. Der Brockhaus vermerkt für eine Adap(ta)tion, dass diese sowohl als kurzzeitige Anpassung in Reaktion auf Veränderungen in der Umgebung passieren kann als auch als »... langdauernder stammesgeschichtl. Prozeß unter der Wirkung von Evolutionsfaktoren (Isolation, Mutation, Selektion u. a.)« vollzogen werden kann [Brockhaus19]. Subramanian und Chung berücksichtigen mit Angabe der Funktionen 3.1 bis 3.3 nur die Adaption als Reaktion, die Adaption im Rahmen der Evolution jedoch nicht. Dem Brockhaus folgend muss ein System für eine Adaption nicht nur Änderungsbedarf und -operationen bestimmen können (eine Reaktion auf die Veränderung der Umgebung vornehmen), sondern das System kann auch getrieben durch die Evolutionsfaktoren eine Adaption durchführen.

Die Ausprägung der Evolutionsfaktoren ist bei Software etwas anders geartet als bei Organismen, sie ist aber mit letzterer sehr gut vergleichbar. Eine *Mutation* kann durch Code-Generatoren geschehen; eine Änderung verfügbarer Parameter ist auch möglich — und der einfachere, aber von den Möglichkeiten beschränktere Weg. Die *Selektion*, d. h. die Auslese der brauchbaren von den weniger brauchbaren Kandidaten, wird durch eine Evaluation vorgenommen (eine Vermehrung der besseren Kandidaten durch Fortpflanzung ist schwer vorstellbar, es sei denn, das Design des gut angepassten Kandidaten wird auf andere (Neu-) Entwicklungen übertragen). Die Evaluation geschieht dadurch, dass entweder die Software ihre Performance, den Servicegrad, selbst bewertet oder die Selektion durch den Nutzer der Software (andere Software oder Mensch) geschieht.

Soll dieser Aspekt berücksichtigt werden, dann müssen die Aussagen zu den Funktionen 3.1 bis 3.3 relativiert werden. Die vorgenannten Funktionen bekommen nicht das Alleinstellungsmerkmal als Voraussetzung für eine Adaption, sondern sie existieren parallel zu der Adaptionsvariante, die durch die Evolutionsfaktoren bestimmt ist.

Validierung nach Adaption. In dem Ansatz von Subramanian und Chung wird nach Angabe der notwendigen Fähigkeit/Funktion 3.4 postuliert, dass die erfüllt()-Funktion eine Validierung und Verifikation durchführt, ob das *geänderte* System tatsächlich den Bedürfnissen der geänderten Umgebung (E') entspricht. Das würde bedeuten, dass bei jeder Adaption *anschließend* eine Validierung/Verifikation stattfindet. Hier ergäbe sich nach Meinung des Autors ein Widerspruch zu einigen in der Literatur vorhandenen Systemen, bei denen eine vermeintliche Adaption durchgeführt wird.

Werden die Beispiele 5 (angepasste Wartungszeit), 6 (Formatanpassung) von Abschnitt 3.2 auf Seite 24 sowie die Transcoding-Systeme OPERA [Lemlouma und Layaïda 2002] und das Framework der TU Dresden [Hübsch *et al.* 2003] herangezogen, dann kann nach einfacher Analyse festgestellt werden, dass bei diesen keine Evaluation *nach* der Adaption stattfindet. Die Beispiele im Einzelnen erörtert:

1. Das System für die Wartungsarbeiten überprüft die Systemauslastung und stößt das Backup oder die Speicherbereinigungsarbeiten dann an, wenn die Auslastung am niedrigsten ist. Auslöser für eine Adaption ist wahrscheinlich die Unterschreitung eines Schwellwertes der Systemauslastung. Wenn die Wartungsarbeiten zu einer angepassten Zeit durchgeführt worden sind (im Vergleich zu einer starren Zeit, z. B. jeden Morgen 9:00 Uhr), dann erfolgt danach in keiner Weise eine Evaluation, ob die vom System gewählte Zeit tatsächlich eine gute war.

   Eine Evaluation und Verifikation würde stattfinden, wenn nach erfolgter Ausführung der Arbeiten die Systemauslastung dahingehend beobachtet werden würde, ob sie noch niedriger unter den Schwellwert sinkt. Daraufhin könnten für den nächsten Wartungsjob zwei Adaptionen durchgeführt werden: (1) Wie gewohnt wird die Wartung nicht zu einer starren Zeit, sondern zu einer an die (niedrige) Systemauslastung angepassten Zeit ausgeführt, und (2) zusätzlich wird der Schwellwert an die Historie angepasst. Im letzten Fall optimiert sich das System, wenn festgestellt wird, dass der vorgegebene Schwellwert zu hoch oder zu niedrig bezüglich der tatsächlich minimalsten Auslastung eingestellt ist.

2. Für das Beispiel der Formatänderung ist die fehlende Validierung leichter festzustellen. So prüft nach Meinung des Autors das System lediglich, welches Format vonnöten ist und wählt dementsprechend für die Kommunikation mit dem Partner Kilometer oder Meilen (bzw. 2- oder 4-stellig) als Format. Keinesfalls probiert das System mögliche Varianten durch und prüft, welche spezielle Variante passt.[51]

3. Am einfachsten kann die Widerspruchsführung für die Inhaltsadaption der Adaptionsframeworks durchgeführt werden. Bei einer Clientanfrage wertet der Server die Anfrage aus und entscheidet, welche Ressource er ausliefert. Falls keine fest vorgegebene oder gecachte passende Variante vorliegt, wird eine passende Ressource erzeugt, indem sie aus einer Master-Ressource oder einer in einem anderen Format vorliegenden Ressource generiert (adaptiert) wird. Der Vorgang geschieht nach einfachen Regeln. Wenn

[51] Eine Trial-Error-Vorgehensweise, wie zuvor ausgeschlossen, wäre denkbar. Allerdings nur unter der Voraussetzung, dass die Funktion 3.3 auf Seite 70 (Bestimmung der notwendigen Änderung δ_S in Berücksichtigung der Änderung δ_E) nicht als unbedingte Voraussetzung gilt.

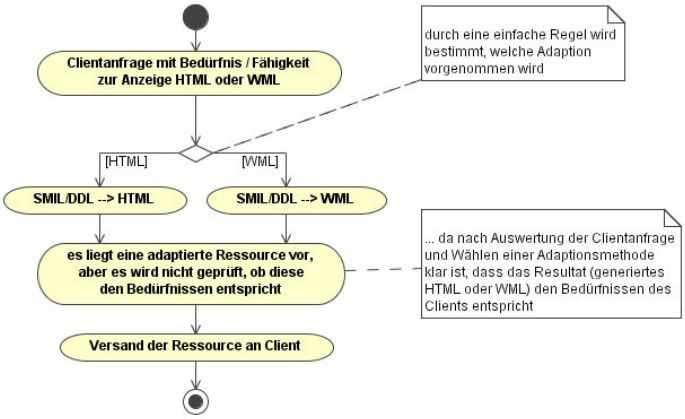

Abbildung 3.10.: Bei vielen Beispielen (hier Transcoding) erfolgt nach der Adaption keine Evaluation, sondern die Adaptionsmethode wird durch einen Determinismus bestimmt (Auswertung der Bedürfnisse) und anschließend nur ausgeführt. Nach der Ausführung wird das Resultat nicht dahingehend überprüft, ob es den Bedürfnissen tatsächlich entspricht.

der Client nur WBMP[52]s anzeigen kann, wird ihm das angeforderte Bild in diesem Format gesendet. Wenn ein Client nur cHTML unterstützt, dann wird ihm die Seite nicht im HTML- und auch nicht im WML-Format zugesandt, sondern in cHTML — falls vorhanden oder generierbar.

Die Entscheidung, in welcher Form eine angeforderte Ressource an den Client geschickt wird, wird basierend auf der Anfrage getroffen. Dann erfolgt eine Adaption (Generierung einer passenden Ressource oder einfach nur Selektieren einer passenden vorhandenen Ressource) und der anschließende Versand der für den Client passenden Ressource. In keiner Weise findet nach oder vor dem Versand eine Validierung oder Verifikation statt, ob die Ressource tatsächlich den Bedürfnissen des Clients entspricht (vgl. Abbildung 3.2 auf Seite 31 und Abbildung 3.10 oben).

Der in Abbildung 3.10 dargestellte Ablauf einer Adaption ist bei vielen anderen Beispielen auffindbar. Es erfolgt zuerst eine Analyse der Bedürfnisse, danach die derterministische Wahl einer Adaptionsmethode mit anschließender Durchführung, und dann liegt das Resultat vor. Was nicht erfolgt, ist die Validierung des Resultates nach Durchführung der Adaptionsmethode.

Entweder findet bei den vorgenannten Beispielen und allen anderen, die nach dem gleichen (deterministischen) Muster verfahren, keine Adaption statt oder die Definition für eine Adaption von [Subramanian und Chung 2002a, b] ist unzureichend formuliert.

[52] Wireless Bitmap

Die Definition für eine Adaption ist nach Meinung des Autors nicht zufriedenstellend.

Ein neuer Ansatz kann wie folgt aussehen.

Neuer formaler Ansatz

Der neue formale Ansatz baut auf dem von Subramanian und Chung [2002a, b] auf. Als erstes werden System und Umgebung geklärt. Bei dem Ansatz werden diese, gemäß den W-Fragen, definiert als:

Adaptionsobjekt (kurz Objekt, O), an dem die Adaption vorgenommen wird.

Adaptionssubjekt (kurz Subjekt, S), das die Adaptionen vornimmt.

Adaptionskontext (kurz Kontext, K). Der Begriff Kontext ist dabei nicht gleichzusetzen mit dem gleichnamigen Begriff aus der Forschungsrichtung Context-Awareness, sondern er ist mehr als dieser (vgl. Abschnitt 3.6.1).

Des Weiteren wird die Evaluationsfunktion erfüllt() mit einer kleinen Änderung übernommen. Diese Funktion liefert nicht mehr einen Boole'schen Wert zurück (d. h. *ob* die Bedürfnisse erfüllt werden), sondern eine Zahl größer oder gleich Null, die den Grad des Angepasstseins bestimmt:

$$\text{erfüllt}(O, \text{bedürfnisseVon}(K)) = \begin{cases} 0 & \text{wenn die Bedingung nicht erfüllt wird} \\ > 0 & \text{wenn die Bedingung erfüllt wird} \end{cases} \tag{3.5}$$

Der von der erfüllt()-Funktion ermittelte Wert ist umso größer, je besser die Bedingung erfüllt wird (hier die Erfüllung der Bedürfnisse von K durch O).

Eine Adaption wird definiert als die Änderung (δ_O) eines Objektes (O), um

1. dem gleichen Kontext besser zu entsprechen (eine präventive Aktion) oder
2. einer Änderung des Kontextes (δ_K) zu entsprechen (eine Reaktion).

Wenn das Adaptionsobjekt (O) ein Bestandteil des Adaptionssubjektes (S) ist, sodass das Subjekt die Adaption an sich selbst vornimmt, dann ist S *adaptiv*.[53]

[53] Zu den Überlegungen für ein Adaptionssubjekt und Adaptionsobjekt Software siehe die Ausführungen in Abschnitt 3.6.4.

Fall 1 — Adaption als präventive Aktion. Die Adaption, die mithilfe der Evolutionsfaktoren passiert, kann ganz kurz definiert werden. Sie besteht aus einer einzigen Funktion, die im ersten Teil den Vorgang der Änderung beschreibt (δ_O, durch zielgerichtetes oder zufälliges Probieren) und im zweiten Teil voraussetzt, dass das Angepasstsein eines Objektes der besseren Entsprechung des Kontexts entspricht:

$$\begin{aligned} \delta_O : \quad O \quad &\rightarrow \quad O', \text{ wobei} \\ \text{erfüllt}(O', \text{bedürfnisseVon}(K)) \quad &> \quad \text{erfüllt}(O, \text{bedürfnisseVon}(K)) \end{aligned} \tag{3.6}$$

Fall 2 — Adaption als Reaktion. Der zweite Fall entspricht nahezu dem Ansatz von Subramanian und Chung und muss lediglich wegen der Berücksichtigung der erfüllt()-Funktion abgeändert werden (Ersetzung der Boole'schen Funktion durch einen nummerischen Rückgabewert, der den Grad der Anpassung angibt, und Beseitigung der Nachbedingung).

Adaption als Reaktion kann formal definiert werden als:

$$K \times K' \times O \rightarrow O', \text{ wobei } \text{erfüllt}(O', \text{bedürfnisseVon}(K')) > 0 \tag{3.7}$$

Bei der Adaption eines Objektes (O), die infolge einer Änderung (δ_K) von einem alten Kontext (K) zu einem neuen Kontext (K') geschieht, sind folgende drei Aufgaben zu erfüllen:

1. die Fähigkeit, δ_K zu erkennen,

$$K' - K \quad \rightarrow \quad \delta_K \tag{3.8}$$

2. die Fähigkeit zur Bestimmung, welche Änderung δ_O für ein Objekt O in Berücksichtigung von δ_K vorzunehmen ist,

$$\delta_K \times O \quad \rightarrow \quad \delta_O \tag{3.9}$$

   und

3. die Fähigkeit, die Änderung herbeizuführen, um das neue Objekt O' zu generieren.

$$\delta_O \times O \quad \rightarrow \quad O' \tag{3.10}$$

Mit der formalen Definition von Adaption als Abschluss konnten die Begriffe Adaptivität, Adaptierbarkeit sowie alle dazugehörigen Begriffe und Übersetzungen geklärt werden — und die Bearbeitung des Themas kann beginnen.

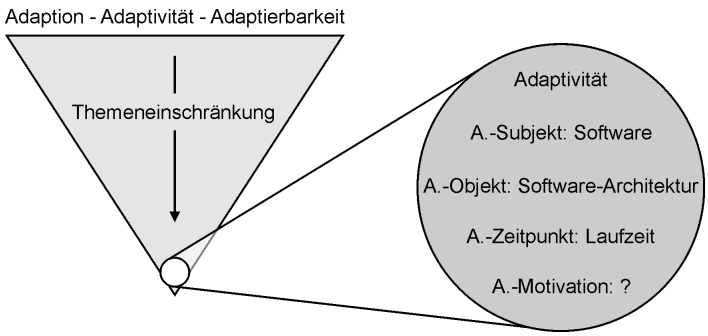

Abbildung 3.11.: Einordnung des Untersuchungsgegenstandes der vorliegenden Arbeit

3.6.8. Thema dieser Arbeit

Welche Konstellation der in Abschnitt 3.6.6 formulierten W-Fragen gilt in dieser Arbeit? Das grobe Thema der Arbeit lautet »adaptive Architekturen für verteilte Systeme«, der erste Schwerpunkt »Definition einer Software-Architektur bzgl./unter den Aspekten Adaptivität und Adaptierbarkeit«. Untersucht werden soll nur Adaptivität. Die nichtfunktionale Eigenschaft Adaptierbarkeit findet dabei keine Berücksichtigung. Adaptionssubjekt ist in der vorliegenden Arbeit die Software. Die Software soll adaptiv sein, d. h. sie soll Anpassungen/Änderungen an sich selbst vornehmen. Das »sich selbst«, also das Adaptionsobjekt, ist die Software-Architektur einer Software, d. h. Software wird nur auf der Architekturebene betrachtet.[54] Für den Zeitpunkt der Adaption wird die Einsatzzeit festgelegt, d. h. Änderungen an der Software-Architektur werden nicht im Rahmen der Entwicklung dieser vorgenommen (Anforderungsanalyse, Design, Implementierung), sondern geschehen zur Einsatzzeit. Für Einsatzzeit kann außerdem noch eine weitere Unterscheidung getroffen werden: Geschieht die Adaption zur Laufzeit (im laufenden Betrieb) oder muss das System gestoppt und wieder hochgefahren werden? Es wird beides untersucht, wobei eine Anpassung im laufenden Betrieb anzustreben, aber auch schwieriger ist. Ein Warum, eine Motivation gibt es keine. Die Software soll einfach nur (selbst-) adaptiv sein (siehe Abbildung 3.11).

Autonomic Computing und adaptive Software. Mit dem Papier »Adaptive Verteilte Systeme« [Dingler 2004], das die Industrieinitiativen Autonomic Computing (IBM), Sun N1 Computing und HP Adaptive Enterprise behandelt, wurde durch die Betreuung bereits eine Richtung für die vorliegende Arbeit vorgegeben. Die vorgenannten Initiativen

[54] Mit dem Adaptionsobjekt Software-Architektur und dem Änderungszeitpunkt zur Laufzeit ergibt sich damit die verwandte Thematik der *dynamischen Architekturen* (dynamic architectures) [Oreizy 1996], [Medvidovic und Taylor 2000] bzw. der *dynamischen Rekonfiguration* (dynamic reconfiguration) [Wermelinger 1999], [Almeida 2001]. Im weiteren Verlauf werden Begriff gleichsam verwendet.

propagieren das Selbstmanagement von IT-Systemen. Das Selbstmanagement muss zum einen erfolgen, um die Administrationskosten für die IT-Landschaft im Unternehmen zu senken, zum anderen werden IT-Systeme zunehmend derart komplex, dass sie nicht mehr durch den Menschen beherrschbar sind.

Wie passen diese Initiativen, die das *Selbstmanagement* von IT-Systemen propagieren, zu »adaptiven Software-Systemen«? Auf den ersten Blick passen Selbstmanagement und Anpassung schlecht zusammen, denn Selbstmanagement bedeutet, dass das System *etwas selbst macht*, Anpassung dahingegen bedeutet, dass *etwas angepasst wird*. Es wird also ein *wie* und ein *was* beschrieben: *Wie* verhält sich ein System? Es verhält sich autonom, indem es sich selbst reguliert etc. und damit Administration durch den Menschen (mit den damit verbundenen Kosten) niedrig hält. Hier liegt die Betonung darauf, dass das System etwas *selbst* macht. Dahingehen: *Was* macht ein System? Es nimmt Anpassungen vor.

Nimmt die »adaptive Software« Anpassung an den Daten vor, z. B. wenn es diese von dem PDF-Format in das RTF-Format transformiert, sodass das angepasste Dateiformat von Geräten gelesen werden kann, die kein PDF anzeigen können, dann ist kein Zusammenhang zum Selbstmanagement herstellbar. Wird der Adaptionsgegenstand Software betrachtet (wobei der Begriff Software nicht klar abgrenzbar ist), z. B. die Komponenten eines Systems, dann können Parallelen zum Selbstmanagement gezogen werden.

Die für das Selbstmanagement formulierten Self-X-Eigenschaften, wie selbst-heilend, selbst-konfigurierend, selbst-optimierend und selbst-schützend (vgl. Kapitel 2.4.1) zielen — wie es die Attribute bereits ausdrücken — nicht auf eine Adaption ab. Eher liefern sie eine Motivation für eine Aktion. Alles im allem passt sich das System jedoch selbst an: Ob nun für eine Kommunikationslösung die GSM[55]-Komponente gegen eine Komponente ausgetauscht wird, die UMTS[56] ermöglicht (Adaption an die heterogene Betriebsumgebung), oder der Austausch aufgrund einer Selbstheilung (alte Komponente ist kaputt oder besitzt Fehler), einer Selbstoptimierung (neue Komponente bringt besseren Algorithmus mit) oder einer Selbstkonfiguration erfolgt, es läuft auf eine Adaption der Software hinaus. Durch die wirtschaftlichen Vorgaben (Adaptionsziele), die von der Unternehmensleitung gestellt werden, muss sich die IT-Architektur anpassen (automatisch Server hinzu-/abschalten, konfigurieren und reparieren etc.), um kostenoptimal zu operieren.

So sind auch die für adaptive Software notwendigen Archichtekturansätze, d. h. die gesamte Architektur, die Adaptionsbedarf erkennt und durchführt, unabhängig von der Motivation meist gleich. Im Kapitel 5 werden Ansätze hierfür untersucht.

[55] Global System for Mobile Communications
[56] Universal Mobile Telecommunications System

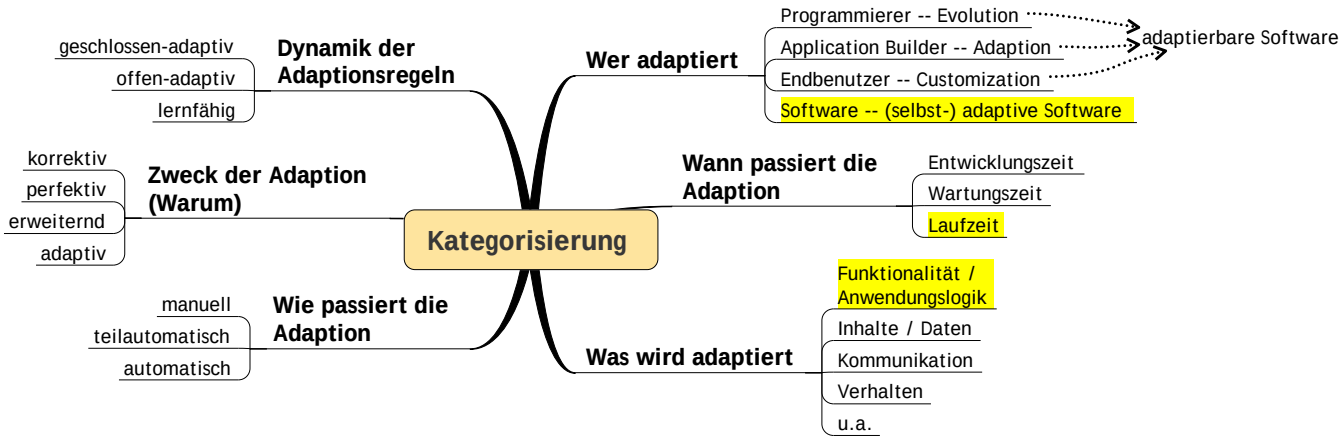

Abbildung 3.12.: Kategorisierung von Adaption

3.7. Adaptionsarten

Obwohl mit dem vorangegangenen Unterkapitel bereits ein Fazit gezogen wurde, ist das Ende des gesamten Kapitels noch nicht ganz erreicht. Abschließend werden im Folgenden die Arten der Anpassung der Software nach ausgewählten Kriterien kategorisiert (siehe Abbildung 3.12). Die Kategorisierung stellt keine Taxonomie für adaptive und adaptierbare Software da, da eine solche zu umfangreich wäre. Stattdessen ist sie auf die Thematik (auch gelb/grau gekennzeichnet) der vorliegenden Arbeit beschränkt und enthält weitere wichtige Begriffe für die weiteren Ausführungen.

Wer adaptiert?

Bei der Fragestellung, wer die Anpassung in der Software initiiert, kann folgende Unterscheidung vorgenommen werden:

Programmierer Modifikation der Softwarekomponente durch den Designer oder Programmierer. Heineman [1998] bezeichnet dies als Evolution.

Application Builder Anpassung für die verschiedenartige Verwendung einer Komponente durch den Application Builder. Heineman [1998] bezeichnet dies als Adaption.

Endbenutzer Die Anpassung der Softwarekomponente erfolgt durch den Endbenutzer durch Auswahl der Einstellungen aus einer, während der Implementierung festgelegten, begrenzten Anzahl an Optionen. Heineman [1998] verwendet hierfür den Begriff Customization.

Software Die Software passt sich selbst an. In der vorliegenden Arbeit wird der Begriff (selbst-) adaptive Software hierfür benutzt.

Nur der letzte Fall ist in dieser Arbeit Untersuchungsgegenstand, denn die anderen Fälle umfassen Adaptierbarkeit.

Wann passiert die Adaption?

Eine *adaptive* Software, wie sie in der vorliegenden Arbeit definiert wird, sollte es ermöglichen, die Adaption während der *Laufzeit* durchzuführen, ohne dass das System heruntergefahren wird. Die eigene Adaption durch Herunterfahren wird nicht angestrebt.

Dem Verständnis für »adaptive Software« nach Lieberherr folgend kann Adaption in allen Lebensphasen vorkommen.

Eine *adaptierbare* Software kann auch in allen Lebensphasen angepasst werden (vgl. Tabelle 3.5 auf Seite 69). Software, die während der Laufzeit angepasst werden kann, wird als *dynamisch adaptierbar* bezeichnet. Die Verwendung des gleichen Präfixes für Adaptivität angewandt resultiert dagegen in einer anderen Bedeutung.

Was wird adaptiert?

Bei dem Gegenstand der Adaption kann bei Software unter anderem zwischen der Adaption der Funktionaliät/Anwendungslogik, der Inhalte/Daten, der Kommunikation, dem Verhalten und anderem unterschieden werden (z. B. Routing-Algorithmus oder Anpassung der Transaktions- oder Sicherheitsregeln).

Für die Adaptionsgegenstände Software-Architektur (Teilaspekt von Software), Daten und Kommunikation werden in Kapitel 4 Adaptionsmechanismen aufgezeigt.

Wie passiert die Adaption (Grad der Autonomie)?

Die Adaption in der adaptiven Software sollte automatisch, oder zumindest teilautomatisch durchgeführt werden. Wenn die Adaption manuell durchgeführt wird, handelt es sich eher um eine Adaption des Softwaresystems (Objekt) mit dem Durchführer/Subjekt Mensch, d. h. die Software ist nicht wirklich adaptiv. Die teilautomatische Durchführung kann weiter in Unterkategorien aufgeteilt werden, je nach Grad der Beteiligung des Menschen. Es sind z. B. folgende Szenarien vorstellbar: (1 / managed) der Adaptionsbedarf wird vom System erkannt oder das System ist behilflich beim Beobachten, aber die notwendige Adaption wird vom Menschen ermittelt sowie durchgeführt, (2 / predictive) Adaptionsmöglichkeiten werden vom System vorgeschlagen, vom Menschen bewertet und eine davon ausgewählt oder (3 / predictive^{++}) Adaptionsbedarf und -lösung werden vom System ermittelt und dem

Menschen lediglich zur endgültigen Annahme vorgelegt (in Anlehnung an [Amberg *et al.* 2003]).

IBM hat eine ähnliche Einteilung für den Grad der Autonomie formuliert [Steinberg 2003a], [Worden 2004]. Diese verschiedenen Level wurden bereits in den Klammern angemerkt, wobei die hier erfolgte Einteilung mit (2) und (3) etwas feiner vorgenommen wurde. Neben *managed* und *predictive* unterscheidet IBM außerdem noch *basic*, *adaptive* (wie hier) und *autonomic*. Beim *basic*-Level, der untersten Ebene, zeigt das System kein selbsttätiges Verhalten, da alle Notwendigen Handlungen (z. B. Installieren, Monitoring, Aufrechterhalten des Systems) durch den Menschen vorgenommen werden. Diese Ebene entspricht einem nicht-adaptiven System. Der Unterschied von *adaptive* zu *autonomic* ist nur im Abstraktionsgrad der Regeln zu sehen. Bei dem *adaptive*-Level sind die Regeln an der IT orientiert (z. B. Komponenten und Tuning-Parameter), bei dem *autonomic*-Level werden die Regeln auf Basis einer globaleren Sicht und auf Basis von Business-Präferenzen aufgestellt (vgl. auch [Cheng 2003]).

Damit ergibt sich bei IBM die Reihenfolge

> basic → managed (1) → predictive (2 + 3) → adaptive → autonomic.

Diese Bezeichnungen sind in ihrer Anlehnung an das Selbstmanagement der IT im Unternehmen einschränkend, weshalb sie in der vorliegenden Arbeit keine Verwendung finden.

Speziell für das Adaptionsobjekt Software-Architektur wird die Änderung, die durch die Software selbst auf Basis des Zustandes der Komponenten und der Topologie vorgenommen wird, »programmed« [Endler 1994], [Goudarzi 1999] oder »constrained run-time modification« [Oreizy 1996] genannt. Änderungen, die durch das Befragen des Nutzers geschehen, finden in der Literatur die Bezeichnungen »ad-hoc« [Endler 1994], »evolutionary« [Goudarzi 1999] oder einfach nur »runtime change« [Oreizy 1996].

Dynamik der Adaptionsregeln

Soll das System geschlossen-adaptiv, offen-adaptiv (dynamisch-adaptiv) oder lernfähig sein? Ein System wird als *geschlossen-adaptiv* bezeichnet, wenn es in sich abgeschlossen ist und somit kein Hinzufügen von neuem adaptiven Verhalten möglich ist. Ein System ist *offen-adaptiv (dynamisch adaptiv)*, wenn ein neues adaptives Verhalten durch neue Adaptionspläne während der Laufzeit eingeführt werden kann. [Oreizy *et al.* 1999].

Eine Steigerung der offen-adaptiven Eigenschaft kann erfolgen, wenn nicht der Mensch, sondern die Software selbst neue Adaptionsregeln einführt. Eine solche Software ist *lernfähig*.

Bei einer offen-adaptiven Anwendung können die Adaptionsregeln zur Laufzeit geändert/adaptiert werden. Dieser Sachverhalt ist nur

bedingt gleichsetzbar mit »adaptierbaren« Systemen, denn bei diesen wird die eigentliche Anwendung angepasst. Bei einer offen-adaptiven Anwendung wird nicht die eigentliche Anwendung angepasst, sondern die Regeln für die Adaption (möglicherweise in Verbindung mit neuen redundanten Komponenten). Dadurch wird erweitertes adaptives Verhalten erzeugt. Als solche stellt eine offen-adaptive Anwendung eine Untermenge der »dynamisch-adaptierbaren«, d. h. zur Laufzeit adaptierbaren Systemen dar.

Zweck der Adaption

Eine Kategorisierung für die Zielsetzung der Adaption ist in [Ghezii *et al.* 1991] (zitiert nach [Oreizy *et al.* 1998]) zu finden und wird im Sinne von [Ketfi *et al.* 2002] um den Punkt »erweiternde Adaption« ergänzt:

Adaptive Adaption[57] Auch wenn die Anwendung korrekt läuft, dann muss sie dennoch manchmal angepasst werden, wenn sich die Betriebsumgebung ändert (Betriebssystem, Hardware oder andere Anwendungen, von denen die Anwendung abhängig ist). In diesem Fall wird die Anwendung in Reaktion auf die Änderungen in der Umgebung angepasst.

Korrektive Adaption Wenn festgestellt wird, dass die laufende Anwendung Fehlverhalten zeigt, dann muss die Komponente identifziert werden, die nicht korrekt arbeitet, und durch eine neue Version (von der angenommen wird, dass sie korrekt ist) ersetzt werden. Dabei bietet die neue Version genau die gleiche Funktionalität wie die alte. Es werden nur die Fehler beseitigt.

Perfektive Adaption Ziel der perfektiven Adaption ist es, die Anwendung zu verbessern, auch wenn diese korrekt abläuft. Beispiel: Wenn die Implementation (Algorithmus oder Programmierumsetzung) einer Komponente nicht genug optimiert ist, dann wird diese Komponente durch eine neue mit einer besseren Implementierung ersetzt. Ein anderes Beispiel ist die Duplizierung einer Komponente, wenn sie nicht imstande ist, alle Anfragen in vorgegebener Zeit zu beantworten und somit die gesamte Anwendung ausbremst.

Erweiternde Adaption Als Antwort darauf, dass neue Funktionalität vom Benutzer benötigt wird und wenn diese zur Entwicklungs- oder Deployment-Zeit noch nicht berücksichtigt werden konnte,

[57] Natürlich ist diese Benennung in ihrem Sinn schwer verständlich. Es wurde »Adaption« statt »Evolution« gewählt, weil es sich bei allen Änderungen an der Software um Adaption/Anpassung handelt, auch wenn es die Evolution der Software betrifft. Auch in der Literatur herrscht keine Einigung. Ghezii *et al.* [1991] verwenden die Einteilung in korrektiv, adaptiv und perfektiv in ihrer Kategorisierung in Verbindung mit dem Begriff Evolution, Ketfi *et al.* [2002] dahingegen i. V. m. Adaption.

muss die Anwendung um neue Komponenten erweitert werden. Die neuen Komponenten erweitern die Anwendung um zusätzliche Funktionalität.

3.8. Zusammenfassung

Das vorliegende Kapitel führte eine umfangreiche Begriffsbestimmung für Adaptierbarkeit und adaptierbare Software sowie Adaptivität und adaptive Software durch. Dazu wurden Einträge in Enzyklopädien, Übersetzungen aus Wörterbüchern, einige Beispiele sowie verschiedene Definitionen angeführt und analysiert. Zusammenfassend lässt sich sagen, dass Adaption im Zusammenhang mit Software sehr unterschiedlich verstanden wird. Dieser Wirrwarr wurde durch eine systematische Vorgehensweise grundlegend untersucht.

Positiv zu beobachten war bei den Definitionen für adaptierbare/adaptable und adaptive Software, dass sie einstimmig bestimmen, dass bei adaptiver Software die Software selbst Anpassungen (an sich) vornimmt (aktive Rolle) und bei adaptierbarer/adaptable Software der Software eine passive Rolle zugeschrieben wird und lediglich ausgedrückt wird, dass oder wie gut eine Software angepasst werden kann. Damit wurde die durch die Übersetzungen eingangs geschaffene Unklarheit ausgeräumt. (Eine Ausnahme hierbei bildet Lieberherr, der in [Lieberherr 1998] bei seiner Erläuterung der Interpretierbarkeit von Kontext dies für adaptive und für adaptierbare Software tätigt, obwohl seine Definition nur für adaptive Software angegeben wird.)

Neben der klaren Bestimmung von Aktivität und Passivität der Software gibt es auch einige Unzulänglichkeiten. Die Definitionen, obwohl ein wichtiger Grundstein in einer wissenschaftlichen Abhandlung, sind sehr unterschiedlich ausgeprägt und zu bemängeln. Es wurde auch beobachtet, wie das unachtsame Erweitern einer Definition, die von anderen Autoren stammt, zu Widersprüchen führen kann. Außerdem können die benutzten Begriffe, die die Definitionen formen, durch den Leser unterschiedlich oder nur unzureichend interpretiert werden. Durch den Verfasser einer Definition werden Begriffe auch unterschiedlich ausgelegt (besonders Kontext). Die Definitionen können missverstanden werden, wenn das Umfeld fehlt, in dem die Definitionen stehen, oder wenn Teile weggelassen werden. Hier ist für eine Begriffsfindung dringend anzuraten, auch das Umfeld zu untersuchen, in dem die Definition steht (dies wurde besonders bei der 3. Definition deutlich).

Die Definitionen sind außerdem auf einen bestimmten Problembereich eingeschränkt, indem Adaptionsgegenstand, Adaptionsmotivation und Verfahrensweise (z. B. Adaption erfolgt nach Evaluierung) angegeben werden. Selbstverständlich kann keine Kritik dafür erfolgen, dass Definitionen beschränkten Geltungsbereich besitzen, da eine Definition oft lediglich mit der Absicht angegeben wird, für eine Arbeit die

Thematik zu beschränken. Jedoch helfen derartige Definitionen nicht jemandem, der wissen möchte, was adaptive Software ist. Die Disparität der spezifischen Definitionen lässt vermuten, dass etwas nicht adaptiv ist, obwohl es laut einer anderen Definition oder einem Beispiel als adaptiv bezeichnet wird. Hier ist es von Vorteil, eine allgemeingültigere Definition angegeben zu können.

Eine bessere Definition wurde in diesem Kapitel formuliert:

> *Adaptierbare Software* ist Software, die (leicht) adaptiert werden kann. Dabei bezeichnet *Adaptierbarkeit* das Maß, wie leicht die Software angepasst werden kann. Dahingegen ist *adaptive Software* Software, die sich anpasst. Der Begriff *Adaptivität* kann als Maßzahl dafür verwendet werden, inwiefern eine Software Adaptionseigenschaften gegenüber einer nicht-adaptiven oder anderen adaptiven Software besitzt.

Verglichen mit den bisherigen Definitionen ist diese Definition sehr einfach und nähert sich damit den allgemeingültigen Erklärungen in Enzyklopädien an. So fehlt dieser Definition der Auslöser für eine Adaption. Dieses Fehlen kann jedoch nicht als Mangel angesehen werden, sondern hat die Ursache darin, dass der Grund für eine Adaption sehr verschieden und vielfältig sein kann. Er kann nicht durch ein Wort formuliert oder durch eine Aufzählung von möglichen Begriffen umschrieben werden.

Eine wichtige Voraussetzung für eine Anpassung ist, dass die Software ihre Umgebung und sich selbst beobachtet. Die geänderte Situation und Beschaffenheit der Umgebung, der von Kommunikationspartnern und der eigenen können Auslöser für eine Adaption sein. Ebenso kann der Anlass für eine Adaption durch den Drang nach Verbesserung der Performance gegeben sein. Bezüglich der Informationsversorgung ergibt sich eine Parallele zum Forschungsbereich Context-Awareness, für den ebenso die Gewinnung, Verarbeitung und Speicherung von Informationen über die Ausführungsumgebung wesentliche Ziele sind. Für adaptive Software ist jedoch mehr als Kontext relevant, sie benötigt Informationen aus *allem Beobachtbaren*, d. h., die gesammelten Informationen umfassen mehr als den Kontext-Begriff.

Mithilfe der Erfahrungen, die bei der Erörterung des Kontext-Begriffes gesammelt werden konnten, wurde eine neue Definition für den Begriff Kontext angegeben.

Wie die Diskussion mit anderen Kollegen gezeigt hat, wird Kontext/Context-Awareness unterschiedlich und schwierig verstanden. Hier wäre es erwägenswert, nachdem dieser Forschungsbereich in vier eher praktisch orientierten Arbeiten an der TU Dresden bearbeitet wurde, Kontext und Context-Awareness einer theoretischeren Untersuchung zu unterziehen. Für die vorliegende Arbeit wurde das Resultat gezogen, dass Kontext nur einen Teilaspekt der notwendigen Informationen für eine Adaption umfasst.

Für eine weitere Begriffsbestimmung von Adaption wurden die vier W-Fragen formuliert (wer adaptiert, was wird adaptiert, wann wird adaptiert und warum wird adaptiert), die helfen, die weithin mögliche Adaption in Software zu katalogisieren. Mittels dieser W-Fragen wurde eine weitere Bearbeitung des Themas wie folgt eingeschränkt:

1. Wer adaptiert: Software.
2. Was wird adaptiert: Software-Architektur.
3. Wann wird adaptiert: zur Laufzeit.
4. Warum wird adaptiert: unbekannt.

Damit wird die Adaptivität auf Ebene der Software-Architektur untersucht. Außer Acht gelassen wird eine Bearbeitung der nichtfunktionalen Eigenschaft Adaptierbarkeit, da sie hinsichtlich der Recherche eine gänzlich andere Thematik umfasst.

Ergebnisse kurz formuliert

- Einführung in die breite Thematik,
- Aufzeigen von Schwachstellen in Übersetzungen, Definitionen und Zuarbeiten,
- allgemeingültige Definition für adaptive Software, die nicht nur auf den Sinn einer einzelnen Arbeit beschränkt ist und bei der Beispiele nicht durch andere Definitionen ausgeschlossen werden,
- kleiner Beitrag zu einem besseren Kontext-Verständnis mit einer neuen Definition für Kontext,
- adaptive Software benötigt mehr Informationen als der Kontext-Begriff umfasst,
- formaler Ansatz für Adaption,
- es kann nicht eindeutig argumentiert werden, ob eine adaptive Software nur wirklich sich selbst oder auch die Anwendungsdaten und die Kommunikation anpasst,
- letztendlich Begriffsbestimmung für die vorliegende Arbeit und eine Festlegung der W-Fragen-Aspekte, für die eine weitere Recherche erfolgt.

Der weitere Verlauf

Nachdem die Begriffsbestimmung abgeschlossen und für eine Fokussierung auf adaptive Software drei der vier W-Fragen geklärt wurden (außer »warum«), kann die eigentliche Bearbeitung beginnen. Im nächsten Kapitel werden dazu Adaptionsmechanismen, die auf Software-Architektur-Ebene operieren, zusammengetragen. Das darauf folgende Kapitel nimmt eine Recherche nach Unterstützungsmechanismen und Architekturansätzen, die zu Adaptivität in einer Software-Architektur führen, vor.

There is nothing permanent except change

Heraclitus of Ephesus, 535–475 v. Chr.

4 Adaptionsmechanismen

Nachdem die vorhergehenden Kapitel geklärt haben, was die Begriffe Adaption, Adaptierbarkeit, Adaptivität und Software-Architektur bedeuten, kann in diesem Kapitel die Erörterung bezüglich Adaptivität und Adaptierbarkeit in einer Software-Architektur erfolgen (erstes Schwerpunktthema »Definition einer Software-Architektur bzgl./unter den Aspekten Adaptivität und Adaptierbarkeit«).

Verständnis Adaptierbarkeit. Die Untersuchung von Adaptierbarkeit, wie es in der Aufgabenstellung steht, wird als »Möglichkeit-Adaptierbarkeit« verstanden, d. h., es wird untersucht, welche Adaptionsmechanismen (Änderungsoperationen) für eine Software-Architektur existieren (»was ist adaptierbar?«) — Beispiel: Operation A ändert die Software-Architektur derart, dass eine Anpassung hinsichtlich ... erfolgt. Die andere Möglichkeit wäre die Betrachtung von »Evolutions-Adaptierbarkeit«, d. h. die Untersuchung von Techniken und Methoden für die nicht-funktionale Eigenschaft Adaptierbarkeit einer Software (»wie kann eine Software/Software-Architektur bereits im Vorfeld so entwickelt werden, dass sie später leicht adaptiert werden kann?«) — Beispiel: Verfahrensweise X in der Software-Technologie ermöglicht die leichte (und damit aufwand- und kostenminimale) Änderbarkeit von Software. Die Software kann so besser angepasst werden. Dieser Aspekt wird *nicht* behandelt, weil er eine andere Thematik umfasst und nicht konform zum Thema »Adaptive Architekturen für verteilte Systeme« ist. Vergleiche hierzu auch Abschnitt 3.6.2.

Für einen umfassenderen Einblick in die Thematik von Adaption soll dieses Kapitel darüber hinaus Adaptierbarkeit der Daten und der Kommunikation betrachten. Damit wird der Fokus von Software-Architektur als Adaptionsgegenstand auf verteilte Anwendungen erweitert. Zwischen einer Software-Architektur und einer verteilten Anwen-

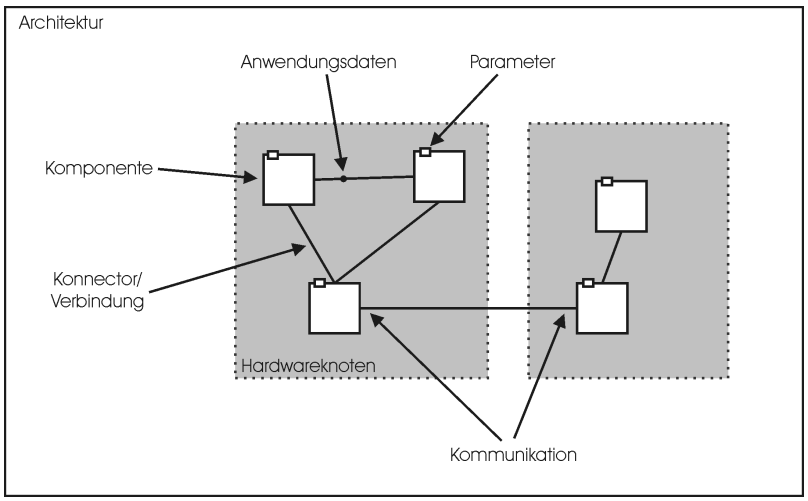

Abbildung 4.1.: Adaption einer verteilten Anwendung, bestehend aus der Software-Architektur (Komponenten und Konnektoren), der Kommunikation zwischen den Komponenten und den Daten

dung i. Allg. bestehen Zusammenhänge. Die verteilte Anwendung hat eine Architektur, bestehend aus Komponenten, die auf Hosts platziert sind. Zwischen den Komponenten findet eine Übertragung von Daten statt. Software-Architektur betrachtet nur die Architektur (und das Designprinzip etc.), hier werden überdies die Anwendungsdaten, die über die Konnektoren übertragen werden, und die Kommunikation beachtet (vgl. Abbildung 4.1).

Kritischer Hinweis

Es soll an dieser Stelle betont sein, dass die vorgestellten Mechanismen nicht allumfassend sind. So sind außer der Anwendungsstruktur, den Daten und der Übertragung auch andere »Bestandteile« einer Anwendung adaptierbar — dazu gleich mehr. Ebenso fehlt der Aufgabenstellung das Adaptionsziel, d. h. die Fragestellung, *an was* oder *warum* angepasst werden soll. Wie sich eine fehlende Vorgabe auswirkt, soll nachfolgend demonstriert werden.

Hierfür ein erstes Beispiel: In [Seifert und Lammersen 2004] ist ein interessanter Anwendungsfall für »adaptive Software« zu finden, bei dem die Software eine Anpassung an die beschränkten Sehfähigkeiten des Anwenders vornimmt. Kernpunkt dieser Anwendung ist, dass farbige Bilder eine bessere Benutzbarkeit (Usability) für den Nutzer bieten als schwarz-weiße oder in Graustufen gehaltene. Durch die Differenzierung in mehrere Farben können beispielsweise Bestandteile einer Geographie (im Beispiel ist ein Stadtplanausschnitt von Dresden gegeben) besser identifiziert werden, da der Fluss Elbe eine blaue Farbe, Grünflächen eine grüne, Straßen, Gebäude usw. eine entsprechend andere Farbe besitzen. Jedoch erfährt in diesem Beispiel die Nutzbarkeit

und der Vorteil der Nutzung von Farben eine Einschränkung, wenn der Anwender ein eingeschränktes Farbsehvermögen besitzt (z. B. Rot-Grün-Schwäche, Fähigkeit zur Differenzierung von Farbabstufungen). Deshalb nimmt die Anwendung zuerst zwei Tests zur Ermittlung der Farbsehfähigkeit vor und passt aufbauend auf diesen Nutzerprofilen fortan die Farbgestaltung von Bildmaterial entsprechend an.

Wird dieses Beispiel analysiert, so können drei Fakten geschlussfolgert werden: (1) Anpassungsgegenstand sind die Daten und (2) es erfolgt eine Anpassung aufgrund der eingeschränkten Farbsehfähigkeit des Anwenders — oder allgemeiner formuliert, eine Anpassung an die heterogenen Eigenschaften des Anwenders. Und (3): Es lassen sich ohne Zweifel noch mehr Beispiele für »adaptive Anwendungen« finden, bei denen Adaptionsmechanismen und Adaptionsziele identifiziert werden können.

Wird eine erste Einschränkung der Thematik vorgenommen, nämlich der Fokus auf Pervasive Computing[1], dann lassen sich neben den Adaptionsmechanismen, die bereits bei DaimlerChrysler [Vögler *et al.* 2004] (Transcoding) und der TU Dresden [Klamar 2004] (Transcoding, Fragmentierung, Caching, Prefetching u. v. a. m.) für die Adaption an die heterogenen Eigenschaften der Ausführungsumgebung gewonnen werden konnten, noch weitere Anwendungsfälle/Adaptionsmechanismen im Bereich Pervasive Computing finden.

So ist in [Schmidt *et al.* 1999a] ein Beispiel für eine Software zu finden, bei der ein PDA[2] in einem fahrenden Auto benutzt wird. Die Software erkennt die Geschwindigkeit des fahrenden Duos sowie die Schüttelbewegung des PDAs und passt entsprechend die Schriftgröße der auf dem PDA dargestellten Seite an, damit der Anwender diese besser lesen kann. Adaptionsziel ist hier wieder bessere Benutzbarkeit, Adaptionsgegenstand sind in gewisser Weise die Daten.

Ein ähnliches Beispiel ist im Rahmen von Pervasive Computing ein PDA, der die Helligkeitssituation erkennt und bei zu dunkler Umgebung automatisch die Hintergrundbeleuchtung einschaltet oder nachregelt. Adaptionsziel ist hier wie im vorgenannten Beispiel die bessere Benutzbarkeit, der Adaptionsgegenstand umfasst jedoch weder Daten, Kommunikation noch Software-Architektur.

. . . und die Schlussfolgerung

Nach diesen vier einfachen Beispielen sollte deutlich sein, dass eine Literaturrecherche für die Feststellung von Adaptionsmechanismen im Rahmen der vorliegenden Arbeit nicht sinnvoll ist, da eine Analyse und Katalogisierung der Adaptionsmechanismen ohne das Wissen eines Adaptionszieles oder einer konkreten Anwendung, bei der Möglichkeiten von Adaptivität untersucht werden könnten, unmöglich wird.

[1] Die Diplomarbeit läuft im Rahmen dieses Projektes bei DaimlerChrysler.
[2] Personal Digital Assistent

In der vorliegenden Arbeit wird deshalb ein pragmatischer Weg gewählt. Die Beschränkung erfolgt wie bereits eingangs angekündigt auf die Adaptionsgegenstände Software-Architektur (aufgrund der Aufgabenstellung) sowie der Daten und der Kommunikation. Für letztere zwei werden die umfassenden Ergebnisse aus der Dissertation von Springer [2004] in gekürzter Form herangezogen.

Die in der vorgenannten Arbeit ermittelten Basismechanismen sind vorrangig dazu geeignet, heterogene und limitierte Ressourcen der Endgeräte sowie der Übertragungsmedien und teilweise auch die Dynamik durch Mobilität zu unterstützen. Damit fällt das Beispiel der Farbanpassung für farbsehgeschädigte Anwender von Seite 88 heraus, denn die Veränderung der Farben lässt sich keinem von Springer genannten Basismechanismus für eine Adaption der Daten zuordnen (vgl. [Springer 2004, Abschnitt 3.2.1] und in der vorliegenden Arbeit Abschnitt 4.2). Mit der Transformation von Bilddaten kann zwar eine Umwandlung zwischen unterschiedlichen Farbräumen erreicht werden, aber die Transformation eines RGB[3]-Bildes in ein CMYK[4]-Bild ist nicht vergleichbar mit der Änderung der Farbe Rot zu Blau in einem Bild. Ebenso nicht eingeordnet werden können die Änderung der Schriftgröße und das Regeln der Hintergrundbeleuchtung für die vorgenannten Beispiele. Ein Ansatz wäre, sowohl den Wert der Farbe eines Objekts als auch die Schriftgröße und die Helligkeit als Parameter eines Objekt zu definieren und diesen Adaptionsmechanismus »Parameteränderung« zu nennen.

Bevor Möglichkeiten zur Adaption der Software-Architektur erörtert werden, soll noch eine andere Anmerkung zu »Software-Architektur« hinsichtlich der Basismechanismen zur Adaption der Daten und Kommunikation erfolgen: Neben dem wissenschaftlichen Charakter einer klaren Klassifikation und Taxonomie der Basismechanismen, wie sie Springer [2004] in seiner Arbeit vorgenommen hat, lässt sich zudem ableiten, welche Komponenten die Software-Architektur enthalten muss (z. B. Transcoding- und Extraktions-Komponente).[5] Die geschaffene Anwendung ist dann jedoch keine »adaptive Software-Architektur«, sondern es wurden Komponenten für die Software identifiziert, die Adaption vornimmt — die Identifikation von Komponenten gehört auch zu den Aufgaben, die im Rahmen der Software-Architektur ausgeführt werden (vgl. Kapitel 2.1).

[3] Rot–Grün–Blau (additives Farbmodell)

[4] Cyan–Magenta–Yellow–Key/blacK (4-Farben-Modell, das sich besonders für das Drucken eignet)

[5] Die Komponenten können beispielsweise mit dem Pipeline-Pattern kombiniert werden. Ansätze hierfür sind in [Klamar 2004] und [Springer 2004] zu finden.

4.1. Adaption der Software-Architektur

In diesem Abschnitt wird geklärt, welche Operationen möglich sind, die eine Anpassung der Software-Architektur zur Folge haben (erstes Schwerpunktthema »Definition einer Software-Architektur bzgl./unter den Aspekten Adaptivität und Adaptierbarkeit«).

4.1.1. Vorbetrachtung

Eine Definition von (Möglichkeit-) Adaptierbarkeit und Adaptivität in der/einer Software-Architektur ist nicht ohne weiteres möglich. Als problematisch sind hier zu nennen

1. die gleichzeitige Betrachtung von Adaptierbarkeit und Adaptivität,
2. es gibt nicht *die eine* Software-Architektur, sondern mehrere in Form von Sichten, und
3. Software-Architektur ist Abstraktion.

Gleichzeitig Betrachtung Adaptierbarkeit und Adaptivität

Das Hauptthema der vorliegenden Arbeit sind adaptive Software-Systeme. Dies impliziert, dass die Software die Adaptionen durchführt. Dennoch soll auch Adaptierbarkeit betrachtet werden. Und die Betreuungsgespräche brachten keine Klarheit, ob auch der Mensch Anpassung an der Software durchführt. Deshalb erfolgt in diesem Kapitel nicht nur eine Untersuchung für adaptive Software-Architekturen, sondern zudem für alle Anpassungen/Adaptionen auf Software-Architektur-Ebene im Rahmen der Evolution einer Software. Dies erfordert an manchen Stellen eine Differenzierung bezüglich des Initiators und der Möglichkeit der Adaption.

Problem Sichten — was genau sind diese Kästchen und Linien?

Eine Software-Architektur wird dokumentiert bzw. besteht in mehreren Sichten (vgl. Abschnitt 2.1.2 auf Seite 4). Selbstverständlich ist es weniger sinnvoll, im Rahmen der Untersuchung für »adaptive Software-Architekturen« beispielsweise die konzeptionelle Sicht oder die Layer/Schichten-Sicht zu betrachten. In der vorliegenden Arbeit sollen nur die Komponenten und ihre Verbindungen, die Konnektoren, untersucht werden. Doch auch diese Sicht wird unterschiedlich verstanden.

Verständnis C&C-Sicht. Für die Komponenten-und-Konnektoren-Sicht (kurz C&C-View/Sicht) sind zwei verschiedene Interpretationen/Einsatzfelder in der Literatur zu finden:

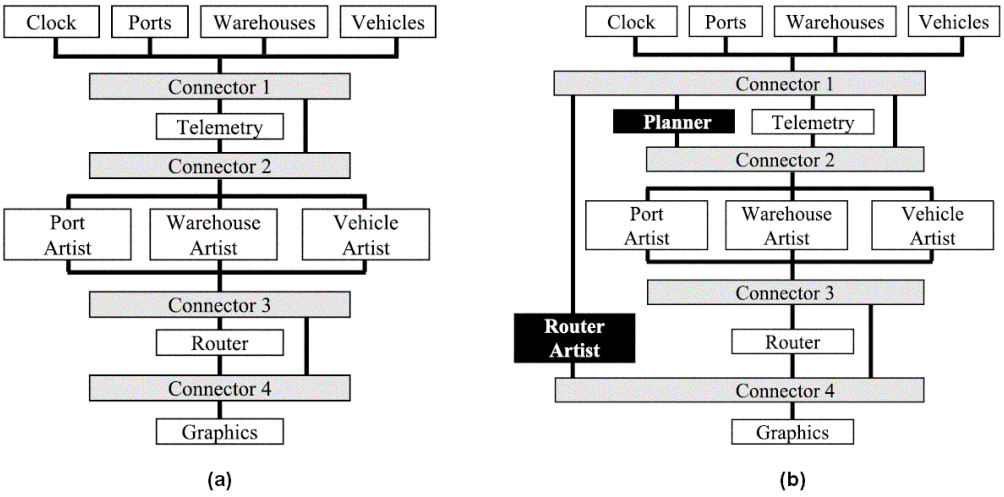

Abbildung 4.2.: C&C-Sicht der Benutzeroberfläche eines Cargo-Routing-Systems im C2-Style [Taylor *et al.* 1996] (a) vor und (b) nach einem Update zur Laufzeit [Oreizy *et al.* 1998]

1. Bei dem ersten Einsatzfeld sind verschiedene Komponenten*typen* über Konnektoren miteinander zu einer Anwendung verbunden. Diese Sichtweise ist konform mit den Komponentendiagrammen der UML [Jeckle *et al.* 2003] und entspricht typischen Architekturdiagrammen. Abbildung 4.2 zeigt die Software-Architektur der Benutzeroberfläche eines Cargo-Routing-Systems im C2-Style [Taylor *et al.* 1996]. Im Gegensatz zu der Sichtweise des nächsten Punktes erfüllt hier jede Architekturkomponente eine andere funktionale Aufgabe.

2. Die zweite Interpretation findet Gebrauch beispielsweise in [Cheng *et al.* 2002b]. Die Autoren in vorgenannter Quelle benutzen die C&C-Sicht für die Modellierung von mehreren Clients und Servern (siehe Abbildung 4.3 auf der nächsten Seite). Die Komponenten (Kästchen/Knoten des Graphen) sind bei ihnen nicht Komponententypen, sondern -*instanzen* (jeweils des Komponententyps Client/Benutzer oder Server). Hier können mehrere Komponenten die gleiche Funktion haben (alle drei dargestellten Server-Komponenten haben dasselbe Interface).

Je nachdem, welche Sichtweise für eine Interpretation der Kästchen und Linien der C&C-Sicht herangezogen wird, erschließt sich für die in den späteren Ausführungen genannten Adaptionsmechanismen ein anderer Sachverhalt.

Beispiel: Mechanismus Hinzufügen einer Komponente. Das Hinzufügen einer Komponente ist beispielsweise in Abbildung 4.2 (C2-Style) zu sehen. Hier wird durch das Hinzufügen eines Planers und einer

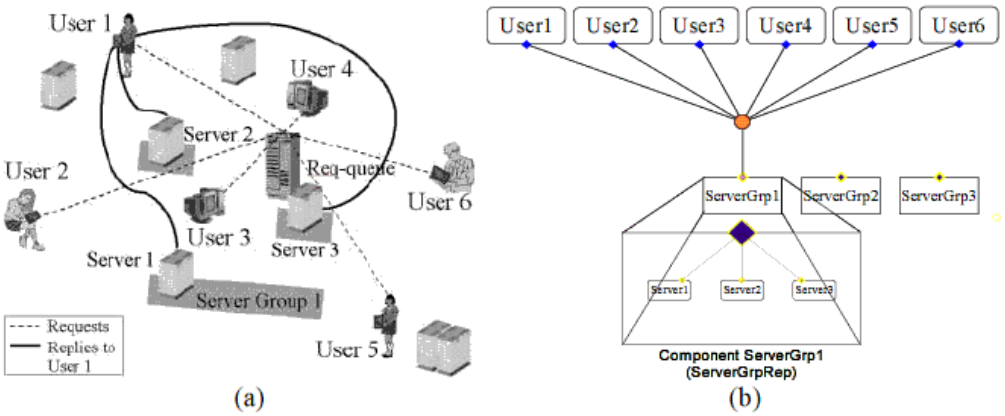

Abbildung 4.3.: Deployment-Architektur (a) und C&C-Sicht (b) eines Beispiel-Systems in [Cheng *et al.* 2002b]

Visualisierungskomponente (Router Artist) die *Funktionalität erweitert*. Ganz anders kann es aussehen, wenn der gleiche Mechanismus für die Sicht in Abbildung 4.3 (Client-Server-Beispiel) angewandt wird. Hier ergibt sich ein vollkommen anderer Tatbestand, wobei zwei Fälle zu unterscheiden sind:

1. *Es wird eine Server-Komponente hinzugefügt.* Das Hinzufügen einer Server-Komponente (Instanz) erweitert das System um *zusätzliche Ressourcen*. Durch den zusätzlichen Server kann das System mehr Anfragen bearbeiten.

2. *Es wird eine Client/User-Komponte hinzugefügt.* Das Hinzufügen einer Client-Komponte ergibt in dem System *mehr Last*, da mehr Anfragen zu bearbeiten sind.

Bei beiden Fällen wird *keine* zusätzliche Funktionalität eingebracht. Fall 1 behandelt eine Adaption, da sich das System an die zusätzliche Last anpasst bzw. seine Performance verbessert (die Verteilung der Anfragen auf mehr Server wird möglich, was sich in einem besseren Antwortverhalten äußert). Fall 2 dagegen behandelt *keine* Adaption. Das Hinzufügen/Aufkommen von mehr Benutzern entspricht einem normalen Systemagieren.

Fazit. Die Ausführungen zuvor haben gezeigt, dass es bei der Interpretation einer Klassifikation für Adaptionsmechanismen in einer Software-Architektur immens wichtig ist, was die Kästchen in einer C&C-Sicht darstellen (Typen oder Instanzen). Je nach verwendeter Sichtweise ergeben sich für einen Adaptionsmechanismus unterschiedliche Auswirkungen in der Software-Architektur. Außerdem ist es wichtig zu unterscheiden, in welchem Kontext ein Mechanismus steht, denn nicht jede Änderung im System ist ein Adaptionsmechanismus,

auch wenn die Änderung bildlich gesehen die gleiche ist wie die eines Adaptionsmechanismus'.

Abstraktion

Eine weitere, hier auch durch eine unterschiedlich mögliche Auffassung geprägte Problematik, betrifft die Abstraktion, die in Software-Architektur (Dokumentationen) getroffen wird. Dadurch sind Adaptionsmechanismen nicht immer klar abgrenzbar.

In Abbildung 4.4 auf der nächsten Seite ist dazu ein Beispiel zu sehen, das [Heise 2003] entstammt. Im Rahmen des EU-Projektes 6WINIT (IPv6 Wireless Internet Initiative, http://www.6winit.org/) wurde eine mobile Internetlösung auf Basis des neuen Internet-Protokolls IPv6 validiert. Das medizinische Notfallsystem *Guardian Angel* soll ohne Verbindungsunterbrechung zwischen den verschiedenen Funknetzen hin- und herschalten und selbstständig das für den jeweiligen Standort optimale Netz wählen können. Fährt ein mit dem Datenübertragungssystem ausgestatteter Krankenwagen über Land, verwendet *Guardian Angel* eine GPRS[6]-Verbindung über das GSM-Mobilfunknetz. In der Stadt wird automatisch auf das UMTS-Netz und im Bereich etwa eines krankenhauseigenen Funk-LAN[7]s auf dieses umgeschaltet. Dafür sind auf Architekturebene verschiedene Interpretationen möglich (siehe Abbildung 4.4 auf der nächsten Seite).

Ansätze in der Literatur

Im Folgenden werden einige ausgewählte Meinungen zur Adaption auf Architekturebene vorgestellt.

Oreizy *et al.* [1999] nennen das *Hinzufügen*, *Entfernen*, *Austauschen* und *Parametrisieren/Konfigurieren* der Komponenten und Konnektoren als Möglichkeiten für selbst-adaptive Systeme. Darüber hinaus kann auch die *Netzwerktopologie* der Komponenten und Konnektoren geändert werden.

de Lemos und Fiadeiro [2002] merken für selbst-heilende Systeme an, dass der Austausch von Komponenten nicht immer möglich sein muss, weil nicht immer für alle benötigten Dienste Redundanz verfügbar sein kann. In dem Fall kann keine Komponente mit einem »äquivalenten« Dienst gefunden werden, sondern es werden, so der Ansatz, Komponenten mit einem alternativen Dienst gesucht und verwendet, auch wenn der erbrachte Dienst in einem niedrigeren (downgraded) Modus ausgeführt wird. Die Konnektoren haben dabei die Aufgabe, den alternativen Dienst derart anzupassen, dass er kompatibel mit den Erwartungen der Komponente ist, die ihn für die Interaktion benötigt.

Kokar *et al.* [1999] nennen für ihr selbst-kontrollierendes Modell neben dem *Austausch* einer Komponente auch die *Transformation* oder

[6] General Packet Radio Service
[7] Local Area Network

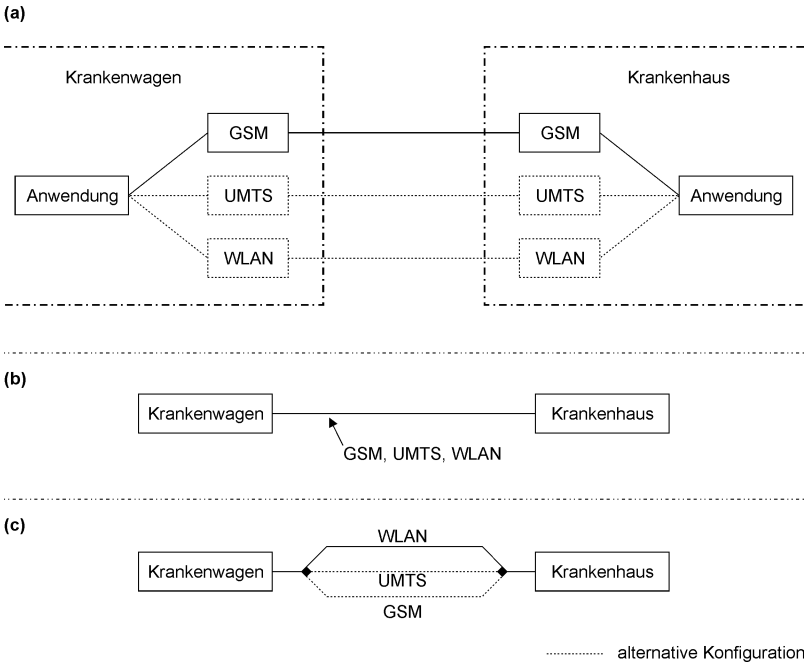

Abbildung 4.4.: Je nach Auffassung und Abstraktionsstufe ist das Krankenwagenbeispiel (a) eine direkte Rekonfiguration der Architektur, (b) eine Parameteränderung oder ein Austausch der Konnektoren oder (c) ein Beschreiten von alternativen Pfaden in dem Architekturgraphen

Komposition der Komponente, um den Servicegrad einhalten zu können. Ähnlicher Meinung sind Cobleigh *et al.* [2002], die ebenso den Austausch, aber auch die *Modifikation* der Komponente als Möglichkeit anbringen — neben der Zuordnung von neuen Ressourcen —, um auf die wechselnden Bedingungen zur Laufzeit zu reagieren.

Die genannten Operationen mögen umfassend in ihrer Aufzählung sein, bedürfen jedoch noch mehr Erklärung. So ist beispielsweise fraglich, welche Operation hinter der Parameteränderung eines Konnektors steht. Außerdem wird keine Angabe dafür gemacht, ob die Komponenten in der C&C-Sicht *Typen* oder *Instanzen* sind.

In der vorliegenden Arbeit werden die nachfolgend beschriebenen Mechanismen für adaptive Software-Architekturen betrachtet.

Betrachtete Adaptionsmechanismen

Die betrachteten Adaptionsmechanismen operieren auf der Ebene der Software-Architektur.[8] Die Überlegung, was in einer Software-Architektur angepasst werden kann, erfolgt durch die Betrachtung von Sichten. Nachfolgend kommen die Sichten Anwendungsarchitektur und Systemarchitektur in Betracht.[9] Die Anwendungsarchitektur beschreibt ein System aus der Sicht der Komponenten und Konnektoren — hinsichtlich verteilter Systeme erfolgt hier eine Betrachtung der funktionalen Verteilung. Die Systemarchitektur betrachtet zudem die Hardware-Geräte, auf denen später die Komponenten verteilt sind, d. h. hier wird die physische Verteilung/Zuordnung der Komponenten auf die Hardware, die Verarbeitungsressourcen anbietet, berücksichtigt. Des Weiteren werden die Adaption der Parameter der Komponenten, die Schnittstellenanpassung sowie die Anpassung der Adaptionsarchitektur berücksichtigt (siehe Abbildung 4.5 auf der nächsten Seite).

4.1.2. Adaption der Anwendungsarchitektur

Hinzufügen von Komponenten

Das Hinzufügen einer Komponente (siehe Abbildung 4.6 auf der nächsten Seite) unterstützt die erweiternde Adaption eines Systems. Die Anwendung erhält durch die Komponente zusätzliche Funktionalität.

Das Einbringen von »zusätzlicher Funktionalität« ist kritisch zu betrachten, da hier zu unterscheiden ist, wer (Software oder Mensch) das System adaptiert. Wirklich neue Funktionalität kann zurzeit nur durch den Menschen zu dem System hinzugefügt werden, da dieser die neuen Anforderungen erkennen und eine entsprechende Komponente

[8] Würden Mechanismen allgemein für verteilte Systeme betrachtet werden, dann kämen beispielsweise die Änderung des Routingalgorithmus', die Änderung der Transaktions- oder Sicherheitsregeln und andere in Betracht.

[9] Die Bezeichnung der Sichten erfolgt in Einklang mit [Medvidovic 1996] und [Lewandowski 2003].

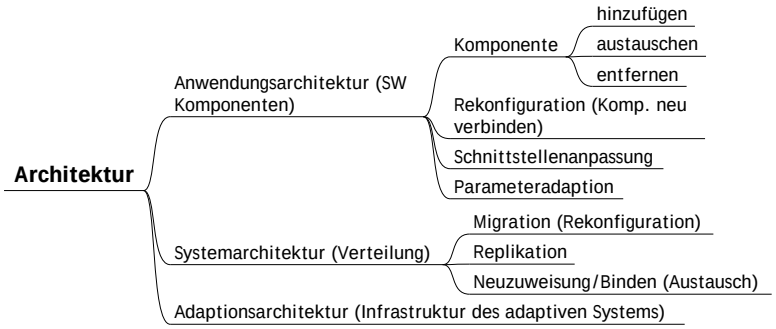

Abbildung 4.5.: Möglichkeiten zur Anpassung einer Software-Architektur

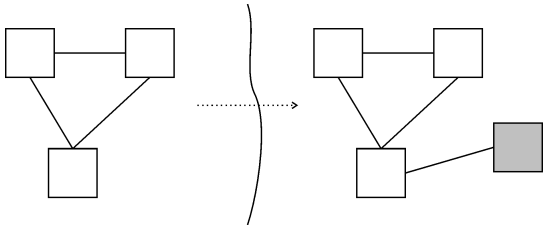

Abbildung 4.6.: Hinzufügen einer Komponente

bereitstellen kann. Gleichwohl können Komponenten hinzugefügt werden, um weitere Verarbeitungsschritte in die Applikation hinzuzufügen. In diesem Fall muss die Komponente bereits vorher modelliert worden sein und das Hinzufügen muss als Regel definiert sein.

Im Gegensatz zur Replikation einer Komponente (siehe Seite 105), was auch als Hinzufügen einer Komponente aufgefasst werden kann, wird hier eine Komponente neuen *Typs* hinzugefügt. Bei der Replikation wird dahingegen eine neue *Instanz* gleichen Typs/Implementierung hinzugefügt.

Wenn eine Komponente zu einem laufenden System hinzugefügt wird, dann muss sie berücksichtigen, dass das System nicht in seinem initialen Zustand ist. Deshalb ist es notwendig, dass die Komponente den Zustand des Systems in Erfahrung bringt und diesen mit ihrem Zustand synchronisiert. Ebenso muss die Komponente dem System bekannt gemacht werden, damit andere Komponenten die neue Komponente nutzen können.

Entfernen von Komponenten

Durch das Entfernen von Komponenten wird unbenötigte Funktionalität aus dem System entfernt (Abbildung 4.7 auf der nächsten Seite). Das Entfernen einer Komponente aus dem laufenden System wirft einige Probleme auf, die berücksichtigt werden müssen.

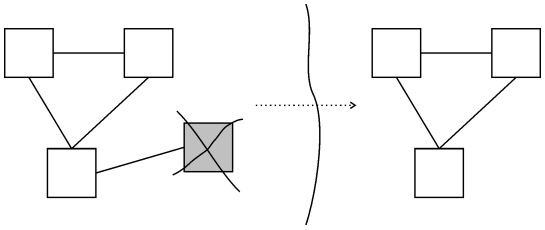

Abbildung 4.7.: Entfernen einer Komponente

Das Entfernen einer Komponente sollte nicht die Ausführung anderer Komponenten beinflussen. Das bedeutet unter anderem die Sicherstellung, dass es keine Komponenten gibt, die von der zu entfernenden Komponente abhängig sind. Ebenso muss sich die zu entfernende Komponente in einem stabilen Zustand befinden. Zum Beispiel wenn eine Komponente eine Datei oder eine Datenbank modifiziert, dann darf sie nicht entfernt werden, bevor der Schreibprozess abgeschlossen ist, da andernfalls Inkonsistenzen entstehen können. Eine weitere Herausforderung betrifft die Berücksichtigung von Daten und Nachrichten, die zwischen der zu entfernenden und anderen Komponenten ausgetauscht werden. Diese Daten dürfen nicht verloren gehen. Bevor die Komponente wirksam entfernt wird, müssen diese ausstehenden Nachrichten erst noch berücksichtigt werden [Ketfi *et al.* 2002].

Austauschen von Komponenten

Streng genommen kann das Austauschen von Komponenten (Abbildung 4.8 auf der nächsten Seite) als zusammengesetzte Folge der Operationen Entfernen und Hinzufügen einer Komponente angesehen werden. Jedoch ist es notwendig, die Austauschoperation separat zu betrachten. Denn zusätzlich zu den für das Entfernen einer Komponente notwendigen Berücksichtigungen sind meist folgende Prämissen zu beachten: (1) Der Zustand der alten Komponente muss zu der neuen Komponente transferiert werden[10]; (2) Beide Komponenten dürfen nicht simultan aktiv werden [Oreizy *et al.* 1998].

Der Austausch einer Komponente durch eine andere ermöglicht die Verwendung einer anderen Implementation. Die neue Version der Komponente kann andere funktionale Eigenschaften besitzen, die sich von der alten Version unterscheiden. Zum Beispiel kann die Komponente Fehler korrigieren (korrektive Anpassung; sowohl als Nachfolgeversion der auszutauschenden Komponente als auch in dem Sinne, dass die auszutauschende Komponente fehlerhaft ist und durch eine redundante Komponente ausgetauscht wird) oder einen verbesserten Algorithmus haben, der bessere und präzisere Ergebnisse liefert. Die

[10] Einen Einblick in die Thematik und den State-of-the-Art geben [Kasten und McKinley 2004], [Chen *et al.* 2001] und [Oreizy *et al.* 1998].

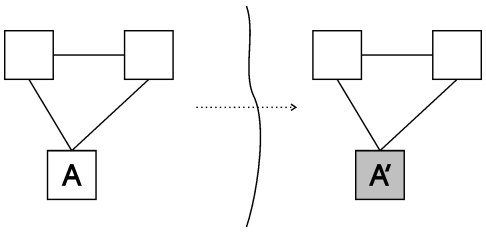

Abbildung 4.8.: Austauschen der Komponente A durch A'

neue Version der Komponente kann ebenso andere nicht-funktionale Eigenschaften besitzen, beispielsweise andere Ressourcenanforderungen oder eine bessere Performance.

Bezüglich des Initiators, der für den Austausch ausschlaggebend ist, muss eine Unterscheidung getroffen werden. So kann für das Beispiel des besseren Algorithmus' dieser sowohl durch die Software als auch durch menschliche Intelligenz angeboten werden. Der erste Fall liegt vor, wenn die Software zwei alternative Algorithmen für die Verarbeitung vorliegen hat und die Performance der Algorithmen abhängig von der Charakteristik der Eingabedaten ist (beispielsweise im Rahmen der Bildverarbeitung, wenn Ecken mithilfe eines Sobel- und eines Laplace-Algorithmus' erkannt werden können [Kokar *et al.* 1999]). Der Austausch einer Komponente durch eine mit einem besseren Algorithmus kann auch durch den Menschen erfolgen, wenn dieser den vorhandenen Algorithmus verbessert hat. Wenn die Software den Austausch vornimmt, dann müssen Alternativen entweder vorliegen oder generierbar sein. Der Mensch kann durch seine (noch) höhere Intelligenz zusätzliche Komponenten für einen Austausch generieren, hier im Rahmen der perfektiven und korrektiven Adaption.

Rekonfiguration

Konnektoren verbinden die Komponenten untereinander und ermöglichen somit die Kommunikation. Die Rekonfiguration der Verbindungen zwischen den Komponenten ermöglicht durch Neukombination neues Systemverhalten durch existierende Funktionalität. Dieser Vorgang wird als *Rekonfiguration der Anwendungsarchitektur* bezeichnet [Medvidovic 1996].

Schnittstellenänderung — eine Diskussion

Die Änderung/Adaption der Schnittstellen[11] ist ein besonders diskutierbares Thema. Deshalb erfolgt die Untersuchung getrennt danach, ob die Schnittstelle explizit adaptiert wird oder die Änderung als Folge einer anderen Adaptionsoperation auftritt.

[11] Bei ADLs und der neuen UML 2.0 auch Ports genannt.

Explizite Schnittstellenänderung. Autoren wie beispielsweise Ketfi *et al.* betrachten bei der Adaption von Komponenten das Verändern der Schnittstellen einer Komponente. So kann beispielsweise eine Schnittstelle (und ihre Implementierung) entfernt werden, wenn ersichtlich wird, dass diese nicht mehr benötigt wird, und dadurch die Performance verbessert werden [Ketfi *et al.* 2002]. In der vorliegenden Arbeit wird sich dieser Auffassung nicht angeschlossen.

Die Anpassung von Schnittstellen wird nicht für *adaptive* Systeme als Möglichkeit angebracht, wobei hier drei Gründe aufgeführt werden können:

1. *Geht auf Komponentenaustausch zurück.* Ketfi *et al.* führen an, dass das Entfernen der Schnittstelle inklusive ihrer Implementierung Vorteile hinsichtlich der Performance bringt. Nach Meinung des Autors muss beim Entfernen einer Implementierung auch zwangsweise die neue Komponente mit der fehlenden Implementierung erneut deployt werden. Daher ist eine Komponente mit einer anderen oder fehlenden Implementation eine andere Komponente. Dieser Vorgang wird durch den Mechanismus des Austauschs einer Komponente abgedeckt.

   Es gibt zwar Komponentenmodelle, wie beispielsweise Hadas [Ben-Shaul *et al.* 2001b], die das Verändern der Struktur (und in dem Fall auch des Verhaltens) einer Komponente zur Laufzeit durch Benutzung von Reflection ermöglichen, wodurch ein Austauschen/Redeployen der Komponente nicht notwendig wird (und dadurch das obige Argument nichtig) — ein Beispiel hierzu ist in Abbildung 4.9 auf der nächsten Seite zu sehen. Der Autor vertritt jedoch die Auffassung, dass frei konfigurierbare Schnittstellen nach diesem Prinzip zwar Flexibilität und leichte Adaptierbarkeit bieten, jedoch in der Entwicklung eher von Nachteil sind. Frei konfigurierbare Schnittstellen senken die Wartungsfreundlichkeit und steigern die Komplexität, da Methodendeklarationen und zulässige Parameter sorgfältig dokumentiert werden müssen und unter anderem Fehler nicht im Entwicklungsprozess entdeckt werden können. Es gibt keine Schnittstellenbeschreibungen wie beispielsweise die IDL, die auf Verträglichkeit bei der Integration von Komponenten geprüft werden können (vgl. [Dittert 2004]).

   Außerdem ist es im Rahmen des Change Managements sehr wichtig, die geänderten Komponenten einzeln identifizieren zu können. Wenn die Implementierung einer Komponente geändert wird, besitzt diese eine andere Identität. Eine solche Verfahrensweise ermöglicht auch, die neue Komponente zu revidieren und die alte Komponente wieder einzufügen, falls die Implementierung der neuen Komponente fehlerhaft ist.

2. *Unvereinbar mit* adaptiven *Systemen.* Wie soll das System wäh-

```
    ...
(1) if (systemLoad() > threshold) {
(2)   selfObject.lock();
(3)   selfObject.invoke("removeMethod", p("buy"));
(4)   selfObject.invoke("addDataItem", p("message",
(5)                "Our shop is temporarily closed"));
(6)   selfObject.invoke("addMethod", p("buy",
(7)                new PrintMessage(selfObject)));
(8)   selfObject.unlock();
(9) }
    ...
```

Abbildung 4.9.: Adaption in Hadas [Ben-Shaul *et al.* 2001b]. Dadurch kann sowohl das Verhalten (hier Methode buy) als auch die Struktur verändert werden (Entfernen der Methode buy, Zeile 3, ohne nachfolgendes Hinzufügen, wie in Zeile 6 dennoch geschehen). In der vorliegenden Arbeit wird diese Form der Adaption nicht betrachtet.

rend des Betriebs selbst bemerken, dass eine Schnittstelle nicht benötigt wird, und sich infolgedessen verschlanken? Es gibt zwar Konzepte wie das in [Amberg *et al.* 2003] vorgestellte »Software-EKG«, das Statistiken zur Benutzung von Methoden einer Komponente anfertigen und dadurch ein Bild für die Notwendigkeit und Optimierung von Schnittstellen offenbaren kann. Im Hinblick auf adaptive Systeme, die sich während der Laufzeit *selbst* an eine Umgebung anpassen und sich optimieren, wird das Verwerfen von Schnittstellen als unsinnig betrachtet.

3. *Trennung Schnittstelle und Implementierung.* Was hat das System davon, wenn es seine Schnittstelle entfernt? Die als Ballast angesehene Implementierung kann auch aus dem deployten Code entfernt werden, ohne dass dies Auswirkungen auf die Existenz einer Schnittstelle haben muss. Falls die selten benutzte Schnittstelle doch angesprochen wird, kann der benötigte Code nachgeladen werden.

Diese Betrachtung hat als Schlussfolgerung, dass es hinsichtlich adaptiver Systeme nur die nachfolgend aufgezeigte Möglichkeit von Schnittstellenänderung gibt. (Für den Menschen als Adaptionssubjekt behält die Schnittstellenänderung dahingegen Geltung.)

Schnittstellenänderung infolge anderer Adaptionen. Die andere Möglichkeit von Schnittstellenänderung tritt implizit auf, wenn beispielsweise beim Austausch einer Komponente die neue Komponente eine andere Schnittstelle besitzt. In dem Fall ändert sich die Schnittstelle infolge einer anderen Adaptionsoperation. Hier sind zwei Fälle unterscheidbar:

1. *Die Schnittstelle ändert sich, weil die Funktionalität der Komponente geändert wurde.* Ein Beispiel hierfür ist in [Bialek und Jul

2004] zu finden: Gegeben ist eine Chat-Applikation mit der Methode send(from, to, txt), die eine Text-Nachricht von einem Teilnehmer an den anderen versendet. Aufgrund neuer Anforderungen an die Anwendung soll es möglich sein, zusätzlich auch den Versand von Grafiken zu ermöglichen — die Schnittstelle ändert sich damit zu send(from, to, txt, *grph*).

Dies entspricht zwar einer möglichen Adaption der Software, aber diese Art Schnittstellenänderung ist unvereinbar mit dem Begriff »(selbst-) adaptiver Software«. Die neue Anforderung (Grafikversand) kann nicht durch die Software erkannt werden, sondern nur durch intelligente Formen, wie den Menschen. Das Einbringen einer Chat-Komponente, die diese neue Anforderung unterstützt, entspricht zwar einer Adaption (im Rahmen der Evolution) der Software. Jedoch kann diese Operation nicht mit *adaptiver* Software in Verbindung gebracht werden, da die Software sich nicht selbst anpasst.

2. *Die neue Komponente ist nicht zu 100 % Schnittstellen-kompatibel.* Dieses »traditionell« bei dem Zusammenbau von Anwendungen aus COTS[12]-Komponenten auftretende Problem kann auch verhäuft bei adaptiven Systemen auftreten, da diese viele alternative Komponenten benutzen, die die Redundanz für einen Dienst anbieten. COTS-Komponenten haben oft die Eigenschaft, dass sie zwar die gleiche Funktionalität anbieten, aber die *Ebenen der Interoperabilität* unterschiedlich sind. Vallecillo *et al.* [2000] definieren für die Interoperabilität

   - die *Signaturebene*, das sind Namen und Signaturen[13] von Operationen,
   - die *Verhaltensebene*, befasst sich mit den Protokollen, d. h. mit der relativen Ordnung der Nachrichten, die ausgetauscht werden, und
   - die *semantische Ebene*, befasst sich mit der Bedeutung, die hinter einer Operation steht.

   Die Signaturebene ist derzeit beispielsweise mit der IDL gut verstanden, Forschung für die Verhaltensebene und für die semantische Ebene im Besonderen ist noch notwendig [Canal 2004]. Im Folgenden soll ein Beispiel gegeben werden, das die Platzierung eines Adapters zwischen zwei Komponenten erforderlich macht, weil deren Verhaltensebenen inkompatibel zueinander sind (das Beispiel entstammt vorgenannter Quelle).

   Das Beispiel für eine Schnittstellenänderung, die sich durch den Austausch einer Komponente ergibt, besteht aus einer `Printer`-

[12] Commercial Off-The-Shelf
[13] Reihenfolge und Typ von Parametern einer Methode

und einer `PrinterClient`-Komponente. Die `PrinterClient`-Komponente geht davon aus, dass sie für das Drucken zuerst die Anzahl der Kopien festlegen muss (Operation setCopies(num)) und dann den Druckauftrag über eine zweite Operation abschicken kann (print(doc)). Das Interaktionsmuster lautet hierfür in Prozessalgebra `setCopies!(num).print!(doc).0`. Aufgrund von irgendwelchen Ursachen wird nun eine andere `Printer`-Komponente benutzt. Diese bringt eine andere Verhaltensebene mit sich, indem sie die Anzahl der Kopien und den Druckauftrag in einer Operation zusammenfasst. Das Interaktionsmuster hat sich damit zu `printc?(doc,num).Printer` umgeändert. Beide Komponenten erbringen zwar die gleiche Funktionalität, jedoch muss für ein Operieren der `PrinterClient`-Komponente mit der zweiten `Printer`-Komponente ein Adapter dazwischengeschaltet werden, der die unterschiedlichen Interaktionsmuster aneinander anpasst.

Derzeitige Forschung befasst sich damit, diesen Adapter dynamisch zu generieren. Einen Einblick in den State-of-the-Art gibt [Canal 2004].

Beide Fälle umfassen Schnittstellenänderung im Rahmen einer Adaption. Der erste Fall, bei dem die Schnittstelle aufgrund von Funktionserweiterungen eine Änderung erfährt, ist jedoch nicht zu »adaptiven Software-Systemen« zu zählen (ggf. »dynamisch adaptierbaren Systemen«, wenn die Änderungen zur Laufzeit möglich sind), da diese Änderung nicht durch die Software vorgenommen wird. Derartige neue Anforderungen können nicht durch die Software erkannt werden. Stattdessen nimmt der Mensch aufgrund geänderter Anforderungen eine Änderung an der Funktionalität der Software vor und adaptiert das System. Der zweite Fall sieht Schnittstelleninkompatibilität als Auslöser. Diese kann allgemein beim Zusammenbau einer Anwendung aus COTS-Komponenten auftreten, aber auch bei der Auswahl alternativer Komponenten, die nicht alle die gleiche Schnittstelle besitzen. Für diesen Kasus muss ein zwischen den Komponenten platzierter Adapter für Interoperabilität sorgen.

Noch eine Anmerkung zu dem Adapter. In der Software-Architektur taucht dieser nicht als separate, neue Komponente zwischen der `PrinterClient`- und `Printer`-Komponente auf. Stattdessen erfolgt eine Abstraktion, d. h. der Adapter wird durch den Konnektor realisiert, da ein Konnektor für die korrekte Interaktion zwischen Komponenten verantwortlich ist (vgl. [Mehta *et al.* 2000]).

Problem Adaptivität und Adaptierbarkeit. Bei der Erörterung zur Schnittstellenanpassung wird das Problem besonders deutlich, das durch die gleichzeitige Betrachtung von Adaptierbarkeit (was kann wie geändert werden) und Adaptivität (Software ist Auslöser und Durchfüh-

rer von Änderungen) einer Software-Architektur entsteht. Für die Findung von Adaptierbarkeit reicht das Rezept, die Beschaffenheit des Adaptionsgegenstandes (hier Software-Architektur, bestehend aus Komponenten und sie verbindenden Konnektoren, bei feingranularerer Betrachtung auch Schnittstellen und Parameter der Komponenten) zu analysieren und daraufhin diese auf Änderbarkeit zu untersuchen — wobei das Ändern aus den Grundoperationen Hinzufügen, Entfernen und Modifizieren besteht. Soll jedoch auch Adaptivität betrachtet werden, was nichts anderes heißt, als dass die Software die Notwendigkeit für Änderungen erkennt und diese vornimmt, dann müssen die Aussagen zur Adaptierbarkeit differenzierter betrachtet werden. Denn nicht alle Änderungen *an* einer Software können *durch* die Software hinsichtlich Notwendigkeit erkannt sowie durchgeführt werden. Sollte die KI[14] fortschreiten, so muss diese Bemerkung relativiert werden (Stichwort: Wer steuert und verändert »die Matrix«?).

Und eine der Motivationen, die eigentlich beseitigt werden soll, wird durch die Entwicklung von adaptiven Systemen noch besonders verstärkt. Autonomic Computing will der Komplexität der Software begegnen (vgl. [Horn 2001]), selbst-adaptive Software wird ebenso als Mittel dafür gesehen [Laddaga 2001]. Durch die Einführung von Adaptierbarkeit in der Software, beispielsweise in Form flexibler Schnittstellen, steigt jedoch die Komplexität in der Softwareentwicklung.

Parameteradaption

Viele entwickelte Komponenten bieten Parameter an, über diese das starre Verhalten einer Komponente angepasst werden kann. Dadurch ist es möglich, die Software an eine gewünschte Umgebung anzupassen und zu optimieren, indem die in den Komponenten enthaltenen Verarbeitungsalgorithmen parametrisiert werden.

Streng genommen kann die Anpassung der Parameter einer Komponente nicht zur Adaption einer Software-Architektur gezählt werden, da eine Software-Architektur nur die Struktur, sprich Komponenten und Schnittstellen sowie das Verbinden dieser betrachtet (vgl. Kap. 2.1). Eine Betrachtung der Parameter von Komponenten gehört bei der grobgranularen Sicht, wie sie die Software-Architektur durch die Abstraktion vornimmt, nicht dazu. In dieser Arbeit wird jedoch die Parametrisierung von Komponenten mitbetrachtet, denn durch Verändern von Parametern kann die Funktionalität einer Komponente auf einfache Weise verändert werden, um Adaptivität in einer Software zu erreichen. Reicht der durch die Parametrisierung erfolgte Wirkungsbereich nicht mehr aus, kann durch Austauschen der Komponente weitergehende Anpassung erfolgen.

[14] Künstliche Intelligenz

4.1.3. Adaption der Systemarchitektur

Der vorhergehende Abschnitt erörterte Möglichkeiten in der Anpassung der Anwendungsarchitektur. Diese besteht aus Komponenten, wodurch eine Verteilung der Funktionalität vorgenommen wird. Der vorliegende Abschnitt befasst sich auch mit Verteilung, hier jedoch in Form der Verteilung der Last, indem die Verarbeitungsoperationen (die Komponenten) auf verschiedene Ausführungs-/Hardwareknoten verteilt werden. Die Ausführungsorte können von kleinen Hardwareelementen wie Prozessoren bis zu großen Maschinen, die weit-verteilt über das Internet verstreut sind, reichen. Eine derartige Verteilung geschieht aus Sicht der Systemarchitektur.

Die Anpassung der Systemarchitektur kann hinsichtlich der nachfolgenden beschriebenen Arten erfolgen: Es wird das Replizieren von Komponenten, das Migrieren von Komponenten sowie das Neuzuweisen/Binden durch Austauschen einer Komponente unterschieden.

Replikation von Komponenten

Durch die Replikation (Klonen) von Komponenten steht der Anwendung die Komponente mehrfach zur Verfügung. Die Replik kann sowohl hinsichtlich der Daten (Redundanz eines Speichers im Netzwerk) als auch hinsichtlich der Funktionalität erfolgen. In beiden Fällen wird durch die Replikation eine Lastverteilung sowie Redundanz als Vorsorge für Ausfälle erreicht.

Wird die Software-Architektur als Graph betrachtet, dann resultiert das Replizieren einer Komponente im Hinzufügen eines Knotens im Graphen. Hier muss eine klare Abgrenzung zum Hinzufügen einer Komponente in der Anwendungsarchitektur (vgl. Seite 96) vorgenommen werden. Das Replizieren einer Komponente fügt eine neue *Instanz* gleichen Typs/Implementierung dieser zu, wohingegen das Hinzufügen einer Komponente in der Anwendungsarchitektur eine Komponente anderer Funktionalität (Typ) hinzufügt. Bei der Replikation hat die neue Instanz hinsichtlich der Verarbeitung und der Speicherung die gleichen Eigenschaften wie ihr Master, von dem sie dupliziert wurden.

Für die Replikation existiert eine entsprechende Gegenoperation, das Löschen des Replikates.

Migration von Komponenten

Die Migration von Komponenten ermöglicht die dynamische Änderung des Ausführungsortes von Komponenten zur Laufzeit (siehe Abbildung 4.10 auf der nächsten Seite). Dieser Aspekt ist in der Literatur auch unter den Begriffen »mobiles Code-System« und »(mobile) Agenten« bekannt [Fuggetta *et al.* 1998]. Die sonst in klassischen verteilten Systemen angestrebte Ortstransparenz [Coulouris *et al.* 2002] wird in mobilen Code-Systemen aufgelöst, um dem Programmierer die explizi-

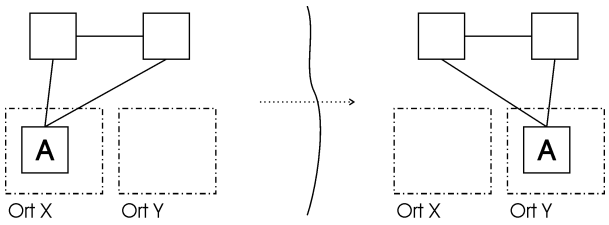

Abbildung 4.10.: Migration einer Komponente von X nach Y

te Kenntnis des Ausführungsortes verfügbar zu machen und ihm die Möglichkeit zu geben, die Migration selbst zu steuern.

Die Migration einer Komponente bringt viele Vorteile mit sich [Fuggetta *et al.* 1998], [Lange und Oshima 1999]; genannt werden sollen hier nur die wichtigsten drei:

- *Lastverteilung.* Die Zuordnung von Komponenten zu Verarbeitungsknoten, die ihre Ressourcen bereitstellen, muss beim initialen Deployen nicht immer optimal gewählt sein, zumal sich die Anforderungen im laufenden Betrieb noch ändern können. Die Migration von Komponenten zu anderen Verarbeitungsknoten erlaubt die dynamische Lastverteilung.

- *Reduzierung der Netzwerkauslastung.* Neben einer Optimierung der Last auf den Verarbeitungsknoten kann die Migration von Komponenten auch dazu genutzt werden, die Last im Netzwerk zu reduzieren. Dies wird erreicht, indem zwei Komponenten, die viele Daten miteinander austauschen müssen, auf einen gleichen Verarbeitungsknoten gebracht werden. Somit bleibt der Datenaustausch lokal und belastet nicht das Netzwerk.

- *Abgekoppeltes Arbeiten.* Vor allem die Netzwerkverbindung von mobilen Geräten ist meist kostenintensiv und unzuverlässig. Die Verlagerung von Verarbeitungsoperationen von dem mobilen Gerät in das Netzwerk in Form von mobilen Agenten erlaubt es dem mobilen Gerät, die Verbindung zu trennen und später wieder aufzunehmen. Während das Endgerät offline ist, führt der Agent autonom und asynchron seine Operationen aus und übergibt die Ergebnisse dem mobilen Endgerät, sobald dieses wieder Kontakt mit ihm aufnimmt.

Eine wichtige Voraussetzung für mobile Code-Systeme ist die Verfügbarkeit einer Ausführungsumgebung auf den Knoten, zu denen eine Migration vollführt werden soll. Unterschieden kann zwischen schwacher und starker Mobilität [Fuggetta *et al.* 1998]. Bei schwacher Mobilität wandert nur der Objektcode und Zustand, bei starker Mobilität zusätzlich der volle Laufzeitkontext inklusive Stack und Zustand.

Die Migration hat dabei keine Auswirkungen auf die Anwendungsarchitektur. Jedoch müssen die Verbindungen zwischen den Komponenten angepasst werden. Eine Lösungsmöglichkeit ist die Verwendung von Proxies, die, an dem alten Ort positioniert, die Kommunikation zu dem neuen Ort weiterleiten. Andernfalls müssen die Referenzen aller Komponenten angepasst werden, die noch auf den alten Ort der migrierten Komponente zeigen (vgl. [Schill 2004]).

Die Migration von Komponenten, d. h. die Neuzuordnung von Komponenten zu anderen Hardware-Ressourcen wird als *Rekonfiguration der Systemarchitektur* bezeichnet [Medvidovic 1996] (nicht zu verwechseln mit der Rekonfiguration in der Anwendungsarchitektur).

Neuzuweisung/Binden (durch Austausch des Kommunikationspartners)

Der dritte Adaptionsmechanismus ist die Neuzuweisung des Dienstnehmers an einen anderen Dienstgeber. Dieser Mechanismus ist ähnlich dem Austausch einer Komponente in der Anwendungsarchitektur. Im Gegensatz zu dieser Adaptionsoperation erfolgt bei der Neuzuweisung allerdings kein Austausch der Implementierung, sondern der Kommunikationspartner wird durch einen mit einer gleichen Implementierung ersetzt — meist in Verbindung mit einer zuvor erfolgten Replikation des Kommunikationspartners.

Ein einfaches Beispiel für diesen Adaptionsmechanismus ist die Neuzuweisung eines Clients an einen anderen Server, weil dieser eine bessere Performance liefert (da der Server weniger ausgelastet ist) oder die verfügbare Bandbreite zu diesem Server eine bessere ist.

Im Grunde genommen wird hier nur die Verbindung (Konnektor) des Dienstnehmers (Client) zu einem anderen Dienstgeber (Anbieter/Server) geändert (vgl. Abbildung 4.11 auf der nächsten Seite), weswegen dieser Mechanismus auch (1) als Anpassung des Konnektors und (2) als »Rekonfiguration der Anwendungsarchitektur« aufgefasst werden könnte. In der vorliegenden Klassifikation werden die Beziehungen zwischen Komponenten aus Sicht des Dienstgeber betrachtet. Konnektoren finden dabei keine explizite Berücksichtigung.[15] Deshalb ist die Ersetzung von A durch A' ein Austausch. Der Vorgang ist *keine* Rekonfiguration, da hier keine Rekonfiguration der Funktionalität erfolgt, sondern nur die Verbindung zu einer anderen Komponente gleichen Typs und gleicher Funktionalität, d. h. nur zu einer anderen Instanz, erfolgt (vgl. den Hinweis zur Unterscheidung von Typen und Instanzen in der C&C-Sicht ab Seite 91).

[15] Der Grund dafür ist folgender: Die Migration einer Komponente könnte als Parameteränderung des Konnektors aufgefasst werden, da dieser eine andere Referenz bekommen und ggf. den Konnektortyp von lokal auf remote abändern muss. Damit wäre diese Konnektorparametrisierung allerdings nicht abgrenzbar zum Neubinden eines Clients an einen anderen Dienstgeber, da hier auch die Referenz es Konnektors abgeändert werden muss.

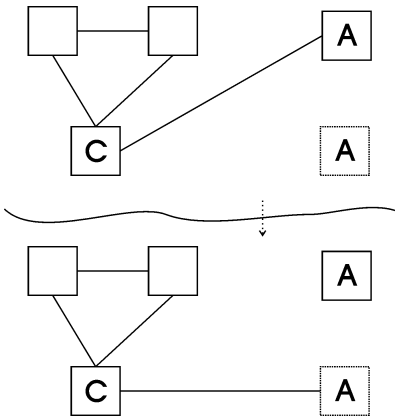

Abbildung 4.11.: Verteilte Anwendung, bei der der Austausch einer Komponente in der System-Architektur nur im Binden des Clients (C) zu einer anderen Instanz des Anbieters (A) resultiert

4.1.4. Anpassung der Adaptionsarchitektur (A.-Infrastruktur)

Die zuvor aufgeführten Adaptionsmöglichkeiten in Software-Architekturen behandeln nur Adaption der eigentlichen Software selbst. Wird der Apparat (Adaptionsinfrastruktur) betrachtet, der die Adaption unterstützt, so ist auch hier eine Adaption des Apparates denkbar — unter der Annahme, dass die Adaptionsinfrastruktur (z. B. Monitoring) getrennt von der Anwendung ist.

Nach allgemeiner Kontrolltheorie benötigt adaptive Software Mechanismen, sich selbst zu beobachten, um aus dem beobachteten Verhalten Änderungsmaßnahmen ableiten zu können. Diese Funktionaltität des Monitorings wird durch Sensoren realisiert, die die Komponenten der Software beobachten. An dieser Stelle werden zwei Aspekte deutlich, die Adaptivität auch der Sensoren erfordern:

1. Wenn eine Sensorkomponente nicht genügend Informationen über den Beobachtungsgegenstand liefert, kann die adaptive Software den Sensore nachregeln oder neue Sensoren einrichten und nicht benötigte Sensoren entfernen, um bessere Ergebnisse in der Beobachtung zu erreichen.

2. Gegeben durch die Dynamik der Software, die beispielsweise in dem Hinzufügen oder Entfernen von Komponenten resultiert, darf die Adaptionsinfrastruktur nicht starr konfiguriert sein. Das bedeutet, dass neuen Komponenten auch Sensoren zugeteilt werden müssen, damit die neuen Komponenten beobachtet werden können. Für das Entfernen von Komponenten müssen die Sensoren entsprechend abgezogen werden. Das heißt, Änderungen des Beobachtungsgegenstandes (Softwarekomponenten der Anwendung) müssen ebenso Änderungen im Beobachtungsapparat (Sensorik) zur Folge haben.

Auch für andere Bestandteile der Adaptionsinfrastruktur sind Adaptionen denkbar.

4.1.5. Zusammenfassung bezüglich Adaptivität / Adaptierbarkeit

Tabelle 4.1 fasst die Adaptionsmechanismen bezüglich (Möglichkeit-) Adaptierbarkeit (allg. was ist adaptierbar, meist durch Mensch; nicht: adaptierbare Software) und Adaptivität (Software erkennt Anpassungsbedarf und nimmt Anpassung vor) zusammen. Diese Tabelle berücksichtigt den State-of-the-Art. Mit zunehmender KI in Software können beide Spalten verschmelzen, d. h. Software wird in der Lage sein, Anforderungen zu erkennen und die Software um neue Komponenten zu erweitern, so wie es der Mensch bereits kann.

	Adaptionssubjekt	
Adaptionsmechanismus	*Mensch bzw. allg. »was ist adaptierbar«*	*Software aka adaptive Software*
Komp. hinzufügen	ja	wenn Komp. vorher modelliert und Regel für Hinzufügen existiert
Komp. entfernen	ja	wenn Regel für Entfernen vorher definiert
Komp. austauschen	Mensch kann »intelligentere« Alternativen generieren	es müssen lediglich Alternativen vorhanden oder generierbar sein
Schnittstellenänderung ▷ als Aktion ▷ als Folge	 ▷ ja ▷ ja	 ▷ nein ▷ ja, Adapter für Interoperabilität generieren
Replikation	theoretisch ja	praktisch nur durch Software selbst vorgenommen
Migration	ja, aber selten	ja
Parameteradaption	ja	ja
Neuzuweisen/Binden	nein	ja

Tabelle 4.1.: Adaptierbarkeit (Mensch) und Adaptivität (Software passt sich selbst an) in einer Software-Architektur

Der Grund für ein Nein in der letzten Zeile der Tabelle ist folgender: Wenn ein Benutzer über das URL-Adresseingabefeld des Browsers einen anderen Webserver ansteuert, ist dies keine Adaption. Wenn für eine Lastverteilung in einer Webserver-Farm der Nutzer selbst einen wenig ausgelasteten Server auswählen muss oder der Administrator dies im laufenden Betrieb ändert, ist dies nur für den Administrator-Fall *theoretisch* eine Adaption, praktisch kaum. Wenn das System

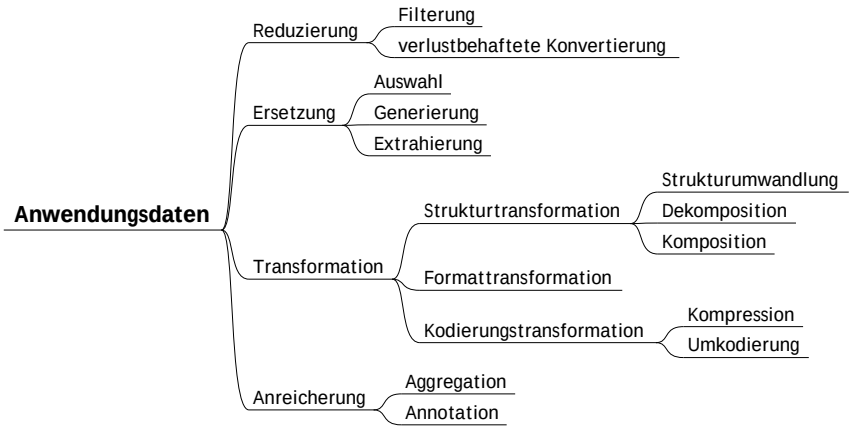

Abbildung 4.12.: Mechanismen zur Adaption von Anwendungsdaten

selbst die Änderung/Zuweisung im laufenden Betrieb vornimmt (nicht zum Zeitpunkt der Anfrage), dann entspricht dies einer Adaption.

4.2. Adaption der Anwendungsdaten

Für die Anpassung von Anwendungsdaten existieren viele Möglichkeiten. Im Umfeld von Pervasive Computing ist vor allem das Umwandeln einer geräteunabhängigen Beschreibungssprache (z. B. DDL, SMIL und UIML) in die Seitenbeschreibungssprachen HTML, WML und cHTML bekannt.[16] Auch die Umwandlung der Bilder bezüglich Format — beispielsweise können WAP[17]-Geräte nur das WBMP-Format anzeigen — und Skalierung für kleinere Displays ist für die Anpassung der Anwendung an die heterogene Client-Umgebung bedeutend. Doch es gibt noch mehr Adaptionsmechanismen für die Anpassung an eine heterogene Systemumgebung.

In der vorliegenden Arbeit liegt der Fokus auf der Adaption der Software-Architektur und nicht der Daten. Daher wird auf eine eigenständige Ausarbeitung verzichtet und stattdessen auf die umfangreiche Arbeit von Springer [2004] zum Thema Adaption von Daten verwiesen. Die Ergebnisse der vorgenannten Arbeit werden nachfolgend in gekürzter Form zusammengetragen. In Abbildung 4.12 sind die Adaptionsmechanismen für Anwendungsdaten in einer Übersicht zusammengefasst.

[16] Transcoding-Mechanismen wurden bereits von der DaimlerChrysler-Forschung untersucht [Vögler *et al.* 2004].
[17] Wireless Application Protocol

4.2.1. Reduzierung

Daten können durch verlustbehaftete Konvertierung oder durch Filterung reduziert werden.

Filterung

Unter Filterung wird die Auswahl von Daten oder Teildaten und das Verwerfen der ausgewählten Daten verstanden. Eine Filterung kann auf Basis von Informationen über die Struktur, den Typ, das Format, die Semantik der Daten sowie externer Informationen erfolgen.

Ein Beispiel für die Filterung auf Basis der Struktur ist das Verwerfen von P- und B-Frames eines MPEG[18]-Stromes. Filterung kann auch bedeuten, dass aufgrund des Datentyps eine Datei im Ganzen wegfällt, beispielsweise auf einer Web-Seite oder in einer E-Mail, weil der Datentyp auf dem Endgerät nicht repräsentiert werden kann. Oder Textdokumente werden auf Basis von Schlagwörtern oder des Themas gefiltert (Semantik).

Verlustbehaftete Konvertierung

Im Gegensatz zur Filterung werden bei einer verlustbehafteten Konvertierung Daten nicht ausgewählt oder verworfen, sondern bezüglich des Informationsgehaltes reduziert.

Es sind vielfältige Konvertierungsmechanismen denkbar, da sie vom Typ der Daten abhängen. Bilder können beispielsweise skaliert oder in der Farbtiefe reduziert werden. Videodaten können zusätzlich in der Framerate angepasst werden, bei Audiodaten sind Konvertierungen ihrer Samplefrequenz und der Amplitude möglich.

Die Kompression von Daten, sofern sie verlustbehaftet ist (beispielsweise Quellenkodierung bei JPEG[19]), wird ebenso in diese Klasse eingeordnet (im Unterschied zur verlustfreien Kompression, die dem Mechanismus Kodierungstransformation zugeordnet wird).

4.2.2. Ersetzung

Bei dem Adaptionsmechanismus Ersetzung werden die Originaldaten gegen alternative Daten ausgetauscht. Je nachdem, ob die alternativen Daten explizit zur Verfügung stehen, implizit aus verfügbaren Daten gewonnen oder aus den Originaldaten generiert werden, werden die Mechanismen in die drei Klassen Auswahl, Extrahierung und Generierung eingeteilt.

[18] Moving Picture Experts Group
[19] Joint Photographic Experts Group

Auswahl

In der Klasse Auswahl liegt die alternative Repräsentation explizit vor. Das originale Datenobjekt wird durch das alternativ vorliegende Datenobjekt ersetzt.

Ein bekanntes Beispiel ist die Ersetzung eines Bildes durch einen beschreibenden Text in HTML, realisiert durch das `ALT`-Element innerhalb des `IMG`-Tags.

Extrahierung

Durch die Extrahierung werden implizit in den Originaldaten enthaltene Informationen zur Ersetzung verwendet.

Ein Bild kann beispielsweise bei der Darstellung durch einen Rahmen mit den Maßen des Bildes ersetzt werden, Datenobjekte allgemein durch ihren Dateinamen.

Generierung

Mechanismen dieser Klasse generieren aus den Informationen des originalen Datenobjektes alternative Daten. Im Gegensatz zur verlustbehafteten Konvertierung besitzt das erzeugte Datenobjekt meist einen anderen Typ als die Originaldaten.

Aus Audiodaten kann beispielsweise eine Textrepräsentation erzeugt werden (Speech-to-Text und umgekehrt). Ein weiteres Beispiel ist die Erzeugung einer Vektorgrafik aus einem Bild mit Hilfe von Konturerkennungsverfahren.

4.2.3. Transformation

Mechanismen zur Transformation verändern die interne Repräsentation von Datenobjekten, verwerfen jedoch keine Originaldaten. Je nachdem, ob die Struktur, das Format oder die Kodierung transformiert wird, wird die Transformationen in die Mechanismen Struktur-, Format- und Kodierungstransformation eingeteilt. Gemeinsam haben alle Mechanismen, dass es einen inversen Mechanismus gibt, der die Originaldaten wiederherstellen kann.

Strukturtransformation

Bei der Strukturtransformation wird die Struktur zusammengesetzter Daten in unterschiedliche Darstellungsformen überführt. Eine Verarbeitung der Datenobjekte wird dabei nicht vorgenommen. Stattdessen wird nur die Verknüpfung zwischen den Datenobjekten verändert. Strukturtransformationen können eingesetzt werden, um eine effizientere Verarbeitung zu erreichen, nicht unterstützte Datenstrukturen in

unterstützte Strukturen zu überführen oder durch eine Restrukturierung des Datenobjektes eine angepasste Übertragung und Darstellung zu ermöglichen.

Die Strukturtransformation kann zudem in die folgenden drei Unterklassen unterteilt werden.

Strukturumwandlung. Die Strukturumwandlung eines Datenobjektes resultiert in einer von der Ausgangsstruktur verschiedenen Darstellung.

Ein Beispiel für die Strukturumwandlung ist die Überführung eines XML[20]-Dokumentes in eine Baumstruktur entsprechend des DOM[21]. Ein nur implizit strukturierter Text kann in eine explizite Struktur überführt werden. Auf Basis der expliziten Struktur kann dann beispielsweise ein Inhaltsverzeichnis aufgebaut und für Kapitel, Abschnitte usw. eine effiziente Präsentation und Navigation durch den Text erreicht werden.

Dekomposition. Durch die Dekomposition strukturierter Daten werden diese in separate Datenobjekte zerlegt. Die Sequenz von Datenobjekten kann nach der Dekomposition unabhängig und durch unterschiedliche Mechanismen verarbeitet bzw. adaptiert werden.

Ein Beispiel für die Dekomposition ist das Zerlegen eines Multimediastroms in die Bestandteile Audio und Video, um diese getrennt zu verarbeiten und anzupassen. Eine E-Mail-Nachricht kann in den Nachrichtenkopf, den Nachrichtentext und die Anhänge zerlegt werden. Auch die Fragmentierung (auch Paginierung genannt) einer großen Seite in mehrere kleine Seiten, um eine angemessene Darstellung auf kleinen Displays oder ein schnelleres Laden zu unterstützen, gehört zu den Mechanismen der Dekomposition.

Komposition. Das Gegenstück zur Dekomposition ist die Komposition. Separate Datenobjekte werden zu einer Datenstruktur zusammengefasst, um diese als Ganzes zu verarbeiten und anzupassen.

Nach einer separaten Adaption und Übertragung der Teildaten kann auf dem Endgerät aus den Teildaten wieder ein zusammengefasstes Datenobjekt erzeugt werden.

Formattransformation

Bei der Formattransformation wird das Format eines Datenobjektes in ein anderes Format überführt. Der Typ der Daten wird dabei nicht umgewandelt. Notwendig wird dieser Adaptionsmechanismus bei inkompatiblen Verarbeitungsoperationen und wenn das Endgerät bestimmte Formate nicht unterstützt.

[20] eXtensible Markup Language
[21] Document Object Model

Bei der Bedienung von WAP-Geräten müssen beispielsweise die Bilder in das WBMP-Format umgewandelt werden. Ein weiteres Beispiel ist die Umwandlung von PDF[22]-Dokumenten in RTF[23] oder HTML-Dokumente.

Im Falle der Transformation eines ausdrucksstärkeren Formates in ein weniger ausdrucksstarkes Format geht dieser Adaptionsmechanismus mit dem Verlust von Informationen einher. Daher stellt er einen Grenzfall in Bezug auf die verlustbehaftete Konvertierung dar (vgl. Kapitel 4.2.1 auf Seite 111). Dennoch werden auch diese Mechanismen in die Klasse Formattransformation eingeordnet, da das primäre Ziel die Umwandlung in ein alternatives Format und nicht die Reduzierung des Informationsgehaltes ist.

Ein Informationsverlust ist bei der Formattransformation jedoch nicht als generell anzusehen. Beispiele dafür sind die Umwandlung eines JPEG-Bildes in ein BMP[24]-Bild oder die Umwandlung von MP3-Daten in das WAV[25]-Format. Die Informationsmenge bleibt hier gleich, nur das Datenformat wird geändert und die Datenmenge ist nach der Konvertierung gar größer als vorher.

Kodierungstransformation

Der Begriff Kodierungstransformation umfasst Mechanismen, die die interne Repräsentation von Anwendungsdaten verändern, ohne den Informationsgehalt zu reduzieren. Im Gegensatz zur zuvor aufgeführten Formatkonvertierung (beispielsweise die Umwandlung von JPEG nach BMP) bleibt das Format unverändert.

Bei der Kodierungstransformation können die folgenden zwei Unterklassen unterschieden werden.

Kompression. Ziel der Kompression ist die Verringerung des Datenvolumens, wobei die Reduzierung verlustfrei ist (Entropiekodierung).

Beispiele für verlustfreie Kompressionsverfahren sind die Huffman- und die Lauflängenkodierung.

Umkodierung. Mechanismen der Umkodierung ermöglichen es, Inkompatibilitäten zwischen Verarbeitungsoperationen auszugleichen oder Daten in eine vom Endgerät unterstützte Kodierung zu überführen. Dabei wird die interne Repräsentation der Daten verändert.

Als Beispiele können die Umkodierung von ASCII[26]-Text nach Unicode sowie die Transformation von Bilddaten zwischen unterschiedlichen Farbräumen erwähnt werden.

[22] Portable Document Format
[23] Rich Text Format
[24] Bitmap
[25] Windows Wave
[26] American Standard Code for Information Interchange

4.2.4. Anreicherung

Mit dem Begriff Anreicherung werden Mechanismen bezeichnet, die den Informationsgehalt eines Datenobjektes um zusätzliche Informationen erhöhen. Dabei können die Aggregation und die Annotation unterschieden werden.

Aggregation

Aggregation bezeichnet die Anreicherung eines Datenobjektes mit Informationen aus weiteren separaten Datenobjekten. Dazu werden bestimmte Informationen jedes Datenobjektes ausgewählt und in das Zieldatenobjekt integriert.

Ein Beispiel ist die Aggregation von Ergebnissen mehrerer Suchmaschinen zu einer Seite, die die jeweils besten Suchergebnisse zusammenfasst.

Annotation

Bei der Annotation wird ein Datenobjekt mit zusätzlichen Informationen angereichert.

Ein Beispiel ist die Hervorhebung von bestimmten Buchstaben oder Wörtern, Telefonnummern, Adressen u. ä. in Web-Seiten durch Anwendung von regulären Ausdrücken oder Heuristiken.

4.3. Adaption der Kommunikation

In der vorliegenden Arbeit liegt der Fokus auf der Adaption der Software-Architektur und nicht der Kommunikation. Daher wird auf eine eigenständige Ausarbeitung verzichtet und stattdessen auf die umfangreiche Arbeit von Springer [2004] zum Thema Adaption der Kommunikation in einem verteilten System verwiesen. Die Ergebnisse der vorgenannten Arbeit werden nachfolgend in gekürzter Form zusammengetragen.

Mechanismen zur Adaption der Kommunikation passen zum einen die Übertragungsprotokolle an, zum anderen wird die Verfügbarkeit von Datenobjekten und Zwischenergebnissen durch Mechanismen zur Zwischenspeicherung und zur Steuerung des Zugriffs auf Daten angepasst (siehe Abbildung 4.13).

4.3.1. Übertragung

Auf den unteren Schichten von Protokollen (Schichten 1 bis 4 nach OSI[27]) werden grundlegende Dienste zur Datenübertragung realisiert.

[27] Open Systems Interconnection

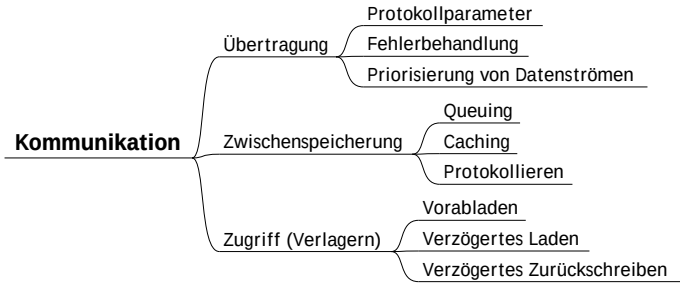

Abbildung 4.13.: Möglichkeiten zur Anpassung der Kommunikation

Dies sind z. B. Mechanismen zur Aufteilung von Daten in Pakete und Rahmen, zur Fehlererkennung und -behandlung, zur Regulierung des Informationsflusses zwischen Netzknoten bzw. Sender und Empfänger und zur Staubehandlung. Protokolle wurden zum Teil für bestimmte Übertragungsmedien konzipiert und treffen bestimmte Annahmen, die aber nicht den Eigenschaften aller Übertragungsmedien entsprechen. Ein Beispiel ist die Staubehandlung in TCP[28]. Außerdem unterstützen konventionelle Protokolle nicht die besonderen Eigenschaften drahtloser oder asymmetrischer Kommunikationsmedien. Insbesondere auf der Sicherungs- und Transportschicht können Protokollparameter und Mechanismen zur Fehlerbehandlung in Abhängigkeit des Übertragungsmediums angepasst werden. Zumeist sind in dieser Klasse angepasste Protokolle einzuordnen, die Mechanismen mehrerer Unterklassen kombiniert einsetzen.

Protokollparameter

Mechanismen dieser Klasse passen Parameter wie die Paketgröße, Fenstergrößen zur Flusssteuerung, Zeitlimits (Timeouts) und die Größe von Werten in Paketköpfen an.

Fehlerbehandlung

Mechanismen zur Fehlerbehandlung werden in Protokollen eingesetzt, um in angepasster Weise Übertragungsfehler zu behandeln. Generell können hier Fehler korrigierende und Fehler behandelnde Verfahren unterschieden werden.

Priorisierung von Datenströmen

Die Priorisierung von Daten bezeichnet Mechanismen, die Datenströme in verschiedene Klassen einordnen und mit unterschiedlicher

[28] Transmission Control Protocol

Dienstgüte übertragen. Dies kann erreicht werden, indem Daten höherer Priorität gegenüber Daten geringer Priorität bevorzugt übertragen werden.

4.3.2. Zwischenspeicherung

Mechanismen zur Zwischenspeicherung von Anwendungsdaten erhöhen zum einen die lokale Verfügbarkeit von Datenobjekten und Zwischenergebnissen. Zum anderen können Daten über einen begrenzten Zeitraum zwischengespeichert werden, um beispielsweise Verbindungsunterbrechungen zu behandeln. Es können die drei Klassen Caching, Queuing und Protokollieren unterschieden werden.

Caching

Unter Caching werden Mechanismen eingeordnet, die Datenobjekte in einem Cache begrenzter Größe temporär zwischenspeichern. Auf die gespeicherten Datenobjekte kann über einen Schlüssel wahlfrei zugegriffen werden. Cachingmechanismen verwenden unterschiedliche Verfahren, um bei zu wenig Speicherplatz enthaltene Anwendendungsobjekte zum Löschen auszuwählen. Darunter zählen LRU[29] und LFU[30]. Für Caching können räumliche und zeitliche Abhängigkeiten in Datenzugriffen ausgenutzt werden. Zeitliche Abhängigkeiten sind gegeben, wenn auf ein gleiches Datenobjekt in kurzen Zeitabständen mehrmals zugegriffen wird. Räumliche Abhängigkeiten bestehen zum Beispiel bei verlinkten Dokumenten. Unter der Annahme, dass einer der Verweise als nächstes verfolgt wird, können beispielsweise Web-Seiten bis zu einer bestimmten Verweistiefe in den Cache geladen werden.

Caching kann genutzt werden, um die zu übertragende Datenmenge und damit gleichzeitig die Antwortzeit zu reduzieren, indem eine mehrfache Übertragung über ein schmalbandiges Netz durch lokale Zwischenspeicherung vermieden wird (z. B. als Browsercache). Weiterhin kann ein Cache eingesetzt werden, wenn aufwendige Verarbeitungsoperationen auf ein Datenobjekt angewendet werden, auf das mehrmals zugegriffen wird. Mechanismen dieser Klasse werden in vielen Projekten vor allem in Kombination mit anderen Mechanismen genutzt. Im Snoop-Protokoll wird Caching benutzt, um nicht bestätigte Pakete auf der Basisstation zwischenzuspeichern und bei Paketverlusten eine Übertragungswiederholung durchzuführen, bevor beim Sender im Festnetz eine Staubehandlung ausgelöst wird. Im verteilten Dateisystem CODA wird Caching in Kombination mit Vorabladen eingesetzt, um bei Abkopplungen die lokale Verfügbarkeit von Dateien zu sichern.

[29] Least Recently Used
[30] Least Frequently Used

Queuing

Queuing-Mechanismen ermöglichen das Einstellen und Entnehmen von Datenobjekten in bzw. aus einer Warteschlange. In dieser Form der Zwischenspeicherung erfolgt im Gegensatz zu Caching eine Ordnung der zwischengespeicherten Daten (z. B. entsprechend der Reihenfolge des Eintreffens oder anhand von Prioritäten). Der Zugriff erfolgt nicht wahlfrei, sondern entsprechend der Ordnung auf das nächste Element. Queuing kann unter anderem eingesetzt werden, um Nachrichten oder Datenobjekte während der Übertragung über unzuverlässigen Verbindungen zwischenzuspeichern.

Protokollieren

Mechanismen zur Protokollierung sind eng verwandt mit Queuing-Mechanismen und können insbesondere durch Warteschlangen realisiert werden. Eine Unterscheidung der beiden Klassen Queuing und Protokollieren erfolgt aufgrund der unterschiedlichen Anwendungsbereiche der Mechanismen. Queuing wird überwiegend zur temporären Zwischenspeicherung von Datenobjekten (z. B. RPC[31]-Nachrichten) zur Überbrückung von Verbindungsunterbrechungen verwendet und zielt damit auf eine robuste Übertragung der Daten. Protokollierung unterstützt dagegen ein autonomes Arbeiten, indem lokal ausgeführte Verarbeitungsschritte protokolliert und nach einem Verbindungsaufbau an einen Server im Festnetz vermittelt werden.

4.3.3. Verlagerung des Zugriffs

Mechanismen zur Steuerung des Zugriffs entkoppeln die Übertragung von Datenobjekten von der Verarbeitung dieser Daten. Generell kann die Übertragung vor oder nach der Verarbeitung der Daten stattfinden. Dadurch erfolgt eine Verteilung der Zugriffe entsprechend der verfügbaren Bandbreite. Die Mechanismen erreichen in der Regel zwar keine Reduzierung der Netzlast, ermöglichen aber eine bessere bzw. effizientere Ausnutzung dieser und erreichen damit verbesserte Antwortzeiten und die subjektiv wahrgenommene Leistungsfähigkeit von Verbindungen.

Vorabladen

Mechanismen zum Vorabladen verlagern den Zeitpunkt der Anforderung und Übertragung vor den Zeitpunkt der Nutzung von Datenobjekten. Im Allgemeinen müssen dazu Informationen über zukünftige Zugriffe auf Daten vorliegen. Diese können entweder durch den Benutzer geliefert werden oder müssen durch Beobachtung des Benutzers bzw.

[31] Remote Procedure Call

unter Berücksichtigung anwendungsabhängiger Informationen gewonnen werden.

In CODA wird ein so genannter Hoarding-Mechanismus eingesetzt, um Dateien für ein abgekoppeltes Arbeiten auf dem Endgerät verfügbar zu machen. Die Informationen über zukünftig benötigte Dateien liefert bei diesem Ansatz der Nutzer in Form einer Konfigurationsdatei. In webbasierten Systemen kann ein Vorabladen anhand der Link-Struktur erfolgen, ohne den Benutzer explizit einzubeziehen. Entsprechend der Verweise können Dateien in bis zu einer festgelegten Hierarchietiefe vorab geladen werden. Vorabladen verbessert die Antwortzeit für Anfragen, falls die Voraussagen zutreffen und erhöht die Verfügbarkeit von Daten bei Abkopplungen. Für das Laden werden aber zusätzliche Ressourcen (z. B. zur Verarbeitung oder zur Übertragung) benötigt, die verschwendet werden, wenn die vorab geladenen Daten nie angefordert bzw. verarbeitet werden.

Verzögertes Laden

Durch Mechanismen zum verzögerten Laden wird der Zeitpunkt des Ladens von Datenobjekten vor den Zeitpunkt deren Nutzung verlagert. Dies bedeutet, dass Anforderungen mit alternativen Daten beantwortet werden müssen, die nachträglich durch die eigentlich angeforderten Daten ersetzt werden können.

Ein Beispiel ist die Generierung von Platzhaltern für Datenobjekte innerhalb einer hierarchisch verlinkten Struktur von Hyperobjekten. Damit wird die Antwortzeit für die Anforderung von Web-Seiten erhöht. Durch Futures ersetzte Hyperobjekte können durch den Nutzer über einen Verweis nachträglich angefordert werden. Eine weitere Reduzierung der Datenmenge wird durch eine feingranulare Strategie zum Laden von Web-Seiten erreicht. Für eine angeforderte Seite wird zunächst nur der auf dem Display darstellbare Bereich geladen. Weitere Teile einer Seite werden durch die Navigation des Benutzers angefordert.

Verzögertes Zurückschreiben

Eine Entkopplung des Zeitpunktes des Zurückschreibens von Änderungen der Daten wird durch ein verzögertes Zurückschreiben ermöglicht. Damit können Daten z. B. zunächst lokal geändert werden, bis alle Operationen abgeschlossen sind (z. B. wenn die Operation close auf einer Datei ausgeführt wird). Danach wird die Summe aller Änderungen über das Netzwerk propagiert. Ein verfeinerter Mechanismus in CODA verteilt das Zurückschreiben von Daten gleichmäßig auf die verfügbare Bandbreite schmalbandiger Verbindungen.

4.4. Zusammenfassung

Das vorliegende Kapitel zeigte neben der umfangreichen Menge an Adaptionsmechanismen für die Kommunikation und die Anwendungsdaten Operationen für die Adaption einer Software-Architektur auf und diskutierte diese. Eine die betrachteten Adaptionsmechanismen zusammenfassende Mindmap ist in Anhang B auf Seite 232 zu finden. Als Mechanismen für adaptive Software auf Ebene der Software-Architektur in der Anwendungsarchitektur wurden das Hinzufügen, Entfernen, Austauschen von Komponenten und die Rekonfiguration der Software-Architektur durch Neuverbinden der Komponenten durch Konnektoren identifiziert. Weitere Operationen sind die Änderung der Parameter von Komponenten, die Migration, Replikation sowie der Austausch (durch Neuzuweisen) von Komponenten in der Systemarchitektur. Neben den vorgenannten Mechanismen wurde außerdem die Anpassung von Schnittstellen sowohl als Initial- als auch als Folgeoperation diskutiert.

Wahrung der Konsistenz und Integrität bei dynamischen Architekturen

Anders als bei den Adaptionsmechanismen für die Daten und die Kommunikation erfordern Änderungen an der Architektur zur Laufzeit mehr Vorsicht. Zum einen ist hier ein Change Management sehr wichtig, da es

- hilft zu identifizieren, was geändert werden muss,
- Kontext bietet für die Erörterung, die Spezifizierung und Implementierung der Änderung und
- kontrolliert, ob bei der Änderung die Systemintegrität bewahrt wird [Oreizy *et al.* 1998].

Zum anderen ist bei den Änderungen am System zur Laufzeit der Zustand zu beachten. In den Ausführungen zu den Adaptionsmechanismen für eine Software-Architektur wurden an ausgewählten Stellen hierzu bereits Anmerkungen gemacht.

Ohne Change Management können die Risiken, die beim Modifizieren des Systems zur Laufzeit entstehen, größer werden als die, das System herunterzufahren und neu zu starten.

Die Rekonfiguration[32] des Systems zur Laufzeit könnte auch durch Nutzung redundant vorgehaltener Hardware geschehen. Das heißt, für die Phase der Rekonfiguration wird der Betrieb auf ein zweites System geschalten, die Änderungen werden offline am ersten System vorgenommen und nach bestandener Überprüfung geht das geänderte System wieder in den Produktionsbetrieb. Dieses Vorgehen ist jedoch

[32] Im Folgenden wird das Wort Rekonfiguration für alle Änderungen an einer Software-Architektur benutzt.

nicht immer machbar, da das Unterhalten eines zweiten, redundanten Systems wegen der damit verbundenen Kosten nicht immer möglich ist. Oder redundante Hardware kann nicht verfügbar sein bzw. wäre nutzlos. Beispiele hierfür sind Kommunikationslösungen, die den Nachrichtenstrom unterbrechungsfrei auf andere Übertragungswege schalten müssen. Ein Konkretes Beispiel hierfür ist auf Seite 94 und in Abbildung 4.4 auf Seite 95 mit dem mobilen Krankenwagen gegeben, der die Übertragung der Patientendaten an die Verfügbarkeit der Netze UMTS, GPRS/GSM und WLAN[33] anpasst.

Zusammenfassend unterscheidet Goudarzi [1999] (zitiert nach [Wermelinger 1999]) drei Fälle, die bei der Rekonfiguration eines Systems beachtet werden müssen, wobei der zweite und dritte Punkt nur für die dynamische Rekonfiguration (dynamisch = zur Laufzeit) von Bedeutung sind:

1. Strukturelle Integrität (z. B. die Architektur muss eine Ring-Topologie bewahren),

2. wechselseitig konsistenter Zustand einer Komponente (z. B. ein Client darf nicht entfernt werden solange der Server noch seinen Request bearbeitet) und

3. Invarianten des Anwendungszustandes (z. B. genau eine Komponente hält den Token in einer Ring-Topologie).

Nur ein System, dass alle drei Fälle beachtet, wird als *korrekt* bezeichnet. Ein resultierendes laufendes System S_{i+1} wird als eine *korrekte, inkrementelle Evolution* eines laufenden Systems S_i bezeichnet, wenn S_{i+1} korrekt ist und wenn das Verhalten der betroffenen Entitäten mit dem Verhalten übereinstimmt, das der nichtbetroffene Teil des Systems erwartet, falls die Rekonfiguration nicht stattgefunden hätte [Almeida 2001].

Ausführlichere Abhandlungen zu der Thematik sind in [Goudarzi 1999] und [Almeida 2001] zu finden.

Auch anzumerken ist, dass vorgenannte Punkte nicht für alle Änderungen an der Software-Architektur zu berücksichtigen sind. Im Allgemeinen sind sie nur bei Änderungen an der Anwendungsarchitektur und der Adaptionsarchitektur zu beachten. Ein Beispiel hierfür ist die Migration einer Komponente. Sollte sich die Zuteilung einer Komponente (oder durch eigenen Antrieb vorgenommen) zu einem anderen Hardware-Knoten nachträglich als ungünstig erweisen, kann die Adaptionsoperation rückgängig gemacht werden. Außerdem müssen bei diesem Mechanismus hinsichtlich der Berücksichtigung anderer Entitäten im System nur die Verbindungen zwischen den Komponten sichergestellt werden (Referenz aktualisieren oder Proxy am alten Ort positionieren).

[33] Wireless LAN

Der weitere Verlauf

Nachdem Adaptionsoperationen für eine Software-Architektur identifiziert worden sind, kann im nächsten Kapitel eine Literaturrecherche vorgenommen werden. Diese sucht zum einen Unterstützungsmechanismen, mit denen die Adaptionsoperationen in der Software-Architektur durchgeführt werden können, zum anderen werden Architekturansätze für die Entwicklung von adaptiven Anwendungen recherchiert. Dabei sollen auch die zuvor angeführten Punkte zur konsistenten und die Integrität bewahrenden Rekonfiguration Berücksichtigung finden.

5 Ansätze in der Literatur

In diesem Kapitel werden Ansätze in der Literatur recherchiert, die eine Adaption an der Software-Architektur vornehmen (vgl. Abschnitt 4.1). Dabei liegt der Fokus auf den Adaptionsmechanismen der Anwendungsarchitektur. Bei einer Recherche müssen deshalb die in der Zusammenfassung des vorhergehenden Kapitels formulierten Aspekte hinsichtlich Sicherung der Integrität und Konsistenz bei der Rekonfiguration der Architektur Berücksichtigung finden.

Unterstützung für die Adaption von Schnittstellen und die Wahrung des Zustandes beim Austausch einer Komponente wurden bereits in den jeweiligen Abschnitten zu den Mechanismen aufgeführt. Diese Ausführungen sind als komplementär zu dem vorliegenden Kapitel zu sehen.

Der erste Teil des vorliegenden Kapitels gibt einen Überblick zu Spezifikationsmöglichkeiten dynamischer Architekturen (5.1). Der sich daran anschließende zweite Teil des Kapitels beschreibt konkrete Architekturansätze für die Entwicklung adaptiver Systeme. Als erstes werden die Modelle der Regelungssysteme vorgestellt (5.2), danach das Architekturmuster Reflection einschließlich Ansätze behandelt (5.3) sowie in Abschnitt 5.4 einige besondere Muster für die Unterstützung adaptiver Systeme beschrieben. Den Abschluss des Kapitels bilden drei weitere konkrete Ansätze (5.5).

5.1. Spezifikationsmethoden

Oft werden die Mechanismen, die eine Adaption vornehmen, ad hoc und extra für ein System entwickelt. Zudem sind die Mechanismen fest in die Applikation integriert und auf Code-Ebene mit der eigentlichen Anwendung fest verdrahtet. Derartige »interne« Mechanismen

bieten den Vorteil, dass ein Fehler an Ort und Stelle detektiert werden kann und dass die Mechanismen in modernen Programmiersprachen (beispielsweise Java's Exception-Handling oder assert-checking) sowie in Laufzeitbibliotheken (z. B. Timeouts für RPC) gut unterstützt werden. Jedoch ist es schwierig, die Adaptionslogik in anderen Applikationen wiederzuverwenden oder zu ändern. Ebenso problematisch ist es, zu analysieren, ob das System nach einer Adaption einwandfrei läuft. Deshalb ist es notwendig, die Adaptionslogik streng von der Anwendung zu trennen [Garlan und Schmerl 2002], [David und Ledoux 2003]. Dieser Ansatz folgt dem Grundgedanken der Trennung der Verantwortlichkeiten (SoC[1]) [Hürsch und Lopes 1995].

Konfigurationsprogrammierung. Viele der Ansätze folgen der Philosophie der Konfigurationsprogrammierung (Configuration Programming, CP) von Kramer und Magee, deren Prinzipien kurz zusammengefasst werden [Kramer und Magee 1998]:

1. Die Konfigurationssprache für die strukturelle Beschreibung sollte separat gehalten sein von der Programmiersprache für die grundlegende Programmierung der Komponenten und auch separat von der Spezifikationssprache für das Verhalten der Komponenten.
2. Die Komponenten sollten als kontextunabhängige Typen definiert werden, die über wohldefinierte Schnittstellen verfügen, und sollten das sichtbare Verhalten für die Komponentenschnittstelle angeben.
3. Bei Benutzung der Konfigurationssprache sollten komplexe Komponenten aus Komponententypen zusammengesetzt werden, wobei die komplexe Spezifikation eine Komposition der Komponentenspezifikationen ist.
4. Eine Änderung sollte auf Konfigurationsebene ausgedrückt werden. Dies trifft auch für Änderungen der Komponenteninstanzen und/oder ihren Verbindungen untereinander und der Komponentenspezifikationen und/oder ihren Interaktionen zu.

5.1.1. Externer, expliziter Ansatz

Verschiedene Autoren propagieren für selbst-adaptive Systeme die Verwendung von Modellen, im Speziellen architektonische Modellen (ebenso sind auch Performance- und Zuverlässigkeitsmodelle möglich), sowie die externe Adaption [Oreizy *et al.* 1999] und [Schmerl und Garlan 2002]. Externe Adaption bedeutet, dass sich die Mechansismen zur Systembeobachtung, Analyse, Adaptionsplanung und Adaptionsausführung *außerhalb* des Systems befinden (siehe Abbildung 5.1). Dafür

[1] Separation of Concern

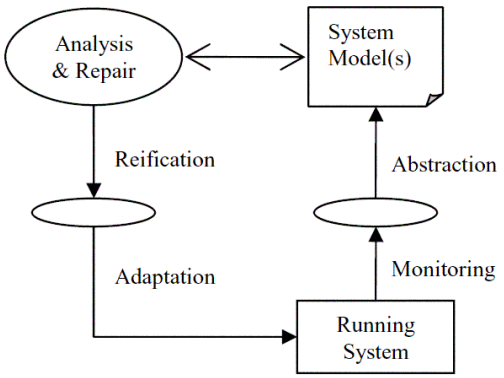

Abbildung 5.1.: (Architektur-) Modell-basierte Adaption [Garlan und Schmerl 2002]. »Reification« (dt. Verdinglichung) sollte in Anlehnung an die Reflection-Terminologie besser »Absorption« heißen.

halten sie Modelle vor, die eine abstrakte und globale Ansicht auf das System liefern.

Eine Architektur-basierte Adaption bietet eine Menge an vorteilhaften Eigenschaften [Schmerl und Garlan 2002], [Garlan und Schmerl 2002]:

1. Gegegeben durch den abstrakten Charakter eines Architekturmodells kann es eine globale Ansicht auf das System liefern.

2. Durch ein Architekturmodell können explizit Integritätsbeschränkungen angeben werden. Dadurch ist es möglich, die Gültigkeit eines Wechsels (Adaption) abzusichern.

3. Entsprechend entwickelte Architekturen erlauben eine flexible Evolution der Systeme, da die Komponenten locker miteinander gekoppelt sind.

4. Architekturen kapseln das Designwissen und bieten dadurch eine Basis für eine grundsätzliche Adaption.

Vorgehensweise und Spezifikationssprachen. In (selbst-) adaptiven Architekturen und in dynamischen Architekturen im Allgemeinen (d. h. Systeme, die »von Hand« durch Herunterfahren oder zur Laufzeit adaptiert werden; d. h. nicht-adaptiv) sind für die Rekonfiguration des Systems auf Basis der Architektur folgende Voraussetzungen notwendig [Oreizy 1996], [Dashofy *et al.* 2002]:

1. Es muss möglich sein, die Architektur des Systems beschreiben zu können — ADLs.

2. Die Änderungen an der Architektur müssen formuliert werden können — AMLs.

3. Das Resultat der Änderungen muss analysiert werden können, um sicherstellen zu können, dass die Änderungen zulässig sind — ACLs.

Zentrale Instanz Konfigurationsmanager. Bei dem zentralen, expliziten Ansatz erfolgt der Rekonfigurationsprozess der Architektur durch explizites Management, d. h. es gibt eine zentrale Instanz, die die in einer Spezifikationssprache (AML) formulierten Änderungen in Kommandos der unterliegenden Plattform übersetzt. Aufgabe des Managers ist es außerdem, die Unterbrechungszeit des laufenden zu minimieren. Für einen solchen Manager sind in der Literatur verschiedene Bezeichnungen zu finden: *Configuration Manager* [Kramer und Magee 1985], *Architectural Evolution Manager* (AEM) [Oreizy *et al.* 1998, 1999], *Coordinator* [Le Métayer 1998] oder *Configuror* [Allen *et al.* 1998].

Nachfolgend wird auf die vorgenannten Punkte näher eingegangen und aufgezeigt, welche Unterstützung derzeit existiert.

Die Architektur beschreiben

Für die Beschreibung von Architekturen hat die Forschergemeinde eine Vielzahl an sog. Architecture Description Languages (ADLs) hervorgebracht. ADLs bieten eine formale Basis für die Beschreibung von Architekturen und erlauben die Spezifikation von Syntax und Semantik bei der Modellierung der Komponenten, Konnektoren sowie der Konfiguration.[2]

Neben der Beschreibung der Struktur, die im Allgemeinen aus Komponenten und die sie verbindenden Konnektoren besteht, können die Architekturbeschreibungssprachen weitere Attribute einer Architektur beinhalten, z. B. Performance-Attribute. Attribute einer Komponente spezifizieren beispielsweise durchschnittliche Bearbeitungszeit für einen Request, benötigte Load; für Konnektoren sind z. B. die Attribute Verzögerung und Bandbreite denkbar [Cheng *et al.* 2002a], [Schmerl und Garlan 2002].

Einen guten Überblick sowie Vergleich der Vielzahl an ADLs bietet [Medvidovic und Taylor 2000].

Die UML ist für die Beschreibung von Architekturen weniger geeignet. Mit UML Version 2.0 unterstützt diese Modellierungssprache zwar endlich auch die Spezifikation von Konnektoren in Komponententdiagrammen (namentlich Delegations- und Assembly-Konnektoren; vgl. [Jeckle *et al.* 2003]), jedoch ist die Semantik der UML-Konnektoren nicht mit

[2] Auch wenn die meisten ADLs weitergehende Modellierungsmöglichkeiten aufzeigen, beispielsweise die der Ports und Rollen, so kann die Existenz der drei oben genannten Element-Typen als »Lackmustest« für die Unterscheidung von ADL zu anderen Notationen verwendet werden [Medvidovic und Taylor 2000, S. 91].

der in ADLs vergleichbar. (Assembly-) Konnektoren in UML spezifizieren nur die Kompatibilität von Interfaces, sodass Komponenten miteinander kommunizieren können. Dagegen besitzen Konnektoren aus ADLs mehr Ausdruckstärke (vgl. Ontologie-Beschreibung für Konnektoren und Rollen in ACME auf Seite 10 und [Mehta *et al.* 2000]). In der Forschungsgemeinde gibt es Bemühungen, Konzepte der ADLs in UML einfließen zu lassen. In [Goulão und e Abreu 2003] findet der interessierte Leser eine Methode, die Architekturaustauschsprache ACME (eine Art kleinster gemeinsamer Nenner aller ADLs) auf UML 2.0 abzubilden; [Kandé *et al.* 2002] und [Medvidovic *et al.* 2002] sind ein guter Einstieg in die Modellierung von Software-Architekturen in UML.

Änderungen an der Architektur beschreiben

Egal ob eine Rekonfiguration des Systems im Rahmen der Selbst-Adaptivität (das System hat selbst festgestellt, dass ein Eingriff notwendig ist) ermittelt wurde oder ob dies durch die menschliche Bedienung identifiziert wurde, die vorzunehmenden Änderungen am System müssen spezifiziert werden. Eine Änderungsbeschreibung gibt an, welche Komponenten und Konnektoren hinzugefügt oder entfernt werden sollen und wie die Topologie geändert werden soll. Für diesen Zweck müssen die ADLs eine dynamische Konfiguration unterstützen.

Bei den meisten ADLs ist die Möglichkeit einer dynamischen Konfiguration nicht gegeben, d. h. sie unterstützen nur die statische Beschreibung des Systems, bieten jedoch keine Möglichkeit für die Spezifikation von Laufzeitänderungen in der Architektur; Ausnahmen sind die ADLs C2, Weaves, Darwin, Rapide und xADL [Oreizy *et al.* 1998], [Medvidovic und Taylor 2000, S. 86].

Bei den ADLs, die eine Rekonfiguration der Architektur zur Laufzeit unterstützen, sind zudem Unterschiede hinsichtlich *geschlossen-adaptiv* und *offen-adaptiv* unterscheidbar (vgl. Kap. 3.7 auf Seite 81):

Darwin [Magee und Kramer 1996] und Rapide [Luckham und Vera 1995] unterstützen nur eingeschränkte Manipulationen der Architektur, d. h. alle möglichen Änderungen müssen bereits während der Entwurfszeit angegeben werden und sind somit fest und unveränderbar in die Anwendung integriert. Damit ist es unmöglich, während der Laufzeit das System um weitere adaptive Fähigkeiten zu erweitern.

Weaves [Gorlick und Razouk 1991] und C2 (in Form einer separaten sog. Architecture Modification Language (AML)) [Medvidovic 1996], [Oreizy 1996] unterstützen dagegen die dynamische Manipulation der Architektur ohne Einschränkung. Jedmögliche Änderung ist erlaubt, wobei die Konsistenz während der Laufzeit sichergestellt wird.

In [Dashofy *et al.* 2002], [van der Westhuizen und van der Hoek 2002] beschreiben die Autoren einen Ansatz, die auf XML basierende xADL (ausgesprochen *zay-dul*) [Dashofy *et al.* 2001], [xADL 2002] um die Möglichkeit der dynamischen Rekonfiguration zu erweitern. Das von ihnen benutzte Vorgehen ist angelehnt an die Arbeitsweise des Pro-

gramms Diff[3], das von Programmierern dazu benutzt wird, den Unterschied zweier Quellcode-Dateien auf Zeilenbasis zu ermitteln. Genau nach dem gleichen Prinzip, allerdings auf semantischen Niveau und für Architekturbeschreibungen, arbeitet ArchDiff. ArchDiff, das Diff für Architekturen, ermittelt aus zwei in xADL gehaltenen Architekturbeschreibungen den Unterschied, d. h. welche Komponenten und Konnektoren in der Architektur hinzugefügt, ausgetauscht oder entfernt wurden (im Kontext dynamischer Architekturen: hinzuzufügen, auszutauschen oder zu entfernen sind). Dieser architektonische Diff kann als Anleitung dazu genutzt werden, die Konfiguration an einer Architektur zu ändern. Als Analogon zu dem Programm Patch[4], mit dem ein Programmierer eine bestehende Quellcode-Datei per Diff anpassen kann (in diesem Fall spricht man von patchen), arbeitet auf Architektur-Niveau das Programm ArchMerge, das die spezifizierten Änderungen an einer Ausgangsarchitektur anwendet und somit die Beschreibung für die Zielarchitektur generiert.

Auch die Architekturaustauschsprache (die Entwickler bezeichnen sie mittlerweile bereits als eigenständige Architekturbeschreibungssprache) ACME unterstützt die Dynamik [Wile 2001] mihilfe von Armani [Monroe 2001].

Die vorzunehmenden Änderungen analysieren

Je nach Art des zu ändernden Systems müssen verschiedene Analysen durchgeführt werden, um sicherstellen zu können, ob eine Änderung an der Architektur zulässig ist.

In der Zusammenfassung zum vorhergehenden Kapitel wurde bereits auf Seite 120 das Problem einer korrekten Rekonfiguration zur Laufzeit angesprochen. Als zu beachtende Anforderungen wurden die Wahrung der Konsistenz, der strukturellen Integrität und der Invarianten des Anwendungszustandes genannt (für detailliertere Ausführungen zur Thematik siehe [Goudarzi 1999] und [Almeida 2001]). Sobald für eine Anpassung die Änderungsoperationen formuliert worden sind, müssen diese dahingehend analysiert werden, ob das adaptierte System S_{i+1} ein korrektes sein wird.

Die Analysen können sehr überschaubar und universal sein. Eine einfache Analyse kann beispielsweise die Prüfung umfassen, ob nach der Änderung der Software-Architektur alle *requires*-Abhängigkeiten der Komponenten erfüllt werden. Dagegen können auch spezifischere Analysen notwendig sein, z. B. wenn in einer Anwendung nur eine Datenbankkomponente erlaubt ist.

Ausgangspunkt für die Analysen ist ein aktuelles Architekturmodell des laufenden Systems. Auf diesem Modell wird mithilfe der Änderungsbeschreibung das resultierende System nach verschiedenen Kri-

[3] http://www.gnu.org/software/diffutils/diffutils.html
[4] http://www.fsf.org/software/patch/patch.html

terien analysiert. Wenn alle Prüfungen ohne Fehlermeldung erfolgreich ausgeführt worden sind, ist dies ein Signal dafür, dass die gleichen Änderungen ebenso am laufenden System ausgeführt werden können. Falls jedoch eine Prüfung fehlschlägt, muss eine neue Planung angestoßen werden, bei der der identifizierte Fehler in die Neuplanung einer andere Adaptionsoperation einbezogen wird.

Architectural Constraint Languages. Für die Spezifikation von Randbedingungen in einer Architektur existieren sog. ACL[5]s. Bei den Ansätzen für die Einschränkung der Struktur werden sowohl imparative [Balzer 1996] als auch deklarative [Magee und Kramer 1996] Spezifikationssprachen verwendet. Ein anderer Mechanismus ist die Einschränkung auf Basis des Verhaltens der Komponenten und ihren Interaktionen [Luckham und Vera 1995] (vgl. [Oreizy *et al.* 1998]).

Formalere Ansätze

Wermelinger stellt in seiner Dissertation [Wermelinger 1999] drei Ansätze für die formale Spezifikation beliebiger dynamischer Rekonfigurationen auf der Architekturebene vor. Die Mehrzahl an verschiedenen Ansätzen begründet er damit, dass es keinen universellen Ansatz gibt.

Alle drei Ansätze nutzen Graphen wegen ihrer Eignung für Architekturen, der einfachen aber entscheidenden mathematischen Basis und der intuitiven Beschreibungsfähigkeiten. Die Ansätze sind nicht neu, sondern stammen aus Arbeiten anderer Forscher; Wermelinger erweitert diese um interessante Aspekte. Allen drei Ansätzen gemeinsam ist beispielsweise die Unterstützung hierarchischer Architekturen. Die Besonderheiten der drei Ansätze werden nachfolgend kurz beschrieben.

Transaktioneller Ansatz Bei dem transaktionellen Ansatz ist der Grundgedanke aus vorherigen Arbeiten [Kramer und Magee 1990], [Goudarzi und Kramer 1996], dass einige Komponenten »eingefroren« werden, um die Konsistenz während einer Rekonfiguration sicherzustellen. Wermelinger ändert dies zu einem Konnektor-basierten Ansatz um, da es hier einfacher ist, die benötigte Menge an Konnektoren zu ermitteln, die »eingefriert« werden muss (der alte Algorithmus ermittelte zu viele Komponenten). Neben der Minimierung der an der Rekonfiguration beteiligten Elemente wird auch die Ausführungszeit verringert, indem die Ausführungsreihenfolge parallelisiert wird.

COMMUNITY Der COMMUNITY-Ansatz gehört zur Klasse der sog. Coordination Languages [Gelernter und Carriero 1992], bei denen besonderer Fokus auf die Trennung der Koordination von der Verarbeitung gelegt wird. Dieser Ansatz basiert auf der Gruppen-Theorie (category theory) und bietet ein abstraktes algebraisches

[5] Architecture Constraint Language

Framework, um die Konnektoren, die Stile und Architekturen beschreiben und auf ihnen operieren zu können. Wie im CHAM-Ansatz werden Rekonfigurationen als bedingtes Umschreiben der Graphen gehandhabt. In späteren Arbeiten [Wermelinger *et al.* 2001] und [Lopes *et al.* 2002] erfährt der COMMUNITY-Ansatz Erweiterung.

CHAM Der dritte Ansatz ist eine Erweiterung von CHAM und wird wegen seiner besonderen Eignung für selbstorganisierende Software-Architektur im nachfolgenden Abschnitt 5.1.2 vorgestellt.

Alle drei aufgeführten Ansätze haben sowohl Vor- als auch Nachteile. Neben den genannten Vorzügen, die jeder Ansatz mit sich bringt, ist ein Vorteil der starke formale, mathematische Charakter der Ansätze. Die Ansätze sind nicht an eine bestimmte Anwendung, Domäne, Framework oder Programmiersprache gebunden. Dies stellt jedoch zugleich einen Nachteil dar. So gibt es noch keinen Prototyp oder Toolunterstützung für die Spezifikation und Analyse — mit Ausnahme des COMMUNITY-Ansatzes [Fiadeiro 2004].

5.1.2. Selbstorganisation der Architekturkomponenten

In einer selbstorganisierenden Software-Architektur gestalten die Komponenten ihre Interaktion derart, dass sie kompatibel mit einer allumfassenden architektonischen Spezifikation ist. Die Motivation liegt in der Minimierung des expliziten Managements.

Alloy

In [Georgiadis *et al.* 2002] untersuchen die Autoren die Umsetzbarkeit, architektonische Randbedingungen (architectural constraints) als Basis für die Spezifizierung, Entwicklung und Implementierung von selbstorganisierenden Architekturen für verteilte Systeme einzusetzen.

Mithilfe von architektonischen Randbedingungen verifiziert ein so genannter Configuration Manager (es gibt auch andere Bezeichnungen wie bspw. Architecture Evolution Manager (AEM), Coordinator oder nur Configurator; vgl. Seite 126), dass sich ein laufendes System entsprechend seiner architektonischen Spezifikation verhält. Konkret bedeutet das die Sicherstellung und Spezifikation, welche Komponente mit welcher anderen Komponente präzise miteinander kommunizieren.

Für dynamische, offene Systeme kann ein solcher Ansatz, wie er im vorhergehenden Abschnitt 5.1.1 geschildert wurde, jedoch zu restriktiv sein, da Komponenten auftauchen und wieder verschwinden können. Die Spezifikation der Struktur ist also nicht anwendbar. Ist dennoch eine architektonische Spezifikation gewünscht (gegeben durch die Motivation, die Vorzüge einer Gesamtarchitekturspezifikation beizubehalten), dann kann diese Spezifikation keine präzise Beschreibung der

Komponenteninstanzen und ihren Kopplungen sein, sondern enthält eher Constraints bezüglich der Komposition der Komponenten.

In ihrer Arbeit berichten Georgiadis und Kollegen von der Umsetzung ihres Ansatzes mithilfe der Constraint Language Alloy. Außerdem haben sie ein dezentralisierten Laufzeitsystem implementiert, das die strukturelle Selbstorganisation unterstützt. Einschränkend ist hier anzumerken, dass sie atomaren Broadcast für das Management verwenden. Dieser Ansatz setzt voraus, dass keine Segmentierung in getrennte Netzwerkpartitionen auftreten. Außerdem ist dies keine effiziente und skalierbare Lösung.

CHAM-Ansatz

Ein anderer Ansatz, dem mehr Formalismus aber keine Laufzeitunterstützung unterliegt, ist der CHAM-Ansatz. Ursprünglich von Inverardi und Wolf [1995] und Le Métayer [1998] für die Spezifikation von statischen Architekturen entwickelt, wurde CHAM durch Wermelinger [1999] für die Unterstützung von Dynamik weiterentwickelt. Der CHAM-Ansatz bietet einen formalen Weg, die architekturelle Rekonfiguration im Sinne einer Reaktion von Molekülen auszudrücken. Komponenten (oder Ansammlungen von Komponenten) können als Atome präsentiert werden, die zu Molekülen verbunden sind. Rekonfigurationen werden als Reaktionen beschrieben. Anhand mehrerer Reaktionsregeln kann festgestellt werden, ob eine bestimmte Rekonfiguration gültig ist, da alle zulässigen Rekonfigurationen in den Reaktionsregeln spezifiziert sind. Dadurch können Komponenten zu einer Architektur zusammengesetzt werden.

5.1.3. Zusammenfassung

Das vorliegende Unterkapitel hat Unterstützung durch verschiedene Spezifikationsprachen für dynamische Software-Architekturen aufgezeigt. Die Jahreszahlen und die Vielzahl der referenzierten Paper zeigen, dass die formale Unterstützung für dynamische Software-Architekturen bereits seit vielen Jahren im Interesse der Forschung steht. In der hier erfolgten Zusammenstellung verfügbarer Ansätze wurden die Spezifikationsmethoden grob nach dem externen Ansatz und nach dem selbst-organisierenden Ansatz eingeteilt.

Bei dem externen Ansatz gibt es eine zentrale Instanz, den Konfigurationsmanager (Configuration Manager, auch genannt AEM, Coordinator, Configuror). Dier verwaltet ein Modell der Software-Architektur und ermöglicht durch sog. AMLs in Verbindung mit ACLs die Rekonfiguration der Software-Architektur.

Ganz anders erfolgt die Rekonfiguration bei selbstorganisierenden Software-Architekturen. Bei diesen gibt es keine zentrale Instanz, die den Rekonfigurationsprozess bewerkstelligt, da das Management minimiert werden soll. Außerdem können Komponenten auftauchen

und wieder verschwinden, sodass bei allen vorgenannten Bedingungen »normale« ADLs passen müssen. Vorgestellt wurden zwei ADL-Ansätze, die dem in selbstorganisierenden geläufigen Verfahren folgen: »... nicht die Beschreibung der Struktur — die ist meist hochkomplex — vorgeben, sondern eher eine Beschreibung der Dynamik, wie sich die Struktur aufbaut oder auch eine Beschreibung der Constraints, die einzuhalten sind,« formuliert Prof. Schmeck von der Universität Karlsruhe das Prinzip für selbstorganisierende Architekturen in [Rademacher 2004].

Des Weiteren wurde für die AML[6]s eine Charakterisierung hinsichtlich der Unterstützung, die Regeln für die Rekonfiguration einer Software-Architektur dynamisch zu ändern (offen-adaptiv), vorgenommen. Eine weitere bewertende Charakterisierung der Ansätze soll nicht durchgeführt werden, da diese nicht im Interesse des Aufgabenstellers liegt. Möglichkeiten für eine weitere Beurteilung, anhand derer eine Bewertung vorgenommen werden könnte, werden beispielsweise in [Mikic-Rakic *et al.* 2002] und [Medvidovic und Taylor 2000] gegeben.

Andere Spezifikationssprachen. Dieses Unterkapitel konnte nur einen kleinen Einblick für die Spezifikation dynamischer Architekturen gegeben. Es gibt noch weitere Ansätze, die von den innerhalb dieser Arbeit beschriebenen differieren. Einen weiteren Überblick bieten [Wermelinger 1999], [Cuesta *et al.* 2001] und [Bradburya *et al.* 2004][7].

Darüber hinaus ist es wichtig anzumerken, dass neben der Spezifikationsmöglichkeit für dynamische Architekturen auch die Spezifikation der speziellen Eigenschaften von Komponenten (funktionale und nichtfunktionale Eigenschaften) notwendig ist. Im Rahmen der Erörterung zur Schnittstellenanpassung wurde bereits auf Seite 102 auf die Spezifikation der funktionalen Eigenschaften auf den unterschiedlichen Ebenen der Interoperabilität eingegangen. Für die Beschreibung nichtfunktionaler Eigenschaften seien hier nur QuO [Loyall *et al.* 1998], [Zinky *et al.* 2002] und CQML$^+$ [Röttger und Zschaler 2003] genannt. Auf eine eingehendere Behandlung dieser Aspekte muss verzichtet werden.

5.2. Kontroll-Theorie

Ein wesentliches Ziel von Softwareanpassung ist *Robustheit* [Meng 2001], [Laddaga 1999]: eine selbst-adaptive Software kann ihr Verhalten zur Laufzeit verändern, um sich in Einklang mit den Änderungen in der Umgebung zu bringen, damit ihr Entwicklungsziel erreicht wird. Autonome Roboter bieten ein konkretes Beispiel für die gleichen Herausforderungen, denen sich auch die Softwareentwickler stellen müssen: die Algorithmen, die in dem Roboter Entscheidungen

[6] Architecture Modification Language
[7] Erst nach Redaktionsschluss recherchiert

treffen, müssen sich an *zur Entwicklungszeit unbekannte* Situationen anpassen, basierend darauf, was zuvor über die Umgebung bekannt war und was die Sensoren jetzt mit ihren Sensoren erkennen. In Analogie dazu die Perspektive der Software: Die *Entwurfsspezifikation* ist die bekannte statische Welt, während der *Laufzeitkontext* der Umgebung die gegenwärtige Situation ist. Das heißt, die Software wird für eine bestimmte Umgebung spezifiziert, die zur Entwicklungszeit nicht im vollen Maße berücksichtigt werden konnte.[8]

Herkömmliche Softwaretechnologie ist nicht fähig, diesen Anforderungen gerecht zu werden, da die Software während der Entwicklung für alle voraussehbaren Änderungen modelliert und verifiziert wird. Adaptive Systeme bieten eine andere Lösung: Eine Rückkoppelungsschleife beobachtet die SystemPerformance und ändert das System entsprechend [Karsai und Sztipanovits 1999].

Auch für andere Probleme als die der Robustheit von Software bieten die Modelle der Regelungssysteme konkrete Ansätze für adaptive Systeme. Beispiele hierfür, die auch in der vorliegenden Arbeit vorgestellt werden, sind das Rainbow-Framework in Kapitel 5.5.2 auf Seite 167 und das Regelkreissystem des Autonomic Computing in Kapitel 5.5.3 auf Seite 170. Neben den explizit vorgestellen Arbeiten ist der Kontrollansatz, hier insbesondere mit der Rückkoppelungssteuerung, in vielen Literaturquellen für Adaptivität und für das Selbst-Management in Software zu finden [Eracar und Kokar 2000], [Robertson und Brady 1999], [Oreizy *et al.* 1999], [Valetto und Kaiser 2002a], [Karsai *et al.* 2003], [Hinnelund 2004], [Taylor und Tofts 2004].

Das vorliegende Unterkapitel nimmt eine Generalisierung der Konzepte vor und beschreibt diese aus Sicht der Software-Architektur (in Anlehnung an [Shaw und Garlan 1996, Kap. 2.8], [Meng 2001] und [Kokar *et al.* 1999]).

Glossar

Bevor die einzelnen Modelle der Regelungssysteme vorgestellt werden, werden an dieser Stelle einige Definitionen und Begriffe angeführt.

Plant In der Kontrollterminologie wird das kontrollierte System als *Plant* bezeichnet, im Folgenden auch kurz nur *System*.

Prozessvariablen Eigenschaften des Prozesses, die gemessen werden können. Es werden zudem spezifischere Arten (kontrollierte Variablen und Inputvariablen) noch unterschieden.

Kontrollierte Variable Prozessvariable, deren Wert das System kontrollieren möchte.

[8] Systeme, deren Verarbeitung nicht von einer komplett vorbestimmten und modellierten Umgebung abhängig ist, werden als Typ-2-Systeme bezeichnet. Die andere Art bilden Typ-1-Systeme, für die die Umgebung, in der sie operieren, komplett vorbestimmt sein muss [Robertson 2001].

Inputvariablen Prozessvariable, die als Eingabe auf den Prozess einwirkt und gemessen werden kann.

Robustheit Fähigkeit eines Systems, sein Ziel auch dann zu einhalten zu können, wenn große, unerwartete Schwankungen auftreten. In anderen Worten: Wenn sich die Umgebung des Systems von dem entfernt, wofür das Systems spezifiziert wurde (Typ-2-Systeme), dann zeigt das System keinen katastrophalen Fehler auf, sondern womöglich nur verminderte Leistung.

Stabilität Eigenschaft eines Systems, die sicherstellt, dass kleine Änderungen (Störungen) des initialen Zustands nur einen geringfügigen Effekt auf das Systemverhalten haben. Das kann beispielsweise heißen, dass das System keine Schwankungen als Antwort auf kleine Eingaben aufzeigt.

Beherrschbarkeit (controllability) Die Fähigkeit, ein System in die richtige Richtung zu steuern. Ein System muss beherrschbar sein, um sicherstellen zu können, dass es sich entsprechend seiner Spezifikation verhält, wenn sich die Eingabewerte unerwartet ändern.

In den nachfolgenden Diagrammen entsprechen Pfeile ohne Quelle Eingaben von der Umgebung, Pfeile ohne Ziel sind Ausgaben an die Umgebung.

5.2.1. Optimalwertsteuerung / offener Regelkreis (feedforward control/open loop)

Das einfachste Modell, das die Kontroll-Theorie berücksichtigt, ist der offene Regelkreis (open loop), auch Optimalwertsteuerung (feedforward control) genannt [Shaw und Garlan 1996, Kap. 2.8], [Meng 2001], [Kokar *et al.* 1999]. Dieses sehr einfache Modell findet mit der Read-Evaluate-Print-Schleife in traditionellen Batch-Programmen oder mit der Ereignisbehandlungsschleife (event–dispatch–handle) in GUIs Verwendung. Die Abstrahierung von diesen beiden Beispielen ist als Controller-Entwurfsmuster in [Larman 2002, Kap. 16.10] formuliert worden.

Das Modell des offenen Regelkreises, dargestellt in Abbildung 5.2 auf der nächsten Seite, beinhaltet einen Prozess, auf den die dynamische Umgebung einwirkt (in Form der *Inputvariablen*). Die Komponenten für die Optimalwertsteuerung sind essenziell die gleichen wie die der Rückkoppelungssteuerung, mit dem Unterschied, dass der Kontext (die Inputvariablen) für den Controller beobachtbar und ebenso die Auswirkungen auf das System dem Controller bekannt sind. Hier werden jedoch die Werte der *kontrollierten Variablen* von anderen Prozessvariablen gemessen und dem Controller als Eingabe übergeben.

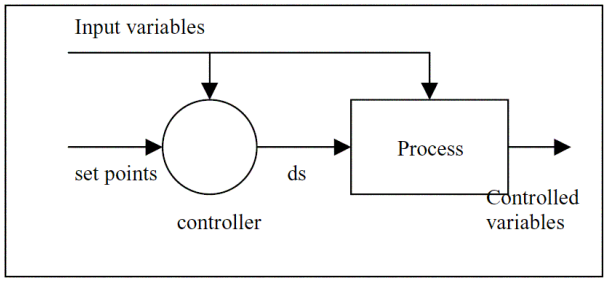

Abbildung 5.2.: Optimalwertsteuerung/Feedforward [Meng 2001]

Offene Regelkreise finden bei Systemen Anwendung, für die gut vorhergesagt werden kann, welche Eingaben sie benötigen, um einen bestimmten Zustand oder eine bestimmte Ausgabe zu erreichen.

5.2.2. Rückkoppelungssteuerung / geschlossener Regelkreis (closed loop/feedback control)

Das Modell der Rückkoppelungssteuerung sieht einen Prozess innerhalb einer dynamischen Umgebung, die über die *Inputvariablen* auf ihn einwirkt (siehe Abbildung 5.3). Die *kontrollierten Variablen* sind eine Untermenge der messbaren Größen eines Systems (»Prozessvariablen«), für die sich der Controller interessiert. Der Controller wirkt steuernd auf den Prozess ein. Dazu vergleicht er die Sollgrößen »set points« mit den Werten der *kontrollierten Variablen* und ermittelt daraus das Delta *ds*.

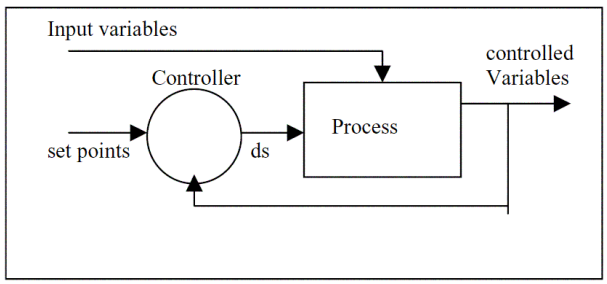

Abbildung 5.3.: Rückkoppelungssteuerung/Feedback [Meng 2001]

Für (selbst-) adaptive Systemen ist der geschlossene Regelkreis der am häufigsten anzutreffende Ansatz (vgl. [Eracar und Kokar 2000], [Oreizy *et al.* 1999], [Valetto und Kaiser 2002a], [Hinnelund 2004], [Taylor und Tofts 2004]).

5.2.3. Vergleich Rückkoppelungs- und Optimalwertsteuerung

Das Modell der *Optimalwertsteuerung* eignet sich für Systeme, bei denen das Verhalten des Prozesses oder der kontrollierten Variablen gut verstanden ist [ApICS 2001]. Deshalb wird dieses Modell auch als »proactive« [Poladian 2003] oder »deliberative« (abwägend) [Meng 2001] bezeichnet.

Der Zweck einer *Rückkoppelungssteuerung* ist die Einhaltung der Ausgabewerte eines Prozesses, d. h. die Sicherstellung, dass diese innerhalb einer hinreichenden Spanne der Sollgrößen liegen (z. B. Stabilität). Dieses Modell eignet sich auch, wenn die bei der Optimalwertsteuerung erreichte Genauigkeit unzureichend für die Performancespezifikation der Anwendung ist [ApICS 2001]. Aufgrund ihres Charakters, dass die Ausgabewerte eines Systems beobachtet werden, wird die Rückkoppelungssteuerung auch »reactive« (rückwirkend) genannt [Meng 2001], [Poladian 2003].

Der Unterschied einer Optimalwertsteuerung zu einer Rückkoppelungssteuerung ist der, dass bei der Optimalwertsteuerung die Inputvariablen beobachtet werden, um den Prozess zu steuern, wohingegen bei der Rückkoppelungssteuerung die kontrollierten Variablen (Ausgabe) gemessen werden (vgl. Abbildungen 5.2 und 5.3). Ein einfaches Beispiel hierfür ist die Temperatursteuerung für einen Raum [Poladian 2003]: Um die Innentemperatur des Raumes zu regeln, misst eine Optimalwertsteuerung die Außentemperatur (die nicht kontrollierbaren Inputvariablen), eine Rückkoppelungssteuerung die Innentemperatur (die kontrollierbaren Variablen).

5.2.4. Ansatz: Kombination Rückkoppelungs- und Optimalwertsteuerung

Nach Meinung von Meng [2001] beschreiben die Modelle der Rückkoppelungs- und Optimalwertsteuerung genau selbst-adaptive Software-Systeme. Diesem Paradigma folgend können gewöhnliche Systeme bis auf einige Ausnahmen als Aktoren oder Prozesse, die kontrolliert werden müssen, verstanden werden: Die Rückkoppelungskomponente (feedback/reactive) nimmt die Anpassungen an die Änderungen in der Umgebung vor. Die Optimalwertsteuerungskomponente (feedforward/deliberative) schreibt das vorausberechnete Verhalten des Systems vor, das über lange Zeit die Konsistenz garantiert. Deshalb integriert Meng für ein selbst-adaptives System idealerweise die Komponenten für Rückkoppelungs- und Optimalwertsteuerung (siehe Abbildung 5.4 auf der nächsten Seite). Die Optimalwertsteuerungskomponente (deliberative/feedforward) stellt die Spezifikation der Software sowie ihre Vorhersagbarkeit bereit, die Rückkoppelungskomponente (reactive/feedback) reagiert auf Änderungen in der Laufzeitumgebung.

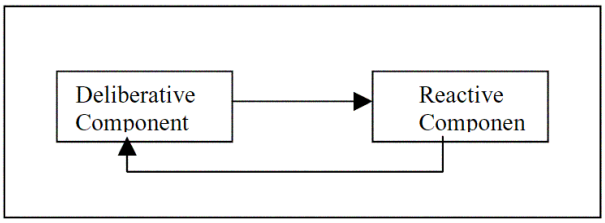

Abbildung 5.4.: Integrierte Architektur für selbst-adaptive Systeme [Meng 2001]

5.2.5. Ansatz einer selbst-kontrollierenden Architektur

Einen anderen Ansatz bieten die Autoren in [Kokar *et al.* 1999]. Sie erweitern die zuvor beschriebenen Modelle der offenen und geschlossenen Regelungssysteme um »Adaptive Control« (adaptive Regelungssystem) und »Reconfigurable Control« (rekonfigurierbares Regelungssystem).

Bevor der Ansatz von Kokar *et al.* für selbst-adaptive Systeme vorgestellt wird, wird kurz auf die vorgenannten erweiterten Techniken für Regelungssysteme eingegangen.

Adaptive Control. Die Grundlagen zu adaptiven Regelungssystemen (auch Adaptivregelung genannt) wurden von Åström und Wittenmark in ihrem 1989 erschienen Buch [Åström und Wittenmark 1989] gelegt (im Folgenden erfolgt die Beschreibung für adaptive Regelungssysteme in Anlehnung an [Landau *et al.* 1997] und [Kokar *et al.* 1999]). Eine Adaptivregelung eignet sich für Systeme mit breiten, dynamischen Störungen jeder Art und strengen Zeiterfordernissen. Auch werden Probleme von Systemen gelöst, deren Störungen unvorhersagbar auftreten sowie sich ändern, sodass wegen der nicht-linearen Merkmale der Anlage eine unvorhersagbare Anwort hervorgerufen wird. Bei der Verwendung einer Adaptivregelung brauchen die exakten Werte der Modellparameter und die des Controllers nicht genau bekannt sein.

Für eine Adaptivregelung gibt es zwei allgemeine Ansätze: das direkt und das indirekt adaptive Regelungssystem. Bei einem *direkt adaptiven Regelungssystem* wird das Anlagenmodell in Bezug auf die Controller-Parameter parametrisiert, wobei die Controller-Parameter direkt geschätzt werden. Bei dem *indirekt adaptiven Regelungssystem* werden die Parameter für das kontrollierte System zuerst geschätzt und anschließend dafür genutzt, die Controller-Parameter zu berechnen. Dieser Zweistufenprozess hat dem indirekten Ansatz seinen Namen gegeben.

Das von Kokar *et al.* für selbst-adaptive Systeme propagierte Modell ist die indirekte Adaptivregelung. Bei diesem Ansatz gibt es, verglichen mit der üblichen Rückkoppelungssteuerung, zusätzlich zwei Subsysteme: Controller Designer und die Komponente mit der Schätzfunktion

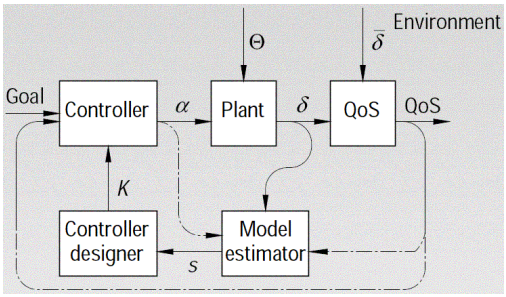

Abbildung 5.5.: Indirekt adaptives Regelungssystem [Kokar *et al.* 1999]. Legende: α = die vom Controller berechneteten Variablen, mit denen das zu kontrollierende System parametrisiert werden kann; δ = System-Output; $\overline{\delta}$ = externer Feedback; Θ = einwirkende Umgebung; K = Kontrollgesetz; s = Signal für Erneuerung des Kontrollgesetzes.

für das Modell (Model Estimator) (siehe Abbildung 5.5). Der *Model Estimator* hat die Aufgabe, das Modell schrittweise zu aktualisieren, nachdem er die Eingaben vom System (plant) erhalten hat. Der *Controller Designer* übernimmt die zweite Stufe und aktualisiert das Kontrollgesetz.

Eine Erweiterung der Adaptivregelung kann vorgenommen werden, wenn diese rekonfigurierbar ist.

Rekonfigurierbares Regelungssystem. Ein adaptives Regelungssystem führt zwar einige Flexibilität gegenüber den konventionellen Rückkoppelungssteuerungen ein, hat jedoch auch einige Einschränkungen zu verzeichnen. Die offensichtlichste Einschränkung besteht darin, dass die Logik für die Identifikation und die Entscheidungsfunktionen während der Laufzeit unveränderlich ist, da sie während des Entwurfs des Controllers implementiert worden ist. Deshalb hat ein adaptiver Controller eine beschränkte Fähigkeit, das Kontrollgesetz auf den neusten Stand zu bringen. Er kann es nur innerhalb einer festgelegten Klasse von Modellen aktualisieren (die durch die Parameter vorgegebene Unsicherheit des Modells). Nicht möglich ist es für den Controller, mit allen Arten von nicht-parametrischen Unsicherheiten, hoch- sowie niederfrequenten nichtmodellierten Dynamiken und Sensorrauschen umzugehen.

Auch wenn eine Adaptivregelung sich an neue Situationen anpassen kann, könnte die nicht-lineare Charakteristik des kontrollierten Systems wieder auftauchen. Das Anpassen an dieses sich wiederholende Ereignis scheint unvernünftig zu sein. Ein wirtschaftlicherer Ansatz wäre es, das Kontrollgesetz für eine bestimmte dynamische Charakteristik zu lernen, es in einer Controller-Datenbank zu speichen und wiederzuverwenden, sobald eine Situation mit einer bekannten Charakteristik wieder auftritt. Diese Fähigkeit erfordert einen intelligenten

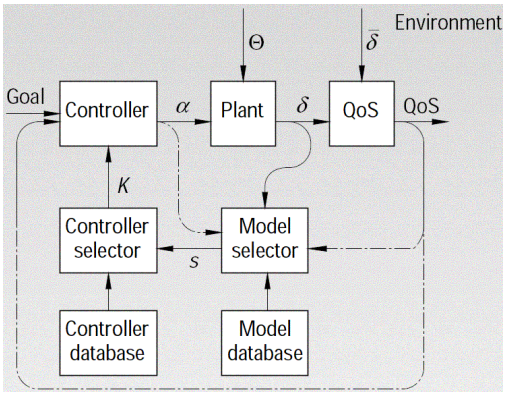

Abbildung 5.6.: Rekonfigurierbarer Controller [Kokar *et al.* 1999]. Legende: α = die vom Controller berechneteten Variablen, mit denen das zu kontrollierende System parametrisiert werden kann; δ = System-Output; $\overline{\delta}$ = externer Feedback; Θ = einwirkende Umgebung; K = Kontrollgesetz; s = Signal für Erneuerung des Kontrollgesetzes.

Controller, der sich nicht nur an ein neues Kontrollgesetz anpassen und sich auch an dieses erinnern kann (lernen), sondern er muss auch ein geeignetes Kontrollgesetz auswählen können.

Für dieses Problem hat Shamma einen Ansatz für ein rekonfigurierbares Regelungssystem entworfen [Shamma 1996]. Verglichen mit dem Modell für den adaptiven Controller (Abbildung 5.5 auf der vorherigen Seite) hat das Modell für einen rekonfigurierbaren Controllern sehr große Gemeinsamkeiten. Der *Controller Designer* ist gegen einen *Controller Selector* ausgetauscht worden, der *Model Estimator* wurde durch einen gleichnamigen *Selector* ersetzt (vgl. Abbildung 5.6). Ebenso hinzugekommen sind zwei Datenbanken, die zu dem Selector gehören (*Controller Database* und *Model Database*). Der *Model Selector* übernimmt alle Aufgaben, die vorher der *Model Estimator* im adaptiven Regelungssystem inne hatte. Zusätzlich dazu kann er ein anderes Modell aus der *Modelldatenbank* (model database) auswählen, wenn er eine signifikante Änderung im Systemmodell erkennt, d. h. wenn sich z. B. eine strukturelle Änderung im kontrollierten System ergibt. Kausal zusammenhängend mit der Auswahl eines anderen Modells ist auch die Änderung des Kontrollgesetzes, die mithilfe der *Controller-Datenbank* (controller database) durch den *Controller Selector* vorgenommen wird.

Kombinierter Ansatz. Aufbauend auf den vorgenannten Ansätzen haben Kokar *et al.* eine integrierte Architektur für eine sich selbstkontrollierende Software in [Kokar *et al.* 1999] vorgestellt (siehe Abbildung 5.7 auf der nächsten Seite).

Diese Architektur weist große Ähnlichkeiten zu dem zuvor beschriebenen rekonfigurierbaren System. Auffallend ist, dass der Model Selec-

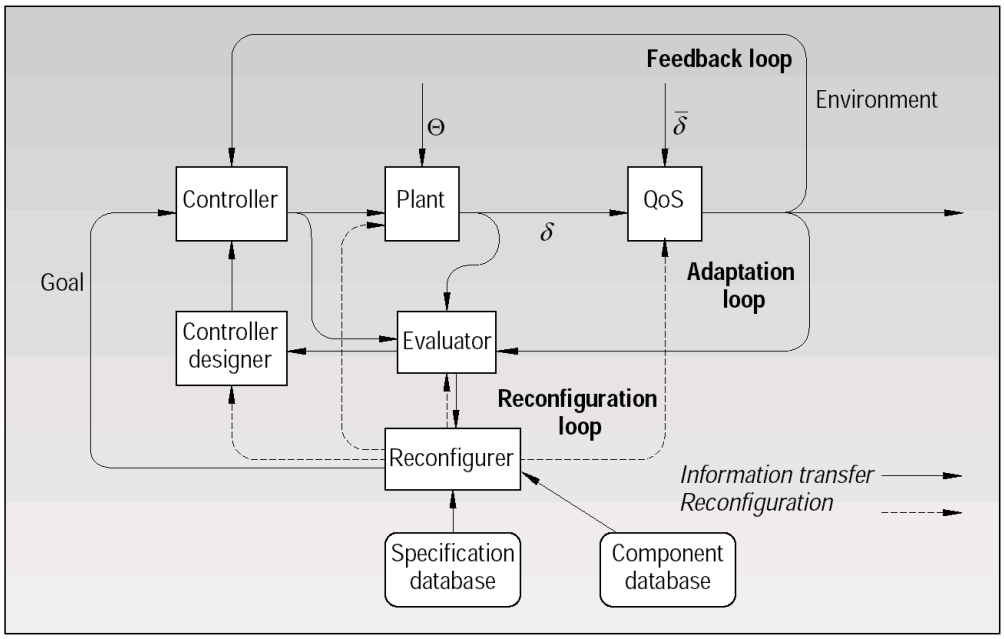

Abbildung 5.7.: Modell einer selbst-kontrollierenden Software [Kokar *et al.* 1999]. Legende: δ = System-Output; $\overline{\delta}$ = externer Feedback; Θ = einwirkende Umgebung.

tor durch die zwei zentralen Komponenten *Evaluator* und *Reconfigurer* ersetzt worden ist. Außerdem kann die Architektur in drei Schleifen eingeteilt werden. Diese werden nachfolgend beschrieben.

Rückkoppelungsschleife (feedback loop)

Die erste Schleife ist die Rückkoppelungsschleife. Diese besteht wie üblich aus dem *Controller* sowie dem *zu überwachenden System (plant)*. Außerdem wird in der Architektur eine *Dienstgüte-Komponente* (QoS[9]) eingesetzt. Kokar *et al.* [1999] begründen dies damit, dass entgegen echten Kontrollsystemen in Softwaresystemen die Rückkoppelung üblicherweise nicht direkt erfolgen kann, sondern durch eine zwischengeschaltete QoS-Komponente vorgenommen wird. Der Controller hat die Aufgabe, die Parameter für das System derart einzustellen, dass das System seine Dienstgüte einhält. Außerdem ist er dabei einigen Einschränkungen unterlegen. So muss der Controller auch die Forderung nach Stabilität und Beherrschbarkeit des Systems erfüllen. Die Auswertung des Systemverhaltens und der SystemPerformance (δ) wird auf Basis der Dienstgüte (QoS) vorgenommen. Die QoS-Komponente misst, wie gut das System und seine Komponenten entsprechend ihrer Aufgabe operieren.

[9] Quality of Service

Anpassungsschleife (adaptation loop)

Die Anpassungsschleife erweitert die Rückkoppelungsschleife um zwei Komponenten: einen *Evaluator* und den bereits aus Adaptivregelungen bekannten *Controller Designer*. Die Aufgaben des *Evaluators* sind prinzipiell die gleichen wie die des *Model Estimators* aus der Adaptivregelung. Das bedeutet, er wertet das Verhalten und die Performance aus, um bestimmen zu können, ob das Modell des zu kontrollierenden Systems geeignet ist. Das Modell wird gegebenenfalls angepasst und eine Änderung des Kontrollgesetzes wird ausgelöst (wird vom *Controller Designer* übernommen).

Rekonfigurationsschleife (reconfiguration loop)

Hauptbestandteil der Rekonfigurationsschleife ist der *Reconfigurer*. Auf Anforderung des *Evaluators* kann dieser eine Rekonfiguration an dem gesamten System vornehmen. Diese Änderung schließt mit Ausnahme des *Reconfigurers*, der fix bleibt, alle anderen Komponenten ein: Strukturänderungen können am Modell des zu kontrollierenden Systems, am QoS-Subsystem, am Evaluator, Controller, Controller Designer und am Ziel (Goal) für den Controller vorgenommen. Da das zu kontrollierende System entgegen der üblichen Systemen kein mechanisches, physisches Objekt, sondern ein Software-System ist, ist selbst die Rekonfiguration dieses Systems möglich.

Für eine Rekonfiguration nutzt der *Reconfigurer* eine *Component Database* und eine *Specification Database*. Erstere enthält die Komponenten für alle rekonfigurierbaren Subsysteme, letztere die Spezifikation der Komponenten. Die Rekonfiguration muss nicht unbedingt durch den Austausch einer Komponente erfolgen, sondern kann auch durch eine Transformation oder eine Komposition erfolgen.

5.2.6. Fazit

Die verfügbaren Modelle der Regelungssysteme bieten konkrete Ansätze für adaptive Anwendungen. Innerhalb dieses Unterkapitels wurden sowohl der offene (Optimalwertsteuerung) als auch der geschlossene Regelkreis (Rückkoppelung) sowie Adaptivregelung und rekonfigurierbares Regelungssystem vorgestellt.

Besonders interessant ist der Ansatz von Kokar *et al.*, die die letzten drei genannten Ansätze zu einer Rückkoppelungssteuerung mit Adaptivregelung, bei der alle Systembestandteile rekonfiguriert werden können, integriert haben. Diese Ansatz bietet hohe Adaptivität. Allerdings muss nicht immer die Notwendigkeit nach einer adaptiven Regelung und der Möglichkeit zur Rekonfiguration des Regelungssystems gegeben sein.

Für viele adaptive Systeme reicht eine Rückkoppelungssteuerung (closed) oder gar eine Optimalwertsteuerung (open loop). Ohne die ge-

nauen Anforderungen zu kennen, kann keine Empfehlung abgegeben werden, welcher Ansatz zu verwenden ist.

Für adaptive Software, insbesondere für die mit dem Selbstmanagement (Self-X-Eigenschaften), werden vermehrt geschlossene Regelkreise als Lösung propagiert [Robertson und Brady 1999], [Oreizy *et al.* 1999], [Valetto und Kaiser 2002a], [Karsai *et al.* 2003], [Hinnelund 2004], [Taylor und Tofts 2004], [Laddaga und Robertson 2004].

Aber auch der offene Regelkreis hat seine Berechtigung. Dieses Modell findet bei Systemen Anwendung, für die gut vorhergesagt werden kann, welche Eingaben sie benötigen, um einen bestimmten Zustand oder eine bestimmte Ausgabe zu erreichen. Darunter fallen beispielsweise Systeme, die nach einer einfachen Ereignisbehandlungsschleife oder nach dem ECA[10]-Paradigma modelliert sind (vor allem genutzt in *Active Databases* [Act-Net Consortium 1996]). Derartige Systeme können regelbasiert arbeiten. Schwierig wird es hier jedoch, wenn entweder sich widersprechende Adaptionsregeln in Einklang zu bringen sind oder die Anzahl der Regeln zu viel wird.

Ein anschauliches Beispiel für sich widersprechende Regeln wird in [Zambrano *et al.* 2004] gegeben: Ein Benutzer ist über eine schnelle Netzwerkverbindung angeschlossen, hat jedoch ein Endgerät mit einem schlechten oder kleinen Display. Die Adaptionsregel für schnelle Netzwerkverbindungen würde das Bild in einer hohen Auflösung ausliefern wollen, weil die Dateigröße nicht beachtet werden muss. Jedoch besagt die Adaptionsregel für eingeschränkte Displays, dass das Bild runterskaliert werden muss. Für solche Konflikte muss es Mechanismen geben, die regeln, welcher Belang Vorrang hat.

Für dieses einfache Beispiel aus dem Bereich Pervasive Computing können entsprechende Vorrangregeln definiert werden. Ohne Zweifel ist dieser Ansatz jedoch beschränkt, denn je mehr Konstellationen in der Umgebung berücksichtigt werden müssen — oder die Umgebung ist nicht vollends vorhersagbar, z. B. für die Klasse der in Fußnote 8 auf Seite 133 genannten Typ-2-Systeme —, desto mehr Regeln müssen formuliert werden. In einem Szenario für Flugzeuge im Kampfeinsatz kann diese Anzahl z. B. auf imposante 20.000 Regeln steigen [Reece 2001]. Mit dieser großen Menge wird die Software für den Programmierer zu komplex und unbeherrschbar.

Genau hier setzen geschlossene Regelkreise an. Das System wird von einem externen Controller reguliert und rekonfiguriert, wobei dieser Rückkoppelung über eine Auswertungsfunktion bekommt. Für geschlossene Systeme besteht die Schwierigkeit in manchen Problembereichen jedoch darin, wie sich die Systeme selbst evaluieren sollen [Laddaga *et al.* 2001], [Laddaga und Robertson 2004]. Eine weitere Schwierigkeit ist der nicht-lineare Charakter von Software-Systemen, wenn kleine Änderungen eine große Variabilität im Ergebnis als Folge

[10] Event–Condition–Action

haben, der das Anwenden der Kontrollansätze komplexer gestaltet als bei Standard-Regelungssystemen [Laddaga 1999].

5.3. Reflection

Ein fundamentaler Ansatz, (selbst-) adaptive und sogar adaptierbare Software zu entwickeln, ist die Benutzung von Reflection [Robertson und Brady 1999], [Karsai und Sztipanovits 1999], [Czarnecki und Eisenecker 2000], [Robertson *et al.* 2001], [Meng 2001], [McGurren und Conroy 2002], [Landauer und Bellman 2003], [Robertson 2003], [Buschmann *et al.* 1998], [Fritsch und Renz 2004]. Die Grundlagen zu Reflection wurden ursprünglich in der Arbeit von Smith [1982] im Kontext von Programmiersprachen gelegt — bekannte, sog. dynamische Programmiersprachen, die Reflection unterstützen, sind beispielsweise Smalltalk, CLOS und in eingeschränkter Form Java[11].

Die Idee von Reflection ist es, die Software sich ihrer selbst bewusst zu machen[12] und ausgewählte Aspekte der Struktur und des Verhaltens für die Adaption und Veränderung zu erschließen. Mit einem Modell über sich selbst ausgestattet kann sich die Software an die dynamische Umgebung anpassen, indem es seine Operationen auf Basis des Zustandes der Umgebung und seiner eigenen Repräsentation »überdenkt« (vgl. [Robertson und Brady 1999], [Karsai und Sztipanovits 1999], [Czarnecki und Eisenecker 2000]). Der Reflection-Mechanismus ermöglicht es dabei, nicht nur den eigenen Zustand und die eigene Beschaffenheit in Erfahrung zu bringen (Introspection), sondern diesen auch zu ändern (Intercession).

Übersetzung und Bedeutung. In der vorliegenden Arbeit wird auf eine Übersetzung des englischen Begriffs *reflective/Reflection* verzichtet. Die in Wörterbüchern verfügbaren Übersetzungen, beispielsweise »reflektierend« und »spiegelartig«, suggerieren das Spiegeln eines Signals an einer Oberfläche, wobei Eintritts- und Austrittswinkel gleich sind. Diese Interpretation ist verwirrend. In die gleiche Richtung gehen Arbeiten wie beispielsweise [Ritter 2000], die »reflexiv/Reflexion« verwenden. All diese Formen sind nach Meinung des Autors unzureichend.

Die Beziehung zwischen Basis- und Metaebene lässt sich jedoch ganz gut mit »sich widerspiegeln« umschreiben, d. h. ein Abbild/Widerschein erzeugen. Die Erzeugung eines Widerscheins des Gegenstandes in einem Spiegel ist jedoch zu einseitig, da hier nur die

[11] Java bietet nur eingeschränkte Form von Reflection, weil es nur Introspection (Entdeckung von Methodennamen und Attributen von unbekannten Klassen zur Laufzeit) ermöglicht. Die Veränderung von Klassen und Methoden (Intercession) ist nicht realisierbar.

[12] Dafür existieren auch einige Self-X-Schlagwörter: self-awareness, self-modeling, self-representation, self-knowledge.

Abbildung des Realgegenstandes erzeugt wird (unidirektional). Reflection bezeichnet jedoch bidirektionale Abbildung/Sich-Widerspiegelung, d. h. Eigenschaften, Struktur ... der Basisebene *spiegeln sich* in der Metaebene *wider* (Lesen/Introspection) *und ebenso* die in der Metaebene vorgenommen Änderungen *spiegeln sich* in der Basisebene *wider* (Modifikation ausführen/Absorption). Maes [1987] hat für diese Verbundenheit von Meta- und Basisebene den Begriff »causally connected« geprägt: Änderungen in der Metaebene (der Selbstrepräsentation) wirken sich umgehend im darunterliegenden System (der Basiebene) aus, und umgekehrt. Um den Bogen zu schließen: Änderungen des Abbildes im Spiegel wirken sich nicht auf das Realobjekt aus. Doch gerade diese Fähigkeit wird durch Reflection erreicht.

Die Benutzung eines neuen deutschen Wortes »reflektiv«, wie in der deutschen Übersetzung des Architekturmuster-Buchs [Buschmann *et al.* 1998] geschehen, wird nicht vorgenommen, da dieses Wort nicht im Duden existiert [Duden 2001].

Nach Meinung des Autors sorgt das Belassen der englischen Form für weniger Verwirrung, da im Informatikbereich *Reflection* aus dynamischen Programmiersprachen gut bekannt ist — dennoch wird auf eine Definition des Begriffs nicht verzichtet.

Reflection-Terminologie. Eine sehr gute Definition für Reflection liefern Bobrow *et al.* [2003]:

> »Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution. There are two aspects of such manipulation: introspection and intercession.
>
> *Introspection* is the ability for a program to observe and therefore reason about its own state. *Intercession* is the ability for a program to modify its own execution state or alter its own interpretation or meaning. Both aspects require a mechanism for encoding execution state as data; providing such an encoding is called *reification*.«

In der Literatur sind neben diesen drei wichtigen Begriffen auch die folgenden zwei zu finden [Kon *et al.* 2002]:

> »*Reification.* The action of exposing the internal representation of a system in terms of programming entities that can be manipulated at runtime. The opposite process, *absorption* consists of effecting the changes made to reified entities into the system, thus realizing the causal connection link.«

Demnach bezeichnet Reflection die Fähigkeit eines Systems, sich selbst zu manipulieren. Eine Manipulation wird als Intercession oder Reification bezeichnet. Dazu ist es notwendig, die Beschaffenheit und den Zustand des Systemes für die Repräsentation in Erfahrung zu bringen (Introspection) bzw. die Änderung in der Repräsentation (Intercession) auf das System auswirken zu lassen (Absorption). Abbildung 5.8 fasst diese Formulierungen und die folgenden zusammen.

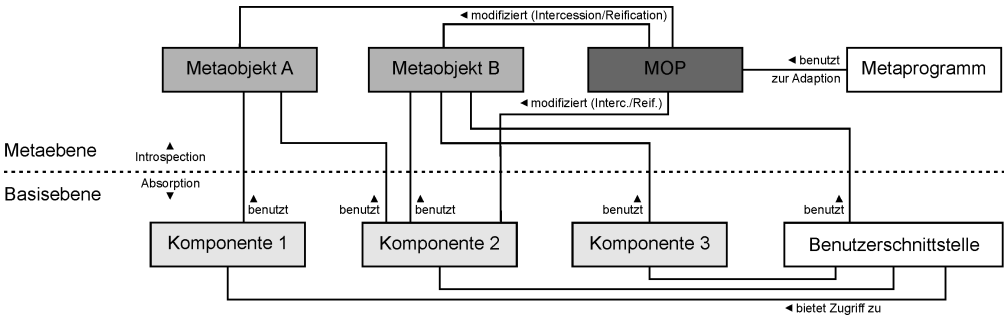

Abbildung 5.8.: Struktur einer Meta-Level-Architektur (in Anlehnung an [Buschmann *et al.* 1998])

Im Zusammenhang mit adaptierbaren Systemen wurde aus dem Reflection-Mechanismus zudem ein Architekturmuster formuliert, das die Erstellung von beliebiger adaptierbarer Software erlaubt — mit geeigneter Adaptionslogik werden diese Systeme zudem (selbst-) adaptiv. Die in Programmiersprachen angebotene Reflection ist dafür keine notwendige Voraussetzung, erleichtert jedoch die Erstellung [Czarnecki und Eisenecker 2000].

Im Folgenden soll Reflection aus Sicht der Architektur betrachtet werden. Danach wird auf repräsentative Umsetzungen im Middleware-Bereich eingegangen.

5.3.1. Meta-Level-Architektur

Das Architekturmuster *Reflection* [Buschmann *et al.* 1998] — auch bekannt als Open Implementation und Meta-Level Architecture — stellt einen Mechanismus für die dynamische Veränderung der Struktur und des Verhaltens von Software-Systemen bereit. Es unterstützt die Modifikation grundlegender Aspekte wie der Typstrukturen und des Mechanismus der Funktionsaufrufe.

Die Architektur eines Systems, das Reflection benutzt, ist in zwei wesentliche Teile aufgeteilt: eine Metaebene und eine Basisebene (Meta-Level und Base-Level; siehe Abbildung 5.8). Derartige Architekturen werden Meta-Level-Architekturen genannt. Die Basisebene definiert die Anwendungslogik, die Metaebene stellt Informationen über ausgewählte Systemeigenschaften zu Verfügung und macht damit eine Software sich ihrer selbst bewusst.

Metaebene

Die Metaebene stellt eine Selbstrepräsentation der Software bereit, um der Software Wissen über ihre eigene Struktur, ihr Verhalten oder ihren Zustand zu geben. Die Metaebene besteht aus mehreren so genannten *Metaobjekten* (siehe Abbildung 5.8). Jedes dieser Metaobjekte

kapselt ausgewählte Aspekte der Struktur, des Verhaltens oder des Zustandes der Basisebene.

Fast jede Systemeigenschaft kann auf diese Weise beschrieben werden. Beispiele sind Typstrukturen, Algorithmen oder sogar Mechanismen für den Aufruf von Funktionen. In einem verteilten System kann es beispielsweise Metaobjekte geben, die Information über die Position von Komponenten anbieten. Letztendlich hängt die Wahl, welche Aspekte ein Metaobjekt zu kapseln hat, davon ab, was adaptierbar sein soll. Nur Systemdetails, die sich wahrscheinlich ändern werden, oder die von Kunde zu Kunde variieren, sollten von Metaobjekten gekapselt werden.

Basisebene

In der Basisebene wird die Anwendungslogik modelliert und implementiert. Ihre Implementierung benutzt die Metaebene, um von denjenigen Aspekten, die sich wahrscheinlich ändern, unabhängig zu bleiben. Die Komponenten der Basisebene (Basiskomponenten) stellen die verschiedenen, vom System bereitgestellten Dienste dar sowie das zugrunde liegende Datenmodell. Die Basisebene spezifiziert darüber hinaus die grundlegende Zusammenarbeit und die strukturellen Beziehungen zwischen den Komponenten.

Komponenten auf der Basisebene sind entweder direkt mit den Metaobjekten verbunden, von denen sie abhängen, oder sie stellen Anforderungen an sie mit Hilfe spezieller Anfragefunktionen, die ebenfalls Teil der Metaebene sind. Die erste Art von Verbindung ist vorzuziehen, wenn die Beziehung zwischen Basisebene und Metaobjekt einigermaßen statisch ist. Die Komponente auf der Basisebene konsultiert immer das gleiche Metaobjekt, zum Beispiel wenn ein Objekt Typinformationen über sich selbst braucht. Die zweite Art von Verbindung wird benutzt, wenn die von der Basisebene benutzten Metaobjekte sich dynamisch ändern.

Metaobjektprotokoll (MOP)

Für die Handhabung der Metaobjekte in der Metaebene wird eine eigene Schnittstelle spezifiziert. Sie wird das *Metaobjektprotokoll* (MOP) genannt. Diese Schnittstelle dient für die Modifikation der Basisebene. Das Metaobjektprotokoll erlaubt es Klienten, besondere Veränderungen wie die Änderung des Metaobjektes für den Funktionsaufrufmechanismus zu spezifizieren. Klienten des Metaobjektprotokolls können Komponenten auf der Basisebene oder andere Anwendungen sein. Programme, die das MOP für die Manipulation anderer oder des eigenenen Programms benutzen, werden Metaprogramme genannt [Karsai und Sztipanovits 1999], [Czarnecki und Eisenecker 2000]. Jede Manipulation von Metaobjekten mittels des Metaobjektprotokolls beeinflusst das Verhalten der Basisebene.

5.3.2. Reflective Middleware

Die im vorhergehenden Abschnitt formulierten Konzepte für Meta-Level-Architekturen erlauben die Erstellung von beliebigen adaptierbaren und — bei geeigneter Auswertungslogik zur Laufzeit — adaptiven Systemen. Im Folgenden werden zwei Middleware-Ansätze, OpenORB und K-Components, vorgestellt, die dieses Muster implementieren. Sie sollen helfen, dieses Muster zu verstehen. Die Wahl fiel auf die zwei vorgenannten Middleware-Ansätze, da bei ihnen die Umsetzung des Reflection-Ansatzes sehr gut zu sehen ist, sie die Operationen zur Adaption auf der Architekturebene ermöglichen und zudem mit Adaptionslogik ausgestattet sind (sie sind nicht nur adaptierbar, sondern dadurch auch adaptiv). Andere Beispiele für die Umsetzung des Reflection-Musters liefern die Literaturquellen eingangs dieses Unterkapitels und — noch speziell aus dem Middleware-Bereich — dynamicTAO [Kon *et al.* 2000] und FORMAware [Moreira 2003], [Moreira und Blair 2004].

Middleware verbindet die Interaktion zwischen Anwendung und Betriebssystem in verteilten Systemen. Normalerweise ist es Aufgabe der Middleware, Heterogenität zu verbergen, d. h. Middleware versteckt die Details der unterliegenden Schichten und betriebssystemspezifischen Schnittstellen [Coulouris *et al.* 2002]. Dies erlaubt es Programmierern von verteilten Applikationen, Code zu schreiben, der wie Code von zentralisierten (d. h. nicht verteilten) Anwendungen aussieht. Die Middleware erweitert den Aufruf von Methoden um Netzwerkaspekte, Marshalling und Scheduling. Dadurch können Anwendungen einfach portiert werden, ohne dass sich der Programmierer um die Interna der Middleware oder des Betriebssystems kümmern muss. Die meisten Anwendungen profitieren ganz klar von Middleware, die die Details der unterliegenden Schichten versteckt.

Für Anwendungen in einem dynamischen Umfeld, d. h. Anwendungen, die adaptives Verhalten zeigen müssen, ist das Verbergen der Heterogenität in der Ablaufumgebung eher von Nachteil. Beispielsweise können Multimedia-Streaming- oder Videokonferenzanwendungen ihre Dienstgüte (QoS) dramatisch verbessern, wenn sie ein Transportprotokoll auswählen, das für die unterliegende Infrastruktur am besten geeignet ist. Außerdem kann eine derartige Anwendung auch profitieren, wenn sie über den physikalischen Kontext Bescheid weiß. Zum Beispiel wenn ein größeres Display verfügbar ist, kann sich die Anwendung rekonfigurieren und das Video auf dem größeren Display anzeigen. Middleware, die nicht die Details der unterliegenden Schichten und der Umgebung versteckt, wird als *reflective Middleware* bezeichnet [Kon *et al.* 2002].

Im Folgenden werden zwei Vertreter von reflective Middleware vorgestellt und anschließend bewertet.

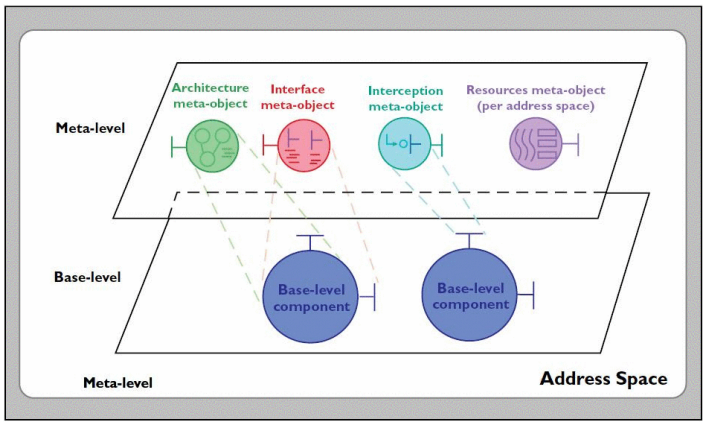

Abbildung 5.9.: Meta- und Basisebene in Open ORB [Kon *et al.* 2002]

Open ORB

Open ORB[13] [Blair *et al.* 2000a, 2002] ist eine an der Lancaster Universität, England, entwickelte Middlewarelösung, die Komponententechnologie — in diesem Falle den CORBA-Standard — und Reflection vereint, um eine rekonfigurierbare Plattform für verteilte Anwendungen zu schaffen.

Modelle der Metaebene. Ein Hauptaspekt von Open ORB ist die Aufteilung der Metaebene in vier getrennte Meta-Modelle respektive Metaobjekte für die Adaption: Architektur-, Schnittstellen-, Interception- und Ressourcen-Metaobjekt (siehe Abbildung 5.9). Die ersten zwei Metaobjekte ermöglichen strukturelle Reflection, die anderen zwei Reflection für das Verhalten (behavioural reflection). Im Kontext der vorliegenden Arbeit ist nur die strukturelle Reflection von Interesse. Der Vollständigkeit halber und da das Reflection-Muster so fundamental und allgemeingültig für beliebig adaptierbare/adaptive Software-Systeme ist, werden alle vier Metaobjekte beschrieben:

Architektur-Metaobjekt Das Architektur-Metaobjekt bietet Zugriff zu der Software-Architektur des Systems. Die Selbstrepräsentation besteht aus zwei Teilen. Ein Komponentengraph repräsentiert

[13] Die genaue Schreibweise, *OpenORB* vs. *Open ORB*, ist nicht einheitlich. Auf der Website des Projektes `http://www.comp.lancs.ac.uk/computing/research/mpg/reflection/` und auch im Titel des referenzierten Papers [Blair *et al.* 2002] wird *OpenORB* verwendet, im Text jedoch *Open ORB*. Verwirrenderweise gibt es ein gleichnamiges Projekt auf Sourceforge `http://openorb.sourceforge.net`, das zwar auch eine den CORBA-Standard implementierende Middlewareplattform ist, jedoch hat diese nichts mit dem hier vorgestellten Projekt zu tun. Im Folgenden wird deshalb — neben dem Hinweis auf den gleichen Namen und auf die nicht konsistente Verwendung einer Schreibweise für ihr Projekt — die Schreibweise *Open ORB* verwendet.

die Verbindungen zwischen den Komponenten. Neben der Änderung des Interaktionsmodells für die Bindung zwischen den Komponenten (z. B. RMI[14], Publish/Subscribe, Gruppenkommunikation) wird auch das Hinzufügen, Entfernen und Austauschen von Komponenten durch das MOP[15] unterstützt [Kon *et al.* 2002]. Der andere Teil der Selbstrepräsentation validiert das Architekturmodell mithilfe von Randbedingungen (architectural constraints, vergleichbar mit den in Kap. 5.1.1 auf Seite 128 vorgestellten ACLs).

Schnittstellen-Metaobjekt Das Schnittstellen-Metaobjekt bietet Zugang zu der externen Repräsentation einer Komponente. Ähnlich der Introspection bei Programmiersprachen (vgl. Termini in der Reflection-Definition) können provides- und requires-Schnittstellen einer Komponente in Erfahrung gebracht werden. Dadurch ist es möglich, mit dynamisch entdeckten Komponenten zu interagieren.

Interception-Metaobjekt Das Metamodell des Interception-Metaobjektes ermöglicht das dynamische Einfügen von Interceptoren. Interceptoren werden typischerweise vor oder nach dem Aufruf einer Funktion aktiv. Dieser Mechanismus ist nützlich, um beispielsweise Monitoring- oder Accounting-Funktionalität dynamisch in das System einzufügen.

Ressourcen-Metaobjekt Für Anwendungen mit QoS-Anforderungen gibt es das Ressourcen-Metaobjekt. Dieses bietet Zugang zu den Ressourcen (-management) der unterliegenden Plattform. Dadurch können Ressourcen wie Speicher und Threads inspiziert und rekonfiguriert werden, indem beispielsweise Parameter oder Algorithmen für das Management geändert werden oder Ressourcen Tasks zugeteilt oder abgezogen werden.

Die getrennten Metamodelle bieten eine Trennung der Belange (SoC) [Hürsch und Lopes 1995], wodurch auch die Komplexität der Schnittstelle der Metaebene (MOP) reduziert wird.

Rekonfiguration. Die sichere Rekonfiguration von Open ORB geschieht auf Basis des Komponentengraphen sowie architektonischer Randbedingungen (architectural constraints; vgl. ACLs auf Seite 128) und wird in [Blair *et al.* 2000b] näher beschrieben. Kurz formuliert wird nach Ermittlung der neuen Middleware-Konfiguration versucht, auf einen Ruhezustand oder auf einen sicheren Zustand[16] zu warten und bei Erreichen die Rekonfiguration durchzuführen.

[14] Remote Method Invocation

[15] Metaobjektprotokoll

[16] In einem sicheren Zustand (*safe state*) gilt die Zusicherung, dass alle ausstehenden Anfragen letztendlich werden und ihre nicht-funktionalen Eigenschaften erfüllt werden. Als andere Möglichkeit nennen Blair *et al.* den Ruhezustand (*idle state*), bei dem für alle ausstehenden Anfragen zugesichert wird, dass die neue

Adaptionslogik. Blair *et al.* nennen zwei Ansätze, die für die Adaptionslogik vorstellbar sind:

- Anwendungen oder Systemdienste ermöglichen das Monitoring und die Adaption als externen Dienst oder
- Managementkomponenten für das Monitoring und die Adaption können in die Metaebene eingefügt werden, um einen solchen Dienst anzubieten.

In Open ORB wird der letztere Ansatz verwendet.

Für das Monitoring gibt es Event Collectoren und Monitore. *Event Collectoren* beobachten das Verhalten der unterliegenden funktionalen Komponenten und generieren relavante QoS-Ereignisse. *Monitore* sammeln QoS-Ereignisse und melden abnormales Verhalten an Interessierte.[17]

Für die Adaption werden zwei verantwortliche Managementkomponenten genannt: Strategy Selector und Strategy Activator. Ein *Strategy Selector* wählt eine passende Adaptionsstrategie (d. h. einen Strategy Activator) basierend auf dem Feedback der Monitore aus. *Strategy Activators* implementieren eine bestimmte Adaptionsstrategie (z. B. die Manipulation des Komponentengraphen).

Die Regeln für das Monitoring und die Auswahl einer Strategie werden in einem Zeitautomaten (timed automata) [Lynch und Vaandrager 1992] formuliert. Die Managementkomponenten agieren dabei als Zeitautomaten-Interpreter, die über Ereignisbenachrichtigung (Registrieren für, Empfangen von, Reagieren auf sowie Aussenden von Ereignissen; wie in »Reactive Objects« [Manna und Pnueli 1992]). Durch die Verwendung eines Automaten können die Regeln formal analysiert werden.

K-Components

K-Components [Dowling und Cahill 2001a, b] ist ein Komponentenmodell für die Erstellung von adaptiven Systemen mittels »architectural reflection« (architektonische Reflection) [Tisato *et al.* 2001]. Das System ist gemäß dem Reflection-Muster in eine Basis- und in eine Metaebene aufgeteilt. Ein Konfigurationsmanager, der als Metaobjekt in der Metaebene fungiert, ist zuständig für die Speicherung und für das Management der Architekturkonfiguration (siehe Abbildung 5.10 auf der nächsten Seite).

Konfiguration die nicht-funktionalen Anforderungen für diese Anfragen erfüllt. Ein triviale Vorbedingung für diesen Fall ist, dass keine Anfragen ausstehen [Blair *et al.* 2000b].

[17] Die Vermutung des Autors ist, dass die Beschreibung der Aufgaben von Monitoren und Event Collectoren in [Blair *et al.* 2002] vertauscht worden ist.

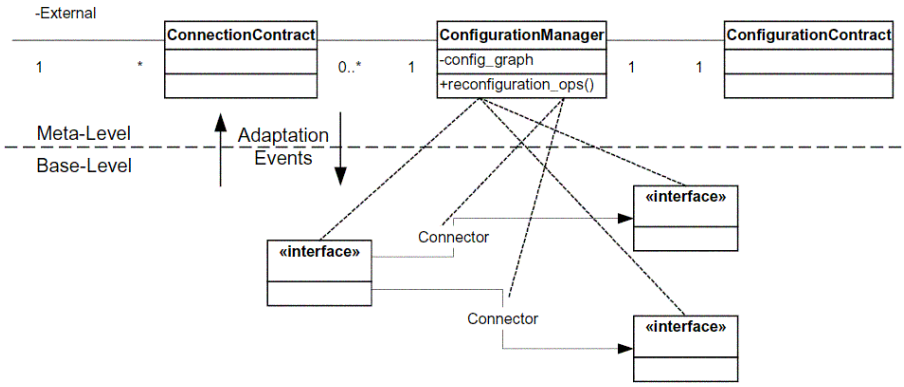

Abbildung 5.10.: Aufbau von K-Components nach dem Reflection-Muster mit dem Konfigurationsmanager und der Adaptionslogik in der Metaebene und der zu konfigurierenden Software-Architektur in der Basisebene [Dowling und Cahill 2001b]

Konfigurationsgraph. Das Architekturmodell (Metamodell der Software-Architektur) ist ein typisierter Graph (typed, connected graph), in dem die Knoten die Schnittstellen repräsentieren, denen Komponenten zugeordnet sind. Kanten sind Konnektoren, die mit Parametern versehen sind. Die Parameter eines Konnektors verdeutlichen beispielsweise das dem Konnektor unterliegende Kommunikationsprotokoll oder die Anzahl der installierten Interceptoren. Bei K-Components sind die »Konnektorstrategien« Lokal, Entfernt, Komprimierung und Interceptor möglich.

Die Spezifikation des Konfigurationsgraphs für die Architektur ist nicht in Form einer ADL notwendig, sondern wird automatisch aus den Komponentendefinitionen, in Form der IDL-3, generiert (*architectural reconstruction* [SEI 2004a]).

Die Rekonfigurationen der Architektur wird als transaktionale Transformation des Konfigurationsgraphen vorgenommen (vgl. Abbildung 5.11 auf der nächsten Seite). Da Graph-Transformationen sicherstellen, dass das Resultat wieder ein Graph ist, kann zusammen mit dem transaktionalen Vorgehen die Wahrung der Konsistenz und der Integrität des Systems gewährleistet werden. K-Components übernimmt hier den transaktionalen Ansatz von Wermelinger [1999], bei dem an der Rekonfiguration unbeteiligte Komponenten weiterhin operieren können, betroffene Komponenten und Konnektoren werden »eingefroren« (vgl. Abschnitt 5.1.1 auf Seite 129).

Die Struktur des Graphen bleibt bei der Konfiguration jedoch statisch, d. h. das Komponentenmodell von K-Components unterstützt in dieser Form kein Hinzufügen oder Entfernen von Komponenten, sondern nur den Austausch von Komponenten und die Änderung der »Konnektorstrategie« (Lokal, Entfernt, Komprimierung und Interceptor).

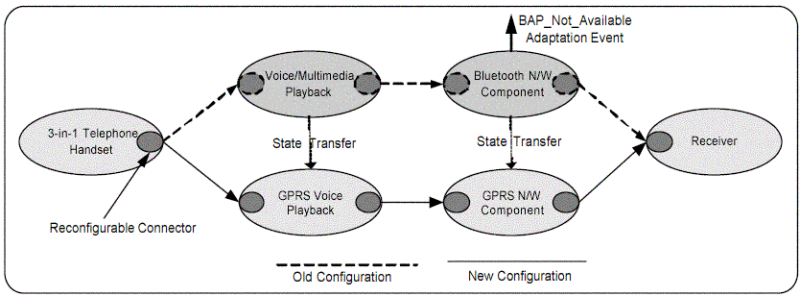

Abbildung 5.11.: Dynamische Rekonfiguration einer 3-in-1-Telefonanwendung vom Bluetooth- zum GPRS-Profil (BAP = Bluetooth Access Point) [Dowling und Cahill 2001a]

Adaptation Events. Für das Auslösen einer Adaption gibt es sog. *adaptation events*. Dies sind Informationen, die von der Konfiguration in der Metaebene und von den Komponenten und Konnektoren in der Basisebene abgefragt werden können.

Adaptionslogik. Das adaptive Verhalten der Architektur wird in einer separaten ACDL[18] angegeben. Diese Regeln bestimmen, wie und wann eine Transformation des Konfigurationsgraphen vorgenommen werden soll. Die in den Regeln spezifizierten Operationen werden auf den Konfigurationsgraphen angewandt, was wiederum in einer Modifikation der unterliegenden Software-Architektur resultiert — und damit Reflection, genauer Intercession, entspricht.

Für die Berücksichtigung von eingehenden und ausgehenden Abhängigkeiten[19] sind die Adaptionsregeln (adaptation contracts) aufgeteilt: Konfigurationskontrakte (*configuration contracts*) verwalten interne Ressourcen und eingehende Abhängigkeiten von Klienten, Verbindungskontrakte (*connection contracts*) sind für ausgehende Abhängigkeiten verantwortlich. Damit erfolgt eine Berücksichtigung der Sicherheit und Integrität sowohl des zu adaptierenden Systems als auch aller anderen Clients, die von dem zu adaptierenden System abhängig sind.

In beiden Kontraktarten wird die Adaptionslogik in ACDL spezifiziert. Der Aufbau eines Kontrakts erfolgt bei beiden nach dem gleichen Muster: Es gibt mehrere bedingte Ausdrücke mit Rekonfigurationsoperationen. Diese werden bei bestimmten Adaptation Events ausgelöst. Bei den Konfigurationskontrakten gibt es zusätzlich noch die Unterstützung, die vom System abhängigen Clients über das Auftreten einer Rekonfiguration zu benachrichtigen.

[18] Adaptation Contract Definition Language

[19] Eingehende Abhängigkeiten entstehen durch andere Systeme oder Nutzer, die von dem System abhängig sind. Ausgehende Abhängigkeiten treten auf, wenn das System von anderen Sytemen abhängig ist. Vergleichend mit den Schnittstellen bei Komponenten könnte man die Parallelen eingehend/provides und ausgehend/requires ziehen.

Die Trennung der Adaptionsregeln von denen des normalen Programmes nach der Philosophie der Konfigrationsprogrammierung (vgl. Seite 124 und [Kramer und Magee 1998]) in Verbindung mit den separaten Adaptation Events hat zur Folge, dass keine Abhängigkeiten zwischen den Adaptionskontrakten auf der Metaebene und den Anwendungskomponenten auf der Basisebene bestehen. Dies erlaubt die Änderung der Regeln zur Laufzeit (offen-adaptives System).

Management. Verantwortlich für das komplette Management der Rekonfiguration ist der Konfigurationsmanager, der in der Metaebene fungiert. Er übernimmt nicht nur die Rekonfiguration der Architektur durch Umschreiben des Architekturgraphen, sondern er ist auch ein Container für das Deployment, die Koordinierung sowie die Ausführung der Adaptionsregeln.

Bewertung

Innerhalb dieses Abschnittes werden mit Tabelle 5.1 auf der nächsten Seite die beiden Middleware-Ansätze einem Vergleich unterzogen, wobei folgende Kriterien aufgestellt wurden:

Strukturierung der Metaebene Wie gut ist die Metaebene in die Metaobjekte aufgeteilt, die die verschiedenen Belange für eine Adaption verwalten, und wird die gesonderte Nutzung des MOP ersichtlich?

sichere Rekonfiguration der SA Berücksichtigt der Ansatz die sichere Rekonfiguration der SA zur Laufzeit?

Unterstützte A.-Operationen Welche Adaptionsoperationen werden unterstützt

- auf Ebene der SA,
- andere?

Adaptionslogik Wie ist die Adaptionslogik realisiert und welche Unterstützung besteht für Versorgung mit (Kontext-) Informationen?

offen-adaptiv Ermöglicht die Middleware-Lösung die Änderung der Adaptionsregeln zur Laufzeit (offen-adaptiv)?

5.3.3. Fazit

Reflection ist ein fundamentales Konzept für adaptierbare *und* für adaptive Systeme, da es zum einen Systemen ermöglicht, Wissen über sich selbst offenzulegen, zum anderen erlaubt es die eigene Manipulation.

Kriterium	*Open ORB (O)*	*K-Components (K)*
Strukturierung Metaebene	++ jeder Belang wird einem separaten Metaobjekt zugeteilt; ermöglicht die einfache Handhabung und Erweiterung der Metaebene, ohne die Schnittstelle des MOP aufzublähen	−− Adaption der SA (Austausch der Komponenten) *und* Änderung der Parameter eines Konnektors (das dem Konnektor unterliegende Kommunikationsprotokoll und Konnektorstrategie) werden nur von *einem* Metaobjekt verwaltet
sichere Rekonfiguration	o Rekonfiguration auf Basis eines Komponentengraphen (alter wird durch neuen ersetzt) mit architektonischen Randbedingungen; für eine Rekonfiguration muss ein Ruhezustand oder sicherer Zustand erreicht werden	++ realisiert durch Graph-Transformation, wodurch die Integrität sichergestellt wird, Rekonfiguration wird auf transaktionaler Basis (mit »Einfrieren« der von der Rekonfiguration betroffenen Komponenten und Konnektoren) vorgenommen; Berücksichtigung von eingehende Abhängigkeiten (Clients) durch Konfigurationskontrakte
unterstützte Adaptionsmechanismen		
▷ Ebene SA	+ MOP unterstützt das Hinzufügen, Entfernen und Austauschen von Komponenten, Einfügen von Interceptoren (bei K. als Konnektorparameter realisiert) sowie Introspection für Komponentenschnittstelle	o MOP unterstützt nur den Austausch von Komponenten und die Änderung der Parameter eines Konnektors; Konzept würde auch Hinzufügen und Entfernen von Komponenten unterstützen
▷ andere	+ Rekonfiguration der Ressourcen	− keine
Adaptionslogik	+ Regeln sind in einem Zeitautomaten formuliert; erlauben formale Analyse	+ Rekonfiguration wird durch Ereignisse angestoßen
offen-adaptiv	keine Aussage	ja
Sonstiges		nur Intercession zur Laufzeit; Introspection nur bei Start, um Architekturgraph zu konstruieren

Tabelle 5.1.: Vergleich von Open ORB und K-Components (mit Skaleneinteilung −−, −, o, +, ++)

Ein System, das das Reflection-Muster umsetzt, ist in zwei Ebenen aufgeteilt (Meta-Level-Architecture): die Basisebene und die Metaebene. In der Basisebene werden die Komponenten der Anwendung vorgehalten. Die Metaebene bietet mit den Metaobjekten über das MOP Zugriff auf die Repräsentation der Struktur und des Verhaltens der unterliegenden Anwendung. Dieser Aufbau erlaubt die Entwicklung von adaptierbaren Systemen. Wird die Metaebene zusätzlich um Adaptionslogik erweitert (und Monitoring der Umwelt), dann entsteht ein System mit adaptiven Fähigkeiten. Dabei wird die Adaptionslogik mit Informationen über das System (Introspection, mittels der Metaobjekte) und die Umwelt versorgt (oder ruft diese ab) und verändert bei entsprechend erfüllter Bedingung über das MOP das Modell des unterliegenden Systemes. Die vorgenommenen Änderungen an den Metaobjekten werden umgehend auf das System in der Basisebene umgesetzt (Absorption).

Die beiden Middleware-Ansätze Open ORB haben die konkrete Anwendung des Reflection-Musters gezeigt. Bei ihnen bieten Metaobjekte Zugriff auf die Architektur, Schnittstellen oder die Ressourcenverwaltung des unterliegenden Systems. Bei beiden Ansätzen wird die Architektur in einem Graphen vorgehalten, der die jeweilige Middleware-Konfiguration repräsentiert.

Zusammenhang MLP/CP Genau hier ergibt sich der Zusammenhang zwischen Reflection bzw. Meta-Level-Architekturen (Meta-Level Programming, MLP) zu dem externen Ansatz mithilfe der Konfigurationsprogrammierung (CP) mit den ADLs als Modell des unterliegenden Systemes, die innerhalb des ersten Unterkapitels behandelt wurden (5.1). Besonders deutlich wird dies bei K-Components, bei dem der Konfigurationsmanager als Metaobjekt in der Metaebene platziert ist. Die ADL repräsentiert dabei das System, der Konfigurationsmanager besitzt eine Schnittstelle (das MOP) für die Veränderung der Architektur des Systemes.

Das Konzept der Metaobjekte, die mit dem Wissen über die Struktur, des Verhaltens und des Zustandes des Systems ausgestattet sind, kann sogar noch fortgeführt werden. In der Beschreibung zu den Metaobjekten wurde bereits formuliert, dass Metaobjekte fast jede beliebige Systemeigenschaft kapseln können, unter anderem auch beispielsweise Informationen zu der Position von Komponenten. Damit ergibt sich ein Zusammenhang zu den Konnektoren in ADLs. Komponenten sind dafür zuständig, die Funktionalität der Anwendung zu implementieren und Zustandsinformationen zu speichern. Konnektoren sind Dienste für den Transport und das Routing von Nachrichten und Objekten. Wie die Eingaben den Komponenten zugestellt und Ausgaben wegtransportiert werden oder was ihre Quellen oder Ziele sind, dass wissen die Komponenten nicht. Dahingegen wissen Konnektoren ganz genau, wer mit wem kommuniziert, da sie der »glue« (Klebstoff)

MLP *Abstraktion*		ADL/CP *Abstraktion*
Komponenten der Basisebene	↔	Komponenten, Module
Reification Points	↔	Ports
Komponenten der Metaebene	↔	Komponenten

Tabelle 5.2.: Übereinstimmung der Abstraktion von MLP und ADL/CP [Loques *et al.* 2000]

in der Architektur sind — aber sie kümmern sich nicht um die Verarbeitung in den Komponenten. Deshalb können Konnektoren mit Metaobjekten in gewisser Weise gleichgesetzt werden. Loques *et al.* haben die Übereinstimmung von Meta-Level Programming (MLP) und ADLs/Konfigurationsprogrammierung (CP) in Tabelle 5.2 zusammengefasst.

5.4. Muster für die Adaptivität in Software

Innerhalb dieses Abschnittes sollen einige weitere Muster vorgestellt werden, die für die Entwicklung von adaptiven Systemen besonders geeignet sind.

In [Oreizy *et al.* 1998] werden einige Muster genannt, deren Einsatz sich bei der Rekonfiguration von Architekturen (hier das Hinzufügen von Komponenten) besonders eignen. Als erstes ist das Observer-Muster [GoF 1996] zu nennen. Dieses Muster separiert die Informationsanbieter von den -beobachtern explizit. Dadurch ist das Hinzufügen von neuen Beobachtern mit minimaler Auswirkung auf das restliche System möglich. Im Kontext der vorliegenden Arbeit eignet sich das Observer-Muster sowohl für das Hinzufügen von Monitoren zu den zu beobachtenden Elementen als auch für die Umsetzung, Anwendungskomponenten zu einem System hinzufügen zu können. Für die lose Koppelung von Komponenten nennen Oreizy *et al.* das Mediator (Vermittler)-Muster [GoF 1996] besonders, da durch die Nutzung eines Vermittlers die Komponenten davon abgehalten werden, aufeinander explizit Bezug zu nehmen. So kann das Zusammenspiel der Komponenten unabhängig variiert werden.

In der vorliegenden Arbeit wurde bereits das Adapter-Muster im Rahmen der Schnittstellenanpassung genannt (Seite 102). Auch im Rahmen der Adaptionsmechanismen wurde das Proxy-Muster genannt, das bei der Migration einer Komponente am alten Ort platziert wird und die Dienstaufrufe an die migrierte Komponente weiterleitet. In FarGo [Holder *et al.* 1999b] wird das Proxy-Muster außerdem dafür verwendet, syntaktische und (teilweise) semantische Transparenz für Remote-Referenzen zu erhalten. Eine Komponente in FarGo hat nur eine lokale Referenz auf das Proxy-Objekt, das mithile eines sog. Trackers die Migration der referenzierten Komponente überwacht und

die Referenz anpasst.[20] Für die Sicherung des Zustandes einer Komponente ist das Memento-Muster geeignet besonders geeignet (für andere Ansätze siehe [Kasten und McKinley 2004] und [Chen *et al.* 2001]).

Neben diesen sehr geläufigen Mustern, die in der Softwareentwicklung ihren festen Platz gefunden haben, sind im Zusammenhang mit selbst-adaptiver Systeme zudem einige spezielle Muster zu nennen. Sie werden nachfolgend näher beschrieben (in Anlehnung an [Laddaga *et al.* 2003], erweitert um Referenzen zu erweiterter Literatur).

5.4.1. Blackboards für die Verschmelzung von Signalen

Viele adaptive Systeme müssen Sensordaten verarbeiten, um aus den Messungen Interpretationen ableiten zu können. Adaptivität benötigt in diesem Umfeld verschiedenartige Wissensquellen, angefangen bei Wissen, das gefiltert wurde, weiter zu Wissen über die Semantik der Domäne bis Context-Awareness.

Blackboards [Laddaga *et al.* 2003], [Buschmann *et al.* 1998] wurden im Umfeld der KI entwickelt und sind eine Erweiterung des Publish-and-Subscribe-Musters (letzteres auch bekannt als Observer-Muster [GoF 1996]). Ein Blackboard erlaubt eine Vielfalt an verschiedenen Wissensquellen (sog. »Knowledge Sources«) auf verschiedenen Abstraktionsstufen, um ein Problem kooperativ lösen zu können (siehe Abbildung 5.12 auf der nächsten Seite). Ein Blackboard besteht aus drei Hauptkomponenten: den Wissensquellen, dem Blackboard und einer Auswertungseinrichtung/Evaluator (vgl. Abbildung 5.13 auf der nächsten Seite. Die *Wissensquellen* bearbeiten das Problem unabhängig voneinander und veröffentlichen die Ergebnisse in einem gemeinsamen Speicher, dem *Blackboard*. Dieser Prozess kann über mehrere Stufen erfolgen, wobei mit jeder Stufe die Abstraktionsstufe von den Rohdaten zunimmt. Sobald der Abstrahierungsprozess die höchste Stufe erreicht hat, übernimmt eine *Auswertungseinrichtung* (Evaluator) die Bewertung jedes Ergebnisses vor und gibt Feedback zurück. Das Feedback wird über mehrere Domains an die Wissensquellen zurückpropagiert, die an dem Prozess beteiligt waren (vgl. [Seth und Kokar 2003]).

Beispielsysteme. Beispiele für adaptive Systeme, die das Blackboard-Muster benutzen, sind in [Seth und Kokar 2003] und [Soto und Khosla 2003] zu finden. Die Anwendung in [Seth und Kokar 2003] ist die bereits in Fußnote 34 auf Seite 54 kurz angerissene Rechtschreibprüfung. Hier nehmen die verschiedenen Wissensquellen parallel eine Analyse des falsch geschriebenen Wortes (Buchstabendreher, Buchstabendoppler, Zeichenentferner etc.) vor und melden die verschieden möglichen

[20] FarGo bietet außerdem adaptive Fähigkeiten. Auslöser sind Regeln, die zur Entwurfszeit oder nachträglich formuliert werden und Monitoringinformationen nutzen. Als Adaptionsoperationen können die Beziehungen zwischen den Komponenten geregelt werden (link, pull, duplicate, stamp) [Holder *et al.* 1999a].

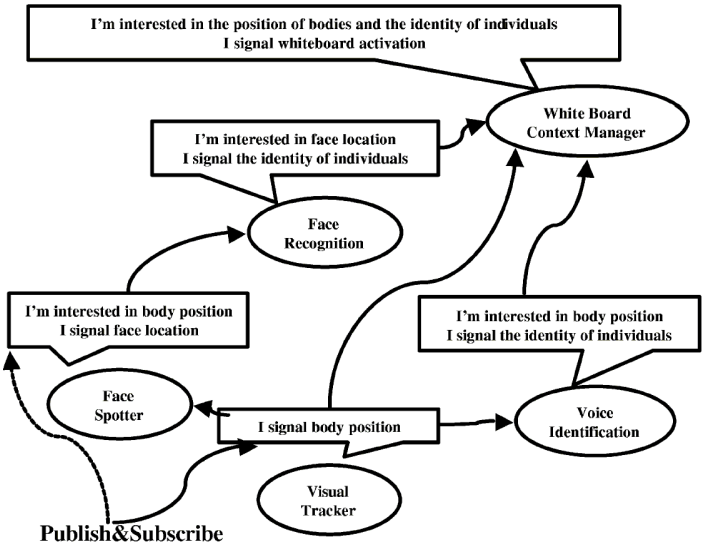

Abbildung 5.12.: Beispiel-Blackboard für die Verschmelzung von Signalen [Laddaga *et al.* 2003]

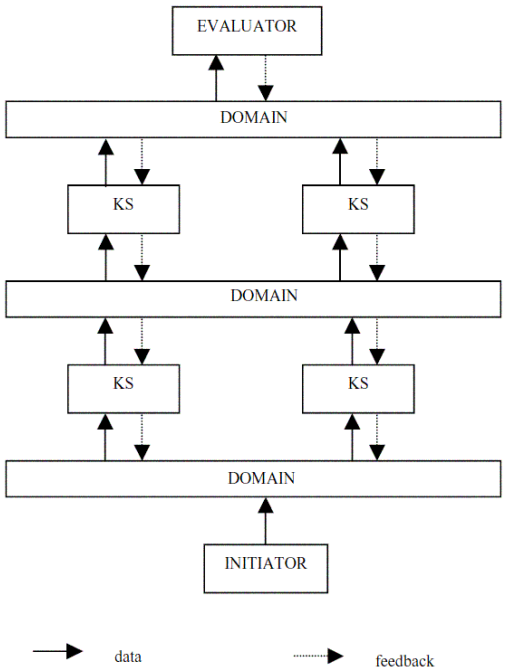

Abbildung 5.13.: Modell für das Blackboard mit mehreren Wissensquellen (Knowledge Sources), einer Auswertungseinrichtung (Evaluator) und dem Blackboard, das als gemeinsamer Datenspeicher für die Kommunikation benutzt wird [Seth und Kokar 2003]

Ergebnisse an die höhere Ebene (in dem Fall die Auswertungseinrichtung), damit diese mithilfe eines Wörtbuchvergleiches zu einem vernünftigeren Ergebnis gelangt. Im Gegensatz zu dem nachfolgend beschriebenen Ansatz wird nur eine Stufe für die Analyse benutzt.

In [Soto und Khosla 2003] wird das Blackboard in einer erweiterten Form bei einen mobilen Roboter für die Zielverfolgung und das Erkennen von Hindernissen benutzt. Adaptivität ist hier in dem Maße erforderlich, dass der Roboter den Prozess für das Erlangen von Wissen an die dynamische Umgebung anpassen muss. Als Motivationsbeispiel nennen sie das System ALVINN [Pomerleau 1993] — ein visuelles Wahrnehmungssystem, um ein Auto in natürlicher Umgebung mithilfe von neuralen Netzlernalgorithmen zu manövrieren. Dieses System hat nach einem Training die Straßenbegrenzung als auschlaggebend für eine Erkennung identifiziert. Damit erreichte es eine beachtliche Performance. Aber als die Straßenbegrenzung von anderen passierenden Autos verdeckt wurde oder auf Brücken oder Straßenkreuzungen nicht existent war, hat das System unheilbar versagt. Das Hauptproblem von ALVINN ist seine fehlende Fähigkeit (fehlende Adaptivität), alternative Informationsquellen wie den Mittelstreifen, anderen Verkehr, Verkehrszeichen etc. zu benutzen. Das in [Soto und Khosla 2003] beschriebene System behebt dieses Manko durch die Benutzung eines Blackboards und einem »intelligent agent paradigm«. Für Details wird auf vorgenannte Literatur verwiesen.

5.4.2. Indirekter Aufruf

So gut wie jede selbst-adaptive Software beruht auf der Entkopplung des Aufrufs eines Dienst und der Methode, die den Dienst anbietet. Dieses Entkopplung erlaubt es, zur Laufzeit entscheiden zu können, wie der angeforderte Dienst am besten erbracht werden kann. Im Gegensatz zu dem standardmäßigen Programmiermodell, wo der Aufrufer den Aufgerufenen direkt per Prozeduraufruf einbezieht, erlauben Techniken für den indirekten Aufruf eine späte Bindung von Aufrufer und Aufgerufenem. Nachfolgend werde einige dieser Techniken aufgeführt.

Method Dispatch Diese Technik bildet den Kern von praktisch allen objektorientierten Programmiersystemen. Beim Aufruf einer Methode wird ihre Signatur geprüft, d. h. die Wahl der Methode wird anhand der Datentypen der Argumente getätigt.

Pattern-directed Invocation Dieses Verfahren ist bei dem regelbasierten Programmieren üblich. Die Prozeduren werden anhand eines Musters bekanntgegeben, wobei das Muster Variablen aus dem Prozedurrumpf einschließt. Für das Vorgehen gibt es zwei Arten: Vorwärts- und Rückwärtsverkettung. Beim Vorwärtsverketten wird eine Datenbank mit Aussagen (assertions) dahingehend beobachtet, ob eine Menge von Aussagen dem Muster einer Regel

entsprechen; der Rumpf einer Regel wird ausgeführt, wenn nach dem Hinzufügen einer neuen Aussage zur Menge der bestehenden Aussagen das Muster einer Regel erfüllt wird. Beim Rückwärtsverketten werden alle Regeln ausgeführt, die dem Aufrufmuster entsprechen.

Data embedded Handles Dies ist die einfachste der Methoden für den indirekten Aufruf und wird deshalb oft in der Implementierung der anderen Methoden verwendet. Bei diesem Ansatz wird die Referenz zu einer Prozedur in einer Art Pointer/Handle innerhalb einer Datenstruktur (vorgehalten von einer gesonderten Komponente) gespeichert. Komponenten fordern den Inhalt des Pointers an und können damit die darin referenzierte Prozedur aufrufen. Wenn die Adresse, auf die der Pointer zeigt, geändert wird, dann resultiert dies im Aufruf einer anderen Prozedur.

5.4.3. Aufruf per Entscheidungstheorie

Diese Technik ist in gewisser Hinsicht die Verallgemeinerung des indirekten Aufrufs. Hier wird ein abstrakter, parametrisierter Dienst aufgrufen, wobei das System eine Auswahl an Methoden für den Dienst zur Verfügung hat. Es ist möglich, dass der Aufrufer nicht für alle Parameter einen Wert mit angibt; die nicht belegten Parameter werden von der Methode mit Werten belegt. Außerdem bestimmen alle in Frage kommenden Methoden die Betriebsmittel, die sie zur Ausführung benötigen, wobei diese mit einer Kostenmetrik bewertet werden. Der Aufrufer wiederum gibt seine Präferenzen an (eine Funktion, die die Methoden dahingehend auswertet, wie sie die Erfordernis des Aufrufers erfüllen). Das System ruft diejenige Methode auf, die den besten Kompromiss bei der Bewertung bietet.

5.4.4. Beobachten, Diagnose und Wiederherstellung

Ein anderes gebräuchliches Muster von selbst-adaptiven Systemen betrifft die Behandlung von Ausfällen. Typischerweise wird hier jede Verarbeitung in viele kleine, geordnete Einheiten geteilt und jede Einheit mit einer Vor- und Nachbedingung versehen (siehe Abbildung 5.14 auf der nächsten Seite). Monitore werden um die kleinen Verarbeitungseinheiten gelegt und überprüfen diese auf die Erfüllung der in einer abstrakten Beschreibung formulierten Bedingungen. Wenn bei der Abarbeitung einer Verarbeitungseinheit eine Bedingung nicht eingehalten wird, dann wird ein Diagnosedienst aufgerufen und ihm die Beschreibung der fehl geschlagenen Bedingung übergeben. Der Diagnosedienst hat die Aufgabe, die für den aufgetretenen Fehler verantwortliche Komponente ausfindig zu machen. Ein Wiederherstellungsdienst sorgt für die Wahl eines Neustarts und der Wiederherstellung der Daten.

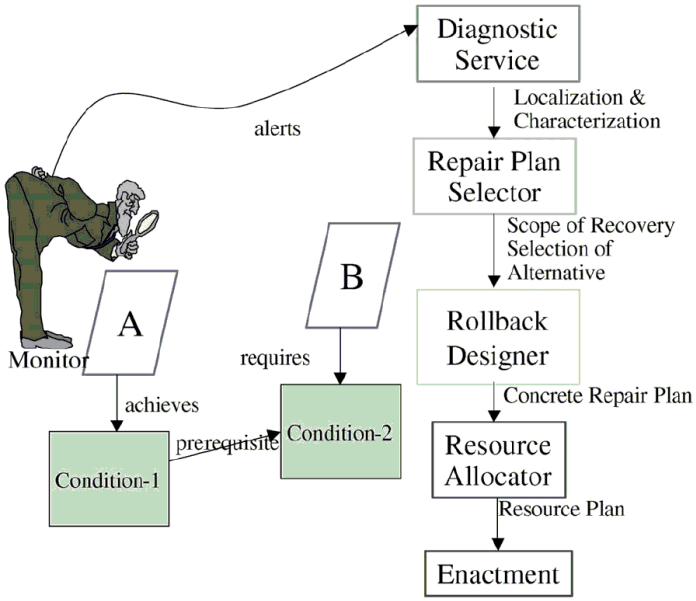

Abbildung 5.14.: Entwurfsmuster für Beobachten, Diagnose und Wiederherstellen [Laddaga *et al.* 2003]

5.5. Weitere Ansätze

In den bisherigen Ausführungen wurden — neben Beschreibungssprachen für dynamische Software-Architekturen und Mustern für die Adaptivität — mit den Regelungssystemen und dem Architekturmuster Reflection bereits konkrete Architekturansätze in der Theorie für adaptive Software-Systeme geschildert. Daneben wurden auch spezielle Arbeiten beschrieben, bei denen der jeweilige Ansatz gut zu verstehen ist. Diese sind das selbst-kontrollierende Modell von Kokar *et al.* [1999] in Abschnitt 5.2.5, das die Kontroll-Theorie besonders gut umsetzt, sowie zwei Vertreter für reflective Middleware in Abschnitt 5.3.2, bei denen der Reflection-Ansatz besonders deutlich wird. Speziell die beiden letzteren Arbeiten boten noch mehr Informationen, wie die Architektur eines adaptiven Systems (mit Fokus Adaption der Software-Architektur) aussehen könnte und wie das Management des Adaptionsprozesses erfolgt.

In diesem letzten Unterkapitel werden drei weitere Arbeiten vorgestellt, die Ansätze für die automatische und dynamische Adaption auf Ebene der Software-Architektur liefern: Cactus (5.5.1), das Rainbow-Framework (5.5.2) und den visionären Ansatz von Autonomic Computing (5.5.3).

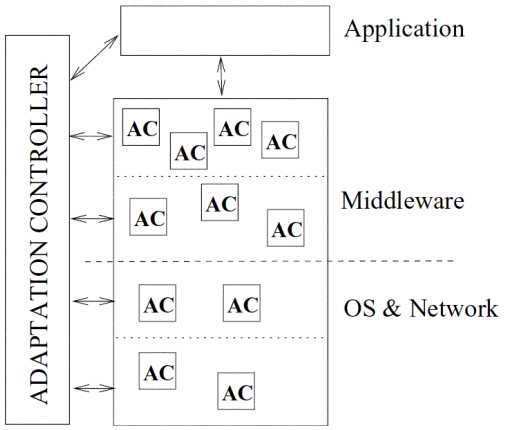

Abbildung 5.15.: Architektur des Cactus-Ansatzes für einen Host [Chen *et al.* 2001]

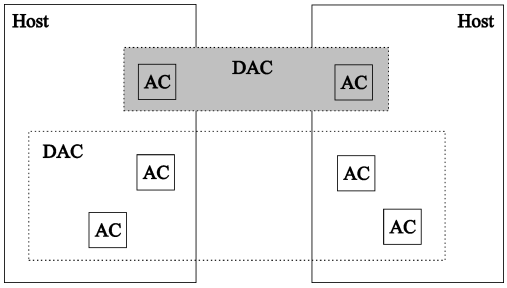

Abbildung 5.16.: Komponententyp *Distributed Adaptive Component* (DAC), der aus verteilten ACs besteht. Die grau hervorgehebene DAC verdeutlicht den in den Ausführungen berücksichtigten Ansatz.

5.5.1. Cactus

Einen etwas anderen Ansatz für adaptive Software beschreiben die Autoren in [Chen *et al.* 2001]. Bei ihrer Architektur (dargestellt in Abbildung 5.15) besteht das System aus mehreren sog. *Adaptive Components* (AC), die in verschiedenen hierarchischen Schichten des Systems angeordnet sind. Die Schichten reichen vom Betriebssystem und dem Netzwerksubsystem auf der unteren Ebene über mehrere Middlewarekomponenten bis hoch zur Anwendung.

Ein weiterer Komponententyp ist die *Distributed Adaptive Component* (DAC). Diese Komponente ist eine Ansammlung von mehreren ACs, die über mehrere Maschinen verteilt sind und durch Kooperation einen Dienst erbringen (siehe Abbildung 5.16).

Die Struktur einer *Adaptive Component*, die in Abbildung 5.17 auf der nächsten Seite zu sehen ist, besteht aus zwei verschiedenen Modulen: Es gibt ein *Component Adaptor Module* (CAM) und mehrere alternative *Adaptation-aware Algorithm Module* (AAM). Die AAMs imple-

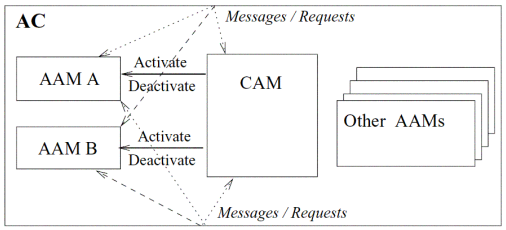

Abbildung 5.17.: Die Struktur einer *Adaptive Component* (AC) besteht aus einem *Component Adaptor Module* (CAM) für das Management und mehreren alternativen *Adaptation-aware Algorithm Modulen* (AAM) [Chen *et al.* 2001]

mentieren die Funktionalität der Komponente, wobei jedes AAM einen anderen Algorithmus realisiert. Das *Component Adaptor Module* kontrolliert das adaptive Verhalten der *Adaptive Component*. Wenn das *Component Adaptor Module* erkennt, dass ein anderes AAM besser ist als das derzeit laufende, dann tauscht es das AAM aus. Das *Component Adaptor Module* hat außerdem die Aufgabe, bei einer Adaption den Zustand zwischen den zwei Modulen zu übertragen und die Koordination zwischen verschiedenen Hosts vorzunehmen.

CAMs und AAMs sind weitestgehend unabhängig voneinander. Es sind nur einige Standardoperationen für das AAM definiert, die jedes AAM implementieren muss; beispielsweise für die Sicherung des Zustandes und für das Management der Adaption, die in mehreren Stufen abläuft. Dies erlaubt die getrennte Entwicklung von CAMs und AAMs.

Koordination. Chen *et al.* veranschlagen zwei mögliche Koordinationsarten für die Adaption:

1. *Inter-component coordination.* Es sind die Adaptionen zwischen den *Adaptive Components* (ACs) und zwischen den Schichten auf einem Host zu koordinieren.

2. *Inter-host coordination.* Die Adaptionen auf den verschiedenen Hosts in dem verteilten System sind zu koordinieren.

Die erste Form der Koordination übernimmt der *Adaptation Controller* (vgl. Abbildung 5.16 auf der vorherigen Seite).

Adaptionsprozess

Für den zuvor konzipierten Architekturansatz beschreiben die Autoren in [Chen *et al.* 2001] den Adaptionsprozess. Allerdings berücksichtigen diese Ausführungen nicht die konzipierte Architektur in ihrem vollem Umfang, sondern nur in einer eingeschränkten Version. Die Einschränkung äußert sich darin, dass das gesamte System nur aus *einer* DAC

Zustandsvektor `sysState[]`	*quantitativer Indikator, für die Eignung des* AAM*s*
Fehlerrate des Netzwerks	voraussichtliche Anzahl der notwendigen Nachrichten, die zu senden sind
Latenz des Netzwerkes	voraussichtliche Antwortszeit

Tabelle 5.3.: Fitnessfunktion eines AAMs, die für jeden Wert im Zustandsvektor einen quantitativen Indikator für die Eignung der Komponente zurückgibt

besteht, die pro Host nur *ein* AC besitzt (vgl. die graue DAC in Abbildung 5.16 auf Seite 162).[21]

Der Adaptionsprozess ist in die folgenden drei Phasen eingeteilt: Erkennung der Änderung, Übereinstimmung und Ausführen der Adaption.

Erkennung der Änderung. In dieser Phase wird eine Änderung in der Ausführungsumgebung erkannt und festgelegt, ob daraufhin eine Adaption vorgenommen werden soll. Die konzipierte Architektur schreibt nicht vor, ob das Beobachten und Erkennen der Änderung Aufgabe des AAMs oder des CAMs ist. Begründet wird diese Entscheidung damit, dass dieser Aspekt abhängig von dem Typ des Dienstes und des Implementierungsansatzes ist.[22]

Die Ermittlung des besten AAMs für die derzeitigen Anforderungen und die Ausführungsumgebung erfolgt auf Basis zweier Funktionen Fitness() und BestFit() und der Kodierung der Zustandes in einem Vektor `sysState[]`. Der Zustand wird als Vektor vorgehalten, um mehrere Kriterien der Ausführungsumgebung kodieren zu können.

Die Fitnessfunktion Fitness() wird von jedem AAM implementiert und gibt die Eignung des AAMs für einen gegebenen Zustand zurück. Ein Beispiel für einen adaptiven Kommunikationsservice ist in Tabelle 5.3 zu sehen. Die Fitnessfunktion gibt für gegebene Fehlerrate und Latenz des Netzwerkes (Zustand `sysState[]`) die geschätzte Anzahl der Nachrichten und die erwartete Antwortszeit zurück (ebenso ein Vektor).

Die zweite Funktion, BestFit(), ist im CAM implementiert. Sie hat die Aufgabe, das beste AAM für die gegebene Situation zu ermitteln. Dazu können ihr mehrere Zustandsvektoren `sysState[]` als Argumente übergeben werden.[23] Die BestFit()-Funktion ruft für jedes AAM, das in-

[21] Mittlerweile wurde dieser Ansatz weiter verfeinert. Bridges [2002] behandelt die Koordinierung mehrerer Adaptionen und ein weiterer Doktorand arbeitet derzeit an einer Familie von Adaptionskoordinierungsprotokollen, berichtet Hiltunen auf Anfrage [Hiltunen 2004].

[22] Es mag stimmen, dass der Beobachtungsgegenstand vielfältig ist, aber gegen die Realisierung der Beobachtung in dem AAM spricht Folgendes: Jedes AAM müsste die Beobachtungsfunktion *erneut* implementieren. Aus Gründen der Wiederverwendung und der Trennung der Belange [Hürsch und Lopes 1995] sollte die Beobachtungsfunktion außerhalb des AAMs erfolgen.

[23] Es ist nicht nachvollziehbar, warum *mehrere* Zustände übergeben werden. Pro Si-

nerhalb des ACs verfügbar ist, die jeweilige Fitness()-Funktion auf, um das beste AAM zu ermitteln. Falls das ermittelte AAM ein anderes ist als das derzeitig in Benutzung befindliche, dann erfolgt in der nächsten Phase eine Übereinstimmung (siehe nächster Absatz), d. h. das »beste« AAM wird noch nicht sogleich eingetauscht.

Neben der Ermittlung des besten AAMs ist die BestFit()-Funktion im AAM außerdem dafür verantwortlich, Oszillationen zwischen alternativen AAMs zu vermeiden und die Kosten für die Adaption zu optimieren.

Übereinstimmung. In der zweiten Phase findet eine Übereinstimmung statt, bei dem zwischen allen ACs ausgehandelt wird, ob eine Änderung erfolgt ist und ob eine Adaption als Antwort erfolgen soll.[24] Für die Übereinstimmung werden zwei Ansätze vorgeschlagen.

Als eine Möglichkeit wird angegeben, einen Konsens über den globalen Zustand zu erlangen. Bei diesem Ansatz wird der lokale Zustand `sysState[]` an alle ACs respektive AAMs gesandt und jedes AAM berechnet mit einer deterministischen Funktion den globalen Zustand `globalSysState[]`. Sobald der globale Zustand ermittelt worden ist, wird mithilfe der `BestFit()`-Funktion das optimale AAM berechnet. Dabei wird angenommen, dass die `Fitness()`-Funktion und die `BestFit()`-Funktion deterministisch sind, sodass jedes CAM zu derselben Entscheidung gelangt.

Bei dem zweiten Ansatz entscheidet jedes CAM/AC auf Grundlage des lokalen Zustandes `sysState[]` für sich, welches AAM das optimale ist. Dabei wird der Übereinstimmungsprozess genutzt, um zwischen den Kandidaten zu entscheiden.[25] Wenn die Auswahlfunktion deterministisch ist und jedes CAM die gleichen Eingabewerte bekommt, dann wird jedes CAM auch die gleiche Entscheidung fällen.

Ausführen der Adaption. Nach Aushandlung, ob eine Adaption vorgenommen werden soll und welches AAM_{alt} durch ein AAM_{neu} auszutauschen ist, erfolgt die Adaption. Dabei sind noch ein- und ausgehende Nachrichten durch das AAM während des Austausches zu berücksichtigen. Außerdem muss der Austausch auf den verschiedenen Hosts zu einer gleichen Zeit vorgenommen werden, damit inkompatible AAMs innerhalb des DACs nicht zur gleichen Zeit miteinander kommunizieren. Im o. g. Paper wird für diese Problematik ein Protokoll angegeben, dass die Koordinierung der ACs/CAMs innerhalb des DACs vornimmt. Details dazu sind in [Chen *et al.* 2001] zu finden.

---

tuation gibt es nach Verständnis des Autors nur einen Zustand — wobei dieser mehrere Kriterien umfassen kann, aber nicht eine Situation entspricht *einer* Tupelrepräsentation des `sysState[]`-Vektors.

[24] Zuvor hieß es, dass dies in der ersten Phase erledigt wird.

[25] Ist auch im Original unklar: »Another approach is to allow each CAM to choose the locally optimal AAM based on its local state, with the consensus process used to select from among the candidates.«

Zusammenfassung und Bewertung

Positiv. Der Ansatz entspricht einer einfachen Architektur. Als adaptives Verhalten wird sowohl die Änderung von Parametern der Module als auch der Austausch der Module — hier jedoch innerhalb einer *Adaptive Component* — unterstützt. Damit wird eine Alternative zum Reflection-Muster, bei dem es ein globales Modell des Systems gibt, aufgezeigt.

Als Adaptionsmechanismen wird die Auswahl eines besten Algorithmus' (AAM) sowie die Parametrisierung ermöglicht.

Für die Adaption kann auf Basis *mehrerer* Kriterien (Fehlerrate, Verzögerung) entschieden werden, welches AAM das beste ist. Dabei ist die jeweilige Belegung der Kriterien nicht statisch, sondern kann vom Zustand des Systems abhängig sein (z. B. für gegebene, derzeitige Fehlerrate gibt das AAM seine Performance in Form der voraussichtlich zu sendenden Nachrichten zurück). Damit wird aufgezeigt, dass die Spezifikation von Eigenschaften verschiedener zur Verfügung stehender Komponenten (hier Module genannt) nicht statisch in einer separaten Beschreibungssprache formuliert sein muss. Sie kann auch (und muss hier) durch Logik in dem Modul implementiert werden, da die Kriterien für die Auswahl einer Komponente vom Zustand des Systems (hier

Es wird außerdem der sichere Austausch von Algorithmen während einer bestehenden Kommunikation behandelt, ohne dass das System gestoppt oder Teile des Systems kurzzeitig »eingefroren« werden müssen (wie sonst im transaktionalen Ansatz von Wermelinger [1999]; vgl. Seiten 129 und 151).

Während der Laufzeit können neue AAMs hinzugefügt werden. Da die Adaptionslogik sowohl von den AAMs als auch von den CAMs erbracht werden kann (der Ansatz diktiert keine feste Verfahrensweise), kann neben zusätzlichen redundanten Algorithmen auch das adaptive Verhalten erweitert werden.

Einschränkend. Für die konzipierte Architektur wird nur ein eingeschränkter Ansatz angegeben (nur ein AC pro Host, das mit ACs auf anderen Hosts kommuniziert).

Ermöglicht nur die Algorithmenauswahl innerhalb einer Komponente (Adaptive Component). Die Komponente selbst bleibt statisch und nur die in ihr enthaltenen Algorithmen werden je nach Anforderung ausgewählt. Die Struktur der Architektur bleibt dabei gleich, d. h., es ist kein Hinzufügen oder Entfernen von Komponenten möglich. Diese Einschränkung resultiert allerdings in einem sehr einfachen Architekturansatz.

Die Ausführungen von Chen *et al.* rufen an einigen Stellen, an denen das Verstehen nicht ganz klar ist, Fragen hervor.

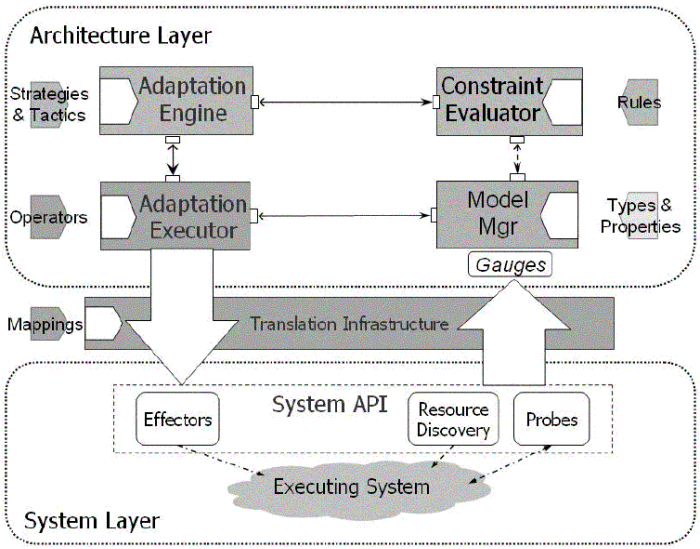

Abbildung 5.18.: Rainbow-Framework für die externe Adaption eines Systems [Cheng *et al.* 2004]

5.5.2. Adaptionsframework Rainbow

Forscher an der bekannten Carnegie Mellon University, USA, haben den in [Oreizy *et al.* 1999] formulierten Ansatz umgesetzt und im Rahmen des ABLE-Projektes das Rainbow-Framework [Cheng *et al.* 2004] für selbst-adaptive Software entwickelt. Diese Architektur ähnelt weitestgehend der im DARPA DASADA[26]-Projekt [Salisan 2004] geschaffenen (vgl. beispielsweise [Wile 2002], [Valetto und Kaiser 2002a]). Im Folgenden wird die Architektur von Rainbow geschildert, wie sie in [Garlan und Schmerl 2002] und [Cheng *et al.* 2004] beschrieben ist.

Kernaspekt des Architekturansatzes ist die externe Adaption, bei der die Adaptionsinfrastruktur außerhalb des laufenden Systems angeordnet ist (siehe Abbildung 5.18). Dementsprechend ist das Framework in zwei Schichten aufgeteilt: Die *Systemschicht* (System layer) enthält das laufende System, in der *Architekturschicht* (Architecture layer) befindet sich die Adaptionslogik, die auf Basis des Architekturmodells Adaptionsentscheidungen für das System trifft. Über eine *Übersetzungsschicht* sind die Architektur- und die Systemschicht miteinander verbunden.

Infrastruktur und Adaptionsprozess des Frameworks

Architekturmodell. Mit dem externen Ansatz einhergehend ist die Verwendung eines Architekturmodells für die Adaption des Systems [Garlan und Schmerl 2002]. Für ein Modell des Systems wird die ADL ACME verwendet (vgl. Abschnitt 2.1.3 auf Seite 9). Der Vorteil von

[26] Dynamic Assembly for Systems Adaptability, Dependability, and Assurance

ACME ist, dass diese Sprache keinen bestimmten Architekturstil fordert, sondern durch die Spezifikation einer bestimmten »Familie« verschiedene Architekturstile ermöglicht [Cheng *et al.* 2002a], [Schmerl und Garlan 2002], — im Gegensatz zu z. B. C2, das hierarchisches Publish-Subscribe vorschreibt [Taylor *et al.* 1996], und Weaves, dem ein Datenfluss-Modell inhärent ist [Gorlick und Razouk 1991].

Der Property-Mechanismus von ACME erlaubt es dabei, den Elementen des Modells verschiedene Attribute zuzuteilen. Properties eines Konnektors können beispielsweise das Interaktionsmodell oder Performanceattribute (z. B. Verzögerung und Bandbreite) beschreiben. Für eine Komponente können die Properties z. B. seine Kernfunktionalität, Performanceattribute u.˙ definieren.

Adaptionskreislauf. Entsprechend des Ablaufes für eine Adaption wird der Architekturansatzes nachfolgend beschrieben (vgl. Abbildung 5.18 auf der vorherigen Seite).

Ein *laufendes System* wird beobachtet, um Aufschluss über sein Verhalten zu bekommen. Das System kann aus einem Komponentenframework wie beispielsweise CORBA oder EJB bestehen, es kann auch ein beliebiges IT-System sein, z. B. eine Webserver-Farm oder jede andere verteilte Anwendung.

Das Beobachten übernehmen sog. *Probes*. Diese sind an verschiedenen Stellen des Systems positioniert und messen Werte oder erkennen Ereignisse (beispielsweise Prozess X öffnet Datei Y). Für das Entdecken von neuen Ressourcen kann ein *Resource Discovery*-Dienst abgefragt werden. Die von den Probes gemessenen Werte können veröffentlicht oder von anderen Einheiten abgefragt werden. *Gauges* übernehmen die Aggregation und Abstraktion der gemessenen Werte und bringen die Werte in Verbindung mit dem Architekturmodell, d. h. der Zustand des Systems wird in dem Architekturmodell reflektiert. Dazu kontaktieren sie den *Model Manager*, der das Architekturmodell verwaltet und Zugriff auf dieses bietet.

Die Adaption findet statt, indem der *Constraint Evaluator* das Architekturmodell in periodischen Abständen überprüft und die Adaption anstößt, wenn eine Randbedingung im Modell verletzt wird. Mit der Überprüfung soll festgestellt werden, ob sich das System entsprechend seiner Spezifikation korrekt verhält. Wird eine Verletzung festgestellt, dann ist es Aufgabe der *Adaptation Engine*, eine Reparatur basierend auf den *Adaptionsstrategien* und *Taktiken* festzulegen.

Die Reparatur wird von der *Adaptation Engine* und dem *Adaptation Executor* ausgeführt. Dafür existieren systemspezifische *Operatoren*, die die Elemente des Systems rekonfigurieren können. Das können die in Abschnitt 4.1 aufgeführten Operationen sein, im Falle einer Web-Anwendung beispielsweise die Hinzunahme einer zusätzlichen Webserver-Instanz (hier Host, nicht Thread).

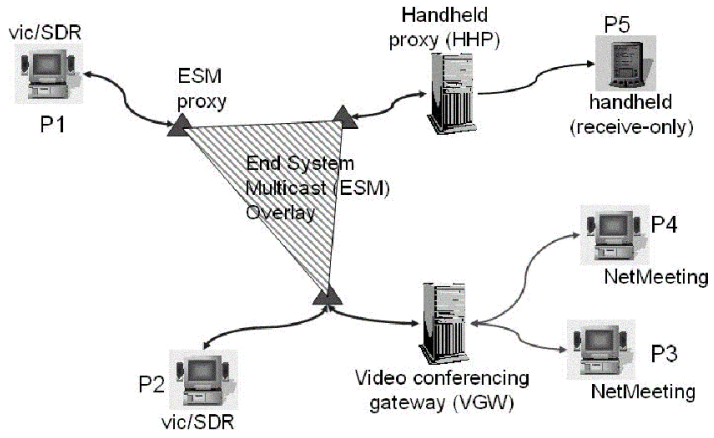

Abbildung 5.19.: Beispiel für die Adaption in einem Videokonferenzsystem [Cheng *et al.* 2004]

Show-Case

Für ein besseres Verständnis zur Wirkungsweise des Rainbow-Frameworks wird als Abschluss noch ein Show-Case skizziert, der die Adaption auf Ebene der Software-Architektur besser verdeutlichen soll (im Vergleich zu dem bis zu diesem Abschnitt nur zu sehenden Austauschen einer »kleinen« Komponente in einem Komponentenframework).

Videokonferenzsystem. Das folgende Szenario hat ein Videokonferenzsystem als Beispiel; verdeutlicht in Abildung 5.19 und auch beschrieben in [Cheng *et al.* 2004]. Einige der Teilnehmer benutzen Vic/SDR-Tools, die IP-Multicast verwenden. Andere Teilnehmer sind über NetMeeting angeschlossen, das nur mit Unicast funktioniert. Außerdem gibt es noch Nutzer mit einem Handheld-Gerät, das eine leicht modifizierte Version von Vic verwendet. Für diese Nutzer wird ein spezieller Handheld-Proxy benötigt. Das System besteht zudem aus einem Videokonferenzgateway, das die Umwandlung zwischen H.323 (Netmeeting) dem SIP[27] (Vic) vornimmt, und diversen Multicast-Proxies, die die Multicast-Funktionalität für die effiziente Kommunikation im Weitverkehrssystem realisieren.

Ausschlaggebend für die benötigte *Adaptivität* in dem System sind die Minimierung der Kosten und die Sicherstellung von genügend Bandbreite zwischen Handheld und dem Handheld-Proxy. Deshalb wird das Handheld-Gerät einem *anderen Proxy* zugeteilt, wenn die Bandbreite zwischen Proxy und Endgerät eine Schwelle unterschreitet. Eine zweite Adaptivität wird bezüglich der Kostenvariablität erreicht, für die das Gateway verantwortlich ist. Hier können die Kosten minimiert werden, indem auf ein *preiswerteres Gateway gewechselt* wird, sobald weniger Netmeeting-Teilnehmer das System benutzen.

[27] Session Initiation Protocol

Verwendete Adaptionsmechanismen. Bezug nehmend auf die in Abschnitt 4.1 beschriebenen Adaptionsoperationen für eine Software-Architektur entspricht der Wechsel zu einem preiswerteren Gateway dem *Austausch einer Komponente* (Abschnitt 4.1.2 auf Seite 98). Die Zuteilung eines Clients zu einem anderen Proxy ist dem Mechanismus *Neuzuweisung/Binden* (Abschnitt 4.1.3 auf Seite 107) zuzuordnen.

Zusammenfassung und Bewertung

Der externe Ansatz von Rainbow eignet sich im Besonderen für existierende Systeme, die um adaptive Eigenschaften erweitert werden sollen, aber auch für neue Systeme. Ein weiterer Vorteil ist die Möglichkeit zur Wiederverwendung von Teilen der Infrastruktur.

Durch die Verwendung von ACME als ADL besteht keine Einschränkung für einen bestimmten Architekturstil. Ein Nachteil hierbei ist allerdings, dass das Framework damit geschlossen-adaptiv ist, statuieren Dashofy *et al.* [2002].

Um einem Framework zu entsprechen, teilt Rainbow die Architektur in zwei Schichten auf: die Systemschicht und die Architekturschicht. Die Systemschicht beinhaltet das eigentliche System, in der Architekturschicht befindet sich die Adaptionslogik, die auf Basis eines Modells des unterliegenden Systems die Adaption vornimmt. System- und Architekturschicht sind über systemspezifische Effektoren und Probes/Ressource-Discovery miteinander verbunden, um Änderungen in der einen Schicht in der anderen Schicht wiederspiegeln zu lassen. Damit ergibt sich ein großer Zusammenhang zum Meta-Level-Architekturen bzw. dem Reflection-Ansatz, bei dem es auch zwei »causally connected« Schichten gibt.

Wie bereits auf Seite 133 kurz erwähnt, verwendet das Rainbow-Framework (und ebenso der Architekturansatz des DASADA-Projektes) einen geschlossenen Regelkreis für die Adaption des Systems.

Der Showcase hat gezeigt, dass die Adaption in einer Software-Architektur nicht nur auf kleinem Niveau im Rahmen des Austausches einer Komponente auf einem Host passieren muss, sondern auch auf großer Ebene vollzogen werden kann — je nach Abstraktionsstufe der betrachteten Komponenten.

5.5.3. Autonomic Computing

Als letztes wird der Ansatz von IBM's Autonomic Computing, wie er in [Keffart und Chess 2003] und [Steinberg 2003a, b] beschrieben ist, geschildert.

Architekturansatz

IBM definiert ein autonomes System als eine Ansammlung von *autonomen Elementen* (autonomic elements, nachfolgend auch nur kurz

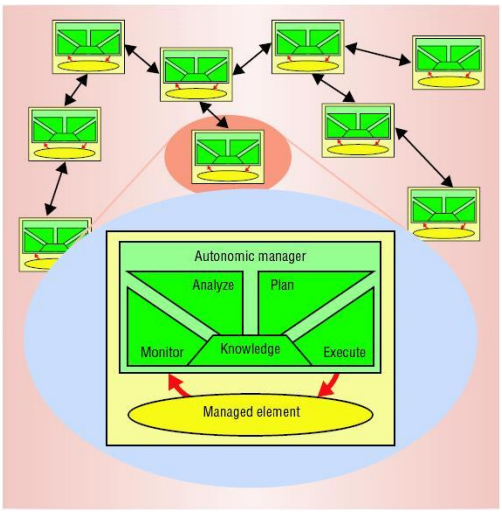

Abbildung 5.20.: Struktur eines autonomen Elements (Autonomic Element = Autonomic Manager + Managed Element) [Keffart und Chess 2003]. Die Elemente interagieren mit anderen Elementen und Menschen über ihren Autonomic Manager.

Element genannt), die miteinander interagieren (siehe Abbildung 5.20). Jedes autonome Element enthält Betriebsmittel und liefert seine Dienstleistungen an Menschen und andere autonome Elemente. Das interne Verhalten sowie die Beziehungen eines autonomen Elements zu anderen Elementen werden durch Regeln und Ziele bestimmt, die Menschen oder andere Elemente aufgestellt haben.

Ein autonomes Element besteht typischerweise aus einem oder mehreren *zu verwaltenden Elementen* sowie dem sog. *Autonomic Manager*, der das autonome Element kontrolliert und nach außen präsentiert. Das zu verwaltende Element kann im Sinne von IBM's Autonomic Computing sowohl eine Hardwareressource sein, wie beispielsweise Speicher, CPU[28] oder ein Drucker, als auch Softwareelemente (Datenbank, Verzeichnisdienst oder ein großes Legacy-System) umfassen.

MAPE-Kontrollschleife. Über *Sensoren* und *Effektoren* ist das zu verwaltende Element mit dem *Autonomic Manager* verbunden (vgl. [Steinberg 2003b]. Die *Sensoren* haben die Aufgabe, Informationen über den Zustand des Elements und der Umgebung zu sammeln, *Effektoren* werden dazu benutzt, den Zustand des Elements zu verändern. Der Autonomic Manager beherbergt eine Kontrollschleife, bestehend aus den Modulen zum *Beobachten (**m**onitor)* des Elements, zum *Analysieren (**a**nalyze)* der gemachten Beobachtungen, für die *Planung (**p**lan)* von Änderungsaktionen und dem *Ausführen (**e**xecute)* dieser. Diese Abfolge hat der Schleife auch ihren kurzen Namen gegeben: MAPE.

[28] Central Processing Unit

Neben dieser Kontrollschleife besitzt jedes autonome Element *Wissen (knowledge)*. Steinberg [2003b] charakterisiert die Bedeutung des Wissens lediglich hinsichtlich eines Datenspeichers für beobachtete Daten. Regeln oder erlerntes Verhalten wird offensichtlich nicht in dieser Komponente gespeichert.

SOA[29]. Das Selbst-Management des Systems wird durch das Zusammenspiel (Myriade) der autonomen Elemente sowie durch das Management des individuellen autonomen Elementes erreicht. Keffart und Chess vergleichen dies mit dem Agieren der individuellen Ameisen in einer Ameisenkolonie. Deshalb werden für die Interaktion mit anderen Elementen Service-orientierte Architekturen (Abk. SOA, [Ullmann 2004]) propagiert, z. B. in Form von Web Services [Kreger 2001] und Grid Services (OGSA[30]) [Foster *et al.* 2002]. In dieser Form können von autonomen Elementen angebotene Dienste von anderen Elementen frei genutzt werden, wobei sich das »frei« mehr auf eine lose Koppelung der Dienste bezieht.

Jedes autonome Element ist verantwortlich für das Management seines eigenen internen Zustandes und Verhaltens sowie für die Verwaltung der Interaktionen mit der Umgebung. Bestimmend für das Verhalten und die Interaktionen sind die *Ziele*, die ein autonomes Element besitzt. Diese wurden von einem Entwickler oder von anderen Elementen mit entsprechender Autorität vorgegeben, oder sie gehen aus Unterverträgen mit anderen Peer-Elementen hervor.

Das Anbieten und Nutzen von Diensten wird demnach durch die Ziele, die ein autonomes Element hat, bestimmt. Deshalb ist die Koppelung nicht wirklich frei, sondern bei einem Dienstaufruf an ein autonomes Element überprüft dieses, ob die Erbringung des Dienstes im Sinne *seines* Zieles ist. Für die Erreichung seiner Ziele nutzt ein autonomes Element die notwendigen Ressourcen anderer Elemente. Damit einhergehend ist es jedoch auch verantwortlich für Ausnahmefälle, wenn die benötigte Ressource einen Fehler aufweist.

Wie bei der komponentenbasierten Programmierung üblich [Szyperski 2002] müssen alle Elemente ihre angebotenen und benötigten Dienste spezifizieren. Für die Registrierung der Fähigkeiten, der genauen Adresse und der unterstützten Protokolle registriert sich ein autonomes Element in einem Directory Service. Je nach gewählter Architektur kann dies der Directory Service UDDI[31] [UDDI] oder der von OGSA [Foster *et al.* 2002] sein.

Ein wichtiger und komplexer Aspekt ist die Aushandlung des zu erbringenden Dienstes, den ein autonomes Element anfordert, nachdem es diesen gefunden hat. Die Aushandlung, ob der angeforderte Dienst erbracht wird, kann je nach zu berücksichtigender Faktoren ganz ein-

[29] Service-Oriented Architecture
[30] Open Grid Services Architecture
[31] Universal Description, Discovery and Integration

fach bis hochkomplex sein. Einfach bzw. nicht-existent ist die Aushandlung, wenn der Dienst nach dem Prinzip »first–come–first–served« oder unterwürfig (der Dienst wird erbracht, solange Ressourcen verfügbar sind) agiert. Komplexer kann sich der Prozess gestalten, wenn bilaterale oder multilaterale Aushandlungen über mehrere Attribute (Preis, Servicegrad, Priorität) und ggf. über eine Auktion stattfinden müssen.

Neben dem Service-Gedanken trägt der Ansatz des Autonomic Computing die Charakteristik des Agenten-Paradigma [Jennings 2000]. Dabei können einzelne autonome Elemente als Agenten und ein gesamtes autonomes System als Multi-Agenten-System angesehen werden.

Zusammenfassung und Bewertung

Die von IBM skizzierte Architektur zeigt einen interessanten Ansatz, bei dem die Elemente eines Systems nicht durch explizites, externes Management verwaltet werden, sondern selbst als autonome Agenten ihre Interaktionen steuern. Die Elemente werden dabei durch Regeln und Ziele gesteuert, die durch Menschen oder andere Elemente vorgegeben sind. Durch das kooperative Zusammenwirken der Elemente wird das gewünschte Systemverhalten mit den in Abschnitt 2.4.1 formulierten CHOP-Funktionen erreicht. Für das Selbstmanagement werden Service-orientierte Architekturen propagiert, da derartige Architekturen eine lose Koppelung erlauben und das Anbieten und Verwenden von Diensten leicht gestalten.

Dem Charakter einer Vision entsprechend ist der Ansatz aktuelles Forschungsthema, wobei einige Produkte von IBM bereits mit autonomen Fähigkeiten ausgestattet wurden [IBM 2004]. Es gibt noch viele Probleme zu lösen, bis die von IBM skizzierte Vision vollständig umgesetzt ist.

Außer einigen Ideen, wie Autonomic Computing realisiert werden könnte, wird nur der Adaptionskreislauf MAPE beschrieben. Mehr lässt sich aus den Ausführungen nicht ableiten, es sei denn es sollen keine die Architektur betreffende Aspekte betrachtet werden, sondern konkrete, spezieller Aspekte bei der Umsetzung des Ansatzes [IBM 2003b], [SAACS 2003, 2004], [Li *et al.* 2004], [Self-∗ 2004]. Wie auch bei den Ansätzen zuvor bleiben bei IBMs Ausführungen Fragen offen, z. B. die in Fußnote 14 auf Seite 204 formulierte.

5.5.4. Fazit

Der erste Architekturansatz, Cactus (5.5.1), liefert eine interessante Verfahrensweise für den Austausch von Komponenten zur Laufzeit sowie zur Berücksichtigung einer sicheren Adaption. Cactus ist prädestiniert für Kommunikationsdienste in verteilten Systemen. Der Ansatz verzichtet auf ein explizites Modell der Software. Stattdessen wird

der Adaptionsvorgang von jeder einzelnen Komponente (AC) im System selbst und in Kooperation mit anderen gesteuert.

Waren die bisherigen Ansätze eher auf dem Niveu von Komponentenframeworks/Middleware, liefert nun das Adaptionsframework Rainbow (5.5.2) Ideen, die auf höherer Software-Architektur-Ebene agieren. Allen zuvor beschriebenen Ansätzen ist gemein, dass sie als Adaptionsmechanismus den Austausch von Komponenten in der Software-Architektur ermöglichen. Bei Rainbow findet der Austausch im Unterschied zu den zuvor geschilderten Ansätzen jedoch nicht auf kleingranularer Ebene eines Komponentenframeworks oder der Middleware statt, sondern die Komponenten können größere Architekturkomponenten sein. Das Abstraktionsniveau für eine Komponente in der Architektur ist auf einer höheren Stufe, in dem Beispiel auf Ebene von Hosts (Austausch des Gateways in der Architektur). Dabei unterstützt Rainbow aufgrund seines Framework-Charakters auch die Adaption auf Basis von Komponenten, die auf einem Host deployt sind (EJB, CORBA). Ein weiterer Vorteil des Framework-Charakters ist die Fokussierung auf Wiederverwendbarkeit. Getrieben durch diese Motivation haben die Verfasser die Software-Architektur des Framework in viele wiederverwendbare Komponenten aufgeteilt. Aus diesem Ergebnis kann für die Entwicklung von eigenen adaptiven Systemen konkrete Ansätze für die Software-Architektur abgeleitet werden.

Der letzte vorgestellte Architekuransatz (5.5.3) behandelt die Vision von IBMs Autonomic Computing. IBM stellt sich ein System als eine Ansammlung von autonomen Elementen vor, die in einer Myriade ein global vorgegebenes Ziel für das Systemverhalten erreichen. Dabei verwenden sie Aushandelungsprozesse für das Benutzen der Ressourcen von anderen Elementen. Der formulierte Ansatz ist noch eine Vision und findet nur teilweise Umsetzung in einigen Produkten von IBM oder anderen Forschungsaktivitäten.

Der Architekturansatz von IBM ist allerdings nicht zu vergleichen mit dem in Abschnitt 5.5.1 beschriebenen Ansatz auf Basis von Cactus. Beide Ansätze haben lediglich als Gemeinsamkeit, dass ein lokaler Manager für die Verwaltung einer Komponente verantwortlich ist.[32] Der große Unterschied zwischen den zwei Systemen ist, dass bei Cactus die Struktur des Komponentennetzes statisch ist. Soll eine Komponente (Modul) ausgetauscht werden, dann geschieht dies lokal für den Knoten im Netz. Bei IBM dahingegen gehen die Elemente (hier Autonomic Manager + zu verwaltendes Element) selbstständig Beziehungen zu anderen Elementen ein, wenn das bis dahin genutzte Element nicht mehr das optimale ist.

Wie auch im Rainbow-Ansatz wird für Autonomic Computing eine geschlossene Kontrollschleife (MAPE), für das Beobachten des zu ver-

[32] Bei Cactus verwaltet ein *Component Adaptor Module* (CAM) die Algorithmen-Module (AAMs), bei IBM überwacht der sog. *Autonomic Manager* das zu verwaltende Element. Dieses Element muss nicht aus einem kleinen Algorithmen-Modul bestehen, sondern kann bis zu einer Größe von Legacy-Systemen bestehen.

waltenden Elements (und damit auch der Auswirkungen von erfolgter Anpassung) benutzt. Die Ausführungen zu Cactus sind hinsichtlich des Controllers nicht

AOP

Dieses letzte Unterkapitel hat drei weitere Ansätze gezeigt. Die Literatur hält noch weitere interessante Arbeiten für die Adaption zur Laufzeit bereit. Einer dieser ist das Paradigma der Aspekt-orientierten Software-Entwicklung (AOSD/AOP) [Kiczales *et al.* 1997]. AOSD wird in vielen Arbeiten dafür genutzt, eine klare Trennung der Adaptions- von der Anwendungslogik zu erreichen [Yang *et al.* 2002], [Zambrano *et al.* 2004], [David und Ledoux 2003]. Die Entwicklung läuft in einem zweistufigen Prozess ab:

1. Zuerst erzeugt der Anwendungsentwickler die eigentliche Kernanwendung (nur mit Business-Logik ausgestatte), ohne sich um die Charakteristik der späteren Umgebung zu sorgen, in der die Anwendung deployt wird. Der Output der ersten Phase ist eine adaptierbare Anwendung ohne adaptive Fähigkeiten

2. Mit dem Output der ersten Phase und dem präziseren Wissen über die Einsatzumgebung erweitert der Deployer oder Administrator die Anwendung um Adaptionslogik. Dazu wird das Weaving von Aspekten von AOP genutzt.

AOP kann als Erweiterung des in Unterkapitel 5.3 beschriebenen Reflection-Musters angesehen werden [Kon *et al.* 2002]. AOP erweitert die einfache Trennung in Basis- und Metaebene zu einer feinstufigeren Granularität: Mehrere überschneidende Problembereiche (»croscutting concerns«, genannt Aspekte) werden separat entwickelt und am Ende zu einem kohäsiven System zusammengewoben (genannt Weaving).

Bisher konnten diese Aspekte zur Laufzeit nicht mehr identifiziert werden, sodass deren Nutzung zur dynamischen Adaption nicht möglich war. Mit beispielsweise [Popovici *et al.* 2003] wird diese Beschränkung aufgehoben.

Einen Ansatz, der auf dem dynamischen Weaving von Aspekten basiert, wird in [Falcarin und Alonso 2004] beschrieben. Die Autoren nutzen dynamisches AOP, um verschiedenen Varianten von Code-Fragmenten *zur Laufzeit* auszutauschen.

5.6. Zusammenfassung

Das vorliegende Kapitel hat einen Überblick über Ansätze für dynamische Software-Architekturen gegeben.

Als erstes (5.1) wurden die Unterstützung von Architekturbeschreibungssprachen (ADLs) beschrieben. Diese sind wichtig, um die Integrität der dynamisch veränderten Architektur sicherzustellen, indem

in sog. ACLs Randbedingungen formuliert werden können, die einzuhalten sind. Außerdem können ADLs dazu genutzt werden, die Dynamik zu spezifizieren. Bei einigen Sprachen erfolgt die Spezifizierung in der gleichen ADL, in der die statischen Aspekte beschrieben werden, bei anderen in einer separaten sog. AML. Neben diesen Ansätzen gibt es einige formalere Spezifikationssprachen. Vorgestellt wurden drei Möglichkeiten zur formaleren Spezifikation. Am interessantesten unter diesen drei Ansätzen ist der transaktionale Ansatz, der es ermöglicht, in einem verteilten System parallel und auf Transaktionsbasis die Änderungen an der Architektur auszuführen. Dabei werden an zu ändernde Bereiche angrenzende Konnektoren »eingefroren«, damit für die Zeitspanne der Rekonfiguration der restliche Teil des Systems weiter operieren kann. Des Weiteren wurde mit CHAM und Alloy Unterstützung für selbst-organisierende Software-Architekturen aufgezeigt.

Der darauf folgende Abschnitt 5.2 hat die Modelle der Regelungssysteme behandelt. Das einfachste Modell ist der offene Regelkreis (auch Optimalwertsteuerung oder »deliberative« genannt). Dieses Modell findet sich in allen einfachen Systemen mit einer Ereignisbehandlungsschleife wieder. Eine Erweiterung des offenen Regelkreises ist es, diesen durch Hinzunahme von Rückkoppelung zu schließen (Rückkoppelungssteuerung, »reactive«). Ein weiterer Unterschied zum offenen Regelkreis ist, das nicht mehr die Inputvariablen (die Umgebung) für den Controller bekannt sind, sondern nur die Werte der kontrollierten Variablen. Neben den zwei bekannten Modellen für offene und geschlossene Regelkreise wurden zwei weitere Ansätze vorgestellt: adaptive Regelung und rekonfigurierbare Regelung. Eine Adaptivregelung eignet sich für Systeme mit breiten, dynamischen Störungen jeder Art und strengen Zeiterfordernissen. Auch werden Probleme von Systemen gelöst, deren Störungen unvorhersagbar auftreten sowie sich ändern, sodass wegen der nicht-linearen Merkmale der Anlage eine unvorhersagbare Anwort hervorgerufen wird. Bei der Verwendung einer Adaptivregelung brauchen die exakten Werte der Modellparameter und die des Controllers nicht genau bekannt sein. Eine Erweiterung der Adaptivregelung ist die Möglichkeit zur Rekonfiguration. Bei einem rekonfigurierbaren Regelungssystem werden verschiedene Modelle des Systems und verschiedene Kontrollgesetze in Datenbanken vorgehalten. Diese können gegen die aktuell genutzten Modelle/Regeln ausgetauscht werden, wenn eine signifikante Änderung im Systemmodell erkannt wird. Als Abschluss des Abschnittes wurden die vorgenannten Regelungsmodelle in einem integrierten Architekturansatz geschildert.

Insbesondere im Fazit zu den Regelungssystemen (5.2.6) wurde der Zusammenhang zwischen Adaptivität und Selbst-Management deutlich. Herkömmliche adaptive Systeme benutzen offene Regelkreise, d. h. eine Optimalwertsteuerung, für die Umsetzung der Adaptionslogik. Ein derartiges Verfahren ist hinreichend für Systeme, bei denen gut vorhergesagt werden kann, welche Eingaben sie benötigen, um einen bestimmten Zustand oder eine bestimmte Ausgabe zu erreichen.

Dieses Verfahren ist so lange anwendbar, wie die Anzahl der Regeln im Rahmen bleibt oder nur einige sich widersprechende Regeln zu behandeln sind (dies ist bei einfachen Umgebungen, die bei der Spezifikation des Systems gut vorherbestimmt werden können, der Fall). Wird eine dieser Bedingungen verletzt, d. h., wird die Anwendung zu komplex, dann wird eine Rückkoppelungssteuerung empfohlen. Diese gibt dem Controller Feedback über die erbrachte Ausgabe des Systems. Damit kann das System, ohne genaue Regeln für jede Konstellation in der Umgebung besitzen zu müssen, Adaptionsentscheidungen treffen. Damit wird der Anwendung die Aufgabe auferlegt, selber für Robustheit (gegenüber einer nicht ganz vorherstimmbaren, teils dynamischen, Umgebung) Sorge zu tragen. Die Software beobachtet sich und die Umbebung und vergleicht das Ergebnis der dabei erfolgten Evaluation mit dem vorgegebenen Ziel. Eine Notwendigkeit, das Ziel durch die Verwendung möglicher alternativer Wege zu erreichen, ist die Self-Awareness des Systems.

Als ein fundamentales Konzept für Self-Awareness wurde im Anschluss an die Kontroll-Theorie Reflection (5.3) vorgestellt. Der Reflektion-Ansatz ist ein universales Konzept, sowohl für adaptive als auch adaptierbare Systeme. Das Prinzip von Reflection beruht darauf, ein System sich selbst bewusst zu machen und ausgewählte Aspekte der Struktur und des Verhaltens für die Adaption und Veränderung zu erschließen. Mit einem Modell über sich selbst ausgestattet kann sich die Software an die dynamische Umgebung anpassen, indem es seine Operationen auf Basis des Zustandes der Umgebung und seiner eigenen Repräsentation »überdenkt«. Die Beschreibung für reflective Systeme erfolgte auf Basis des Architekturmusters Reflection, auch Open Implementation oder Meta-Level Architecture genannt. Dieses teilt ein System in zwei Ebenen: die Basisebene und die Metaebene. In der Basisebene wird die Anwendungslogik vorgehalten. Die Metaebene bietet über mehrere Metaobjekte und dem Metaobjektprotokoll (MOP) Zugriff auf die Repräsentation des Systems sowohl lesend (Introspection) als auch schreibend (Intercession). Die Änderungen sowohl in der Metabene als auch in der Basisbene wirken sich dabei umgehend auf die andere Ebene aus. Diese Verbundenheit von Basis- und Metaeben wird als Reflection oder »causally connected« bezeichnet. Mit dem Vorstellen von zwei Beispielen, die das Reflection-Muster für Adaptivität umsetzten, wurden zwei Vertreter von sog. reflective Middleware beschrieben und einander gegenübergestellt. Dabei wurde der Zusammenhang zum Konfigurationsmanager aus der Konfigurationsprogrammierung um die ADLs deutlich. Die beiden vorgestellten Middleware-Systeme bieten konkrete Ansätze für die Entwicklung von adaptiven Systemen, sowohl in Form der Nutzung als auch als Veranschaulichung für eine eigene Umsetzung des Reflection-Ansatzes.

Konkret für eine Umsetzung wurden im Anschluss daran einige weitere Muster für die Entwicklung von adaptiven Systemen beschrieben

(Observer, Mediator, Blackboard, Memento, Proxy, indirekter Aufruf, Aufruf per Entscheidungstheorie etc.).

Den Abschluss des vorliegenden Kapitels haben drei weitere Ansätze gebildet (5.5): Cactus, Rainbow-Framework sowie IBMs Autonomic Computing. Diese haben einige weitere Ideen für die Entwicklung von adaptiven Systemen geliefert. Darunter der Service-Gedanke und das Agenten-Paradigma eines autonomen Systems bei IBM, die einfache und außerdem sichere Lösung für einen Austausch einer Komponente/Modul in Cactus sowie die Spezifizierung von Performanceattributen im Modul selbst. Der Adaptions-Kreislauf bei Rainbow im Besonderen (bei IBMs MAPE-Schleife eher weniger) hat aufgezeigt, wie die Adaption in einem System ablaufen kann.

Der weitere Verlauf

Nachdem das vorliegende Kapitel einen Überblick über bestehende Ansätze in der Literatur gegeben hat, erfolgt im nächsten Kapitel ein Fazit der Recherche. Dieses bietet auf hoher Ebene einen generellen Ansatz für die Entwicklung von adaptiven Software-Architekturen.

You know you have achieved perfection in design,
not when you have nothing more to add,
but when you have nothing more to take away

Antoine de Saint Exupéry

6 Empfehlungen für die Software-Architektur

In diesem Kapitel sollen Ausführungen dahingehend erfolgen, wie die Software-Architektur eines adaptiven Systems aussehen könnte. Dazu werden die im bisherigen Verlauf der Arbeit gewonnen Erkenntnisse in einer großen Übersicht zusammengefasst. Dieses Kapitel kann als Fazit der vorhergehenden Kapitel angesehen werden, wobei an einigen Stellen Ansätze aus nicht gesondert behandelten Papern [Cobleigh *et al.* 2002], [Oreizy *et al.* 1999] und einigen mehr aus dem DASADA-Projekt [Salisan 2004] referenziert werden.

Das Kapitel gibt zuerst Empfehlungen hinsichtlich der dynamischen Aspekte (6.1), danach werden statische Aspekte beschrieben (6.2). Die Empfehlungen für eine adaptive Software-Architektur werden im Anschluss daran mithile einiger konstruierter Anwendungsfälle evaluiert.

Architektur-Begriffe

In den folgenden Ausführungen wird der Begriff Architektur in verschiedenen Konstellationen verwendet: Architektur, Software-Architektur, Adaptionsarchitektur sowie die ähnlichen Begriffe Adaptionsinfrastruktur, Infrastruktur und auch Anwendung. Hier ist es wichtig, eine klare Unterscheidung vorzunehmen, da die Begriffe für verschiedene Aspekten stehen. Die vorgenannten Begriffe bilden in diesem Kapitel drei Klassen, welche in Tabelle 6.1 auf der nächsten Seite zusammgefasst sind.

Wie eingangs bereits formuliert wurde, gibt das vorliegende Kapitel Empfehlungen hinsichtlich der *Software-Architektur* (Ablauf- und Bauplan) eines adaptiven Systems. Als Adaptionsgegenstand wird in der Arbeit die *Architektur* eines Systemes betrachtet. Gleichwohl

vorrangig genutzter Begriff	*Synonyme*	*Bedeutung*
Software-Architektur		Ablauf- und Bauplan für ein System, hier für ein adaptives; jeweils beschrieben in Abschnitt 6.1 und 6.2
Architektur	Anwendung, System	Adaptionsobjekt/-gegenstand, d. h. die Anwendung, die adaptiert wird (Anwendungsarchitektur und Systemarchitektur, ggf. auch Infrastruktur selbst); die Infrastruktur ist auch für andere Adaptionsobjekte notwendig, deshalb auch Verwendung von Anwendung
(Adaptions-) Infrastruktur	Adaptions-architektur	Komponenten, die die Infrastruktur des adaptiven Systemes umfassen (Abschnitt 6.2)

Tabelle 6.1.: Verwendete Architektur-Begriffe in Kapitel 6; Software-Architektur eines adaptiven Systems (mit Adaptionsobjekt Architektur) = Infrastruktur + Dynamik + Architektur (Anwendung)

sind viele Aspekte gleichbedeutend für die Entwicklung für adaptive Systeme, die Anpassung an einem anderen Adaptionsobjekt vornehmen. Zum Beispiel kann bei zu wenig verfügbarer Bandbreite entschieden werden, die teurere UMTS-Verbindung zu benutzen (d. h. die Verbindungskomponente austauschen) oder den zu übertragenden Medienstrom (Daten) anzupassen (siehe Adaptionsmechanismen für Daten in Abschnitt 4.2). Für beide Adaptionsmechansismen bzw. Adaptionsgegenstände sind Beobachtungen der Umgebung sowie eine Planung/Entscheidung für eine Adaptionsmaßnahme notwendig. Daher gibt es Überschneidungen in der *Infrastruktur* und der Dynamik derartiger adaptiver Systeme. Um nicht die Vermutung aufkommen zu lassen, dass die Empfehlungen für die *Software-Architektur* nicht nur für den Adaptionsgegenstand *Architektur* Geltung besitzen, wird an manchen Stellen auch der einfache Begriff *Anwendung* benutzt.

6.1. Dynamische Aspekte — Ablaufplan

Neben der Spezifizierung statischer Aspekte (Bauplan) sind auch die dynamischen Aspekte (Ablaufplan) bei der Dokumentation einer Software-Architektur wichtig (vgl. [Bass *et al.* 2003, Kap. 1.3] und [Starke 2002, Kap. 2.1]). In den bisherigen Ausführungen wurden letztere noch nicht deutlich. Im theoretischen Teil zu Adaption, d. h. bei der formalen Definition einer Adaption in Abschnitt 3.6.7, wurde mit der Benennung der Aufgaben, die bei einer Adaption zu erfüllen sind,

bereits ein Workflow formuliert. Bei den Literaturansätzen wurde lediglich im Rahmen der Ausführungen zum Rainbow-Framework in Abschnitt 5.5.2 ein möglicher Adaptionskreislauf beschrieben; an anderen Stellen im Text konnte nur ein ungefährer Ablauf erkannt werden (Regelkreise, Trio ADL–AML–ACL u. a.). Ansonsten wurden in den Architekturdiagrammen/Abbildungen des vorhergehenden Kapitels weitestgehend nur statische Aspekte (Aufbau/Bauplan) einer adaptiven Software ersichtlich.

Der vorliegende Abschnitt 6.1 vermittelt Empfehlungen für den Ablaufplan einer Adaption in einem System. Es werden die dynamischen Aspekte in einem adaptiven System beschrieben.

Die dynamischen Aspekte einer Software-Architektur (hier Infrastruktur genannt) werden in gesonderten Diagrammen dokumentiert.[1] So statuieren Bass *et al.* [2003], dass jede Software-Architektur mindestens mit einer statischen und mit einer dynamischen Sicht dokumentiert werden muss. Keinesfalls sollten statische und dynamische Aspekte in der Dokumentation vermischt werden. (Bei diesem Punkt fällt auf, dass selbst in der Software-Architektur-Szene sehr angesehene Wissenschafter wie Garlan und Schmerl in [Garlan und Schmerl 2002, Abbildung 2] und in [Cheng *et al.* 2004, Abbildung 1] (in der vorliegenden Arbeit Abbildung 5.18 auf Seite 167) diese Regel nicht beachten.)

Für die Beschreibung der dynamischen Aspekte wird in der vorliegenden Arbeit ein Aktivitätsdiagramm gewählt (siehe Abbildung 6.1 auf der nächsten Seite). Die folgenden Ausführungen erläutern die einzelnen Aktivitäten, die bei einer Adaption des Systems vorgenommen werden.

Anmerkung zur Interpretation des Aktivitätsdiagramms

Bedingung für eine Transition und Lebenszyklus von Aktivitäten. Eigentlich entspricht Abbildung 6.1 auf der nächsten Seite keinem korrekten Aktivitätsdiagramm, da die Ausführung der Aktivität System läuft nach dem Hochfahren des Systems niemals abgeschlossen wird (erst beim Herunterfahren) und deshalb nie eine Transition zur waagerechten Gabelung (fork) stattfinden kann (vgl. [Hitz und Kappel 2003, Abschnitt 2.7]). Würde das Diagramm jedoch ohne die Aktivitäten System hochfahren und System läuft dargestellt werden (vom Startpunkt würde eine Transition direkt zur Gabelung erfolgen), dann könnte der *Rücklaufpfeil* von dem unteren waagerechten Synchronisationsbalken nicht verwendet werden. Das heißt, ein Kreislauf wäre nicht möglich. Denn der Rücklaufpfeil müsste bei der oberen Gabelung (die in dem Fall aufgrund zweier eingehender Transitionen zusätzlich eine Synchronisation darstellt) vergeblich auf die Transition vom Startpunkt

[1] Die UML 2 bietet für die Modellierung dynamischer Aspekte die Verhaltensdiagramme Use-Case-Diagramm, Aktivitätsdiagramm, Zustandsautomat, Sequenzdiagramm, Kommunikationsdiagramm, Timing-Diagramm und Interaktionsübersichtsdiagramm an [Jeckle *et al.* 2003].

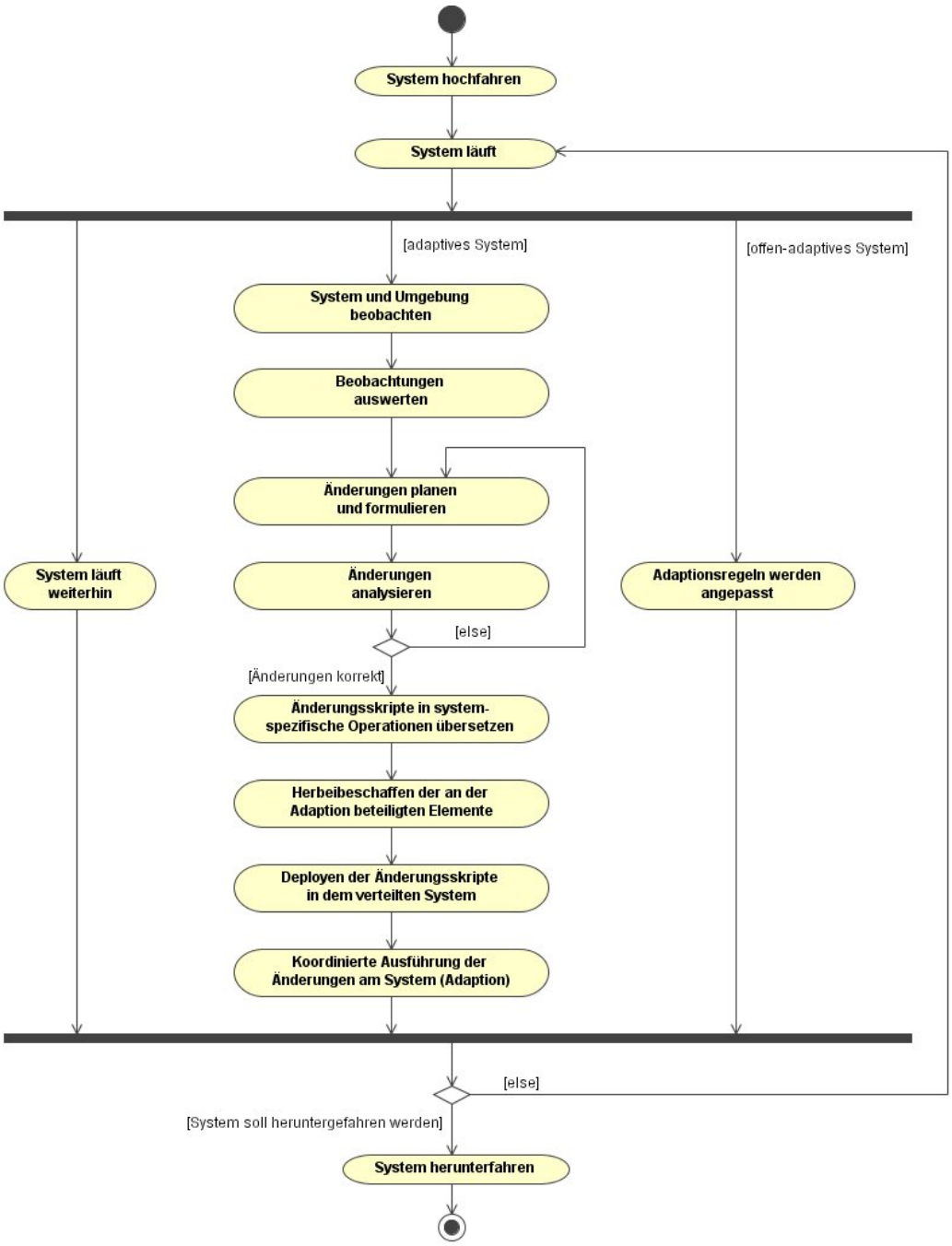

Abbildung 6.1.: Aktivitätsdiagramm für ein adaptives System. Durch die Überwachungsbedingungen nach der Gabelung (die im Vergleich zum Zustandsdiagramm nur im Aktivitätsdiagramm möglich sind) wird zugleich ein normales (links), ein adaptives (mittig) und ein offen-adaptives (rechts) System beschrieben.

warten, weil diese nur einmal vorgenommen wird. Deshalb beinhaltet das Diagramm diesen »Fehler«. Gleiches gilt für die Aktivität System läuft weiterhin. Der Autor konnte allerdings keine bessere Darstellungsform und kein geeigneteres Verhaltensdiagramm finden, das die dynamischen und nebenläfigen Aspekte besser modellieren lässt.

Adaption als präventive Aktion oder als Reaktion. Eine Vereinfachung wird vorgenommen, indem nur ein Aktivitätsdiagramm sowohl für die Adaption in Form einer präventiven Aktion als auch für die Reaktionsform (vgl. Abschnitt 3.6.7) angegeben wird.

Im Aktivitätsdiagramm ergibt sich nach dem Start für die Aktivitäten die (vereinfachte) Reihenfolge System hochfahren, ..., System und Umgebung beobachten und auswerten, planen ... und schließlich Ausführung der Änderungen im System (Adaption). Streng genommen ermöglicht dieser Ablauf nur eine Adaption als Reaktionsform, da zuerst die Beobachtungen vorgenommen werden und danach in Reaktion auf eine erkannte Änderung die Adaption erfolgt. Damit wird eine Präventivadaption ausgeschlossen, da bei dieser zuerst die Änderungsaktion und danach die Beobachtung, ob das Resultat ein gutes ist, erfolgen. Für diesen Sachverhalt müssten die Aktivitäten System und Umgebung beobachten und Beobachtungen auswerten nach der Aktivität koordinierte Ausführung der Änderung am System (Adaption) am Ende der Aktivitätsfolge angeordnet sein. Diese erforderliche Reihenfolge kann sich allerdings ergeben, wenn der Kreislauf im Gange ist, da in der Iteration i die Adaption erfolgt und in der Iteration $i + 1$ die Beobachtung und Auswertung. Demnach können Präventiv- und Reaktionsadaption erfolgen, sofern ggf. die Aktivitäten für die Beobachtung und die Auswertung ausgelassen oder nach der Adaption ausgeführt werden.

6.1.1. Laufendes System

In den Aktivitäten System läuft und System läuft weiterhin wird das System ausgeführt. Das System erfüllt seine normale, d. h. nicht-adaptive, Funktionalität. Obwohl als zwei getrennte Aktivitäten eingezeichnet, sind diese Aktivitäten als die gleichen anzusehen. Ebenso finden sie andauernd statt, d. h., sie werden eigenlich erst dann beendet, wenn die Aktivität System herunterfahren ausgeführt wird.

Eine Transition von der Aktivität System läuft weiterhin zum unteren Synchronisationsbalken findet statt, wenn die koordinierte Ausführung der Änderung am System (Adaption) stattgefunden hat.[2]

[2] Für solche Fälle gibt es eigentlich bedingte Transitionen. Eine bedingte Transition wird allerdings »nur dann ausgeführt, wenn die Aktivität des vorangegangene Zustands abgeschlossen *und* die Bedingung erfüllt ist« [Hitz und Kappel 2003, S. 163]. Da der vorangegangene Zustand (System läuft weiterhin) genau genommen nie abgeschlossen wird, wird auf diese Modellierungsform verzichtet.

6.1.2. System und Umgebung beobachten

Eine wichtige Aktivität, die ein adaptives System ausführen muss, ist die Beobachtung des Systems (sich selbst) und der Umgebung (alles, was außerhalb des Systems ist).

Der Beobachtungsgegenstand kann sehr groß sein. Er ist abhängig davon, *an was* eine Adaption erfolgen soll und welche Motivation dahintersteht. Diese zwei Dinge sind getrennt zu beachten. Denn für ein gegebenes System, das eine Anpassung an die heterogene Ausführungsumgebung vornimmt, kann zusätzlich die Motivation bestehen, dass das System zudem robust sein soll. Ein Beispiel hierfür wurde in den Ausführungen zum Fazit für die Regelungssysteme (5.2.6) aufgezeigt. Sobald die Adaptionsregeln zu komplex sind oder Widersprüche zu beachten sind, kann die Adaption durch eine Rückkoppelung erfolgen. Genau in dem Fall muss die adaptive Anwendung nicht nur die Bedürfnisse des Clients in Erfahrung bringen, sondern auch sich selbst beobachten, damit die Auswertung hinsichtlich der eigenen Performance erfolgen kann.

6.1.3. Beobachtungen auswerten

Die gesammelten Beobachtungen müssen ausgewertet werden. Je nach Informationstyp können die Beobachtung unbearbeitet verwendet werden oder weitere Auswertungsprozesse erfordern, um aus den Rohdaten verwertbare Informationen in Erfahrung zu bringen.

Die Auswertungsaktivität in einem System ist neben der Grundvoraussetzung des Beobachtens die wichtigste Tätigkeit in einem adaptiven System, da sie entscheidet, in welchem Umfang und mit welcher Qualität eine Adaption erfolgt. Die für einen Beobachtungsgegenstand, an den eine Anpassung erfolgen soll, notwendige Planung einer Adaptionsoperation besitzt zwar auch eine große Bedeutung (weil dessen Schwierigkeit darin besteht, eine passende Operation auszuwählen). Aber so lange ein System auswerten kann, ob seine erfolgte vermeintliche Anpassungsoperation adäquat war, kann es die Planung der Operationen weniger qualitativ gestalten und nach endlicher Zeit zu einem guten Anpassungsergebnis kommen. Gerade um der Robustheit Rechnung zu tragen, ist es wichtiger, die eigene Performance beobachten zu können (die Beobachtung der Performance erfolgt durch die Auswertung der gemessenen Rohdaten) als eine gute Planung zu haben, die versucht, alle Konstellationen in der Umgebung und im System zu berücksichtigen (Typ-1-Systeme).

6.1.4. Änderungen planen und formulieren

Keine Adaption kann stattfinden, ohne die Änderungen am System vorher bestimmt zu haben — auch bei einer zufällig bestimmten Präventivaktion (vgl. Seite 75).

Wie in Abschnitt 4.1.4 bereits deutlich wurde, umfasst die Adaption zwei verschiedene Arten. Es kann die eigentliche Anwendung (Architektur) angepasst werden *und* auch die Adaptionsarchitektur (Infrastruktur). Das hat zur Folge, dass es ebenso zwei Planungsarten gibt: eine Planung für Änderungen an der Architektur und eine Planung für Änderungen an der Infrastruktur.

Planung für Änderungen an der Architektur. Bei dieser Planung wird entschieden, welche Anpassung an der Architektur (oder allgemein Anwendung, d. h. auch Daten und Kommunikation etc.) vorgenommen werden soll. Für eine zu erfolgende Adaption stehen dem Planer verschiedene Möglichkeiten offen, im Fall von Komponenten verschiedene Alternativen, auch genannt »reuse asset base« [Laddaga 2001]. Für andere Adaptionsmechansismen müssen entsprechend andere Verfahren verfügbar sein — bei der Inhaltsadaption sind dies meist Filter (vgl. [Springer 2004]).

Die Einsetzbarkeit der Adaptionsmechanismen hängt neben der Verfügbarkeit von Alternativen auch von der Umgebung ab, in der die Mechanismen eingesetzt werden können. Beispielsweise ist es nutzlos, eine GSM-Komponente durch eine UMTS-Komponente zu ersetzen, um die verfügbare Bandbreite zu erhöhen, wenn in der Umgebung kein UMTS-Netz verfügbar ist. Neben dieser Einsetzbarkeit, die durch die Umgebung bestimmt ist, muss bei der Planung auch berücksichtigt werden, ob alle notwendigen Voraussetzungen, die eine einzusetzende Komponente an ihre Ausführungsplattform stellt, erfüllt werden können (z. B. *requires*-Schnittstellen).

Planung für Änderungen an der Infrastruktur. Im einfachsten Fall wird bei einer Anpassung der Infrastruktur das Beobachten kontrolliert, indem ein Monitor oder eine Gauge nachgeregelt wird, Daten in einem präziseren Zeitintervall zu liefern. Schwieriger kann es für die Planung der Beobachtung werden, wenn entschieden werden muss, welche Beobachtungen an welchem Ort notwendig sind. In dem Fall muss der Planer überprüfen, welche Monitore ihm zur Verfügung stehen, wo diese eingesetzt werden können und welche Kosten oder andere Aspekte damit verbunden sind.

Deterministik vs. Probieren. Je nach Ansatz kann die Adaptionsmaßnahme klar bestimmt werden oder durch zufälliges Probieren erfolgen. Viele adaptive Systeme bestimmen die notwendige Handlung durch Regeln. Doch gerade das Vorbild, die Natur, hat durch zufällige Mutationen das größte Angepasstsein im Laufe vieler Jahre im Rahmen der Evolution geschaffen. Das Planen von Änderungen in der Software sollte deshalb auch diese Möglichkeit nicht außer Acht lassen, wobei mithilfe einer sehr guten Auswertung eine Software sehr großes Adaptionspotential sowie Robustheit besitzt.

Autonomie. Abhängig vom Grad der Autonomie (Abschnitt 3.7) kann bei der Planungsaktivität auch der Mensch mit einbezogen werden. Bei einer teilautomatischen Durchführung der Adaption kann der Mensch die komplette Planung übernehmen (Typ 3) oder die Adaptionsmaßnahme wird vom System vorgeschlagen (Typ 2 und 3). Deshalb kann in dieser Aktivität eine Interaktion mit dem Nutzer vorgenommen werden.

Resultat. Resultat einer vorgenommen Planung von Adaptionsoperationen oder allgemein Änderungen[3] der Anwendung ist die Formulierung der vorzunehmenden Operationen. Die Formulierungen dienen zur Kommunikation mit der Ausführungseinheit, die die Änderung am System letztendlich vornimmt.

6.1.5. Änderungen analysieren

Nach erfolgter Formulierung der Änderungen ist es je nach verwendetem Adaptionsmechanismus und dem damit verbundenen Risiko notwendig, die angestrebte Adaptionsoperation vor der Ausführung zu analysieren. In der Zusammenfassung zu den Adaptionsmechanismen der Software-Architektur (Abschnitt 4.4) wurden einige Punkte beschrieben, die dabei zu beachten sind. Abschnitt 5.1.1 hat hierfür Unterstützung bei ADLs angeführt, wobei als gesonderte Beschreibungssprache die ACLs genannt wurden, und ein Vorgehen skizziert.

Stellen sich die formulierten Änderungen als nicht korrekt heraus, sodass das resultierende System bei Ausführung der Änderungen keine korrekte, inkrementelle Evolution umfasst, dann muss eine Neuplanung der Adaptionsoperation durchgeführt werden. Dadurch erfolgt ein Rücksprung zu der vorherigen Aktivität Änderungen planen und formulieren (vgl. Abbildung 6.1 auf Seite 182).

Nicht bei allen Adaptionsmechanismen an der Software-Architektur (bei anderen Adaptionsobjekten, z. B. der Inhaltsadaption, überhaupt nicht) müssen die vorzunehmenden Änderungen vor ihrer Ausführung am laufenden System analysiert werden, wenn sie nicht risikobehaftet sind.

Die Analyse der vorzunehmenden Änderungen hinsichtlich Wahrung der Integrität und der Konsistenz wird als separate Aktivität identifiziert, da die während der Planung formulierten Änderungen nicht immer korrekt sein müssen. Die Regeln, denen die Planung unterliegt, können falsch formuliert sein. Auch möglich ist es, dass die Planung durch zufälliges Probieren nicht immer ein korrektes Resultat liefert. Des Weiteren muss das System bei einer Adaption mit den Autonomiegraden 0/basic (nicht-adaptive) oder 1/managed (vgl. Abschnitt 3.7)

[3] Eine Änderung entspricht einer Adaption, wenn die Änderung in einem besseren Angepasstsein resultiert. Insbesondere bei einer präventiv erfolgten Aktion (siehe Seite 75) ist nicht a priori bekannt, ob die Änderung das System verbessert.

die vom Menschen formulierten Änderungen prüfen, da der Mensch einen Fehler gemacht haben könnte. Deshalb existieren die Aktivität zur Planung und Formulierung der Änderungen getrennt von deren Überprüfung. Allerdings ist es von Vorteil, wenn Planungs- und Änderungsanalyseaktivitäten eng miteinander zusammenarbeiten, um effizient und effektiv vorzugehen. Deshalb können die zwei Aktivitäten als Teilaktivitäten innerhalb einer großen Aktivität Planung angesehen werden, deren Resultat formulierte Adaptionsoperationen sind, die bei Anwendung wieder ein korrektes System als Resultat erzeugen.

6.1.6. Änderungsskripte in systemspezifische Operationen übersetzen

Die zuvor in einer gesonderten Spezifikationssprache formulierten Änderungen sind noch abstrakt, da sie nur bezüglich eines Modells des Systems gültig sind. Für das unterliegende System (bestimmtes Komponentenframework, IT-System) muss deshalb eine Übersetzung der formulierten Änderungen in systemspezifische Änderungsoperationen erfolgen.

Bei den übersetzten Änderungsskripten ist zu beachten, dass diese für die Installation einer Komponente auf einem Knoten außerdem Bootstrapping-Code bereitstellen müssen.

6.1.7. Herbeischaffen der an der Adaption beteiligten Elemente

Zusätzlich zu den zuvor übersetzten *Änderungsskripten*, die die Operationen *beschreiben*, sind noch weitere Bestandteile für die Adaptionsoperation notwendig:

- hinzuzufügende oder auszutauschende *Komponenten und Konnektoren*[4] der Anwendungsarchitektur, der Systemarchitektur (bei Replikation eines Dienstes) und der Adaptionsarchitektur sowie
- ihre *Beschreibungen* (Schnittstelle, nicht-funktionale Eigenschaften, initiale Parameter).

Diese drei Bestandteile (Änderungsskripte, Komponenten, Beschreibungen) werden für jede Site[5] paketiert.

[4] Im Folgenden werden Konnektoren nicht gesondert betrachtet, da sie lediglich aufgrund der Abstraktion in der Software-Architektur existieren. Im Idealfall stellen sie einen einfachen oder einen etwas umfangreicheren Link (z. B. einen Bus) zwischen zwei Komponenten dar, in anderen Fällen können sie komplexe Code-Fragmente wie z. B. ein Adapter oder ein Konnektor für den Zugriff auf Datenquellen (inkl. Initialisierungs- und Aufräum-Code) sein; vgl. [Mehta *et al.* 2000]. Aufgrund der Tatsache, dass Konnektoren, erzeugbare oder existierende Code-Fragmente sind, werden sie nicht explizit, sondern als »Komponenten« behandelt.

[5] Die verschiedenen Hardwareknoten im System, die aufgrund der Verteilung die Änderung lokal und selbst vornehmen müssen.

Im Folgenden wird der Verbund der drei notwendigen Bestandteile *Adaptionspaket* genannt.

Push- und Pull-Prinzip. Das soeben beschriebene Verfahren entspricht dem Push-Modell, bei dem alle notwendigen Bestandteile zu der Zielplattform transferiert werden, worauf im Anschluss die Adaption erfolgen kann. Auch denkbar ist ein Pull-Prinzip, bei dem nur minimale Logik zu der Zielplattform transferiert wird und erst von dort aus die notwendigen Bestandteile aus den Datenbanken geladen werden. In dem Fall erfolgt das Herunterladen der Bestandteile zusätzlich in der nachfolgend beschriebenen Aktivität.

6.1.8. Deployen der Änderungsskripte in dem verteilten System

Mit Fertigstellung des Adaptionspaketes kann dieses an die verschiedenen Sites zugestellt werden, damit diese im nächsten Schritt die Adaption vornehmen können. Wird ein Pull-Prinzip für die Versorgung des Rekonfigurationsprozesses mit Komponenten verwendet, dann erfolgt in dieser Aktion zuerst das Transferieren eines minimalen Ausführungsskriptes zu dem jeweiligen Host, das im Anschluss daran das Herunterladen (pull) der notwendigen Bestandteile des Adaptionspaketes vornimmt.

6.1.9. Koordinierte Ausführung der Änderungen im System (Adaption)

Sobald ein Änderung ausgewählt, formuliert, auf Validität geprüft wurde etc., kann sie ausgeführt werden. Selbstverständlich muss die unterliegende Infrastruktur die Änderungen zur Laufzeit auch unterstützen, d. h., die gewünschten dynamischen Adaptionsoperationen müssen in der gewählten Plattform möglich sein. Ist die Änderung nicht zur Laufzeit möglich, erfolgt die Adaption durch ein Herunterfahren und ein anschließendes Wiederhochfahren des Teilsystems.

In einem verteilten System ist eine koordinierte Änderung besonders schwierig, da die Änderungen auf verschiedenen Knoten gleichzeitig vorgenommen werden müssen. Die Knoten unterliegen aufgrund der Verteilung der Anwendung keiner zentralen Kontrolle. Stattdessen muss eine Koordinierung der Abläufe zwischen den Knoten vorgenommen werden. Andererseits kann auch argumentiert werden, dass gerade in einem verteilten System mit zeitlichen Ausfällen zu rechnen ist,[6] weshalb eine sichere Adaption durch Herunterfahren des Teilsystems erfolgen kann. Kommt ein Herunterfahren nicht in Frage (in sog. Mission-critical-Systemen) oder ist aufgrund der damit verbundenen

[6] Diesen Sachverhalt hebt Leslie Lampert (der Erschaffer von LaTeX) bereits in seiner Definition für ein verteiltes System hervor (zitiert nach [Weber 1998, S. 1]): »Ein verteiltes System ist ein System, mit dem ich nicht arbeiten kann, weil irgendein Rechner abgestürzt ist, von dem ich nicht einmal weiß, daß es ihn überhaupt gibt.«

Kosten keine redundante Hardware verfügbar, dann muss die Adaption im laufenden Betrieb erfolgen.

6.1.10. Adaptionsregeln anpassen

In Abschnitt 3.7 wurde für adaptive Systeme eine Untermenge definiert: offen-adaptive Systeme. Bei einem Software-System mit dieser Eigenschaft können zur Laufzeit (ohne Herunterfahren des Systems) die Adaptionsregeln geändert werden. Falls die Software die Änderung selbst vornehmen kann, dann ist sie lernfähig. Diese zwei Tätigkeiten (Anpassung der Regeln von außen oder durch Software selbst) werden innerhalb der Aktivität Adaptionsregeln werden angepasst ausgeführt.

6.2. Statische Aspekte — Infrastruktur

Aus den im vorhergehenden Abschnitt formulierten Aktivitäten, die bei einer Adaption ausgeführt werden, können die notwendigen Komponenten, die die Infrastruktur für ein adaptives System aufweisen muss, abgeleitet werden. In diesem Abschnitt werden die sog. statischen Aspekte einer (hier adaptiven) Software-Architektur beschrieben.

Im Folgenden werden die statischen Aspekte für eine adaptive Anwendung beschrieben. In Abbildung 6.2 auf der nächsten Seite ist im Voraus eine bildliche Zusammenfassung gegeben.

6.2.1. Informationslieferanten — Monitore

Entsprechend der in 6.1.2 beschriebenen Aktivität System und Umgebung beobachten muss es Informationslieferanten geben, die die Adaptionslogik mit notwendigen Informationen zur Umgebung und des Systems beliefern. Die Beobachtungen werden durch Monitore vorgenommen.

Wie bereits in Abschnitt 6.1.2 deutlich wurde, können die Beobachtungen sehr unterschiedlich und breit ausfallen. Sie können Metainformationen in der Dienstanfrage oder Kontrollinformationen während der Kommunikation sowie mittelbare und unmittelbare (weil physisch entfernte, z. B. (SNMP) Agenten) Informationen von Sensoren sein, die im System positioniert sind.

Bis auf einige Ausnahmen können die notwendigen Informationen unter dem Begriff Kontext zusammengefasst werden (vgl. Abschnitt 3.6.1). Deshalb kann in vielen Anwendungsdomainen der Informationslieferant durch einen Kontext-Dienst gegeben sein. Dabei ist sich der Entwickler nicht immer bewusst, dass er »Kontext« verarbeitet. In manche Anwendungen sind die Monitor-Komponenten in einer ad-hoc-Weise entwickelt, indem die Informationsgewinnung implizit in der Anwendung implementieren ist (z. B. Abfragen des `/proc`-

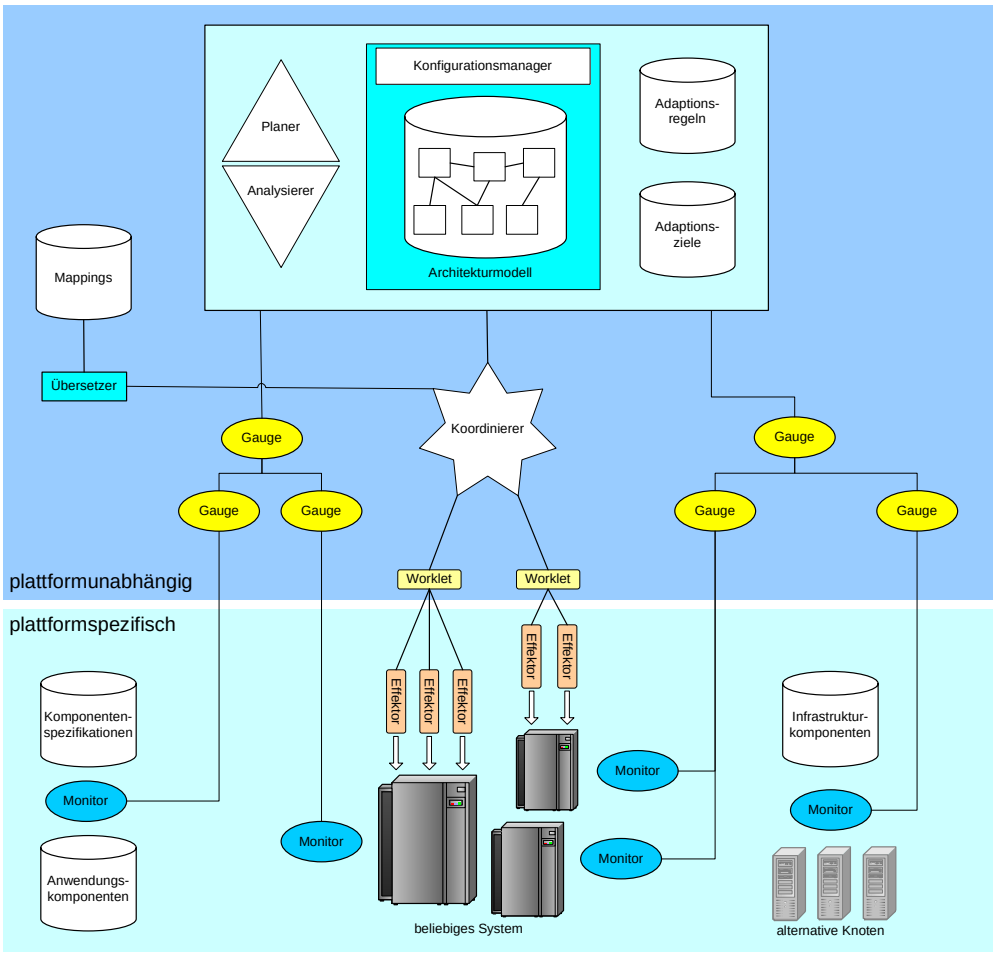

Abbildung 6.2.: Infrastruktursicht für Adaption an einem beliebigen, verteilten System

Pseudodateisystems in Linux, um Informationen zum System- und Netzwerkstatus zu erhalten). Das bedeutet, bei ihnen erfolgt keine Nutzung durch einen expliziten Kontext-Dienst. Zumal nicht wenige notwendige Informationen zudem ad-hoc vor vorliegen, z. B. die Metainformationen bei der Anfrage an einen Webserver.

Gegebenenfalls können die Beobachtungen in einem persistenten Speicher permanent gespeichert werden. Dies erlaubt die Analyse von historischen Daten. Eine solche Speicher-Komponente wird in der vorliegenden Arbeit dennoch nicht explizit für die Infrastruktur angeführt.

Je nach Systemart und Beobachtungsgegenstand fallen die Beobachtungen unterschiedlich aus. Sie können durch Ereignisbenachrichtigung erfolgen — beispielswiese per Observer-Muster (Publish-Subscribe-Muster) [Buschmann *et al.* 1998], [GoF 1996] — oder explizit durch Abfrage (Polling) der Monitore (Abschnitt 6.2.1 auf Seite 189). Auch ist das Erhalten von Informationen im Rahmen der Kommunikation mit dem Partner möglich (als Kontrollmeldungen). Ein einfaches Beispiel hierfür ist die Stau- und Paketverlustbehandlung in TCP (eine exzellente Übersicht bietet [Springer 2004, Abschnitt 2.1.2.1]), bei der verschiedene Informationen vom Empfänger an den Sender geschickt werden und diesen über den Status von gesendeten Paketen unterrichten.

Für die Beobachtung sollte es möglich sein, die Anzahl, den Umfang und die Detailliertheit der Beobachtungen und Messungen justieren zu können, um Overhead bei den Messungen zu vermeiden und keine Kommunikationsbandbreite durch unnütze Beobachtungen zu verschwenden [Oreizy *et al.* 1999].

Monitore = Probes + Resource Discovery + Observer + X

In der Literatur werden für Informationsgewinnung verschiedene Komponenten deklariert. Beispiele hierfür sind die Probes in der Software-Architektur des DASADA-Projektes und im Rainbow-Framework (vgl. Abschnitt 5.5.2), Resource Discovery ebenso im Rainbow-Framework und Observer in Oreizy *et al.* [1999]. Bei genauer Betrachtung sind die propagierten Komponenten für die Informationsgewinnung nicht klar gegeneinander abgrenzbar, wobei sich Überschneidungen, aber auch Differenzen ergeben (vergleichbar mit Kontext, Einsatzumgebung und Betriebsumgebung; wie in Abbildung 3.3 auf Seite 46 dargestellt). Reilly *et al.* [2002] nennen gar eine Reihe von Informationsgewinnungskomponenten — Monitore, Probes, Gauges sowie LookUp Service, Discovery Service und Leasing Service —, ohne dass spezielle Aufgabengebiete beschrieben oder ersichtlich werden.

Da in der vorliegenden Arbeit keine spezielle Anwendung zu entwickeln ist, für die bestimmte Dienste/Services als Informationsgewinnungskomponenten gegeben sind, wird der pragmatische Weg gewählt. Ähnlich zum Resultat, das bezüglich des Auslösers für eine Adaption in Abschnitt 3.5.4 gezogen wurde, ist für Adaption die Vereinigung aller

Komponenten notwendig. Die Beobachtungen, die durch die Monitore in der vorliegenden Arbeit vorgenommen werden, umfassen die Summe der Beobachtungen der vorgenannten Komponenten.

Der Vollständigkeit halber und für ein erweitertes Verständnis (z. B. für Resource Discovery) der notwendigen Charakteristik des Beobachtens werden bis zum Abschnitt Informationsgewinnung als Herausforderung auf der nächsten Seite die einzelnen Komponenten beschrieben und erläutert, warum diese nicht ausreichend für eine Informationsgewinnung sind.

Probe. Eine Probe wird in [Gross *et al.* 2001] definiert als

> »individual sensor attached, either statically or dynamically, to a running program. Probes emit events that describe some aspect of a program's execution, either at specific point in time or over some duration.«

Daraus folgt, dass Probes nur die Software (Komponenten) beobachten. Insbesondere die Umgebung des Systems wird dadurch nicht berücksichtigt, ebenso nicht die Interaktion mit Nutzern und anderen Systemen.

Resource Discovery. Besonders ersichtlich wird eine unklare Abgrenzbarkeit für den Resource-Discover-Dienst (kurz RD-Dienst), den das Rainbow-Framework zusätzlich zu den Probes in der Systemschicht deklariert (siehe Abschnitt 5.5.2).

Ein Resource-Discover-Dienst RD-Dienst ist definiert als

> »service that returns the location of matching resources in response to a query with requirements.« [Vanthournout *et al.* 2004]

Dabei bezeichnet eine Ressource

> »any source of supply, support, or aid a component in a networked environment can readily draw upon when needed.« [Vanthournout *et al.* 2004]

Als Beispiele für Ressourcen nennen Vanthournout *et al.* Dateien, Messungen, CPU-Zyklen, Speicher, Kontrollgeräte, Foren, Online-Shops etc.

Damit ergibt sich eine Überschneidung zu den Probes, die ebenfalls Informationen zur Auslastung der Ressourcen eines Systems (Speicher, verfügbare Netzwerkbandbreite) geben. Beide Komponenten sind nicht klar trennbar.

Anhand der Ressourcen-Definition lässt sich gut ableiten, welche Informationen benötigt werden. So muss nicht nur die Anwendungsstruktur beobachtet werden, sondern auch die Adaptionsarchitektur. Zum Beispiel muss ein RD-Dienst melden, wenn neue Komponenten in der Komponentendatenbank (Abschnitt 6.2.3) verfügbar sind oder Hardware-Ressourcen zugänglich sind.

Observer. Oreizy *et al.* [1999] benennen für die Informationsgewinnung sog. Observer. Diese kommen dem Verständnis der Monitore, wie sie in der vorliegenden Arbeit für eine Informationsgewinnung herangezogen werden, am nächsten. Die Observer werten das Verhalten der selbst-adaptiven Anwendung aus und beobachten seine »Betriebsumgebung«. Entsprechend der Definition für selbst-adaptive Software, die Oreizy *et al.* [1999] angeben, betrachten sie als Informationen alles, was beobachtbar ist (vgl. Abschnitt 3.5.2). Mit dieser Formulierung besitzen Oreizy *et al.* das gleiche Verständnis für die notwendigen Informationen für (selbst-) adaptive Software, wie es innerhalb des Abschnittes »Kontext + X« auf Seite 51 formuliert wurde, d. h. alles was beobachtbar ist.

Fazit. Aufgrund der unklaren Abgrenzbarkeit werden in der vorliegenden Arbeit Probes, RD-Dienst und Observer etc. gleichgestellt und Monitore genannt. Monitore sammeln für eine adaptive Software Informationen, wobei sie alles einbeziehen können, was beobachtbar ist.

Auswertung — Informationsgewinnung als Herausforderung

Im Rahmen der Software-Entwicklung sollte es — auch wenn viele benötigten Informationen ad-hoc vorliegen — Ziel sein, wiederverwendbare Beobachtungsmechanismen zu bauen. Im Rahmen der Kontext-Gewinnung bietet beispielsweise [Löschau 2002] einen minimalen Ansatz für wiederverwendbare Informationsmechanismen auf Basis eines Frameworks.

Des Weiteren besteht eine Schwierigkeit bei der Unterstützung von neuen als auch alten (bereits vorhandenen) Komponenten, die beobachtet werden müssen.

Bei neuen Komponenten stellt sich die Frage, wie diese entwickelt werden können, damit sie leichter beobachtet werden können. Bei Benutzung von Reflection (siehe Abschnitt 5.3) ist die Beobachtbarkeit bereits durch die Architektur gegeben, da ein reflectives System Introspection unterstützt. Doch bei Reflection und anderen Architekturansätzen ist es schwierig, den Zugriff auf Komponenten derart einzuschränken, dass nur Befugte Zugriff auf den internen Zustand der Komponente haben.

Bei bereits vorhandenen Komponenten besteht kein Zugriff auf die Interna, da im Rahmen der immer noch aktuellen objektorientierten Software-Entwicklung das Prinzip der Kapselung angewendet wird. Einen Ansatz bietet der im Rahmen des DASADA-Projektes [Salisan 2004] entwickelte ProbeMeister [Objs 2002]. Sofern Java-Bytecode vorliegt, ermöglicht ProbeMeister das dynamische Einfügen von Code (z. B. zum Monitoring) in die Anwendung. Ein anderer Ansatz ist das Einfügen von Vermittlern (Mediator als Konnektor), die Aufrufe an Windows' DLLs dynamisch überwachen und ersetzen können [Balzer und Goldman 1999].

6.2.2. Gauges

Insbesondere im Rainbow-Framework (Abschnitt 5.5.2) wurden die im Rahmen des DASADA-Projekts [Salisan 2004] entwickelten Gauges deutlich. Gauges haben die Aufgabe, die von den Monitoren gesammelten Rohinformationen zu interpretieren. Gauges werden gemäß Gross *et al.* [2001] beschrieben als

> »software entities that gather, aggregate compute, analyze, disseminate and/or visualize measurement information about software systems. Gauges support a simple configuration interface. The proposed DASADA gauge standard includes the concept of a "Gauge Reporting Bus," which is specially for communicating gauge reports to consumers (who might, e. g., authorize repairs). Consumers supply callbacks to the reporting bus, which are called when an event of interest occurs, allowing them to respond to the event.«

Gauges übernehmen die in Abschnitt 6.1.3 beschriebene Aktivität des Auswertens der Beobachtungen. Bei der Beschreibung zu den Gauges wird außerdem das in Abschnitt 5.4.1 beschriebene leistungsfähige Blackboard-Muster deutlich, das die Abstraktion vornimmt und über eine Auswertungseinrichtung (Evaluator) in der obersten Ebene Feeback zurückpropagieren kann.

Wie bereits in der Definition deutlich wird, haben die Gauges nicht nur die Aufgabe, die Rohdaten auszuwerten, sondern diese auch zu visualisieren. Damit besitzen Gauges auch in nicht-adaptiven Systemen (genauer: Systeme mit der Autonomiestufe 1 bzw. managed; vgl. Abschnitt 3.7), eine wichtige Rolle. Der Autor empfiehlt, den Gauges lediglich den Part des Modells im MVC[7]-Muster [Buschmann *et al.* 1998], [GoF 1996] zuzuweisen, da die Grundaufgabe der Gauges die Auswertung von Informationen ist. Die gesamte Interaktion mit dem Menschen sollte extern vorgenommen werden — und wird in den vorliegenden Empfehlungen für eine Software-Architektur nicht explizit berücksichtigt.

Anmerkung. Gegebenenfalls können die ausgewerteten Beobachtungen in einem persistenten Speicher permanent gespeichert werden. Dies erlaubt die weitergehende Analyse von historischen Daten. Hier gilt die gleiche Anmerkung wie zuvor bei den Monitoren, dass die persistente Komponente für die Infrastruktur keine Berücksichtigung findet.

Die Auswertung von Informationen durch Interpretation, Aggregation, Reasoning, Abstraktion etc. ist mit der Kontext-Verarbeitung ein wichtiger Teilbereich des Forschungsbereichs Context-Awareness.[8]

[7] Model-View-Controller

[8] Dennoch umfassen die Beobachtungen mehr als Kontext; vgl. Abschnitt 3.6.1.

QoS-Komponente für die Evaluation

Für die Anwendbarkeit von geschlossenen Regelkreisen ist Rückkoppelung (Feedback) notwendig. In der vorliegenden Arbeit werden geschlossene Regelkreise befürwortet, wenn das Modell des offenen Regelkreises nicht mehr ausreichend ist (für Erreichung von Robustheit, bei zu hoher Komplexität und für Typ-2-Systeme; vgl. die Kontroll-Theorie in Abschnitt 5.2).

Kokar *et al.* [1999] empfehlen für die Rückkoppelung eine separate QoS-Komponente. Entgegen klassischen Regelungssystemen kann in Software-Systemen die Rückkoppelung nicht direkt erfolgen. In manchen Fällen muss für das Feedback eine Funktion auf Basis des Vergleichs von Eingabe- und Ausgabevariablen des Systems (Plant) einen Feedback-Wert berechnen. In anderen Fällen kann die Rückkoppelung extern durch den Nutzer erfolgen (dargestellt als $\overline{\delta}$ in den Abbildungen 5.5 bis 5.7 auf Seiten 138–140).

Eine separate QoS-Komponente ist zwar notwendig, aber ihre Funktion wird durch die Kombination von Monitoren und Gauges übernommen, die das System beobachten und evaluieren. Deshalb gibt es in der Infrastruktur keine spezielle QoS-Komponente, sondern sie wird durch die vorgenannten Komponenten realisiert.

6.2.3. Komponentendatenbank für die Anwendungsarchitektur

Adaptivität, also das Angepasstsein, wird durch alternative Wege erreicht. Anpassungsvarianten gibt es viele; Unterkapitel 4.1 hat die in der vorliegenden Arbeit betrachteten Mechanismen für die Anpassung der Software-Architektur beschrieben. Die Adaptionsmechanismen für die Adaption der Anwendungsarchitektur (Komponenten hinzufügen, entfernen, austauschen) resultieren in der Notwendigkeit alternativer Komponenten (und Konnektoren, siehe Fußnote 4 auf Seite 187) für das System.

Eine Datenbank DB mit Komponenten hält alternative Komponenten — auch genannt »reuse asset base« [Laddaga 1999] — für die Adaptionsmechanismen der Anwendungsarchitektur vor. Sie stellt neue als auch alternative Komponenten bereit und nimmt alte Komponenten auf. Die Datenbank enthält auch alte Versionen einer Komponente mit der gleichen Funktionalität, um im Fall einer fehlerhaften Komponente wieder die alte Version der Komponente installieren zu können. Die neue Komponente wird revidiert und für die Entwicklung einer Nachfolgeversion einer Analyse unterzogen

Auch für einen Adaptionsmechanismus in der Systemarchitektur, die Replikation, ist die Datenbank mit Komponenten notwendig. Soll eine Komponente auf einen Knoten repliziert werden, wird die Komponente aus der Datenbank beschafft und auf dem Knoten installiert. Dies erfolgt, wenn der Knoten in einem separaten Adressraum läuft. Bei einer Replikation auf einen Knoten innerhalb des gleichen

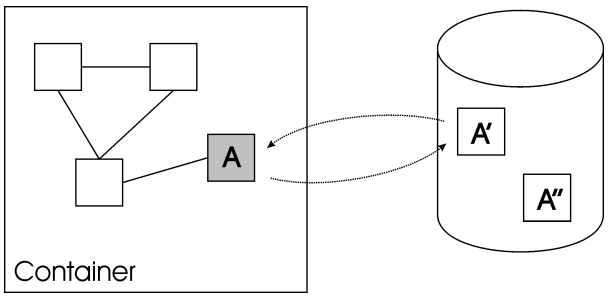

Abbildung 6.3.: Läuft die verteilte Anwendung in einem Container, ist eine Komponentendatenbank für alternative Komponenten notwendig.

Adressraumes (z. B. Prozessor oder Knoten in einem Parallelrechner mit Shared-Memory) kann die Replikation durch die Duplikation der bereits auf einem Knoten laufenden Komponente geschehen. In dem Fall ist nicht erst das Beschaffen der Komponente aus einer zentralen Datenbank notwendig (in dem Fall kann auch der Shared-Memory als Datenbank für Komponenten angesehen werden).

Einschränkung

Nicht jede adaptive Software-Architektur muss eine Komponentendatenbank haben — auch ohne die Klausel bei der Replikation. Die Notwendigkeit für einen Speicher für Komponenten hängt von der Art des verteilten System ab. Es können hier zwei Fälle unterschieden werden: Container-artig und ohne Container.

Container-artig. Dieser Typ verteilte Anwendung kann als Container gesehen werden, in dem die Komponenten, die die Anwendung darstellen, deployt sind (siehe Abbildung 6.3). Bei einem Austausch einer Komponente in der Software-Architektur wird die alte Komponente entfernt und die sie ersetzende Komponente wird aus einer Komponentendatenbank beschafft.

Kein Container, z. B. Service-orientierte Architektur Die andere Art verteiltes System besteht aus einer Anwendung mit lose gekoppelten Diensten (z. B. Währungsrechner, Wetterinformationsdienst, Übersetzungsdienst), z. B. im Internet (siehe Abbildung 6.4 auf der nächsten Seite). Die Dienste sind dadurch charakterisiert, dass sie von Dritten angeboten werden oder keinem zentralen Management unterstehen.[9]

[9] Zum Beispiel Administrator schließt mehrere Drucker an das Netzwerk an, die Findung und Auswahl eines passenden Clients wird nicht fest vorgegeben, sondern obliegt dem Client (womöglich ad-hoc angeschlossenes Notebook); anderes Beispiel: mehrere vernetzte Geräte (Toaster, Kaffemaschine, Gebäudesensoren, Internet-Uplinks etc.) im intelligenten Haus, die sich selbst finden müssen.

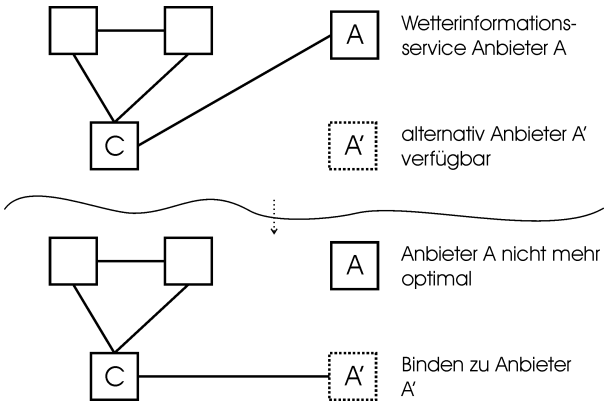

Abbildung 6.4.: Verteilte Anwendung, bei der der Austausch einer Komponente in der Architektur nur im Binden zu einem anderen Anbieter resultiert, der nicht deployt werden muss

Je nach Verfügbarkeit und Eigenschaften der Dienste (Kosten, Performance) kann die Anwendung unter den verschiedenen Diensten den für sie günstigsten auswählen.

Wenn in der Anwendungsarchitektur der Austausch oder das Hinzufügen einer Komponente erfolgen soll, dann ist in dem Fall keine Komponentendatenbank vonnöten, aus der die Komponenten geholt und deployt werden. Der Grund liegt darin, dass die Komponenten durch Dritte zur Verfügung gestellt werden, sodass über die Komponenten keine Verwaltungsherrschaft besteht. Die Komponenten können dabei permanent verfügbar sein (Web Services) oder hinsichtlich der Präsenz stark fluktuieren (z. B. in Ad-Hoc-Netzwerken). Somit ist ein Instantiieren der Komponenten nicht vonnöten bzw. möglich. Für das Hinzufügen oder den Austausch der alten Komponente durch die neue erfolgt nur ein Binden zu der neuen Komponente, ggf. auch eine vorherige Initialisierung des zu verwendenden Dienstes mit Parametern.

In gewisser Hinsicht stellen das LAN, WAN[10] oder das Internet einen Container dar, in dem durch Dritte Architektur-Komponenten »deployt« werden.

Fazit. Nicht immer muss die Infrastruktur eine Komponentendatenbank anbieten. Je nach Art des verteilten Systems müssen Komponenten erst aus einer Datenbank geholt und deployt werden, oder der Austausch der Komponente erfolgt in der Nutzung eines bereits verfügbaren Dienstes, z. B. in Service-orientierten Architekturen mittels Web Services [Kreger 2001] oder lokal mittels Jini-Service [Jini].

[10] Wide Area Network

6.2.4. Komponentendatenbank für die Adaptionsarchitektur/A.-Infrastruktur

In den Abschnitten 4.1.4 (Adaption Adaptionsarchitektur), 5.2.5 (Rekonfigurationsschleife bei Ansatz von Kokar *et al.* [1999]) und 6.1.4 (Aktivität Änderungen planen und formulieren) wurde die Adaption der Adaptionsarchitektur (Infrastruktur) propagiert. Für eine Rekonfiguration der Infrastruktur ist ebenso wie für eine Adaption der eigentlichen Anwendung eine Datenbank mit Komponenten der Infrastruktur notwendig.

Die Komponentendatenbank für die Adaptionsarchitektur enthält alle notwendigen Komponenten der Infrastruktur, die für den Beobachtungs-, Interpretations- und Auflösungsapparat notwendig und rekonfigurierbar sein sollen.

Einschränkung

Wie bei der Komponentendatenbank für die Anwendungsarchitektur kann hier eine Einschränkung vorgenommen werden, d. h., die Infrastruktur-Datenbank ist nicht zwingend notwendig. Der triviale Fall ergibt sich, wenn keine Rekonfiguration an der Infrastruktur vorgenommen werden muss oder diese durch Parametrisierung der bestehenden Infrastruktur möglich ist. Des Weiteren besteht der Fall, dass keine Datenbank notwendig ist, in Analogie zu Anwendungsarchitektur-Datenbank, wenn auf vorhandene Dienste zugegriffen werden kann. Beispielsweise kann ein unbemanntes Flugzeug den örtlichen Wetterdienst in Anspruch nehmen (keine Infrastruktur-Datenbank notwendig) oder eigene Ressourcen dafür benutzen (Aufklärungsflugzeug hinschicken).

6.2.5. »Knoten-Datenbank« für die Systemarchitektur — alternative Verarbeitungsknoten

Wenn Komponenten für die Skalierung und bessere Lastverteilung der Anwendung repliziert oder auf andere Verarbeitungsknoten migriert werden müssen (oder dies selber vornehmen), dann werden alternative Verarbeitungsknoten benötigt. Diese bieten eine Laufzeitumgebung für die Ausführung von Komponenten an.

In der kleineren und tradionellen Form besitzt eine Organisation eine Farm mit Servern oder ein Cluster im Rechenzentrum, die die Ressourcen für die Anwendung anbieten. Für einen größeren Rahmen sind die sog. Grid Services [Foster *et al.* 2002] ein Schlagwort für die Bereitstellung von Ressourcen für verteilte Systeme. In mobilen Umgebungen mit beschränkten Eigenschaften der Ausführungsumgebung können die Verarbeitungsknoten die mobilen Endgeräte oder Knoten im Festnetz sein. (Durch die Migration von Komponenten, in dem Fall auch

(mobile) Agenten genannt, ist ein abgekoppeltes Arbeiten des Endgerätes sowie die Schonung der Ressourcen auf dem Endgerät möglich.)

6.2.6. Spezifikationsdatenbank für Komponenten

Zusätzlich zu den Datenbanken für Komponenten der Anwendungsarchitektur (6.2.3) und der Adaptionsarchitektur/A.-Infrastruktur (6.2.4) gibt es eine Datenbank, die die Spezifikationen der vorgenannten Komponentenarten enthält.[11]

Die Spezifikationsdatenbank enhält ein Verzeichnis aller verfügbaren Komponenten (und Konnektoren, siehe Fußnote 4 auf Seite 187). Die Beschreibung für eine Komponente kann in der Maximalversion umfassen

1. den global eindeutigen Namen für eine klare Identifikation [van Ommering *et al.* 2000],

2. Angaben zu funktionalen Aspekten, d. h. die Beschreibung zur angebotenen (provides) und zur benötigten (requires) Funktionalität der Schnittstellen einer Komponente. Die Schnittstellenbeschreibung ist mit allen Angaben zu den Ebenen der Interoperabilität (vgl. Seite 102 und Vallecillo *et al.* [2000]) versehen, wobei hier im einfachsten Fall nur für die Signaturebene (z. B. IDL) zu berücksichtigen ist. Außerdem kann die Spezifikationsdatenbank

3. Angaben zu nicht-funktionalen Aspekten der Komponenten und

4. initiale Parameter enthalten.

Die Angaben zu den nicht-funktionalen Eigenschaften einer Komponente können vielfältig sein und hängen von dem Anwendungsgebiet der Software ab. Beispielsweise können diese Angaben die Kosten umfassen. Die Kosten können hier in der Tat als monetäre Größe aufgefasst werden, z. B. wenn die alternativ gewählte Komponente als Web Service von einem anderen Anbieter zur Verfügung steht und für deren Nutzung Geld zu zahlen ist. Ebenso können die Kosten den Bedarf an Ressourcen bedeuten (quantitive Eigenschaften), die die Ausführung der Komponente auf dem System benötigt — beispielsweise im Bereich von multimedialen Anwendungen, bei denen Kodierungsalgorithmen mit unterschiedlichen Anforderungen an CPU-Zeit, Bandbreite und schlussendlich erreichter Auflösung für das kodierte Video zur Verfügung stehen. Neben dem Bedarf an Ressourcen sind auch andere quantitative Eigenschaften denkbar, die die Komponente auszeichnen.

[11] Für die im vorhergenden Abschnitt 6.2.5 beschriebene »Knoten-Datenbank« ist die Spezifikationsdatenbank durch den Resource-Discovery-Dienst gegeben (vgl. Abschnitt 6.2.1).

Auswahlprozess

Bei einer anstehenden Rekonfiguration der Software-Architektur wird die Datenbank nach passenden Komponenten durchgesucht und die Rekonfiguration der Architektur vorgenommen. Die Auswahl der Komponenten erfolgt anhand des Namens oder spezieller Eigenschaften, die den Zweck der Komponente vorgeben. Als zweites muss die Schnittstelle der Komponente kompatibel sein. Kompatibilität muss nicht durch 1:1-Übereinstimmung der Schnittstellen von alter und neuer Komponente erfolgen, sondern die neue Komponente kann auch mehr Funktionen anbieten — wichtig ist, dass sie alle Funktionen erbringt, die von den Komponenten benötigt werden, die zuvor mit der alten Komponente verbunden waren, d. h. von dieser abhängen (vgl. [van Ommering *et al.* 2000]). Andernfalls kann ein Adapter dynamisch erzeugt oder beschafft werden, der die Ebenen der Interoperabilität aneinander anpasst [Canal 2004].

Sind für eine gewünschte Funktionalität mehrere Alternativen verfügbar, kann eine weitere Auswahl anhand der nicht-funktionalen Eigenschaften der Komponente erfolgen.

Einschränkung

Verschiedene Konstellationen in der Software-Architektur können dafür ausschlaggebend sein, dass gar keine Spezifikationsdatenbank benötigt wird:

1. Beispiel Cactus (Abschnitt 5.5.1). Bedingt durch die konzipierte Software-Architektur, dass eine AC mehrere AAMs kapselt, für die ein Austausch möglich ist, wird aus folgenden Gründen keine Spezifikationsdatenbank benötigt:

   a) Ein Name für die Beschreibung der Komponenten ist nicht notwendig, da die Komponente über ihren Index in der lokalen Komponentendatenbank identifiziert wird.

   b) Die funktionalen Eigenschaften und die Schnittstellenbeschreibung sind für jedes AAM die gleichen, da diese durch die sie kapselnde AC bestimmt werden.

   c) Die Spezifikation nicht-funktionaler Eigenschaften erfolgt nicht extern, sondern wird aufgrund der Abhängkeit vom Zustand des Systemes durch Logik *in der* Komponente ermittelt.

2. In selbstorganisierenden Architekturen ohne explizites Management, wie z. B. in Jini [Jini] und JXTA[12] [JXTA], gibt es keine

[12] Juxtapose (von »to juxtapose«, deutsch »nebeneinander stellen«)

Spezifikationsdatenbank. Stattdessen ist die Spezifikationsdatenbank als Dienstanbieter nicht explizit im System gegeben, sondern »echt verteilt«. Die Knoten bauen einen lokalen Cache mit Dienstbeschreibungen auf.

Für verschiedene Architekturformen des sog. Resource Discovery siehe [Vanthournout *et al.* 2004].

6.2.7. Planer

Der Planer erfüllt eine wichige Funktion, da er die Entscheidung trifft, welche Adaption δ_S an dem System zu erfolgen hat. In der Kontroll-Terminologie hat der Planer den Namen *Controller* (siehe Abschnitt 5.2), bei Open ORB wird er *Strategy Selector* (Abschnitt 5.3.2) genannt. Der Planer entscheidet anhand von Regeln, auch Kontrollgesetze genannt, bei welcher Konstellation des Systems und der Umgebung welche Adaption vorgenommen werden soll. Dabei sind einige Schwierigkeiten zu berücksichtigen. Der Planer muss (in Anlehnung an [Garlan und Schmerl 2002])

1. aus mehreren möglichen Aktionen die beste auswählen,
2. in Konflikt stehende Aktionen in Einklang bringen und
3. ein System verbessern, wenn gar kein spezieller Fehler oder eine schlechte Performance gegeben ist.

Bei einfachen Systemen, bei denen nicht die vorgenannten Komplikationen vorliegen, kann der Planer auf Basis eines offenen Regelkreises agieren. Wie im Fazit zu der Kontroll-Theorie (5.2.6) ersichtlich wurde, sind geschlossene Regelkreise notwendig, sobald der Regelsatz zu komplex und umfangreich oder die Umgebung nicht komplett spezifiziert werden kann. In dem Fall werden (geschlossene) Regelkreise mit Rückkoppelung empfohlen, die dem Planer Information über die Systemperformance geben.

Anmerkung. Der in Abschnitt 5.1.2 vorgestellte CHAM-Ansatz eignet sich nicht nur für die Selbstorganisation der Architektur, sondern auch dafür, Eigenschaften des Systems vor und nach der Rekonfiguration zu prüfen. Die in CHAM möglichen Reaktionsregeln können als Basis dafür genutzt werden, Adaptionspläne auszuwählen oder zu konstruieren.

Intern kann der Planer aus den verschiedenen Subsysteme der Adaptivregelung, rekonfigurierbarer Regelung und anderer Modelle von Regelungssystemen bestehen.

6.2.8. Analysierer

Entsprechend Aktivität 6.1.5 gibt es in der Infrastruktur den Analysierer, der die formulierten Änderungen dahingehend analysiert, ob sie korrekt sind und bei Anwendung am System die Integrität bewahren.

Anmerkung. Wenn keine Adaption auf Basis eines Architekturmodells vorgenommen wird (Beispiel Cactus in Abschnitt 5.5.1), dann ist keine Analyse möglich bzw. der Ansatz für eine Adaption in dem System ist derart einfach gehalten, dass keine Analyse mithilfe eines Modells notwendig ist. Oder die Adaptionsoperationen sind nicht risikobehaftet (z. B. Inhaltsadaption).

6.2.9. Datenbank für die Adaptionsregeln

Insbesondere für offen-adaptive Systeme ist es notwendig, die Adaptionsregeln nicht fest in der Anwendung zu kodieren, sondern diese in einer separaten Beschreibungssprache formulieren und in einem gesonderten Speicher (im einfachsten Fall eine Datei) zu sichern.

Auch für eine geschlossen-adaptive Software sollten die Regeln nicht fest in der Anwendung kodiert sein, sondern explizit angegeben werden können. Dies erlaubt zudem die formale Analyse der Regeln.

Anmerkung. In den Abschnitten 5.1.1 wurden mit den AMLs und in Abschnitt 5.1.2 mit den Reaktionsregeln des CHAM-Ansatzes einige Spezifikationssprachen für dynamische Architekturen beschrieben. Diese beschreiben eine mögliche oder vorzunehmende Operation. Die für eine auszuführende Operation notwendigen Bedingungen können beispielsweise auf Basis des ECA-Paradigmas aus *Active Databases* [Act-Net Consortium 1996] formuliert werden. In Open ORB heißen die Regeln *Strategy Activator* (vgl. Abschnitt 5.3.2). Bei diesem Ansatz werden Zeitautomaten [Lynch und Vaandrager 1992] für die Formulierung (und formale Analyse) der Regeln verwendet. In der Terminologie der Regelungssysteme werden die Regeln als Kontrollgesetze bezeichnet (siehe Abschnitt 5.2.5).

6.2.10. Spezifikationsdatenbank für die Adaptionsziele

Die Spezifikationsdatenbank für die Adaptionsziele enthält mehrere Profile, in denen Informationen zu den erfolgenden Adaptionsresultaten stehen. Erfolgt beispielsweise eine Adaption an die heterogenen Eigenschaften der Ausführungsumgebung, dann muss die Datenbank Profile zu den Nutzern, Endgeräten, Netzwerk etc. besitzen.

Die Datenbank mit den Profilen muss nicht statisch gehalten sein, sondern die einzelnen Profile können im Laufe des Betriebs durch

Profilierung entstehen (z. B. durch den auf Seite 88 genannten Farbsehtest oder beim initialen Trainieren einer Spracherkennungssoftware durch den Nutzer) oder nur transient/temporär für den Zeitpunkt der Dienstanfrage bestehen (z. B. gewünschte nicht-funktionale Eigenschaften des Clients an die Dienstausführung in COMQUAD [Schill *et al.* 2000], [Aigner *et al.* 2003]).

Zusammenhang Adaptionsregeln und Adaptionsziele. Zwischen der im vorhergehenden Abschnitt beschriebenen Datenbank für die Adaptionsregeln und der in diesem Abschnitt empfohlenen Datenbank für die Adaptionsziele besteht ein großer Zusammenhang: In den *Adaptionsregeln* ist die Logik formuliert, mit der der Planer eine Adaptionsentscheidung trifft. Die *Adaptionsziele* sind die Parameter für die Regeln bzw. das notwendige Resultat, das bei der Anwendung der Regel entstehen muss.

Die Adaptionsziele können auf verschiedenen Abstraktionsebenen formuliert werden, woraus sich verschiedene Schwierigkeiten in der Umsetzung ergeben. Für die Adaption eines Bildes gibt [Franke 2001, S. 47] folgende Beispiele für Abstraktionsebenen an: JPEG-Qualitätsfaktor 70 % (niedrig), Übertragungszeit nicht länger als 10 Sekunden (mittel), Kosten nicht größer 0,10 Cent (hoch). In diesem Beispiel ist die Umsetzung der auf der hohen Abstraktionsebene formulierten Forderung einfach: Es werden die Kosten pro Datenmenge oder Zeit sowie die verfügbare Bandbreite herangezogen, um das Bild so lange anzupassen, bis es den maximalen Kostenrahmen unterschreitet.

Ein zweites Beispiel, das dem Zeitungsartikel in [Rademacher 2004] entnommen ist, ist im Rahmen der Interaktion des Nutzers mit einem Auto die Formulierung: »Senke vorübergehend den Benzinverbrauch, weil der Tank fast leer ist.« Diese globale und unscharfe Direktive muss daraufhin in konkrete Veränderungen des System umgesetzt werden. Professor Christian Müller-Schloer charakterisiert in dem gleichen Zeitungsartikel die notwendigen Aktivitäten als »... große Herausforderung für die Informatik — es ist die Lücke zwischen einer deklarativen Zielvorgabe und einer prozeduralen oder strukturellen Ausführung zu überbrücken.« Wie das Zitat bereits ausdrückt, ruft dieses Beispiel mehr Komplikationen hervor.

6.2.11. Konfigurationsmanager

Der Konfigurationsmanager (KM) übernimmt das Management des Rekonfigurationsprozesses in der Architektur. Er

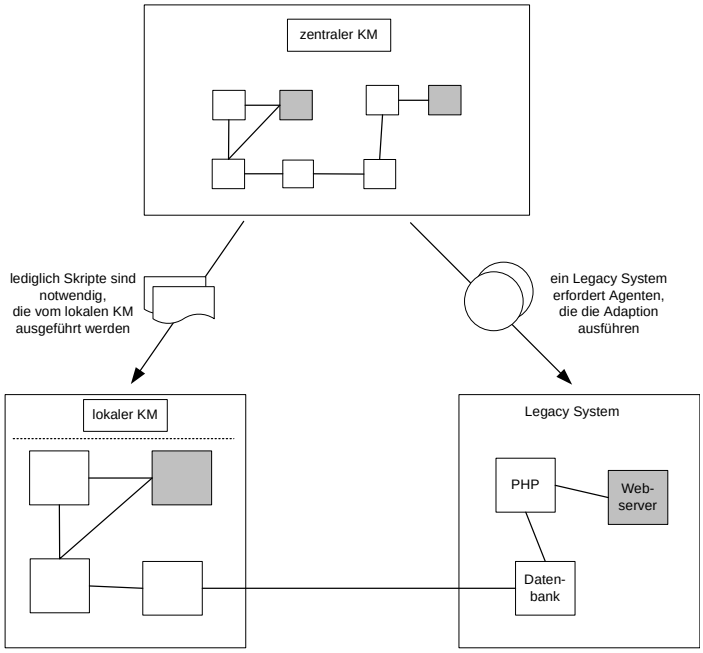

Abbildung 6.5.: Verteiltes System mit einem zentralen und einem lokalen Konfigurationsmanager (KM) und Agenten für die Unterstützung von Legacy Systemen

- enthält Modell[13] der Architektur (im Fall einer Meta-Level-Architektur ist er deshalb ein Metaobjekt,
- ist zentrale Managementinstanz für das verteilte System, die eine globale Sicht auf das System besitzt, wobei zentral getroffene Entscheidungen für eine Adaptionsoperation durch die lokalen Konfigurationsmanager auf den verschiedenen Sites ausgeführt werden (siehe Abbildung 6.5),
- kann auch nur lokal auf den Sites enthalten sein. Bei Autonomic Computing wäre das der Autonomic Manager[14] und bei einem selbstorganierenden System besitzt jeder Knoten einen Konfigurationsmanager, um seine eigene Struktur ändern zu können [Dowling 2003]. Dieser Sachverhalt ist in Abbildung 6.6 auf der nächsten Seite dargestellt.

[13] Im Idealfall ist das Modell in einer gesonderten Datenbankkomponente gespeichert, die über ein festes API Zugriff auf das Modell bietet. Um die Software-Architektur nicht zu überfrachten, wird das Architekturmodell in den Ausführungen nicht gesondert behandelt.

[14] Die Beschreibungen des Architekturansatzes von Autonomic Computing [Keffart und Chess 2003] bieten keine Klarheit, ob bei einem Austausch einer Komponente das zu verwaltende Element innerhalb des autonomen Elements ausgetauscht wird oder der Lebenszyklus des autonomen Elementes beendet und das gesamte autonome Element ausgetauscht wird.

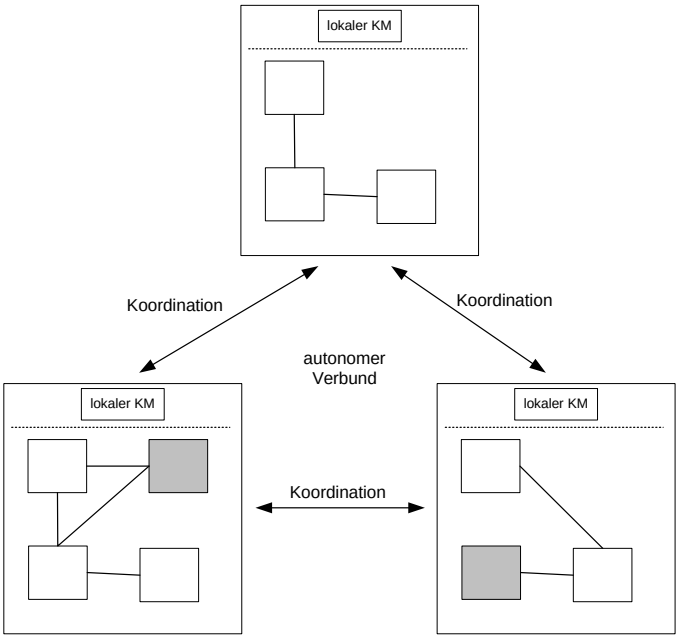

Abbildung 6.6.: Verteiltes System ohne einen zentralen Konfigurationsmanager (KM), jedoch mit einem lokalen KM auf jedem Host. Die Rekonfiguration in dem autonomen System erfolgt selbstorganisierend.

Überschneidung mit Koordinierer. Der Konfigurationsmanager führt ebenso wie die noch im Folgenden beschriebenen Koordinierer die Spezifikation der Änderungsoperationen an der Architektur aus. Die Änderungsbeschreibung kann aus der losen Aneinanderreihung der Operationen (welche Komponenten und Konnektoren hinzuzufügen oder zu entfernen sind) bestehen.

Sobald Legacy Systeme (Altsysteme, vorhandene Systeme) unterstützt werden müssen, kann die Rekonfiguration nicht durch einen lokalen Konfigurationsmanager, wie er von entsprechenden Komponententechnologien angeboten wird,[15] erfolgen. In dem Fall sind *Agenten* erforderlich, die die Rekonfiguration vornehmen (in Abbildung 6.5 auf der vorherigen Seite das rechte System). Hierfür werden Worklets und Effektoren als Agenten in der Infrastruktur eingeführt.

6.2.12. Koordinierer einer Adaption

Koordinierer übernehmen die im Abschnitt 6.1.9 geforderte Koordinierung der Adaption in einem verteilten System. Aufgabe der Koordinierer ist es, die lose Folge der formulierten Änderungsoperationen so zu

[15] Gemeint ist die in Abschnitt 5.3.2 beschriebene reflective Middleware, die Laufzeitsysteme der ADLs/AMLs (unterliegendes Komponentenframework) und andere Komponentenframeworks, die die (ggf. dynamische) Rekonfiguration unterstützen.

optimieren, dass die Unterbrechungszeit des Systems minimal gehalten wird. Die Zeit kann minimiert werden, indem Änderungen in dem verteilten System parallelisiert werden. Zu den Optimierungen gehört ebenso die Aufgabe, das System weitestgehend am Laufen zu halten. Das bedeutet, dass die Bestandteile des verteilten Systems, die von der Änderung nicht direkt betroffen sind, weiter im Betrieb bleiben.

In besonderen Anwendungsfällen (z. B. in Kommunikationslösungen) kann die Forderung bestehen, dass es bei dem Austausch einer Komponente für den Dienst, den alte und neue Komponente erbringen, keine Unterbrechungszeit gibt. Das heißt, das von der Adaption nicht betroffene Teilsystem *und* das von der Adaption betroffene Teilsystem bleiben (müssen) weiterhin operabel (ohne kurzzeitige Unterbrechung durch z. B. »Einfrieren« der angrenzenden Konnektoren).

Globale und lokale Koordinierer — Ansätze

Für die Koordinierer sind je nach Art des Systems (Grad der Verteilung, Komponentenarchitektur), das adaptiert werden soll, verschiedene Ansätze für die Koordinierer denkbar. Im Folgenden wird eine kleine Auswahl der Ansätze in der Literatur gegeben.

Workflakes und Worklets. Im Rahmen des DASADA-Projektes [Salisan 2004] haben Valetto und Kaiser sog. Workflakes entwickelt [Valetto und Kaiser 2002b]. Workflakes sind ein dezentralisiertes Workflow-System für die Instantiierung und Koordinierung aller Arten von Adaptionen in einem laufenden System. Eng mit den Workflakes zusammen arbeiten die sog. Worklets, die die *lokal* koordinierte Adaption an einem Knoten des Systems vornehmen. Dabei übernehmen die Workflakes die *globale* Koordinierung der Worklets.

Worklets sind Agenten, die mobilen, in sich abgeschlossenen (self-contained) Code mit sich führen, der auf dem Zielsystemen für die Rekonfiguration des Systems ausgeführt werden kann [Kaiser *et al.* 1999], [Valetto *et al.* 2001]. Sie entsprechen dem in Abschnitt 6.1.7 geschaffenen Begriff »Adaptionspaket«, das alle notwendigen Bestandteile für eine Adaption beinhaltet, wobei die Worklets nach dem Pull-Prinzip arbeiten (notwendige Komponenten etc. werden von einem zentralen Repository heruntergeladen). Ein Worklet führt alle notwendigen Operationen für eine Adaption auf dem Zielsystem durch, angefangen von der Installation über die Konfiguration aller notwendigen Parameter, Entfernen der Komponente etc.

In Abbildung 6.7 auf der nächsten Seite ist der Ansatz der Workflakes und Worklets dargestellt.

Workflakes senden die Worklets zu den einzelnen Sites aus und koordinieren die Worklets in dem verteilten System untereinander. Worklets nehmen die einzelnen Adaptionen an dem jeweiligen Host im System vor.

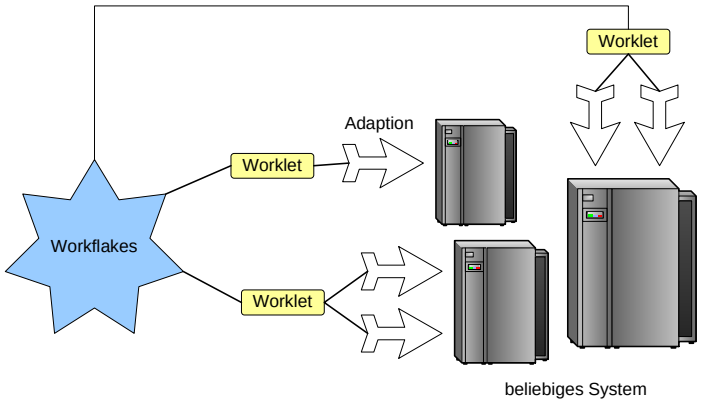

Abbildung 6.7.: Verteilte Koordinierung einer Adaption mit Workflakes und Worklets

Transaktionaler Ansatz. Während der Durchführung vor allem von komplexen Änderungen kann die Anwendung unter Umständen mehrere ungültige Zustände durchlaufen, bevor der Endzustand erreicht wird. In dem Fall wäre es nicht anwendbar, Änderungspfade durch Randbedingungen einzuschränken, da dadurch einige gültige Laufzeitänderungen unterbindet werden würden. Stattdessen sollte ein transaktionaler Mechanismus angeboten werden [Oreizy *et al.* 1998].

In Abschnitt 5.1.1 wurde bereits ein transaktionaler Ansatz in der erweiterten Form von Wermelinger vorgestellt. Dieser kann wie folgt einfach in vier Schritten formuliert werden (in Anlehnung an Dashofy *et al.* [2002], die den Ansatz in einer Light-Version verwenden):

- *Stabilen Zustand herbeiführen.* Komponenten und Konnektoren, die entfernt werden sollen, führen Aufräum-Code aus und senden ihre Zustandsdaten an andere Komponenten, damit die Daten, für den Fall, dass die Komponente ersetzt werden soll, später verfügbar sind.[16]

  (Dieser Schritt ist bei fast allen Systemen notwendig, sofern die Systeme einen Zustandsverlust nicht tolerieren. Bei Systemen ohne Zustand, z. B. bei den Komponenten im Cactus-Ansatz (siehe nachfolgende Ausführungen und die Vorstellung in Abschnitt 5.5.1), die nur Nachrichten oder einen Nachrichtenstrom verarbeiten, muss kein stabiler Zustand herbeigeführt werden, sondern nur ein unterbrechungsfreier und transparenter Wechsel der Komponenten auf Empfänger- und Senderseite.)

- *Partielle Stilllegung.* Komponenten und Konnektoren, die an das betroffene Gebiet in der Architektur angrenzen, werden vorüber-

[16] Einen Einblick in die Thematik und den State-of-the-Art für die Sicherung des Zustandes geben [Kasten und McKinley 2004], [Chen *et al.* 2001] und [Oreizy *et al.* 1998].

gehend stillgelegt. Damit wird das Senden von Nachrichten an die Komponenten oder Konnektoren verhindert, während diese entfernt oder hinzugefügt werden.

- *Adaption.* Komponenten und Konnektoren werden entsprechend der Änderungsbeschreibung entfernt, neu hinzugefügt oder ausgetauscht.

- *Wiederinbetriebnahme.* Sowohl neue Komponenten und Konnektoren als auch die, die zuvor stillgelegt worden sind, werden in Betrieb gesetzt.

Die Rekonfiguration in dem Ansatz von Dashofy *et al.* wird von dem AEM vorgenommen (entspricht dem Konfigurationsmanager in der vorliegenden Arbeit). In dieser Form ist der Ansatz allerdings nur für die Rekonfiguration auf *einem* Host anwendbar, d. h. eine echte Verteilung auf mehrere Hosts gibt es nicht. Der Ansatz könnte um einen globalen AEM erweitert werden, wie ihn Oreizy *et al.* [1999] auch in ihrem generellen Ansatz für selbst-adaptive Software beschreiben.

Damit besteht eine große Ähnlichkeit zu dem zuvor beschriebenen Workflakes/Worklets-Ansatz, wobei der global agierende AEM den Workflakes entspricht, der lokale den Worklets.

Cactus. Die Ausführungen zu Cactus in Abschnitt 5.5.1 haben gezeigt, dass bei einer statischen Struktur des Komponentennetzes (d. h. keine Komponenten sind zu entfernen oder hinzuzufügen, sondern nur auszutauschen) ein Austausch auch ohne partielle Stillegung, wie er im transaktionalen Ansatz vorgenommen wird, möglich ist. Bei dem Ansatz von Cactus können die Nachrichten, die von auszutauschenden Komponenten verarbeitet werden, während des Austausches weiterhin gesendet und empfangen werden. Der Austausch erfolgt dabei transparent für die Anwendung, indem das Umschalten von eingehendem und ausgehendem Nachrichtenstrom durch ein dreiphasiges Koordinationsprotokoll geschieht. Bei dem Protokoll können sowohl alte als auch neue Komponente gleichzeitig aktiv sein und den Nachrichtenstrom verarbeiten. Den übergeordneten Koordinatoren des Austauschs (in dem Fall die ACs/CAMs in den ACs auf Empfänger- und Senderseite, die die Nachrichten an die richtigen AAMs routen) kommt dabei die Aufgabe des korrekten Routings zu, damit der durch die neue Komponente verarbeitete Nachrichtenstrom auf der Empfängerseite auch von einer neuen Komponente empfangen wird.

Verglichen mit den zuvor beschriebenen Ansätzen können bei Cactus *nur lokale Koordinierer* identifiziert werden, wobei es für die lokalen Koordinierer zwei verschiedene Typen gibt:

1. *Adaptation Controller.* Der Adaptation Controller ist in Cactus auf jedem Host fest installiert und übernimmt die Koordination der

Adaption zwischen den einzelnen ACs und zwischen den Schichten des Systems auf einem Host.

2. AC/CAM. Ein zweiter lokaler Koordinierer ist mit den CAMs bzw. ACs gegeben. Diese koordinieren den Austausch von AAMs innerhalb einer DAC über mehrere Hosts hinweg.

Einen übergeordneten Koordinierer wie im Fall der Workflakes gibt es bei Cactus nicht. Eine Parallelität kann zu den Worklets gezogen werden, die bei Cactus den Adaptation Controllern entsprechen. Die Besonderheit von pro Komponente zugeordneten Koordinierern (AC/CAM) ist nur bei Cactus zu finden.

Weitere Ansätze. Weitere Ansätze für eine sichere und koordinierte Rekonfiguration der Architektur bieten [Goudarzi 1999], [Almeida 2001] und [Bridges 2002].

6.2.13. Worklets

Entsprechend dem Ansatz von Valetto und Kaiser, der bereits im vorhergehenden Abschnitt beschrieben wurde, werden für die Unterstützung von Adaptionen von beliebigen Systemen Worklets als Infrastruktur-Komponenten übernommen.

Worklets sind Agenten, die mobilen, in sich abgeschlossenen (self-contained) Code mit sich führen, der auf dem Zielsystemen für die Rekonfiguration ausgeführt werden kann [Kaiser *et al.* 1999], [Valetto *et al.* 2001]. Sie entsprechen dem in Abschnitt 6.1.7 geschaffenen Begriff »Adaptionspaket«, das alle notwendigen Bestandteile für eine Adaption beinhaltet oder diese herunterladen kann.

Ein Worklet führt alle notwendigen Operationen für eine Adaption auf dem Zielsystem durch, angefangen von der Installation über die Konfiguration aller notwendigen Parameter, Entfernen der Komponente etc.

6.2.14. Effektoren

Die in der vorliegenden Arbeit vorgestellten Ansätze Rainbow-Framework (5.5.2) und Autonomic Computing (5.5.3) sowie die von im DASADA-Projekt [Salisan 2004] entwickelte sog. »KX Infrastructure« [Valetto und Kaiser 2002a] für die Adaption eines Systems besitzen Effektoren. Effektoren nehmen die Adaption in dem System vor. Sie haben die Aufgabe, das System zu rekonfigurieren, Parameter einer Komponente oder die gesamte Komponente anzupassen oder die gesamte Architektur des Systems zu adaptieren.

Unterschied Worklet und Effektor. Effektoren und Worklets nehmen beide die Rekonfiguration und Parametrisierung auf einem Knoten vor. Der Unterschied ist, dass Effektoren die einzelnen systemspezifischen Operationen sind, währenddessen in einem Worklet der Adaptionsprozess als kleiner Workflow spezifiziert ist (klein, weil die Koordinierer die Adaption im Großen koordinieren). Worklets führen die einzelnen Effektoren (Adaptionsoperationen) auf einem Host hintereinander aus. Deshalb sind Effektoren mit den *Strategy Activatorn* von Open ORB vergleichbar — mit dem Unterschied, dass in dem Ansatz der vorliegenden Arbeit plattformunabhängige Adaptionsoperationen in der Datenbank für Adaptionsregeln (6.2.9) spezifiziert sind, die vom Übersetzer (6.2.15) in plattformspezifische Operationen (Effektoren) transformiert werden. Die *Strategy Activator* sind plattformunabhängige und plattformspezifische Operatoren zugleich, da sie nur für die Plattform Open ORB Geltung besitzen (eine Trennung in plattformunabhängige und plattformspezifische Operatoren gibt es nicht).

6.2.15. Übersetzer für die Spezifikationssprachen und Mapping-Datenbank

Der Übersetzer nimmt mithilfe einer Mapping-Datenbank die Übersetzung der im plattformunabhängigen Modell formulierten Kommandos in plattformspezifische Operationen vor. Resultat sind die systemspezifischen Effektoren und Worklets, die die Adaption auf dem beliebigen System ausführen.

Plattformabhängigkeit. Effektoren und Worklets sind Agenten, die Operationen auf den Hosts ausführen. Eine wichtige Voraussetzung für mobile Agenten ist die Verfügbarkeit einer Ausführungsumgebung auf den Knoten [Fuggetta *et al.* 1998], z. B. in Form einer VM oder eines Skript-Interpreters. Bei Vorhandensein einer plattformunabhängigen Ausführungsumgebung ist es dann lediglich notwendig, die in den Adaptionsregeln (Spezifikationssprachen) enthaltene Logik in Kommandos umzusetzen, die in der Ausführungsumgebung ausgeführt werden können. Falls die Ausführungsumgebung heterogen ist und keine einheitliche Ausführungsumgebung für die Skripte bietet, dann muss zusätzlich eine Übersetzung der Änderungsskripte für die verschiedenen Zielplattformen erfolgen.

Bei einer Erörterung zur Plattformabhängigkeit müssen Worklets und Effektoren getrennt betrachtet werden. Denn im Gegensatz zu den Effektoren sind Worklets nur von dem Betriebssystem auf dem Host abhängig. Effektoren dahingegen sind für jede Anwendung (beliebiges IT-System, Komponentenframework) spezifisch.

Genauso wie die Monitore sind Effektoren besonders vom System abhängig. Sie müssen für die gewählte Plattformtechnologie spezialisiert werden, d. h. meist von Hand geschrieben werden. Die restlichen Be-

standteile der Infrastruktur sind weniger vom unterliegenden System abhängig.

Wie bereits bei der Beschreibung zu den Monitoren formuliert, sollte es Ziel sein, wiederverwendbare Effektoren zu entwickeln

6.3. Evaluation der Empfehlungen

Die zwei vorhergehenden Abschnitte haben sowohl dynamische als auch statische Aspekte für eine adaptive Software-Architektur beschrieben. Dabei wurden bereits an ausgewählten Stellen Anmerkungen formuliert, wenn für eine bestimmte Aktivität oder für eine Komponente in der Infrastruktur Ausnahmen festgestellt werden können, bei denen die Voraussetzung für die Existenz der Aktivität/Komponente nicht gegeben ist. Aus Gründen eines klaren Leseflusses wurden derartige Ausführungen auf ein Minimum beschränkt.

In dem vorliegenden Abschnitt werden einige markante Anwendungsfälle entworfen. Für jeden Anwendungsfall wird daraufhin auf Basis des in den bisherigen Ausführungen beschriebenen und empfohlenen »Baukastens« für die Software-Architektur demonstriert, wie eine spezifische Anwendung, die Adaptionen an der Architektur vornimmt, aussieht — sowohl hinsichtlich dynamischer als auch statischer Aspekte.

Im Folgenden wird jeder Anwendungsfall gesondert zuerst beschrieben und im gleichen Abschnitt der Entwurf einer möglichen Software-Architektur vorgenommen. Um die Evaluation der Architekturansätze einfach zu gestalten, wird für alle Anwendungsfälle nur der Adaptionsmechanismus *Komponente austauschen* betrachtet. Dies ermöglich einen besseren Vergleich. Falls die Anwendungsfälle und der damit verbundene Architekturansatz darüber hinaus weitere Adaptionsmechanismen zulässt, werden diese genannt und kurz charakterisiert.

Zum Abschluss des Kapitels werden die Software-Architekturen der konzipierten Anwendungsfälle untereinander verglichen und ein Fazit für den zuvor empfohlenen Software-Architektur-Ansatz gezogen.

6.3.1. Anwendungsfall 1 — beliebiges, verteiltes System

Anwendungsfall 1 besteht aus einem beliebigen, verteilten und heterogenen System. Auf den Hosts sind verschiedene Komponenten einer IT-Landschaft installiert, z. B. Web-, Datenbank-, Applikationsserver und andere.

Adaptionsoperationen

Für das System sind verschiedene Adaptionsoperationen denkbar.

1. Administrator stellt in der Komponentendatenbank mehrere Updates von vorhandenen Komponenten zur Verfügung. Das System

passt sich an, indem es die Komponenten selbst installiert (austauscht) und konfiguriert. Zusätzlich kann noch ein Regressionstest zur Überprüfung der korrekten Funktion der neuen Komponenten durchgeführt werden (in Anlehnung an [Keffart und Chess 2003]).

2. Show-Case des Videokonferenzsystems aus dem Rainbow-Framework, der in Abschnitt 5.5.2 beschrieben ist.

3. Beliebig andere Adaptionsoperationen aus Abschnitt 4.1 mit beliebiger Motivation für eine Adaption (d. h. für Komponentenaustausch nicht nur vom Administrator bereitgestellte Komponente, sondern Erfordernis nach einer anderen Komponente aufgrund einer anderen Umgebung oder für die Verbesserung der Performance).

Software-Architektur

Die statische Software-Architektur für das verteilte System ist die gleiche, wie sie bereits in Abbildung 6.2 auf Seite 190 dargestellt ist — mit der Ausnahme, dass es keine DB für Infrastrukturkomponenten und keine alternative Verarbeitungsknoten gibt, da hier nur der Fokus auf dem Austausch und nicht der Migration/Replikation einer Komponente liegt. Die dynamischen Aspekte entsprechen in vollem Umfang denen, die in dem Aktivitätsdiagramm in Abbildung 6.1 auf Seite 182 spezifiziert worden sind. Damit entspricht die Software-Architektur für Anwendungsfall 1 der im Rahmen dieses Kapitels empfohlenen, d. h. alle Aktivitäten und alle Komponenten sind in der Software-Architektur vorhanden.

Trotz eingeschränktem Fokus unterstützt Anwendungsfall 1 außer dem Komponentenaustausch alle Adaptionsmechanismen.

6.3.2. Anwendungsfall 2 — Cactus

Der zweite Anwendungsfall lehnt sich an Cactus an, das bereits in Abschnitt 5.5.1 vorgestellt wurde.

Der in Cactus verwendete Show-Case besteht aus einen Gruppenkommunikationsdienst, der Nachrichten zuverlässig und in einer korrekten Reihenfolge (total ordering) versenden muss. Ein »Total-Ordering«-Protokoll stellt sicher, dass alle Mitglieder einer Prozessgruppe alle Nachrichten, die an die Gruppe gesendet wurden, in der gleichen (totalen) Reihenfolge empfangen. Für die Umsetzung dieser Bedingung gibt es verschiedene Protokolle/Algorithmen (Details in [Chen *et al.* 2001]), die in den Komponenten/AAMs implementiert sind.

Die Wahl einer Algorithmen-Komponente hat Auswirkungen sowohl auf den System-Overhead und die Ende-zu-Ende-Latenz der Anwendungsnachrichten. Beispielsweise ist bei dem sequentiellen Algorithmus der Overhead fix, da eine Nachricht pro Anwendungsnachricht

gesendet werden muss, wohingegen bei dem Token-Ansatz der Overhead abhängig davon ist, wie viele Nachrichten jedes Mal übertragen werden, wenn der Token den Host erreicht.

Adaptionsoperationen

Cactus erlaubt

1. den *Austausch Komponente* und
2. *Parameter einer Komponente anpassen.*

Wie eingangs bereits erwähnt, wird für eine Evaluation nur der Adaptionsmechanismus *Komponente austauschen* betrachtet.

Software-Architektur

Obwohl der gleiche Adaptionsmechanismus *Komponente austauschen* verwendet wird wie bei Anwendungsfall 1, entspricht die Software-Architektur für Anwendungsfall 2 nur minimal der von Anwendungsfall 1 bzw. der im Rahmen dieses Kapitel konzipierten.

Im Folgenden werden die einzelnen Punkte für dynamische und statische Aspekte kurz behandelt.

Beobachten (6.1.2) Nur System (Status des Netzwerks).

Beobachtungen auswerten (6.1.3) Die Interpretation einzelner Statuszähler benötigt keine weitere Auswertung.

Änderungen planen/formulieren (6.1.4) Nur planen.

Änderungen analysieren (6.1.5) Keine Analyse erforderlich, da keine Integrität verletzt werden kann (Ansatz sieht festes Komponentennetz vor, bei dem alle AAMs innerhalb einer AC (Knoten in dem Komponentennetz) die gleiche Funktionalität/Schnittstelle besitzen.

Änderungsskripte in systemspezifische Operationen übersetzen (6.1.6) Es gibt keine Änderungsskripte, die übersetzt werden müssten.

Adaptionspaket vorbereiten (6.1.7) Es ist kein Adaptionspaket erforderlich.

Adaptionspaket deployen (6.1.8) Nein.

Koordinierte Ausführung/Adaption (6.1.9) Eine koordinierte Adaption ist notwendig, wobei nicht der Zustand eines AAMs gesichert werden muss. Die Koordination läuft wie in Abschnitt 6.2.12 beschrieben ab.

Monitore (6.2.1) Ja. Ist jedoch in jedem AAM (»Komponente«) separat, d. h. erneut, implementiert. Könnte durch CAM übernommen werden; vgl. Fußnote 22 auf Seite 164.

Gauges (6.2.2) Die Interpretation einzelner Statuszähler benötigt keine weitere Auswertung.

DB Anwendungskomponenten (6.2.3) Ja, siehe Abbildung 5.17 auf Seite 163.

DB Komponentenspezifikationen (6.2.6) Nein, vgl. die Ausführungen in Abschnitt 6.2.6.

Planer (6.2.7) Ja. Wird auf Basis der Fitness()-Funktion im AAM (»Komponente«) und der Funktion BestFit() im CAM (Managementeinheit) verteilt vorgenommen. Geschlossener Regelkreis denkbar.

Analysierer (6.2.8) Nein.

DB Adaptionsregeln (6.2.9) Nein. Wird auf Basis der Fitness()-Funktion im AAM (»Komponente«) und der Funktion BestFit() im CAM (Managementeinheit) verteilt vorgenommen.

DB Adaptionsziele (6.2.10) Nein.

Konfigurationsmanager (6.2.11) Nein, es gibt kein globales Modell der Architektur.

Koordinierer (6.2.12) Nein, einen globalen Koordinierer gibt es nicht.

Worklets (6.2.13) Ja. Beachte die Ausführungen in Abschnitt 6.2.12

Effektoren (6.2.14) Nein.

Übersetzer PIM–PSM Worklets/Effektoren (6.2.15) Nein.

6.3.3. Anwendungsfall 3 — Cache-Policy

Die Idee zu dem dritten Anwendungsfall entstammt [David und Ledoux 2003]. Das Anwendungsbeispiel besteht aus einem Bildbetrachter, mit dem verschiedene Bildformate (JPEG, PNG[17]...) betrachtet werden können. Die Bilder können von einem lokalen oder entfernten Ort geladen werden. Die Anwendung besteht somit aus den drei Komponenten GUI zum Betrachten, Decoder-Komponente sowie einem URL-Loader.

[17] Portable Network Graphic

Adaptionsoperationen

Für das Anwendungsbeispiel skizzieren David und Ledoux drei mögliche Adaptionen:

1. *Komponente hinzufügen.* Dynamisches Zuschalten einer Cache-Komponente, wenn die durchschnittliche Bearbeitungszeit für das Laden oder Dekodieren[18] eine bestimmte Zeit unterschreitet.

2. *Parameter einer Komponente anpassen.* Die Größe des Cache wird automatisch an den verfügbaren Speicher des Systems angepasst.

3. *Komponente austauschen.* Die Cache-Policy (LRU[19] vs. MRU[20]) wird angepasst.

Wie eingangs bereits erwähnt, wird für eine Evaluation nur der (hier letztgenannte) Adaptionsmechanismus *Komponente austauschen* betrachtet.

In dem Anwendungsbeispiel wird standardmäßig die LRU-Komponente verwendet. LRU verwirft dasjenige Objekt, das am längsten nicht benutzt wurde (least used). Dieser Algorithmus arbeitet gut für einen zufälligen Zugriff auf die Bilder. Ein Nachteil ergibt sich allerdings, wenn die Bilder sequentiell angefordert werden, da dann bereits zuvor angeforderte Bilder längst aus dem Cache gelöscht sein können. In dem Fall ist es besser, den MRU-Algorithmus zu verwenden.

Eine Adaptionsregel für den Austausch der LRU-Komponente gegen die MRU-Komponente und umgekehrt kann wie folgt sein:

1. Wenn die letzten n Zugriffe aufeinander folgend waren, dann tausche die LRU-Komponente gegen die MRU-Komponente aus.

2. Wenn die letzten m Zugriffe *nicht* aufeinander folgend waren, dann tausche die MRU-Komponente gegen die LRU-Komponente aus.

Software-Architektur

Auch bei diesem Anwendungsfall ist die Software-Architektur nur eine sehr kleine Untermenge der in der vorliegenden Arbeit konzipierten. Zudem ist sie nicht mit Cactus vergleichbar.

Beobachten (6.1.2) Ja, Umgebung (Zugriffsverhalten des Clients) beobachten.

[18] Der Cache wird somit zwischen GUI und Decoder platziert, damit er nicht nur das Laden der Dateien beschleunigt, sondern auch die Dekodierungskosten senkt.
[19] Least Recently Used
[20] Most Recently Used

Beobachtungen auswerten (6.1.3) Keine weitere Analyse notwendig.

Änderungen planen/formulieren (6.1.4) Es muss nur geplant werden, wobei die zwei oben aufgeführten Regeln die Entscheidung herbeiführen. Eine Formulierung der Änderungsoperationen ist nicht zwingend notwendig, da die Rekonfigurationsoperation fest im Programm verdrahtet werden kann.

Änderungen analysieren (6.1.5) Keine notwendig.

Änderungsskripte in systemspezifische Operationen übersetzen (6.1.6) David und Ledoux benutzen nur eine Plattform.

Adaptionspaket vorbereiten (6.1.7) Nein.

Adaptionspaket deployen (6.1.8) Nein.

Koordinierte Ausführung/Adaption (6.1.9) Keine Koordination notwendig, da nur Austausch von einer Komponente auf einem Host. Es muss nur der Zustand der alten Komponente zur neuen transferiert werden.

Monitore (6.2.1) Ja, Umgebung (Zugriffsverhalten des Clients) beobachten.

Gauges (6.2.2) Keine weitere Analyse notwendig.

DB Anwendungskomponenten (6.2.3) Ja.

DB Komponentenspezifikationen (6.2.6) Ja.

Planer (6.2.7) Ja, offener Regelkreis.

Analysierer (6.2.8) Nein.

DB Adaptionsregeln (6.2.9) Ja, wenn nicht fest verdrahtet.

DB Adaptionsziele (6.2.10) Nein. Die Anwendung muss nur global einen Zähler für aufeinander folgende Zugriffe verwalten.

Konfigurationsmanager (6.2.11) Kein globaler KM notwendig. Ob es einen lokalen KM gibt, hängt von dem genutzten Komponentenframework ab.

Koordinierer (6.2.12) Nein, keine Koordination ist notwendig, weil sich die Adaption nur auf einen Host beschränkt. Ebenso ist der Austausch im laufenden System nicht kritisch. Lediglich der Zustand der LRU- oder MRU-Komponente muss gesichert/transferiert werden.

Worklets (6.2.13) Nein, weil keine Koordination auf dem Host notwendig ist.

Effektoren (6.2.14) Nein. Adaptionsoperation ist in der Anwendung fest verdrahtet.

Übersetzer PIM–PSM Worklets/Effektoren (6.2.15) Nein. Adaption an dem System wird nicht auf Basis eines Architekturmodells vorgenommen, das ein Übersetzen in die jeweilige Plattform erfordern würde.

6.3.4. Anwendungsfall 4 — frei verfügbare Währungsrechner

Der vierte und letzte Anwendungsfall baut auf dem bereits auf Seite 196 beschriebenen und in Abbildung 6.4 auf Seite 197 dargestellten Anwendungsfall einer Service-orientierten Architektur (SOA) auf. Charakter dieses Anwendungsfalles ist nicht unbedingt das SOA-Prinzip, denn der wesentlichste Punkt von SOA ist die lose Koppelung [Ullmann 2004]. Das Merkmal des hier beschriebenen ist die freie Nutzung von Diensten Dritter oder das fehlende Management, dem die Architekturkomponenten für den Dienstnehmer (Client) bzw. die gesamte Anwendung untersteht. Damit ist die Anwendung mit Service-orientierten Architekturen stark vergleichbar.

Der Anwendungsfall im Konkreten ist eine Software, die einen Währungsrechner benutzt. Der Währungsrechner kann von Dritten im Internet bereitgestellt sein oder einer von vielen im Unternehmen deployten Diensten sein.

Adaptionsoperation

Über Resource Discovery erfährt der Client, dass mehrere Währungsrechner verfügbar sind. Diese unterscheiden sich hinsichtlich monetärer Kosten, Anzahl der stellbaren Anfragen pro Minute (verhalten sich proportional zu den Kosten), verfügbare Währungen, für die eine Umrechnung vorgenommen werden kann, unterstützter Wertebereich etc.

Wie auch bei den Anwendungsfällen zuvor wird der Austausch einer Komponente betrachtet. Die konzipierte Anwendung nutzt eine andere Komponente, wenn die derzeitig genutzte Währungsrechner-Komponente für die gegebene Situation nicht mehr optimal ist.

Software-Architektur

Beobachten (6.1.2) Ja.

Beobachtungen auswerten (6.1.3) Die verschiedenen Faktoren, die die jeweilige Währungsrechnerkomponente charakterisieren, müssen ausgewertet werden, um die Verwendbarkeit für die gegebene Situation ermitteln zu können.

Änderungen planen/formulieren (6.1.4) Nur Planung.

Änderungen analysieren (6.1.5) Der Client muss überprüfen, ob die Schnittstelle des neuen Währungsrechner zu seiner kompatibel ist. Gegebenenfalls müssen verschiedenen Ebenen der Interoperabilität auf Client-Seite angepasst werden. Beispiel: Die gewünschte Genauigkeit für die Nachkommastellen für ein Ergebnis wird entweder vorher festgelegt oder bei jeder Anfrage über einen Parameter angegeben (vergleichbar mit dem Drucker-Beispiel auf Seite 102). Das Interaktionsmuster muss entsprechend angepasst werden, indem ggf. ein Adapter generiert wird.

Änderungsskripte in systemspezifische Operationen übersetzen (6.1.6) Nein.

Adaptionspaket vorbereiten (6.1.7) Nein. Es ist kein Adaptionspaket notwendig.

Adaptionspaket deployen (6.1.8) Nein.

Koordinierte Ausführung/Adaption (6.1.9) Keine Koordination notwendig. Der Client muss lediglich zu dem anderen Dienstanbieter binde. Eine Berücksichtigung des Zustandes der alten Währungsrechner-Komponente ist nicht notwendig. Falls sowohl alte als auch neue Währungsrechner-Komponente die Voreinstellung für die gewünschte Anzahl an Nachkommastellen erlaubt, dann ist dies zwar ein Zustand, aber dieser muss nicht von der alten zur neuen Komponente transferiert werden. Stattdessen initialisiert der Client die neu genutzte Währungsrechner-Komponente selbst.

Monitore (6.2.1) Ja. Resource Discovery abfragen oder benachrichtigen lassen.

Gauges (6.2.2) Die verschiedenen Faktoren, die die jeweilige Währungsrechnerkomponente charakterisieren, müssen ausgewertet werden, um die Verwendbarkeit für die gegebene Situation ermitteln zu können.

DB Anwendungskomponenten (6.2.3) Nein. Es gibt keine Datenbank mit Anwendungskomponenten (siehe Abschnitt 6.2.3).

DB Komponentenspezifikationen (6.2.6) Ja, über einen RD-Dienst.

Planer (6.2.7) Ja, offener Regelkreis.

Analysierer (6.2.8) Ja. Siehe Ausführungen zur gleichnamigen Aktivität.

DB Adaptionsregeln (6.2.9) Ja. Können kompliziert werden, wenn verschiedene Faktoren ausschlaggebend sind, z. B.: Stets günstigsten Anbieter nutzen, aber immer mindestens zwei Nachkommastellen notwendig und ...

DB Adaptionsziele (6.2.10) Nein.

Konfigurationsmanager (6.2.11) Nein, es gibt kein zentrales Modell der Architektur.

Koordinierer (6.2.12) Keine Koordination notwendig (siehe oben).

Worklets (6.2.13) Nein.

Effektoren (6.2.14) Nein. Initialisierung der neuen Komponente wird vom Client selbst im Rahmen der Dienstanfrage vorgenommen.

Übersetzer PIM–PSM Worklets/Effektoren (6.2.15) Nein.

Mit diesem vierten konstruierten Anwendungsfall für den Austausch einer Komponente soll die Evaluation der Empfehlungen im Einzelnen abgeschlossen werden. Im nächsten Abschnitt wird ein Fazit gezogen, das die Empfehlungen aus den Abschnitten 6.1 und 6.2 in Bezug auf die Ergebnisse der Evaluation im Großen bewertet.

6.4. Fazit

Tabelle 6.2 auf der nächsten Seite fasst die im vorhergehenden Abschnitt vorgenommene Evaluation der empfohlenen Software-Architektur zusammen.

Aus der Auswertungstabelle können die folgenden Schlüsse gezogen werden.

1. Die tatsächliche Verwendung von empfohlenen Komponenten bzw. Aktivitäten variiert sehr stark von Anwendungsfall zu Anwendungsfall. Insbesondere für die Datenbank für Anwendungskomponenten und ihre zugehörige Datenbank für Komponentenspezifikationen kann keine klare Aussage getroffen werden, obwohl bei dem Adaptionsmechanismus Komponente austauschen anzunehmen ist, dass vorgenannte DB-Komponenten in der Software-Architektur enthalten sein müssen.

   Des Weiteren besteht kein Zusammenhang zwischen der Verwendung der Komponentendatenbank zur Verwendung der Spezifikationsdatenbank. In Cactus ist erstere in der Software-Architektur vorhanden, dafür letztere nicht. Für das Währungsrechner-Beispiel verhält es sich genau umgekehrt.

2. Für den gleichen Adaptionsmechanismus gibt es je nach zu unterstützendem System einen einfachen Ansatz (LRU/MRU sowie Währungsrechner) und einen aufgeblähten, aber allumfassenden, generellen Architekturansatz (der im Rahmen der Arbeit konzipierte).

Aktivitäten und Komponenten	Anwendungsfälle: beliebiges System	Cactus	LRU/ MRU	Währungs-rechner
Beobachten	•	•	•	•
Beobachtungen auswerten	•	–	–	•
Änderungen planen/formulieren	•	○	○	○
Änderungen analysieren	•	–	–	–
Änderungsskripte in systemspezifische Operationen übersetzen	•	–	–	–
Adaptionspaket vorbereiten	•	–	–	–
Adaptionspaket deployen	•	–	–	–
Koordinierte Ausführung (Adaption)	•	•	○	○
Monitore	•	•	•	•
Gauges	•	–	–	•
DB Anwendungskomponenten	•	•	•	–
Alternative Verarbeitungsknoten[a]	•	–	–	–
DB Komponentenspezifikationen	•	–	•	•
Planer	•	•	•	•
Analysierer	•	–	–	–
DB Adaptionsregeln	•	–	•	•
DB Adaptionsziele	•	–	–	–
Konfigurationsmanager	•	–	–	–
Koordinierer	•	•	–	–
Worklets	•	–	–	–
Effektoren	•	–	–	–
Übersetzer PIM–PSM Worklets/Effektoren	•	–	–	–

Tabelle 6.2.: Vergleich der Anwendungsfälle hinsichtlich Umsetzung der empfohlenen dynamischen und statischen Aspekte für Adaptionsmechanismus Komponente austauschen. Legende: • = vorhanden, – = nicht vorhanden, ○ = beschränkt vorhanden in der Software-Architektur. Im Fall von beschränkt vorhandenen Aktivitäten/Infrastruktur-Komponenten siehe die Anmerkungen zu den jeweiligen Punkten in Abschnitt 6.3.
[a] Für Adaptionsmechanismus Komponente austauschen eigentlich nicht relevant.

3. Die in Abschnitt 5.1 vorgestellten Spezifikationsmethoden sind nur obligatorisch. Methoden zur Beschreibung dynamischer Architekturen finden nur einmal Verwendung. Die Beschreibung von funktionalen und nicht-funktionalen Eigenschaften von Komponenten wird bei drei von vier Anwendungsfällen vorgenommen.

4. Obwohl als fundamental in Abschnitt 5.3 charakterisiert, findet das Reflection-Muster nur in Anwendungsfall 1 und in Anwendungsfall 3 Verwendung.

5. Die in Abschnitt 4.4 (Fazit Adaptionsmechanismen) beschriebene Notwendigkeit eines Change Management, das die Wahrung der Konsistenz und der Integrität sicherstellt, ist relativ zu sehen.

6. Ob zentrales oder verteiltes Management zu nutzen oder überhaupt nutzbar ist, das ist abhängig von der Art des verteilten Systems.

7. Der kleinste gemeinsame Nenner für eine Software-Architektur, die den Adaptionsmechanismus Komponente austauschen ermöglicht, besitzt nur die Monitor-Komponenten für die Beobachtung der Umgebung und/oder sich selbst sowie einen Planer, der entscheidet, ob und welche Adaption vorgenommen werden soll.

Zusammenfassend lässt sich feststellen, dass eine klare Empfehlung für die Software-Architektur eines adaptiven Systems nicht gegeben werden kann. Selbst wenn, wie hier geschehen, nur ein Adaptionsmechanismus unterstützt werden soll, sind die möglichen Software-Architekturen sehr unterschiedlich. Der konzipierte »Baukasten«, der Empfehlungen hinsichtlich der dynamischen und statischen Aspekte gibt, ist für den Benutzer wenig hilfreich. Er empfiehlt Komponenten oder Aktivitäten, die nur eingeschränkt anwendbar sind. Der Benutzer des »Baukastens« muss selbst abwägen, ob die beschriebene Aktivität/Komponente für sein System tatsächlich notwendig ist. Das seltene Vorkommen von •-Punkten in der Auswertungstabelle zeigt eindrucksvoll, dass die Empfehlungen selten Anwendung finden (brauchen). Damit ist zwar ein allumfassender Software-Architektur-Ansatz entwickelt worden, aber er ist nur eingeschränkt nutzbar.

As our circle of knowledge expands, so does the circumference of darkness surrounding it.

Albert Einstein

7 Zusammenfassung

Die vorliegende Arbeit hatte adaptive Architekturen für verteilte Systeme als Untersuchungsgegenstand. Das Ziel bestand darin, eine Literaturrecherche vorzunehmen und verschiedene Ansätze für die Software-Architektur vorzustellen, um am Schluss Empfehlungen für die Software-Architektur abzugeben.

Kapitel 2. Zu Beginn der Arbeit wurden die Begriffe Software-Architektur, verteiltes System und Selbstorganisation erläutert sowie die verwandten Forschungsthemen Autonomic Computing und Organic Computing beschrieben. Weitere verwandte Themen, die jedoch nicht explizit vorgestellt wurden, sind die mit Autonomic Computing vergleichbaren Initiativen in der Industrie sowie die Projekte SAS und DASADA der DARPA (vgl. Mindmap auf Seite 230).

Kapitel 3. Für die Arbeit bestand kein Anlass in der Entwicklung einer spezifischen Software, sondern es sollten Architekturenansätze genereller adaptiver Software-Systeme aufgezeigt werden. Deshalb wurde in Kapitel 3 eine Begriffsbestimmung vorgenommen. Ziel der Begriffsbestimmung war es, Merkmale von adaptiver Software zu identifizieren, um daraus Schlussfolgerungen für die statischen und dynamischen Aspekte einer Software-Architektur ableiten zu können. Die bei der Herangehensweise genutzten Wörterbücher für die englischen Begriffe, Enzyklopädieeinträge, Beispiele und Definitionen brachten nur wenig Klarheit. Sie boten allerdings genug Diskussionsstoff, um die Thematik im Ganzen zu verstehen. Das Nutzen von einer Fülle von Definitionen brachte den Vorteil, dass alle Aspekte für Adaptivität und Adaptierbarkeit berücksichtigt werden konnten — eine Untermenge hätte nur zu einem eingeschränkten Verständnis geführt. Außerdem konnte demonstriert werden, dass es vollkommen unzureichend ist, eine

Definition nur anzuführen und — wenn überhaupt geschehen (z. B. bei Kontext-Definition von [Dey 2000] nicht) — zu interpretieren. Viel mehr muss auch das Umfeld untersucht werden, in dem die Definition steht. Für das konkrete Beispiel der »Anpassung an den Kontext« konnte in Tabelle 3.4 auf Seite 40 eindrucksvoll aufgezeigt werden, dass Lieberherr und Lopes für »Anpassung an Kontext« etwas ganz anderes meinen als beispielsweise Springer oder die Allgemeinheit, die den Kontext-Begriff mit der gleichnamigen aktuellen Forschungsrichtung verbindet.

Mithilfe der im Rahmen der Begriffsbestimmung vorgenommenen Analyse konnte ein kleiner Beitrag für ein besseres Kontext-Verständnis[1] geleistet werden sowie die vorhandenen textuellen als auch formalen Definitionen für adaptive und für adaptierbare Software verbessert werden. Für eine Kategorisierung der breiten Thematik wurden die vier W-Fragen formuliert und mit ihnen eine Einschränkung der zu vertiefenden Themas für die Arbeit vorgenommen. Als zu bearbeitendes Problem ist Adaptivität auf Software-Architektur-Ebene (Adaptionsgegenstand Komponenten und Beziehungen) zur Laufzeit festgelegt worden.

Kapitel 4. In dem sich an die Begriffsbestimmung anschließenden Kapitel wurden Adaptionsmechanismen für eine Software-Architektur identifiziert und bezüglich Adaptivität und Adaptierbarkeit eingeordnet. Die Einordnung konnte nicht immer klar vorgenommen werden, da verschiedene Ansichten zu der Problematik möglich sind (z. B. eigene Schnittstellenanpassung). Außerdem sind viele Aussagen für die Möglichkeit, dass die Software die Adaptionsoperation vornimmt, davon abhängig, wie gut die KI in der Software ist. Heute formulierte Aussagen können für zukünftige Software nicht mehr zutreffen — und umgekehrt.

Für ein erweitertes Verständnis von Adaptionsvorgängen in Software wurden außerdem die Basismechanismen für die Adaption der Daten und der Kommunikation aus einer verwandten Arbeit angeführt.

Entgegen den Adaptionsmechanismen für die Daten und die Kommunikation erfordern Adaptionsmechanismen wie der Austausch oder das Entfernen von Komponenten aus der Software-Architektur zur Laufzeit mehr Vorsicht, um die Integrität und Konsistenz für das laufende System nicht zu gefährden. Zum einen muss der Zustand der auszutauschenden Komponente berücksichtigt werden. Außerdem erschwert die Adaption in einem verteilten System die Koordinierung von zusammenhängenden Operationen.

Kapitel 5. Für die betrachtete Thematik der Adaptivität auf Software-Architektur-Ebene wurde im fünften Kapitel eine Literaturrecherche

[1] Für den Kontext-Begriff aus der Forschungsrichtung Context-Awareness, nicht für den Kontextbegriff von Lieberherr und Lopes.

nach Unterstützungsmechanismen und konkreten Architekturansätzen vorgenommen. Vorgestellt wurden für die Beschreibung von dynamischen Architekturen die ADLs in Verbindung mit den AMLs sowie, zur Überprüfung der architektonischen Randbedingungen, die ACLs und des Weiteren einige formale Ansätze.

Danach wurde eine Einführung in die Kontroll-Theorie gegeben. Dabei fanden der offene (Optimalwertsteuerung) und der geschlossene Regelkreis (Rückkoppelungssteuerung) und darüber hinaus die erweiterten Modelle der Adaptivregelung und des rekonfigurierbaren Regelungssystems Berücksichtigung. Als konkrete Architekturansätze wurde zum einen die Kombination aus offenem und geschlossenem Regelkreis aufgezeigt. Zum anderen wurde ein integrierter Ansatz vorgestellt, der eine Rückkoppelungssteuerung, Adaptivregelung und ein rekonfigurierbares Regelungssystems vereint. Zusammenfassend konnte keine klare Empfehlung abgegeben werden, welches Modell für adaptive Systeme zu verwenden ist, da für jedes Modell eine Berechtigung in der Software-Architektur festgestellt werden kann. Gegeben wurden deshalb einige Empfehlungen, die bei der Auswahl eines konkreten Modells helfen. Außerdem konnten der essentielle Zusammenhang zwischen Adaptivität und Selbstmanagement hergestellt werden.

Im Anschluss daran wurde mit Reflection bzw. den Meta-Level-Architekturen ein fundamentales Muster für die Entwicklung von (selbst-) adaptiver und auch adaptierbarer Software beschrieben. Reflection ist ein fundamentales Konzept für adaptierbare und für adaptive Systeme, da es zum einen Systemen ermöglicht, Wissen über sich selbst offen zu legen, zum anderen erlaubt es die eigene Manipulation. Die theoretische Beschreibung durch Praxisbeispiele abrundend wurden zum Abschluss zwei Middleware-Ansätze vorgestellt und bewertet.

Konkret für die Entwicklung adaptiver Software wurden außerdem einige weitere Muster für die Entwicklung von adaptiven Systemen beschrieben

Den Abschluss des fünften Kapitels bildete die Vorstellung drei weiterer konkreter Architekturansätze. Bei den Ansätzen konnten der einfache und sichere Austausch von Kommunikationskomponenten im laufenden Betrieb, der Austausch von Komponenten auf Architekturebene sowie autonomes Agieren des Systems aufgezeigt werden.

Kapitel 6 — Fazit. Im letzten Kapitel wurden die im Rahmen der Recherche gewonnen Erkenntnisse in einem generellen Ansatz zusammengefasst. Es wurden Empfehlungen sowohl für die dynamischen als auch statischen Aspekte einer adaptiven Software-Architektur abgeben. Im Anschluss daran wurde die konzipierte Software-Architektur mithilfe von vier konstruierten Anwendungsfällen evaluiert.

Zusammenfassend lässt sich feststellen, dass eine klare Empfehlung für die Software-Architektur eines adaptiven Systems nicht gegeben werden kann. Selbst wenn, wie hier geschehen, nur ein Adaptions-

mechanismus unterstützt werden soll, sind die möglichen Software-Architekturen sehr unterschiedlich. Der konzipierte »Baukasten«, der Empfehlungen hinsichtlich der dynamischen und statischen Aspekte gibt, ist für den Benutzer wenig hilfreich. Er empfiehlt Komponenten oder Aktivitäten, die nur eingeschränkt anwendbar sind. Der Benutzer des »Baukastens« muss selbst abwägen, ob die beschriebene Aktivität/Komponente für sein System tatsächlich notwendig ist. Das seltene Vorkommen von •-Punkten in der Tabelle 6.2 auf Seite 220 zeigt eindrucksvoll, dass die Empfehlungen selten Anwendung finden (brauchen). Damit ist zwar ein allumfassender Software-Architektur-Ansatz entwickelt worden, aber er ist nur eingeschränkt nutzbar.

Für die vorliegende Arbeit wäre es besser gewesen, nur eine konkrete Anwendung zu betrachten.

Ausblick

Lackmustest. In Abschnitt 3.6.7 wurde versucht, im Rahmen einer formalen Definition für einen Adaptionsprozess eine Art Lackmustest zu entwickeln, der einen normalen Vorgang von einem Adaptionsvorgang unterscheiden kann. Dabei konnten bereits wichtige Merkmale einer Adaption sowohl für eine präventive Aktion als auch für eine Reaktion auf Änderungen des Adaptionskontextes formuliert werden. Für eine Identifizierung ist diese formale Definition zwar besser als die in der Literatur vorhandenen Definitionen, aber für eine eindeutige Unterscheidbarkeit ist sie noch nicht voll ausreichend.

Ein Beispiel für die fehlende Eindeutigkeit der Definition wurde auf den Seiten 91 bis 94 deutlich. Ob das Hinzufügen einer Komponente in der Software-Architektur ein Adaptionsmechanismus ist, hängt essentiell davon ab, was die Komponenten sind (hier Typ vs. Instanz) und wer die Operation vornimmt. Der gleiche kritische Sachverhalt kann sich beispielsweise auch für die Parameteränderung einer Komponente ergeben: Wenn der Benutzer in seinem Web-Browser in das URL-Feld eine andere Adresse eingibt, ist dies keine Adaption. Wenn er jedoch in dem Optionen-Dialog die Startseite in dem URL-Feld ändert, entspricht dies einer Adaption der Software an die persönlichen Bedürfnisse des Benutzers.

Für konkrete Beispiele ist es leicht, eine Unterscheidung nach Adaption und Nicht-Adaption vorzunehmen. Ein allgemeingültiger Lackmustest konnte im verfügbaren Rahmen der Arbeit jedoch nicht entwickelt werden. Für dieses Problem könnte eine weitere Bearbeitung fortgeführt werden.

Koordination und Zustandssicherung. Koordination einer Adaption in einem verteilten System sowie die Sicherung des Zustandes einer Komponente sind wichtige zu berücksichtigende Faktoren in dynamischen

Architekturen. Im Rahmen der Arbeit konnten diese nur am Rande betrachtet werden, indem auf Drittliteratur verwiesen wurde. Eine tiefer gehende Recherche könnte diese Aspekte ausführlicher betrachten.

Selbstorganisation. Die Selbstorganisation in einer Software-Architektur konnte nur minimal untersucht werden. Sie fand lediglich Berücksichtigung in Abschnitt 5.1.2 mit der Unterstützung durch ADLs. Hier könnte eine tiefer gehende Untersuchung aufsetzen.

Bei der Betrachtung »selbstorganisierender Software-Architekturen« muss allerdings zuvor geklärt werden, was »Software-Architektur« in dem Fall bedeutet. Ähnlich dem Ergebnis in Abschnitt 4.1.1 (Problem Sichten) ist es ein Unterschied, ob mit Architekturkomponenten Komponententypen oder -instanzen gemeint sind. Für den Fall der Betrachtung von Typen organisiert sich die Software-Architektur derart selbst, dass die für die Erbringung einer bestimmten Funktionalität notwendigen Komponenten selbst gefunden werden. Die Anwendung baut sich selbst zusammen. Für den anderen Fall (Instanzen) ergibt sich eine andere Selbstorganisation, indem mehrere gleiche Komponenten kooperativ eine Aufgabe lösen — ähnlich der Selbstorganisation in einer Ameisenkolonie oder bei einem Bienenvolk.

Für ein besseres Verständnis wäre es nach Meinung des Autors außerdem angebracht, den Begriff der Architektur sparsamer zu verwenden. Eine Verwendung sollte tatsächlich nur für die Struktur verwendet werden, d. h. auf der Ebene von Typen und nicht von Instanzen (vgl. die Seiten 91 bis 94). Damit würde sich auch der gleiche Adaptionsgegenstand ergeben, wenn *adaptierbare* Software-Architektur behandelt wird.

In dem Zusammenhang steht auch die Interpretierbarkeit des Titels »adaptive Architekturen für verteilte Systeme«, der der vorliegenden Arbeit unterliegt. Es ist und es wurde nicht klar ersichtlich, ob eine Software-Architektur (Software) konzipiert werden sollte, die bliebige Adaptionen in einem verteilten System vornehmen sollte (Adaptionsgegenstände Daten, Kommunikation, Routing, Sicherheits- und Transaktionsregeln, Architektur etc.) oder ob »lediglich« die Software-Architektur der Adaptionsgegenstand ist.

A Mindmap Diplom

Umseitig ist eine Mindmap abgebildet, die einen groben Überblick über die vorliegende Diplomarbeit bietet.

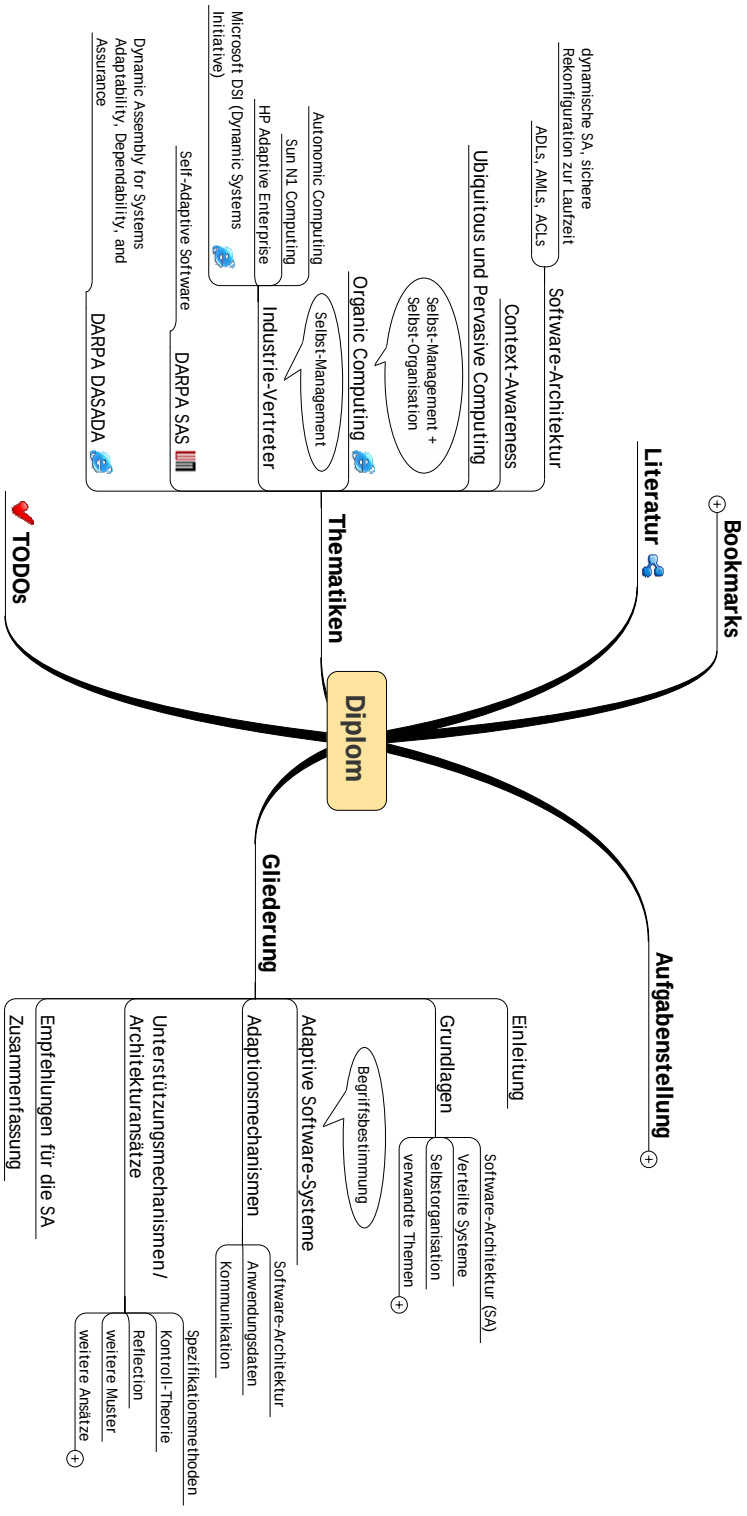

Abbildung A.1.: Mindmap zum Diplom

B

Mindmap Adaptionsmechanismen

Umseitig ist eine Mindmap abgebildet, die die in Kapitel 4 angeführten Adaptionsmechanismen zusammenfasst.

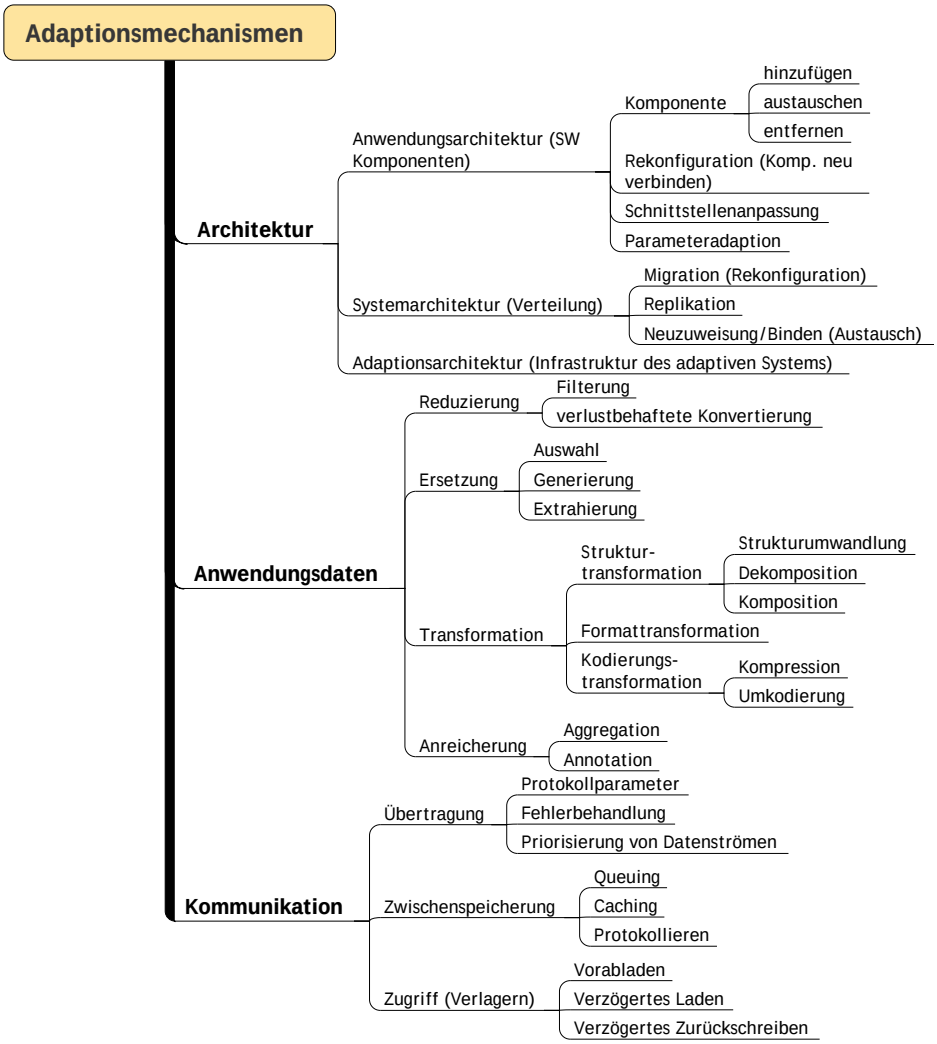

Abbildung B.1.: Mindmap für die betrachteten Adaptionsmechanismen

C
Inhalt der CD

Zusätzlich zu der vorliegenden Printausgabe der Diplomarbeit finden sich auf der beigelieferten CD weitere Materialien. Die nachstehende Aufstellung beschreibt die Verzeichnisstruktur und den jeweiligen Inhalt der CD:

`build/dist/` beinhaltet die Diplomarbeit im PDF sowie die gesamte Bibliographie mit Abstracts ebenso im PDF und HTML-Format (noch mehr als die referenzierte, rd. 400 Einträge, nicht deckungsgleich mit den nachfolgend genannten Dokumenten),

`Literatur/` die für die Diplomarbeit recherchierte Literatur für tiefer gehende Nachforschung zur Thematik (rd. 400 Dokumente),

`src/images/` alle in der Arbeit verwendeten Abbildungen im PDF, EPS- und GIF/JPEG-Format,

`src/tex/` die LaTeX- und BibTeX-Quellen,

`src/uml/` die in UML modellierten Diagramme,

`src/mindmaps/` eine Vielzahl an Mindmaps mit strukturiert aufbereiteten Informationen und Links zu weiterer Literatur/Konferenzen. Ein kostenloses Betrachtungsprogramm kann von der Website des Herstellers `www.mindjet.de` heruntergeladen werden.

DaimlerChrysler-Literatur. Die in der vorliegenden Arbeit genutzte Literatur der DaimlerChrysler AG [Vögler *et al.* 2004] konnte nicht auf die CD gebrannt werden, da dieses Material Vertraulichkeitsstatus besitzt.

Lizenzrechtlicher Hinweis. Das Verzeichnis `Literatur/`, das die für die Recherche verwendete komplette Literatur in PDF-Form beherbergt, enthält in einigen Unterverzeichnissen geschütztes Material von ACM[1], IEEE und des Springer-Verlages. Die Recherche in den Digitalen Bibliotheken dieser Organisationen wurde durch die Campus-Lizenz der SLUB[2] ermöglicht. Die Verwendung dieser Materialien durch Nicht-TU-Dresden-Angehörige stellt deshalb einen Verstoß gegen die Lizenzbestimmungen dar.

Die lizenzrechtlich geschützten Materialien der o. g. Institutionen sind besonders gekennzeichnet, indem sie sich in separaten Verzeichnissen befinden. Die nachfolgend genannten Verzeichnisse beinhalten geschütztes Material:

- `Literatur/Adaptive/acm.org/`
- `Literatur/Adaptive/IEEE Xplore/`
- `Literatur/Adaptive/LNCS 1936/`
- `Literatur/Adaptive/LNCS 2618/`
- `Literatur/Adaptive/springer/`
- `Literatur/Adaptive/WOSS 2002 - 1st Workshop on Self-healing systems/`
- `Literatur/Software Architecture/acm.org/`

Im Literaturverzeichnis wurden diese Materialien um OpenURLs und DOI[3]s in Form von URLs ergänzt, die zu dem jeweiligen Dokument in der Digitalen Bibliothek der veröffentlichenden Institution führen und ein Erwerben der elektronischen Version ermöglichen.

Das Material in anderen Verzeichnissen konnte frei aus dem WWW bezogen werden.

[1] Association for Computing Machinery
[2] Sächsische Landesbibliothek — Staats- und Universitätsbibliothek Dresden
[3] Digital Object Identifier

Literaturverzeichnis

Hinweis: Für Internetadressen, bei denen das Veröffentlichungs- oder Änderungsdatum vom Autor der Seite nicht angegeben wurde, wird für eine Datierung im Verzeichnis das Zugriffsdatum verwendet. Das Zugriffsdatum als solches wird nicht explizit angeführt, stattdessen gilt hierfür das Abgabedatum der vorliegenden Arbeit. Das bedeutet auch, dass die Aktualität der Internetadressen für diesen Zeitpunkt sichergestellt wird. Falls danach Internetadressen nicht mehr erreichbar sind, so wird auf den Dienst von archive.org verwiesen.

Es wurde außerdem versucht, Literatur aus Digitalen Bibliotheken zu verwenden, da in dem Fall sowohl für eine leichte als auch allzeitige Erreichbarkeit gesorgt ist. Dafür sind diese Quellen mit einem OpenURL/DOI versehen (vgl. [Stieler und Marsiske 2003]), dessen URL zu dem Artikel führt — für das Herunterladen der Literatur ist jedoch eine Autorisation (Geld/Lizenz) vonnöten.

Die einzelnen Zahlen hinter den Literatureinträgen geben jeweils die Seitenzahl ihrer Verwendung in der vorliegenden Arbeit an.

Åström und Wittenmark 1989 ÅSTRÖM, Karl J. ; WITTENMARK, Björn: *Adaptive Control*. Addison-Wesley, 1989. – ISBN 0-201-09720-6 137

ACME 1998 CARNEGIE MELLON UNIVERSITY: *The Acme Architectural Description Language*. 1998. – URL http://www-2.cs.cmu.edu/~acme/ 9

Act-Net Consortium 1996 ACT-NET CONSORTIUM, Corporate: The Active Database Management System Manifesto: A Rulebase of ADBMS Features. In: *SIGMOD Record* 25 (1996), Nr. 3, S. 40–49. – DOI http://doi.acm.org/10.1145/234889.234896. – ISSN 0163-5808 142, 202

Aigner *et al.* 2003 AIGNER, Ronald ; FRANZ, Elke ; GÖBEL, Steffen ; HÄRTIG, Hermann ; HUSSMANN, Heinrich ; MEISSNER, Klaus ; MEYER-WEGENER, Klaus ; MEYERHÖFER, Marcus ; PFITZMANN, Andreas ; POHL, Christoph ; POHLACK, Martin ; RÖTTGER, Simone ; SCHILL, Alexander ; WEHNER, Frank ; ZSCHALER, Steffen: Zwischenbericht der DFG-Forschergruppe 428: „Components with Quantitative Properties and Adaptivity (COMQUAD)“ / Technische Universität Dresden und Friedrich-Alexander Universität Erlangen-Nürnberg. URL ftp://ftp.inf.tu-dresden.de/pub/berichte/tud03-10.pdf, 10. August 2003 (TUD-FI-03-10-August). – Forschungsbericht. – ISSN 1430-211X 47, 203

Allen *et al.* 1998 ALLEN, Robert ; DOUENCE, Rémi ; GARLAN, David: Specifying and Analyzing Dynamic Software Architectures. In: *Fundamental Approaches to Software Engineering* Bd. 1382 der Reihe *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, 1998, S. 21–37. – OpenURL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=1382&spage=21. – ISSN 0302-9743 126

Almeida 2001 ALMEIDA, João Paulo A.: *Dynamic Reconfiguration of Object-Middleware-based Distributed Systems*, University of Twente, Enschede, The Netherlands, Master's Thesis, Juni 2001. – URL http://www.cs.utwente.nl/~alme/cvitae/MScThesis[Almeida]1.0.pdf 77, 121, 128, 209

Amberg *et al.* 2003 AMBERG, Michael ; WEHRMANN, Jens ; OKUJAVA, Shota: Adaptive Software: Protokollierung, Verwaltung und Verwertung von Nutzungsprofilen in betriebswirtschaftlicher Standardsoftware unter besonderer Berücksichtigung von Personalisierungsfunktionen / Friedrich-Alexander Universität Erlangen-Nürnberg. URL http://www.wi3.uni-erlangen.de/forschung/publikation/PDF/Adaptive_Software.pdf, 2003 (Wirtschaftsinformatik III, Nr. 01/2003). – Forschungsbericht 25, 27, 29, 30, 32, 81, 101

Amsterdam 1972 AMSTERDAM, B. K.: Mirror self-image reactions before age two. In: *Developemental Psychology* (1972), Nr. 5, S. 297–305 15

Andresen 2003 ANDRESEN, Andreas: *Komponentenbasierte Softwareentwicklung mit MDA, UML und XML*. Carl Hanser Verlag, 2003. – ISBN 3-446-22282-0 4, 54

ApICS 2001 APPLIED INDUSTRIAL CONTROL SOLUTIONS LLC (APICS LLC): *When to use feedforward feed-forward control and feedback control in industrial automation applications*. 2001. – URL http://www.apicsllc.com/apics/Misc/ff.html 136

ASF 2004 APACHE SOFTWARE FOUNDATION (ASF): *Multi-Processing Modules (MPMs) — Apache HTTP Server*. 2004. – URL http://httpd.apache.org/docs-2.0/mpm.html 33

Ayars *et al.* 2001 AYARS, Jeff ; BULTERMAN, Dick ; COHEN, Aaron ; DAY, Ken ; HODGE, Erik ; HOSCHKA, Philipp ; HYCHE, Eric ; JOURDAN, Muriel ; KIM, Michelle ; KUBOTA, Kenichi ; LANPHIER, Rob ; LAYAÏDA, Nabil ; MICHEL, Thierry ; NEWMAN, Debbie ; OSSENBRUGGEN, Jacco van ; RUTLEDGE, Lloyd ; SACCOCIO, Bridie ; SCHMITZ, Patrick ; KATE, Warner ten: *Synchronized Multimedia Integration Language (SMIL 2.0)*. 7. August 2001. – URL http://www.w3.org/TR/2001/REC-smil20-20010807/. – W3C Recommendation 30, 54

Balzer 1996 BALZER, Robert: *Enforcing Architecture Constraints*. Oktober 1996. – International Software Architecture Workshop (ISAW-2) 129

Balzer und Goldman 1999 BALZER, Robert M. ; GOLDMAN, Neil M.: Mediating Connectors. In: *Electronic Commerce and Web-based Applications/Middleware, Proceedings. 19th IEEE International Conference on Distributed Computing Systems Workshops*, IEEE, Juni 1999, S. 73–77. – DOI http://doi.ieeecomputersociety.org/10.1109/ECMDD.1999.776417 193

Balzert 1996 BALZERT, Helmut: *Lehrbuch der Software-Technologie: Software-Entwicklung*. Bd. 1. Spektrum-Verlag, 1996. – ISBN 3-8274-0042-2 23

Bass *et al.* 2003 BASS, Len ; CLEMENTS, Paul ; KAZMAN, Rick: *Software architecture in practice*. 2. Auflage. Addison-Wesley, 2003. – ISBN 0-321-15495-9 xvii, 4, 5, 6, 180, 181

Ben-Shaul *et al.* 2001a BEN-SHAUL, Israel ; GAZIT, Hovav ; HOLDER, Ophir ; LAVVA, Boris: Dynamic Self Adaptation in Distributed Systems. In: *Self-Adaptive Software: First International Workshop, IWSAS 2000, Oxford, UK, April 2000* Bd. 1936 der Reihe *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, 2001, S. 134–142. – OpenURL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=1936&spage=134. – ISSN 0302-9743 52

Ben-Shaul *et al.* 2001b BEN-SHAUL, Israel ; HOLDER, Ophir ; LAVVA, Boris: Dynamic Adaptation and Deployment of Distributed Components In Hadas. In: *IEEE Transactions on Software Engineering* 27 (2001), Nr. 9, S. 769–787. – DOI http://dx.doi.org/10.1109/32.950315 100, 101

Bialek und Jul 2004 BIALEK, Robert ; JUL, Eric: A Framework for Evolutionary, Dynamically Updateable, Component-based Systems. In: *Proceedings of the 24th International Conference on Distributed Computing Systems Workshops - W7: EC (ICDCSW'04)*, IEEE Computer Society, 2004, S. 326–331. – URL http://dares.enst-bretagne.fr/dares2004/Session2/paper6.pdf. – ISBN 0-7695-2087-1 101

Bierce 1989 BIERCE, Ambrose: *The Enlarged Devil's Dictionary*. Penguin Books, 1989 3

Blair *et al.* 2000a BLAIR, G[ordon] S. ; ANDERSEN, A[nders] ; BLAIR, L[ynne] ; COULSON, G[eoff] ; SÁNCHEZ, D.: Supporting dynamic QoS management functions in a reflective middleware platform, URL http://www.comp.lancs.ac.uk/computing/

users/geoff/Publications/IEESW00.pdf, Februar 2000, S. 13–21 148

Blair *et al.* 2000b BLAIR, Gordon S. ; BLAIR, Lynne ; ISSARNY, Valérie ; TUMA, Petr ; ZARRAS, Apostolos: The Role of Software Architecture in Constraining Adaptation in Component-based Middleware Platforms. In: *IFIP/ACM International Conference on Distributed systems platforms* Bd. 1795 der Reihe *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, 2000, S. 164–184. – OpenURL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=1795&spage=164. – auch beziehbar über ACM http://portal.acm.org/citation.cfm?id=338361&coll=ACM&dl=ACM&CFID=18423705&CFTOKEN=31181706##. – ISSN 0302-9743 149, 150

Blair *et al.* 2002 BLAIR, Gordon S. ; COULSON, Geoff ; BLAIR, Lynne ; DURAN-LIMON, Hector ; GRACE, Paul ; MOREIRA, Rui ; PARLAVANTZAS, Nikos: Reflection, self-awareness and self-healing in OpenORB. In: *Proceedings of the first workshop on Self-healing systems*, ACM Press, 2002, S. 9–14. – DOI http://doi.acm.org/10.1145/582128.582131. – ISBN 1581136099 14, 148, 150

Bobrow *et al.* 2003 BOBROW, Daniel G. ; GABRIEL, Richard P. ; WHITE, Jon L.: *CLOS in Context: The Shape of the Design Space*. 2003. – URL http://www.dreamsongs.com/NewFiles/clos-book.pdf. – Vermutlich elektronische Version des 1993 bei MIT Press erschienenen gleichnamigen Buchs 144

Boehm 1986 BOEHM, Barry: A spiral model of software development and enhancement. In: *SIGSOFT Software Engineering Notes* 11 (1986), Nr. 4, S. 14–24. – DOI http://doi.acm.org/10.1145/12944.12948. – ISSN 0163-5948 36

Bradburya *et al.* 2004 BRADBURYA, Jeremy S. ; CORDYA, James R. ; DINGELA, Juergen ; WERMELINGER, Michel: *A Classification of Formal Specifications for Dynamic Software Architectures*. 2004. – URL http://www.cs.queensu.ca/~cordy/Papers/Bradbury_FSE2004.pdf. – eingereicht zu *FSE-12 - ACM SIGSOFT 2004 12th International Symposium on the Foundations of Software Engineering* im April 2004, 10 Seiten 132

Bridges 2002 BRIDGES, Patrick G.: *Composing and Coordinating Adaptations in Cholla*, University of Arizona, Dissertation, 2002. – Zusand per Mail 164, 209

Brockhaus17 BROCKHAUS (Hrsg.): *Brockhaus-Enzyklopädie: in 20 Bänden*. 17. Auflage. Mannheim : Brockhaus. – ISBN 3-7653-0000-4 62

Brockhaus19 BROCKHAUS (Hrsg.): *Brockhaus-Enzyklopädie: in 24 Bänden*. 19. Auflage. Mannheim : Brockhaus. – ISBN 3-7653-1100-6 12, 22, 62, 72

Buschmann *et al.* 1998 BUSCHMANN, Frank ; MEUNIER, Regine ; ROHNERT, Hans ; SOMMERLAD, Peter ; STAL, Michael: *Pattern-orientierte Software-Architektur — Ein Pattern-System*. Addison-Wesley, 1998. – Deutsche Übersetzung von Christiane Löckenhoff. – ISBN 3-8273-1282-5 32, 53, 143, 144, 145, 157, 191, 194

Canal 2004 CANAL, Carlos: On the dynamic adaptation of component behaviour. In: *First International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT04)*, URL http://wcat04.unex.es/wcat04/papers/10_canal.pdf, 2004 102, 103, 200

Chen *et al.* 2001 CHEN, Wen-Ke ; HILTUNEN, Matti A. ; SCHLICHTING, Richard D.: Constructing Adaptive Software in Distributed Systems. In: *The 21st International Conference on Distributed Computing Systems (ICDS'2001)*, URL http://www.cs.arizona.edu/people/chwk/CactusAD.pdf, 2001, S. 635–643 98, 157, 162, 163, 165, 166, 207, 212

Cheng 2003 CHENG, Owen: *17-811 Self-Healing Systems: Class Discussion Summary*. 12. März 2003. – URL http://www-2.cs.cmu.edu/~garlan/17811/Summaries/03-12-Cheng.html 81

Cheng *et al.* 2002a CHENG, Shang-Wen ; GARLAN, David ; SCHMERL, Bradley ; SOUSA, João P. ; SPITZNAGEL, Bridget ; STEENKISTE, Peter: Using Architectural Style as a Basis for Self-repair, IEEE Computer Society, August 2002, S. 45–59. – URL http://www-2.cs.cmu.edu/afs/cs/project/able/ftp/wicsa3-arch/WICSA-web.pdf 126, 168

Cheng *et al.* 2002b CHENG, Shang-Wen ; GARLAN, David ; SCHMERL, Bradley ; SOUSA, João P. ; SPITZNAGEL, Bridget ; STEENKISTE, Peter ; HU, Ningning: Software Architecture-Based Adaptation for Pervasive Systems. In: *Trends in Network and Pervasive Computing - ARCS 2002 : International Conference on Architecture of Computing Systems, Karlsruhe, Germany, April 8–12, 2002* Bd. 2299 der Reihe *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, 2002, S. 67–82. – OpenURL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=2299&spage=67. – ISSN 0302-9743 xviii, 92, 93

Cheng *et al.* 2004 CHENG, Shang-Wen ; HUANG, An-Cheng ; GARLAN, David ; SCHMERL, Bradley ; STEENKISTE, Peter: *Rainbow: Architecture-based Self-healing with Reusable Infrastructure*. 19. Januar 2004. – URL http://www-2.cs.cmu.edu/afs/cs/project/able/ftp/icac/autonomic-web.pdf 167, 169, 181

Clements *et al.* 2002 CLEMENTS, Paul ; BACHMANN, Felix ; BASS, Len ; GARLAN, David ; IVERS, James ; LITTLE, Reed ; NORD, Robert ; STAFFORD, Judith: *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2002. – ISBN 201703726 5, 6

Cobleigh *et al.* 2002 COBLEIGH, Jamieson M. ; OSTERWEIL, Leon J. ; WISE, Alexander ; LERNER, Barbara S.: Containment Units: A Hierarchically Composable Architecture for Adaptive Systems. In: *Proceedings of the tenth ACM SIGSOFT symposium on Foundations of software engineering*, ACM Press, 2002, S. 159–165. – DOI http://doi.acm.org/10.1145/587051.587076. – ISBN 1-58113-514-9 96, 179

Coulouris *et al.* 2002 COULOURIS, George ; DOLLIMORE, Jean ; KINDBERG, Tim: *Verteilte Systeme — Konzepte und Design*. 3. Auflage. Addison-Wesley, 2002. – ISBN 3-8273-7022-1 12, 105, 147

Cuesta *et al.* 2001 CUESTA, Carlos E. ; DE LA FUENTE, Pablo ; BARRIO-SOLÓRZANO, Manuel: Dynamic coordination architecture through the use of reflection. In: *Proceedings of the 2001 ACM symposium on Applied computing*, ACM Press, 2001, S. 134–140. – DOI http://doi.acm.org/10.1145/372202.372298. – ISBN 1-58113-287-5 132

Czarnecki und Eisenecker 2000 CZARNECKI, Krzysztof ; EISENECKER, Ulrich W.: *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000. – ISBN 0-201-30977-7 15, 27, 33, 34, 43, 44, 45, 46, 59, 143, 145, 146

Dashofy *et al.* 2001 DASHOFY, Eric M. ; HOEK, André van der ; TAYLOR, Richard N.: A Highly-Extensible, XML-Based Architecture Description Language. In: *Proceedings of The Working IEEE/IFIP Conference on Software Architecture*, IEEE Computer Society Press, 2001, S. 103–112. – URL http://www.ics.uci.edu/~edashofy/papers/wicsa2001.pdf 127

Dashofy *et al.* 2002 DASHOFY, Eric M. ; VAN DER HOEK, André ; TAYLOR, Richard N.: Towards architecture-based self-healing systems. In: *Proceedings of the first workshop on Self-healing systems*, ACM Press, 2002, S. 21–26. – DOI http://doi.acm.org/10.1145/582128.582133. – ISBN 1581136099 125, 127, 170, 207, 208

David und Ledoux 2003 DAVID, Pierre-Charles ; LEDOUX, Thomas: Towards a Framework for Self-Adaptive Component-Based Applications. In: STEFANI, Jean-Bernard (Hrsg.) ; DEMEURE, Isabelle (Hrsg.) ; HAGIMONT, Daniel (Hrsg.): *Proceedings of Distributed Applications and Interoperable Systems 2003, the 4th IFIP WG6.1 International Conference, DAIS 2003* Bd. 2893 der Reihe *Lecture Notes in Computer Science (LNCS)*. Paris : Springer-Verlag, November 2003, S. 1–14. – OpenURL http://www.springerlink.com/openurl.asp?

genre=article&issn=0302-9743&volume=2893&spage=1 124, 175, 214, 215, 216

de Lemos und Fiadeiro 2002 DE LEMOS, Rogério ; FIADEIRO, José L.: An architectural support for self-adaptive software for treating faults. In: *Proceedings of the first workshop on Self-healing systems*, ACM Press, 2002, S. 39–42. – DOI http://doi.acm.org/10.1145/582128.582136. – ISBN 1581136099 14, 94

Dey 2000 DEY, Anind K.: *Providing Architectural Support for Building Context-Aware Applications*, Georgia Institute of Technology, Dissertation, November 2000. – URL http://www.cc.gatech.edu/fce/ctk/pubs/dey-thesis.pdf 37, 38, 39, 40, 42, 47, 48, 49, 52, 224

Dey 2004a DEY, Anind K.: *Re: Interpretation of your context definition*. 30. August 2004. – E-Mail an den Autor – Message-ID <8351028360c1.8360c1835102@EECS.Berkeley.EDU> 48, 49

Dey 2004b DEY, Anind K.: *Re: Interpretation of your context definition*. 2. September 2004. – E-Mail an den Autor – Message-ID <8e09b68df307.8df3078e09b6@EECS.Berkeley.EDU> 51

Dey 2004c DEY, Anind K.: *Re: Interpretation of your context definition*. 3. September 2004. – E-Mail an den Autor – Message-ID <8fda718fdc7f.8fdc7f8fda71@EECS.Berkeley.EDU> 51

Dingler 2004 DINGLER, Thomas: Adaptive Verteilte Systeme / DaimlerChrysler, RIC/SA. 18. Januar 2004. – Forschungsbericht 16, 25, 77

Dittert 2004 DITTERT, Kerstin: Softwarearchitektur: Mythen und Legenden. In: *OBJEKTspektrum* (2004), Nr. 3, S. 34–39. – ISSN 0945-0491 53, 100

Dowling 2003 DOWLING, Jim: Coordinating Self-Adaptive Components for Emergent Distributed System Behaviour and Structure. In: *8th CaberNet Radicals Workshop*, URL http://www.newcastle.research.ec.org/cabernet/workshops/radicals/2003/papers/dowling_tcd_self_organising.pdf, 2003 204

Dowling und Cahill 2001a DOWLING, Jim ; CAHILL, Vinny: Dynamic Software Evolution and the K-Component Model / Trinity College Dublin. URL http://www.cs.tcd.ie/publications/tech-reports/reports.01/TCD-CS-2001-51.pdf, 2001 (TCD-CS-2001-51). – Forschungsbericht 150, 152

Dowling und Cahill 2001b DOWLING, Jim ; CAHILL, Vinny: The K-Component Architecture Meta-model for Self-Adaptive Software. In: *Metalevel Architectures and Separation of Crosscutting Con-*

cerns : Third International Conference, REFLECTION 2001, Kyoto, Japan, September 25–28, 2001 Bd. 2192 der Reihe *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, 2001, S. 81–88. – OpenURL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=2192&spage=81. – ISSN 0302-9743 150, 151

Duden 2001 WERMKE, M. (Hrsg.) ; KLOSA, A. (Hrsg.) ; KUNKEL-RAZUM, K. (Hrsg.) ; SCHOLZE-STUBENRECHT, W. (Hrsg.): *Duden: Die deutsche Rechtschreibung*. Bd. 1. 22. Auflage. Mannheim, Dudenverlag, 2001 144

Duden-Oxford 1990 DUDENREDAKTION UND OXFORD UNIVERSITY PRESS (Hrsg.): *Duden-Oxford Großwörterbuch Englisch*. Dudenverlag, 1990. – ISBN 3-411-02075-X 19, 20

Endler 1994 ENDLER, Markus: A Language for Implementing Generic Dynamic Reconfigurations of Distributed Programs. In: *Proceedings of the 12th Brazilian Symposium on Computer Networks*, URL http://www-di.inf.puc-rio.br/~endler/paperlinks/sbrc94.ps, 1994, S. 175–187 81

Eracar und Kokar 2000 ERACAR, Yönet A. ; KOKAR, Mieczyslaw M.: An Architecture for Software that Adapts to Changes in Requirements. In: *Journal of Systems and Software* 50 (2000), Nr. 3, S. 209–219. – DOI http://dx.doi.org/10.1016/S0164-1212(99)00098-9. – URL http://citeseer.ist.psu.edu/eracar98architecture.html. – ISSN 0164-1212 66, 67, 133, 135

Falcarin und Alonso 2004 FALCARIN, Paolo ; ALONSO, Gustavo: Software Architecture Evolution through Dynamic AOP. In: *First European Workshop on Software Architecture (EWSA 2004), St Andrews, Scotland, 21-22 May 04*, URL http://www.mics.ch/getDoc.php?docid=832&docnum=1, 2004 175

Fiadeiro 2004 FIADEIRO, José L.: *CommUnity*. 2004. – URL http://www.fiadeiro.org/jose/CommUnity/ 130

Fiadeiro und Maibaum 1995 FIADEIRO, José L. ; MAIBAUM, Tom: Interconnecting Formalisms: Supporting Modularity, Reuse and Incrementality. In: *Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, ACM Press, 1995, S. 72–80. – DOI http://doi.acm.org/10.1145/222124.222141. – ISBN 0-89791-716-2 9

Filman *et al.* 2002 FILMAN, Robert E. ; BARRETT, Stuart ; LEE, Diana D. ; LINDEN, Ted: Inserting Ilities by Controlling Communications. In: *Communications of the ACM* 45 (2002), Nr. 1, S. 116–122. – DOI http://doi.acm.org/10.1145/502269.502274. – ISSN 0001-0782 32

Foster *et al.* 2002 FOSTER, Ian ; KESSELMAN, Carl ; NICK, Jeffrey M. ; TUECKE, Steven: *The Physiology of the Grid - An Open Grid Services Architecture for Distributed Systems Integration*. Juni 2002. – URL www.globus.org/research/papers/ogsa.pdf 172, 198

Franke 2001 FRANKE, Gregor: *Möglichkeiten der Adaption des mobilen Webzugriffs an die Eigenschaften der Ausführungsumgebung*, Institut für Systemarchitektur,Fakultät Informatik,Technische Universität Dresden, Großer Beleg, 31. Oktober 2001 203

Fritsch und Renz 2004 FRITSCH, Claudia ; RENZ, Burkhardt: Four Mechanisms for Adaptable Systems: A Meta-Level Approach to Building a Software Product Line. In: *Software Product Lines: Third International Conference, SPLC 2004, Boston, MA, USA, August 30–September 2, 2004* Bd. 3154 der Reihe *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, 2004, S. 51–71. – OpenURL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=3154&spage=51. – DOI 10.1007/b100081. – ISBN 3-540-22918-3 143

Fuggetta *et al.* 1998 FUGGETTA, Alfonso ; PICCO, Gian P. ; VIGNA, Giovanni: Understanding Code Mobility. In: *IEEE Transactions on Software Engineering* 24 (1998), Nr. 5, S. 342–361. – DOI http://dx.doi.org/10.1109/32.685258. – URL https://www.cis.strath.ac.uk/teaching/ug/classes/52.477/e0342.pdf. – ISSN 0098-5589 105, 106, 210

Gajos *et al.* 2003 GAJOS, Krzysztof ; WEISMAN, Luke ; SHROBE, Howard: Design Principles for Resource Management Systems for Intelligent Spaces. In: *Self-Adaptive Software — Applications: Second International Workshop (IWSAS 2001)* Bd. 2614 der Reihe *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, 2003, S. 198–215. – OpenURL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=2614&spage=198. – ISSN 0302-9743 45

Garlan *et al.* 1997 GARLAN, David ; MONROE, Robert ; WILE, David: Acme: an architecture description interchange language. In: *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, IBM Press, 1997, S. 7. – URL http://portal.acm.org/citation.cfm?id=782017&dl=ACM&coll=portal## 9, 10, 11

Garlan und Schmerl 2002 GARLAN, David ; SCHMERL, Bradley: Model-based adaptation for self-healing systems. In: *Proceedings of the first workshop on Self-healing systems*, ACM Press, 2002, S. 27–32. – DOI http://doi.acm.org/10.1145/582128.582134. – ISBN 1581136099 124, 125, 167, 181, 201

Garland und Anthony 2003 GARLAND, Jeff ; ANTHONY, Richard: *Large-Scale Software Architecture: A Practical Guide using UML*. John Wiley & Sons, Ltd, 2003. – ISBN 0-470-84849-9 5, 63

Gelernter und Carriero 1992 GELERNTER, David ; CARRIERO, Nicholas: Coordination languages and their significance. In: *Communications of the ACM* 35 (1992), Nr. 2, S. 97–107. – DOI http://doi.acm.org/10.1145/129630.129635. – ISSN 0001-0782 129

Georgiadis *et al.* 2002 GEORGIADIS, Ioannis ; MAGEE, Jeff ; KRAMER, Jeff: Self-organising software architectures for distributed systems. In: *Proceedings of the first workshop on Self-healing systems*, ACM Press, 2002, S. 33–38. – DOI http://doi.acm.org/10.1145/582128.582135. – ISBN 1581136099 130

Ghezii *et al.* 1991 GHEZII, Carlo ; JAZAYERI, Mehdi ; MANDRIOLI, Dino: *Fundamentals of Software Engineering*. Prentice Hall, 1991 82

GoF 1996 GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John (GANG OF FOUR): *Entwurfsmuster — Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley, 1996. – Deutsche Übersetzung von Dirk Riehle. – ISBN 0-201-63361-2 26, 156, 157, 191, 194

Gorlick und Razouk 1991 GORLICK, Michael M. ; RAZOUK, Rami R.: Using weaves for software construction and analysis. In: *Proceedings of the 13th international conference on Software engineering*, IEEE Computer Society, 1991, S. 23–34. – URL http://portal.acm.org/citation.cfm?id=256677&dl=ACM&coll=portal&CFID=11111111&CFTOKEN=2222222. – ISBN 0-89791-391-4 127, 168

Goslar 2001 GOSLAR, Kevin: *Gewinnung, Darstellung und Verwaltung von Kontextinformationen – Personalisierung*, Lehrstühle für Rechnernetze sowie Wirtschaftsinformatik, TU Dresden, Diplomarbeit, 2001 37, 38, 42, 48

Goudarzi 1999 GOUDARZI, Kaveh M.: *Consistency Preserving Dynamic Reconfiguration of Distributed Systems*, Imperial College London, Dissertation, März 1999. – URL http://www-dse.doc.ic.ac.uk/~km2/phd/thesis.ps.gz 81, 121, 128, 209

Goudarzi und Kramer 1996 GOUDARZI, Kaveh M. ; KRAMER, Jeff: Maintaining Node Consistency in the Face of Dynamic Change. In: *Proceedings of the 3rd International Conference on Configurable Distributed Systems*, IEEE Computer Society, 1996, S. 62–69. – ISBN 0-8186-7395-8 129

Goulão und e Abreu 2003 GOULÃO, Miguel ; E ABREU, Fernando B.: Bridging the gap between Acme and UML 2.0 for CBD. In: *Specification and Verification of Component-Based Systems (SAVCBS'2003), Helsinki, Finland*, URL http://www.cs.iastate.edu/~leavens/SAVCBS/2003/papers/posters/goulao-brito-e-abreu.pdf, September 2003 127

Griffel 1998 GRIFFEL, Frank: *Componentware — Konzepte und Techniken eines Softwareparadigmas*. dpunkt.verlag, 1998. – ISBN 3-932588-02-9 8

Gross *et al.* 2001 GROSS, Philip N. ; GUPTA, Suhit ; KAISER, Gail E. ; KC, Gaurav S. ; PAREKH, Janak J.: An Active Events Model for Systems Monitoring. In: *Working Conference on Complex and Dynamic Systems Architecture*, URL http://www.psl.cs.columbia.edu/ftp/psl/CUCS-011-01.pdf, 2001 192, 194

Göbel *et al.* 2001 GÖBEL, Steffen ; BUCHHOLZ, Sven ; ZIEGERT, Thomas ; SCHILL, Alexander: Device Independent Representation of Web-based Dialogs and Contents. In: *Proceedings of the IEEE Youth Forum in Computer Science and Engineering (YUFORIC ´01), Valencia, Spain*, URL http://www.rn.inf.tu-dresden.de/scripts_lsrn/veroeffent_print/YUFORIC2001.pdf, November 2001 54

Hasenbein und Schreiber 2002 HASENBEIN, Herbert ; SCHREIBER, Cecilia: Online-Übersetzer: Englisch-Wörterbücher im Netz. In: *c't — magazin für computer technik* (2002), Nr. 13, S. 170–175. – ISSN 0724-8679 22

Heineman 1998 HEINEMAN, George T.: Adaptation and software architecture. In: *Proceedings of the third international workshop on Software architecture*, ACM Press, 1998, S. 61–64. – DOI http://doi.acm.org/10.1145/288408.288424. – ISBN 1-58113-081-3 63, 79

Heise 2003 HEISE NEWSTICKER: *Stabiler Datentransfer über wechselnde Funknetze*. 3. Februar 2003. – URL http://www.heise.de/newsticker/meldung/34208 94

Hiltunen 2004 HILTUNEN, Matti A.: *Re: Status of your Adaptive Software based on Cactus*. 19. Juli 2004. – E-Mail an den Autor – Message-ID <40FC1C31.4000405@research.att.com> 164

Hinnelund 2004 HINNELUND, Patrick: *Autonomic Computing — a method for automated systems management*, Royal Institute of Technology, Master's Thesis, März 2004. – URL http://www.student.nada.kth.se/~d99-phi/mthesis.pdf 133, 135, 142

Hitz und Kappel 2003 HITZ, Martin ; KAPPEL, Gerti: *UML@Work*. 2. Auflage. dpunkt.verlag, November 2003. – ISBN 3-89864-194-5 181, 183

Hitz *et al.* 2002 HITZ, Martin ; KAPPEL, Gerti ; RETSCHITZEGGER, Werner ; SCHWINGER, Wieland: Ein UML-basiertes Framework zur Modellierung ubiquitärer Web-Anwendungen. In: *Wirtschaftsinformatik* 44 (2002), Nr. 3, S. 225–235. – URL http://www.schwinger.at/LIB/WIN2002.pdf 50

Hochmüller *et al.* 2003 HOCHMÜLLER, Elke ; MITTERMEIR, Roland ; POZEWAUNIG, Heinz: *Architektur: Stellenwerte, Ziele, Inhalt, Grundformen*. 2003. – URL http://www.isys.uni-klu.ac.at/ISYS/Courses/03WS/sete/bemerkungen/Bs03 5

Hofmeister *et al.* 2000 HOFMEISTER, Christine ; NORD, Robert ; SONI, Dilip: *Applied Software Architecture*. Addison-Wesley, 2000. – ISBN 201325713 5

Holder *et al.* 1999a HOLDER, Ophir ; BEN-SHAUL, Israel ; GAZIT, Hovav: Dynamic layout of distributed applications in FarGo. In: *Proceedings of the 21st international conference on Software engineering (ICSE'99)*, IEEE Computer Society, 1999, S. 163–173. – URL http://dsg.technion.ac.il/publications/icse99.ps. – ISBN 1-58113-074-0 157

Holder *et al.* 1999b HOLDER, Ophir ; BEN-SHAUL, Israel ; GAZIT, Hovav: System Support for Dynamic Layout of Distributed Applications. In: *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS'99)*, IEEE Computer Society, 1999, S. 403–411. – DOI http://doi.ieeecomputersociety.org/10.1109/ICDCS.1999.776542. – URL http://dsg.technion.ac.il/publications/icdcs99.ps 156

Horn 2001 HORN, Paul: *Autonomic Computing: IBM's Perspective on the State of Information Technology*. Oktober 2001. – URL http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf. – Manifest der Autonomic Computing Initiative 13, 15, 104

Hübsch *et al.* 2003 HÜBSCH, Gerald ; SPRINGER, Thomas ; SCHILL, Alexander ; SPRIESTERSBACH, Axel ; ZIEGERT, Thomas: Systemlösungen für die Entwicklung adaptiver Anwendungen für mobile und ubiquitäre Infrastrukturen. In: *HMD: Praxis der Wirtschaftsinformatik* 229 (2003), Februar, S. 42–55 30, 54, 73

Hürsch und Lopes 1995 HÜRSCH, Walter ; LOPES, Cristina V.: Separation of Concerns / Northeastern University, Boston, MA. URL ftp://ftp.ccs.neu.edu/pub/people/crista/publications/techrep95/separation.pdf, 1995 (NU-CCS-95-03). – Forschungsbericht 124, 149, 164

IBM 2003a IBM: *Autonomic Computing: Die Vision wird Wirklichkeit*. 16. Januar 2003. – URL http://www-5.ibm.com/de/promotions/autonomic/autonomic_computing.pdf 13

IBM 2003b IBM: *IBM Systems Journal, Vol. 42, Nr. 1, 2003, Special Issue on Autonomic Computing*. 2003. – URL http://researchweb.watson.ibm.com/journal/sj42-1.html 173

IBM 2004 IBM: *IBM Autonomic Computing Product and Services*. 2004. – URL http://www-3.ibm.com/autonomic/products.shtml 173

IEEE 1471-2000 IEEE ARCHITECTURE WORKING GROUP (AWG): *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems (IEEE Std 1471)*. – URL http://www.pithecanthropus.com/~awg/public_html/. – Website der AWG 3, 4, 8, 63

Inverardi und Wolf 1995 INVERARDI, Paola ; WOLF, Alexander L.: Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model. In: *IEEE Transactions on Software Engineering* 21 (1995), April, Nr. 4, S. 373–386. – DOI http://dx.doi.org/10.1109/32.385973. – ISSN 0098-5589 131

Jeckle *et al.* 2003 JECKLE, Mario ; RUPP, Chris ; HAHN, Jürgen ; ZENGLER, Barbara ; QUEINS, Stefan: *UML 2 glasklar*. Carl Hanser Verlag, November 2003. – ISBN 3-4462-2575-7 92, 126, 181

Jennings 2000 JENNINGS, Nicholas R.: On Agent-based Software Engineering. In: *Artif. Intell.* 117 (2000), Nr. 2, S. 277–296. – DOI http://dx.doi.org/10.1016/S0004-3702(99)00107-1. – ISSN 0004-3702 173

Jini O. V.: *Jini.org — Domain home page*. – URL http://www.jini.org/ 197, 200

JXTA O. V.: *jxta.org*. – URL http://www.jxta.org 200

Kadner 2003 KADNER, Kay: *Konzeption eines verteilten Dienstes zur Unterstützung von Context-Awareness*, Technische Universität Dresden, Großer Beleg, 19. Dezember 2003 48

Kadner 2004a KADNER, Kay: *Erweitung eines verteilten Kontextdienstes um Konzepte zur Unterstützung von ad hoc Szenarien*, Technische Universität Dresden, Diplomarbeit, 31. August 2004 38, 42, 48

Kadner 2004b KADNER, Kay: *Re: Kontext bla von mir beurteilen*. 18. August 2004. – E-Mail an den Autor – Message-ID <46735.192.35.17.21.1092831123.squirrel@www.inf.tu-dresden.de> 51

Kadner 2004c KADNER, Kay: *Re: Kontext bla von mir beurteilen*. 24. August 2004. – E-Mail an den Autor – Message-ID <36352.192.35.17.21.1093347059.squirrel@192.35.17.21> 51

Kaiser *et al.* 1999 KAISER, Gail ; VALETTO, Giuseppe ; KC, Gaurav S.: A Mobile Agent Approach to Lightweight Process Workflow. In: *Proceedings of the International Process Technology Workshop, Villard de Lans, France*, URL http://www.psl.cs.columbia.edu/ftp/psl/CUCS-021-99.pdf, 1.–3. September 1999 206, 209

Kandé *et al.* 2002 KANDÉ, Mohamed M. ; CRETTAZ, Valentin ; STROHMEIER, Alfred ; SENDALL, Shane: Bridging the gap between IEEE 1471, an architecture description language, and UML. In: *Software and Systems Modeling* 1 (2002), Dezember, Nr. 2, S. 113–129. – OpenURL http://www.springerlink.com/openurl.asp?genre=article&issn=1619-1366&volume=1&issue=2&spage=113. – ISSN 1619-1366 127

Karsai *et al.* 2003 KARSAI, Gabor ; LEDECZI, Akos ; SZTIPANOVITS, Janos ; PECELI, Gabor ; SIMON, Gyula ; KOVACSHAZY, Tamas: An Approach to Self-adaptive Software Based on Supervisory Control. In: *Self-Adaptive Software — Applications: Second International Workshop (IWSAS 2001)* Bd. 2614 der Reihe *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, 2003, S. 24–38. – OpenURL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=2614&spage=24. – ISSN 0302-9743 133, 142

Karsai und Sztipanovits 1999 KARSAI, Gabor ; SZTIPANOVITS, Janos: A Model-Based Approach to Self-Adaptive Software. In: *IEEE Intelligent Systems* 14 (1999), Nr. 3, S. 46–52. – DOI http://doi.ieeecomputersociety.org/10.1109/5254.769884. – ISSN 1094-7167 59, 133, 143, 146

Kasten und McKinley 2004 KASTEN, Eric P. ; MCKINLEY, Philip K.: Perimorph: Run-Time Composition and State Management for Adaptive Systems. In: *Proceedings of the 24th International Conference on Distributed Computing Systems Workshops - W7: EC (ICDCSW'04)*, IEEE Computer Society, 2004, S. 332–337. – URL http://dares.enst-bretagne.fr/dares2004/Session2/paper7.pdf. – ISBN 0-7695-2087-1 98, 157, 207

Keffart und Chess 2003 KEFFART, Jeffrey O. ; CHESS, David M.: The Vision of Autonomic Computing. In: *Computer Magazine* 36 (2003), Nr. 1, S. 41–50. – URL http://www.research.ibm.com/autonomic/research/papers/AC_Vision_Computer_Jan_2003.pdf 14, 15, 170, 171, 172, 204, 212

Ketfi *et al.* 2002 KETFI, Abdelmadjid ; BELKHATIR, Noureddine ; CUNIN, Pierre-Yves: Automatic Adaptation of Component-based Software: Issues and Experiences. In: *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA '02, June 24–27, 2002, Las Vegas, Nevada, USA, Volume 3*, CSREA Press, 2002, S. 1365–1371. – URL http://www-adele.imag.fr/Les.Publications/intConferences/PDPTA2002Ket.pdf. – ISBN 1-892512-89-0 82, 98, 100

Kiczales *et al.* 1997 KICZALES, Gregor ; LAMPING, John ; MENDHEKAR, Anurag ; MAEDA, Chris ; LOPES, Cristina V. ; LOINGTIER, Jean-Marc ; IRWIN, John: Aspect-Oriented Programming. In: *ECOOP '97 - Object-Oriented Programming: 11th European Conference, Jyväskylä, Finnland* Bd. 1241 der Reihe *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, 1997, S. 220–242. – OpenURL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=1241&spage=220. – ISSN 0302-9743 37, 175

Klamar 2004 KLAMAR, Sebastian: *Komponentenbasierte Architektur zur Adaption web-basierter Anwendungen*, Technische Universität Dresden, Großer Beleg, 28. Januar 2004 89, 90

Kokar *et al.* 1999 KOKAR, Mieczyslaw M. ; BACLAWSKI, Kenneth ; ERACAR, Yonet A.: Control Theory-Based Foundations of Self-Controlling Software. In: *IEEE Intelligent Systems* 14 (1999), Nr. 3, S. 37–45. – DOI http://doi.ieeecomputersociety.org/10.1109/5254.769883. – ISSN 1094-7167 94, 99, 133, 134, 137, 138, 139, 140, 141, 161, 195, 198

Kon *et al.* 2002 KON, Fabio ; COSTA, Fabio ; BLAIR, Gordon ; CAMPBELL, Roy H.: The Case for Reflective Middleware. In: *Communications of the ACM* 45 (2002), Nr. 6, S. 33–38. – DOI http://doi.acm.org/10.1145/508448.508470. – ISSN 0001-0782 144, 147, 148, 149, 175

Kon *et al.* 2000 KON, Fabio ; ROMÁN, Manuel ; LIU, Ping ; MAO, Jina ; YAMANE, Tomonori ; MAGALHÃES, Luiz C. ; CAMPBELL, Roy H.: Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In: *Middleware 2000: IFIP/ACM International Conference on Distributed Systems Platforms, New York, NY, USA, April 2000* Bd. 1795 der Reihe *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, 2000, S. 121–143. – OpenURL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=1795&spage=121. – ISSN 0302-9743 147

Kramer und Magee 1985 KRAMER, Jeff ; MAGEE, Jeff: Dynamic configuration for distributed systems. In: *IEEE Transactions on Software Engineering* 11 (1985), Nr. 4, S. 424–436. – ISSN 0098-5589 126

Kramer und Magee 1990 KRAMER, Jeff ; MAGEE, Jeff: The Evolving Philosophers Problem: Dynamic Change Management. In: *IEEE Transactions on Software Engineering* 16 (1990), November, Nr. 11, S. 1293–1306. – DOI http://dx.doi.org/10.1109/32.60317. – ISSN 0098-5589 129

Kramer und Magee 1998 KRAMER, Jeff ; MAGEE, Jeff: Analysing dynamic change in distributed software architectures. In: *IEE Proceedings - Software* 145 (1998), Oktober, Nr. 5, S. 146–154. – URL http://www.win.tue.nl/~hmei/SoftwareUpdate/Analysing%20dynamic%20change%20in%20distributed%20software%20architectures.pdf 124, 153

Kreger 2001 KREGER, Heather: *Web Services Conceptual Architecture*. Mai 2001. – URL http://www-3.ibm.com/software/solutions/webservices/pdf/WSCA.pdf 172, 197

Kruchten 1995 KRUCHTEN, Phillip: Architectural Blueprints — The »4+1« View Model of Software Architecture. In: *IEEE Software* 12 (1995), November, Nr. 6, S. 42–50. – URL http://www.rational.com/media/whitepapers/Pbk4p1.pdf 5, 8

Laddaga 1997 LADDAGA, Robert: *Self-Adaptive Software*. Dezember 1997. – URL http://www.darpa.mil/ito/Solicitations/CBD_9812.html. – DARPA SOL BAA 98-12, Original-URL nur noch zu erreichen über den Web-Archivierungsdienst *archive.org*: http://web.archive.org/web/19990221110757/http://www.darpa.mil/ito/Solicitations/CBD_9812.html 28, 29, 34, 58

Laddaga 1999 LADDAGA, Robert: Creating Robust Software through Self-Adaptation. In: *IEEE Intelligent Systems* 14 (1999), Nr. 3, S. 26–29. – DOI http://doi.ieeecs.org/10.1109/MIS.1999.769879. – ISSN 1094-7167 34, 58, 132, 143, 195

Laddaga 2001 LADDAGA, Robert: Active Software. In: *Self-Adaptive Software: First International Workshop, IWSAS 2000, Oxford, UK, April 2000* Bd. 1936 der Reihe *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, 2001, S. 11–26. – OpenURL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=1936&spage=11. – ISSN 0302-9743 27, 29, 58, 104, 185

Laddaga und Robertson 2004 LADDAGA, Robert ; ROBERTSON, Paul: Self Adaptive Software: A Position Paper. In: *SELF-STAR:*

International Workshop on Self- Properties in Complex Information Systems, 31 May–2 June 2004, University of Bologna*, URL http://www.cs.unibo.it/self-star/papers/laddaga.pdf, 2004 142

Laddaga *et al.* 2001 LADDAGA, Robert ; ROBERTSON, Paul ; SHROBE, Howie: Results of the First International Workshop on Self-Adaptive Software. In: *Self-Adaptive Software: First International Workshop, IWSAS 2000, Oxford, UK, April 2000* Bd. 1936 der Reihe *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, 2001, S. 242–247. – OpenURL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=1936&spage=242. – ISSN 0302-9743 142

Laddaga *et al.* 2003 LADDAGA, Robert ; ROBERTSON, Paul ; SHROBE, Howie: Results of the Second International Workshop on Self-Adaptive Software. In: *Self-Adaptive Software — Applications: Second International Workshop (IWSAS 2001)* Bd. 2614 der Reihe *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, 2003, S. 281–290. – OpenURL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=2614&spage=281. – ISSN 0302-9743 157, 158, 161

Landau *et al.* 1997 LANDAU, Ioan D. ; LOZANO, Rogelio ; M'SAAD, Mohammed: *Adaptive Control*. Springer-Verlag, 1997. – ISBN 3-540-76187-X 137

Landauer und Bellman 2003 LANDAUER, Christopher ; BELLMAN, Kirstie L.: Self-modeling Systems. In: *Self-Adaptive Software — Applications: Second International Workshop (IWSAS 2001)* Bd. 2614 der Reihe *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, 2003, S. 238–256. – OpenURL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=2614&spage=238. – ISBN 3-540-00731-8 143

Langdon und Poli 2002 LANGDON, William B. ; POLI, Riccardo: *Foundations of genetic programming*. Springer-Verlag, 2002. – ISBN 3-540-42451-2 56

Lange und Oshima 1999 LANGE, Danny B. ; OSHIMA, Mitsuru: Seven Good Reasons for Mobile Agents. In: *Communications of the ACM* 42 (1999), Nr. 3, S. 88–89. – DOI http://doi.acm.org/10.1145/295685.298136. – ISSN 0001-0782 106

Langenscheidt 1989 LANGENSCHEIDT (Hrsg.): *Langenscheidts Fremdwörterbuch*. Langenscheidt, 1989. – ISBN 3468203969 22

Larman 2002 LARMAN, Craig: *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall, 2002. – ISBN 0-13-092569-1 134

Le Métayer 1998 LE MÉTAYER, Daniel: Describing Software Architecture Styles Using Graph Grammars. In: *IEEE Transactions on Software Engineering* 24 (1998), Nr. 7, S. 521–533. – DOI http://dx.doi.org/10.1109/32.708567. – ISSN 0098-5589 9, 126, 131

Lemlouma und Layaïda 2002 LEMLOUMA, Tayeb ; LAYAÏDA, Nabil: Content Adaptation and Generation Principles for Heterogeneous Clients. In: *W3C Workshop on Device Independent Authoring Techniques, 25–26 September 2002, SAP University, St. Leon-Rot, Germany*, URL http://opera.inrialpes.fr/people/Tayeb.Lemlouma/Papers/DIAT-PositionPaper.pdf, 2002 30, 31, 73

Lewandowski 2003 LEWANDOWSKI, Rudolf: *Software-Architektur I.* 2003. – URL http://www.objectarchitects.de/tu2002/slides2003/(04)Architektur1.pdf. – Vorlesungsskript zur gleichnamigen Vorlesung an der TU Wien 32, 96

Li *et al.* 2004 LI, Z. ; LIU, H. ; PARASHAR, M.: Enabling Autonomic, Self-managing Grid Applications. In: *SELF-STAR: International Workshop on Self-* Properties in Complex Information Systems, 31 May–2 June 2004, University of Bologna*, URL http://www.cs.unibo.it/self-star/papers/parashar.pdf, 2004 173

Lieberherr 1995 LIEBERHERR, Karl: *Adaptive Object-Oriented Software: The Demeter Method.* PWS Publishing Company, 1995. – URL http://www.ccs.neu.edu/research/demeter/book/book-download.html. – ISBN 0-534-94602-X 27, 32, 36, 44, 56

Lieberherr 1998 LIEBERHERR, Karl: *Adaptive Software: 93/94 Research Report.* 29. Oktober 1998. – URL http://www.ccs.neu.edu/research/demeter/aop/history/93-94-report/ 35, 36, 37, 39, 43, 44, 45, 46, 55, 56, 69, 80, 83

Lieberherr und Lopes 1995 LIEBERHERR, Karl ; LOPES, Cristina V.: Workshop on Adaptable and Adaptive Software, URL http://www.ccs.neu.edu/research/demeter/adaptable-systems/report.ps, 1995, S. 149–154. – Addendum to the Proceedings of the 10th Annual Conference on Object-Oriented Programming Systems, Languages and Applications, Oct. 15–19, 1995, Austin, TX, USA 27, 33, 224

Lieberman und Selker 2004 LIEBERMAN, Henry ; SELKER, Ted: *Out of Context: Computer Systems That Adapt to, and Learn From, Context.* 2004. – URL http://lieber.www.media.mit.edu/people/lieber/Teaching/Context/Out-of-Context-Paper/Out-of-Context.html. – Media Laboratory, Massachusetts Institute of Technology 38

Lopes *et al.* 2002 LOPES, Antónia ; FIADEIRO, José L. ; WERMELINGER, Michel: Architectural primitives for distribution and mobility. In: *SIGSOFT Software Engineering Notes* 27 (2002), Nr. 6, S. 41–50. – DOI http://doi.acm.org/10.1145/605466.605473. – ISSN 0163-5948 130

Loques *et al.* 2000 LOQUES, Orlando ; SZTAJNBERG, Alexandre ; LEITE, Julius ; LOBOSCO, Marcelo: On the Integration of Configuration and Meta-level Programming Approaches. In: *Reflection and Software Engineering* Bd. 1826 der Reihe *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, 2000, S. 189–208. – OpenURL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=1826&spage=189. – ISSN 0302-9743 156

Loyall *et al.* 1998 LOYALL, Joseph P. ; BAKKEN, David E. ; SCHANTZ, Richard E. ; ZINKY, John A. ; KARR, David A. ; VANEGAS, Rodrigo ; ANDERSON, Kenneth R.: QoS aspect languages and their runtime integration. In: *Languages, Compilers, and Run-Time Systems for Scalable Computers: 4th International Workshop, LCR'98, Pittsburgh, PA, USA, May 1998* Bd. 1511 der Reihe *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, 1998, S. 303–318. – OpenURL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=1511&spage=303. – Online in SpringerLink seit: Juni 2003. – ISSN 0302-9743 132

Luckham und Vera 1995 LUCKHAM, D. ; VERA, J.: An Event-Based Architectural Definition Language. In: *IEEE Transactions on Software Engineering* 21 (1995), Nr. 9, S. 717–734 9, 127, 129

Lynch und Vaandrager 1992 LYNCH, Nancy ; VAANDRAGER, Frits: Forward and backward simulations for timing-based systems. In: *Proceedings of Real-Time: Theory in Practice* Bd. 600 der Reihe *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, 1992, S. 397–446. – URL http://theory.lcs.mit.edu/tds/papers/Lynch/REX91.ps.gz. – ISSN 0302-9743 150, 202

Löschau 2002 LÖSCHAU, Frank: *Realisierung kontextsensitiver Anwendungen auf Basis eines Frameworks zur Kontextverarbeitung und -verwaltung*, Lehrstühle für Rechnernetze sowie Wirtschaftsinformatik, TU Dresden, Diplomarbeit, 9. Oktober 2002 39, 42, 48, 193

Maes 1987 MAES, Pattie: Concepts and Experiments in Computational Reflection. In: *Conference proceedings on Object-oriented programming systems, languages and applications*, ACM Press, 1987, S. 147–155. – DOI http://doi.acm.org/10.1145/38765.38821. – Paper zur gleichnamigen Dissertation. – ISBN 0-89791-247-0 144

Magee und Kramer 1996 MAGEE, Jeff ; KRAMER, Jeff: Dynamic structure in software architectures. In: *Proceedings of the 4th*

ACM SIGSOFT symposium on Foundations of software engineering, ACM Press, 1996, S. 3–14. – DOI http://doi.acm.org/10.1145/239098.239104. – ISBN 0-89791-797-9 127, 129

Manna und Pnueli 1992 MANNA, Zohar ; PNUELI, Amir: *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., 1992. – ISBN 0-387-97664-7 150

McGurren und Conroy 2002 MCGURREN, Finnbar ; CONROY, Damien: *X-Adapt: An Architecture for Dynamic Systems*. Juni 2002. – URL http://joint.org/use2002/sub/late/mcGurren-x-Adapt.pdf 143

Medvidovic 1996 MEDVIDOVIC, Nenad: ADLs and Dynamic Architecture Changes. In: *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops*, ACM Press, 1996, S. 24–27. – DOI http://doi.acm.org/10.1145/243327.243340. – ISBN 0-89791-867-3 8, 96, 99, 107, 127

Medvidovic *et al.* 1996 MEDVIDOVIC, Nenad ; OREIZY, Peyman ; ROBBINS, Jason E. ; TAYLOR, Richard N.: Using object-oriented typing to support architectural design in the C2 style. In: *Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, ACM Press, 1996, S. 24–32. – DOI http://doi.acm.org/10.1145/239098.239106. – ISBN 0-89791-797-9 9

Medvidovic *et al.* 2002 MEDVIDOVIC, Nenad ; ROSENBLUM, David S. ; REDMILES, David F. ; ROBBINS, Jason E.: Modeling software architectures in the Unified Modeling Language. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 11 (2002), Nr. 1, S. 2–57. – DOI http://doi.acm.org/10.1145/504087.504088. – ISSN 1049-331X 8, 127

Medvidovic und Taylor 2000 MEDVIDOVIC, Nenad ; TAYLOR, Richard N.: A Classification and Comparison Framework for Software Architecture Description Languages. In: *IEEE Transactions on Software Engineering* 26 (2000), Januar, Nr. 1, S. 70–93. – DOI http://dx.doi.org/10.1109/32.825767. – ISSN 0098-5589 9, 10, 77, 126, 127, 132

Mehta *et al.* 2000 MEHTA, Nikunj R. ; MEDVIDOVIC, Nenad ; PHADKE, Sandeep: Towards a taxonomy of software connectors. In: *Proceedings of the 22nd international conference on Software engineering*, ACM Press, 2000, S. 178–187. – DOI http://doi.acm.org/10.1145/337180.337201. – ISBN 1-58113-206-9 10, 103, 127, 187

Meng 2001 MENG, Alex C.: On Evaluating Self-Adapive Software. In: *Self-Adaptive Software: First International Workshop, IWSAS 2000, Oxford, UK, April 2000* Bd. 1936 der Reihe *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, 2001, S. 65–74. – OpenURL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=1936&spage=65. – ISSN 0302-9743 132, 133, 134, 135, 136, 137, 143

Mikic-Rakic *et al.* 2002 MIKIC-RAKIC, Marija ; MEHTA, Nikunj ; MEDVIDOVIC, Nenad: Architectural style requirements for self-healing systems. In: *Proceedings of the first workshop on Self-healing systems*, ACM Press, 2002, S. 49–54. – DOI http://doi.acm.org/10.1145/582128.582138. – ISBN 1581136099 132

Miller und Mukerji 2003 MILLER, Joaquin ; MUKERJI, Jishnu: *MDA Guide Version 1.0.1*. 12. Juni 2003. – URL http://www.omg.org/docs/omg/03-06-01.pdf 54

Monroe 2001 MONROE, Robert T.: Capturing software architecture design expertise with Armani. URL http://reports-archive.adm.cs.cmu.edu/anon/1998/abstracts/98-163R.html, Januar 2001 (CMU-CS-98-163). – Forschungsbericht. revised Version 128

Moreira 2003 MOREIRA, Rui S.: *FORMAware: Framework of Reflective Components for Managing Architecture Adaptation*, Lancaster University, Dissertation, Oktober 2003. – URL http://academia.gda.itesm.mx/~hduran/RuiPhDThesisFinal.pdf 147

Moreira und Blair 2004 MOREIRA, Rui S. ; BLAIR, Gordon S.: Supporting Adaptable Distributed Systems with FORMAware. In: *Proceedings of the 24th International Conference on Distributed Computing Systems Workshops - W7: EC (ICDCSW'04)*, IEEE Computer Society, 2004, S. 320–325. – URL http://dares.enst-bretagne.fr/dares2004/Session2/paper5.pdf. – ISBN 0-7695-2087-1 147

Mülle 2001 MÜLLE, Jutta: *Dynamische Gebäude — Verteilte Systeme und Middleware*. 19. Februar 2001. – URL http://www.ipd.uka.de/~dyngeb/topics/versys.html 12

Objs 2002 OBJECT SERVICES & CONSULTING, INC.: *ProbeMeister*. 2002. – URL http://www.objs.com/ProbeMeister/ 193

OC 2003 VDE/ITG/GI: *Organic Computing: Computer- und Systemarchitektur im Jahr 2010*. Juli 2003. – URL http://www.gi-ev.de/download/VDE-ITG-GI-Positionspapier%20Organic%20Computing.pdf. – Positionspapier der Organic-Computing-Initiative 15

van Ommering *et al.* 2000 OMMERING, Rob van ; LINDEN, Frank van der ; KRAMER, Jeff ; MAGEE, Jeff: The Koala Component Model

for Consumer Electronics Software. In: *IEEE Computer* 33 (2000), Nr. 3, S. 78–85. – URL http://www.liacs.nl/~marcello/CBSE/koala.pdf 199, 200

Oreizy 1996 OREIZY, Peyman: Issues in the Runtime Modification of Software Architectures. URL http://www.ics.uci.edu/~peymano/papers/TR-UCI-ICS-96-35.pdf, August 1996 (UCI-ICS-TR-96-35). – Forschungsbericht 77, 81, 125, 127

Oreizy *et al.* 1998 OREIZY, Peyman ; MEDVIDOVIC, Nenad ; TAYLOR, Richard N.: Architecture-Based Runtime Software Evolution. In: *Proceedings of the 20th international conference on Software engineering*, IEEE Computer Society, 1998, S. 177–186. – URL http://www.ics.uci.edu/~peymano/papers/ICSE98.pdf. – ISBN 0-8186-8368-6 42, 45, 82, 92, 98, 120, 126, 127, 129, 156, 207

Oreizy *et al.* 1999 OREIZY, Peyman ; GORLICK, Michael M. ; TAYLOR, Richard N. ; HEIMBIGNER, Dennis ; JOHNSON, Gregory ; MEDVIDOVIC, Nenad ; QUILICI, Alex ; ROSENBLUM, David S. ; WOLF, Alexander L.: An Architecture-Based Approach to Self-Adaptive Software. In: *IEEE Intelligent Systems* 14 (1999), Nr. 3, S. 54–62. – DOI http://doi.ieeecomputersociety.org/10.1109/5254.769885. – ISSN 1094-7167 28, 34, 39, 41, 42, 44, 81, 94, 124, 126, 133, 135, 142, 167, 179, 191, 193, 208

Perry und Wolf 1992 PERRY, Dewayne E. ; WOLF, Alexander L.: Foundations for the study of software architecture. In: *SIGSOFT Software Engineering Notes* 17 (1992), Nr. 4, S. 40–52. – DOI http://doi.acm.org/10.1145/141874.141884. – ISSN 0163-5948 4

Poladian 2003 POLADIAN, Vahe: *17-811 Self-Healing Systems: Class Discussion Summary*. 17. März 2003. – URL http://www-2.cs.cmu.edu/~garlan/17811/Summaries/03-17-Poladian.html 136

Polya 1949 POLYA, George: *How to solve problems*. Princeton University Press, 1949 32, 256

Polya 1995 POLYA, George: *Schule des Denkens: Vom Lösen mathematischer Probleme*. 4. Auflage. Francke Verlag, 1995. – Deutsche Übersetzung von [Polya 1949] durch Elisabeth Behnke. – ISBN 3-7720-0608-6 33

Pomerleau 1993 POMERLEAU, Dean: *Neural network perception for mobile robot guidance*. Kluwer Academic Publishing, Juli 1993. – ISBN 0792393732 159

PONS 2002 REDAKTION PONS WÖRTERBÜCHER: *PONS Großwörterbuch für Experten und Universität Englisch*. 1. Auflage. Ernst Klett Sprachen GmbH, Stuttgart, 2002. – ISBN 3-12-517168-7 19, 20

Popovici *et al.* 2003 POPOVICI, Andrei ; ALONSO, Gustavo ; GROSS, Thomas: Just-In-Time Aspects: Efficient Dynamic Weaving for Java. In: *Proceedings of the 2nd international conference on Aspect-oriented software development*, ACM Press, 2003, S. 100–109. – DOI http://doi.acm.org/10.1145/643603.643614. – ISBN 1-58113-660-9 175

Rademacher 2004 RADEMACHER, Rochus: IT-Architektur à la nature. In: *Computer Zeitung* 10 (2004), 1. März, Nr. 1, S. 1, 6 15, 17, 41, 61, 132, 203

Reece 2001 REECE, Steven: Self-Adaptive Multi-sensor Systems. In: *Self-Adaptive Software: First International Workshop, IWSAS 2000, Oxford, UK, April 2000* Bd. 1936 der Reihe *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, 2001, S. 224–241. – OpenURL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=1936&spage=224. – ISSN 0302-9743 142

Reilly *et al.* 2002 REILLY, D. ; TALEB-BENDIAB, A. ; LAWS, A. ; BADR, N.: An instrumentation and control-based approach for distributed application management and adaptation. In: *Proceedings of the first workshop on Self-healing systems*, ACM Press, 2002, S. 61–66. – DOI http://doi.acm.org/10.1145/582128.582140. – ISBN 1-58113-609-9 191

Ritter 2000 RITTER, Thorsten: *Entwurf und Implementierung eines Rahmenwerks für persistente Objekte*, Fernuniversität Hagen, Diplomarbeit, 2000. – URL http://www.informatik.fernuni-hagen.de/import/pi3/thorsten/DiplomarbeitRitter.pdf 143

Robertson 2001 ROBERTSON, Paul: An Architecture for Self-Adaptation and Its Application to Aerial Image Understanding. In: *Self-Adaptive Software: First International Workshop, IWSAS 2000, Oxford, UK, April 2000* Bd. 1936 der Reihe *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, 2001, S. 199–223. – OpenURL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=1936&spage=199. – ISSN 0302-9743 133

Robertson 2003 ROBERTSON, Paul: Confidence from Self-knowledge and Domain Knowledge. In: *Self-Adaptive Software — Applications: Second International Workshop (IWSAS 2001)* Bd. 2614 der Reihe *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, 2003, S. 84–105. – OpenURL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=2614&spage=84. – ISBN 3-540-00731-8 143

Robertson und Brady 1999 ROBERTSON, Paul ; BRADY, J. M.: Adaptive Image Analysis for Aerial Surveillance. In: *IEEE Intelligent Systems* 14 (1999), Nr. 3, S. 30–36. – DOI http://doi.ieeecomputersociety.org/10.1109/5254.769882. – ISSN 1094-7167 133, 142, 143

Robertson *et al.* 2001 ROBERTSON, Paul ; LADDAGA, Robert ; SHROBE, Howie: Introduction: The First International Workshop on Self-Adaptive Software. In: *Self-Adaptive Software: First International Workshop, IWSAS 2000, Oxford, UK, April 2000* Bd. 1936 der Reihe *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, 2001, S. 1–10. – OpenURL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=1936&spage=1. – ISSN 0302-9743 143

Röttger und Zschaler 2003 RÖTTGER, Simone ; ZSCHALER, Steffen: CQML$^+$: Enhancements to CQML. In: *Proceedings 1st International Workshop on Quality of Service in Component-Based Software Engineering, Toulouse, France*, Cépaduès- Éditions, Juni 2003, S. 43–56. – URL http://www-st.inf.tu-dresden.de/comquad/qoscbse03-language.pdf 132

SAACS 2003 SAACS: *1st International Workshop on Autonomic Computing Systems, Prague, Czech Republic, September 1–5.* 2003. – URL http://cms1.gre.ac.uk/conferences/DexaWS_Autonomic/ACS.htm 173

SAACS 2004 SAACS: *2nd International Workshop on Self-Adaptive and Autonomic Computing Systems (SAACS 04), Zaragoza, Spain, August 30–September 4th.* 2004. – URL http://cms1.gre.ac.uk/conferences/dexa2004/ws-2004/index.htm 173

Salisan 2004 SALISAN, John: *Dynamic Assembly for Systems Adaptability, Dependability, and Assurance (DASADA).* 2004. – URL http://www.rl.af.mil/tech/programs/dasada/ 167, 179, 193, 194, 206, 209

Schill 2004 SCHILL, Alexander: *Objektmigration.* 2004. – URL http://www.rn.inf.tu-dresden.de/scripts_lsrn/lehre/ds/print/09_Objektmigration.pdf. – Skript zur Vorlesung »Verteilte Systeme« an der TU Dresden 107

Schill *et al.* 2000 SCHILL, Alexander ; HÄRTIG, Herrmann ; HUSSMANN, Heinrich ; MEISSNER, Klaus ; MEYER-WEGENER, Klaus ; PFITZMANN, Andreas: *Antrag auf Sachbeihilfe zur Einrichtung einer Forschergruppe an der Technischen Universität Dresden.* August 2000. – Erstantrag COMQUAD 47, 203

Schmeck 2004 SCHMECK, Hartmut: *Organic Computing: Selbstorganisation*. 27. April 2004. – URL http://www.aifb.uni-karlsruhe.de/Lehre/Sommer2004/OC/20040427_Organic_Computing_Schmeck.pdf 15, 16

Schmerl und Garlan 2002 SCHMERL, Bradley ; GARLAN, David: Exploiting architectural design knowledge to support self-repairing systems. In: *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, ACM Press, 2002, S. 241–248. – DOI http://doi.acm.org/10.1145/568760.568804. – ISBN 1-58113-556-4 124, 125, 126, 168

Schmidt *et al.* 1999a SCHMIDT, Albrecht ; AIDOO, Kofi A. ; TAKALUOMA, Antti ; TUOMELA, Urpo ; LAERHOVEN, Kristof V. ; VELDE, Walter V. de: Advanced Interaction in Context. In: *Handheld and Ubiquitous Computing: First International Symposium, HUC'99, Karlsruhe, Germany, September 1999* Bd. 1707 der Reihe *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, 1999, S. 89–101. – OpenURL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=1707&spage=89. – ISSN 0302-9743 89

Schmidt *et al.* 1999b SCHMIDT, Albrecht ; BEIGL, Michael ; GELLERSEN, Hans-W.: There is more to Context than Location. In: *Computers and Graphics* 23 (1999), Nr. 6, S. 893–901. – URL http://www.comp.lancs.ac.uk/~albrecht/pubs/pdf/schmidt_cug_elsevier_12-1999-context-is-more-than-location.pdf 37

SEI 2004a SOFTWARE ENGINEERING INSTITUTE (SEI): *Architecture Reconstruction*. 15. Juni 2004. – URL http://www.sei.cmu.edu/activities/ata/ata_extraction.html 151

SEI 2004b SOFTWARE ENGINEERING INSTITUTE (SEI): *How do you define Software Architecture*. 20. August 2004. – URL http://www.sei.cmu.edu/architecture/definitions.html 3

Seifert und Lammersen 2004 SEIFERT, Julia ; LAMMERSEN, Christiane: *Modellierung der Farbsehfähigkeiten*. 4. März 2004. – URL http://www.uni-paderborn.de/fachbereich/AG/agdomik/studienarbeiten/Seifert_Lammersen/ 88

Self-∗ 2004 SELF-∗: *SELF-STAR: International Workshop on Self-* Properties in Complex Information Systems, 31 May–2 June 2004, University of Bologna*. 2004. – URL http://www.cs.unibo.it/self-star/ 15, 173

Seth und Kokar 2003 SETH, Deepak ; KOKAR, Mieczyslaw M.: SSCS: A Smart Spell Checker System Implementation. In: *Self-Adaptive Software — Applications: Second Interna-*

tional Workshop (IWSAS 2001) Bd. 2614 der Reihe *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, 2003, S. 187–197. – OpenURL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=2614&spage=187. – ISBN 3-540-00731-8 28, 54, 157, 158

Shamma 1996 SHAMMA, Jeff S.: *Linearization and Gain-Scheduling*. Kap. 20.3, S. 388–398. In: LEVINE, William S. (Hrsg.): *The Control Handbook*, 1996 (The electrical engineering handbook series). – ISBN 0-8493-8570-9 139

Shaw und Garlan 1996 SHAW, Mary ; GARLAN, David: *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, New Jersey, 1996. – ISBN 0131829572 133, 134

Smith 1982 SMITH, Brian C.: *Reflection and Semantics in a Procedural Language*, MIT Laboratory for Computer Science, Cambridge, MA, Dissertation, 1982 143

Soto und Khosla 2003 SOTO, Alvaro ; KHOSLA, Pradeep: Adaptive Agent Based System for State Estimation Using Dynamic Multidimensional Information Sources. In: *Self-Adaptive Software — Applications: Second International Workshop (IWSAS 2001)* Bd. 2614 der Reihe *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, 2003, S. 66–83. – OpenURL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=2614&spage=66. – ISBN 3-540-00731-8 28, 157, 159

Springer 2004 SPRINGER, Thomas: *Ein komponentenbasiertes Meta-Modell kontextabhängiger Adaptionsgraphen für mobile und ubiquitäre Anwendungen*, Technische Universität Dresden, Dissertation, März 2004 28, 34, 35, 36, 37, 38, 42, 43, 45, 50, 57, 90, 110, 115, 185, 191, 224

Stal 2003 STAL, Michael: Der Stein der Weisen (Editorial). In: *JavaSPEKTRUM* 2 (2003), S. 3. – ISSN 1431-4436 54

Starke 2002 STARKE, Gernot: *Effektive-Softwarearchitekturen: Ein praktischer Leitfaden*. Carl Hanser Verlag, 2002. – ISBN 3-446-21998-6 4, 6, 180

Steinberg 2003a STEINBERG, Daniel H.: *What you need to know about autonomic computing, Part 1: An introduction and overview*. August 2003. – URL http://www-106.ibm.com/developerworks/ibm/library/i-autonom1/ 14, 25, 81, 170

Steinberg 2003b STEINBERG, Daniel H.: *What you need to know about autonomic computing, Part 2: The Infrastructure*. August 2003. – URL http://www-106.ibm.com/developerworks/ibm/library/i-autonom2/ 170, 171, 172

Stieler und Marsiske 2003 STIELER, Wolfgang ; MARSISKE, Hans-Arthur: Internet mit Gedächtnisschwund. In: *c't — magazin für computer technik* (2003), Nr. 25, S. 57. – ISSN 0724-8679 235

Subramanian und Chung 2002a SUBRAMANIAN, Nary ; CHUNG, Lawrence: Software architecture adaptability: an NFR approach. In: *Proceedings of the 4th international workshop on Principles of software evolution*, ACM Press, 2002, S. 52–61. – DOI http://doi.acm.org/10.1145/602461.602470. – ISBN 1-58113-508-4 24, 28, 62, 63, 70, 74, 75

Subramanian und Chung 2002b SUBRAMANIAN, Nary ; CHUNG, Lawrence: Tool support for engineering adaptability into software architecture. In: *Proceedings of the international workshop on Principles of software evolution*, ACM Press, 2002, S. 86–96. – DOI http://doi.acm.org/10.1145/512035.512056. – ISBN 1-58113-545-9 28, 62, 63, 70, 71, 72, 74, 75, 76

Szyperski 2002 SZYPERSKI, Clemens: *Component Software — Beyond Object-Oriented Programming*. Zweite Auflage. Auflage. Addison-Wesley, 2002 (Component Software Series). – ISBN 0-201-74572-0 172

Süddeutsche 2004 O. V.: Einsatz für die Fehler-Feuerwehr. In: *Süddeutsche Zeitung* (2004), 27. März, Nr. 73 17

Taylor und Tofts 2004 TAYLOR, Richard ; TOFTS, Chris: Self Managed Systems: A Control Theory Perspective. In: *SELF-STAR: International Workshop on Self-* Properties in Complex Information Systems, 31 May–2 June 2004, University of Bologna*, URL http://www.cs.unibo.it/self-star/papers/tofts.pdf, 2004 133, 135, 142

Taylor *et al.* 1996 TAYLOR, Richard N. ; MEDVIDOVIC, Nenad ; ANDERSON, Kenneth M. ; WHITEHEAD, JR., E. J. ; ROBBINS, Jason E. ; NIES, Kari A. ; OREIZY, Peyman ; DUBROW, Deborah L.: A Component- and Message-Based Architectural Style for GUI Software. In: *IEEE Transactions on Software Engineering* 22 (1996), Nr. 6, S. 390–406. – DOI http://dx.doi.org/10.1109/32.508313. – ISSN 0098-5589 92, 168

Tekinerdogan und Aksit 1996 TEKINERDOGAN, Bedir ; AKSIT, Mehmet: *Adaptability in Object-Oriented Software Development Workshop Report*. 1996. – URL http://trese.cs.utwente.nl/publications/papers/sioop96aiood.pdf. – 10th European Conference on Object-Oriented Programming, July 8–12, 1996, Linz, Austria 27, 32, 33, 53

Tisato *et al.* 2001 TISATO, Francesco ; SAVIGNI, Andrea ; CAZZOLA, Walter ; SOSIO, Andrea: Architectural Reflection: Realising Software Architectures via Reflective Activities. In: *Engineering Distributed Objects: Second International Workshop (EDO 2000)* Bd. 1999 der Reihe *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, 2001, S. 102–115. – OpenURL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=1999&spage=102. – ISSN 0302-9743 150

Turowski 2002 ACKERMANN, Jörg ; BRINKOP, Frank ; CONRAD, Stefan ; FETTKE, Peter ; FRICK, Andreas ; GLISTAU, Elke ; JAEKEL, Holger ; KOTLAR, Otto ; LOOS, Peter ; MRECH, Heike ; ORTNER, Erich ; RAAPE, Ulrich ; OVERHAGE, Sven ; SAHM, Stephan ; SCHMIETENDORF, Andreas ; TESCHKE, Thorsten ; TUROWSKI, Klaus ; TUROWSKI, Klaus (Hrsg.): *Vereinheitlichte Spezifikation von Fachkomponenten*. Arbeitskreis 5.10.3 der Gesellschaft für Informatik (GI), Februar 2002. – URL http://wi2.wiwi.uni-augsburg.de/downloads/gi-files/MEMO/Memorandum-final-2-44-mit-literatur-Web.pdf. – Memorandum, Website: http://www.fachkomponenten.de 60, 61

UDDI ORGANIZATION FOR THE ADVANCEMENT OF STRUCTURED INFORMATION STANDARDS (OASIS): *Universal Description, Discovery and Integration*. – URL http://www.uddi.org/ 172

UIML O. V.: *UIML.org*. – URL http://www.uiml.org/ 54

Ullmann 2004 ULLMANN, Andreas: Buzzword SOA: Was verbirgt sich hinter serviceorientierten Architekturen? In: *Javamagazin* (2004), Nr. 10, S. 55–61. – ISSN 1619-795X 172, 217

Ultsch 2004 ULTSCH, A.: *AG Datenbionik*. 2004. – URL http://www.informatik.uni-marburg.de/~databionics/ 12, 13, 16

Valetto und Kaiser 2002a VALETTO, Giuseppe ; KAISER, Gail: A case study in software adaptation. In: *Proceedings of the first workshop on Self-healing systems*, ACM Press, 2002, S. 73–78. – DOI http://doi.acm.org/10.1145/582128.582142. – ISBN 1581136099 133, 135, 142, 167, 209

Valetto und Kaiser 2002b VALETTO, Giuseppe ; KAISER, Gail: Combining Mobile Agents and Process-based Coordination to Achieve Software Adaptation. URL http://www.cs.columbia.edu/~library/TR-repository/reports/reports-2002/cucs-007-02.pdf, März 2002 (CUCS-007-02). – Forschungsbericht 206, 209

Valetto *et al.* 2001 VALETTO, Giuseppe ; KAISER, Gail ; KC, Gaurav S.: A Mobile Agent Approach to Process-Based Dynamic Adaptation of Complex Software Systems. In: *Software Pro-*

cess Technology: 8th European Workshop, EWSPT 2001, Witten, Germany, June 19–21, 2001 Bd. 2077 der Reihe *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, 2001, S. 102–116. – OpenURL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=2077&spage=102. – ISSN 0302-9743 206, 209

Vallecillo *et al.* 2000 VALLECILLO, Antonio ; HERNÁNDEZ, Juan ; TROYA, José M.: New Issues in Object Interoperability. In: *Object-Oriented Technology: ECOOP 2000 Workshop Reader* Bd. 1964 der Reihe *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, 2000, S. 256–269. – OpenURL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=1964&spage=256. – ISSN 0302-9743 102, 199

van der Westhuizen und van der Hoek 2002 VAN DER WESTHUIZEN, Chris ; VAN DER HOEK, André: Understanding and Propagating Architectural Change. In: *Proceedings of the Working IEEE/IFIP Conference on Software Architecture 2002 (WICSA 3), August 2002*, URL http://www.ics.uci.edu/~andre/research/papers/WICSA2002.pdf, 2002 127

Vanthournout *et al.* 2004 VANTHOURNOUT, Koen ; DECONINCK, Geert ; BELMANS, Ronnie: A Taxonomy for Resource Discovery. In: *Organic and Pervasive Computing — ARCS 2004: International Conference on Architecture of Computing Systems, Augsburg, Germany, March 23–26, 2004* Bd. 2981 der Reihe *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, 2004, S. 78–91. – OpenURL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=2981&spage=78. – ISSN 0302-9743 192, 201

Vrandecic 2001 VRANDECIC, Zdenko D.: *Was ist Software Architektur*. 2001. – URL http://www.nodix.de/denny/studium/infotik/sa.htm. – Ausarbeitung zum Hauptseminar Software Architektur 5

Vögler *et al.* 2004 VÖGLER, Gabriel ; FLOR, Thomas ; HOHLFELD, Bernhard ; GRIMM, Klaus ; MERKER, Wolfgang: Multi-Client-Architekturen für Pervasive Computing / DaimlerChrysler, RIC/SA. 5. Februar 2004. – Forschungsbericht 30, 89, 110, 233

Walsh 2004 WALSH, Norman: *DocBook*. 2004. – URL http://www.docbook.org/ 63

Weber 1998 WEBER, Michael: *Verteilte Systeme*. Spektrum Akademischer Verlag, 1998. – ISBN 3-8274-0221-2 188

Wermelinger *et al.* 2001 WERMELINGER, Michel ; LOPES, Antónia ; FIADEIRO, José L.: A graph based architectural (Re)configuration language. In: *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, ACM Press, 2001, S. 21–32. – DOI http://doi.acm.org/10.1145/503209.503213. – ISBN 1-58113-390-1 130

Wermelinger 1999 WERMELINGER, Miguel A.: *Specification of Software Architecture Reconfiguration*, Universidade Nova de Lisboa, Dissertation, September 1999. – URL http://ctp.di.fct.unl.pt/~mw/pubs/1999/thesis.pdf 9, 54, 77, 121, 129, 131, 132, 151, 166, 207

Wile 2001 WILE, David S.: Using Dynamic ACME. In: *Proceedings of a Working Conference on Complex and Dynamic Systems Architecture. Brisbane, Australia. Dec. 2001.*, URL http://mr.teknowledge.com/wile/Papers/Using_Dynamic_Acme_Distribution_Copy.pdf, 2001 128

Wile 2002 WILE, David S.: Towards a Synthesis of Dynamic Architecture Event Languages. In: *Proceedings of the first workshop on Self-healing systems*, ACM Press, 2002, S. 79–84. – DOI http://doi.acm.org/10.1145/582128.582143. – ISBN 1581136099 167

Worden 2004 WORDEN, Daniel: *Understand autonomic maturity levels*. 2004. – URL http://www.developers.net/node/view/61 81

xADL 2002 INSTITUTE FOR SOFTWARE RESEARCH, UNIVERSITY OF CALIFORNIA: *xADL 2.0 — A Highly Extensible, xArch-based Architecture Description Language*. November 2002. – URL http://www.isr.uci.edu/projects/xarchuci/index.html 127

Yang *et al.* 2002 YANG, Z. ; CHENG, B. H. C. ; STIREWALT, R. E. K. ; SOWELL, J. ; SADJADI, S. M. ; MCKINLEY, P. K.: An aspect-oriented approach to dynamic adaptation. In: *Proceedings of the first workshop on Self-healing systems*, ACM Press, 2002, S. 85–92. – DOI http://doi.acm.org/10.1145/582128.582144. – ISBN 1581136099 175

Zambrano *et al.* 2004 ZAMBRANO, Arturo ; GORDILLO, Silvia ; JAUREGUIBERRY, Ignacio: Aspect-Based Adaptation for Ubiquitous Software. In: *Mobile and Ubiquitous Information Access: Mobile HCI 2003 International Workshop, Udine, Italy, September 8, 2003* Bd. 2954 der Reihe *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, 2004, S. 215–226. – OpenURL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=2954&spage=215. – ISBN 3-540-21003-2 142, 175

Zave und Jackson 1997 ZAVE, Pamela ; JACKSON, Michael: Four dark corners of requirements engineering. In: *ACM Transactions Software Engineering Methodology* 6 (1997), Nr. 1, S. 1–30. – DOI http://doi.acm.org/10.1145/237432.237434. – ISSN 1049-331X 66

Zinky *et al.* 2002 ZINKY, John ; LOYALL, Joseph ; SHAPIRO, Richard: Runtime Performance Modeling and Measurement of Adaptive Distributed Object Applications. In: *On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE: Confederated International Conferences CoopIS, DOA, and ODBASE 2002* Bd. 2519 der Reihe *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, 2002, S. 755–772. – OpenURL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=2519&spage=775. – Online in SpringerLink seit: Juni 2003. – ISSN 0302-9743 132

Selbstständigkeitserklärung

Hiermit erkläre ich, dass die vorliegende Arbeit durch mich selbstständig und nur unter Verwendung der angegebenen Literatur angefertigt wurde.

Meusegast am 27. September 2004 Sebastian Klamar