Middleware-Architektur für mobile Informationssysteme

Diplomarbeit

Helge Sichting

Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Institut für Technische und Betriebliche Informationssysteme

14. Januar 2004

Aufgabenstellung : Prof. Dr. Gunter Saake

Betreuung : Dipl.-Inf. Sven Apel

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation 1
1.2 Zielstellung 3
1.3 Terminologie 3
1.4 Gliederung 4

2 Grundlagen **5**

2.1 Kommunikation 5
2.1.1 Prozeßkommunikation im verteilten System 5
2.1.2 Nachrichten 6
2.1.3 Entfernte Funktionsaufrufe 10
2.1.4 Objektorientierung im verteilten System 11
2.2 Softwaretechnische Grundlagen 13
2.2.1 Domain Engineering 13
2.2.2 Merkmalsbasierte Analyse 15
2.2.3 Programmfamilie 17
2.2.4 Objektorientierung 18
2.2.5 Kollaborationenentwurf 19
2.2.6 GenVoca-Grammatiken 22
2.2.7 Implementierung von Schichten-Architekturen 23
2.2.8 Aspektorientierung 26
2.2.9 Merkmalsbasierte Konfiguration 29
2.3 Mobile Middleware 31
2.3.1 Middleware 31
2.3.2 Mobile Kommunikation 33
2.3.3 Middleware Technologien 36
2.3.4 XML-Nachrichtensysteme 39

3 Domänenanalyse **43**
3.1 Merkmalsbindung 44
3.2 Client-Server Architektur 44
3.3 Merkmale der Server 45
3.4 Merkmale der Clients 46
3.5 Allgemeine Merkmale 47

4 Entwurf **50**
4.1 Überblick 51
4.2 Die Clients 51
4.3 Die Server 62
4.4 Dienstgütearchitektur 69
4.4.1 Funktionale Hierarchie von Dienstgütemechanismen 71
4.4.2 Dienstgüte-Integration 71

5 Implementierung **75**
5.1 Client- und Server-Schichten 77
5.2 Die Clients 90
5.3 Die Server 94
5.4 Fallstudien/Konfiguration 99

6 Zusammenfassung, Bewertung und Ausblick **109**
6.1 Zusammenfassung und Bewertung 109
6.2 Ausblick 112

Abbildungsverzeichnis

2.1 Lokale und entfernte Prozeßkommunikation 6
2.2 Synchrone Kommunikation 7
2.3 Asynchrone Kommunikation 8
2.4 Dimensionen der Nachrichtenkommunikation für entfernte Aufrufe 9
2.5 Phasen des Produktlinien-Engineerings 14
2.6 Domain Engineering: Abfolge und Techniken 15
2.7 Beziehungen zwischen Merkmalen (angelehnt an [KCH+90]) 16
2.8 Funktionale Hierarchie in einer Programmfamilie (angelehnt an [Hab76]) 18
2.9 Programmfamilie und Objektorientierung 19
2.10 Kollaborationen, Objekte und Rollen (angelehnt an [SB02]) 22
2.11 Beispiel einer GenVoca Grammatik 23
2.12 Mixins in C++ 24
2.13 Mixin-Hierarchie in C++ 24
2.14 Mixin-Schichten und innere Mixins 25
2.15 Implementierung von Mixin-Schichten in C++ 26
2.16 Modularisierung eines Crosscutting Concerns als Aspekt [SGSP02] 28
2.17 Zusammenweben von Komponenten- und Aspektcode 29
2.18 Klassische Middleware 31
2.19 Schmatischer Aufbau einer SOAP-Nachricht 41
2.20 SOAP-RPC-Nachrichten: Anfrage und Antwort 42

3.1 Architekturentwurf [AP03a] 43
3.2 Server-Merkmale 45
3.3 Client-Merkmale 47
3.4 Allgemeine Merkmale 47

4.1 Funktionale Hierarchie der Clients: Von der Nachricht zum Objektaufruf 52
4.2 Integration von Verbindung(en) 54
4.3 Konfiguration des Merkmals “Datentyp” 55
4.4 Deserialisierung einer Nachricht von einem Datenstrom 56

4.5 Alternative Synchronisationsstrategien 57
4.6 Serialisierung und Senden eines OneWay 58
4.7 Ausführung eines TwoWay 58
4.8 Konfiguration des Merkmals “Richtung” 59
4.9 Lebenszyklus eines entfernten Objektes 61
4.10 Konfiguration des Serviceparadigmas 62
4.11 Separierung von Client- und Servermerkmalen 63
4.12 Servermerkmale Connection und ReceiveCalls 65
4.13 Serverschichten SRPC, SRCI und SROI 66
4.14 Funktionale Hierarchie der Middlewarefamilie 68
4.15 Protokollimplementierung als Aspekt 69
4.16 Kommunikationspfad in einem verteilten System 71
4.17 Crosscutting Concern “Reliability” 72
4.18 Beispiel eines Dienstgüte-Mechanismus [Bec01] 74

5.1 Abstrakte Nachrichtenschicht 78
5.2 Parametrisierte Vererbung der Verbindungsschicht 78
5.3 Instantiierung der Verbindungsschicht 79
5.4 Allgemeine und protokollspezifische Schicht 80
5.5 Mixin-Implementierung der Parameterschicht 80
5.6 Spezialisierung der Parameterschicht mit einem Protokoll 81
5.7 Implementierung der untersten drei Kollaborationen 81
5.8 Mixin-Implementierung der Richtungstrennungsschicht 82
5.9 Das Entwurfsmuster *Strategie* [GHJV95] 83
5.10 Vermittler zwischen Kontext und Strategien 85
5.11 Zwei Synchronisationsstrategien 85
5.12 Strategie-Vermittler (Messenger) 86
5.13 Strategie-Basisklasse für späte Bindung 86
5.14 Zwei Synchronisationsstrategien mit Bindungsparameter 87
5.15 Dynamische Rekonfigurierung von Strategien 87
5.16 Früh gebundene Strategien 88
5.17 Klassendiagramm zur Implementierung von Strategien 88
5.18 Konfigurierung des Kontext 89
5.19 Instantiierung von mxOneWay und mxTwoWay 91
5.20 Klassenbeziehungen von mxOneWay und mxTwoWay 91
5.21 Klassendiagramm der Nachrichten-Mixins 92
5.22 Implementierung einer Client-Applikation 93

5.23 Konfiguration einer Client-Applikation mit einer Middleware 94
5.24 Serververbindung 95
5.25 Server-Mixin-Schichten als Klassendiagramm 96
5.26 Funktionen beim Server anmelden 97
5.27 Klassen beim Server anmelden 98
5.28 Klassenobjekte beim Server anmelden 98
5.29 Konfiguration eines WebService-Clients 102
5.30 Implementierung eines WebService-Clients 103
5.31 Konfiguration eines Sensor-Clients 104
5.32 Konfiguration eines Actor-Servers 105
5.33 Konfiguration innerhalb des Actor-Servers 105
5.34 Konfiguration eines stationären Servers 106
5.35 GenVoca-Grammtik der Middlewarefamilie 107
5.36 Konfiguration der Protokollimplementierung 108

Tabellenverzeichnis

2.1 Beispiele wichtiger Nachrichtenmuster 9
2.2 Beispiele technologiebasierter Dienstgüten [CS99] 36
2.3 Beispiele nutzerbasierter Dienstgüten [CS99] 37

5.1 Konfigurationsmöglichkeiten für Strategien 89
5.2 Übersicht der Middleware-Schichten 100

Abkürzungsverzeichnis

AOP	Aspect-Oriented Programming
COM	Component Object Model
CONSUL	CONfiguration SUpport Library
CORBA	Common Object Request Broker Architecture
DBMS	Database Management System
DCOM	Distributed Component Object Model
FODA	Feature-Oriented Domain Analysis
GPRS	General Packet Radio Service
GSM	Global System for Mobile Communications
HTTP	Hypertext Transfer Protocol
J2ME	Java 2 Micro Edition
J2SE	Java 2 Standard Edition
OOP	Object-Oriented Programming
OOVM	Object-Oriented Virtual Machine
ORB	Object Request Broker
P2P	Peer-to-Peer
PDA	Personal Digital Assistent
QoS	Quality of Service
RCI	Remote Class Invocation
RMI	Remote Method Invocation
ROI	Remote Object Invocation
RPC	Remote Procedure Call
SOAP	Simple Object Access Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UMTS	Universal Mobile Telecommunications System
WLAN	Wireless Local Area Network
WSDL	Web Service Definition Language
XML	Extensible Markup Language

Kapitel 1

Einleitung

1.1 Motivation

Eine neue Generation von mobilen, kleinen Computern soll zusammen mit einer neuen Generation von Funknetzen die Informationsgesellschaft so grundlegend verändern, wie vielleicht einst das Internet oder das Handy. Das jedenfalls meinen Hardwarehersteller, Netzbetreiber und Wissenschaftler.

Die Tage der elektronischen Terminkalender, Fremdsprachenführer und Notizbücher, der Gameboys und der tragbaren Audiogeräte sind gezählt. Vollwertige Minicomputer dieser Größe (PDA[1]), die von unterschiedlichen Herstellern mit unterschiedlicher Hardware angeboten werden, können sich zwar nicht mit aktuellen PCs messen, doch sie besitzen genug Rechenleistung und Speicherkapazität, damit selbst (abgespeckte) Multitasking-Betriebssysteme wie *Embedix Linux* oder *Windows CE* auf ihnen mitsamt der genannten Anwendungen zum Einsatz kommen können. Selbst Office-Anwendungen und Anwendungen für Internet-Dienste wie WWW oder Email sind auf den Geräten lauffähig, die oft über ein Touchscreen-Farbdisplay und Audiofähigkeiten verfügen.

Gleichzeitig arbeiten verschiedene Netzbetreiber und Hersteller von Kommunikationstechnik in ganz Europa an einem Mobilfunknetz der dritten Generation - *UMTS*[2] - das mit einer Übertragungsgeschwindigkeit von bis zu zwei Megabit pro Sekunde einen Durchbruch in der mobilen Datenübertragung schaffen wird. UMTS ist das Medium für die Datenübertragung mobiler Geräte vom Smartphone, PDA bis zum Laptop.

Funknetze und mobile Geräte sollen in naher Zukunft eine beispiellose Symbiose eingehen und mit oder ohne stationären Computern neue Netzwerke bilden.

Daraus werden neue mobile Anwendungen und Dienste hervorgehen, die alltägliche, aus der Mobilität der Menschen resultierende Probleme dem Computer übertragen. Die neue Qualität der mobilen Kommunikation von tragbaren Computern wird mit den Begriffen

[1] Personal Digital Assistent
[2] Universal Mobile Telecommunications System

ubiquitous computing[3], *pervasive computing*[4] und und *nomadic computing*[5] beschrieben. Wichtige Anwendungsbeispiele für die Technologie lassen sich ableiten aus der mobilen Beschaffung von Informationen, der Kommunikation, der Unterhaltung oder der mobilen Steuerung technischer Einrichtungen.

Die Entwickler mobiler Applikationen stehen nicht nur vor der Herausforderung, verschiedene Geräte mit unterschiedlicher Ressourcenausstattung und inkompatiblen Betriebssystemen zu integrieren, sondern sie müssen für mobile Kommunikation auch die Eigenschaften zellularer Funknetze berücksichtigen. Bandbreitenschwankungen, Übertragungsstörungen, Verbindungsabbrüche und Änderungen in der Netztopologie sind keine Ausnahmeerscheinungen, sondern Normalzustand bei mobiler Kommunikation und erfordern eine dynamische Anpassung der Kommunikationsfunktionen. Die Kommunikationsfunktionen sind unter diesen Umständen sehr aufwendig und sollen nicht für jede Applikation neu entwickelt werden. Deshalb bietet sich an, diese Funktionen durch eine Middleware für mobile Anwendungen und Geräte bereitzustellen.

Herkömmliche Middleware-Technologie ist dazu - wie später genauer gezeigt wird - nicht geeignet. Die vorhandenen Systeme sind abhängig von statischen Netzen sowie stabilen Kommunikationseigenschaften, und sie benötigen viele Systemressourcen. Ihre Kommunikationsfunktionen werden transparent ausgeführt - Entwickler, Applikationen und Nutzer haben keine Möglichkeit, auf dynamische Änderungen in einem Kommunikationsszenario Einfluß zu nehmen.

Eine Middleware für mobile Systeme soll Standardschnittstellen und Infrastrukturdienste anbieten, um die Entwicklung von mobilen Anwendungen zu erleichtern. Die Middleware stellt Kommunikationsmechanismen bereit, die die Heterogenität der Geräte und des Netzes und die besonderen Charakteristiken im mobilen Umfeld berücksichtigen. Kommunikationsfunktionen sollen nicht nur transparent, sondern auch "unter Aufsicht" und unter Berücksichtigung des aktuellen Kontext ausgeführt werden können[6].

Um mobilen Applikationen den Zugriff auf Informationen bzw. Daten zu erleichtern, sollen mobile, verteilte Informationssysteme etabliert werden. Dazu können stationäre und mobile DBMS zur Datenhaltung und -verwaltung kombiniert werden. Mit Hilfe der Middleware wird die Integration der von verschiedenen Systemen angebotenen Informationsdienste unterstützt.

[3] ubiquitous (engl.) = allgegenwärtig [Wei93]
[4] pervasive (engl.) = durchdringend
[5] nomadic (engl.) = nomadisch
[6] "Context Awareness"

1.2 Zielstellung

Ziel dieser Arbeit ist es, ein bestehendes grobes Konzept einer Middleware-Architektur nach [AP03a] auszubauen und weiter zu verfeinern.

Ausgehend von einem leichtgewichtigen Kommunikationsprotokoll sollen die Kommunikationsmechanismen für funktions- und objektbasierte Dienstaufrufe entworfen und implementiert werden. Dabei soll das Problem "Dienstgüte" Berücksichtigung finden.

Das Ergebnis dieser Diplomarbeit soll als Grundlage für eine Middleware-Familie im Sinne des Programmfamilienkonzeptes dienen. Durch den Einsatz von Programmfamilienkonzept, Aspektorientierung und Objektorientierung soll bei dem Entwurf und der Implementierung ein Höchstmaß an Wiederverwendbarkeit, Flexibilität, Erweiterbarkeit und Konfigurierbarkeit der Familie erreicht werden. Das aspektorientierte Entwurfs- und Implementierungskonzept soll die Modellierung nicht-funktionaler Eigenschaften des Systems unterstützen.

Die genannten Entwurfs- und Implementierungsmethoden werden hinsichtlich ihrer Eignung und ihres Nutzens für die weitere Entwicklung der Middleware-Plattform untersucht.

1.3 Terminologie

Wenn von den Anforderungen an eine mobile Middleware für kleine Geräte die Rede ist, fallen immer wieder Begriffe wie Konfigurierbarkeit, Erweiterbarkeit und Flexibilität. Im Folgenden werden zuerst diese Begriffe geklärt:

- Als *Konfigurierbarkeit* ist in dieser Arbeit statische Konfigurierbarkeit gemeint, gleichbedeutend mit *Anpaßbarkeit*. Konfigurierbarkeit ist eine Voraussetzung für Maßschneiderung.
- Dagegen ist *Rekonfigurierbarkeit* die Fähigkeit eines Softwaresystems, sich dynamisch zur Laufzeit zu verändern.
- Unter *Erweiterbarkeit* einer Software wird die Eigenschaft verstanden, zur Implementierungszeit leicht neue Funktionen hinzufügen zu können.
- *Wiederverwendbarkeit* beschreibt die Eigenschaft einer Software oder eines Softwarebausteins, ihn leicht für viele ähnliche Probleme einsetzen zu können.

1.4 Gliederung

Die Arbeit ist wie folgt gegliedert.

- **Kapitel 2: Grundlagen**
  In drei Abschnitten werden die Kommunikationsgrundlagen, die softwaretechnischen Grundlagen und der Problemraum "Mobile Middleware" analysiert.

- **Kapitel 3: Domänenanalyse**
  Die Merkmale des Zielsystems werden definiert und in Beziehung zueinander gesetzt.

- **Kapitel 4: Entwurf**
  Es wird aufgezeigt, wie grundlegende Bestandteile der Middleware unter Verwendung vorgestellter Softwaretechniken entworfen werden.

- **Kapitel 5: Implementierung**
  Ausgewählte Teile des Entwurfs werden implementiert, so daß einzelne Beispielmitglieder der Familie erstellt werden können. Dabei werden vorgestellte Implementierungstechniken verwendet. Es wird eine sehr flexible Implementierung des bekannten *Strategy Pattern* [GHJV95] vorgestellt. Anschließend wird die Konfiguration der Familie erläutert.

- **Kapitel 6: Bewertung, Zusammenfassung und Ausblick**
  Die bei der Realisierung gewonnenen Erkenntnisse zu den vorgestellten Entwurfs- und Implementierungsmethoden werden im letzten Kapitel bewertet und diskutiert. Abschließend wird ein Ausblick auf zukünftige Entwicklungsmöglichkeiten gegeben.

Kapitel 2

Grundlagen

2.1 Kommunikation

Für das weitere Verständnis werden die Grundlagen der Kommunikation in einem verteilten System vorgestellt. Die Subjekte der Kommunikation sind Prozesse, die mit Hilfe entfernter Funktionsaufrufe mit anderen entfernten Prozessen Daten austauschen oder sie zu Aktionen veranlassen. Datenaustausch und Funktionsaufrufe können durch Nachrichten bereitgestellt werden.

2.1.1 Prozeßkommunikation im verteilten System

Ein Prozeß wird auf einem Prozessor ausgeführt und besitzt einen Zustand. Dazu zählen die Inhalte der Prozessorregister sowie der Adreßraum. Der Adreßraum eines Prozesses ist von anderen Prozessen isoliert[1].

Prozeßkommunikation kann über den gegenseitigen, koordinierten Zugriff auf Adreßräume realisiert werden. Dazu überträgt ein Prozeß Daten in den Adreßraum eines anderen, der diese Daten übernehmen kann. Um die Isolierung der Adreßräume zu überwinden, muß das Betriebssystem besondere Funktionen bereitstellen.

Befinden sich Prozesse auf getrennten Rechnersystemen, wird über Kommunikationsgeräte eine physikalische Verbindung zwischen den Systemen hergestellt, über die der Datentransport abgewickelt wird.

In Abbildung 2.1 ist die Prozeßkommunikation auf einem lokalen System (links) und in einem verteilten System (rechts) dargestellt.

[1]Sog. "leichtgewichtige Prozesse"haben keinen eigenen Adreßraum, sondern teilen sich einen Adreßraum mit anderen Prozessen. Da im Folgenden Prozeßkommunikation in einem verteilten System beschrieben werden soll, wird für zwei Prozesse generell angenommen, daß ihre Adreßräume voneinander isoliert sind.

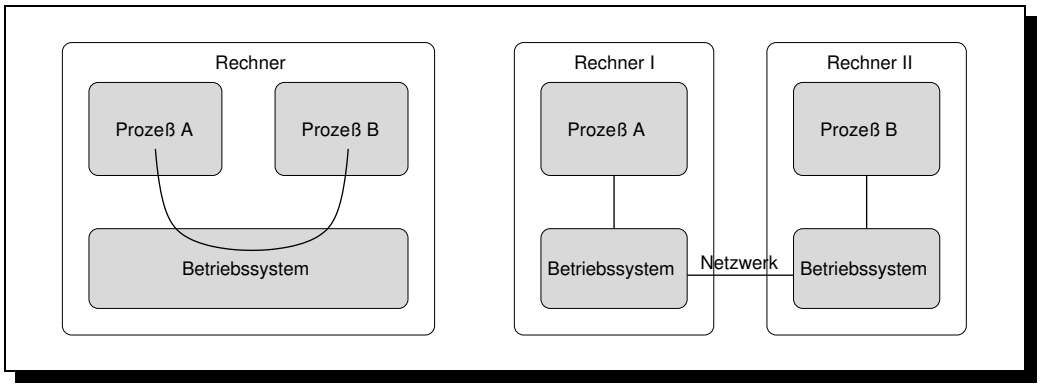

Abbildung 2.1: Lokale und entfernte Prozeßkommunikation

Kommunikationsverbindung

In Netzwerken können grundsätzlich zwei Arten der Kommunikation unterschieden werden: die *verbindungsorientierte* und die *verbindungslose* Kommunikation.

Die verbindungsorientierte Kommunikation besteht aus drei Phasen: Verbindungsaufbau, Datenübertragung und Verbindungsabbau. Zuerst wird zwischen zwei Kommunikationspartnern eine logische Punkt-zu-Punkt Verbindung hergestellt. Anschließend benutzen beide Partner diesen Kommunikationskanal zum Datenaustausch. Die Verbindung bleibt bis zum Verbindungsabbau bestehen, auch wenn zwischendurch keine Daten übertragen werden. Über eine Verbindung kommen die Daten in derselben Reihenfolge beim Empfänger an, in der sie beim Sender abgeschickt wurden. Oft besitzen verbindungsorientierte Protokolle[2] eine Fehlerkorrektur und garantieren eine korrekte Auslieferung der Daten bzw. liefern eine Fehlermeldung bei unkontrollierten Verbindungsabbrüchen. Verbindungsorientierte Kommunikation wird besonders dann eingesetzt, wenn eine zuverlässige, unterbrechungsfreie Verbindung zwischen den Partnern hergestellt werden kann, die intensiv genutzt wird.

Bei der verbindungslosen Kommunikation wird auf den Auf- und Abbau einer Verbindung verzichtet, es gibt nur eine Kommunikationsphase. Verbindungslose Protokolle[3] übertragen Daten nicht kontinuierlich über eine geschaltete Leitung, sondern in Form von Paketen, von dem jedes mit einer eigenen Zieladresse versehen ist.

2.1.2 Nachrichten

Zur Kommunikation können Prozesse Nachrichten als Informationsträger austauschen. Nachrichten sind in diesem Zusammenhang Daten, die von einem Sender verschickt werden und an einen Empfänger gerichtet sind. Zum Nachrichtenaustausch stehen zwei Basisoperationen bereit:

[2] z.B. TCP
[3] z.B. UDP

- die Sendeoperation *send(target, message)* zur Übermittlung einer Nachricht *message* an *target*
- die Empfangsoperation *receive(source, message)* zum Empfang einer Nachricht *message* von *source*, wobei *source* auch unbestimmt sein kann.

Eine Nachricht enthält eine Botschaft vom Sender an den Empfänger.

Synchronisation

Der Nachrichtenaustausch zwischen zwei kommunizierenden Prozessen kann entweder synchron oder asynchron erfolgen. Vor einer Kommunikation muß feststehen, welcher Prozeß als Empfänger und welcher als Sender auftritt.

Synchroner Nachrichtenaustausch

Bei einem *synchronen Nachrichtenaustausch* wartet mindestens einer der Prozesse (Sender oder Empfänger) auf seinen Kommunikationspartner, der wartende Prozeß ist solange in seiner Ausführung blockiert. Die verwendeten Kommunikationsoperationen sind daher *blockierende Operationen*.

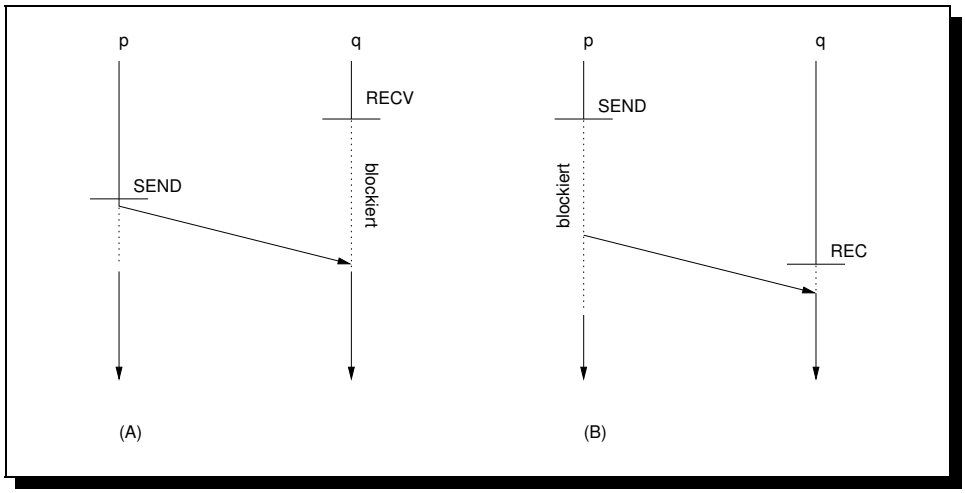

Abbildung 2.2: Synchrone Kommunikation

In Abbildung 2.2 sind zwei kommunizierende Prozesse p und q entlang einer Zeitachse dargestellt. 2.2 (A) zeigt einen synchronen, blockierten Empfänger q. Erst durch ein *send* von p wird die Blockierung von q aufgehoben. In 2.2 (B) ist ein synchroner, blockierter Sender p dargestellt. Die Blockierung kann erst nach der Empfangsbereitschaft von q aufgehoben werden. Blockierungen sind bei synchroner Nachrichtenkommunikation unvermeidlich. Oft wird jedoch für ein System entschieden, auf welcher Seite die Blockierung möglichst kurz gehalten werden soll.

Asynchroner Nachrichtenaustausch

Bei einem *asynchronen* Nachrichtenaustausch können auch *nicht-blockierende Operationen* bereitgestellt werden. Asynchroner Nachrichtenaustausch erfolgt über Puffer. Der sendende Prozeß kopiert eine Nachricht in den Puffer. Anschließend holt sich der Empfangsprozeß die Nachricht zu einem beliebigen Zeitpunkt aus dem Puffer[4]. Blockierungen können auftreten, wenn nicht genügend Pufferspeicher zur Verfügung steht. Der Puffer kann auch durch einen oder mehrere Pufferprozesse realisiert werden, der Daten blockierungsfrei in Empfang nimmt bzw. bereitstellt. Die Kommunikationspartner werden durch den Puffer(prozeß) entkoppelt.

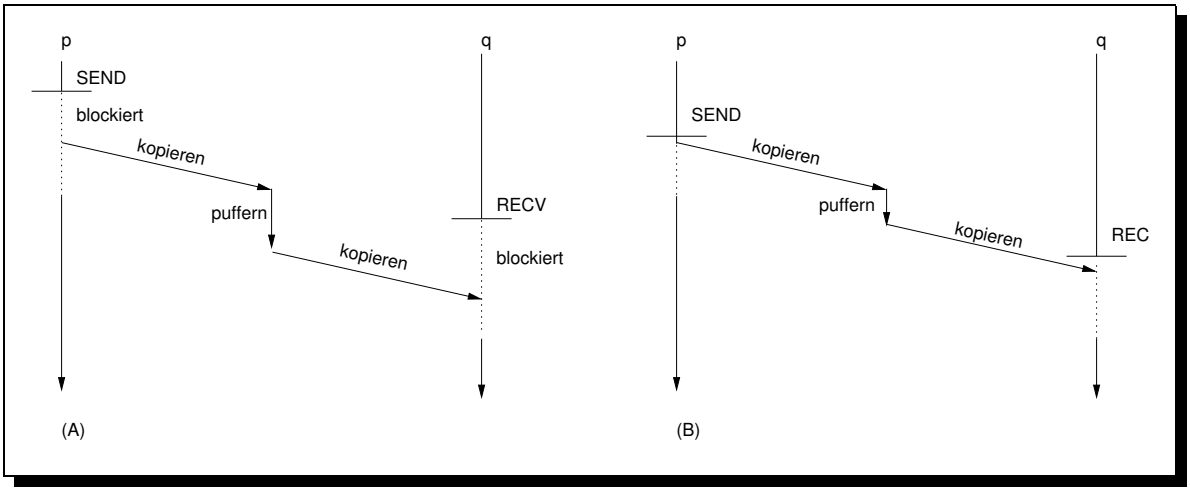

Abbildung 2.3: Asynchrone Kommunikation

In Abbildung 2.3 sind zwei Prozesse p und q dargestellt, die über einen Puffer(prozeß) miteinander kommunizieren. In 2.3 (A) findet der Nachrichtenaustausch über blockierende Operationen statt. In 2.3 (B) überträgt der Prozeß p eine Nachricht blockierungsfrei in den Puffer, Prozeß q nimmt sie blockierungsfrei entgegen.

Nachrichtenkommunikation als Grundlage für entfernte Funktionsaufrufe

Über *entfernte Funktionsaufrufe* können Prozesse Funktionen auf anderen Rechnern ausführen. Dazu muß die entsprechende Funktion auf dem entfernten Rechner vorhanden sein. Das Ergebnis der Berechnung wird an den Aufrufer zurück übermittelt.

Entfernte Funktionsaufrufe können über Nachrichtenkommunikation realisiert werden. Ein Funktionsaufruf entspricht dem Senden (*send*) einer entsprechenden Nachricht von einem Client an den Funktionsserver, der diese Nachricht empfängt (*recv*). Um ein Ergebnis zurückzuliefern, sendet der Server eine Ergebnisnachricht zurück an den Client.

[4] ähnlich dem "publish-and-subscribe"-Verfahren

Für die Nachrichtenkommunikation werden zum Aufruf entfernter Funktionen, wie in Abbildung 2.4 dargestellt, drei Dimensionen als besonders wichtig klassifiziert, die im Rahmen dieser Arbeit aufgestellt wurden.

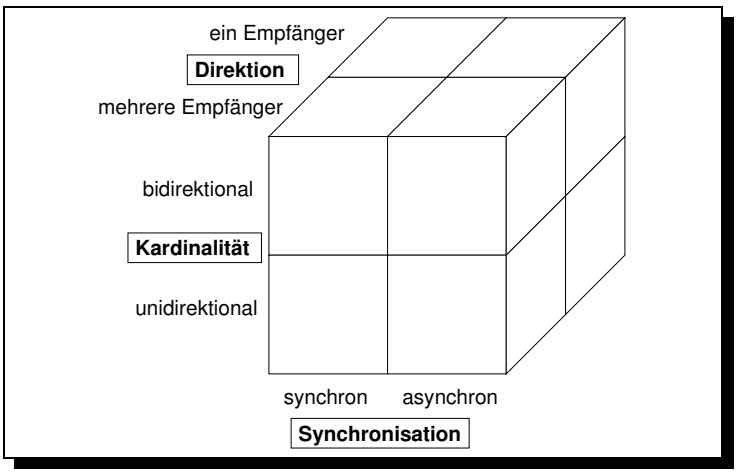

Abbildung 2.4: Dimensionen der Nachrichtenkommunikation für entfernte Aufrufe

- Synchronisation : Die Kommunikation kann synchron oder asynchron verlaufen.
- Direktion : Nachrichten können nur in eine Richtung (unidirektional) oder in beide Richtungen (bidirektional) verschickt werden.
- Kardinalität : Es kann einen oder mehrere Empfänger einer Nachricht geben.

Dabei entstehen acht verschiedene Nachrichtenmuster. Für jedes dieser Muster finden sich Anwendungsfälle; für einige Muster gibt es bekannte Realisierungen. In Tabelle 2.1 sind die fünf wichtigsten Nachrichtenmuster dargestellt.

Synchronisation	Direktion	Kardinalität	Bezeichnung
synchron	unidirektional	ein Empfänger	RPC[5] (ohne Ergebnis)
synchron	bidirektional	ein Empfänger	RPC (mit Ergebnis)
asynchron	unidirektional	ein Empfänger	Asynchroner OneWay
asynchron	bidirektional	ein Empfänger	Asynchroner RPC
asynchron	unidirektional	mehrere Empfänger	Asynchroner Multicast

Tabelle 2.1: Beispiele wichtiger Nachrichtenmuster

Marshalling

Um einen entfernten Funktionsaufruf auszuführen, werden der Funktionsname und die Parameter durch den Aufrufer in eine Nachricht verpackt. Ebenso müssen die Rückgabewerte nach der Bearbeitung in einer Nachricht an den Aufrufer übermittelt werden. Oft werden entfernte Funktionsaufrufe in heterogenen Umgebungen eingesetzt. Auf unterschiedlichen Systemen können die Parameter unterschiedlich repräsentiert sein, beispielsweise können Zahlenwerte in einer unterschiedlichen Byte-Reihenfolge im Speicher abgelegt werden[6]. Die Parameter werden daher in einem Format übertragen, das beiden Seiten bekannt ist. Der Sender konvertiert seine Parameter vor der Übertragung in dieses Übertragungsformat. Dieser Vorgang wird *Marshalling* oder *Serialisierung* genannt. Die Daten werden im Übertragungsformat an den Empfänger geschickt. Dieser dekodiert diese Daten dann in seine Repräsentation. Dieser Vorgang heißt *Unmarshalling* oder *Deserialisierung*.

2.1.3 Entfernte Funktionsaufrufe

Entfernte Funktionsaufrufe sind Protokolle, die die Implementierung verteilter Anwendungen vereinfachen. Im Unterschied zu lokalen Funktionsaufrufen wird die Zielfunktion durch einen anderen Prozeß in einem anderen Adreßraum ausgeführt. Nach dem Client-Server-Modell stellt ein Server Funktionen bereit, die von Clients aufgerufen werden können.

Bei einem *synchronen entfernten Funktionsaufruf* wartet ein Serverprozeß auf einen Funktionsaufruf durch einen Client, oder der Client wartet auf die Bereitschaft eines Serverprozesses. Ruft ein Client eine Funktion auf dem Server auf, wird der Client-Prozeß solange blockiert, bis das Ergebnis vom Serverprozeß bereitsteht. Entfernte synchrone Funktionsaufrufe können auf lokale Funktionsaufrufe in Syntax und Semantik mit großer Ähnlichkeit abgebildet werden. Für einen Aufrufer ist im Idealfall transparent, ob eine Funktion durch denselben Prozeß oder einen anderen Prozeß, auf demselben oder einem anderen Prozessor ausgeführt wird.

Asynchrone entfernte Funktionsaufrufe werden als *Asynchroner RPC (Async RPC)* bezeichnet. Beim Einsatz blockierungsfreier Kommunikationsoperationen können asynchrone RPCs blockierungsfrei ausgeführt werden. Die Programmausführung auf Client- und Serverseite verläuft hier parallel[7].

Asynchrone RPCs können dann auf lokale Funktionsaufrufe mit großer Ähnlichkeit abgebildet werden, wenn sie keine Parameter zurückliefern. Anwendungsbeispiele sind Ankündigungen ("notifications") oder Erzeuger-Verbraucher-Kommunikation ("fire - and - forget").

Asynchrone RPCs mit Parameterrückgabe unterscheiden sich von lokalen Funktionsaufrufen, da zusätzliche Operationen zur Abfrage des Ergebnisstatus und zur Übergabe des

[6] *byte order*

[7] Wird das Serverergebnis vom Client benötigt, um im Programmfluß fortzufahren, ist der Client auch dann blockiert, wenn das Ergebnis noch nicht eingetroffen ist.

Ergebnisses benötigt werden. Für das Ergebnis muß ein Pufferspeicher bereitstehen, und es muß festgestellt werden, ob dort ein Ergebnis eingetragen wurde. Wurden mehrere Anfragen an einen Server verschickt, können die Ergebnisse in einer veränderten Reihenfolge eintreffen. Der Client muß diese Ergebnisse dann selbst ordnen. Asynchrone RPCs mit Parameterrückgabe lassen sich daher nicht wie lokale Funktionsaufrufe verwenden und erfordern einen anderen Programmierstil.

Gegenüber lokalen Funktionsaufrufen besteht jedoch der wesentliche Unterschied, daß ein "call-by-reference"über physikalische Adreßraumgrenzen hinweg nicht ohne Weiteres bereitgestellt werden kann[8].

2.1.4 Objektorientierung im verteilten System

In vielen Anwendungsfällen für verteilte Systeme dürfen mehrere Clients auf einen Server zugreifen. Ein Server, der für jeden Client einen eigenen Zustand speichern kann, wird hier als *zustandsbehaftet* bezeichnet. Server, die das nicht können, sind *zustandslos*.

Entfernte Funktionen besitzen - wie ihre lokalen Vertreter - nur einen globalen Zustand. Das heißt, alle Funktionen eines Programmes/Servers können nur gemeinsame programmglobale Variablen oder andere global verfügbare Ressourcen zum Speichern ihres Zustandes nutzen. Deshalb können Funktionen nur schwer unterschiedliche Zustände bereitstellen und speichern und sind daher in der Regel *zustandslos*[9]. Viele Anwendungsszenarien setzen jedoch voraus, daß Clients auf einem Server einen Zustand besitzen.

Entfernte Klassenmethodenaufrufe

Entfernte Klassenmethodenaufrufe (RCI = "remote class invocation") sind Aufrufe auf statischen Methoden entfernter Klassen. Sie sind eine Zwischenstufe vom entfernten Funktionsaufruf zum entfernten Objektaufruf.

Ein RCI muß nicht nur eine Methode adressieren, sondern muß auch einer Klasse zugeordnet werden können. Klassen bilden für ihre statischen Variablen und Methoden einen eigenen Namensraum. Damit kann eine Klasse einen Zustand kapseln. Eine Klasse jedoch kann nur einmal in einem Serverprozeß uneingeschränkt vorkommen. Somit sind auch Klassen zur Speicherung von Zuständen nicht geeignet.

Entfernte Klassenmethodenaufrufe können in vielen Fällen entfernte Funktionsaufrufe wegen ihrer Ähnlichkeit ersetzen. Sie bieten zusätzlich die Möglichkeit, Funktionen zu gruppieren.

[8] Eine Möglichkeit besteht hier im aufwendigen Kopieren von Speicherbereichen ("call-by-copy/restore").

[9] Theoretisch kann der Zustand für einen Client entweder beim Client gespeichert und bei jedem Funktionsaufruf an den Server übergeben werden oder in globalen Datenstrukturen beim Server gespeichert werden.

Entfernte Objektaufrufe

Entfernte Objektaufrufe (ROI - "remote object invocation") sind Aufrufe von Methoden (oder Attributzugriffe) auf entfernten Objekten. Dazu muß neben der Methode das Objekt adressiert werden. Objekte kapseln eigene Variablen und können, abgesehen von Einschränkungen durch Systemressourcen, beliebig oft instantiiert werden. So können unterschiedlichen Clients unterschiedliche Objekte zugeordnet werden. Damit sind entfernte Objekte in der Lage, Zustände zu speichern. Objektmethoden sind zustandsbehaftete Funktionen und für zustandsbehaftete Anwendungsszenarien geeignet. Bevor ein entferntes Objekt aufgerufen werden kann, muß es durch einen Client oder Server erzeugt werden. Ein nicht mehr benötigtes Objekt wird zerstört.

Entfernte Klassen- und Objektmethoden können ähnlich komfortabel aufgerufen werden wie entfernte Funktionen. Eine hohe Transparenz zu lokalen Aufrufen wird erreicht, wenn für die einzelnen Funktionen, Klassen oder Objekte Proxyfunktionen, -klassen bzw. -objekte bereitstehen, die wie lokale Strukturen aufgerufen werden. Die Proxies können mit Hilfe von Generatorwerkzeugen erzeugt werden, die eine spezielle Schnittstellenbeschreibungssprache verwenden.

2.2 Softwaretechnische Grundlagen

In der Zielstellung dieser Arbeit sind verschiedene softwaretechnische Konzepte benannt worden, die zur Realisierung der Middleware-Programmfamilie notwendig sind. Diese und weitere Konzepte, die den Entwicklungsprozeß unterstützen, werden nun vorgestellt.

2.2.1 Domain Engineering

Domain Engineering[10] ist ein Teilprozeß des *Produktlinien-Engineering* (vgl. Abb. 2.5). Der Zweck des Domain Engineerings liegt darin, Wiederverwertung zwischen verschiedenen Applikationen zu erreichen. Domain Engineering ist ein systematischer Prozeß, um eine gemeinsame Kernarchitektur für die Applikationen bereitzustellen.

Zunächst soll der Begriff *Domäne* präzisiert werden.

> Eine Domäne kann als eine Menge von Problemen oder Funktionen definiert werden, die durch Applikationen aus dieser Domäne gelöst werden können (vgl. [Har02]).

Dieser Gedanke wird beim Programmfamilienkonzept später in diesem Abschnitt aufgegriffen.

Produktlinie

Die *Produktlinie* ist ein allgemeines, nicht auf die Informatik beschränktes Konzept. Eine allgemeine Definition, die auch den wirtschaftlichen Aspekt von Produktlinien anreißt, ist:

> Eine Produktline ist eine Menge von Applikationen, die eine Menge von gemeinsam zu bewältigenden Fähigkeiten und Eigenschaften teilen, um den speziellen Anforderungen eines ausgewählten Marktes zu entsprechen, und damit die Mission einer Organisation zu erfüllen (vgl. [DH00]).

Um Software-Produktlinien zu ermöglichen, wird das Produktlinien-Engineering in zwei Prozesse aufgeteilt: Das *Domain Engineerung* und das *Application Engineering*. Der Zweck des Domain Engineerings ist, wiederverwendbare Artefakte für das Application Engineering bereitzustellen. Das Ergebnis ist eine Produktlinien-Infrastruktur oder -Plattform, die die einfache und systematische Erstellung von Produkten in der Phase des Application Engineerings unterstützt.

Die beiden parallelen Prozesse sind in Abbildung 2.5 dargestellt. Der doppelt gerichtete Pfeil zwischen den Prozessen soll andeuten, daß erstens das Domain Engineering nur mit Wissen über die zukünftigen Applikationen Erfolg haben kann, und daß zweitens das Application Engineering die Artefakte des Domain Engineering verwendet. Der Engineering-Prozeß wird grob unterteilt in Analyse, Entwurf und Implementierung.

[10] *Domain Engineering* ist ein gebräuchlicher englischer Begriff. Daher wird hier das engl. *Domain* verwendet, während ansonsten die deutsche Variante *Domäne* benutzt wird.

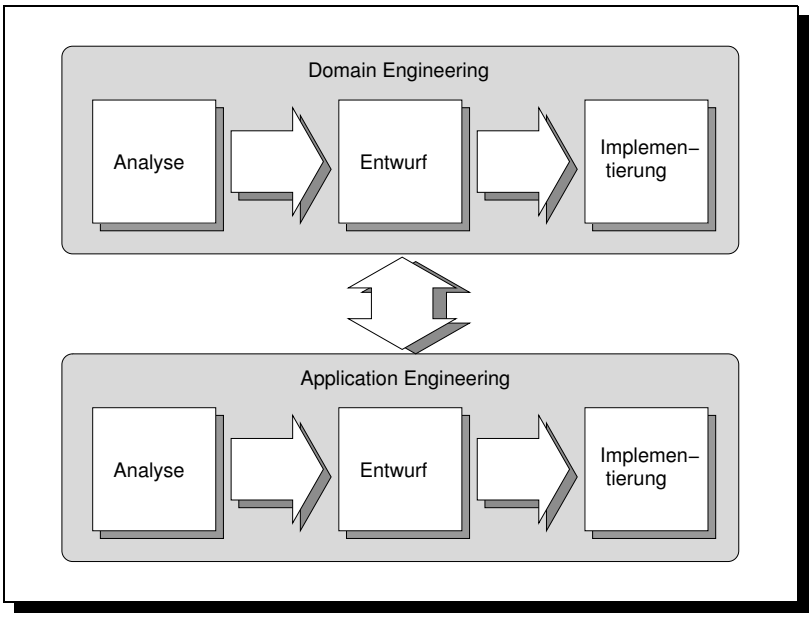

Abbildung 2.5: Phasen des Produktlinien-Engineerings

Das Domain Engineering kann mit konventionellem Software Engineering verglichen werden, wobei die einzelnen Phasen beim Domain Engineering eine Domäne betreffen und beim Software Engineering ein (einzelnes) Softwaresystem. Die Anforderungsanalyse aus dem Software Engineering besteht in der Analyse eines einzelnen Systems, während die Domänenanalyse Anforderungen für eine ganze Familie von Systemen beschreibt. Entsprechend wird bei Systementwurf und -implementierung ein einzelnes System erstellt. Domänenentwurf- und implementierung erzeugen dagegen eine wiederverwertbare Architektur und wiederverwertbare Komponenten für eine ganze Familie von Programmen [DH00].

Domänenanalyse, Domänenentwurf und Domänenimplementierung

In Abbildung 2.6 sind die Phasen des Domain Engineerings mit entsprechenden softwaretechnischen Konzepten und Methoden im Überblick dargestellt, die in diesem Kapitel genauer erklärt werden. Als zusätzliche Phasen im Engineering Prozeß wurden *Konfigurierung* und *Betrieb* eines Systems mit aufgenommen. Bei der Konfigurierung wird aus der Menge der Komponenten der Domänen-Implementierung ein System zusammengestellt (z.B. eine Bibliothek), das für die Implementierung einer konkreten Applikation benötigt wird. Die Phase *Betrieb* stellt dar, daß das Ende des Domain Engineering-Prozesses eine betriebsbereite Konfiguration ist. Im Konfigurierungsprozeß entsteht ein *maßgeschneidertes Softwaresystem*. Maßschneiderung hat das Ziel, den Ressourcenverbrauch eines Softwaresystems zu minimieren, und das System optimal seinem Zweck anzupassen.

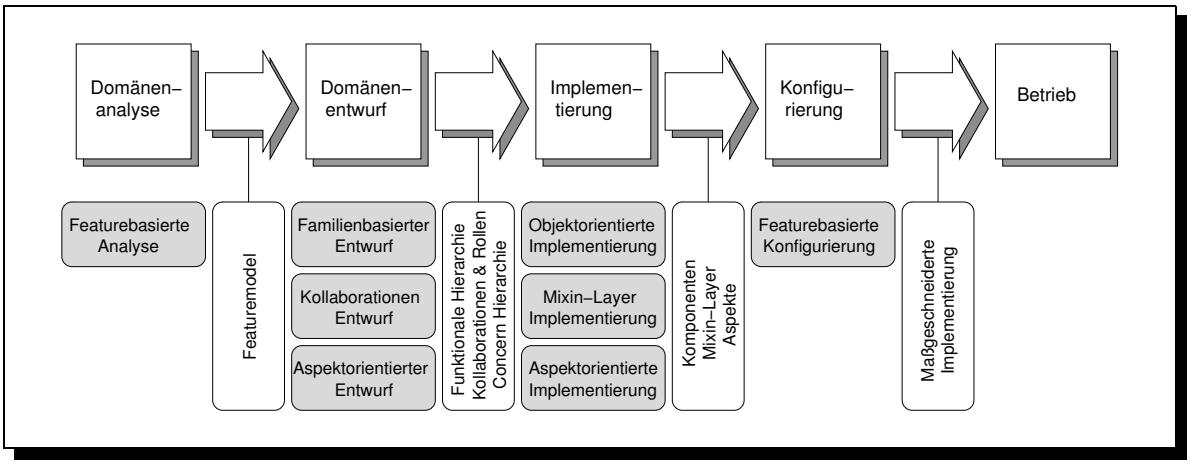

Abbildung 2.6: Domain Engineering: Abfolge und Techniken

Im Folgenden sollen die in Abbildung 2.6 benannten Techniken und Methoden für den Domain Engineering Prozeß genauer vorgestellt werden. Teilweise sind diese Techniken und Methoden durch die Aufgabenstellung vorgegeben, andere wurden zusätzlich ausgewählt.

2.2.2 Merkmalsbasierte Analyse

Eine Domäne wird definiert über eine Reihe von gemeinsamen oder unterschiedlichen Merkmalen oder Eigenschaften (engl. Features). Diese zu analysieren ist Aufgabe der Domänenanalyse (engl. Domain Analysis). Die Analyse erfordert zunächst genügend Wissen über die Domäne und eine Vorstellung über konkrete Applikationen der Domäne.

Ein Konzept zur Modellierung der gefundenen Merkmale sind die sogenannten *Merkmalsmodelle (engl. Featuremodels)* aus der merkmalsorientierten Domänenanalyse *FODA (Feature-Oriented Domain Analysis)* [KCH+90]. Mit Merkmalsmodellen kann auf einem sehr abstrakten Niveau ausgedrückt werden, welche allgemeinen und variablen Eigenschaften in ein System integriert werden können[11]. Ein Merkmal ist in FODA definiert als eine "für den Endanwender erkennbare Charakteristik des Systems" [KCH+90] [BSSP03]. Zu einem Merkmal gehört auch eine informale Beschreibung, die einem Endanwender verständlich macht, welche Charakteristik der Domäne mit diesem Merkmal ausgedrückt wird.

Merkmalsmodelle werden über *Merkmalsdiagramme* dargestellt. Merkmalsdiagramme sind gerichtete, azyklische Graphen, deren Knoten die Merkmale sind. Die Merkmale eines Systems werden so in einer Hierarchie angeordnet. Das oberste Merkmal ist die Wurzel des Graphens. Merkmale, die den gleichen Elternknoten besitzen, können in beliebiger Gruppenstärke zueinander in Beziehung stehen. Über diese Beziehung wird

[11] Merkmalsmodelle sagen nichts darüber aus, *wie* Merkmale in ein System integriert werden.

das Auftreten eines Merkmals im System in Abhängigkeit von den anderen Merkmalen derselben Gruppe modelliert. Merkmale, die einer gemeinsamen Gruppe angehören, werden an den Verbindungslinien mit dem Elternknoten über einem Kreisbogen verbunden (Abb. 2.7) Erlaubte Merkmalsbeziehungen sind:

- **Zwingend** (engl. mandatory) Ein zwingendes Merkmal tritt immer dann auf, wenn auch sein Elternmerkmal auftritt. Zwingend notwendige Merkmale erhalten einen ausgefüllten Kreis. Eine Gruppierung zwingender Merkmale ist nicht vorgesehen. Zum Beispiel besitzt jedes Merkmal "Auto" die zwingenden Merkmale "Motor" und "Getriebe".

- **Optional** (engl. optional) Das Auftreten eines optionalen Merkmals ist völlig freigestellt und ist nur dann nicht möglich, wenn das Elternmerkmal nicht vorhanden ist. Dieses Merkmal steht nicht in Konflikt mit anderen Merkmalen. Optionale Merkmale werden mit einem leeren Kreis dargestellt, eine Gruppierung ist nicht notwendig. Als optionale Sonderausstattung einer Auto-Produktlinie kann z.B. eine Standheizung angeboten werden.

- **Alternativ** (engl. alternative bzw. "xor") Aus einer Gruppe alternativer Merkmale kann nur genau eines im System auftreten. Alternative Merkmale besitzen einen ausgefüllten Kreis. Eine alternative Gruppe ist durch einen (leeren) Kreisbogen verbunden. Beispielsweise besitzt ein Fahrzeug entweder ein automatisches oder ein manuelles Getriebe[12].

- **Optional-Alternativ** (engl. optional-alternativ bzw. "or") Eine Gruppe von Merkmalen, aus der mindestens ein Merkmal im System auftreten muß, wird als optional-alternative Gruppe bezeichnet. Optional-alternative Merkmale sind mit ausgefüllten Kreisen gekennzeichnet, gruppiert werden sie mit einem ausgefüllten Kreisbogen. Ein Autohersteller kann ein Fahrzeug mit einem Verbrennungsmotor, einem Elektromotor oder mit beiden Motoren - "Hybridmotor" genannt - ausstatten.

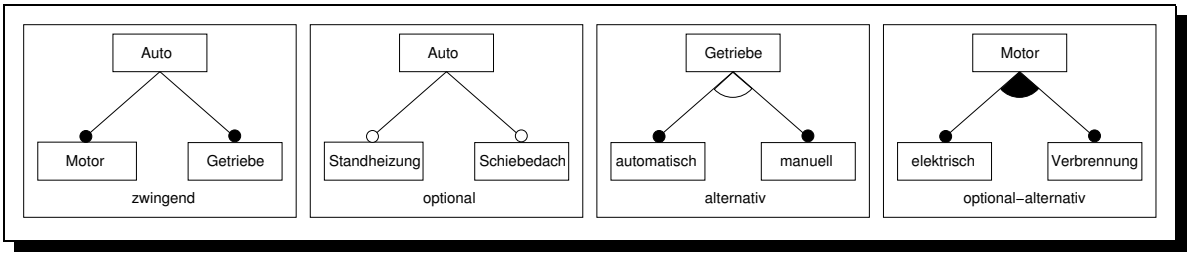

Abbildung 2.7: Beziehungen zwischen Merkmalen (angelehnt an [KCH+90])

[12]Einige Oberklassenfahrzeuge besitzen beide Getriebearten.

Ein Merkmal kann generell nur dann auftreten, wenn auch sein Elternmerkmal auftritt. Ist ein bestimmtes Merkmal in einem konkreten System nicht vorhanden, so können auch dessen untergeordnete Merkmale in dem System nicht auftreten.

Merkmalsmodelle können nur sehr einfache logische Beziehungen zwischen Merkmalen modellieren, und jedes Merkmal kann in maximal einer Gruppe vorkommen. Die tatsächlichen Beziehungen zwischen einzelnen Merkmalen sind oft viel komplexer, so daß zusätzliche Beschreibungen der Beziehungen notwendig sein können [BSSP03]. Mit dem Merkmalsmodell kann nur ausgedrückt werden, ob ein Merkmal *vorhanden* oder *nicht vorhanden* ist, andere Quantitäten außer "0" und "1" sind nicht vorgesehen. Auch Querbeziehungen zwischen Merkmalen unterschiedlicher Elternknoten können nicht modelliert werden.

Die Vorteile des Merkmalsmodells sind seine Einfachheit und Übersichtlichkeit. Soll die Domäne zu einem späteren Zeitpunkt um weitere Eigenschaften ergänzt oder verändert werden, kann auch das entsprechende Merkmalsmodell leicht angepaßt werden.

2.2.3 Programmfamilie

Eine Programmfamilie ist eine Menge von Programmen, die auf einer gemeinsamen Grundlage entwickelt wurden, und deren Gemeinsamkeiten viel deutlicher sind als deren Unterschiede [Par76]. Die *Programmfamilie* ist ein Konzept, um Software-Produktlinien zu realisieren.

Um eine Programmfamilie zu erstellen, müssen andere Methoden und Vorgehensweisen angewendet werden, als bei der Erstellung einzelner Programme. Nach David L. Parnas wird eine Programmfamilie durch sequentielle Vervollständigung erstellt [Par76]. Ausgangspunkt einer jeden Programmfamilie ist eine *minimale funktionale Basis*, die in möglichst kleinen Schritten um neue Funktionen erweitert bzw. verfeinert wird. Jede Erweiterung bildet die Basis für weitere Verfeinerungen usw. Eine Erweiterung (Schicht) nutzt nur Funktionalitäten tieferer Schichten und bietet ihre Funktionalität nur für spätere Erweiterungen an. Durch diese *Nutzt*-Relation entsteht eine sogenannte *funktionale Hierarchie*. Jede einzelne Schicht bildet eine lauffähige *virtuelle Maschine*, deren Funktionsfähigkeit und Fehlerfreiheit validiert werden kann.

Oft ist die Erweiterung einer Schicht nur für ein einzelnes Familienmitglied oder für eine Gruppe von Familienmitgliedern notwendig. Andere Familienmitglieder benötigen diese Funktionalität nicht. In diesem Fall kann die funktionale Hierarchie in mehrere Äste aufgespalten werden. Die folgenden Schichten enthalten dann für die unterschiedlichen Äste unterschiedliche Implementierungen (siehe Abb. 2.8).

Solche Entwurfsentscheidungen sollten so weit wie möglich hinausgezögert werden - also möglichst spätere Erweiterungen betreffen - da sie das Entstehen neuer Familienmitglieder behindern können. Zwei Familienmitglieder teilen soweit wie möglich gemeinsame Funktionen. Diese Forderung kann unterstützt werden, indem eine funktionale Erweiterung so gewählt wird, daß die Funktionalität der Familie mit jedem Schritt nur *minimal* anwächst. Dabei entsteht im Idealfall eine sehr feingranulare funktionale Hierarchie,

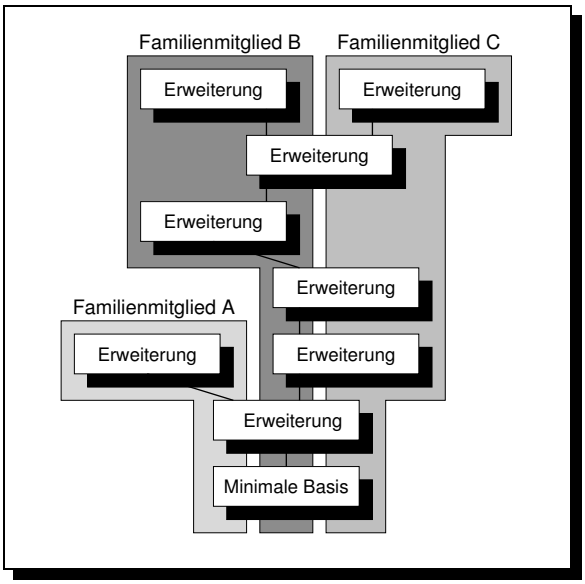

Abbildung 2.8: Funktionale Hierarchie in einer Programmfamilie (angelehnt an [Hab76])

die eine hohe Wiederverwendbarkeit von Funktionen unterstützt. Gleichzeitig wird vermieden, daß gleiche oder ähnliche Funktionen doppelt implementiert werden und die Familienstrukturen aufblähen.

Jedes Programm, das aus dieser Schichtenarchitektur abgeleitet werden kann, ist ein Mitglied der Familie. Die funktionale Hierarchie wächst, in dem neue Mitglieder hinzukommen oder bestehende erweitert werden.

2.2.4 Objektorientierung

Zum weiteren Verständnis werden nun die Konzepte *frühe und späte Bindung* vorgestellt.

Anschließend wird gezeigt, daß sich Objektorientierter Entwurf und Objektorientierte Implementierung zur Erstellung von Programmfamilien eignen, denn viele Merkmale von Objektmodellen bieten ausgezeichnete Voraussetzungen, das Konzept Programmfamilie umzusetzen [BMSP+00].

Frühe und späte Bindung Um eine Methode (oder Funktion) aufzurufen, muß einem Aufrufer die Adresse dieser Methode bekannt sein. In den meisten Fällen wird diese Adresse dem Aufrufer zur Compilierungszeit zugewiesen, dies wird als *frühe Bindung* bezeichnet. Eine früh gebundene Methode kann mit einem Prozessorbefehl direkt angesprungen werden.

Die Adresse einer spät gebundenen (*virtuellen*) Methode kann einem Aufrufer erst zur Laufzeit zugeordnet werden. Um eine virtuelle Methode anzuspringen, muß das Laufzeitsystem die Adresse zuerst aus einer *Virtuellen Funktionstabelle* heraussuchen [Str00].

Der Aufruf ist dadurch nicht nur langsamer als bei früher Bindung, sondern es wird auch zusätzlicher Speicher für die Tabelle benötigt. Für ressourcenarme Anwendungsszenarien sollten virtuelle Methoden daher sparsam eingesetzt werden.

Vererbung - Funktionale Hierarchie

Eine Hierarchie von Funktionalitäten läßt sich sehr gut über eine Hierarchie von Klassen realisieren (siehe Abb. 2.9).

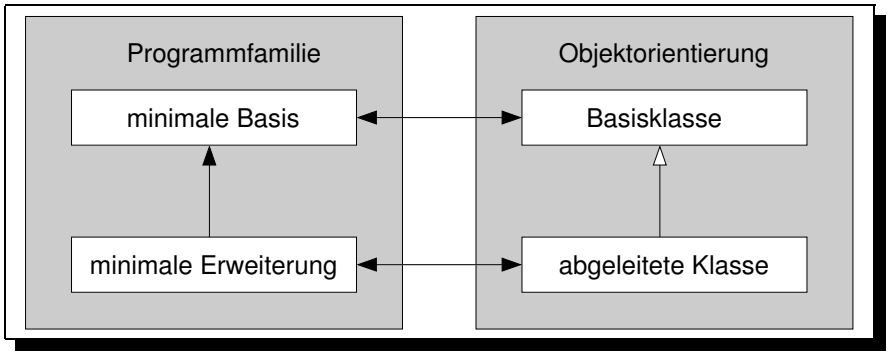

Abbildung 2.9: Programmfamilie und Objektorientierung

Jede Klasse kapselt eine minimale Funktionserweiterung, nutzt dabei die Schnittstellen der tieferen Klassen und stellt eine Schnittstelle für spätere Erweiterungen bereit. Klassen können in unterschiedlichen Ausprägungen, jedoch mit gleicher Schnittstelle auftreten. So können funktionale Erweiterungen erstellt werden, die alternativ verwendet werden. Über eine Konfigurierung der Klassenstruktur können später Familienmitglieder erstellt werden [BMSP+00].

Späte Bindung - Späte Entwurfsentscheidung

Um eine Entwurfsentscheidung zu verzögern, kann bei der Implementierung einer Programmfamilie das Problem auftreten, daß die Schnittstelle einer funktionalen Erweiterung schon früher bekannt sein muß als die Erweiterung selbst. Mit Hilfe *später Bindungen*, die von vielen objektorientierten Programmiersprachen angeboten werden, kann eine Methodenschnittstelle zu einem früheren Zeitpunkt (tiefer in der Hierarchie) definiert werden als die Methode selbst [BMSP+00].

2.2.5 Kollaborationenentwurf

Ausgehend von der Idee der funktionalen Hierarchie wird hier ein weiteres softwaretechnisches Konzept vorgestellt, das die Konfigurierbarkeit und Wiederverwendbarkeit

von Funktionen einer Programmfamilie unterstützt und erleichtert. Da die funktionalen Erweiterungen in einer Programmfamilie *minimal klein* sind (sein sollen), bestehen die Familien großer Softwareprojekte aus einer Unmenge von Komponenten [SGSP02]. Die unterschiedlichen Beziehungen zwischen diesen Bausteinen und die vielen Kombinationsmöglichkeiten erschweren die Konfiguration von Familienmitgliedern. Das Ziel des in den folgenden Abschnitten vorgestellten *kollaborationenbasierten Entwurfs* [SB02] ist es, die Zahl konfigurierbarer Komponenten zu reduzieren, die Beziehungen zwischen Komponenten zu vereinheitlichen und die Konfiguration eines Familienmitgliedes zu vereinfachen. Zusätzlich bietet diese Methode begrenzte Möglichkeiten, Merkmale zu modularisieren, die mehrere Implementierungseinheiten betreffen. Im Anschluß wird eine Implementierungstechnik für diese Entwurfsmethode vorgestellt.

Modularisierung vs. Merkmal

Modularität ist ein wichtiges Entwurfs- und Implementierungskonzept. Module kapseln einfache oder komplexe Funktionen und sollen nach Möglichkeit für andere Applikationen wiederverwendet werden. Die Entwicklung geht in eine Richtung, daß die Granularität der Module wächst: von einst kleinen zu immer größeren funktionalen Einheiten. Anfangs kapselten Module Funktionen, später Klassen und mittlerweile enthalten Module oft größere Komponenten (Cluster von Klassen) bis hin zu ganzen Bibliotheken [SB02]. Große Module haben den Vorteil, daß Familienmitglieder aus wenigen großen Einheiten leichter erstellt werden können als aus vielen kleinen, denn größere Module kapseln und verbergen Implementierungsdetails und sind leichter zu kombinieren. Größere Module enthalten jedoch in der Regel viel Funktionalität, die von einem Programm nicht genutzt wird. Damit verringern sich gleichzeitig die Erweiterbarkeit, die Flexibilität und die Möglichkeiten der Wiederverwertung eines Moduls. Dieser Widerspruch läßt sich nicht mit der Wahl einer kleineren oder größeren Modulgröße lösen, wenn von herkömmlichen Modulen als Träger funktionaler Einheiten ausgegangen wird [SB02].

Oftmals kann ein Merkmal für eine Programmfamilie nicht in einem Modul gekapselt werden, sondern das Merkmal betrifft unterschiedliche Implementierungseinheiten. Dadurch wird die Konfigurierung einer Programmfamilie erschwert, da in den Modulen Anpassungen vorgenommen werden müssen. Mit herkömmlichen Techniken lassen sich derartige Merkmale nicht modularisieren.

Y. Smaragdakis und D. Batory stellen in [SB02] ein Lösung für dieses Granularitätsproblem in der Umsetzung einer funktionalen Hierarchie vor, die als *kollaborationenbasierter Entwurf* bzw. *rollenbasierter Entwurf* bezeichnet wird. Dieses Konzept ermöglicht auch die Kapselung schwer modularisierbarer Merkmale, wie später gezeigt wird. Zentrale Begriffe des Kollaborationenentwurfs sind *Refinement*, *Kollaboration* und *Rolle*, anhand derer das Konzept erläutert wird.

Refinements

Ein *Refinement* entspricht analog zur *funktionalen Hierarchie* einer funktionalen Erweiterung eines Programmes, die einen neuen Dienst, eine neue Fähigkeit oder ein neues Merkmal einführt. Ein Refinement kann eine Funktion auch verbessern, verfeinerern oder konkretisieren. Vergleichbar mit den funktionalen Erweiterungen besteht ein Programm nach [SB02] aus aufeinander aufbauenen *Refinements*. Ein Refinement kann - wie später erklärt wird - mehrere Implementierungseinheiten betreffen. Diese Refinements werden als *Large-Scale Refinements*[13] bezeichnet.

Kollaborationen und Rollen

Im objektorientierten Entwurf sind Objekte die Kapselungseinheiten für Eigenschaften und Verhalten. Um ihre Aufgabe zu erfüllen, müssen sie mit anderen Objekten zusammenarbeiten. Die Zusammenarbeit einer Gruppe von Objekten innerhalb einer Ebene der funktionalen Hierarchie wird in [SB02] als *Kollaboration* bezeichnet. Die Interaktion innerhalb der Kollaboration findet über ein Protokoll statt, das die Menge der in der Kollaboration erlaubten Verhaltensweisen definiert. Erlaubte Verhaltensweisen zwischen Objekten sind über Schnittstellen beschrieben. Da für eine Kollaboration zwischen Objekten oft nur ein Teil einer Objektschnittstelle genutzt wird, besteht das Protokoll üblicherweise nur aus einer Untermenge der von den Objekten angebotenen Schnittstellen. Der Teil eines Objektes (das *Objektfragment*), der innerhalb einer Kollaboration für die Umsetzung des Protokolls verantwortlich ist, wird *Rolle* eines Objektes genannt. Andere Teile eines Objektes können für andere Aufgaben zuständig sein, bilden dazu Kollaborationen mit anderen Objekten und haben andere Rollen. *Rolle* und *Objektfragment* beleuchten Funktionalität von zwei Seiten. Ein Fragment enthält eine Teilfunktionalität eines Objektes. Die Rolle drückt aus, daß diese Funktionalität eine ganz bestimmte Aufgabe erfüllt, die in Abhängigkeit von den anderen Rollen/Objektfragmenten unterschiedliche Auswirkungen auf das Verhalten der Kollaboration haben kann.

In Abbildung 2.10 sind Kollaborationen als waagerechte und Objekte als senkrechte Balken dargestellt. Die Schnittflächen sind die Rollen. Objekte (hier *OA*, *OB* und *OC*) setzen sich aus Objektfragmenten bzw. Rollen zusammen. Beispielsweise besitzt das Objekt *OA* die Rollen *A1* bis *A4*. Auch die Kollaborationen bestehen aus Rollen, z.B. enthält die Kollaboration *C1* die Rollen *A1* und *B1*.

Ein Programm besteht erstens aus Kollaborationen und zweitens aus Objekten. Ein Objekt existiert nicht, bevor es nicht aus mehreren Rollen (Fragmenten) zusammengesetzt ist. Die Bindungen zwischen den Rollen eines Objektes werden beim Konfigurationsvorgang oder zur Laufzeit erzeugt. Programme wachsen bzw. verfeinern sich nicht durch die Erweiterung um Objekte, sondern durch die Erweiterung um Kollaborationen, und diese erweitern die Objekte.

[13] kann mit "Grobgranulare Erweiterungen" übersetzt werden

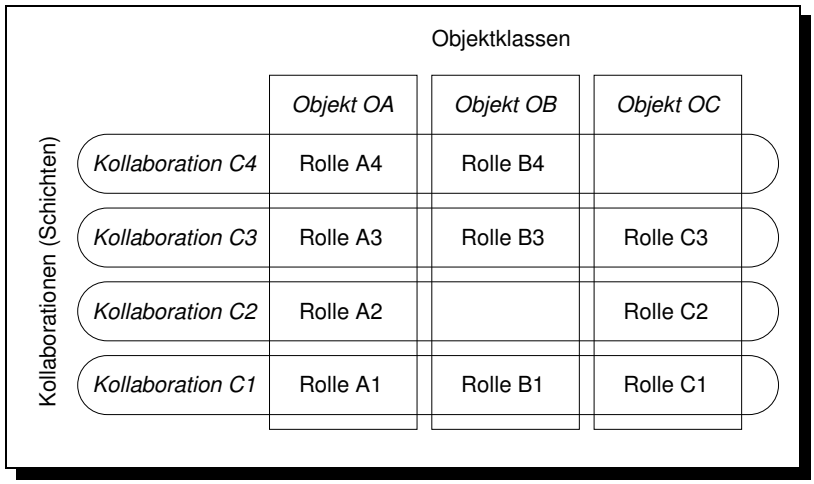

Abbildung 2.10: Kollaborationen, Objekte und Rollen (angelehnt an [SB02])

Merkmale als Module

Die Stärke von Kollaborationen für die Anforderungen an eine Programmfamilie wächst mit deren Unabhängigkeit von anderen Kollaborationen. Im günstigsten Falle haben Kollaborationen keine Abhängigkeiten nach außen, und damit ergeben sich für eine Programmfamilie viele Vorteile.

Erstens sind diese Kollaborationen leicht wiederverwendbar: überall dort, wo die Funktionalität einer Kollaboration benötigt wird, d.h. wo Objekte die Rollen der Kollaboration benötigen, können die Rollen auf die entsprechenden Objekte abgebildet werden. Idealerweise sind Kollaborationen vielfältig gegeneinander austauschbar oder optional, so wie Merkmale einer Programmfamilie austauschbar oder optional sein sollten. Dazu müssen austauschbare Kollaborationen dieselben Schnittstellen definieren.

Zweitens betreffen alle funktionalen Änderungen innerhalb einer Kollaboration nur die Kollaboration selbst und haben keine Auswirkungen auf andere Teile des Programmes.

Kollaborationen sind daher als funktionale Einheiten einer Familie geeignet und bieten gute Voraussetzungen, Merkmale und Aufgaben zu kapseln. Damit tragen sie dem Wunsch Rechnung, bei der Erstellung eines Programmes das *Was* über das *Wie* zu stellen. Kollaborationen erfüllen die Voraussetzungen von *Large-Scale Refinements*, um Merkmale über Objektgrenzen hinweg zu modularisieren.

2.2.6 GenVoca-Grammatiken

GenVoca ([BO92]) ist ein verwandter Entwurfsansatz, der sich ebenfalls mit Problemen der Modularisierung und der Wiederverwertung in der funktionalen Hierarchie von Programmfamilien sowie mit der formalisierten Beschreibung von Schichtenarchitekturen beschäftigt. Das Modell unterstützt insbesondere die Austauschbarkeit von Komponen-

ten mit gleicher Schnittstelle. Dazu bietet jede Schicht für darüberliegende Schichten eine bestimmte Schnittstelle an und nutzt nur tiefere Schichten, von denen sie eine bestimmte Schnittstelle erwartet.

Eine Programmfamilie, die aus einer solchen Schichtenarchitektur aufgebaut ist, kann durch eine reguläre Grammatik (*GenVoca-Grammatik*) beschrieben werden: Jeder Satz dieser Grammatik ist ein gültiges Familienmitglied. Beispielsweise besteht eine Familie aus zwei Schichten R und S, für die jeweils drei Implementierungsvarianten (x, y und z bzw. g, h und i) existieren, und R *nutzt* S. Eine entsprechende Grammatik ist in Abbildung 2.11 dargestellt.

```
S := x | y | z
R := g S | h S | i S
```

Abbildung 2.11: Beispiel einer GenVoca Grammatik

Mit Hilfe von GenVoca-Grammatiken kann die Konfigurierung von GenVoca-Programmfamilien vereinfacht werden.

2.2.7 Implementierung von Schichten-Architekturen

Schichtenarchitekturen nach dem Kollaborationenmodell oder nach GenVoca können als Mixin-Schichten implementiert werden [SB98] [SB02].

Mixin

Ein *Mixin* ist ein Klassenfragment, das eine unbekannte Basisklasse über eine Vererbungsbeziehung erweitert. Die Basisklasse (Superklasse) wird über einen Parameter festgelegt (parametergesteuerte Vererbung) [SB02]. Mixins werden daher auch *abstrakte Subklassen* genannt, da die Implementierung von einer Superklasse ausgeht, von der sie erst zu einem späteren Zeitpunkt abgeleitet wird [SB02]. Der Begriff ist angelehnt an das objektorientierte Konzept der *abstrakten Superklasse*, deren dynamisch gebundenen (virtuellen) Methoden erst in späteren Ableitungen der Klasse definiert werden. Mit Hilfe von Mixins kann in vielen Fällen auf den Einsatz virtueller Methoden verzichtet werden.

Mixins eignen sich als Klassenfragmente, um Rollen entsprechend dem kollaborationenbasierten Entwurf zu implementieren. Sie können in C++ mit *Templates* [Str00] ausgedrückt werden[14] (siehe Abb. 2.12).

Um die Übersichtlichkeit und Konfigurierbarkeit besonders bei großen Hierarchien zu verbessern, kann eine konkrete Mixin-Hierarchie als Typ definiert werden. In Abbildung

[14]Für Java gibt es ebenfalls Möglichkeiten, parametrisierte Vererbung und verschachtelte Klassen ähnlich C++ bereitzustellen [AFM97].

```
1  template <class Base>
2  class Mixin : public Base {
3    ...
4    ...
5  };
```

Abbildung 2.12: Mixins in C++

2.13 wird gezeigt, wie Mixins genutzt werden können, um flexibel Objekttypen zu erstellen. Die Typen `myObject1` und `myObject2` unterscheiden sich in der Basisklasse von `Mixin_D`, die hier entweder `Mixin_K` (Zeile 3) oder `Mixin_I` (Zeile 4) sein kann. Später werden diesen Typen per Typendefinition Bezeichner zugeordnet (Zeilen 8-9), über die die Objekte `myObject3` und `myObject4` instantiiert werden (Zeilen 13-14).

```
1  /** Mixin-Instantiierung bei Objekt-Instantiierung **/
2
3  Mixin_D<Mixin_K<Mixin_B<Mixin_A>>>>  myObject1;
4  Mixin_D<Mixin_I<Mixin_B<Mixin_A>>>>  myObject2;
5
6  /** oder konkrete Mixin-Instantiierung als Typ ... **/
7
8  typedef Mixin_D<Mixin_K<Mixin_B<Mixin_A>>>>  DKBA_Type;
9  typedef Mixin_D<Mixin_I<Mixin_B<Mixin_A>>>>  DIBA_Type;
10
11 /** ... vor der Objekt-Instantiierung **/
12
13 DKBA_Type myObject3;
14 DIBA_Type myObject4;
```

Abbildung 2.13: Mixin-Hierarchie in C++

Mixin-Schichten

Da Mixins parametrisierte Klassen(fragmente) sind, können sie auch wie Klassen geschachtelt werden. Eine äußere Klasse enthält die Deklaration und Definition innerer Klassen [Str00]. Bei der Kapselung von Mixins werden die gekapselten Mixins als *innere Mixins* und die kapselnden als *äußere Mixins* bezeichnet. Ein äußeres Mixin wird als *Mixin-Schicht (engl. mixin layer)* bezeichnet, wenn der Parameter (die Superklasse) dieser Schicht alle Parameter der inneren Mixins bereitstellt (deren Superklassen) [SB02]. Dieser Sachverhalt ist in Abbildung 2.14 dargestellt. Die Superklasse von Mixin-Schicht *M2* ist die Mixin-Schicht *M1*. *M1* kapselt die drei Mixins *A*, *B* und *C*. *M2* erweitert zwei Klassen aus *M1* durch Vererbung (*A* und *B*), übernimmt eine Klasse durch Vererbung (*C*) und fügt eine neue innere Klasse hinzu (*D*). Die Vererbung findet auf zwei Ebenen statt. Erstens kann eine Schicht innere Klassen von der darüberliegenden Schicht erben.

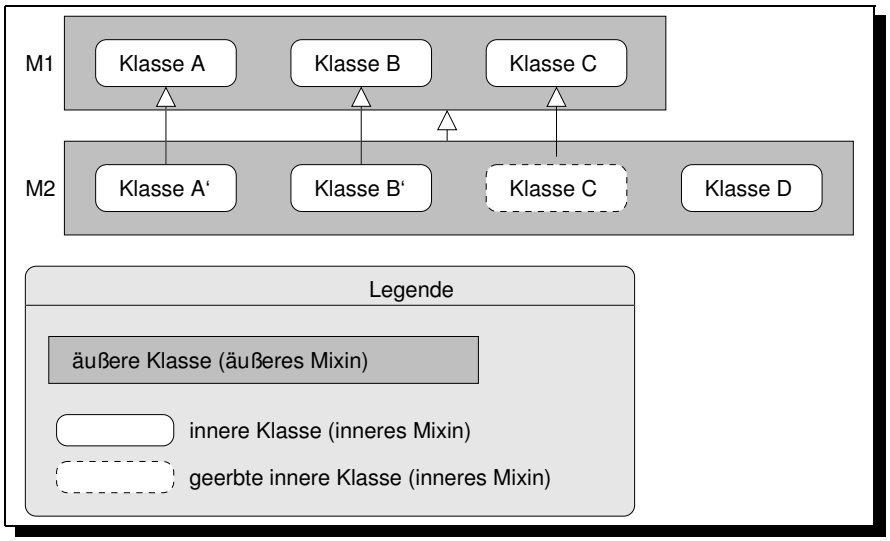

Abbildung 2.14: Mixin-Schichten und innere Mixins

Zweitens können innere Mixins Objektvariablen, Methoden oder andere Klassen von ihrer Superklasse erben.

Implementierung in C++

Die Implementierung von Mixin-Schichten in C++ wird in Abbildung 2.15 am Beispiel beschrieben. Wie in der Abbildung zu sehen ist, spezifiziert der Parameter *BaseLayer* der Mixin-Schicht *SubLayer* (Zeile 1) nicht nur die Basisschicht, sondern automatisch auch die Basisklassen innerer Mixin-Klassen (Zeilen 4-5). Das Beispiel orientiert sich an Abbildung 2.14. *SubLayer* kann mit den Mixin-Schichten *SuperLayer1* (Zeile 9) und *SuperLayer2* (Zeile 16) parametrisiert werden. Dabei werden zwei verschiedene Applikationen *foo* und *bar* instantiiert (Zeilen 22-23).

Mixin-Schichten eignen sich nach [SB02] zur Implementierung von Kollaborationen. Eine Kollaboration kann durch Mixin-Schichten, Rollen können durch innere Mixins implementiert werden. Die Zusammenarbeit zwischen Rollen entspricht der Zusammenarbeit innerer Mixins. Das Protokoll innerhalb einer Mixin-Schicht wird durch die Schnittstelle der inneren Mixins definiert. Die nach außen exportierte Schnittstelle einer Schicht ist die Gesamtheit aller nicht-privaten Klassen, Methoden und Variablen einer Schicht. Mixin-Schichten sind gegeneinander austauschbar, wenn sie dieselbe Schnittstelle implementieren. Objekte bilden sich durch Vererbungsbeziehungen zwischen den Mixin-Schichten heraus.

Mixin-Schichten implementieren funktionale Erweiterungen ihrer Basisschicht, und sie können mehrere Klassen gleichzeitig erweitern. Eine Mixin-Schicht erfüllt somit die Voraussetzungen, als *Large-Scale Refinement* einer Basis-Mixin-Schicht bezeichnet zu werden [SB02].

```
 1  template <class BaseLayer>
 2  class SubLayer : public BaseLayer {
 3    public:
 4      class A : public BaseLayer::A { ... };
 5      class B : public BaseLayer::B { ... };
 6      class D { ... };
 7  };
 8
 9  class SuperLayer1{
10    public:
11      class A { ... };
12      class B { ... };
13      class C { ... };
14  };
15
16  class SuperLayer2 {
17    /* SuperLayer2 besitzt dieselben Mixins wie SuperLayer1 */
18  };
19
20  /** zwei verschiedene Applikationen **/
21
22  SubLayer<SuperLayer1>> foo;
23  SubLayer<SuperLayer2>> bar;
24
25  /** eine Mixin-Hierarchie wird typisiert **/
26
27  typedef SubLayer<SuperLayer1>> foo_Type;
28  typedef SubLayer<SuperLayer2>> bar_Type;
```

Abbildung 2.15: Implementierung von Mixin-Schichten in C++

Mixin-Schichten sind die Implementierungskomponenten des kollaborationenbasierten Entwurfs. Ein Applikation kann als Kombination mehrerer, weitgehend unabhängiger Mixin-Schichten formuliert werden, wie in Abbildung 2.15, Zeilen 22-28 dargestellt ist.

2.2.8 Aspektorientierung

Ziel der Aspektorientierung ist die Verbesserung der Wartbarkeit und Wiederverwendbarkeit insbesondere von Programmfamilien. Mit aspektorientierten Techniken wird versucht, modulübergreifende Merkmale - sogenannte *Crosscutting Concerns* - zu modularisieren [SGSP02].

Crosscutting Concern

Komplexe Softwaresysteme sind nur durch Modulbildung beherrschbar. Modularisierung unterstützt die Konfigurierbarkeit, die Wartbarkeit, die Wiederverwendbarkeit und hilft bei der Fehlervermeidung. Viele Merkmale eines Systems lassen sich mit herkömmlichen Techniken nicht modularisieren. Sie treten an vielen unterschiedlichen Stellen im Sys-

tem(code) auf und werden daher als *Crosscutting Concern*[15] bezeichnet [KLM+97]. Crosscutting Concerns beschreiben häufig nicht-funktionale Eigenschaften eines Systems. Einen Crosscutting Concern zu beherrschen bedeutet, in mehreren Implementierungseinheiten eines Systems simultan und konsistent Anpassungen vornehmen zu können.

Es gibt mehrere Ansätze, Crosscutting Concerns zu modularisieren. Insbesondere typische Aspektsprachen wie in [SGSP02] vorgestellt, aber auch der vorgestellte kollaborationenbasierte Entwurf sowie Templateprogrammierung bieten im Allgemeinen Möglichkeiten zur Modularisierung [SGSP02].

Die Kollaborationen im Kollaborationenentwurf setzen sich aus Rollen zusammen, die nichts anderes sind als Objektfragmente. In objektorientierten Programmiersprachen sind Crosscutting Concerns nicht modularisierbar, da sie objektübergreifend auftreten. Im Kollaborationenentwurf können Crosscutting Concerns modularisiert werden, wenn es gelingt, die einzelnen Bestandteile eines Crosscutting Concerns in den daran beteiligten Objekten als Objektfragmente zu kapseln, und die entstehenden Rollen zu einer Kollaboration zusammenzuführen, so daß sich ein Crosscutting Concern in einer oder in mehreren Schichten konzentriert.

> Ein typischer nicht-funktionaler Crosscutting Concern sind Sperrstrategien für kritische Abschnitte in einer vielfädigen Umgebung. Je nach Größe der kritischen Abschnitte müssen an sehr vielen unterschiedlichen Stellen im System Veränderungen vorgenommen werden, wenn die Sperrstrategie geändert werden soll. Diese Stellen werden als *Verbindungspunkte (engl. join point)* bezeichnet. Eine feingranulare Sperrstrategie verkleinert zwar die kritischen Abschnitte, definiert dafür mehr Abschnitte und damit mehr und andere Verbindungspunkte als eine grobgranulare Strategie. Einen Crosscutting Concern wie die Sperrstrategie von Hand zu implementieren, ist unwirtschaftlich, birgt viele Fehlerquellen und schränkt Wiederverwendbarkeit und Erweiterbarkeit ein.

Aspektorientierte Programmierung

Ziel der *aspektorientierten Programmierung* ist die Modularisierung von Crosscutting Concerns. Diese Module werden als *Aspekte* bezeichnet in Abgrenzung zu den Code-Fragmenten, die die funktionalen Eigenschaften kapseln und *Komponenten* genannt werden.

In Abbildung 2.16 ist ein Software-System dargestellt (links), das einen Crosscutting Concern (grau) enthält, der sich über mehrere Quelltexteinheiten erstreckt. Eine Modularisierung wird durch Herauslösung und Separierung des Concerns in einem Aspekt erreicht (rechts).

[15] Crosscutting Concern (engl.) = modulübergreifendes Anliegen

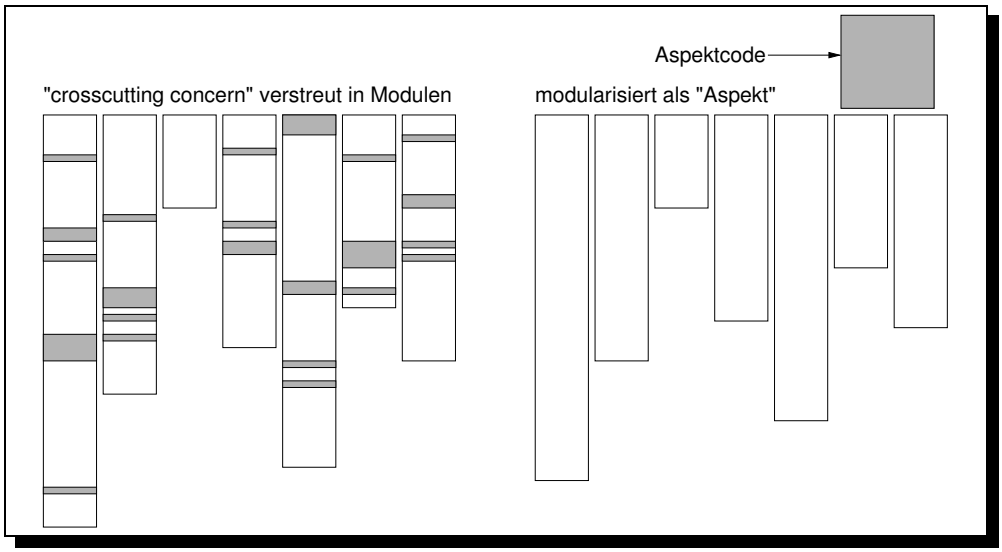

Abbildung 2.16: Modularisierung eines Crosscutting Concerns als Aspekt [SGSP02]

Typische Aspektsprachen sind entweder problemspezifische Sprachen oder Erweiterungen bestehender Programmiersprachen[16] [SGSP02]. Sie bieten sprachliche Mittel, Aspekte zu definieren und zu modularisieren. Zum Beispiel können unterschiedliche Sperrstrategien für kritische Abschnitte in Aspekten gekapselt werden, ohne den Komponentencode zu verändern. Innerhalb eines Aspektes ist definiert, an welchen Stellen im Komponentencode welcher *Aspektcode* eingefügt werden soll. Aspektcode besteht erstens aus Anweisungen der Aspektsprache, um die Verbindungspunkte zu definieren und zweitens aus Anweisungen der Programmiersprache des Komponentencodes, um die Einfügungen und Manipulationen an den Verbindungspunkten zu implementieren [KHH+01] [SGSP02].

Das Zusammenfügen von Aspektcode und Komponentencode wird *Aspektweben* genannt und durch einen *Aspektweber* geleistet. Das Weben kann vor oder während der Programmübersetzung aber auch zur Laufzeit geschehen [SGSP02]. Dafür sind entweder Code-Transformationswerkzeuge [SGSP02] [KHH+01], spezielle Compiler oder ein Aspekt-Laufzeitsystem notwendig [PGA02] [ABE01]. Dieser Vorgang ist in Abbildung 2.17 schematisch dargestellt: Komponentencode und Aspektcode liegen getrennt vor. Ein Aspektweber fügt Komponentencode und Aspektcode zu Programmcode zusammen.

Eine verbreitete Methode ist das Aspektweben vor der Übersetzung durch ein Code-Transformationssystem [SGSP02]. Dafür ist erstens kein ressourcenintensives Laufzeitsystem erforderlich. Zweitens können die vorhandenen Compiler für den Komponentencode und für den eingewobenen Aspektcode weiter genutzt werden. Der resultierende Programmcode unterscheidet sich nicht von manuell gewobenen Code.

[16] AspectJ erweitert Java, AspectC++ erweitert C++

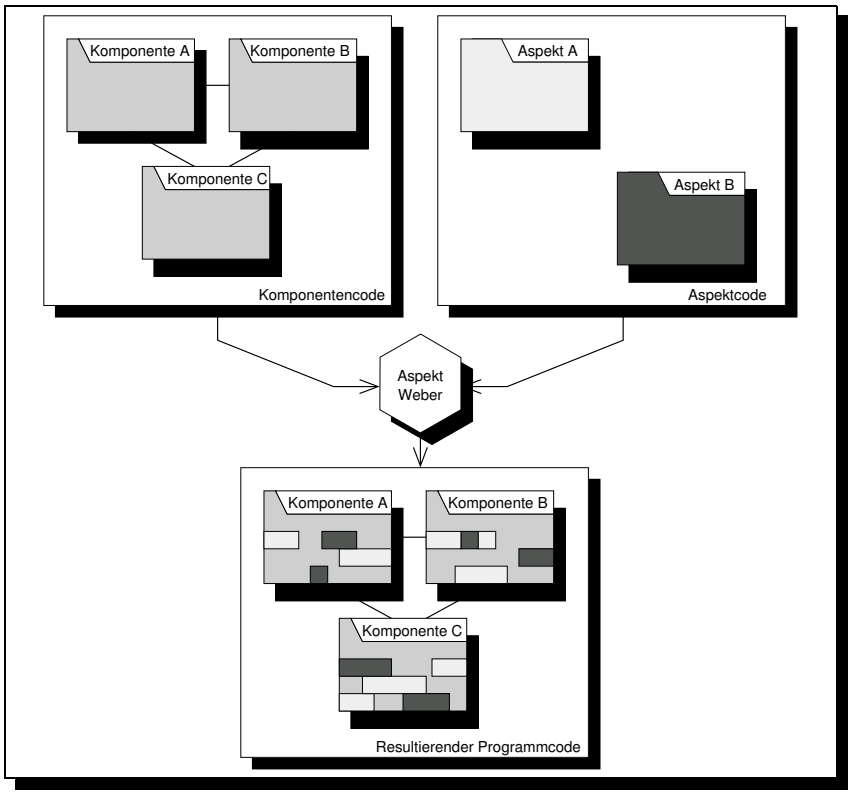

Abbildung 2.17: Zusammenweben von Komponenten- und Aspektcode

2.2.9 Merkmalsbasierte Konfiguration

Die vorgestellten Entwurfs- und Implementierungsmethoden können bei der Entwicklung einer Programmfamilie helfen, Merkmale modular umzusetzen. Im letzten Schritt müssen Module zu Programmen zusammengesetzt werden. Dieser meist sehr komplizierte Prozeß (*Variabilitätsmanagement*) ist bei großen Softwareprojekten nur mit Werkzeugunterstützung möglich [BSSP03].

Die *merkmalsbasierte Konfiguration* stellt einen Bezug zur bereits vorgestellten *merkmalsbasierten Analyse* her. In der Analyse wird beschrieben, welche Merkmale einer Domäne zugeordnet werden, und in welchen Abhängigkeiten sie zueinander stehen. Dieses Modell kann auch für den Konfigurierungsprozeß genutzt werden.

Ein Endanwender eines Systems ist in der Regel nicht daran interessiert, wie bestimmte Merkmale implementiert sind, sondern welche Merkmale verwendet werden. Die Idee der merkmalsbasierten Konfiguration ist, anhand einer (z.B. graphisch unterstützten) Auswahl von Merkmalen über einen mehr oder weniger automatisierten Prozeß ein maßgeschneidertes System zu generieren. Aus der Menge der Merkmale kann - eingeschränkt durch die Merkmalsbeziehungen - eine Auswahl getroffen und verifiziert werden. Bei der Verifizierung einer Auswahl kann auch anderes Wissen über die Programmfamilie

einfließen, z.B. semantisches Wissen (sinnlose Konfigurationen) oder andere Auswahleinschränkungen. Mit einer gültigen Auswahl wird dann aus den Bausteinen der Programmfamilie ein Familienmitglied generiert.

CONSUL

In [BSSP03] wird das System *CONSUL - CONfiguration SUpport Library)* beschrieben. CONSUL ist eine Kette von Werkzeugen, die merkmalsbasierte Konfigurierung von der graphischen Auswahl bis zur Compilierung des Programmes unterstützen.

Variabilität kann nach [BSSP03] auf unterschiedlichen Ebenen erreicht werden: Erstens kann sie auf Programmiersprachenebene in der jeweiligen Sprache ausgedrückt werden. Auf der zweiten Ebene, der *Metasprachen-Ebene*, werden Aspektsprachen und Preprozessorsprachen genannt. Die dritte Variabilitätsebene ist der Transformationsprozeß von der Hochsprache in die Maschinensprache. Alle Werkzeuge wie Compiler, Linker, Lader usw. sind konfigurierbar und damit Teil des Variabilitätsmanagements. CONSUL bietet für alle diese Ebenen Werkzeuge an.

Eine zentrale Rolle spielt das Merkmalsmodell. CONSUL verwendet eine erweiterte Version des Merkmalsmodells, mit der auch komplexere Beziehungen, wie Querbeziehungen, zwischen Merkmalen formuliert werden können. Außerdem besitzt CONSUL ein eigenes abstraktes Komponentenmodell, das eine Abbildung des erweiterten Merkmalsmodells auf Komponenten unterstützt.

Letztendlich stellt CONSUL auch ein graphisches Modellierungswerkzeug zur Verfügung, mit dem anhand des Merkmalsmodells eine Merkmalsauswahl erstellt und verifiziert werden kann. Über Transformationswerkzeuge kann anhand einer Merkmalsauswahl ein maßgeschneidertes System erstellt werden.

2.3 Mobile Middleware

Im letzten Grundlagenabschnitt wird die Domäne *Kommunikations-Middleware* näher beleuchtet. Bestehende Middleware-Lösungen, die für den mobilen Bereich entwickelt wurden, sind von besonderem Interesse für diese Arbeit, um daraus eigene Entwurfsentscheidungen ableiten zu können.

2.3.1 Middleware

Middleware ist eine Software, die einer Anwendung allgemeine, häufig benötigte Funktionen und Dienste bereitstellt. Sie befindet sich in der Systemarchitektur zwischen der Systemplattform (Hardware und Betriebssystem) und der Anwendung. Der Trend der letzten Jahre zeigt, daß zunehmend Funktionalität aus der Betriebssystemschicht in die Middleware überführt wird.

Middleware wird häufig eingesetzt, um die Erstellung und den Einsatz verteilter Applikationen zu erleichtern. Sie soll in einem verteilten System die Heterogenität der eingesetzten Hardware, der eingesetzten Betriebssysteme und Programmiersprachen überwinden und Applikationen allgemeine Kommunikationsfunktionen bereitstellen. Dazu nutzt die Middleware Funktionen der Betriebssystem- und Netzwerkschicht. Dieser Zusammenhang ist in Abbildung 2.18 dargestellt.

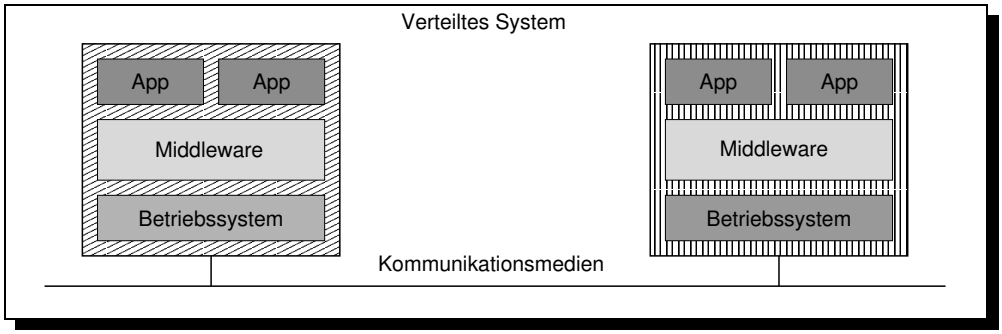

Abbildung 2.18: Klassische Middleware

In den folgenden Abschnitten wird unter einer *Middleware* diese Art von Kommunikation-Middleware für verteilte Systeme verstanden.

Bekannte Middlewarelösungen lassen sich nach [Emm00] in sieben Kategorien klassifizieren:

1. **Transaktionsorientierte Middleware**
   Die Kommunikationseinheiten zwischen verschiedenen Knoten sind Transaktionen nach dem Zwei-Phasen-Commit-Protokoll. Serviceanfragen werden in Transaktionen gekapselt, eine Transaktion kann mehrere Anfragen enthalten. Diese Lösung

eignet sich insbesondere für verteilte Datenbankmanagementsysteme. Nachteilig ist der hohe Protokolloverhead, ebenso das Fehlen eines Standards, wie serverseitig Dienste angeboten werden. Implementierungsbeispiele: IBM CICS [Hud94], BEA Tuxedo [Hal96].

2. **Nachrichtenorientierte Middleware**
   Kommunikation wird über Nachrichtenaustausch zwischen Client und Server bereitgestellt. Eine Nachricht kann eine Benachrichtigung über ein Ereignis, Daten oder eine Serviceanfrage incl. Parametern enthalten. Nachrichten werden beim Versenden bzw. Empfang in Warteschlangen (engl. Queues) verarbeitet. Client und Server sind stark entkoppelt. Beispiele: IBM MQSeries [GS96], Java Message Queue (SUN) [HBS99]

3. **Prozedurale Middleware**
   Ein Client kann auf einem entfernten Server eine Prozedur wie eine lokale Prozedur aufrufen. Werden Parameter übertragen, serialisiert die Middleware diese Parameter vor der Übertragung in eine Nachricht bzw. deserialisiert sie nach dem Empfang. Dazu werden Kommunikationsproxies verwendet, die mit einem Werkzeug aus der Schnittstellen-Beschreibung der entfernten Funktionen erzeugt werden. Der Aufrufmechanismus ist synchron. Beispiel: SUN-RPC [Sri95].

4. **Objektorientierte Middleware**
   Objektorientierte Middleware kann als Weiterentwicklung prozeduraler Middleware gesehen werden. Anstelle von Funktionen können entfernte Objekte referenziert und aufgerufen werden. Die Aufrufschnittstelle ist in der Regel statisch. Dynamisch generierte Methodenaufrufe werden von einigen Technologien unterstützt. Der Aufrufmechanismus ist in der Regel synchron. Objektorientierte Middleware integriert die meisten Fähigkeiten der bisher genannten Middlewaretypen. Beispiele: OMG CORBA [COR03], Java/RMI [SUN99], Microsoft COM/DCOM [BK96].

5. **Context Aware Middleware**
   Damit sich Applikationen an die Unterschiedlichkeiten der Geräte, der Netze und der Umgebung anpassen können, stellt eine Middleware einer Applikation ihren Kontext bereit, in dem sie ausgeführt wird. Zum Kontext können Gerätecharakteristiken, Kommunikationsparameter (Bandbreite etc.), geographische Ortsangaben und Eigenschaften des Nutzers und seiner Aktivitäten gezählt werden. Systeme mit diesen Fähigkeiten sind *Context Awared* Middleware-Systeme. Beispiele: Nexus [FKV00], Oracle iASWE [OTN00].

6. **Reflektive Middleware**
   Reflektion erlaubt es einem Programm, interne Eigenschaften explizit zu machen. Ein reflektives Programm hat Zugang zu Informationen über sich und seine Struktur und kann sich selbst verändern (*Inspektion & Adaption*) [Smi82]. Eine reflektive Middleware wird von der Applikation dynamisch rekonfiguriert und an die

Bedürfnisse der Applikation angepaßt [Cap02]. Reflektion paßt gut mit dem Konzept *Context Awareness* auf ressourcenarmen Geräten zusammen: Einer Applikation werden dynamisch nur die Middleware-Merkmale bereitgestellt, die benötigt werden, und für die ausreichend Ressourcen verfügbar sind. Beispielsysteme die das Reflektionsprinzip beinhalten: OpenORB [BCB+02], dynamicTAO [KRL+00], OpenCorba [Led99].

7. **TupleSpace-basierte Middleware**
   Eine TupleSpace-basierte Middleware entkoppelt die Teilnehmer eines verteilten Systems sehr stark in dem Sinne, daß eine Programmausführung auch dann nicht blockiert wird, wenn keine Kommunikationsressourcen zur Verfügung stehen. Die Kommunikationsmechanismen sind stark asynchron. Ein *Tuple Space* ist ein systemübergreifend verfügbarer Raum, der als Behälter für Datenstrukturen - sogenannte Tuples - dient. Auf diesen globalen Raum können Prozesse der beteiligten Systeme mit den Operationen *Lesen*, *Schreiben* und *Nehmen* auf Tupel anderer Prozesse zugreifen. Tupel sind anonym, sie werden deshalb per Mustererkennung (engl. Pattern Matching) durch Prozesse selektiert. Sender und Empfänger eines Tupels benötigen keine gleichzeitige Kommunikationsverbindung, sie können zeitversetzt auf den TupleSpace zugreifen. Beispiele: Lime [PMR99], TSpaces [WMLF99], JavaSpace [Wal98].

2.3.2 Mobile Kommunikation

Ein aktueller technischer Trend ist die Miniaturisierung von leistungsfähigen Computern und die Verbreitung von Funknetzwerktechnologie [AP03b]. Mobiltelefone und Smartphones entwickeln sich zunehmend zu mobilen Mehrzweck-Minicomputern. Die Geräte sind in ihrer Hardware und Ausstattung sehr unterschiedlich. Gemeinsamkeiten bestehen darin, daß Geräte und Netze nur geringe Ressourcen aufweisen [AP03b]. WLAN[17] und Bluetooth besitzen eine für den mobilen Bereich relativ hohe Übertragungsgeschwindigkeit und verursachen nur geringe Übertragungskosten. Ihre Verfügbarkeit ist dagegen gering. Sie werden daher vorwiegend innerhalb von Gebäuden eingesetzt. GSM und GPRS[18] haben eine geringere Bandbreite, sind störanfälliger und verursachen hohe Übertragungskosten. Sie sind großflächig in- und außerhalb von Gebäuden verfügbar. UMTS[19] soll eine hohe Übertragungsgeschwindigkeit als auch eine hohe Verfügbarkeit aufweisen, sobald das sehr aufwendige Netz ausgebaut ist.

Besonderheiten im mobilen Kontext

Verglichen mit verdrahteten Netzwerken existieren im mobilen Kontext Besonderheiten, die bei der Erstellung von verteilten Applikationen berücksichtigt werden müssen. Viele

[17] Wireless Lan
[18] GSM = Global System for Mobile Communications, GPRS = General Packet Radio Service
[19] Universal Mobile Telecommunications System

Annahmen über Geräte und Netze treffen im mobilen Kontext nicht zu [AP03b]:

- **Netztopologie**
  Standortwechsel von Knoten sind im mobilen Kontext nicht die Ausnahme, sondern die Regel. Mit dem Wechsel des Kommunikationsnetzes oder der Kommunikationszelle ändert sich auch die Netztopologie.
- **Verbindungsqualität**
  Die Verbindungen sind vergleichsweise langsam. Verbindungsunterbrechungen und Bandbreitenschwankungen treten häufig auf und müssen als Normalzustand angesehen werden.
- **Performance**
  Eingeschränkte Ressourcen wie Rechenleistung und Hauptspeicher haben eine geringere Gesamtperformance des Systems zur Folge.
- **Sicherheit**
  Verdrahtete Netzwerke können durch Firewalls etc. gegen Angreifer geschützt werden. Viele Anwendungsszenarien mobiler Geräte gehen von wechselnden Ad-Hoc-Verbindungen mit anderen Geräten aus, die ein Sicherheitsrisiko darstellen können.
- **Laufzeit**
  Mobile Geräte sind batteriebetrieben und haben eine eingeschränkte Laufzeit, die zudem vom Kommunikationsvolumen, der Prozessorbelastung u.a. Parametern abhängig ist.
- **Speicher**
  Aufgrund des geringen Primär- und Sekundärspeichers sind mobile Anwendungen stark auf externe Datenquellen angewiesen.
- **Darstellung**
  Die Darstellungs- und Eingabemöglichkeiten sind stark eingeschränkt.

Dynamischer Kontext: Awareness vs. Transparency

Damit Applikation und Middleware sich den häufig wechselnden Kommunikationsparametern dynamisch anpassen können, müssen sich beide Teilsysteme Wissen über den Kontext gegenseitig zur Verfügung stellen.

Unter einem *Kontext* werden hier alle Informationen verstanden, die das Verhalten einer Applikation beeinflussen können. Dies beinhaltet die internen Ressourcen des Gerätes wie freier Speicher, Prozessorauslastung und Bildschirmgröße, externe Ressourcen wie Bandbreite, die Netztopologie oder die benachbarten Knoten sowie Informationen über den Nutzer [MCE02]. In einem verkabelten, verteilten System ist der Kontext konstant: die Bandbreite ist gleichbleibend hoch, die Netztopolgie ist statisch, der Standort ist nicht von Bedeutung usw. [MCE02].

Middleware-Technologien für verkabelte, verteilte Systeme können daher den Kontext vernachlässigen und sich gegenüber der Applikation als "Black Box" präsentieren. Dem Anwendungsentwickler bleiben viele Merkmale verborgen, die die Verteiltheit eines Systems betreffen, denn die Kommunikationsfunktionen werden auf einem hohen abstrakten Niveau bereitgestellt. Diese Eigenschaft wird als *Transparenz (engl. Transparency)* bezeichnet. Transparenz ist in einem verteilten System ein hoher Komfort, denn nichtfunktionale Charakteristiken der Kommunikationsfunktionen müssen von einer Applikation nicht berücksichtigt werden. Beispiele für Transparenz sind *Zugriffs-, Migrations-, Replikations-* und *Ortstransparenz*. In einem ortstransparenten System kann beispielsweise vernachlässigt werden, ob eine Kommunikation innerhalb des selben Adreßraums oder zwischen zwei Adreßräumen stattfindet. Tranzparenz ist in Kabelnetzen sinnvoll, denn die Netze sind breitbandig, stabil und ausfallsicher genug, um Charakteristiken eines nicht-verteilten Systems zu simulieren.

In verteilten Systemen, die drahtlose Übertragungsmedien nutzen, ist das Gegenteil der Fall: Unerreichbarkeit, Netzausfall, Bandbreitenschwankungen und andere Ressourcenverknappungen treten im normalen Umgang auf. Im mobilen Umfeld darf der Kontext vor einer Anwendung nicht verborgen werden. Statt dessen muß die Middleware mit der Anwendung zusammenarbeiten, um das Verhalten von Middleware und Applikation dynamisch dem Kontext anzupassen. Diese Fähigkeit wird als *Awareness*[20] bezeichnet.

> Beispiel: Während einer Multimediapräsentation (Bild und Sprache) bricht die Bandbreite ein. Mit transparenten Kommunikationsfunktionen wird die Präsentation entweder in einer inakzeptablen Qualität fortgesetzt, oder sie muß abgebrochen werden.
>
> Eine angemessene Reaktion auf das Szenario kann sein, die Bilder der Präsentation nicht mehr oder nur verkleinert zu übertragen, damit der Ton verstanden werden kann. Oder es kann auf ein anderes Mobilfunknetz umgeschaltet werden. Da die Applikation mehr Informationen über den Inhalt der übertragenen Daten besitzt als die Middleware, muß sie eine Entscheidung über das weitere Vorgehen treffen.

Dienstgüte

Während in verdrahteten Netzen von annähernd konstanten Kommunikationseigenschaften ausgegangen werden kann, treffen diese Annahmen im mobilen Umfeld nicht zu. Verteilte Anwendungen oder Nutzer erwarten jedoch von einer Middleware für die Datenübertragung die Einhaltung bestimmter *Dienstgüten (engl. Quality of Service, QoS)*. Beispielsweise soll für eine Multimediapräsentation eine Mindestbandbreite bereitstehen, und bestimmte Übertragungskosten sollen nicht überschritten werden. Unter *Dienstgüte* werden alle Kommunikationseigenschaften verstanden, die die Qualität eines Dienstes

[20] Awareness (engl.) = Bewußtheit. In der Arbeit wird der englische Begriff verwendet, da die deutsche Übersetzung in der Fachwelt ungebräuchlich ist.

beeinflussen [CS99]. Dienstegütebeispiele sind in den Tabellen 2.2 und 2.3 zusammengefaßt [CS99].

Kategorie	Parameter	Beschreibung/Beispiel
Rechtzeitigkeit	Verzögerung	Zeitdauer um eine Nachricht zu übertragen
	Antwortzeit	Zeitdauer für einen Request/Response-Zyklus
Bandbreite	Systemdatenrate	Benötigte oder bereitgestellte Bandbreite in Bits oder Bytes pro Sekunde
	Anwendungsdatenrate	Benötigte oder bereitgestellte Bandbreite in einer anwendungsspezifischen Einheit pro Sekunde, z.B. Bildfrequenz für ein Video
Zuverlässigkeit	Mean time to failure (MTTF)	Durchschnittliche Ausführungszeit zwischen zwei Fehlern
	Mean time to repair (MTTR)	Durchschnittliche Zeit vom Auftreten eines Fehlers bis zu dessen Beseitigung
	Verlustrate	Verhältnis zwischen gesendeten und empfangenen Daten

Tabelle 2.2: Beispiele technologiebasierter Dienstgüten [CS99]

Eine *dienstgütefähige Middleware* ist eine Middleware, die Dienstgüte für Applikationen bereitstellen kann. Applikationen können Dienstgüten anfordern bzw. mit der Middleware aushandeln. Dabei muß die Middleware die eigenen Ressourcen berücksichtigen, d.h. die Ressourcen der beteiligten Geräte und Netze. Eine ausgehandelte Dienstgüte wird der Applikation zugesichert. Kann die vereinbarte Dienstgüte von der Middleware nicht eingehalten werden, muß die Applikation während der Übertragung darüber informiert werden (Awareness). Eine Applikation kann beispielsweise ihr Verhalten an die neue Situation anpassen, oder die Dienstgüte mit der Middleware neu aushandeln.

2.3.3 Middleware Technologien

Nachdem die Anforderungen an Middleware-Systeme für kleine mobile Geräte dargestellt wurden, werden nun vorhandene Middleware-Technologien, die ursprünglich für drahtgebundene Netze konzipiert wurden, auf ihre Eignung im mobilen Bereich untersucht.

Das Ziel dieser Arbeit ist die Verfeinerung eines groben Architekturentwurfs einer mobilen Middleware nach [AP03a]. Die Basisfunktionalität der Ziel-Middleware kann nach [Emm00] als *Prozedurale* und *Objektorientierte Middleware* klassifiziert werden.

Daher werden im Folgenden prozedurale und objektorientierte Middleware-Technologien wie CORBA[21] [Vos97], DCOM[22] [BK96], RMI[23] [SUN99] oder SUN-RPC [Sri95] auf ihre

[21] Common Object Request Broker Architecture
[22] Distributed Component Object Model
[23] Remote Method Invocation

Kategorie	Parameter	Beschreibung/Beispiel
Priorität	Priorität	Grad der Wichtigkeit für einen Nutzer z.B. von verschiedenen Medien in einem Multimedia-Stream etc.
Empfundene Dienstgüte	Bilddetail	Pixelauflösung
	Audioqualität	Audio Samplingrate
Kosten	Pro Nutzung	Um eine Verbindung aufzubauen oder auf eine Ressource zuzugreifen
	Pro Einheit	Für eine Zeit- oder Dateneinheit
Sicherheit	Vertraulichkeit	Zugang zu Informationen verhindern, z.B. durch Verschlüsselung oder Zugriffskontrolle
	Integrität	Nachweis, daß gesendete Daten nicht verändert wurden

Tabelle 2.3: Beispiele nutzerbasierter Dienstgüten [CS99]

Eignung für den mobilen Bereich untersucht.

Alle diese Systeme stellen auf Basis einer Client-Server-Architektur Programmierschnittstellen zur Verfügung, um entfernte Funktionen oder Objekte nach dem *Request/Response*-Mechanismus aufzurufen. Die von den Systemen bereitgestellten Aufrufschnittstellen für entfernte Dienste sind traditionell statisch. Zuerst werden die Schnittstellen der beteiligten Funktionen oder Objektklassen mit einer speziellen Schnittstellenbeschreibungssprache definiert. Ein Generatorwerkzeug erzeugt daraus für Client und Server Kommunikations-Proxies, die die Middleware-Funktionalität bereitstellen. Entfernte Funktions- oder Objektaufrufe greifen auf die Funktionalität der Proxies zurück. Die mit Hilfe der Proxies bereitgestellten Funktions- und Objektaufrufe haben eine sehr große Ähnlichkeit zu lokalen Aufrufen. Der Programmkontext wird bei einem Aufruf kaum berücksichtigt.

Einige Implementierungen bieten zusätzlich wesentlich flexiblere dynamische Aufrufschnittstellen an, über die Funktions- oder Objektaufrufe zur Laufzeit generiert werden können. Dynamische Schnittstellen haben den Nachteil, daß die Aufrufe wenig Ähnlichkeit zu lokalen Aufrufen haben.

CORBA

Die *Common Object Request Broker Architecture* ist eine sehr umfangreiche objektorientierte Middleware, die für stationäre Systeme und ein relativ ausfallsicheres Netz konzipiert wurde. CORBA stellt synchrone und asynchrone, statische und dynamische Aufrufschnittstellen bereit und unterstützt objektorientierte Sprachen wie C++, Java u.a. gleichermaßen [COR03]. In der *Minimum CORBA*-Spezifikation, die für ressourcenarme Geräte geeignet ist, ist die dynamische Aufrufschnittstelle nicht enthalten [COR03]. Es gibt einige schlanke ORB-Implementierungen, die hinsichtlich ihres Ressourcenver-

brauchs auch für mobile Geräte geeignet sind [SLM98]. CORBA-Systeme besitzen eine monolithische Systemstruktur und sind schwer erweiterbar. Context-Awareness und Dienstgüte werden durch den Standard nicht unterstützt. Es gibt mehrere Ansätze, ORBs[24] mit diesen nicht-funktionalen Fähigkeiten auszustatten. Projekte wie "AspectIX" [HBG+01], "Quality Objects" (QuO) [ZBS97] und "TAO" [SLM98] erweitern die Schnittstellenbeschreibung von Objekten um eine Dienstgütebeschreibung, um mit Werkzeugunterstützung nicht-funktionale Eigenschaften statisch in die Middleware zu integrieren.

RMI

Remote Method Invocation [SUN99] ist ein Java-basierter Aufrufmechanismus für entfernte Objekte. Die Verwendung von RMI setzt die Benutzung der Sprache Java sowie eine Java-Umgebung voraus. RMI ist Bestandteil der *Java 2 Standard Edition (J2SE).* Auch für die wesentlich kleinere *Micro Edition (J2ME)* existiert ein RMI-Profil[25]. Jedoch gibt es für viele kleine Gerätetypen keine J2ME-fähige "Virtuelle Java-Maschine".

RMI stellt eine statische Aufrufschnittstelle bereit, es können aber selbst unbekannte Objekte als Parameter eines Methodenaufrufes übergeben werden. Es werden synchrone und asynchrone Kommunikationsmechanismen angeboten. Das System gilt - wie Java auch - als deutlich weniger performant als andere Systeme. Auch RMI besitzt aus eine monolithische, schwer anpaßbare Systemstruktur. Dienstgütemechanismen und Context-Awareness werden nicht unterstützt.

DCOM

Das *Distributed Component Object Model* [BK96] ist die verteilte Version des proprietären COM-Modells von und (nur) für Windows und ist auch für das kleinere Windows-CE verfügbar. Die Middleware unterstützt verschiedene Programmiersprachen aus dem Windows-Bereich, z.B. Visual Basic, Visual C, Delphi. Die Architektur des Systems ist monolithisch und unflexibel, der Funktionsumfang ist sehr groß und kann nicht angepaßt werden. Dienstgüte und Context-Awareness werden nicht unterstützt.

Sun-RPC

Der RPC-Mechanismus von SUN ist sehr weit verbreitet, für viele Plattformen existieren Implementierungen. Sun-RPC unterstützt nur entfernte Funktionsaufrufe und prozedurale Programmierung in der Sprache C. Die Middleware stellt nur einen synchronen Kommunikationsmechanismus bereit, die Aufrufschnittstelle ist statisch. Auch Erweiterungs- oder Anpassungsmöglichkeiten sind nicht vorgesehen. Context-Awareness und Dienstgüten werden nicht unterstützt [Sri95].

[24] Object Request Broker

[25] auf Basis der *Connected Device Configuration*-Konfiguration (CDC)

Zusammenfassung

Die vorgestellten Middlewaresysteme lassen sich zusammenfassend wie folgt charakterisieren:

- **Allzweck Systeme**
  Die Systeme bzw. Protokolle sind sehr umfangreich. Sie sind nicht auf minimalen Ressourcenverbrauch optimiert und lassen sich in ihrer Funktionalität und ihrer Ressourcenbelegung nicht konfigurieren.

- **Monolithische Systemstruktur**
  Die monolithische Architektur ist eine schlechte Grundlage zur Erstellung maßgeschneiderter Software.

- **Transparente Kommunikationseigenschaften**
  Den vorgestellten Middlewaretechnologien ist gemein, daß sie für verkabelte Netzwerke mit leistungsfähigen Knoten entwickelt wurden. Die Systeme gehen von konstanten Verbindungsparametern wie einer statischen Netzwerktopologie und einer hohen Verbindungsqualität aus. Dienstüte und Awareness werden nicht unterstützt.

- **Proprietäre Technologie**
  Die Systeme sind mit Ausnahme des CORBA-Standards proprietär, was eine Anpassung unmöglich macht.

Herkömmliche Middleware-Technologien sind für den mobilen Bereich daher nicht geeignet [AP03b]. Einzig CORBA ist ein offener Standard und bietet eingeschränkte Erweiterungsmöglichkeiten an. Nach [AP03b] sind Forschungsansätze, vorhandene Middlewarelösungen an die Bedürfnisse im mobilen Bereich anzupassen, nicht erfolgversprechend. Daher wird nun untersucht, ob XML-Nachrichtensysteme als die Kommunikationsbasis einer prozeduralen bzw. objektorientierten Middleware für mobile Systeme bereitstellen können.

2.3.4 XML-Nachrichtensysteme

XML[26]-basierte, leichtgewichtige Nachrichtensysteme auf Basis der offenen Protokolle SOAP[27] oder XML-RPC[28] ermöglichen das Versenden und Empfangen von XML-kodierten Nachrichten über Internet-Standardprotokolle. Die übertragenen Daten werden in einem für Menschen lesbaren Format kodiert. [AP03b] enthält einen Vergleich verschiedener XML-Nachrichtensysteme hinsichtlich ihrer Eignung für eine mobile Middleware.

[26] Extensible Markup Language
[27] Simple Object Access Protocol
[28] http://www.xml-rpc.org

XML-RPC

XML-RPC ist ein einfaches RPC-Protokoll, das XML-Nachrichten zur Kodierung und HTTP[29] als Transportprotokoll benutzt. Der Standard unterstützt die wichtigsten Programmiersprachen-Datentypen, auch komplexe und nicht-typisierte Datenstrukturen können bei entfernten Funktionsaufrufen übertragen werden. Der Aufruf von Objekten ist nicht spezifiziert. Der Standard wird von sehr vielen leichtgewichtigen Implementierungen in unterschiedlichen Programmiersprachen umgesetzt.

SOAP

Der offene SOAP-Standard [SOA03] ist dem XML-RPC ähnlich aber komplexer. SOAP kann beliebige Internetprotokolle als Transportprotokolle[30] nutzen, die meisten Implementierungen nutzen HTTP. Objektaufrufe werden - auch wenn der Name das anders suggeriert - nicht unterstützt. Der Standard beschreibt einen XML-basierten Umschlag (Envelope) für die zu übertragenen Informationen sowie die Kodierung von plattform- und anwendungsspezifischen Datentypen in XML. Arrays und komplexe Strukturen können frei definiert werden. Implementierungen des SOAP-Standards existieren für sehr viele Plattformen und Programmiersprachen. Die Interoperatiblität der einzelnen Implementierungen gilt als hoch. Es existieren Transaktions- und Authentifizierungsmechanismen sowie ein Fehlerprotokoll. Verschlüsselungsmechanismen eines Transportprotokolls können genutzt werden. Viele SOAP-Implementierungen sind mit eigenem XML-Prozessor und Serverfunktionalität ausgestattet und sind trotzdem schlank.

SOAP Nachrichten Eine SOAP-Nachricht besteht aus einem *Nachrichtenumschlag (Envelope)*, der optional einen *Nachrichtenkopf (Header)* und einen zwingend erforderlichen *Nachrichtenkörper (Body)* enthält, wie in Abbildung 2.19 dargestellt ist [SOA03]. Der Header enthält Informationsblöcke, die beschreiben, wie die Nachricht vom Empfänger verarbeitet werden soll. Das können Angaben zum Routing und zur Auslieferung, über die Authentifizierung oder Autorisierung sowie Transaktionskontexte sein. Im *Body* ist die eigentliche XML-Nachricht enthalten, beispielsweise eine RPC-Kodierung [SOA03].

Diese Kodierung ist in Abbildung 2.20 dargestellt. Es sind zwei Nachrichten dargestellt: eine Anfrage (Zeilen 1-12) und eine Antwort darauf (Zeilen 15-23). Der Umschlag enthält in den Beispielen jeweils keinen Nachrichtenkopf. Es wird eine Anfrage *getStockAmount* an eine Lagerverwaltung für Textilien gestellt (Zeile 3), wieviele Einheiten eines bestimmten Artikels vorrätig sind. Als Parameter werden hier eine Produktbezeichnung (`id`) als String übergeben (Zeilen 4-6) und eine Größenangabe (`size`) als Integer (Zeilen 7-9). Die Lagerverwaltung schickt eine Nachricht über die Vorräte (`amount`) des Artikels zurück (Zeilen 15-23). Die Parameter sind typisiert und benannt. Eine Anfragenachricht wird genauso kodiert wie eine Antwortnachricht. Als Dienstbezeichner wird ein einfacher Name

[29] Hypertext Transfer Protocol

[30] Unter Verwendung bestehender Transportprotokolle können Firewalls leicht überwunden werden.

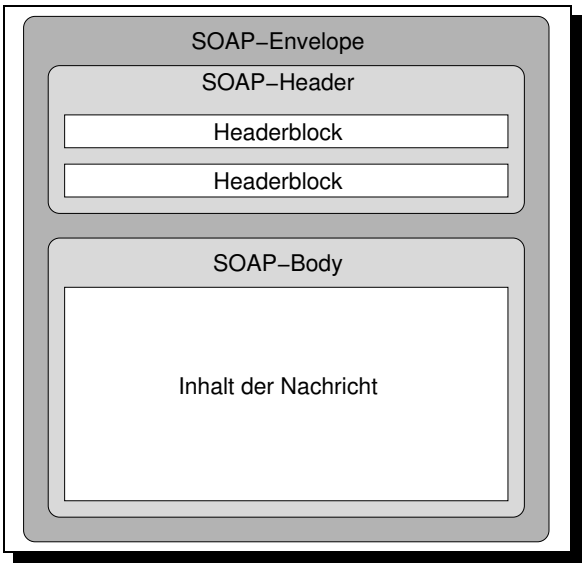

Abbildung 2.19: Schmatischer Aufbau einer SOAP-Nachricht

(hier *getStockAmount*) benutzt, der um einen Namensraum (*urn:StockService*) ergänzt werden kann (Abb. 2.20 Zeile 3). Die Nachrichten werden in ein Transportprotokoll eingebettet und verschickt.

Web-Dienste

Web-Dienste (engl. Web Services) sind im Internet verbreitete Funktionen, die über Internet-Technologie von einem entfernten Client aufgerufen werden können. Mit Hilfe der XML-basierten *Web Service Definition Language (WSDL)* können diese Dienste beschrieben werden [ABC+02]. Ziel des Standards ist die Vereinheitlichung von entfernten Funktionsaufrufen für Web-Applikationen[31]. Der WSDL-Standard ist kompatibel zum SOAP-Standard, d.h. *Web-Dienste* können per *SOAP-RPC* aufgerufen werden.

Zusammenfassung

Die XML-basierten Nachrichtensysteme weisen folgende Eigenschaften auf:

- **Leichtgewichtig**
  Schlanke Protokolle und schlanke Implementierungen prädestinieren die Systeme für den Einsatz auf ressourcenarmen Geräten.

- **Offenheit**
  Die Protokolle sind offengelegt, viele Implementierungen auch.

[31] Beispielsweise zur Standardisierung von B2B-Kommunikation (Geschäftskommunikation)

```
 1  <s:Envelope xmlns:s="http://www.w3.org/2001/06/soap-envelope">
 2      <s:Body>
 3         <ns1:getStockAmount xmlns:ns1="urn:StockService">
 4            <id xsi:type="xsd:string">
 5               blauer Strickpulli
 6            </id>
 7            <size xsi:type="xsd:int">
 8               50
 9            </size>
10         </ns1:getStockAmount>
11      </s:Body>
12  </s:Envelope>
13
14
15  <s:Envelope xmlns:s="http://www.w3.org/2001/06/soap-envelope">
16      <s:Body>
17         <ns1:getStockAmountResponse xmlns:ns1="urn:StockService">
18            <amount xsi:type="xsd:int">
19               17
20            </amount>
21         </ns1:getStockAmountResponse>
22      </s:Body>
23  </s:Envelope>
```

Abbildung 2.20: SOAP-RPC-Nachrichten: Anfrage und Antwort

- **Anpaßbarkeit**
  Die Protokolle beschränken sich auf wenige wesentliche Vorgaben wie Datentypen oder Fehlerbehandlung. Die Realisierung von Aufrufmechanismen, Aufrufschnittstelle, Context-Awareness oder Dienstgüten ist der Implementierung bzw. Erweiterungen der Implementierung überlassen.

- **Systemstruktur**
  Die Systemstruktur kann bei der Implementierung des Standards festgelegt werden. Die Architektur bestehender Systeme ist zwar schwer anpaßbar, aber zumindest liegen viele Quellen offen.

Insgesamt zeichnen sich diese Nachrichtensysteme neben ihrer oft geringen Größe durch ihre Offenheit und wegen ihrer dynamischen Aufrufschnittstellen aus, was eine Erweiterbarkeit, Wiederverwendbarkeit und Anpaßbarkeit potentiell erleichtert. Sie unterstützen jedoch keine Fernaufrufe auf Objekten. Die Integration der in keinem System vorhandenen Dienstgüte- und Context-Awareness-Eigenschaften ist bei den XML-Nachrichtensystemen nachträglich möglich [AP03a]. XML-basierte Nachrichtensysteme sollen daher als Kommunikationsgrundlage für eine mobile Middleware dienen.

Kapitel 3

Domänenanalyse

In Kapitel 2 sind grundlegende Eigenschaften der Zielsysteme beschrieben worden. Da sich die Zielgeräte deutlich in ihrer Ausstattung unterscheiden, wird eine Middleware als Programmfamilie entworfen, um später maßgeschneiderte Familienmitglieder einfach konfigurieren zu können. Der Familienentwurf soll möglichst flexibel und erweiterbar sein. In diesem Kapitel werden die Mermkale einer Middleware-Familie für mobile Systeme zunächst in einem Merkmalsmodell beschrieben. Die Merkmalsauswahl wird begrenzt auf die unterste funktionale Ebene in einem durch die Aufgabenstellung vorgegebenen Architekturentwurf [AP03a]. Wie in Abbildung 3.1 dargestellt ist, sind Funktions- und Objektaufrufe Bestandteile dieser Schicht. Dienstgüte-Merkmale werden in der Analyse nicht modelliert, sondern erst im Entwurfskapitel behandelt.

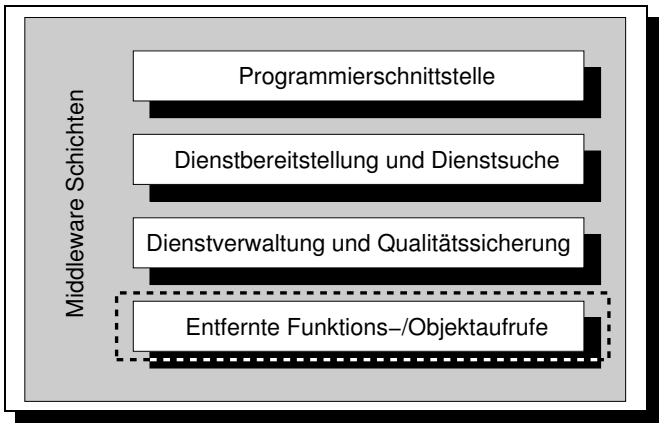

Abbildung 3.1: Architekturentwurf [AP03a]

Um die Bedürfnisse der unterschiedlichen Systeme und Applikationen optimal zu unterstützen, wird eine statische sowie dynamische Konfigurierbarkeit möglichst vieler Eigenschaften der Middleware angestrebt. Da sich diese Eigenschaft nicht mit einem Merkmalsmodell darstellen läßt, wird das Modell zunächst erweitert.

3.1 Merkmalsbindung

Ein Merkmal kann zur Compilierungszeit fest in die Middleware integriert werden (statische Konfiguration) oder bei Bedarf zur Laufzeit geladen werden (dynamische Rekonfiguration).

Statisch gebundene Merkmale werden beim Programmstart in den Hauptspeicher geladen. Auch Merkmale, die nur selten benötigt werden, belegen so zur gesamten Programmlaufzeit Hauptspeicher. Viele Betriebssysteme lagern daher selten verwendete oder ungenutzte Speicherseiten in den Sekundärspeicher aus (*Swapping*). Für die Applikationen ist dieser Vorgang transparent. Zielbetriebssysteme wie *Windows CE* und *Embedix Linux* unterstützen die Auslagerung von Speicherseiten, allerdings besitzen viele der kleinen Zielgeräte keinen oder nur sehr wenig Sekundärspeicher (Flash-ROM etc.). Beim *Swapping* liegt jede ausgelagerte Funktion zwei mal im Sekundärspeicher vor, erstens im Middleware-Programmcode und zweitens im Auslagerungsspeicher. Die Folge eines knappen Hauptspeichers ist oft ein häufiges und rechenintensives Ein- und Auslagern von Speicherseiten durch das Betriebssystem.

Dynamisch gebundene Merkmale können durch eine Applikation oder Middleware zur Laufzeit aus dem Hauptspeicher entfernt oder gegen andere Merkmale ausgewechselt werden, sobald sie nicht mehr benutzt werden. Merkmale werden erst dann geladen, wenn sie benötigt werden. *Swapping* ist für ein solches System weniger erforderlich. Jedes Merkmal liegt so maximal einmal im Sekundärspeicher und nur dann im Hauptspeicher vor, wenn es benötigt wird. Dynamische Bindungen sind notwendig, um die Variabilität von Merkmalen für speicherarme Geräte zu erhöhen, oder um Merkmale über ein Netzwerk ad-hoc zu laden. Bei dynamischer Rekonfiguration wird der Prozessor nur während der Anpassung selbst belastet.

Statische und dynamische Anpassungsfähigkeit können sich in vielen Fällen gut ergänzen: Der funktionale Kern eines Systems wird statisch, optionale Merkmale werden dynamisch konfiguriert.

Da diese Eigenschaften mit dem beschriebenen Merkmalsdiagramm nicht ausgedrückt werden können, wird das Modell erweitert: Dynamisch konfigurierbaren Merkmalen wird in Merkmalsdiagrammen im Folgenden ein Rechteck mit abgerundeten Ecken einbeschrieben (Abb. 3.3 und 3.4).

3.2 Client-Server Architektur

Um eine Kommunikation zu ermöglichen, werden die Teilnehmer als Dienstnutzer und Diensterbringer bzw. als Clients und Server aufgefaßt. Ein Diensterbringer kann gleichzeitig ein Dienstnutzer sein.

Mit Hilfe dieser Client-Server-Architektur können sowohl *Infrastrukturnetzwerke*[1] als

[1] In einem Infrastrukturnetzwerk vermittelt eine Basisstation zwischen zwei Knoten.

auch *Ad-Hoc-Netzwerke*[2] realisiert werden.

Ein Knoten kann als Client, als Server oder mit beiden Merkmalen auftreten. Um *Ad-Hoc-Peer-to-Peer*[3]-Systeme zu modellieren, besitzt mindestens einer der Peers maßgeschneiderte Serverfunktionalität. *Client* und *Server* sind die wichtigsten und gleichzeitig zwingende Merkmale einer Middleware. Wie in Abbildung 3.2 dargestellt, wird für die Merkmale trotzdem ein optional-alternativer Zusammenhang gewählt, um Clients und Server einzeln modellieren zu können.

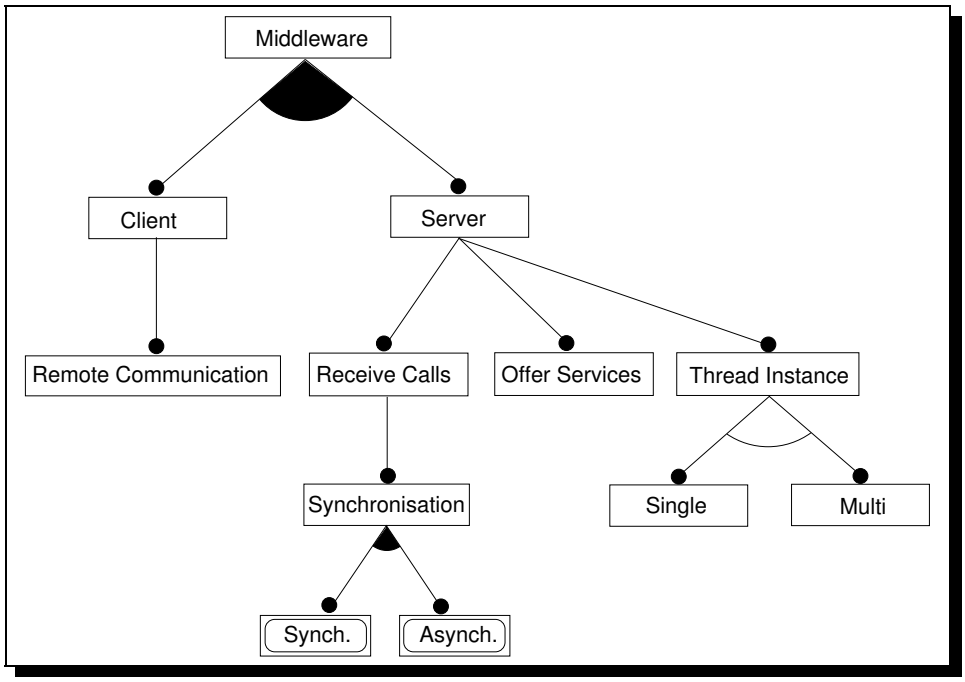

Abbildung 3.2: Server-Merkmale

3.3 Merkmale der Server

Als zwingende Merkmale eines Servers können das Merkmal *Bereitstellen von Diensten (Offer Services)* sowie das *Empfangen von entfernten Aufrufen*[4] ausgemacht werden (siehe Abb. 3.2). Letzteres besitzt das Untermerkmal *Synchronisation*. Entfernte Aufrufe können wie in Kapitel 2 beschrieben, synchron oder asynchron empfangen werden.

Ein weiteres Merkmal eines Servers ist seine Fähigkeit, mehrere Instanzen (*Threads*[5]) seiner selbst auf einem Knoten mit der nebenläufigen Entgegennahme von Aufträgen

[2]Für ein Ad-Hoc-Netzwerk wird keine Infrastruktur zwischen zwei Knoten benötigt, die Kommunikation findet direkt von Knoten zu Knoten statt.

[3]Peer (engl.) = Gleichgestellter

[4]Funktions-, Klassenfunktions-, Objekt- oder Dienstaufrufe

[5]Thread (engl.) = Faden; leichtgewichtiger Prozeß ohne eigenen Adreßraum

oder der nebenläufigen Bereitstellung von Diensten zu beauftragen. Das Merkmal *Thread Instance* (vgl. Abb. 3.2) wird mit seinen Alternativen *Single Thread* und *Multi Thread* hier nicht weiter verfeinert, da es nicht Inhalt des Systementwurfs ist. *Single Thread* entspricht einer iterativen, *Multi Thread* einer nebenläufigen Verarbeitung von Diensten. Einfache Single-Thread-Server sind für P2P-Kommunikation ausreichend. Stark genutzte stationäre Server können in einem mehrfädigen Modus arbeiten.

3.4 Merkmale der Clients

Das wichtigste Merkmal eines Clients ist der Aufruf entfernter Dienste (*Remote Communication*), welches in Abbildung 3.2 dargestellt ist. *Remote Communication* besitzt vorerst drei Untermerkmale (siehe Abb. 3.3), die den drei Dimensionen der RPC-Nachrichtenkommunikation *Synchronisation*, *Direktion* sowie *Kardinalität* (vgl. Abb. 2.4) entsprechen. Jede der Dimensionen besitzt zwei optional-alternative Untermerkmale: Ein Client kann einen Aufruf entweder *synchron* oder *asynchron* ausführen. Für beide Varianten sind unterschiedliche Ausprägungen der Aufrufe notwendig. Synchrone Kommunikationsfunktionen haben wegen ihres Aufrufkomforts und ihrer Einfachheit auch in einer mobilen Middleware eine Berechtigung, insbesondere wenn Kabelnetze oder stabile Funknetze (in einem Büro) zur Verfügung stehen. In instabilen Netzen ist das Kommunikationsfenster nur zeitweise Zeit geöffnet. Diese Periode wird für den Datenaustausch genutzt. Mit asynchronen Kommunikationsfunktionen kann ein Client einen Dienst zwar verzögert, aber blockierungsfrei aufrufen, wenn kein Netzwerkzugang besteht. Eine Middleware kann einer Applikation eine oder mehrere Synchronisierungsvarianten zur Verwendung anbieten. Um Client und Server unterschiedliche Ausprägungen dieses Merkmals zu gestatten, wird es für beide Seiten getrennt modelliert. Eine dynamisch austauschbare Synchronisationsstrategie ist von Vorteil, um bei wechselnden Netzwerkeigenschaften Dienstgüteparameter wie *Kosten* oder *Rechtzeitigkeit* optimal zu unterstützen.

Das Merkmal *Direction* gibt an, ob Aufrufe in nur eine Richtung zugelassen sind (*One Way*), oder ob auch Antworten verarbeitet werden können (*Two Way*). Für *Publish-Subscribe*- oder *Sensor-Actor*-Systeme reichen Aufrufe in eine Richtung aus. Auch für die Modellierung eines einzelnen Servers kann *OneWay*-Client-Funktionalität ausreichen. Bei einem typischen Dienstaufruf wird dagegen auch eine Antwort erwartet (*Two Way*).

Das Merkmal *Cardinality* bestimmt, ob eine Nachricht an einen oder auch an mehrere Empfänger gerichtet ist.

In dem bis jetzt beschriebenen Merkmalsmodell besitzt ein Server keine eigenen Merkmale, um einen Aufruf zu beantworten. Zu diesem Zweck kann er mit Client-Merkmalen konfiguriert werden. Ein Server, der Aufrufe beantworten soll, kann eine Antwort in einer *OneWay-SingleReceiver*-Nachricht kodieren und an den Client zurücksenden. Client-Merkmale sind für einen Server auch dann notwendig, wenn er selbst gegenüber einem anderen Server als Client auftritt.

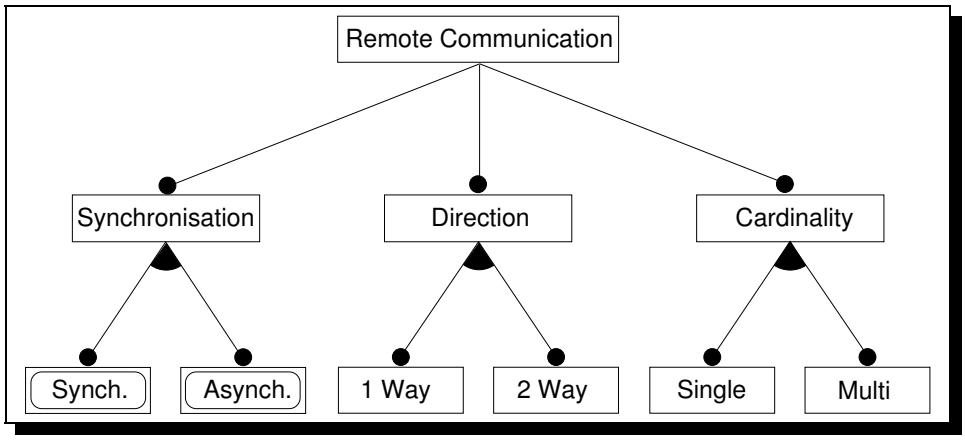

Abbildung 3.3: Client-Merkmale

3.5 Allgemeine Merkmale

Vier Merkmale, die im Folgenden vorgestellt werden, sind zwingender Bestandteil jedes Familienmitglieds und werden daher - wie in Abbildung 3.4 dargestellt - direkt unter dem Wurzelknoten des Middleware-Merkmalsmodells angeordnet.

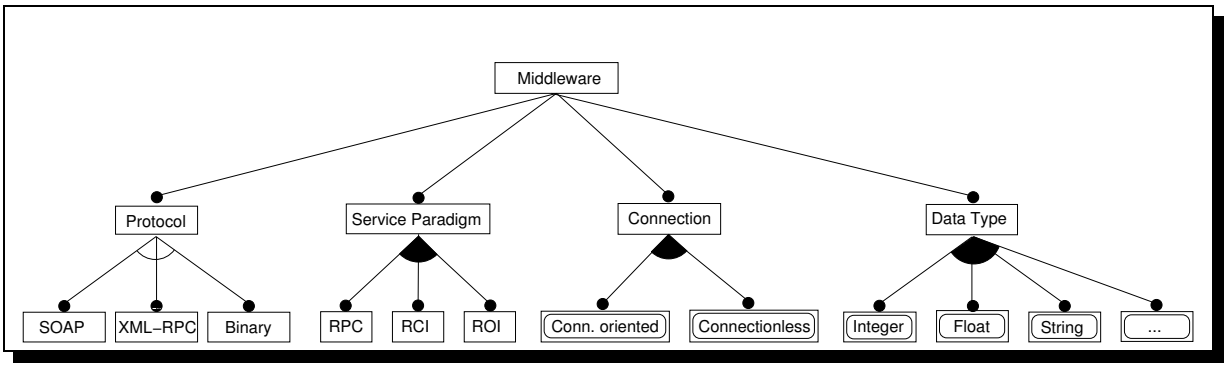

Abbildung 3.4: Allgemeine Merkmale

- **Verbindung** (*Connection*)

  Eine Kommunikation kann verbindungslos oder verbindungsorientiert stattfinden. Abhängig vom Einsatzgebiet oder vom verwendeten Kommunikationsnetz kann eine der beiden Varianten besser geeignet sein. In einem stabilen, drahtgebundenen Ethernet ist ein Verbindungsaufbau insbesondere bei häufiger Kommunikation sinnvoll. In drahtlosen Netzen sollte verbindungslose Kommunikation unterstützt werden. Eine Middleware, die in mehreren Netzen und unterschiedlichen Umgebungen eingesetzt wird, sollte beide Varianten unterstützen. Anhand des Kontext kann dynamisch eine Variante ausgewählt werden.

- **Serviceparadigma** (*Service Paradigm*)

  Das *Serviceparadigma* beschreibt, auf welcher Grundlage Dienste angeboten und aufgerufen werden können: als Funktionen und Funktionsaufrufe (*RPC*), als statische Klassenfunktionen und Klassenaufrufe (*RCI*) oder als Objekte bzw. Objektmethoden und Objektaufrufe (*ROI*). Da mindestens ein Paradigma unterstützt werden muß, ist ein optional-alternativer Zusammenhang für die Merkmalsgruppe gewählt worden. Mischformen müssen möglich sein, um beispielsweise auch rein objektorientierte Anwendungen in die Lage zu versetzen, externe *Web-Service*-Funktionen nutzen zu können.

- **Protokoll** (*Protocol*)

  Über das *Protokoll* kann ein Nachrichtensystem (*Protokollimplementierung*) für die Middleware ausgewählt werden. *Binary* steht für ein Binärprotokoll. Dahinter steht der Gedanke, für ein minimalistisches - insbesondere eingebettetes - System ein Binärprotokoll zur Dienstekommunikation zu verwenden. Bei allen Vorteilen der XML-Nachrichtenkommunikation wird ein hoher prozentualer Anteil der Kommunikationsressourcen für den XML-Overhead benötigt (vgl. Abb. 2.20).

  Das Merkmal *Protokoll* berücksichtigt nicht, daß für ein Protokoll auch mehrere Implementierungsvarianten eingesetzt werden können. In diesem Modell wird davon ausgegangen, daß für jedes Protokoll nur eine Implementierungsvariante vorliegt, und diese über das Protokoll ausgewählt wird. Die Funktionalität der Middleware ist in großem Maße vom Funktionsumfang des Protokolls und auch der Protokollimplementierung abhängig.

  In späteren Modellen kann berücksichtigt werden, Protokolle dynamisch auszutauschen, sowie mehrere Protokolle parallel zu unterstützen. Letzteres erfolgt, um eine Kopplung zwischen unterschiedlichen Protokollen zu ermöglichen. Dazu besitzt ein Knoten für jedes Protokoll eine Kommunikationsschnittstelle und kann - ähnlich einem Gateway - Nachrichten des einen Protokolls in das andere übersetzen. Protokollkoppler können in Infrastrukturnetzwerken als Zugangspunkt für mobile Geräte in ein konventionelles Leitungsnetz eingesetzt werden.

- **Datentypen** (*Data Types*)

  Das Merkmal *Datentypen* stellt optional-alternativ de/-serialisierbare Standarddatentypen bereit. Eine Middleware kennt mindestens einen Datentyp. Im Diagramm in Abbildung 3.4 sind beispielhaft einige Datentypen eingezeichnet. Weitere Typen im Sinne von Merkmalen sind beispielsweise Felder oder Strukturen. Applikationen können eigene Datentypen aus diesen Grunddatentypen ableiten. Eine Konfiguration ist hier nicht nur aus Ressourcengründen geboten. XML-RPC unterstützt weitaus weniger Datentypen als SOAP[6], die Typen eines zukünftigen Binärprotokolls sind noch nicht abzusehen.

[6]XML-RPC kennt sechs Grundatentypen sowie Felder und Verbunde. SOAP 1.2 definiert allein mehr als 20 Grunddatentypen sowie ein komplexeres Typsystem.

Zusammenfassung

Damit sind die Merkmale der Zielsysteme im Rahmen einer Domänenanalyse beschrieben worden. Das in Kapitel 2 vorgestellte Merkmalsmodell wurde erweitert, um die statische und dynamische Konfigurierbarkeit von Merkmalen darstellen zu können. Die Zielsysteme sind in Clients und Server aufgeteilt worden. Server treten als Diensterbringer, Clients als Dienstnutzer auf. Die Kommunikation findet über entfernte Funktions-, Klassenfunktions- oder Methodenaufrufe statt. Client- und Server-Merkmale werden jeweils mit allgemeinen Merkmalen kombiniert. Eine Kombination von Client- und Server-Merkmalen ist ebenfalls möglich und notwendig, um Server zu modellieren, die Anfragen beantworten sollen oder selbst als Client auftreten.

Kapitel 4

Entwurf

Die als Ergebnis der Domänenanalyse herausgearbeiteten Merkmale sollen die Grundlage für die Architektur einer Middleware-Programmfamilie bilden. Ziel des Entwurfs ist es nun, diese Merkmale mit Hilfe der in Kapitel 2 beschriebenen Techniken in leicht konfigurierbare Komponenten zu kapseln, um daraus maßgeschneiderte Einzelsysteme erstellen zu können.

Wichtige Entwurfskriterien sind die Wiederverwendbarkeit eines Merkmals innerhalb der Domäne sowie die Erweiterbarkeit der Architektur, um nach [AP03a] leicht beliebige Dienste, Dienstgüten sowie Dienstverwaltungs- und -suchmechanismen ergänzen zu können.

Beim Entwurf werden die in Kapitel 2 vorgestellten Konzepte *Programmfamilie*, *Objektorienterung*, *kollaborationenbasierter Entwurf* und der *aspektorientierte Entwurf* angewendet, deren Merkmale hier noch einmal kurz zusammengefaßt werden. Das Programmfamilienkonzept erleichtert die Erstellung maßgeschneiderter Software. Anstatt jedes System einer Domäne neu zu entwerfen und zu implementieren, unterstützt das Familienkonzept die Wiederverwertung gemeinsam genutzter Funktionalitäten. Der objektorientierte Entwurf eignet sich gut zur Umsetzung einer Programmfamilie (vgl. S. 18). Der Kollaborationenentwurf unterstützt eine noch bessere Kapselung von Merkmalen als der rein objektorientierte Entwurf, erleichtert die Konfiguration und ermöglicht die Modularisierung verschiedener Crosscutting Concerns. Die Vorgehensweise beim Entwurf ist, die analysierten Merkmale in eine funktionale Hierarchie zu bringen und daraus Schichten für den kollaborationenbasierten Entwurf abzuleiten. Die Kollaborationen bilden die wiederverwertbaren Komponenten der Programmfamilie. Innerhalb der Kollaborationen sollen Rollen identifiziert werden, die in Klassen- bzw. Objektfragmenten gekapselt werden. Für Merkmale, die sich schlecht in dieser Hierarchie kapseln lassen, wird über den Aspektorientierten Entwurf eine Modularisierung gesucht.

4.1 Überblick

Die Besonderheit der zu entwickelnden Programmfamilie ist, daß es sich eigentlich um zwei Programmfamilien handelt: die Familie der Clients und die der Server. Der Entwurf hat gezeigt, daß beide Familien eine gemeinsame funktionale Basis besitzen. Auch in der Domänenanalyse wurde festgestellt, daß Client und Server viele Merkmale teilen. Daher werden Client und Server in einer gemeinsamen Hierarchie entwickelt. In diesem Kapitel werden zuerst die Client-Kollaborationen modelliert. Anschließend wird die gemeinsame funktionale Basis definiert, und die Server-Kollaborationen werden in den Entwurf integriert. Dabei wird berücksichtigt, daß Merkmale, die nur dem Server zugeschrieben werden, von Client-Merkmalen separiert sind.

Eine besondere Stellung im Entwurf nimmt die Protokollimplementierung ein, die von Client und Server genutzt wird und in Komponenten unter einer festgelegten Schnittstelle gekapselt wird. Die Integration einer Protokollimplementierung wird nach dem Entwurf von Client und Server in einem gesonderten Abschnitt behandelt.

4.2 Die Clients

Das Merkmal eines Clients und auch dessen Aufgabe im Sinne des kollaborationenbasierten Entwurfs ist *entfernte Kommunikation*. Diese Aufgabe wird in einer Kollaborationenhierarchie nach und nach abstrahiert. Die Hierarchie ist nach dem Programmfamilienkonzept eine funktionale Hierarchie und ist in Abbildung 4.1 dargestellt. Die Schichten der Hierarchie werden im Folgenden einzeln beschrieben. Zur einfachen Bennenung der Schichten werden Abkürzungen eingeführt, die auch in Abbildung 4.1 verwendet werden.

Abstrakte Nachrichten (A-Schicht)

Zur minimalen Basis der Hierarchie gehört eine *Nachricht* mit minimalen Eigenschaften und Funktionen. Sie ist richtungslos, d.h. sie kann weder gesendet oder empfangen werden, und sie ist leer, d.h. sie kann keine Informationen wie Parameter aufnehmen. Sie besitzt einzig eine Identifikation und einen Operationsnamen und definiert eine minimale Schnittstelle für alle Nachrichten. Eine Nachricht wird durch den Operationsnamen und die Identifikation eindeutig bestimmt.

Die zweite Rolle in dieser Kollaboration ist ein Datum, das sich später selbst de-/serialisieren kann, und das im Folgenden kurz *Marshalable* genannt wird. Marshalables sollen einmal die Parameter einer Nachricht kapseln. In dieser Schicht hat ein Datum noch keinen Typ, es kann somit auch keinen Wert aufnehmen und keine Funktion ausführen. Es besitzt lediglich einen Namen und definiert eine Basisschnittstelle für alle Marshalables. Anders als für Nachrichten ist für die Marshalables die Kommunikationsrichtung schon vorgegeben: ein Marshalable kann in zwei Richtungen kommunizieren,

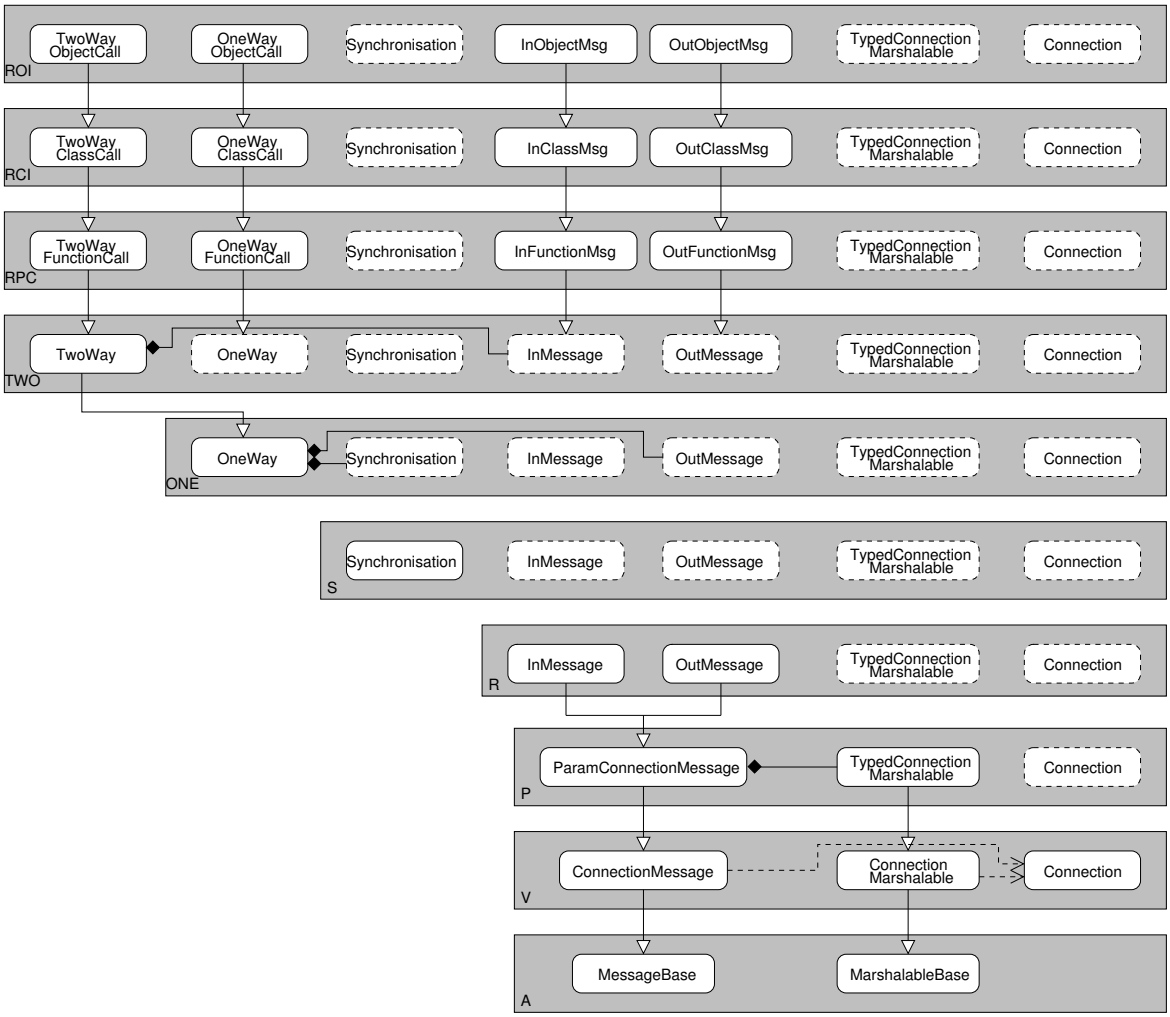

Abbildung 4.1: Funktionale Hierarchie der Clients: Von der Nachricht zum Objektaufruf

d.h. es kann sich sowohl serialisieren als auch deserialisieren[1].

Zwischen den Rollen findet noch keine Zusammenarbeit statt. Die Schnittstelle der Schicht bietet allgemeine Basisfunktionen für Nachrichten und Marshalables wie z.B. Schreib- und Lesezugriffe auf Operation und Identifikation einer Nachricht und den Namen eines Marshalables.

Nachrichten mit Verbindung (V-Schicht)

Die zweite Schicht erbt die beiden Rollen der minimalen Basis und führt die Rolle *Verbindung* ein, um Nachrichten mit einer Verbindung auszustatten. Unter einer Verbindung soll keine Verbindung im Sinne verbindungsorientierter Kommunikation verstanden werden, sondern es handelt sich um das verbindungsorientierte oder verbindungslose Merkmal. Die Verbindung kennt einen Endpunkt für den Adressaten. Deren Schnittstelle besteht u. a. aus Methoden zum Öffnen und Schließen einer Verbindung. Ein verbindungsorientiertes Merkmal baut über diese Funktionen eine Verbindung zum Endpunkt auf oder ab. Für verbindungslose Kommunikation wird über diese Instanz nur eine Zuordnung zwischen Client und Server hergestellt. Ein solches Merkmal simuliert ein verbindungsorientertes Merkmal und präsentiert seine Verbindung ständig als "geöffnet". Die Kommunikationsschnittstelle für verbindungsorientierte und verbindungslose Kommunikation ist somit identisch, das Merkmal ist daher leicht austauschbar.

Die Rollen Nachricht und Marshalable werden erweitert, so daß ihnen eine Verbindung zugewiesen werden kann. In Abbildung 4.1 ist diese Assoziation mit einer gestrichelten Linie dargestellt. Ein Verbindungsobjekt kann mehrmals bzw. von unterschiedlichen Nachrichten nacheinander verwendet werden. Es ist z.B. denkbar, daß eine Verbindung später einmal gleichzeitig von unterschiedlichen Nachrichten benutzt wird. Die Unterstützung eines Verbindungsmerkmals ist auch abhängig von der verwendeten Protokollimplementierung.

Da sich Marshalables selbst de-/serialisieren, müssen sie eine Verbindung kennen. Ihre Schnittstelle wird daher um die Kommunikationsfunktionen *marshal* und *unmarshal* erweitert. Weil das Marshalable in dieser Schicht noch keinen konkreten Typ/Wert hat, handelt es sich bei diesen Funktionen nur um eine Schnittstelle.

Unterschiedliche Verbindungsarten werden mit Hilfe unterschiedlicher Kollaborationen konfiguriert. In Abbildung 4.2 ist in A eine Kollaboration dargestellt, die verbindungsorientierte Kommunikation unterstützt. In B können alternativ zwei Verbindungsarten verwendet werden. Jedoch muß die Verbindungsart für eine Nachricht zur Compilierungszeit festgelegt sein, d.h. zwischen den Methoden der Nachricht(en) und einer Verbindungsart besteht eine frühe Bindung. In Kollaboration C ist die Verbindungsart zusätzlich zur Laufzeit austauschbar. Es wird ein Vermittler benutzt, der zwischen mehreren Verbindungsarten zur Laufzeit umschalten kann. Dazu sind späte Bindungen zwischen Nach-

[1] In einem späteren Entwurf soll noch einmal überprüft werden, ob die Richtung eines *Marshalables* schon zu diesem frühen Zeitpunkt als universal eingeführt werden soll. Ein *Marshalable* kann auch erst einmal als richtungslos gelten und wird später erst - wie die Nachricht - mit einer Richtung spezialisiert.

richten und Verbindung notwendig.

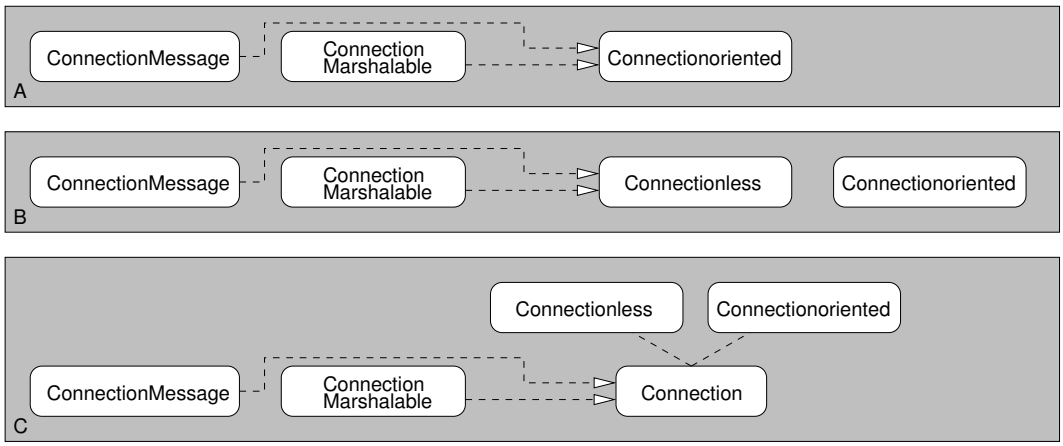

Abbildung 4.2: Integration von Verbindung(en)

Parameter (P-Schicht)

In der dritten Schicht wird ein Marshalable mit Datentypen spezialisiert, indem dessen Schnittstelle durch die verwendeten Datentypen implementiert wird. Die Methoden *marshal* und *unmarshal* können Variablen dieses Types de-/serialisieren. Einer Nachricht können in dieser Kollaboration Marshalables zugewiesen werden (Aggregation). Die Marshalables benutzen die Verbindung der Nachricht. Wird eine Nachricht de-/serialisiert, werden die Parameter mit derselben Verbindung wie die Nachricht unter Nutzung der Methoden *marshal* und *unmarshal* de-/serialisiert. Die Parameter besitzen in der Nachricht eine Reihenfolge. Bei einer De-/Serialisierung besteht eine Nachricht aus drei aufeinanderfolgenden Teilen: dem Nachrichtenkopf mit dem Operationsnamen, den Marshalables und einem Nachrichtenabschluß.

Welche Datentypen von den Marshalables gekapselt werden können, hängt eng mit der Protokollimplementierung zusammen. Weitere Datentypen können davon abgeleitet oder daraus zusammengesetzt werden und erben dabei insbesondere die Eigenschaften, sich zu de-/serialisieren. Zusammengesetzte Datentypen nutzen diese Eigenschaften zur De-/Serialisierung ihrer Bestandteile. Somit ist es möglich, komplexe Datenstrukturen bzw. Objekte zu übertragen.

Die Konfiguration des Merkmals *Datentyp* ist in Abbildung 4.3 skizziert. Je nachdem welche Datentypen benötigt werden, werden alternative Kollaborationen als Parameterschicht eingesetzt, die diese Datentypen enthalten. Kollaboration A enthält nur einen Datentyp *Integer*. Kollaboration B enthält die Typen *Float* und *String*. Die Bereitstellung der Datentypen in einer Kollaboration wird im Konfigurierungsprozeß vorgenommen.

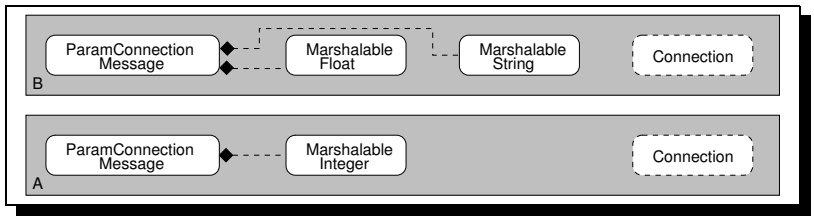

Abbildung 4.3: Konfiguration des Merkmals “Datentyp”

Richtungstrennung (R-Schicht)

Im vierten Schritt wird die Richtung einer Nachricht festgelegt. Dazu wird die geerbte Nachrichtenrolle in zwei Varianten spezialisiert: eine ausgehende (*OutMessage/Out-Nachricht*) und eine eingehende Nachricht (*InMessage/In-Nachricht*). Um den Entwurf überschaubar zu halten, wird in dieser und in den folgenden Schichten darauf verzichtet, die In- und die Out-Nachrichten in getrennten Kollaborationen zu entwickeln. Eine getrennte Modellierung ermöglicht jedoch eine feingranularere Konfigurierung. Die ausgehende Nachricht besitzt eine Funktion *marshal*, um die Nachricht zu serialisieren. Die eingehende Nachricht besitzt eine Funktion *unmarshal*, um die Nachricht zu deserialisieren. Ausgehende Nachrichten interpretieren den Operationsnamen als Adressaten, eingehende als Absender.

Eine Out-Nachricht serialisiert ihre Bestandteile in der Reihenfolge: Nachrichtenkopf mit dem Operationsnamen, dann die Marshalables und zuletzt der Nachrichtenabschluß. Eine In-Nachricht kann nur eine solche Nachricht deserialisieren, deren Marshalables in Typ, Anzahl und Reihenfolge den eigenen Marshalables entsprechen, und deren Operationsname dem eigenen Operationsnamen entspricht.

Eine *Verbindung* kann eine Nachricht entweder gepuffert oder ungepuffert versenden bzw. empfangen. Werden Puffer zur Serialisierung und Deserialisierung verwendet, sind Serialisieren und Senden bzw. Deserialisieren und Empfangen zwei getrennte Operationen. Eine zu sendende Nachricht wird zuerst in einen Puffer serialisiert und dann aus dem Puffer gesendet. Eine zu empfangende Nachricht wird in den Empfangspuffer empfangen und später daraus deserialisiert.

Beim Versenden ohne Puffer wird eine Nachricht direkt auf einen Datenstrom (engl. Stream) serialisiert bzw. beim Empfangen direkt von einem Datenstrom deserialisiert. Die Deserialisierung einer In-Nachricht direkt von einem Datenstrom ist in Abbildung 4.4 skizziert.

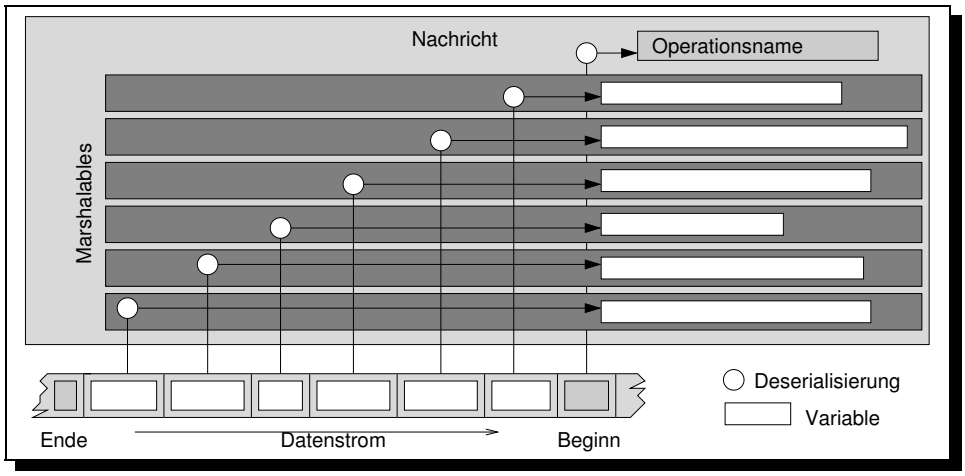

Abbildung 4.4: Deserialisierung einer Nachricht von einem Datenstrom

Synchronisation (S-Schicht)

Eine Synchronisationsstrategie implementiert Methoden zum Empfangen (*recv*) und Senden (*send*) einer Nachricht (synchron/asynchron)[2]. In der Domänenanalyse ist dieses Merkmal statisch sowie dynamisch austauschbar modelliert. Die Strategie wird als neue Kollaboration in den Entwurf integriert, die den Entwurf um eine Rolle *Synchronisation* erweitert (Abb. 4.1).

Eine Voraussetzung zur Bereitstellung von Synchronisationsstrategien ist, daß De-/Serialisierung sowie Versand und Empfang einer Nachricht zwei getrennte Operationen sind. Zuerst wird eine Nachricht serialisiert, dann entsprechend der Synchronisationsstrategie versendet. Wird durch eine Protokollimplementierung die Trennung von Serialisierung und Senden nicht untertützt, sondern nur in einer Operation angeboten, können zusätzliche Synchronisationsstrategien nicht unterstützt werden. Das Versenden wird schon bei der Serialisierung (durch den Aufruf von *marshal*) durchgeführt. Innerhalb der Methoden *send* und *recv* kann eine Synchronisationsstrategie dann keinen Einfluß mehr auf den Sende- oder Empfangsvorgang nehmen.

Eine Beziehung zwischen einer Synchronisationsstrategie und Nachrichtenrollen wird in den Schichten *OneWay* und *TwoWay* hergestellt.

Die Konfigurierung des Merkmals Synchronisationsstrategie wird über den Wechsel der Synchronisationsschicht realisiert. In Abbildung 4.5 werden drei austauschbare Kollaborationen gezeigt. Die Kollaboration A enthält eine Synchronisationsstrategie. B enthält zwei Strategien, die alternativ verwendet werden können. In C sind ebenfalls mehrere Strategien enthalten, die zudem zur Laufzeit austauschbar sind. Dazu wird ein Vermittler (*Messenger*) benutzt, der zwischen mehreren Strategien (hier synchron/asynchron) zur

[2]Die Synchronisationsstrategie kann später, wenn gewünscht, in eine Sende- und Empfangsstrategie aufgeteilt werden.

Laufzeit umschalten kann. Ein Nachricht, die eine dynamisch austauschbare Synchronisationsstrategie benötigt, wird nicht mit einer Strategie, sondern mit dem Vermittler konfiguriert.

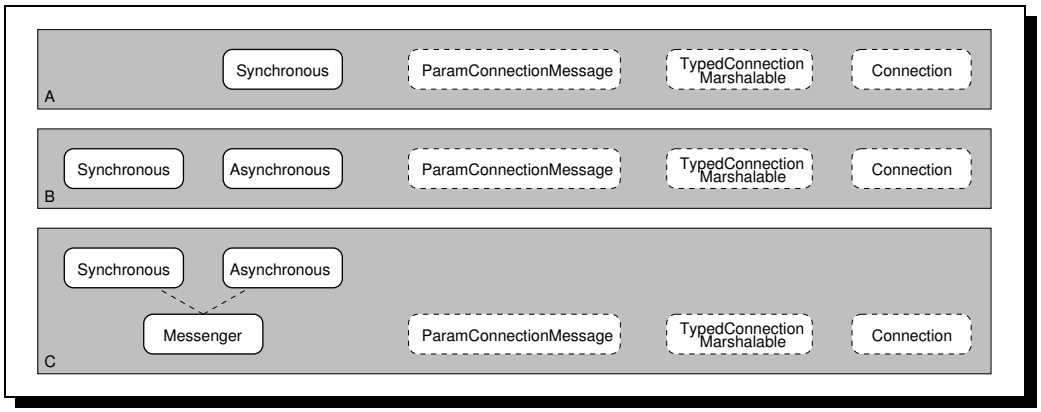

Abbildung 4.5: Alternative Synchronisationsstrategien

OneWay (ONEWAY-Schicht)

Die sechste Schicht implementiert das Merkmal *OneWay*. Dazu wird in der Kollaboration eine gleichnamige Rolle *OneWay (OneWay-Aufruf)* eingeführt. Aufgabe dieser Kollaboration ist die Bereitstellung von Dienstaufrufen ohne Antwortnachricht. Der OneWay-Aufruf vereint eine *OutMessage* mit einer Synchronisationsstrategie (Aggregation), wobei anstelle der Synchronisationsstrategie auch der erwähnte Messenger verwendet werden kann.

Die OneWay-Kollaboration stellt eine Schnittstelle zur Generierung und zur Ausführung eines OneWay-Aufrufes bereit. Ein OneWay-Aufruf wird zuerst mit Parametern und einem Operationsnamen für die ausgehende Nachricht versehen, anschließend serialisiert und mit Hilfe der Synchronisationsstrategie über die Verbindung versendet (Methode *send*)[3]. Serialisierung und Senden sind im OneWay zur einer *send*-Methode vereint, die eine Nachricht erst serialisiert und dann entsprechend der Synchronisationsstrategie versendet.

In Abbildung 4.6 ist der Sendevorgang eines OneWay-Aufrufes skizziert. Die Daten einer *OutMessage* - deren Marshalables - werden in einen Puffer serialisiert (*marshal*) und dann entsprechend der Synchronisationsstrategie gesendet (*send*). Die Pfeile sollen andeuten, welcher Rolle welche DatenStrukturen zugeordnet sein können. Die *Marshalables* gehören zur *OutMessage*. Der Datenstrom ist der Rolle *Connection* zugeordnet. Der Serialisierungspuffer kann in der Implementierung entweder ebenfalls der *OutMessage*, der

[3]Eine Nachricht besitzt nur Funktionen, sich zu serialisieren oder zu deserialisieren, aber keine Funktionen, sich zu senden oder zu empfangen. Diese Funktionen werden erst durch den OneWay- und den TwoWay-Aufruf durch Kombination mit einer Synchronisationsstrategie hergestellt.

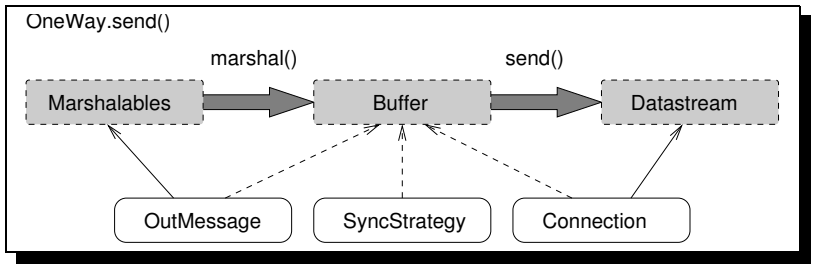

Abbildung 4.6: Serialisierung und Senden eines OneWay

Verbindung oder auch der *Synchronisationsstrategie* zugeordnet sein. Bietet eine Protokollimplementierung nur die Serialisierung der Daten direkt auf einem Datenstrom an, ist dieser Puffer nicht vorhanden. Eine Synchronisationsstrategie hat dann keine Einflußmöglichkeiten auf den Sende- bzw. Empfangsvorgang.

TwoWay (TWOWAY-Schicht)

Schicht sieben stellt das Merkmal *TwoWay* bereit und integriert eine neue Rolle *TwoWay (TwoWay-Aufruf)* in diese Schicht. Ziel dieser Kollaboration ist die Bereitstellung von Dienstaufrufen mit Antwortnachricht. Ein TwoWay-Aufruf ist ein erweiterter OneWay-Aufruf. Er erbt die Strategie und die *OutMessage* vom OneWay-Aufruf und besitzt zusätzlich eine *InMessage*, wobei *OutMessage* und *InMessage* dieselbe Verbindung und Synchronisationsstrategie benutzen.

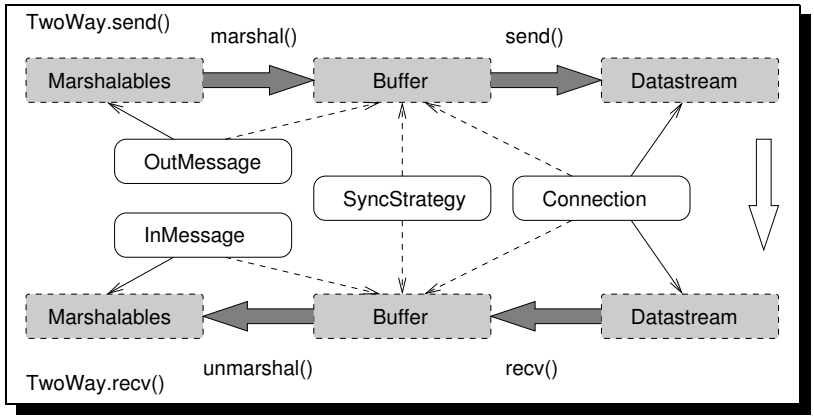

Abbildung 4.7: Ausführung eines TwoWay

Eine TwoWay-Kollaboration stellt eine Schnittstelle zur Generierung und zur Ausführung eines TwoWay-Aufrufes bereit. Bevor der Aufruf ausgeführt werden kann, wird er mit Parametern und Operationsnamen für die eingehende und ausgehende Nachricht versehen. Anschließend wird die *OutMessage* serialisiert und entsprechend der Syn-

chronsationsstrategie versendet (siehe Abb. 4.7). Danach wird eine *InMessage* als Antwort empfangen (*recv*) und deserialisiert (*unmarshal*). In der Abbildung ist auch dargestellt, welchen Rollen welche Datenstrukturen zugeordnet sein können.

Die OneWay-Schicht ist eine notwendige, TwoWay ist eine optionale Schicht bei der Konfigurierung eines Clients. OneWay und TwoWay sind nicht, wie in der Analyse gefordert, zwei völlig unabhängige optional-alternative Merkmale. Der Grund ist, daß das Merkmal *OneWay* in jedem *TwoWay* als Funktionsuntermenge enthalten ist und sich praktisch nicht "herauskonfigurieren" läßt. Ein TwoWay-Aufruf ist daher eine Erweiterung eines OneWay-Aufrufes. Die Konfigurationsmöglichkeiten für die beiden Schichten sind in Abbildung 4.8 dargestellt.

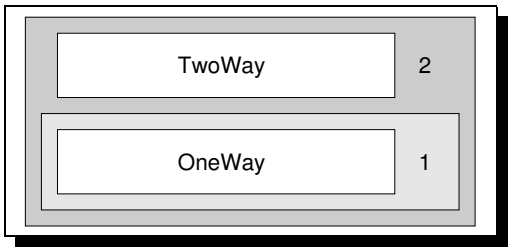

Abbildung 4.8: Konfiguration des Merkmals "Richtung"

Entfernte Funktionsaufrufe (RPC-Schicht)

Mit dieser und den folgenden zwei Schichten werden entfernte Funktionsaufrufe, Klassenfunktionsaufrufe und Objektmethodenaufrufe bereitgestellt (vgl. auch Kapitel 2), um von einem Client einen Dienst auf einem entfernten Server aufzurufen. Dies entspricht dem Merkmal *Service Paradigma* bzw. deren Untermerkmalen *RPC*, *RCI* und *ROI* aus der Domänenanalyse. Jedes dieser Merkmale wird in je einer Kollaboration gekapselt.

In der RPC-Schicht werden die beiden aus der Richtungstrennungsschicht geerbten Nachrichtenrollen erweitert in eine *RPC-OutNachricht (OutFunctionMessage)* und eine *RPC-InNachricht (InFunctionMessage)* (vgl. Abb. 4.1). Mit dieser Kollaboration ist ein Client in der Lage, entfernte Dienste auf Funktionsbasis (RPC's) aufzurufen. Eine entfernte Funktion wird über einen Funktionsnamen (Dienstnamen) identifiziert. Die bereits vorgestellten OneWay- und TwoWay-Aufrufe erfüllen in dieser Kollaboration die Rollen, entfernte Funktionen mit Hilfe einer Synchronisationsstrategie ohne bzw. mit Antwortnachricht aufzurufen. Diese Rollen werden in der Kollaboration als *OneWay-Funktionsaufruf (OneWay-FunctionCall)* bzw. *TwoWay-Funktionsaufruf (TwoWay-FunctionCall)* bezeichnet. Ein *OneWay-* oder ein *TwoWay-Funktionsaufruf* wird von einem Client zum Aufruf einer Dienstfunktion auf einem Server verwendet. Der OneWay-Funktionsaufruf eignet sich auch, um das Ergebnis eines Client-Funktionsaufrufes vom Server an den Client zu übermitteln.

Entfernte Klassenaufrufe (RCI-Schicht)

Die RCI-Schicht ermöglicht einem Client, auf einem entfernten Server einen Dienst aufzurufen, der als statische Klassenfunktion bereitgestellt wird. Dazu werden die aus der RPC-Schicht geerbten Rollen *RPC-OutNachricht* und *RPC-InNachricht* in *RCI-OutNachricht (OutClassMessage)* und *RCI-InNachricht (InClassMessage)* erweitert. Eine entfernte Klassenfunktion wird auf einem Server über einen Klassennamen und über einen innerhalb der Klasse eindeutigen Namen einer statischen Methode identifiziert. Diese beiden Namen identifizieren den Dienst in Form einer Klassenfunktion eindeutig.

Polymorphe Klassenmethoden[4] werden in der vorliegenden Implementierung nicht unterstützt, da eine Unterscheidung allein anhand des Dienstnamens erfolgt. (Polymorphe Klassenmethoden können leicht durch Hinzufügen einer weiteren Kollaboration integriert werden.) Der OneWay-Aufruf dieser Schicht kann eine entfernte Klassenfunktion ohne Antwortnachricht mit einer Synchronisationsstrategie aufrufen. Er kann gleichzeitig von einem Server verwendet werden, um den Klassenfunktionsaufruf eines Clients zu beantworten. Mit dem TwoWay-Aufruf können entfernte Klassenfunktionen aufgerufen und eine Antwortnachricht empfangen werden, welche das Ergebnis beinhaltet.

Entfernte Objektaufrufe (ROI-Schicht)

Mit der ROI-Schicht werden Aufrufe bereitgestellt, um Methoden entfernter Objekte auszuführen. Die aus der RCI-Schicht geerbten Rollen *RCI-OutNachricht* und *RCI-InNachricht* werden erweitert zu einer *ROI-OutNachricht (OutObjectMessage)* und einer *ROI-InNachricht (InObjectMessage)*. Eine entfernte Objektmethode ist auf einem Server eindeutig identifizierbar über einen Klassennamen, eine Objektnummer und einen Methodennamen. Ein Dienstname besteht somit aus drei Teilen: Klasse, Objekt, Methode. Polymorphe Objektmethoden werden in der Implementierung nicht unterstützt (können aber ebenfalls später leicht durch eine weitere Kollaboration integriert werden).

Der TwoWay-Aufruf dieser Kollaboration kann ein entferntes Objekt aufrufen und erwartet Parameter zurück. Der OneWay-Aufruf kann zum Aufruf einer Objektmethode ohne Parameterrückgabe oder zur Beantwortung eines Objektmethodenaufrufes an den Client verwendet werden.

Bevor ein Dienstobjekt aufgerufen wird, muß das Objekt auf dem Server erzeugt werden. Jede instantiierbare Klasse muß dazu eine statische Klassenfunktion als Dienst anbieten, die ein neues Objekt dieser Klasse erzeugt und dessen Objektnummer zurückliefert. Ein Dienstnutzer kann "sein Objekt" damit später eindeutig adressieren. Das Löschen eines nicht mehr benötigten Objektes wird ebenfalls über eine Funktion der instantiierbaren Klasse bereitgestellt. Mit diesen Funktionen ist es möglich, den Lebenszyklus eines Objektes auch über Dienstaufrufe aus der Ferne zu steuern.

In Abbildung 4.9 sind in einem Sequenzdiagramm ein Client- und ein Objektserver-Prozeß dargestellt. Der Objektserver kennt die Funktionen *executeClass* und *executeOb-*

[4] Methoden mit gleichem Namen aber unterschiedlicher Signatur

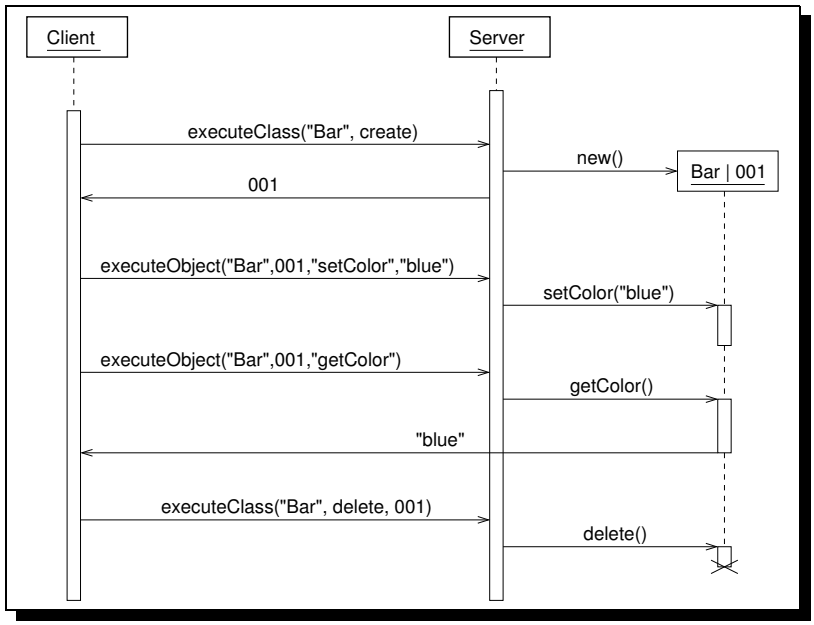

Abbildung 4.9: Lebenszyklus eines entfernten Objektes

ject. *ExecuteClass* ruft unter Angabe eines Klassennamens und eines Methodennamens sowie optionaler Parameter eine statische Klassenmethode auf der Klasse auf. *ExecuteObject* erwartet einen Klassennamen, eine Objektnummer, einen Methodennamen sowie optionale Parameter. In Abbildung 4.9 ruft der Client per TwoWay-Aufruf die Klassenfunktion *create* auf der instantiierbaren Klasse *Bar* auf. Der Server erzeugt daraufhin eine Instanz von *Bar*, ordnet dieser eine laufende Objektnummer innerhalb der Klasse zu (*001*) und schicht die Nummer an den Client zurück. Der Client kann nun Methoden auf diesem Objekt aufrufen. Im Beispiel wird dem Objekt in einem OneWay-Objektaufruf eine fiktive Farbe zugeordnet. Das Objekt *Bar* muß eine entsprechende Methode *setColor* besitzen, die dann vom Server aufgerufen wird. In einem zweiten TwoWay-Objektaufruf erfragt der Client die Farbe von seinem entfernten Objekt *001*. Der Server leitet die Anfrage an das Objekt weiter und dieses sendet die Antwort an den Client zurück. Benötigt der Client das Objekt nicht mehr, löscht er es über den Aufruf *delete* auf der Klasse *Bar* unter Angabe der Objektnummer.

Das Merkmal *Service-Paradigma* wird über das Weglassen bzw. Einfügen von Schichten konfiguriert. Da die drei Kollaborationen RPC - RCI - ROI aufeinander aufbauen, sind drei Kombinationen möglich, die in Abbildung 4.10 dargestellt sind.

Kardinalität

Für das Merkmal *(Aufruf)Kardinalität* wird bislang die Variante *Single Receiver* unterstützt. Um einen Aufruf an mehrere Empfänger zu verschicken, muß das Merkmal *Direction* als Kollaboration (OneWay/TwoWay) beachtet werden. Ein OneWay-Multicast

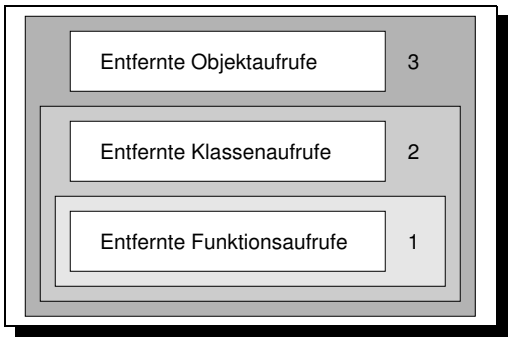

Abbildung 4.10: Konfiguration des Serviceparadigmas

kann leicht in den Entwurf eingefügt werden: Der OneWay wird für jeden Empfänger einzeln verschickt. Dazu wird dem OneWay für jeden Empfänger eine neue Verbindung zugewiesen bzw. der Endpunkt der Verbindung wird für jeden Empfänger neu gesetzt. Da die Dienste auf unterschiedlichen Servern unterschiedlich benannt sein können - insbesondere wenn es sich um Objekte handelt (Objekt-ID) - muß ggf. der Dienstname in der RPC-/RCI- oder ROI-Schicht angepaßt werden.

Bei einem TwoWay-Multicast wird für jeden Aufruf eine Antwort erwartet. Der Umgang mit den Rückgabeergebnissen kann sowohl Aufgabe der Middleware als auch der Applikation sein, wenn die Middleware nicht weiß, wie sie die Ergebnisse verarbeiten soll. Zum Zwischenspeichern der Ergebnisse durch die Middleware wird unter Umständen viel Speicherplatz benötigt. Um Speicher zu sparen, kann ein TwoWay-Multicast durch einen leistungsstarken Knoten im Netz realisiert werden. Ein Client beauftragt diesen Knoten mit der Ausführung der Aufrufe und der Entgegennahme der Ergebnisse, die z.B. in einer Datenbank gespeichert und ggf. vorverarbeitet werden. Der Client kann später über Datenbankdienste des Zwischenknotens Ergebnisse des Multicast abfragen. Fazit: Das Merkmal *Kardinalität* ist ganz offensichtlich in höheren Schichten der Middleware angesiedelt. Entwurfsentscheidungen zu diesem Merkmal sollen daher erst später getroffen werden.

4.3 Die Server

Im Folgenden wird gezeigt, wie die in der Domänenanalyse definierten Merkmale der Server in den bestehenden Kollaborationenentwurf des Clients integriert werden, so daß Client- und Server-Komponenten Teil einer gemeinsamen Architektur sind, aber trotzdem getrennt konfiguriert werden können (vgl. Abb. 4.1). Zunächst wird an einem Beispiel gezeigt, wie mit Mitteln des Kollaborationenentwurfes Merkmale von Client und Server getrennt oder zusammengeführt werden können.

Separierung von Client- und Servermerkmalen

Eine Merkmalstrennung für Client und Server kann über eine Implementierung der Merkmale in unterschiedlichen Kollaborationen erreicht werden. Kollaborationen, die allein Merkmale des Servers implementieren, sollen dabei unabhängig von Kollaborationen, die nur Merkmale des Clients besitzen, eingesetzt werden können. Um einen Server mit zusätzlichen Client-Merkmalen zu konfigurieren, sollen eine Vermischung von Client- und Server-Kolloborationen möglich sein.

Die Kollaborationen müssen so entworfen sein, daß reine Serverkollaborationen für den Client optional sind und umgekehrt. So können Client und Server innerhalb einer Familie getrennt konfiguriert werden. In Abbildung 4.11 sind drei Kollaborationen dargestellt, die - unterschiedlich miteinander kombiniert - Merkmale des Clients, des Servers oder eines Servers mit Client-Merkmalen implementieren.

Kollaboration A ist eine Client-Kollaboration mit einer Rolle *OutFunctionMessage*, die entfernte Funktionsaufrufe bereitstellt. B ist eine Kollaboration mit einer Rolle *InFunctionMessage*, die für Client und Server verwendet wird. Der Client benötigt diese Rolle, wenn auf entfernte Funktionsaufrufe auch eine Antwortnachricht mit einem Ergebnis empfangen werden soll. Sind die Kollaborationen A und B vorhanden, so erbt B das Merkmal *OutFunctionMessage* von A, und die Middleware kann in Schicht B das Merkmal *Funktionsaufrufe mit Antwortnachricht* für einen Client bereitstellen.

Der Funktionsserver in Kollaboration C (eine reine Serverkollaboration) stellt Funktionsdienste bereit. Dazu benötigt er auch die Rolle *InFunctionMessage* aus Schicht B, um *entfernte Funktionsaufrufe empfangen* zu können. Sind die Schichten B und C vorhanden, können auf diesem Server Funktionen aufrufen werden, jedoch ohne Antwortnachricht, denn der Server unterstützt keine ausgehenden Nachrichten. Um dieses Merkmal bereitzustellen, muß auch die Schicht A in der Server-Hierarchie vorhanden sein, aus der der Server die *OutFunctionMessage* erbt.

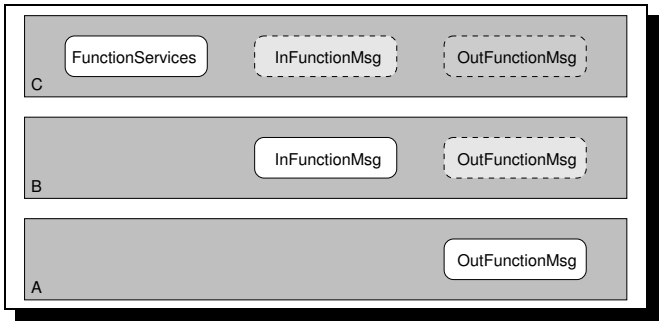

Abbildung 4.11: Separierung von Client- und Servermerkmalen

Mit diesem Beispiel soll das Prinzip verdeutlicht werden, wie durch getrennte Kapselung von Client-Rollen, Server-Rollen und allgemeinen Rollen in unterschiedlichen Kollaborationen eine Trennung aber auch eine Zusammenführung von Client- und Servermerk-

malen in einer Middleware-Familie sehr einfach möglich ist.

(Im vorliegenden Entwurf sind ausgehende und eingehende Nachrichten aus pragmatischen Gründen nicht als getrennte Kollaborationen entwickelt, da sich damit die Schichtenanzahl der Middleware verdoppelt hätte. Eine Trennung der Nachrichten soll in einem späteren Entwurf umgesetzt werden.)

Gemeinsame funktionale Basis von Client und Server

Client und Server besitzen mehrere gemeinsame Merkmale, die in der Domänenanalyse als *allgemeine Middlewaremerkmale* bezeichnet werden. Darunter fallen die Merkmale *Datentypen*, *Verbindung*, *Synchronisation*, *Serviceparadigma* und *Protokoll*. Die ersten drei Merkmale werden durch die untersten fünf Kollaborationen der bisher beschriebenen Client-Hierarchie implementiert. Auch die für das Servermerkmal *ReceiveCalls* notwendigen Rollen *InMessage* und *Connection* sind innerhalb dieser Kollaborationen definiert. Die *OutMessage* der *Richtungstrennungsschicht* ist nicht Bestandteil der Servermerkmale, sie läßt sich ohnehin nur mit der *InMessage* gemeinsam konfigurieren.

Daher können die Kollaborationen *Abstrakte Nachrichtenschicht*, *Verbindungsschicht*, *Parameterschicht*, *Richtungstrennungsschicht* und *Synchronisationsschicht* als gemeinsame funktionale Basis für Client und Server angesehen werden.

In den nächsten Abschnitten wird nun gezeigt, wie serverspezifische Merkmale wie *entfernte Aufrufe entgegennehmen (ReceiveCalls)* und *Dienste bereitstellen (OfferServices)* oberhalb der gemeinsamen funktionalen Basis in die Hierarchie integriert werden. Das in der Domänenanalyse beschriebene Merkmal *Aufrufe empfangen (ReceiveCalls)* wird durch die die zwei Kollaborationen umgesetzt, die in den nächsten beiden Abschnitten beschrieben werden.

Serververbindungsschicht (SV-Schicht)

Die *Verbindungsschicht* mit der Verbindungsrolle ist für den Server nicht ausreichend: Die Rolle *Verbindung* kann eine (reale oder logische) Verbindung aufbauen, aber sie kann keine Verbindungsanfrage eines anderen Knotens entgegennehmen.

Dieses Merkmal wird daher als zusätzliche Kollaboration für den Server in die Kollaborationenhierarchie eingefügt (siehe Abb. 4.12). Ein Server nimmt über eine Rolle *Serververbindung (AcceptConnection)* eine Verbindungsanfrage entgegen. Daraus kann eine *Connection* zum Client-Endpunkt abgeleitet werden, über die die Kommunikation mit einem Client abgewickelt wird. Die in der neu eingeführten *Serververbindungsschicht* enthaltene Rolle *AcceptConnection* setzt daher die Rolle *Connection* voraus. Die *Serverberbindungsschicht* ist wie in Abbildung 4.12 dargestellt, oberhalb der Synchronisationsschicht S angeordnet und erbt dort auch die *Connection*.

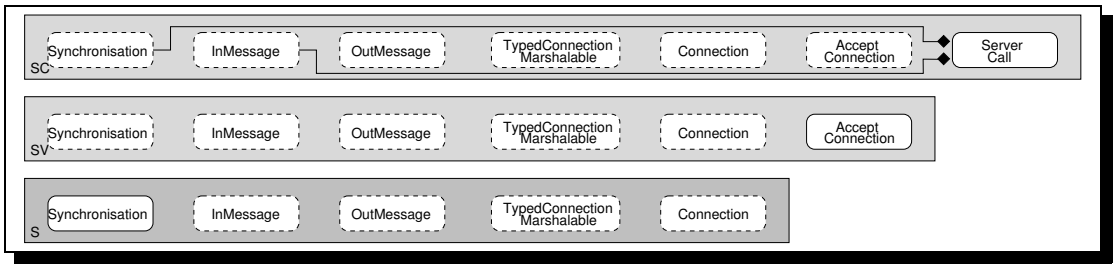

Abbildung 4.12: Servermerkmale Connection und ReceiveCalls

Serveraufrufschicht (SC-Schicht)

Ein *ServerCall* ist eine Nachricht, die unter Verwendung einer Synchronisationsstrategie empfangen wird. Der Empfang findet über eine *Connection* statt, die zuvor vom Server nach dem Verbindungsaufbau aus einer *AcceptConnection* abgeleitet wird. Der *ServerCall* wird über eine *InMessage* und eine Synchronisationsstrategie implementiert (Abb. 4.12 (C)). (Eine Verknüpfung zwischen einer *InMessage* und einer Strategie findet bisher nur in der TwoWay-Schicht statt. Dies soll jedoch eine reine Client-Schicht bleiben, sie enthält zudem eine synchronisierte *OutMessage* und wird daher nicht für den Server verwendet.) Für den Server wird eine *Serveraufruf-Schicht* mit einer neuen Rolle *ServerCall* oberhalb der *Serververbindungsschicht* (Abb. 4.12 (B)) eingeführt, die diese Verknüpfung durchführt (Abb. 4.12 (C)). Die *Serververbindungsschicht* und die *Serveraufruf-Schicht* stellen damit das Merkmal *ReceiveCalls* bereit.

Dienste anbieten und Serviceparadigmen (SRPC, SRCI und SROI-Schichten)

Die Integration der Merkmale *Dienste anbieten (Offer Services)* und *Serviceparadigma* wird zusammen beschrieben, da beide Merkmale sehr eng miteinander verknüpft sind. Entsprechend den drei Serviceparadigmen werden drei verschiedene Servertypen angeboten: ein Funktionsserver, ein Klassenserver und ein Objektserver. Ein Funktionsserver kann entfernte Funktionsaufrufe empfangen und Dienste in Form von Funktionen bereitstellen. Der Klassenserver kann zusätzlich entfernte Aufrufe auf statischen Klassenfunktionen entgegennehmen und Dienste in dieser Form anbieten. Der Objektserver nimmt auch Objektaufrufe entgegen und stellt Dienstobjekte zur Verfügung.

In Abbildung 4.13 ist dargestellt, wie diese drei Server als Kollaborationen *Server-RPC (SRPC)*, *Server-RCI (SRCI)* und *Server-ROI (SROI)* oberhalb der entsprechenden Client-Schichten *RPC*, *RCI* und *ROI* integriert werden. Der Funktionsserver befindet sich oberhalb des Funktions-Clients, damit er ggf. auch einen OneWay-Funktionsauruf, der von der Client-Schicht bereitgestellt wird, als Antwortnachricht auf einen Dienstaufruf verwenden kann.

Jede dieser Schichten erweitert das Merkmal *Receive Calls* und implementiert das Merkmal *Offer Services* für ein anderes Serviceparadigma. Der Funktionsserver (*SRPC*) ist

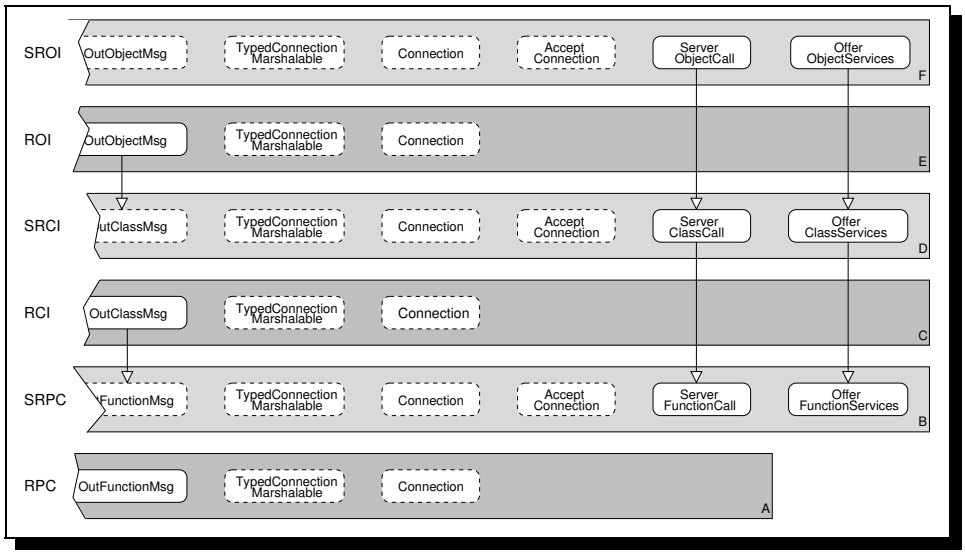

Abbildung 4.13: Serverschichten SRPC, SRCI und SROI

in der Lage, entfernte Funktionsaufrufe zu empfangen und kann Dienste in Form von Funktionen anbieten, der Klassenserver kann Klassenmethodenaufrufe empfangen und anbieten usw.

In diesen drei Serverschichten wird die Rolle *InMessage* dazu in eine *InFunctionMessage*, eine *InClassMessage* und eine *InObjectMessage* erweitert (vgl. Abb. 4.13). Damit arbeitet auch die Rolle *Serveraufruf* in diesen Schichten nicht mit einer *InMessage*, sondern mit einer *InFunction-*, *InClass-* oder einer und *InObjectMessage* zusammen und kann damit Nachrichten des entsprechenden Paradigmas deserialisieren.

Die Rolle *OfferService* stellt in den drei Schichten die Fähigkeiten bereit, Funktionen, Klassen bzw. Objekte auszuführen, die von höheren Schichten der Middleware oder von eine Applikation angeboten werden. Ein Funktionsserver kann Funktionen registrieren und ausführen. Bei einem Klassenserver können Klassen sowie statische Klassenmethoden angemeldet und ausgeführt werden. Ein Objektserver kann Instanzen von Klassen erzeugen, verwalten und entfernen sowie Methoden auf Objekten ausführen.

Konfiguration der Servermerkmale

Das Merkmal *Serviceparadigma* wird für einen Server über das Weglassen bzw. Einfügen der Serverschichten konfiguriert. Aufgrund der funktionalen Abhängigkeiten zwischen Funktionsserver, Klassenserver und Objektserver ergeben sich für das Serviceparadigma drei Konfigurationsmöglichkeiten (ähnlich Abb. 4.10).

Mit der Konfiguration des Serviceparadigmas werden gleichzeitig die Merkmale *Receive Calls* und *Offer Services* konfiguriert. Eine getrennte Konfigurierung dieser Merkmale ist nicht notwendig, denn ein Server, der nur eines dieser Merkmale besitzt, ist nutzlos. Somit

ergeben sich für einen Server für die drei Merkmale *Serviceparadigma*, *Receive Calls* und *Offer Services* insgesamt auch nur drei Konfigurationsmöglichkeiten: Funktionsserver, Klassenserver bzw. Objektserver.

Das Merkmal *Serviceparadigma* kann als grobgranularer Crosscutting Concern aufgefaßt werden, denn es tangiert die anderen Servermerkmale. An diesem Beispiel wird deutlich, wie mit Hilfe des kollaborationenbasierten Entwurfs ein Crosscutting Concern erfolgreich mit anderen Merkmalen verknüpft und gleichzeitig modularisiert werden kann.

Bis hierhin wurden bis auf das Merkmal *Protokoll (Protokollimplementierung)* alle Merkmale der Domänenanalyse für den Client und den Server im Entwurf verarbeitet. Ein Schema des um die Servermerkmale erweiterten Entwurfes ist in Abbildung 4.14 dargestellt.

Protokoll

Das Merkmal *Protokoll* betrifft Client und Server als allgemeines Merkmal gleichermaßen. Es wird in Form einer Protokollimplementierung in den Entwurf integriert. Im Vergleich zu den anderen Merkmalen ist die Protokollimplementierung außerordentlich eng mit wohl allen anderen Merkmalen der Familie verbunden: ob Verbindung, Parameterunterstützung, Synchronisationsstrategien, Serverfunktionalität usw., alle diese Merkmale greifen auf Funktionen der Protokollimplementierung zurück.

Die Protokollimplementierung ist orthogonal zur Basisfunktionalität und so eng mit allen bisher eingeführten Merkmalen verknüpft, daß sie aus der aspektorientierten Perspektive ein massiver Crosscutting Concern bezüglich anderer Merkmale ist. Jede bislang vorgestellte Kollaboration bzw. deren Rollen, die ein Merkmal implementieren, greifen direkt auf Funktionen der Protokollimplementierung zu.

Trotzdem soll das Protokoll, so wird in der Domänenanalyse gefordert, austauschbar in die Familie interiert werden. Erschwerend für den Entwurf ist, daß jede Protokollimplementierung eine andere Schnittstelle zur Verfügung stellt und die anderen Merkmale unterschiedlich stark unterstützt.

Damit wird deutlich, daß die Implementierung der Kollaborationen und Rollen völlig auf eine Protokollimplementierung zurechtgeschnitten sein muß. Für jede Protokollimplementierung ist im schlimmsten Fall eine Re-Implementierung der in Abbildung 4.14 dargestellten Kollaborationen notwendig. Merkmale, die von einer Protokollimplementierung ungenügend oder nicht unterstützt werden, müssen, soweit die Voraussetzungen dafür gegeben sind, innerhalb der Kollaborationen nachgebildet werden, oder sie können nicht unterstützt werden.

Für die komplette Implementierung der Kollaborationen einer konkreten Protokollimplementierung wird im Folgenden der Begriff *Protokolladapter* verwendet. Ein Protokolladapter stellt eine Protokollimplementierung für höhere Schichten der Middleware oder für Applikationen unter einer einheitlichen Schnittstelle zur Verfügung. Oberhalb der funktionalen Ebene "Entfernte Funktions-/Objektaufrufe" im Architekturentwurf von [AP03a] (vgl. Abb. 3.1) soll der verwendete Protokolladapter für Middleware und

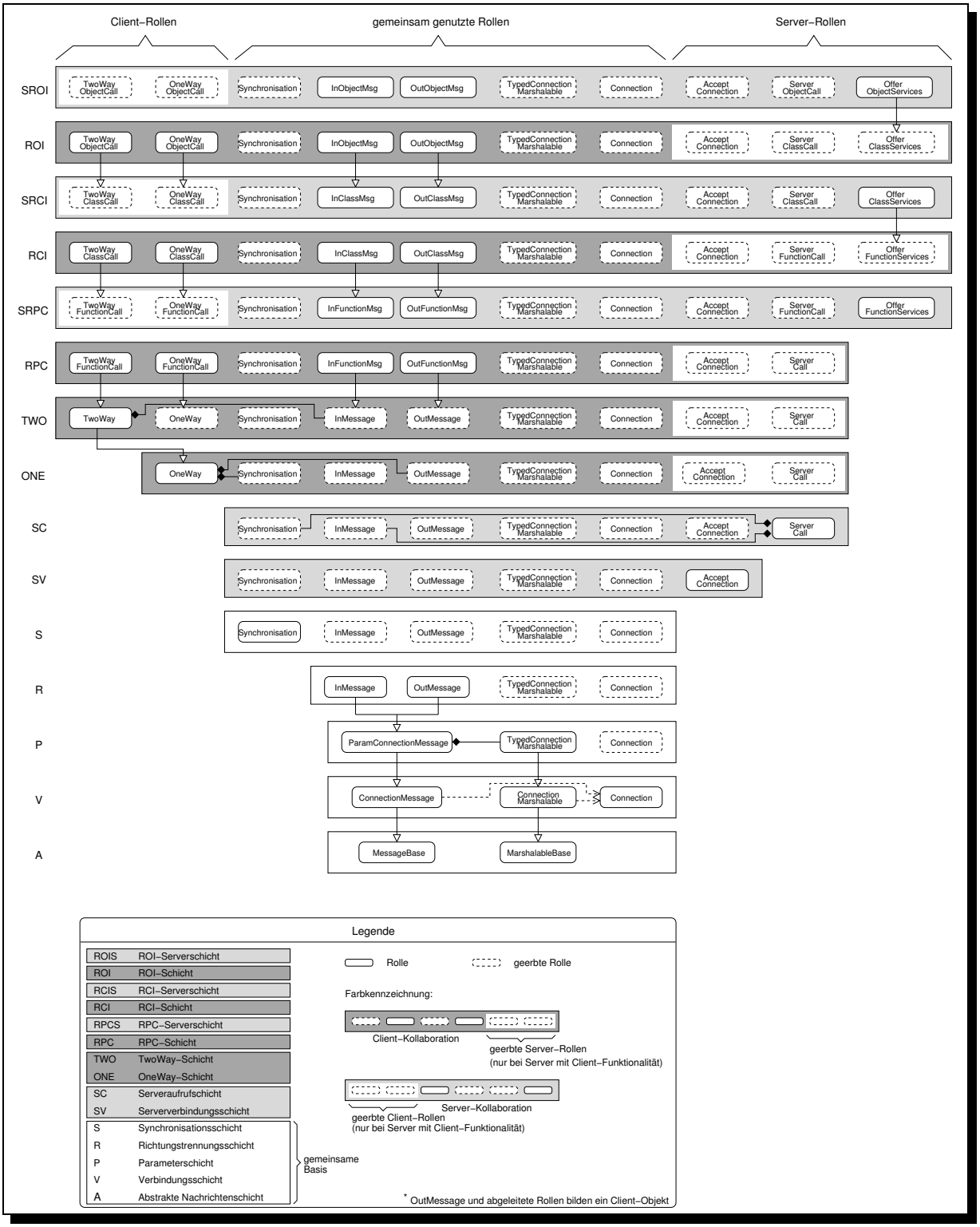

Abbildung 4.14: Funktionale Hierarchie der Middlewarefamilie

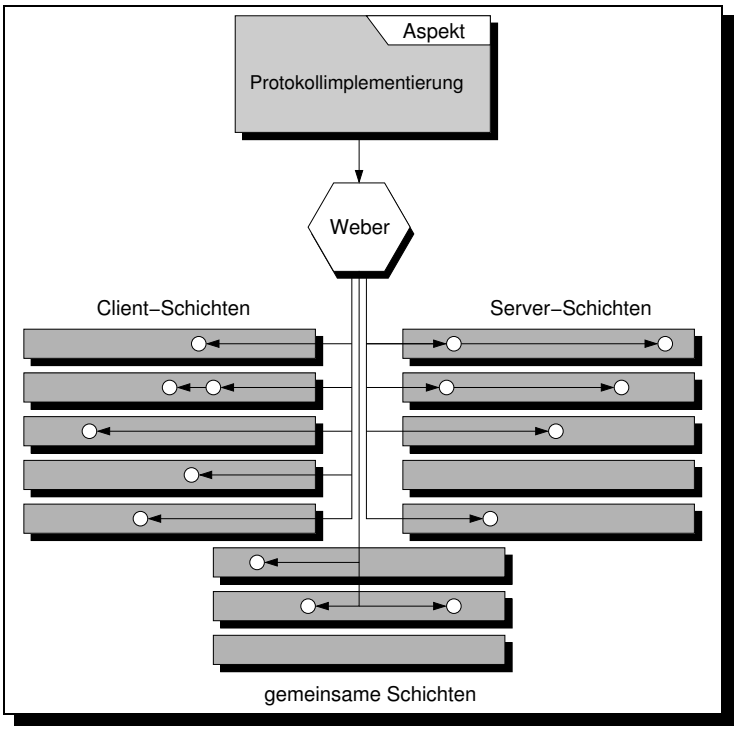

Abbildung 4.15: Protokollimplementierung als Aspekt

Applikationen möglichst transparent sein.

In Abbildung 4.15 ist dargestellt, wie eine Protokollimplementierung in einem Aspekt gekapselt ist und über einen Aspektweber in die Schichten der Middleware integriert wird. Die Kollaborationen liegen dazu in einer protokoll-unspezifischen Form vor und werden mit dem Protokoll "spezialisiert".

Die Integration eines Adapters in den Entwurf wird im Kapitel *Implementierung* gezeigt.

4.4 Dienstgütearchitektur

Ein Teil der Aufgabenstellung ist, die Integration von Dienstgüte-Mechanismen in die Middleware aufzuzeigen. Dabei wird auf in Kapitel 2 vorgestellte softwaretechnische Konzepte zurückgegriffen.

Wegen der enormen Vermaschung von Dienstgüte-Funktionalität mit Netzwerk-, Betriebssystem-, Middleware- (und evtl. Anwendungsfunktionen) spielt Aspektorientierung in Dienstgütearchitekturen eine große Rolle [Bec01]. Konzepte wie *Reflektion & Selbstadaption*, konkrete Fragestellungen zur Dienstgüte-Spezifikation oder Aushandlung von Dienstgüten etc. werden in der Arbeit nicht behandelt. Interessierte Leser werden zum Thema Reflektion auf [Smi82], [KdRB91] und [CBM+02] verwiesen. Dienstgütefragen für Middleware-Architekturen wird in den Arbeiten [ZBS97], [CS99],

[Bec01] und [HBG+01] behandelt.

Viele der folgenden Aussagen stützen sich auf die Arbeit "Dienstgüte-Management in verteilten Objektsystemen" von C. R. Becker [Bec01], in der eine Dienstgüte-Erweiterung für CORBA beschrieben wird. Trotz der Unterschiede zwischen CORBA und einer auf einem Nachrichtensystem basierenden Middleware sind viele Fragestellungen zur Dienstgüte-Integration ähnlich. In [Bec01] werden im Entwurf ebenfalls softwaretechnische Konzepte wie Objektorientierung, funktionale Hierarchie und Aspektorientierung eingesetzt und Ziele wie Wiederverwertbarkeit und Konfigurierbarkeit von Dienstgüte angestrebt. Daher bietet diese Arbeit eine gute Grundlage zur Dienstgüte-Integration in den vorliegenden Entwurf.

Becker führt in [Bec01] folgende Begriffe ein:

- Dienstgüte-Charakteristik
  "Eine *Dienstgüte-Charakteristik* ist eine quantifizierbare Einheit, durch die ein Teil der Dienstgüte-Merkmale eines Systems beschrieben wird. Der aktuelle Wert einer Dienstgüte-Charakteristik ist an eine Klient/Dienst-Interaktion gebunden." [Bec01] Eine Dienstgüte-Charakteristik muß keine unmittelbare Entsprechung im System besitzen, sondern ist als eine konzeptionelle Größe zu sehen. Beispielsweise kann die Dienstgüte-Charakteristik *Zuverlässigkeit (engl. Reliability)* unterschiedlich quantifiziert werden.

- Dienstgüte-Parameter
  Dienstgüte-Charakteristiken werden durch *Dienstgüte-Parameter* repräsentiert, wobei zwischen beiden keine Eins-zu-Eins Beziehung bestehen muß. Dienstgüte-Parameter können in Qualitätsstufen wie *gut, mittel, schlecht* oder durch systemnahe Werte wie Bandbreite oder Verzögerungszeit ausgedrückt werden. Dienstgüte-Parameter werden zur Festlegung der Soll-Werte oder der Ist-Werte (*Dienstgüte-Niveau*) einer Diensterbringung benutzt.

- Dienstgüte-Mechanismus
  Die Durchsetzung von Dienstgüte-Anforderungen im System ist Aufgabe von *Dienstgüte-Mechanismen*. Sie steuern die Einhaltung einer zwischen Client und Dienst ausgehandelten *Dienstgüte-Vereinbarung* und können an sehr unterschiedlichen Stellen im System wirken (Hardware, Betriebssystem, Netzwerk, Middleware etc.). Oft sind mehrere Dienstgüte-Mechanismen an der Bereitstellung einer konkreten Dienstgüte beteiligt.

- Dienstgüte-Vereinbarung
  Client und Dienst einigen sich vor einer Kommunikation in einer *Dienstgüte-Verhandlung* auf bestimmte Ausprägungen von Dienstgüte-Charakteristiken bzw. Dienstgüte-Parametern, mit der eine Dienstleistung erbracht wird.

4.4.1 Funktionale Hierarchie von Dienstgütemechanismen

Die Kommunikation in einem verteilten System verläuft durch mehrere funktionale Ebenen. In Abbildung 4.16 ist der übliche Pfad einer Kommunikation zwischen einem Dienstnutzer und einem Dienst schematisch dargestellt. Daraus geht hervor, daß sich auf dem Kommunikationspfad Anwendungssoftware, Middleware, Betriebssystem, Netzwerksoftware sowie Hardware befinden.

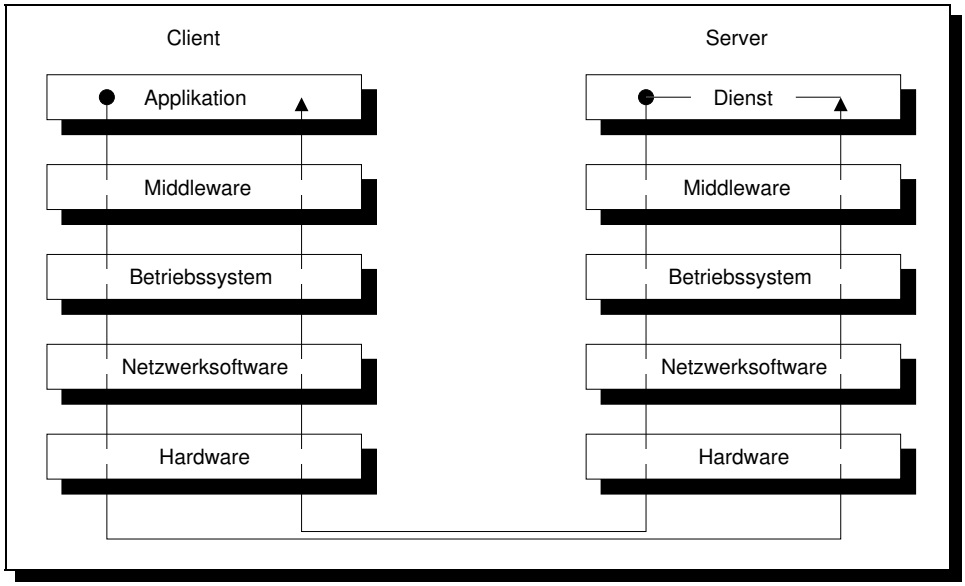

Abbildung 4.16: Kommunikationspfad in einem verteilten System

Eine Schicht benutzt dabei die Kommunikationsfunktionen tieferer Schichten. Dienstgüte-Mechanismen müssen an unterschiedlichen Stellen im System entlang des Kommunikationspfades auf das System einwirken, um eine Dienstgüte-Vereinbarung erfüllen zu können. Dazu nutzen Mechanismen höherer Schichten Mechanismen tieferer Schichten. Mit der funktionalen Hierarchie wird die Strukturierung und Wiederverwertbarkeit von Dienstgüten unterstützt. Beispielsweise können von einer Dienstgüte mehrere Spezialisierungen in höheren Schichten abgeleitet werden, die auf einer gemeinsamen funktionalen Basis beruhen. Eine besondere Schwierigkeit ergibt sich aus der Heterogenität der Zielsysteme. Dienstgüte-Mechanismen müssen an unterschiedliche Hardware, Netzwerk-Geräte, Netzwerktreiber, Betriebssysteme und an eine äußerst variabel konfigurierbare Middleware angepaßt werden.

4.4.2 Dienstgüte-Integration

Um maßgeschneiderte dienstgütefähige Middlewaresysteme zu konfigurieren, müssen die funktionalen Kommunikationsstrukturen von nicht-funktionalen Dienstgütestrukturen

getrennt bleiben. Nur so ist es möglich, später maßgeschneiderte Systeme zu konfigurieren, auch solche, die keine Dienstgüte-Befähigung besitzen oder benötigen. Neben der Integration neuer Mechanismen können auch bestehende Dienstgüte-Mechanismen genutzt werden, die z.B. von einem Netzwerk oder einem Betriebssystem bereitgestellt werden.

Wegen der schichtenübergreifenden Vermaschung von Dienstgüte-Funktionalität mit den Kommunikationsfunktionen können Dienstgüte-Mechanismen als Crosscutting Concerns identifiziert werden. Eine Modularisierung und damit auch eine Trennung funktionaler und nicht-funktionaler Strukturen der Middleware kann über eine Kapselung der Mechanismen in Aspekten unter Nutzung von AOP erreicht werden. Damit können Client und Server unabhängig von Dienstgüten entworfen und implementiert werden. Die Dienstgüte-Aspekte werden beim Konfigurationsvorgang mit den Middlewarekomponenten durch einen Aspektweber zusammengefügt.

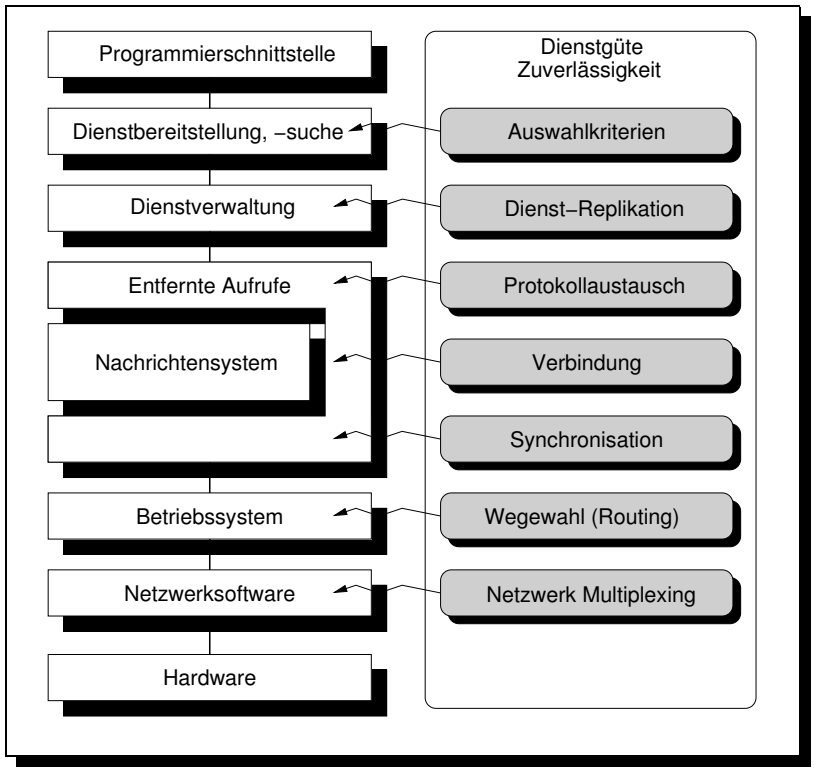

Abbildung 4.17: Crosscutting Concern “Reliability”

Dieser Sachverhalt ist für einen Client in Zeichnung 4.17 dargestellt. Die funktionale Hierarchie des Clients (Abb. 4.17 links) orientiert sich an Abbildung 4.16. Die Dienstgüte-Charakteristik *Zuverlässigkeit (engl. Reliability)* kann über verschiedene Mechanismen entlang des Kommunikationspfades implementiert werden (Abb. 4.17 rechts). (In der Abbildung wurde der Middleware-Entwurf nach [AP03a] zugrunde gelegt.) Unter *Zuverlässigkeit* werden Parameter zusammengefaßt, die beschreiben, wie fehleranfällig ein

System oder eine Kommunikation ist (vgl. Tab. 2.2, S. 36). Die Fehleranfälligkeit einer Kommunikation kann auf allen Ebenen entlang des Kommunikationspfades auch zur Laufzeit beeinflußt werden. Stehen mehrere Netzwerktypen zur Auswahl, kann die Zuverlässigkeit einer Kommunikation durch die Wahl eines Netzwerkes mit geringerer Fehlerrate erhöht werden. Falls zur Laufzeit für dieses Netzwerk noch kein Treiber installiert ist, muß dieser eingebunden werden (siehe "Netzwerk Multiplexing" in Abb. 4.17). Gleichzeitig wird die Wegewahl (engl. Routing) im Betriebssystem angepaßt, damit Pakete an das ausgewählte Netzwerk geleitet werden[5]. In den unteren Schichten der Middleware kann die Dienstgüte *Zuverlässigkeit* durch Austausch der Kollaborationen oder durch Einstellungen innerhalb der Kollaborationen statisch und dynamisch angepaßt werden. In einem Netzwerk mit hoher Fehlerrate kann eine *verbindungslose Kommunikation* weniger Fehler verursachen, als eine verbindungsorientierte, da jeder unkontrollierte Verbindungsabbruch als Fehler gewertet werden kann. Auch die Wahl der Synchronisationsstrategie und die Art des von der Middleware verwendeten Kommunikationsprotokolls können Auswirkungen auf die Zuverlässigkeit einer Kommunikation haben, wenn z.B. zwei unterschiedlich zuverlässige Dienstanbieter den gleichen Dienst jeweils über ein anderes Kommunikationsprotokoll anbieten. Protokolle, die fehlerkorrigierende Codes[6] verwenden, sind weniger störanfällig für Übertragungsfehler der Hardware. In höheren Schichten der Middleware (Dienstverwaltung) kann Zuverlässigkeit z.B. durch Redundanzmechanismen hergestellt werden: Ein Client instantiiert ein Dienstobjekt gleichzeitig auf mehreren Servern und führt Operationen transparent und parallel auf allen Objekten der *Replikatgruppe* aus (Abb. 4.18). Alle Dienstobjekte besitzen daher denselben Zustand. Fallen ein oder mehrere Dienstobjekte aus (weil sie nicht erreichbar sind etc.), wird die Arbeit auf den verbliebenen Objekten fortgesetzt[7] [Bec01].

Auch die Dienstsuche (Abb. 4.17) mit Hilfe von Dienstverzeichnissen etc. kann zugunsten von *mehr Zuverlässigkeit* eines Dienstes beeinflußt werden. Werden bei einer Dienstsuche mehrere Server gefunden, die einen gleichartigen Dienst anbieten, kann die Entscheidung auf den (oder die) Server fallen, der am besten erreichbar ist und die Dienstgüte-Anforderungen des Clients optimal unterstützt.

Es lassen sich weitere Parameter auf unterschiedlichen funktionalen Ebenen finden, um die Zuverlässigkeit entfernter Kommunikation über Dienstaufrufe zu erhöhen oder zugunsten anderer Kriterien zu vernachlässigen. Wie gut eine Dienstgüte-Charakterisitik unterstützt werden kann, hängt sehr stark davon ab, welche Dienstgüte-Mechanismen vorhanden sind, oder wie gut sie integriert werden (können). Je mehr und je bessere Mechanismen bei der Umsetzung einer Charakteristik Berücksichtigung finden, desto besser kann die Charakteristik durchgesetzt werden. Für ressourcenarme Computer wird eine Dienstgüte eher durch wenige, grundlegende Mechanismen eingesetzt, um Speicher und Rechenzeit zu sparen. Auf ressourcenreichen Geräten können sehr viel mehr Mechanismen installiert sein, um eine Dienstgütevereinbarung optimal umsetzen zu können. An-

[5] Ein OSI-Protokoll-Stack ist Teil vieler Betriebssysteme.

[6] z.B. CRC, CCITT

[7] Beim quasi-parallelen Zugriff auf mehrere Objekte können Konsistenzprobleme auftreten, die vermieden werden müssen.

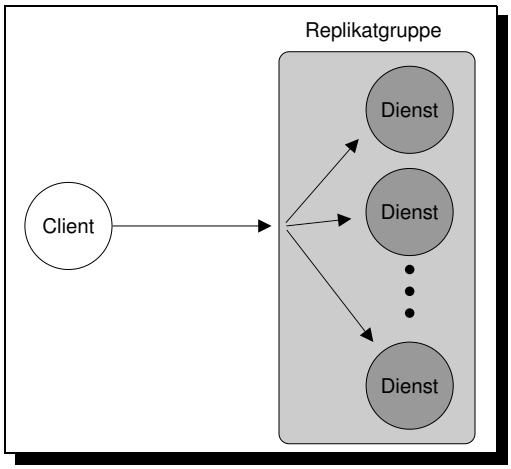

Abbildung 4.18: Beispiel eines Dienstgüte-Mechanismus [Bec01]

forderungen nach Wiederverwendbarkeit, Konfigurierbarkeit und Erweiterbarkeit stellen sich auch für Dienstgüte-Mechanismen. Die Mechanismen sollen daher in Aspekten implementiert werden.

In der Implementierung der Middleware findet dieser Abschnitt keine Berücksichtigung, da dies den Rahmen dieser Arbeit sprengen würde. Zur Integration von Dienstgüte-Mechanismen in die Middleware werden zusätzliche zentrale Komponenten benötigt, die in [Bec01] Kap. 4 beschrieben sind. Nach [AP03a] sind die zentralen Dienstgütefunktionen und Datenstrukturen erst in höheren Middlewareschichten angesiedelt, die nicht Teil der Aufgabenstellung sind.

Kapitel 5

Implementierung

In den folgenden Abschnitten wird gezeigt, wie sich die in Kapitel 3 vorgestellten Implementierungstechniken zur Umsetzung des in Kapitel 4 erarbeiteten Entwurf einer Middleware-Programmfamilie eignen. Ziel der Implementierung ist es, Kollaborationen in einer Programmiersprache zu entwickeln, die in Funktionalität, Schnittstelle und Konfigurierbarkeit den Anforderungen des Entwurfs entsprechen. Im Ergebnis dieser Arbeit wurden prototypisch Kollaborationen für Client und Server bis zur Ebene der Objektaufrufe (Schichten ROI/SROI, vgl. Abb. 4.14, S. 68) als Mixin-Schichten entwickelt, um die einfache Erstellung von Familienmitgliedern mit Hilfe ausgewählter Techniken zu demonstrieren.

Wahl der Programmiersprache

Ein wichtiger Implementierungsaspekt ist die verwendete Programmiersprache. Aufgrund der Eigenschaften der Zielsysteme, die in [AP03a] definiert sind, sind Geschwindigkeit, Ressourcenverbrauch und Portabilität wichtige Entscheidungskriterien. Aus softwaretechnischer Sicht sind die Unterstützung der vorgestellten Implementierungskonzepte für die Wahl der Sprache entscheidend. Für den Programmierer spielt Komfort in der Implementierung eine wichtige Rolle. Java und C++ sind verbreitete objektorientierte Programmiersprachen, die sich auch für große Softwareprojekte eignen. Sie unterstützen beide das objektorientierte Paradigma. Aspektorientierte Erweiterungen sind für diese beiden Objektsprachen verfügbar: AspectC++[1] und AspectJ[2]. Eine Mixin-Layer-Implementierung ist in C++ über die bereits vorgestellte parametrisierte Vererbung von Klassen möglich. Java unterstützt Mixins und Mixin-Schichten nicht. Jedoch gibt es Spracherweiterungen, die diese Funktionalität bereitstellen [AFM97].

Beide Sprachen sind mit Einschränkungen portabel. So sind C++ Compiler bzw. Crosscompiler für die Zielsysteme vorhanden. Die Portabilität von C++ sinkt jedoch bei der Verwendung von C++ Bibliotheken [AP03b]. Java-Binärcode ist auf jeder Plattform

[1] http://www.aspectc.org/
[2] http://www.aspectj.org/

ausführbar, für die eine *Virtuelle Java Maschine* sowie eine Klassenbibliothek existiert. Für die wegen ihres optimierten Ressourcenverbrauches besonders geeignete *Java 2 Platform, Micro Edition (J2ME)*[3] gibt es Virtuelle Maschinen und Bibliotheken für alle Zielplattformen außer Embedix Linux[4] ([AP03a]).

Ein Geschwindigkeitsvergleich im Rahmen dieser Arbeit zwischen je einer C++-[5] und einer Java-Implementierung[6] des SOAP-Standards hat ergeben, daß die Übertragung einer Nachricht (Serialisierung, Übertragung, Deserialisierung) zwischen zwei PDAs bzw. zwischen einem PDA und einem Desktop-Computer unter gleichen Bedingungen in Java spürbar langsamer ist als in C, durchschnittlich um den Faktor zehn. Als Implementierungssprache wurde daher C++ gewählt.

Wahl der Protokollimplementierung

Für die Zielsysteme ist *gSOAP* wegen seiner Portabilität und Schlankheit besonders geeignet [AP03a], und wird daher in dieser Arbeit als Protokollimplementierung für die Middleware-Architektur verwendet. gSOAP verwendet Werkzeuge zur Generierung von Stub und Skeleton aus der Schnittstellenbeschreibung eines Dienstes in Form einer C++ Headerdatei. Anschließend können der Dienst und der Dienstaufruf in Stub und Skeleton implementiert werden. Eine solche Programmierschnittstelle ist jedoch für die Middleware nicht geeignet, da die von einer Client-Applikation aufgerufenen und von einem Server angebotenen Dienste schon zur Compilierungszeit der Middleware bekannt sein müssen. Auch die Integration von Dienstgüte-Mechanismen wird durch die Generatorwerkzeuge nicht unterstützt, eine nachträgliche Integration der Dienstgüte-Mechanismen ist unter diesen Umständen sehr kompliziert. Daher wird die (schlecht dokumentierte) Low-Level-Schnittstelle der gSOAP-Bibliothek verwendet

Aspektsprachen-Unterstützung

Im Entwurfskapitel wird vorgeschlagen, die Protokollintegration in einer Aspektsprache zu formulieren und über einen Aspektweber in die Mixin-Schichten einzufügen. Die Kollaborationen liegen dazu als protokoll-unspezifische Kollaborationen vor. Ein Aspektweber fügt dann diese Komponenten mit den Aspekten der Protokollimplementierung zusammen und erzeugt die Kollaborationen für eine konkrete Protokollimplementierung. Für die Sprache C++ steht mit *AspectC++*[7] eine aspektorientierte Erweiterung zur Verfügung. AspectC++ unterstützt die Templateprogrammierung (noch[8]) nicht vollständig, weil keine Aspekte in parametrisierte Klassen gewoben werden können

[3] http://java.sun.com/j2me/
[4] vgl. http://www-3.ibm.com/software/wireless/wme/
[5] gSOAP: http://gsoap2.sourceforge.net
[6] kSOAP: http://ksoap.enhydra.org
[7] http://www.aspectc.org
[8] Lösungsmöglichkeiten zur Templateunterstützung werden auf der AOSD'04 vorgestellt.

[9]. Somit kann mit AspectC++ der kollaborationenbasierte Entwurf (noch) nicht unterstützt werden. Daher sind die Kollaborationen für den gSOAP-Adapter vorerst ohne Einsatz von AspectC++ implementiert. Mit Hilfe späterer Versionen von AspectC++ kann eine Reimplementierung vorgenommen werden.

5.1 Client- und Server-Schichten

Zuerst werden in den folgenden Abschnitten die unteren Schichten behandelt, die die gemeinsame Basis von Client und Server bilden (vgl. Abb. 4.14). Darauf aufbauend werden die Schichten für den Client implementiert und anschließend die Server-Schichten. Es hat sich gezeigt, daß sich die Kollaborationen und Rollen aus dem Entwurf sehr gut in C++ in Implementierungskonstrukte umsetzen ließen.

Die im Entwurf entwickelten Kollaborationen und Rollen konnten, wie von Smaragdakis und Batory in [SB02] beschrieben, oft "eins zu eins" zu Mixin-Schichten und inneren Mixinklassen modelliert werden. Da auf deren Funktionalität bereits im Entwurfskapitel eingegangen wurde, werden an dieser Stelle nur die Besonderheiten bei der Implementierung herausgearbeitet.

Viele Schichten wurden so implementiert, daß sie für unterschiedliche Protokollimplementierungen leicht wiederverwendbar sind.

Abstrakte Nachrichtenschicht

Die minimale Basis, im Entwurf *Abstrakte Nachrichtenschicht* genannt, ist als unterste Mixin-Schicht `mxMessageLayer` ohne Basisklassenparameter implementiert. (Zur späteren Unterscheidung zwischen Middleware- und Applikationsklassen besitzen die Bezeichner der Middlewareklassen den Präfix `mx`). Die Rollen *MessageBase* und *MarshalableBase* aus dem Entwurf (vgl. Abb. 4.14) werden, wie in Abbildung 5.1 dargestellt, durch die inneren Klassen `mxMessageBase` und `mxMarshalableBase` bereitgestellt. Die Schicht ist protokollunahbängig und kann für andere Protokollimplementierungen wiederverwendet werden. Im Code-Auszug in Abbildung 5.1 wird die Kapselung der inneren Klassen in eine Mixin-Schicht gezeigt, die der Entwurfsvorlage (vgl. Abb. 4.1) entspricht.

Verbindungsschicht

Die zweite im Entwurf identifizierte Schicht, die *Verbindungsschicht*, wird als Mixin-Schicht `mxConnectionLayerPremature` implementiert, die von einer *Abstrakten Nachrichtenschicht* (Parameter `MessageLayer`, Abb. 5.2, Zeilen 1-2) abgeleitet wird, z.B. von `mxMessageLayer` aus dem vorigen Abschnitt.

[9] Es können keine *Advices* für Joinpoints innerhalb von Templates oder für Zugriffe auf Templates definiert werden. http://www.aspectc.org

```
1  class mxMessageLayer {
2   public:
3    class mxMessageBase;
4    class mxMarshalableBase;
5    ..
6  };
```

Abbildung 5.1: Abstrakte Nachrichtenschicht

Die beide inneren Mixinklassen `mxConnectionMessage` und `mxConnectionMarshalable` erweitern die Rollen `mxMessageBase` und `mxMarshalableBase` aus der Schicht `MessageLayer` (Abb. 5.2) und entsprechen den gleichnamigen Rollen aus dem Entwurf. Die Vererbung der inneren Mixins wird über den Parameter `MessageLayer` gesteuert (Abb. 5.2, Zeilen 4, 10). Die Flexibilität virtueller Funktionen wurde in der Middleware, soweit möglich, mit Techniken der Template-Programmierung nachgebildet, um den mit *virtuellen Funktionstabellen* verbundenen Ressourcen-Overhead zu minimieren. Für die Methoden `marshal()` und `unmarshal()` in `mxConnectionMarshalablePremature` (Abb. 5.2) war dies nicht möglich. Sie werden später durch entsprechende Methoden konkreter Marshalables überlagert.

Die im Entwurf modellierte Rolle *Connection* wird über den Template-Parameter `CONNECTION` für die Schicht spezifiziert (Abb. 5.2, Zeile 1). Diese Schicht ist frei von protokollspezifischen Anweisungen und für unterschiedliche Verbindungsarten und Protokollimplementierungen wiederverwertbar.

```
 1  template <class MessageLayer, class CONNECTION>
 2  class mxConnectionLayerPremature : public MessageLayer {
 3  public:
 4    class mxConnectionMarshalable : public MessageLayer::mxMarshalableBase {
 5      CONNECTION *m_connection;
 6      virtual int marshal() { return OK; }
 7      virtual int unmarshal() { return OK; }
 8      ...
 9    };
10    class mxConnectionMessage : public MessageLayer::mxMessageBase {
11     protected:
12      CONNECTION *m_connection;
13     public:
14      void setConnection(CONNECTION* connection) {m_connection = connection;}
15      ...
16    };
17  };
```

Abbildung 5.2: Parametrisierte Vererbung der Verbindungsschicht

Bevor die Kollaboration verwendet werden kann, muß sie mit einer konkreten Verbindung instantiiert werden. In Abbildung 5.3 ist dargestellt, wie die-

se Schicht mit einer `mxGsoapConnection` (einer gSOAP-Verbindung) parametrisiert wird. Die `mxGsoapConnection` wird durch eine *protokollspezifische* Erweiterungsschicht `mxConnectionLayer` bereitgestellt, die ebenfalls als Mixin-Schicht implementiert ist (Abb. 5.3). Die gSOAP-Verbindungsschicht `mxConnectionLayer` wird verwendet, um die allgemeine Verbindungsschicht `mxConnectionLayerPremature` so mit dem Merkmal *gSOAP-Protokoll* zu spezialisieren (5.3, Zeilen 5-6). In Abbildung 5.4 ist dieser Sachverhalt dargestellt[10]. Diese Spezialisierungsschicht wird wie die allgemeine Schicht ebenfalls mit einer *Abstrakten Nachrichtenschicht* `MessageLayer` (Abb. 5.3, Zeile 4) parametrisiert, und reicht diesen Parameter zusammen mit der gSOAP-Verbindung an die allgemeine Verbindungsschicht weiter (5.3, Zeile 6). Die gSOAP-Verbindungsschicht wird daher als eigentliche *Verbindungsschicht* verwendet, die die allgemeine Verbindungsschicht überdeckt. Dieser Sachverhalt ist für höhere Schichten transparent, da die allgemeine Schicht durch die protokollspezifische Schicht automatisch eingebunden wird (Abb. 5.3, Zeile 1). Höhere Schichten verwenden immer die protokollspezifische Schicht, die nur noch mit einer Nachrichtenschicht (aber nicht mehr mit einer protokollspezifischen Verbindungsklasse) parametrisiert wird (siehe auch Abb. 5.4). Die Verbindungsklasse `mxGsoapConnection` wird höheren Schichten unter einem protokoll-unspezifischen Bezeichner `mxConnection` bereitgestellt (Abb. 5.3, Zeile 7).

Die Integration mehrerer dynamisch austauschbarer Verbindungsarten konnte wegen fehlender Unterstützung durch die gSOAP-Protokollimplementierung nicht realisiert werden. Für andere Protokollimplementierungen kann dazu eine Implementierungsvariante verwendet werden, die im Abschnitt *Synchronisationsschicht* erläutert wird.

```
1  #include "mxConnectionLayerPremature.h"
2  #include "mxGsoapConnection.h"
3
4  template <class MessageLayer>
5  class mxConnectionLayer :
6    public mxConnectionLayerPremature<MessageLayer, mxGsoapConnection> {
7    typedef mxGsoapConnection mxConnection;
8  };
```

Abbildung 5.3: Instantiierung der Verbindungsschicht

Parameterschicht

Auch die *Parameterschicht* kann so implementiert werden, wie im Entwurf vorgesehen. Die Schicht wird wie die *Verbindungsschicht* durch eine allgemeine und eine protokollspezifische Mixin-Schicht implementiert. Die allgemeine Schicht ist in Abbildung

[10]Da in der Abbildung ein Klassendiagramm dargestellt ist, sind funktional “tiefere” Schichten als Basisschichten “oben” angeordnet.

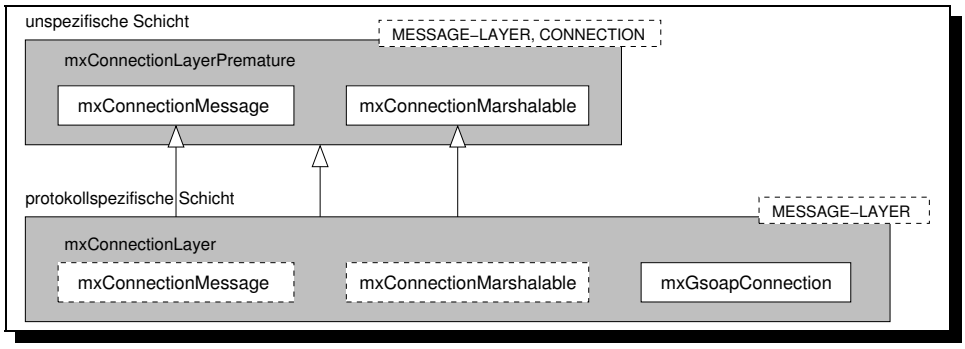

Abbildung 5.4: Allgemeine und protokollspezifische Schicht

5.5 dargestellt. Hauptziel der Implementierung dieser Schicht ist, Marshalables typisieren zu können. Dazu wird das aus der Basisschicht (`ConnectionLayer`) geerbte `mxConnectionMarshalable` mit einem Typ (`TYPE`) parametrisiert (Abb. 5.5, Zeile 4).

```
 1  template <class ConnectionLayer>
 2  class mxParamLayerPremature : public ConnectionLayer {
 3   public:
 4    template <class TYPE>
 5    class mxTypedMarshalable :
 6        public ConnectionLayer::mxConnectionMarshalable {
 7     protected:
 8      TYPE m_value;
 9    ... };
10
11    class mxParamMessage :
12        public ConnectionLayer::mxConnectionMessage {
13     public:
14      // einer Nachricht einen Parameter (Marshalable) hinzufuegen
15      int addParam(ConnectionLayer::mxConnectionMarshalable *cmarsh);
16    ... };
17  ... };
```

Abbildung 5.5: Mixin-Implementierung der Parameterschicht

Um mehrere de-/serialisierbare Typen bereitzustellen, werden die geerbten `mxTypedMarshalables` mit konkreten Typen parametrisiert. Dazu wird von dieser Schicht eine protokollspezifische Schicht `mxParamLayer` abgeleitet, die die ursprüngliche `mxParamLayerPremature` mit einem Protokoll spezialisiert (Abb. 5.6, Zeilen 6-7). In Abbildung 5.6 ist eine mögliche Konfiguration dieser Schicht mit drei gSOAP-Marshalables dargestellt, die unter protokoll-unspezifischen Bezeichnern für höhere Schichten bereitgestellt werden (5.6, Zeilen 9-11). Neben einfachen Datentypen können in höheren Schichten auch komplexe Datentypen definiert werden, die sich aus Grunddatentypen zusammensetzen. Sie implementieren dazu die Methoden `marshal()` und `unmarshal()`, indem sie ihre Bestandteile nacheinander de-/serialisieren, d.h. deren `marshal()`- bzw.

`unmarshal()`-Methoden der Reihenfolge nach aufrufen. Da komplexe Datentypen so dieselbe Schnittstelle implementieren wie Grunddatentypen, können sie von Nachrichten wie Grunddatentypen behandelt werden.

```
 1  #include "mxParamLayerPremature.h"
 2  #include "mxGsoapInt.h"
 3  #include "mxGsoapFloat.h"
 4  #include "mxGsoapString.h"
 5
 6  template <class ConnectionLayer>
 7  class mxParamLayer : public mxParamLayerPremature <ConnectionLayer> {
 8   // mit dieser Schicht werden drei Datentypen bereitgestellt
 9   typedef mxGsoapInt    mxInt;
10   typedef mxGsoapFloat  mxFloat;
11   typedef mxGsoapString mxString;
12  };
```

Abbildung 5.6: Spezialisierung der Parameterschicht mit einem Protokoll

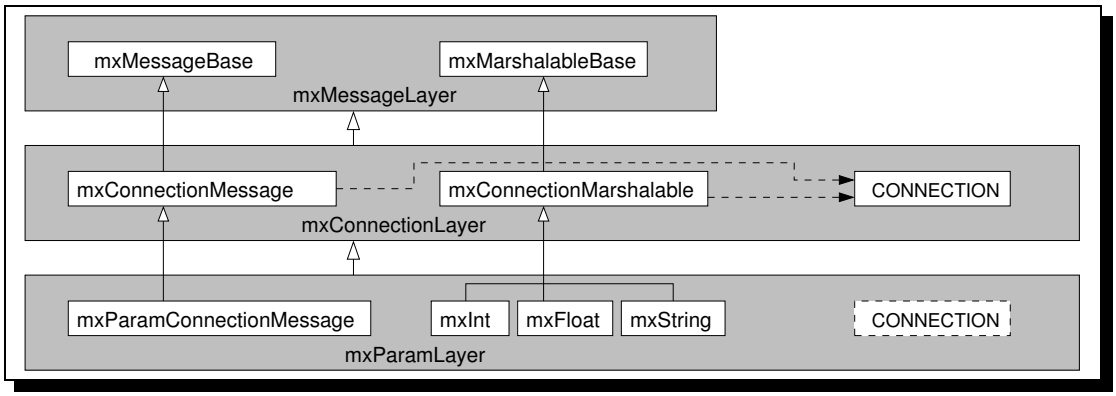

Abbildung 5.7: Implementierung der untersten drei Kollaborationen

Um die weiteren Erläuterungen nicht unnötig kompliziert zu machen, werden die allgemeinen und spezifischen Mixin-Schichten einer Kollaboration im Folgenden nur als eine Schicht benannt. Die hohe Ähnlichkeit, die zwischen Entwurf und Implementierung innerhalb der untersten drei Kollaborationen erreicht wird, ist in Abbildung 5.7 zu erkennen (vgl. Abb. 4.1, S. 52). Die *Parameterschicht* ist hier in einer Konfiguration mit drei Datentypen dargestellt, andere Konfigurationen sind möglich.

Richtungstrennungsschicht

Innerhalb der Richtungstrennungsschicht werden von der `mxParamMessage` die Klassen `mxOutMessage` und `mxInMessage` abgeleitet (Abb. 5.8, Zeilen 4, 9). Die Implementierung

dieser Klassen ist eng mit der Protokollimplementierung verbunden. Sie sind in der Lage, sich zu serialisieren bzw. zu deserialisieren. Die in Abbildung 5.8, Zeilen 6 und 11 dargestellten Methoden `marshal()` und `unmarshal()` serialisieren bzw. deserialisieren eine gesamte Nachricht[11]. Nachdem der Nachrichtenkopf de-/serialisiert wurde, werden die `marshal()`- bzw. `unmarshal()`-Methoden der zur Nachricht gehörenden Marshalables der Reihe nach aufgerufen.

```
 1  template <class ParamLayer>
 2  class mxSeparationLayer : public ParamLayer {
 3  public:
 4    class mxOutMessage : public ParamLayer::mxParamMessage {
 5     public:
 6      int marshal();
 7      ...
 8    };
 9    class mxInMessage : public ParamLayer::mxParamMessage {
10     public:
11      int unmarshal();
12    ...
13  };
```

Abbildung 5.8: Mixin-Implementierung der Richtungstrennungsschicht

Synchronisationsschicht

Die Integration unterschiedlicher Synchronisationsstrategien ist bei Verwendung der gSOAP-Protokollimplementierung kompliziert, da die De-/Serialisierung direkt auf einem Datenstrom durchgeführt wird. Die quasi gleichzeitig stattfindende Nachrichtenübertragung ist synchron. Die Synchronisationsstrategie wird in einer Kollaboration gekapselt, jedoch sind die Synchronisationsfunktionen `send()` und `recv()` bedeutungslos und leer, da eine Nachricht schon durch `marshal()` und `unmarshal()` (synchron) gesendet bzw. empfangen wird. Bei Verwendung der gSOAP-Protokollimplementierung kann daher vorerst nur das Merkmal *synchroner Nachrichtenaustausch* bereitgestellt werden.

Im Folgenden wird eine allgemeine, im Rahmen dieser Arbeit entwickelte Möglichkeit aufgezeigt, wie Strategien und insbesondere Synchronisationsstrategien wiederverwendbar implementiert werden können. Diese Technik kann beispielsweise zur Implementierung von Dienstgütestrategien, oder für Verbindungsstrategien und Synchronisationsstrategien für Protokollimplementierungen genutzt werden, wo dies besser möglich ist als bei gSOAP.

[11] nicht zu verwechseln mit mxMarshalable::(un)marshal(), welche ein Marshalable de-/serialisieren

Implementierung wiederverwendbarer Strategien

In der Domänenanalyse wurde festgestellt, daß eine Synchronisationsstrategie für einen Kontext[12] zur Compilierungszeit oder zur Laufzeit austauschbar sein soll. Diese Forderung kann auch für andere Strategien wie z.B. Dienstgütestrategien, Verbindungsstrategien, Speicherstrategien etc. von Bedeutung sein. Die Bindung der Funktionen einer Strategie an einen Kontext sollte dabei ebenfalls entweder zur Compilierungszeit oder zur Laufzeit erfolgen. Für frühe oder späte Kontextbindung muß eine Strategie jeweils eine andere Schnittstelle implementieren, da spät gebundene Methoden als *virtuell* deklariert sind. Dies erschwert die Wiederverwendbarkeit einer Strategie für unterschiedliche Bindungsarten. Mit Hilfe der Templateprogrammierung kann die Bindungseigenschaft zwischen Kontext und Strategie zur Compilierungszeit variiert werden.

Die Verwendung des Begriffes *Strategie* bezieht sich auf das bekannte Entwurfsmuster (engl. Design Pattern) *Strategie*, das in [GHJV95] beschrieben wird[13]. Das Entwurfsmuster *Strategie* bietet eine objektorientierte Lösung an, wie das *Verhalten* eines Objektes von seiner *Schnittstelle* getrennt werden kann, um zwischen gleichartigen Objekten mit unterschiedlichem Verhaltensvarianten umschalten zu können. Im Fall *Synchronisation* sollen Nachrichten mit Hilfe unterschiedlicher Synchronisationsstrategien (mit gleicher Schnittstelle) versendet oder empfangen werden können. Eine jede Strategie wird daher als Objekt gekapselt und unter einer einheitlichen Schnittstelle angeboten: `send(void *buf)` sendet eine Nachricht, die in einem Puffer abgelegt ist entsprechend der Strategie, `recv(void *buf)` empfängt eine Nachricht und legt sie in einem Puffer ab.

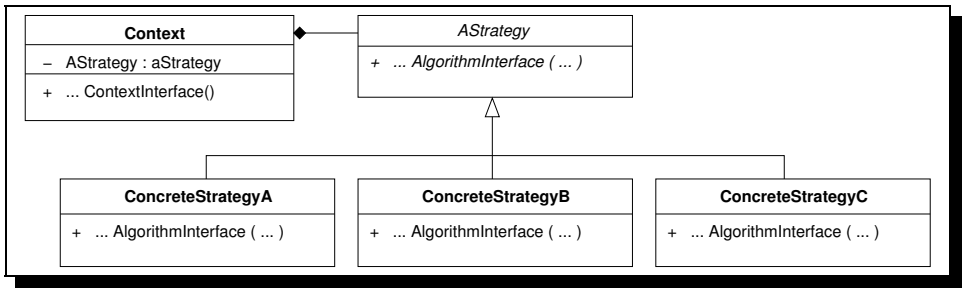

Abbildung 5.9: Das Entwurfsmuster *Strategie* [GHJV95]

In Abbildung 5.9 sind drei verschiedene Strategieklassen dargestellt, die von einem Kontext `Context` unter einer gemeinsamen Schnittstelle verwendet werden können. Der Kontext besitzt dazu ein Schnittstellenobjekt `AStrategy`, das in der Abbildung über eine Vererbungsbeziehung mit einem konkreten Strategieobjekt verbunden ist. Für die Implementierung dieses Musters in C++ bieten sich zwei verschiedene Varianten an: eine

[12] Ein Kontext kann z.B. eine Klasse oder ein Objekt sein, das eine Strategie besitzt oder kennt und benutzt.

[13] Unter dem Begriff *Entwurfsmuster* werden Lösungen für häufig auftretende Probleme aus der Softwareentwicklung zusammengefaßt.

mit früher Kontextbindung mit Hilfe von *Template*-Programmierung und die zweite mit später Kontextbindung, wie in [GHJV95] vorgeschlagen:

(A) Die Auswahl einer konkreten Strategie erfolgt zur Compilierungzeit. Der Kontext (die Kontextklasse) kann dazu mit einer Strategie (einer Strategieklasse) mit Hilfe des Template-Mechanismus parametrisiert werden. Jede Strategie implementiert eine festgelegte Strategieschnittstelle. Ein separates Schnittstellenobjekt, wie in Abbildung 5.9 dargestellt, ist nicht notwendig. Zwischen Kontext und Strategie wird eine *frühe Bindung* hergestellt, ein Strategiewechsel ist zur Laufzeit nicht möglich.

(B) Die Auswahl einer konkreten Strategie erfolgt zur Laufzeit. Der Kontext muß ein Schnittstellenobjekt besitzen (vgl. Abb. 5.9), dessen Implementierung zur Laufzeit geändert werden kann. In [GHJV95] wird dazu vorgeschlagen, eine Vererbungsbeziehung zwischen Schnittstellenobjekt und konkreter Strategieimplementierung herzustellen, wobei die Implementierung einer Strategie mit Hilfe *virtueller Methoden spät* an eine Schnittstelle gebunden wird. Dadurch ist die Implementierung der Schnittstelle zur Laufzeit austauschbar. Diese Flexiblität ist jedoch mit einem höheren Laufzeit-Overhead verbunden.

Variante A vermeidet späte Bindungen und spart damit Ressourcen, die Strategie kann jedoch zur Laufzeit nicht rekonfiguriert werden. Variante B ermöglicht einen Strategieaustausch zur Laufzeit, jedoch unter höherem Ressourcenverbrauch. Da eine spät gebundene Strategie eine andere Schnittstelle verwendet[14] als eine früh gebundene, kann eine Strategie nur für jeweils eine Bindungsart wiederverwendet werden. Ziel ist jedoch die Verwendung einer Strategie-Implementierung für beide Bindungsarten, damit ein Kontext entweder die Vorteile der frühen oder der späten Bindung nutzen kann.

Das Problem kann gelöst werden, indem eine dynamisch austauschbare Strategie über ein Vermittlerobjekt angesprochen wird, das zwischen Kontext und Strategie auftritt (siehe Abb. 5.10). Der Kontext verwendet das Vermittlerobjekt, als wäre es selbst eine Strategie. Dazu muß sich der Vermittler gegenüber dem Kontext mit der Schnittstelle der Strategien präsentierten.

Zwei Synchronisationsstrategien `mxSyncStrategy` und `mxAsyncStrategy` sind schnittstellengleich implementiert (siehe Abb. 5.11) und dienen in den folgenden Erläuterungen als Beispiele.

In Abbildung 5.12 ist ein Vermittler `mxMessenger` (Zeile 3) kodiert, der einen Zeiger auf ein Strategieobjekt vom Typ `mxDynamicStrategy` besitzt (Zeile 5). Er implementiert dieselbe Schnittstelle - `send(void* buf)` und `recv(void* buf)` - wie die Strategien `mxAsyncStrategy` und `mxSyncStrategy` (Abb. 5.11), zwischen denen er umschalten kann. Zusätzlich besitzt der Vermittler eine Methode

[14] Spät gebundene Methoden werden als *virtual* deklariert [Str00].

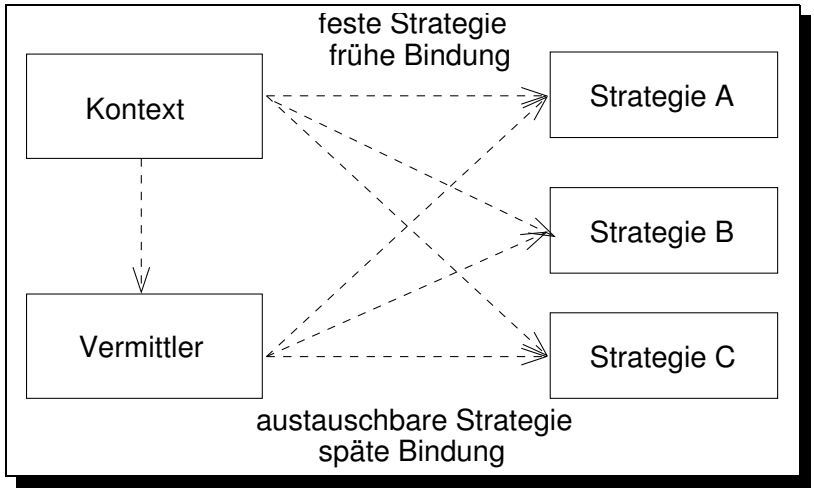

Abbildung 5.10: Vermittler zwischen Kontext und Strategien

```
1  // Strategie fuer synchrones Senden aus, Empfangen in einen Buffer
2  class mxSyncStrategy {
3   public:
4    int send(void* buf) { /* Synchrones Senden    */ };
5    int recv(void* buf) { /* Synchrones Empfangen */ };
6  };
7
8  // Strategie fuer asynchrones Senden aus, Empfangen in einen Buffer
9  class mxAsyncStrategy {
10  public:
11   int send(void* buf) { /* Asynchrones Senden     */ };
12   int recv(void* buf) { /* Asynchrones Empfangen  */ };
13 };
```

Abbildung 5.11: Zwei Synchronisationsstrategien

`setStrategy(mxDynamicStrategy* s)` (Zeile 10), über die das verwendete Strategieobjekt ausgetauscht wird. Die Klasse `mxDynamicStrategy` ist in Abbildung 5.13 dargestellt und besitzt ebenfalls dieselbe Schnittstelle wie die Strategien und der Messenger: `send(void* buf)` und `recv(void* buf)`.

```
1  // Vermittler zur dynamischen Rekonfigurierung der Synchronisationsstrategie
2
3  class mxMessenger{
4  protected:
5    mxDynamicStrategy* m_strategy;
6  public:
7    int send(void* buf) { return m_strategy->send(buf); }
8    int recv(void* buf) { return m_strategy->recv(buf); }
9    // besitzt zusaetzlich die Methode setStrategy()
10   void setStrategy(mxDynamicStrategy* s) { m_strategy = s; };
11 };
```

Abbildung 5.12: Strategie-Vermittler (Messenger)

```
1  // optionale virtuelle Basisklasse fuer eine Strategie
2  class mxDynamicStrategy {
3  public:
4    virtual int send(void* buf) = 0;
5    virtual int recv(void* buf) = 0;
6  };
```

Abbildung 5.13: Strategie-Basisklasse für späte Bindung

Um die beiden Strategien `mxAsyncStrategy` und `mxSyncStrategy` als Parameter der Methode `setStrategy(s)` des Vermittlers verwenden zu können, müssen sie als `mxDynamicStrategy` typisiert werden. Dies kann erreicht werden, indem eine Strategieklasse von `mxDynamicStrategy` abgeleitet wird. Wegen der Schnittstellengleichheit beider Klassen erben die Methoden der Strategieklasse die späte Bindung von den Methoden der Basisklasse, die als *virtuell* deklariert sind (vgl. Abb. 5.13). Dazu wird die Schnittstelle der Strategien geringfügig geändert, so daß sie von einer unbekannten Basisklasse abgeleitet werden (Abb. 5.14).

Somit kann eine Strategie mit Hilfe parametrisierter Vererbung von der virtuellen Basisklasse abgeleitet werden. Dieser Sachverhalt ist in Abbildung 5.15 in den Zeilen 1 bis 5 dargestellt. Die spät gebundenen Strategien können nun als Parameter für die Methode `mxMessenger.setStrategy(s)` verwendet werden (Abb. 5.15, Zeilen 8-10). Ein Kontext, der anstelle einer konkreten Strategie einen Messenger besitzt, kann diesen wie eine Strategie verwenden und besitzt zudem die Fähigkeit, die Strategie zu wechseln.

Ist eine dynamische Rekonfigurierung von Strategien nicht erwünscht, wird der Messenger nicht benötigt. Die Strategien werden in diesem Fall früh an den Kontext gebunden.

```
 1   template <class BINDING>
 2  class mxSyncStrategy : public BINDING {
 3   public:
 4    int send(void* buf) { /* Synchrones Senden    */ };
 5    int recv(void* buf) { /* Synchrones Empfangen */ };
 6  };
 7
 8  template <class BINDING>
 9  class mxAsyncStrategy : public BINDING {
10   public:
11    int send(void* buf) { /* Asynchrones Senden     */ };
12    int recv(void* buf) { /* Asynchrones Empfangen  */ };
13  };
```

Abbildung 5.14: Zwei Synchronisationsstrategien mit Bindungsparameter

```
 1  // eine spaet gebundene synchrone Strategie
 2  mxSyncStrategy <mxDynamicStrategy>  sync_late;
 3
 4  // eine spaet gebundene asynchrone Strategie wird dynamisch erzeugt
 5  mxDynamicStrategy* async_late = new mxAsyncStrategy<mxDynamicStrategy>();
 6
 7  // Strategien koennen dem Messenger zugewiesen werden
 8  mxMessenger messenger;
 9  messenger.setStrategy(&sync_late);
10  messenger.setStrategy(async_late);
11
12  delete async_late;
```

Abbildung 5.15: Dynamische Rekonfigurierung von Strategien

In Abbildung 5.16 ist dargestellt, wie die Synchronisationsstrategien mit einer leeren Basisklasse `mxStatic` (Zeile 2) parametrisiert werden (Zeilen 5, 8). Die Strategiemethoden `send` und `recv` erben von `mxStatic` keine Bindungseigenschaft, sondern behalten ihre frühe Bindung bei. Ein Kontext, der eine konkrete Strategie anstelle eines Messengers besitzt, kann seine Strategie nicht wechseln, verwendet daür eine fühe Bindung.

```
1  // leere Basisklasse fuer eine Strategie
2  class mxStatic {};
3
4  // eine frueh gebundene synchrone Strategie
5  mxSyncStrategy <mxStatic>  sync_early;
6
7  // eine frueh gebundene asynchrone Strategie
8  mxAsyncStrategy<mxStatic>  async_early;
```

Abbildung 5.16: Früh gebundene Strategien

In Abbildung 5.17 ist das Klassendiagramm der Strategieimplementierung mit drei Strategievarianten dargestellt. Über den Parameter `BINDING` wird die Basisklasse einer Strategie konfiguriert. Aus der Abbildung wird deutlich, daß Strategien für frühe und späte Bindungen wiederverwendet werden können.

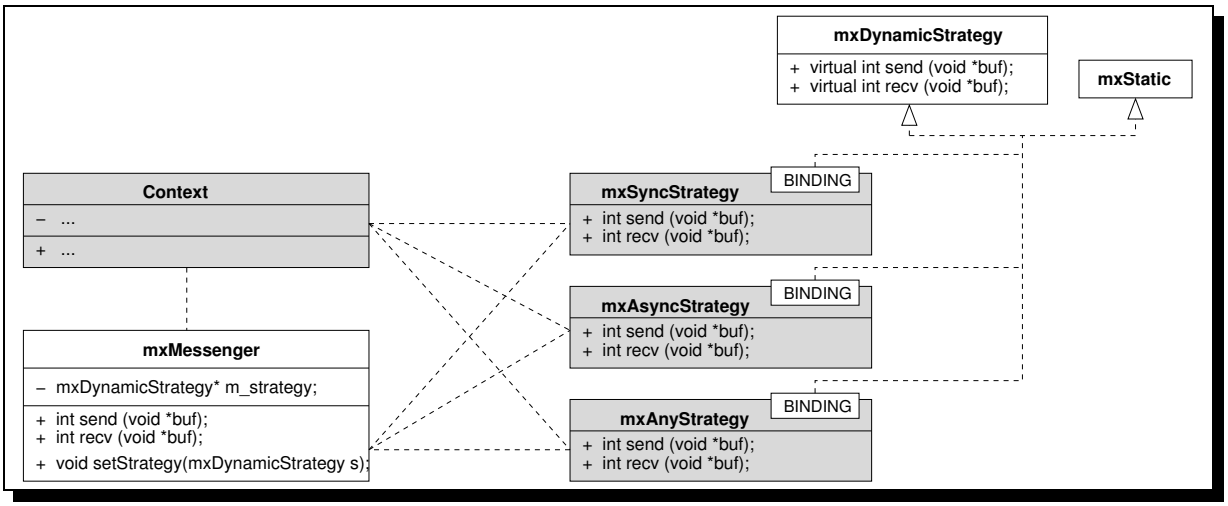

Abbildung 5.17: Klassendiagramm zur Implementierung von Strategien

Um einen Kontext mit einer Strategie *oder* mit einem Vermittler zu konfigurieren, wird eine Kontextklasse entweder mit einer Strategie oder mit dem Vermittler parametrisiert. In Abbildung 5.18 ist als Kontext der Synchronisationsstrategien eine Nachricht (`mxMessage`) gewählt worden, die entsprechend einer Strategie versendet oder empfangen werden soll. Die Strategie wird hier als Basisklasse für den Kontext (die Nachricht)

verwendet. Ebenso kann die Strategie als Typ *in* einem Kontext eingesetzt werden. In Abbildung 5.18, Zeile 7 wird eine Nachricht mit früher Strategiebindung für eine synchrone Kommunikation instantiiert. Der Kontext `message2` in Zeile 11 kann seine Strategie zur Laufzeit austauschen, wie in den darauf folgenden Code-Zeilen demonstriert wird.

```
1  template <class STRATEGY>
2  class mxMessage : public STRATEGY {
3   // ...
4  };
5
6  // Nachricht mit frueh gebundener Strategie
7  mxMessage<mxSyncStrategy<mxStatic>> message1;
8  message1.send();                          // synchron senden
9
10 // Nachricht mit spaet gebundenen Strategien
11 mxMessage<mxMessenger> message2;
12 mxSyncStrategy<mxDynamicStrategy> sync;
13 mxAsyncStrategy<mxDynamicStrategy> async  // zwei spaet gebundene Strategien
14 message2.setStrategy(&async);
15 message2.send();                          // synchron senden
16 if (condition) {                          // wenn Bedingung erfuellt
17   message2.setStrategy(&sync);            // Strategie umsetzen
18 }
19 message2.recv();                          // und asynchron empfangen
```

Abbildung 5.18: Konfigurierung des Kontext

Zusammenfassung

Ein Kontext kann bei zwei Strategien in sechs Varianten konfiguriert werden (siehe Tab. 5.1), wobei die Varianten 2 und 4, die Verwendung eines Vermittlers bei einer Strategie, keine praktische Relevanz haben.

1	Synchron		
2	*Synchron*		*Vermittler*
3		Asynchron	
4		*Asynchron*	*Vermittler*
5	Synchron	Asynchron	
6	Synchron	Asynchron	Vermittler

Tabelle 5.1: Konfigurationsmöglichkeiten für Strategien

Die Verwendung einer Strategie oder eines Vermittlers ist für den Kontext transparent. Wird der Kontext mit einer konkreten Strategie statisch konfiguriert, kann die Strategie nicht ausgetauscht werden. Der Aufruf der Strategiemethoden erfolgt nicht zu Lasten der Effizienz, und es werden keine zusätzlichen Ressourcen verwendet.

Bei einer dynamischen Konfiguration kann die Strategie eines Kontext zur Laufzeit ausgetauscht werden. Dabei werden zusätzliche Ressourcen in Anspruch genommen. In beiden Konfigurationen werden durch den Kontext dieselben Strategieimplementierungen verwendet.

Diese Implementierungsvariante des *Strategy Pattern* wird verwendet, um das Server-Merkmal *Receive Call* sowie die Client-Merkmale *OneWay* und *TwoWay Receive Call* mit einer Synchronisationsstrategie zu konfigurieren, wie im nächsten Abschnitt gezeigt wird. Die Technik kann aber auch für beliebige andere Strategien universell eingesetzt werden.

5.2 Die Clients

Auch die Client-Merkmale konnten nach den Vorstellungen des Entwurfs als Mixin-Schichten implementiert werden, ähnlich der bereits gezeigten Mixin-Schichten.

OneWay und TwoWay

Die Merkmale *OneWay* und *TwoWay* werden in aufeinander aufbauenden Kollaborationen implementiert, wobei sie als `mxOneWay` und `mxTwoWay` als inneren Mixinklassen implementiert sind. Damit ein `mxOneWay` bzw. ein `mxTwoWay` unterschiedliche Synchronisationsstrategien aus der *Synchronisationsschicht* verwenden kann, werden diese Klassen - wie in Abbildung 5.19, Zeilen 2, 5 dargestellt - mit einem Messenger oder einer Strategie parametrisiert. Anders als im Entwurf vorgesehen, ist die Synchronisationsstrategie kein Aggregat, sondern eine Basisklasse des `mxOneWays` bzw. `mxTwoWays`. Auch die Beziehung zwischen einem `mxOneWay` und einer *OutMessage* ist keine Aggregationsbeziehung, sondern die *OutMessage* ist eine Basisklasse des `mxOneWay`. Ebenso ist der `mxTwoWay` nicht nur von `mxOneWay` abgeleitet, sondern auch von einer *InMessage* (siehe Abb. 5.19, 5.20). Damit erben die `mxOneWay`- und `mxTwoWay`-Schnittstellen erstens die Methoden der Basisklassen und zweitens deren Konfigurierungsmöglichkeiten, z.B. die Methode `setStrategy()`.

Die Klassen `mxOneWay` und `mxTwoWay` können, wie in Abbildung 5.20 dargestellt, mit unterschiedlichen Nachrichten parametrisiert werden. Die Implementierung der Rollen *OneWayFunctionCall*, *TwoWayFunctionCall*, *OneWayClassCall*, *TwoWayClassCall*, *OneWayObjectCall* und *TwoWayObjectCall* kann damit sehr elegant gelöst werden, wie im folgenden Abschnitt gezeigt wird.

Serviceparadigma

Die Client-Schichten *RPC*, *RCI* und *ROI* werden jeweils als Mixin-Schicht implementiert. Die Nachrichtenrollen der *RPC*-Schicht `mxOutFunctionMessage` und `mxInFunctionMessage` sind in der Lage, eine Nachricht für einen entfernten Funktionsaufruf zu senden bzw. zu empfangen. Entsprechend können `mxOutClassMessage` und

```
1  // ein synchroner OneWay
2  mxOneWay<mxOutMessage, mxSyncStrategy<mxStatic> > oneWaySync;
3
4  // ein TwoWay mit austauschbarer Synchronisationsstrategie
5  mxTwoWay<mxOutMessage, mxInMessage, mxMessenger> twoWayUniversal;
6  ...
7  // ein OneWay/TwoWay erbt die Strategie(methoden)
8  twoWayUniversal.setStrategy(async)
9  // call() ist eine Funktion von mxOneWay und mxTwoWay
10 twoWayUniversal.call();
11 oneWaySync.call();
```

Abbildung 5.19: Instantiierung von mxOneWay und mxTwoWay

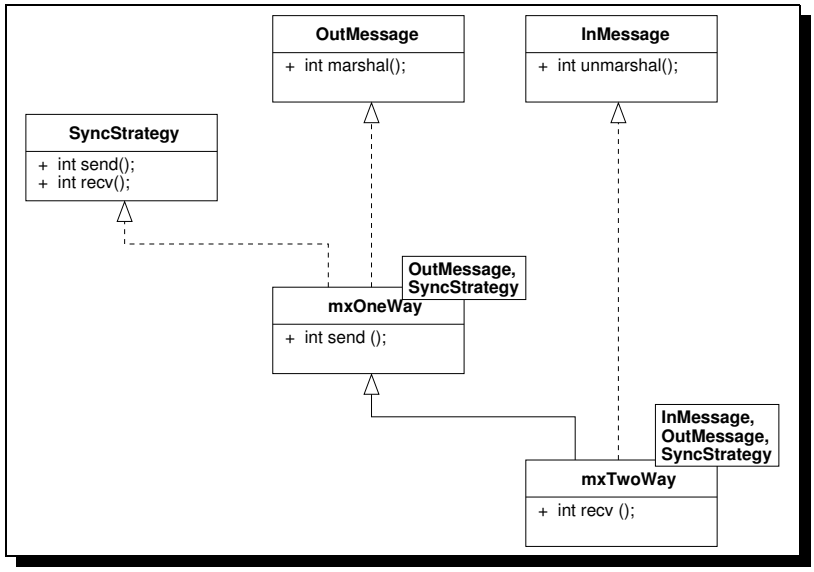

Abbildung 5.20: Klassenbeziehungen von mxOneWay und mxTwoWay

`mxInClassMessage` einen Klassenfunktionsaufruf senden bzw. empfangen usw. Die *OutMessages* und *InMessages* dieser Schichten erweitern dazu jeweils die Funktionen ihrer Basisklassen, um den Empfänger anzugeben bzw. den Absender zu ermitteln. Dieser Sachverhalt ist in einem Klassendiagramm ist in Abbildung 5.21 dargestellt.

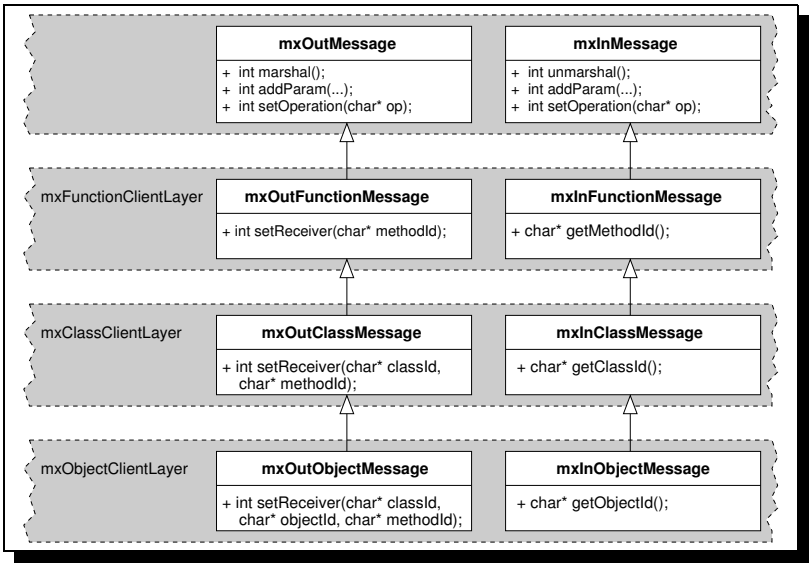

Abbildung 5.21: Klassendiagramm der Nachrichten-Mixins

Innerhalb der Klassen `mxOneWay` und `mxTwoWay` werden nur Methoden wie `marshal()` und `unmarshal()` aus der Richtungstrennungsschicht und Methoden noch tieferer Schichten aufgerufen. Daher können die OneWay- und TwoWay-Rollen der *RPC*-, *RCI*- und *ROI*-Schicht per Parametrisierung der Klassen `mxOneWay` und `mxTwoWay` mit einer *InMessage* oder *OutMessage* aus der *RPC*-, *RCI* bzw. *ROI*-Schicht erzeugt werden. Um einen Überblick über die Konfigurationsmöglichkeiten zu erhalten, wird in Abbildung 5.22 dargestellt, wie eine Client-Applikation für das Beispiel "Lebenszyklus eines entfernten Objektes" (Abb. 4.9 S. 61) implementiert werden kann. In der Abbildung wird in den Zeilen 1 bis 14 deutlich, wie beispielsweise mit Hilfe von `typedefs` komplette Nachrichtenmuster sehr elegant erstellt werden können (vgl. Abb. 2.1 für Nachrichtenmuster). Der Beispiel-Client `ExampleClient` in Abbildung 5.22, Zeile 17 ist als Mixin-Schicht überhalb der *ROI*-Schicht implementiert und wird *mit* einer Middleware parametrisiert (Zeilen 16-17). Die Middleware ist damit für eine Applikation leicht austauschbar, z.B. um die Protokollimplementierung auszutauschen[15]. Der Austausch einer Middleware ist in Abbildung 5.23 dargestellt: Eine Applikation `mxExampleClient` wird in den Zeilen 8 und 11 jeweils mit einer anderen Middleware instantiiert und gestartet.

[15]Wie später in diesem Kapitel gezeigt wird, gibt es auch andere Möglichkeiten, um die Protollimplemetierung für eine Applikation zu konfigurieren.

```
 1  typedef mxSyncStrategy<mxDynamicStrategy> SyncLate
 2  typedef mxASyncStrategy<mxDynamicStrategy> AsyncLate
 3
 4  typedef mxSyncStrategy<mxStatic> SyncEarly
 5  typedef mxAsyncStrategy<mxStatic> AyncEarly
 6
 7  // ein typischer RPC, mit dem auch Web-Services aufgerufen werden koennen
 8  typedef mxTwoWay<mxOutFunctionCall, mxInFunctionCall, SyncEarly>  RPC;
 9
10  typedef mxOneWay<mxOutClassCall,                 SyncEarly>   Class_OneWay;
11  typedef mxTwoWay<mxOutClassCall, mxInClassCall,  AsyncEarly>  Class_TwoWay;
12
13  typedef mxOneWay<mxOutObjectCall,                 mxMessenger> Object_OneWay;
14  typedef mxTwoWay<mxOutObjectCall, mxInObjectCall, mxMessenger> Object_TwoWay;
15
16  template <class ObjectClientLayer>
17  class ExampleClient : public ObjectClientLayer {
18   public:
19    void run() {                      // startet die Applikation
20      SyncLate  sync;                 // eine synchrone spaet gebundene Strategie
21      AsyncLate async;                // eine asynchrone spaet gebundene Strategie
22
23      Class_TwoWay  om1;              // asynchroner TwoWay-Klassenaufruf
24      Object_OneWay om2;              // OneWay-Objektaufruf noch ohne Strategie
25      Object_TwoWay om3;              // TwoWay-Objektaufruf noch ohne Strategie
26      Class_OneWay  om4;              // synchroner OneWay-Klassenaufruf
27
28      mxConnection con;                       // eine Verbindung
29      con.setEndpoint("http://localhost");  // Server laeuft lokal
30      con.open();                             // Verbindung oeffnen
31
32      // entferntes Objekt mit TwoWay erzeugen
33      om1.setReceiver("Bar", "create");      // Empfaenger ist eine Klassenmethode
34      om1.addRecvParam<mxString>();          // Ergebnis ist ein String
35      om1.setConnection(&con);               // Verbindung zuweisen
36      om1.call();                            // Dienst aufrufen
37      char* object_id = om1.inParam<char*, mxString>();  // Objekt-Id zurueck
38
39      // Farbe auf entferntem Objekt mit OneWay setzen
40      om2.setStrategy(&sync);                // synchrone Strategie setzen
41      om2.setReceiver("Bar", object_id, "setColor");  // Empfaenger ist Objekt
42      om2.addSendParam<char*, mxString>("blue");      // Parameter hinzufuegen
43      om2.setConnection(&con);                        // Verbindung zuweisen
44      om2.call();
45
46      // Farbe von entferntem Objekt mit TwoWay holen
47      om3.setStrategy(&async);               // asynchrone Strategie verwenden
48      om3.setReceiver("Bar", object_id, "getColor");  // Empfaenger ist Objekt
49      om3.addRecvParam<mxString>();        // Ergebnis ist ein String
50      om3.setConnection(&con);
51      om3.call();                            // asynchroner Aufruf ...
52      do {} while(om3.gotResult() == false);// daher auf Ergebnis warten
53      char* color = om3.inParam<char*, mxString>();  // Farbe zurueck
54
55      // entferntes Objekt mit OneWay loeschen
56      om4.setReceiver("Bar", "delete");      // Empfaenger ist Klassenmethode
57      om4.addSendParam<char*, mxString>(object_id);// Parameter ist Objekt-Id
58      om4.setConnection(&con);
59      om4.call();
60
61      con.close();                           // Verbindung schliessen
62    }
63  }
```

Abbildung 5.22: Implementierung einer Client-Applikation

```
 1  typedef mxObjectClientLayer<mxClassClientLayer<mxFunctionClientLayer<mxTwoWayLayer<
 2          mxOneWayLayer<mxSyncronisationLayer<mxSeparationLayer<mxParamLayer<
 3          mxConnectionLayer<mxMessageLayer> > > > > > > > > > mxExampleMiddleware1;
 4
 5  typedef ... mxExampleMiddleware2;
 6
 7  int main() {
 8    ExampleClient<mxExampleMiddleware1> myClient1;
 9    myClient1.run();
10
11    ExampleClient<mxExampleMiddleware2> myClient2;
12    myClient2.run();
13  }
```

Abbildung 5.23: Konfiguration einer Client-Applikation mit einer Middleware

5.3 Die Server

Auch die Serverschichten konnten entsprechend dem Entwurf in Mixin-Schichten implementiert werden. Ausgewählte Teile der Implementierung werden davon nun vorgestellt.

Serververbindung und Serveraufrufe

Das Merkmal *Serververbindung*, um Verbindungsanfragen entgegennehmen zu können, wird entsprechend der Vorgabe (vgl. Abb. 4.14) in einer Mixin-Schicht gekapselt. Die Rolle *AcceptConnection* wird in dieser Schicht durch die Mixinklasse `mxServerConnection` implementiert und stellt u.a. die Schnittstelle `bind()` und `accept()` bereit. Wie in Abbildung 5.24 (1) dargestellt ist, bindet ein Server per `bind()` ein Serververbindungsobjekt (`mxServerConnection`) an einen Port, wo es auf Verbindungsanfragen wartet. Die Methode `accept()` liefert bei einer Verbindungsanfrage (Abb. 5.24 (2)) ein Verbindungsobjekt `mxConnection` zurück (Abb. 5.24 (3)), das für den Datenaustausch über diese Verbindung genutzt wird.

Da gSOAP nur verbindungsorientierte Kommunikation unterstützt, kann die Serververbindung nur eine *verbindungsorientierte Verbindung* zurückliefern. Für andere Protokollimplementierungen kann das Merkmal so umgesetzt werden, daß je nach Verbindungstyp unterschiedliche Verbindungsobjekte erzeugt werden.

Die Rolle *ServerCall*, deren Aufgabe darin besteht, Nachrichten mit einer Synchronisationsstategie zu empfangen, ist innerhalb der *Serveraufrufschicht* ähnlich einem `mxOneWay` implementiert (vgl. Abb. 5.20). Ein `mxServerCall` verbindet eine *InMessage* mit einer Synchronisationsstategie. Dazu wird dieser mit einer *InMessage* und einer Strategie parametrisiert, wobei beide Parameter ähnlich Abbildung 5.20 Basisklassen des `mxServerCalls` angeben. Die *ServerCall*-Rollen der Funktions-, Klassen- und Objektserverschicht können so durch Parametrisierung des `mxServerCalls` mit einer `mxInFunctionMessage`, `mxInClassMessage` bzw. einer `mxInObjectMessage` erzeugt wer-

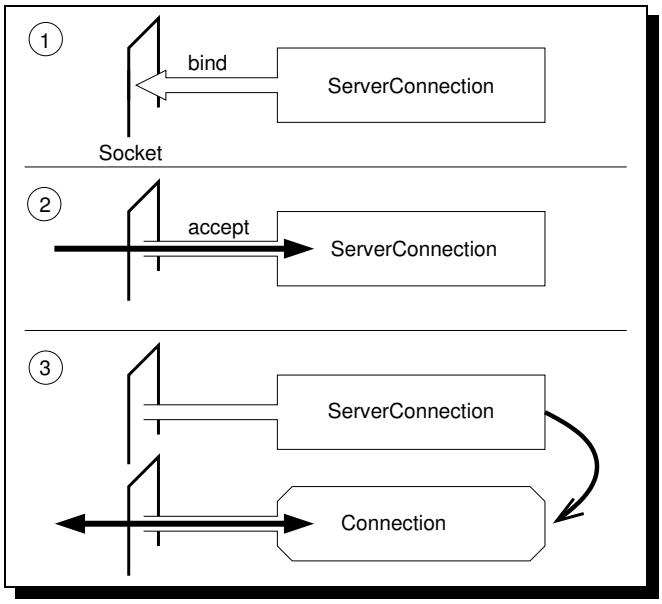

Abbildung 5.24: Serververbindung

den, wie dies bereits für den `mxOneWay` und `mxTwoWay` beschrieben wurde.

Serviceparadigma

Die im vorigen Kapitel beschriebenen Serverschichten *SRPC*, *SRCI* und *SROI* sind als Mixin-Schichten `mxFunctionServerLayer`, `mxClassServerLayer` und `mxObjectServerLayer` implementiert, siehe dazu Abbildung 5.25. In der Abbildung ist zu erkennen, daß die Rollen *OfferFunctionServices*, *OfferClassServices* und *OfferObjectServices* jeweils durch mehrere Mixin-Klassen implementiert werden. Zur besseren Wiederverwertung sind protokollspezifische und allgemeine Serverfunktionen entkoppelt, indem jede Entwurfsschicht durch jeweils zwei Implementierungsschichten umgesetzt wird: eine allgemeine Serverschicht und eine gSOAP-spezifische Schicht, die in der Abbildung durch "A" bzw. "G" gekennzeichnet sind. Die allgemeinen Schichten stellen Server bereit, die jeweils durch die gSOAP-Schichten spezialisiert werden. Die gSOAP-Server implementieren die Methoden `run()` und `serve()`. `run()` startet den Server, `serve()` nimmt eine Nachricht entgegen und identifiziert daraus den Empfänger (Funktion, Klasse, Objekt). Die Serverrollen der allgemeinen Schichten sind dagegen für den Bereich *Dienstverwaltung* zuständig, wobei Dienste bei einem Funktionsserver Funktionen, bei einem Klassenserver statische Klassenmethoden und bei einem Objektserver Objektmethoden sind.

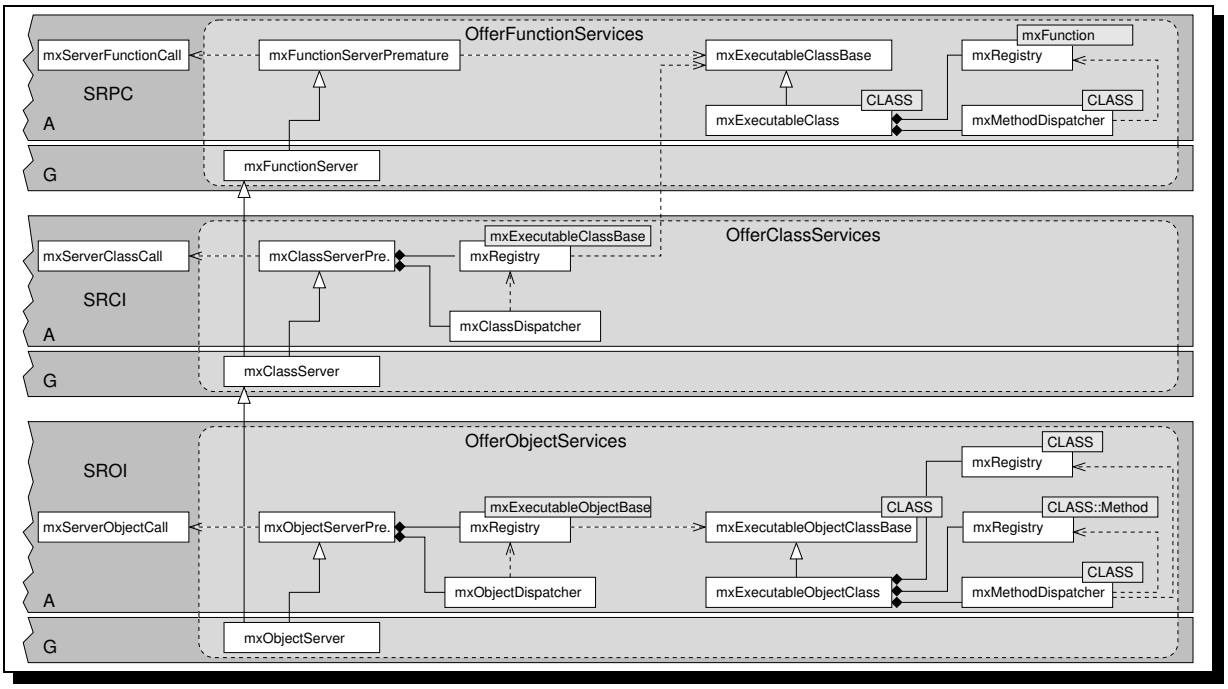

Abbildung 5.25: Server-Mixin-Schichten als Klassendiagramm

Der Funktionsserver

Ein Funktionsserver hat die Aufgabe, RPC's bereitzustellen. In der Abbildung 5.25 ist zu erkennen, daß bereits der Funktionsserver Klassen besitzt, die schon von der Namensgebung her eher dem Klassenserver zuzuordnen sind (z. B. `mxExecutableClass`, `mxMethodDispatcher`). Diese Klassen werden bereits im Funktionsserver benötigt, da Dienstfunktionen nicht als herkömmliche C-Funktionen bereitgestellt werden, sondern in Form von statischen Klassenfunktionen simuliert werden. Der Grund dafür ist, daß ein Server über jede Dienstfunktion Zusatzinformationen wie den Namen, unter dem sie aufgerufen wird, benötigt. Diese Informationen können über eine Klasse komfortabler bereitgestellt werden. Bei einem Funktionsserver wird genau eine standardisierte Klasse angemeldet, die alle Dienstfunktionen enthält. Diese Klasse wird als Parameter für eine `mxExecutableClass` verwendet, wie im Code-Beispiel in Abbildung 5.26, Zeile 2 dargestellt ist. Eine Klasse `mxExecutableClass` besitzt ein Registrierungobjekt `mxRegistry`, das die Adressen der angemeldeten statischen Methoden verwaltet (siehe Abb. 5.25). (`mxRegistry` ist eine Templateklasse, die mit sehr unterschiedlichen Entitäten parametrisiert werden kann und daher zur Registrierung von Klassen, Objekten, Methoden und Objektinstanzen verwendet wird.) Eine `mxExecutableClass` ist in der Lage, die ihr bekannten statischen Methoden über eine Klassenmethode `exec()` unter Angabe des Methodennamens aufzurufen. Dazu besitzt jede `mxExecutableClass` einen eigenen Methoden-Dispatcher vom Typ `mxMethodDispatcher` (vgl. Abb. 5.25). Dieser

sucht aus der Methodenregistrierung der Klasse die passende Methode heraus und ruft diese inclusive Parameter auf[16].

Um die Funktionsklasse bei einem Funktionsserver (bzw. Klassen- oder Objektserver) anzumelden, wird die Serverfunktion `setFunctions()` verwendet (Abb. 5.26, Zeile 3). Ruft ein Client eine Serverfunktion auf, so wird die Anfrage vom Server an die angemeldete Funktionsklasse weitergeleitet, die wie beschrieben die entsprechende Dienstfunktion aufruft.

Der gSOAP-Funktionsserver kann so konfiguriert werden, daß die Dienstfunktionen *"WebService"*-kompatibel bereitgestellt werden. Damit können die Serverfunktionen auch durch beliebige Clients aufgerufen werden, die den *WSDL*-Standard unterstützen.

```
1  FunctionServer fs;              // ein Funktionsserver
2  mxExecutableClass<F> f;         // Funktionsklasse F kapseln
3  fs.setFunctions(f);             // und beim Server anmelden
4  fs.run(1977);                   // Funktionsserver auf Port 1977 starten
```

Abbildung 5.26: Funktionen beim Server anmelden

Der Klassenserver

Der Klassenserver `mxClassServerPremature` erweitert den Funktionsserver dahingehend, daß mehrere Klassen mit statischen Methoden als Dienste angeboten werden können. Diese Klassen werden ebenfalls in Form einer `mxExecutableClass` per `signStaticClass()` beim Server registriert (Abb. 5.27, Zeilen 2-5) und besitzen jeweils einen eigenen Namen, unter dem sie aufgerufen werden. Zur Verwaltung der Klassen besitzt der Server ein Registrierungsobjekt, das die Klassen aufnimmt (Abb. 5.25). Trifft bei einem Klassenserver eine Nachricht zum Aufruf einer statischen Klassenmethode ein, findet der Server über sein Registrierungsobjekt die entsprechende Klasse heraus und übergibt die empfangene Nachricht als `mxInMessage` an diese. Dazu wird ein Klassendispatcher `mxClassDispatcher` eingesetzt (vgl. Abb. 5.25). Die Klasse identifiziert die statische Methode und ruft diese auf, wobei die Nachricht als einziger Parameter übergeben wird. Nur der Dienst selbst "weiß", welche Parameter er erwartet und kann diese aus der Nachricht extrahieren. Die bereits erwähnte Servermethode `serve()` stattet diese Nachricht vorher mit einer Verbindung (`mxConnection`) aus, die von dem Dienst zur Beantwortung der Nachricht genutzt werden kann (vgl. Abb. 5.24).

[16] Als Parameter wird immer eine `mxInMessage` übergeben, aus der die Dienstfunktion bzw. -methode ihre Parameter deserialisieren kann.

```
1  ClassServer cs;                  // ein Klassenserver
2  mxExecutableClass<A> ea;         // die Klassen A ...
3  mxExecutableClass<B> eb;         // und B incl. ihrer statischen Methoden
4  cs.signStaticClass(ea);          // werden  beim Server angemeldet
5  cs.signStaticClass(eb);
6  cs.run(1977);                    // Klassenserver auf Port 1977 starten
```

Abbildung 5.27: Klassen beim Server anmelden

Der Objektserver

Der Objektserver erweitert den Klassenserver und kann Dienste in Form von Objekten bzw. Objektmethoden bereitstellen. Ebenso wie ein Klassenserver in der Lage ist, seine Klassen aufzurufen, und diese wiederum ihre statischen Methoden aufrufen können, so kann ein Objektserver `mxObjectServerPremature` Objekte verwalten und aufrufen. Jede Klasse, die von einem Objektserver instantiiert werden soll, muß als Parameter einer `mxExecutableObjectClass` (Abb. 5.28, Zeilen 2-5) beim Server per `signClass()` angemeldet werden. Eine von einem Objektserver instantiierbare Klasse besitzt zusätzliche Informationen über ihren Methoden. Wie in Abbildung 5.25 dargestellt ist, besitzt ein Objektserver ein weiteres Registrierungsobjekt, daß alle *instantiierbaren* Klassen enthält. Für jede angemeldete Klassen legt der Server eigene Strukturen an, damit eine Klasse ihre Objektinstanzen selbst verwalten kann. Diese Strukturen bestehen im Wesentlichen aus einem Methodendispatcher für Objektmethoden sowie einem Registrierungsobjekt für die Objektmethoden und einem für die Objektinstanzen. Wird ein Objekt angelegt oder gelöscht, so vermerkt sich dies die Klasse des Objektes in ihrer Instanzen-Registrierung. Erhält ein Objektserver einen Dienstaufruf, der ein Objekt adressiert, wird die Nachricht zunächst an die Klasse weitergeleitet, zu der das Objekt gehört. Die Klasse sucht aus ihrer Instanzen-Registrierung über eine statische Methode das richtige Objekt und aus ihrer Methoden-Registrierung die ensprechende Methode heraus. Anschließend ruft die Klasse die Methode auf "ihrem" Objekt auf. Eine angemeldete Objektserver-Klasse besitzt zusätzliche statische Methoden `createObject()` und `destroyObject()` zum Erzeugen und Löschen von Instanzen. Diese können "remote" wie statische Klassenfunktionen aufgerufen werden.

```
1  ObjectServer os;                 // ein ObjectServer
2  mxExecutableObjectClass<A> oa;   // die Klassen A ...
3  mxExecutableObjectClass<B> ob;   // und B werden incl. ihrer  Objektmethoden
4  os.signClass(oa);                // beim Server angemeldet
5  os.signClass(ob);
6  os.run(1977);                    // Objektserver auf Port 1977 starten
```

Abbildung 5.28: Klassenobjekte beim Server anmelden

Zusammenfassung der Implementierung

Die Implementierung von *allgemeinen Merkmalen*, *Client-Merkmalen* und *Server-Merkmalen* als Mixin-Schichten einer gemeinsamen Middleware-Architektur wurde entsprechend dem Architekturentwurf umgesetzt. Funktionsfähige, unterschiedlich konfigurierte Client- und Server-Applikationen konnten erstellt werden, um die Arbeit zu verifizieren. Das Ergebnis der Implementierung ist der funktionale Kern einer erweiterbaren, flexiblen Middlewarefamilie. Um dieses Ziel zu erreichen, wurde sehr intensiv von der Templateprogrammierung Gebrauch gemacht, vor allem auch, um die Beziehungen zwischen Mixin-Schichten herzustellen, wie in [SB98] und [SB02] beschrieben. Aspektorientierte Programmierung konnte aufgrund der beschriebenen Konflikte mit der Template-Programmierung (noch) nicht eingesetzt werden. Auch ohne AOP konnten bisher mehr oder weniger modulübergreifende Merkmale wie die *Protokollimplementierung* implementiert werden, ohne gravierende Abstriche bei Konfigurierbarkeit oder Wiederverwertbarkeit der Komponenten hinnehmen zu müssen. Später einmal soll das Merkmal *Protokollimplementierung* als Aspekt formuliert und beim Konfigurationsvorgang in die Middleware eingewoben werden.

Am Ende des Kapitels wird nun gezeigt, daß die Konfigurationsvorgaben, so wie sie in der Domänenanalyse für die Middleware-Merkmale definiert wurden, durch die Implementierung unterstützt werden.

5.4 Fallstudien/Konfiguration

Die Konfigurationsmöglichkeiten der Middlewarefamilie werden nun anhand von drei Fallbeispielen dargestellt: ein mobiler WebService-Client, ein Sensor-Actor-System mit einem mobilen Sensor sowie ein umfangreich ausgestatteter Objektserver. Für jedes System werden zuerst die Merkmale definiert. Anschließend wird je ein neues Mitglied der Middlewarefamilie mit diesen Merkmalen erstellt. Die Implementierung von Applikationen für die Fallbeispiele wird am Beispiel gezeigt.

Um die Fallstudien anschaulich zu halten, wird das Merkmal *Protokollimplementierung* in den Beispielen vernachlässigt. Es wird davon ausgegangen, daß dieses Merkmal in den Kollaborationen bereits integriert ist. Die Konfiguration dieses Merkmals wird am Ende des Abschnitts behandelt. Zum weiteren Verständnis sind in Tabelle 5.2 die Schichten der Middleware noch einmal mit ihren Abkürzungen zusammengefaßt.

Die Konfiguration eines Familienmitgliedes findet in zwei Schritten statt:

1. Erstellen von Mixin-Schichten

2. Verknüpfen von Mixin-Schichten

SROI	Server-ROI-Schicht	Server
ROI	Client-ROI-Schicht	Client
SRCI	Server-RCI-Schicht	Server
RCI	Client-RCI-Schicht	Client
SRPC	Server-RPC-Schicht	Server
RPC	Client-RPC-Schicht	Client
TWO	TwoWay-Schicht	Client
ONE	OneWay-Schicht	Client
SC	Serveraufrufschicht	Server
SV	Serververbindungsschicht	Server
S	Synchronisationsschicht	Allgemein
R	Richtungstrennungsschicht	Allgemein
P	Parameterschicht	Allgemein
V	Verbindungsschicht	Allgemein
A	Abstrakte Nachrichtenschicht	Allgemein

Tabelle 5.2: Übersicht der Middleware-Schichten

Erstellen von Kollaborationen

Bevor Mixin-Schichten über parametrisierte Vererbung miteinander verbunden werden können, müssen die Schichten erstellt werden. Die meisten Kollaborationen (A, R, SC, ONE, TWO, RPC, RCI und ROI) brauchen jedoch nicht verändert werden. Die Merkmale, die durch diese Kollaborationen implementiert werden, werden durch Hinzufügen oder Weglassen dieser Schichten konfiguriert. Die Kollaborationen V und SV können mit verschiedenen Verbindungsarten ausgestattet werden. Da gSOAP nur eine Verbindungsart unterstützt, gibt es von diesen Schichten vorerst jeweils nur eine Variante, die nicht angepaßt werden braucht. SRPC, SRCI und SROI brauchen bei der Erstellung eines Familienmitgliedes auch nicht verändert werden (Sie können zwar mit einem Protokoll konfiguriert werden, doch diese Möglichkeit soll hier vernachlässigt werden). Diese festen Konfigurationen werden im Folgenden *Standardkollaborationen* genannt.

Um die Datentypen eines Familienmitgliedes zu konfigurieren, werden in der *Parameterschicht* P unterschiedlich viele Datentypen implementiert. Der Applikationsentwickler fügt die Datentypen-Implementierungen mit Werkzeugunterstützung oder per Hand[17] in eine "leere" Parameterschicht ein und erhält eine maßgeschneiderte Kollaboration. Um diese Konfigurierung nicht für jede Applikation erneut vornehmen zu müssen, können auf diese Weise häufig verwendete Parameterschicht-Konfigurationen bereits im Voraus erstellt und als vorgefertigte Mixin-Schichten so wie die Standardkollaborationen archiviert werden. Grundsätzlich können alle Varianten einer Kollaboration auch schon im Voraus erstellt und archiviert werden. Bei manchen ist dies jedoch nicht sinnvoll, da wie

[17] unter Verwendung von `#include` und `typedef`-Anweisungen

bei der Parameterschicht, sehr viele Varianten existieren können.

Ebenso wird mit der Konfiguration der Synchronisationsschicht verfahren, um unterschiedliche Synchronisationsstrategien bzw. einen zusätzlichen Vermittler für dynamische Rekonfiguration einer Strategie in Kollaborationen zu kapseln. Da hier nach Tabelle 5.1 bei zwei Synchronisationsstrategien nur vier Kombinationen möglich sind, können diese vier Kombinationen vorkonfiguriert und archiviert werden und als Standardkollaborationen bereits vorliegen.

Verknüpfen von Kollaborationen

Werden die nun vorliegenden Kollaborationen bzw. Mixin-Schichten miteinander verknüpft, entsteht ein konkretes Mitglied der Middlewarefamilie. Die Verknüpfung der Schichten findet innerhalb der Applikation statt. Für den Applikationsentwickler bedeutet dies nur einen geringeren Mehraufwand, da das Einbinden von Schnittstellen und die Definition eigener Typen Bestandteile jeder Implementierung sind.

Fallbeispiel 1: WebService-Client

Mit einer mobilen Applikation soll der Zugriff auf das Preisinformationssystem eines Online-Warenhauses ermöglicht werden, das über eine WebService-Funktion per SOAP/HTTP bereitgestellt wird. Ein Produkt des Warenhauses wird über eine ID (String) und eine Größe (Integer) identifiziert, der Preis wird als *Float* bereitgestellt. Der WebService-Client besitzt daher die Datentypen *Integer*, *Float* und *String*. Da die Kommunikation über SOAP/HTTP stattfindet, wird das Merkmal *verbindungsorientierte Kommunikation* verwendet. Es sollen *synchrone* und *asynchrone* Kommunikationsstrategien einsetzbar sein, die dynamisch austauschbar sind. Um *WebServices* mit Ergebnisrückgabe aufzurufen, wird ein *TwoWay*-Funktionsaufruf benötigt.

In Abbildung 5.29 ist dargestellt, wie eine dafür maßgeschneiderte Middleware aufgebaut ist. Die Beispielapplikation selbst ist in der Abbildung nicht eingezeichnet, sie befindet sich als Schicht oberhalb der RPC-Schicht. Um dieses Familienmitglied zu konfigurieren, werden acht Schichten benötigt (Abb. 5.29). Die Schichten (A, V, R, S sowie ONE, TWO und RPC) sind Standardkollaborationen. Eine maßgeschneiderte Parameterschicht muß mit den Datentypen *Integer*, *Float* und *String* konfiguriert werden. Um eine Middleware für einen mobilen WebService-Client zu erstellen, muß somit nur eine Schicht konfiguriert werden. Die restlichen Konfigurierungsaufgaben werden innerhalb der Client-Applikation vorgenommen und werden anhand des Code-Beispiels 5.30 beschrieben.

In Abbildung 5.30, Zeilen 1-2 werden die Schnittstellen der benötigten Mixin-Schichten eingefügt. In den Zeilen 5-7 wird das Familienmitglied erstellt, indem die benötigten Schichten (die fertig vorliegen), zu einer Hierarchie verschmolzen werden. In den Zeilen 19-21 wird eine weitere Konfiguration vorgenommen, um einen dynamischen *TwoWay*-Funktionsaufruf zu generieren. In Zeile 40 wird die Beispielapplikation mit der Middleware instantiiert und danach ausgeführt.

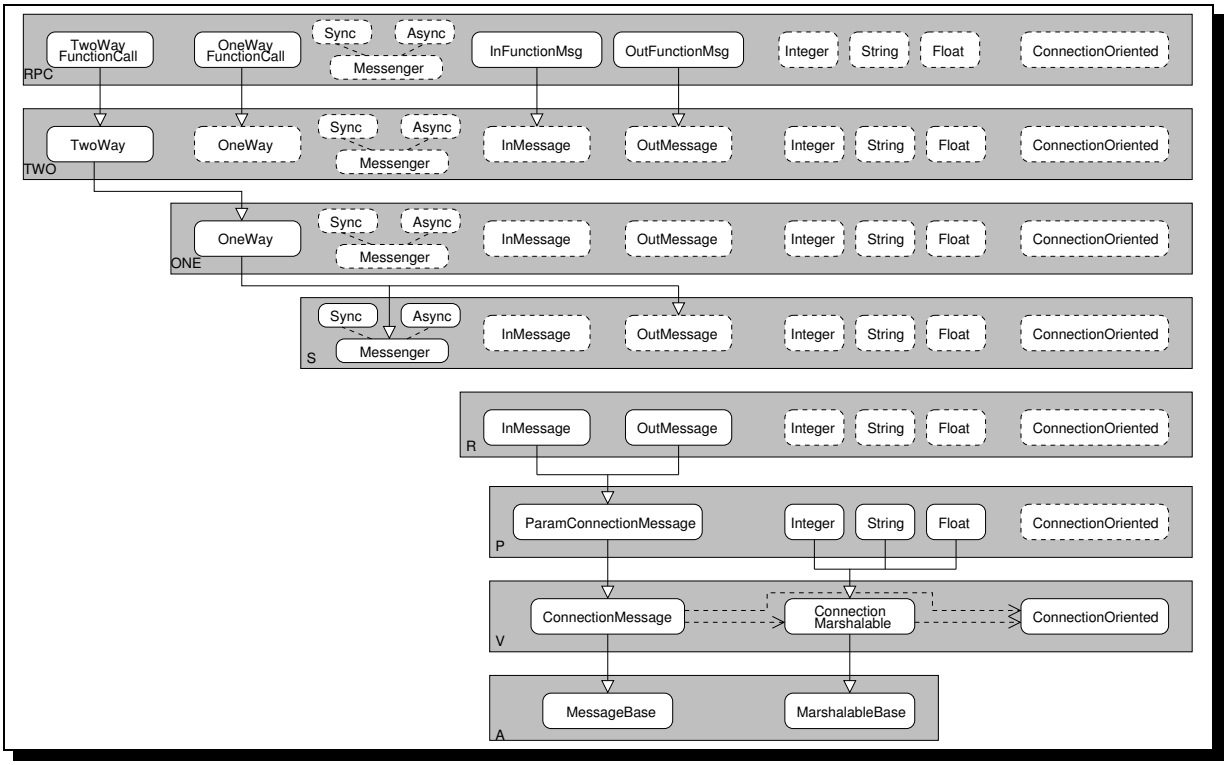

Abbildung 5.29: Konfiguration eines WebService-Clients

```
1  #include "mxMessageLayer.h"
2  #include ...     // Mixin-Schichten einbinden
3
4  // Schichten werden miteinander zu einer Middleware verknuepft
5  typedef mxFunctionClientLayer<mxTwoWayLayer<mxOneWayLayer
6          <mxSyncronisationLayer<mxSeparationLayer<mxParamLayer
7          <mxConnectionLayer<mxMessageLayer> > > > > > > WebServiceMW;
8
9  // entspricht RPC<TWOWAY<ONEWAY<S<R<P<V<A>>>>>>>
10
11 // Applikationsschicht wird mit einer Middleware parametrisiert
12 template <class Middleware>
13 class WebServiceApp : public Middleware {
14  public:
15   void quoteRequest(char* id, int size) {          // Preisanfrage
16    mxConnection con;                               // Verbindung
17    mxSyncStrategy<mxDynamicStrategy>    sync;      // synchrone Strategie
18
19    mxTwoWay<mxOutFunctionMessage, mxInFunctionMessage,
20             mxMessenger> myReq;
21    myReq.addSendParam<char*, mxString>(id);        // Ausgangsparameter..
22    myReq.addSendParam<int,   mxInt>(size);         // ..an die Nachricht haengen
23    myReq.addRecvParam<mxFloat>();                  // Eingangsparameter
24
25    myReq.setReceiver("getPrice");                  // Dienstname setzen
26    myReq.setConnection(&con);                      // Verbindung setzen
27    myReq.setStrategy(&sync);                       // synchrone Strategie
28
29    con.setEndpoint("http://xxl-superstore.com");   // Preisauskunftsserver
30    con.open();                                     // Verbindung oeffnen
31    myReq.call();                                   // Dienstaufruf ausfuehren
32    con.close();                                    // Verbindung schliessen
33
34    float price = myReq.inParam<float, mxFloat>(); // Ergebnis holen
35    printf("Price:␣%.2f\n", price);                 // und ausgeben
36   };
37 };
38
39 int main() {
40   WebServiceApp<WebServiceMW> myApp;   // Applikation instantiieren
41   myApp.quoteRequest("bike273", 50);   // und Dienst aufrufen
42   return 0; }
```

Abbildung 5.30: Implementierung eines WebService-Clients

Fallbeispiel 2: Sensor-Actor-System

Das Sensor-Actor-System besteht aus einem mobilen Client (Sensor) und einem mobilen oder stationären Server (Actor). Der Client besitzt einen Sensor, um dem Server z.B. per GPRS regelmäßig Meßwerte zu senden, der diese in einer Datenbank ablegt. Ein Meßwert liegt dem Client vor der Übertragung als *Float* vor. Die Kommunikation ist für den Client verbindungslos und asynchron. Die Daten werden über einen *OneWay*-Funktionsaufruf an den Server übermittelt. Der Server muß eine Funktion bereitstellen, die vom Client aufgerufen werden kann, um den Meßwert als Parameter zu übergeben. Der Server schickt keine Antwortnachrichten an den Client zurück, empfängt die Nachrichten aber synchron.

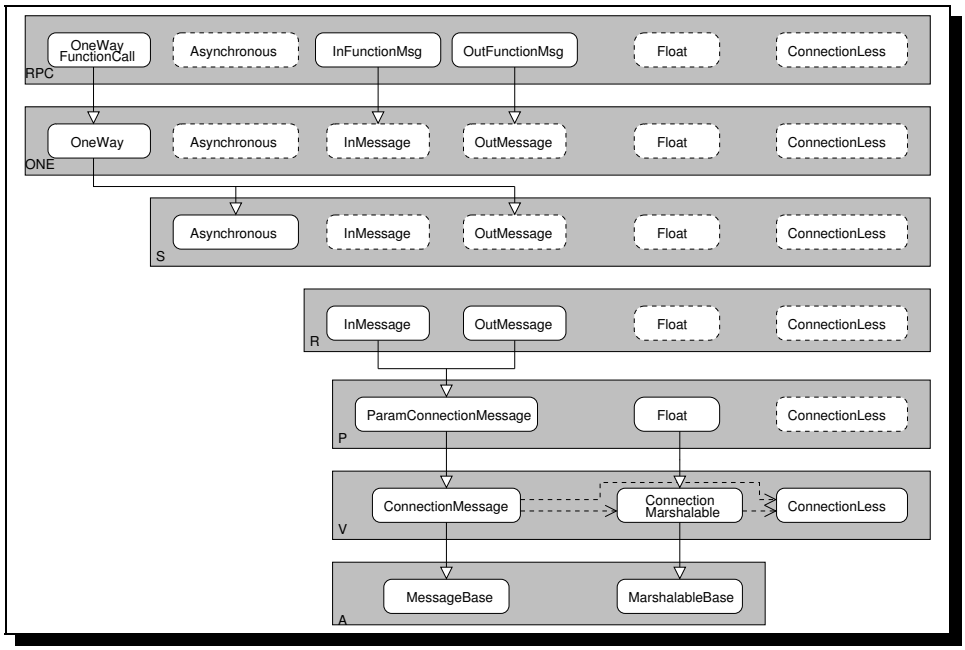

Abbildung 5.31: Konfiguration eines Sensor-Clients

Die Sensor-Middleware kommt mit sieben Schichten aus (Abb. 5.31). Die Schichten A, V, R, S, ONE und RPC liegen vorkonfiguriert vor, da es sich um Standardkollaborationen handelt. Die Parameterschicht P muß per Hand oder mit einem Werkzeug - wie im ersten Fallbeispiel beschrieben - erstellt werden, um den Datentyp *Float* zu integrieren. Die restlichen Konfigurierungsaufgaben werden innerhalb der Applikation vorgenommen.

Die Middleware des Actor-Servers besteht aus acht Schichten (Abb. 5.32). Die Schichten A, V, R, S, SV, SC und SRPC sind vorkonfigurierte Standardschichten. Als Parameterschicht wird die bereits für den Client mit einem *Float*-Parameter erstellte Parameterschicht eingesetzt.

Um diese beiden Mitglieder der Middlewarefamilie zu erstellen, muß somit ebenfalls nur eine Schicht, die Parameterschicht, vorkonfiguriert werden. Die restlichen Konfigurationsaufgaben werden innerhalb der Client- bzw. der Serverapplikation vorgenommen. Das

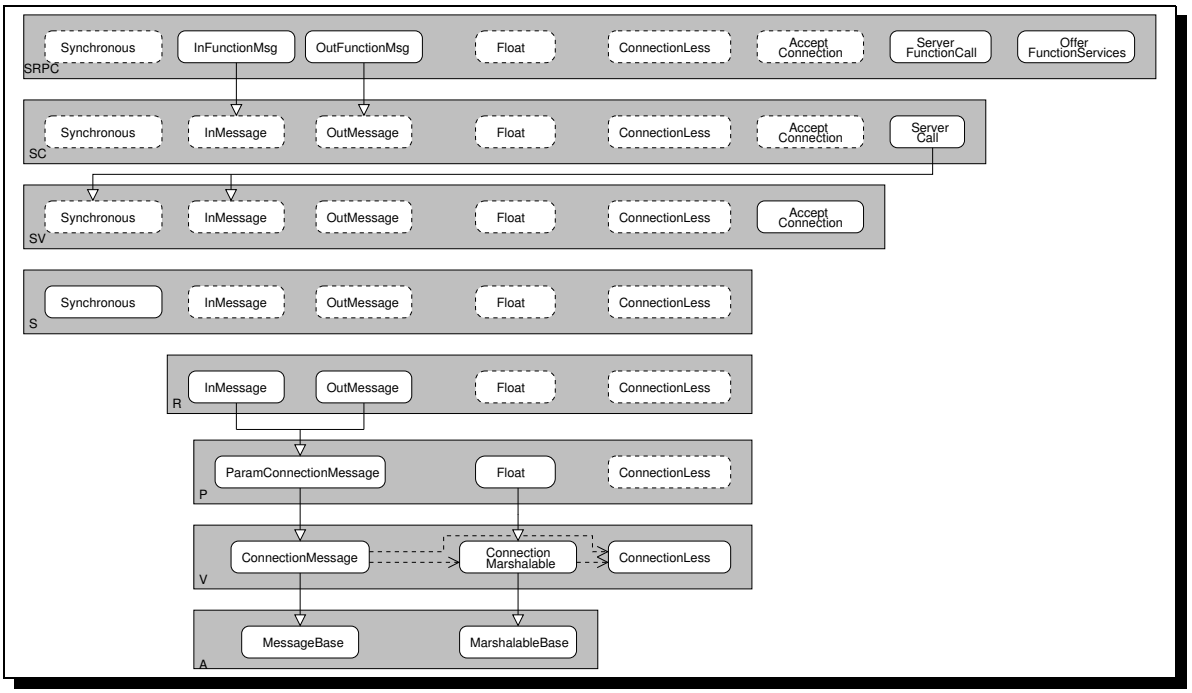

Abbildung 5.32: Konfiguration eines Actor-Servers

Prinzip wurde für den Client schon im ersten Fallbeispiel beschrieben. Die Konfigurierung innerhalb einer Serverapplikation ist ähnlich und in Abbildung 5.33 dargestellt.

```
1  #include ...      // Mixin-Schichten einbinden
2
3  // Schichten werden miteinander zu einer Server-Middleware verknuepft
4  typedef mxFunctionServerLayer<mxServerCallLayer<mxServerConnectionLayer
5          <mxSyncronisationLayer<mxSeparationLayer<mxParamLayer
6          <mxConnectionLayer<mxMessageLayer> > > > > > > ActorServerMW;
7
8  // entspricht SRPC<SC<SV<S<R<P<V<A>>>>>>>
```

Abbildung 5.33: Konfiguration innerhalb des Actor-Servers

Zuerst werden die notwendigen Schichten eingebunden (Abb. 5.33, Zeile 1). Anschließend werden diese Schichten über parametrisierte Vererbung zu einer Server-Middleware verknüpft. Die restlichen Konfigurationsaufgaben für den Server finden innerhalb der Server-Applikation statt.

Fallbeispiel 3: Stationärer Server

Im letzten Fallbeispiel wird ein fiktiver mobiler oder stationärer Server mit allen nur möglichen Merkmalen ausgestattet. Es handelt sich somit um einen Objektserver, der verschiedene Synchronisationsstrategien, unterschiedliche Verbindungsarten und Datentypen unterstützt[18]. Er bietet Dienste in Form aller Serviceparadigmen an. In Abbildung 5.34 sind die Schichten V, P, S und SROI dieses Familienmitgliedes abgebildet. Insgesamt besteht der Server aus 15 Schichten (vgl. S. 68).

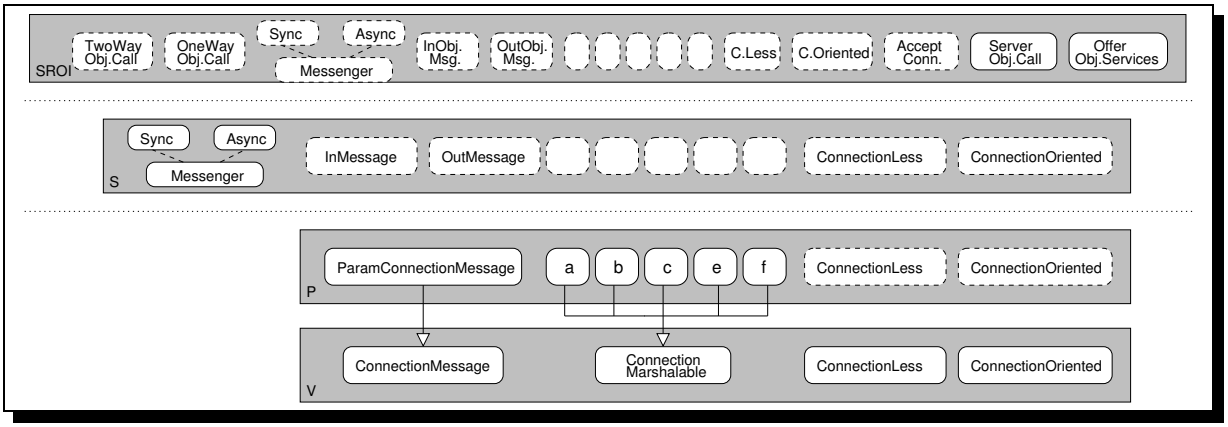

Abbildung 5.34: Konfiguration eines stationären Servers

Zur Konfiguration dieses Server muß ebenfalls nur die Parameterschicht erstellt werden. Alle anderen Schichten liegen als Standardkollaborationen vor, da die Variantenanzahl dieser Kollaborationen klein ist.

Es wird deutlich, daß für die bislang modellierten gSOAP-Schichten allenfalls die Parameterschicht beim Konfigurationsvorgang neu erstellt werden muß, um die Datentypen-Unterstützung zu konfigurieren. Der Konfigurationsaufwand ist damit sehr klein, da das Zusammenführen der Komponenten innerhalb der Applikation stattfindet und vom Applikationsentwickler vorgenommen wird.

GenVoca-Grammatik der Middlewarefamilie

Um einem Applikationsentwickler (oder einem Werkzeug) die Verifikation gültiger Familienmitglieder zu erleichtern, können die gültigen Verknüpfungen von Middleware-Schichten mit Hilfe einer regulären Grammatik (GenVoca-Grammatik) ausgedrückt werden. Die Grammatik für die Middleware ist in Abbildung 5.35 dargestellt. Da gSOAP nur einen Verbindungstyp und nur eine Synchronisationsstrategie unterstützt, wird für die Grammatik zur besseren Veranschaulichung eine leicht modifizierte Middleware zugrunde gelegt, die zwei Synchronisationsstrategien und zwei Verbindungstypen unterstützt.

[18] die nicht alle von gSOAP unterstützt werden

```
A     := a                                          // Abstrakte Nachrichtenschicht
V     := lA | oA | loA                              // Verbindungsschicht
P     := fV | sV | iV | fsV | fiV | siV | fsiV      // Parameterschicht
R     := rP                                         // Richtungstrennungsschicht
S     := aR | sR | asR | asmR                       // Synchronisationsschicht
SV    := svS                                        // Serververbindungsschicht
SC    := scSV                                       // Serveraufrufschicht
ONE   := oneS | oneSC                               // OneWay-Schicht
TWO   := twoONE                                     // TwoWay-Schicht
RPC   := rpcONE | rpcTWO                            // Client-RPC-Schicht
SRPC  := srpcSC   | srpcRPC                         // Server-RPC-Schicht
RCI   := rciRPC                                     // Client-RCI-Schicht
SRCI  := srciSRPC | srciRCI                         // Server-RCI-Schicht
ROI   := roiRCI                                     // Client-ROI-Schicht
SROI  := sroiSRCI | sroiROI                         // Server-ROI-Schicht
```

Abbildung 5.35: GenVoca-Grammtik der Middlewarefamilie

Von der *Abstrakten Nachrichtenschicht* `A` ist nur eine Variante `a` implementiert. Die Verbindungsschicht `V` liegt in den Varianten *verbindungslos* (`l`), *verbindungsorientiert* (`o`) und in einer Variante mit beiden Merkmalen vor (`lo`). Sie benutzt immer die Schicht `A`. Für das Beispiel wird angenommen, daß drei verschiedene Datentypen existieren (`f`, `s` und `i`), die miteinander beliebig kombiniert werden können. Die *Parameterschicht* `P` besitzt somit sechs Varianten. Die *Richtungstrennungsschicht* `R` ist nur in einer Variante `r` implementiert. Für die *Synchronisationsschicht* `S` können vier Varianten abgeleitet werden, da diese auch einen Messenger besitzt: asynchron (`a`), synchron (`s`), beide Strategien ohne (`as`) und mit Messenger (`asm`). Die Serververbindungsschicht `SV` liegt nur in einer Variante `sv` vor, ebenso die Serveraufrufschicht `SC`. Von der *OneWay-Schicht* `ONE` gibt es nur eine Variante. Diese kann für einen Client entweder mit der *Synchronisationsschicht* oder für einen Server mit der *Serververbindungsschicht* parametrisiert werden. Die *TwoWay-Schicht* `TWO` hat nur eine Variante. Die *Client-RPC-Schicht* `RPC` wird entweder mit der *OneWay-* oder mit der *TwoWay-Schicht* parametrisiert. Die Client-Schicht `RCI` kann nur von `RPC` und die Schicht `ROI` nur von `RCI` abgeleitet werden. Die *Server-RPC-Schicht* `SRPC` wird für einen Server ohne Antwortfunktionalität mit der *Serveraufrufschicht* und für einen vollwertigen Funktionsserver mit der *Client-RPC-Schicht* parametrisiert. Dieses Familienmitglied besitzt somit Merkmale des Servers und des Clients. Ebenso können die Schichten `SRCI` und `SROI` entweder mit oder ohne Client-Merkmalen ausgestattet werden.

Jeder Satz, der durch diese Grammatik produziert werden kann, ist ein gültiges Mitglied der Middleware-Familie. Jedoch sind nur die Familienmitglieder ab der RPC-Schicht funktionsfähig. Die Anzahl der Familienmitglieder der realen gSOAP-Middleware kann mit Hilfe der folgenden Gleichungen berechnet werden.

Konfiguration der Protokollimplementierung

Im Kapitel 5.1 wurde bereits beschrieben, daß einige Kollaborationen durch zwei Mixin-Schichten implementiert sind: eine allgemeine Schicht und eine abgeleitete, protokollspezifische Schicht (z.B. Schicht V).
Um die Protokollimplementierung einer Middleware zu konfigurieren, wird der erste Konfigurationsschritt *Erstellen von maßgeschneiderten Mixin-Schichten* bei

- *protokollunabhängigen* Kollaborationen auf deren (einzige) Mixin-Schicht angewendet.
- *protokollabhängigen* Kollaborationen entweder auf die unspezifische, die protokollspezifische oder auf beide Mixin-Schichten angewendet, je nachdem wo das gewünschte Merkmal implementiert ist.

Die eigentliche Auswahl der Protokollimplementierung wird dann im zweiten Konfigurationsschritt *Verknüpfen von Mixin-Schichten* durch den Applikationsentwickler innerhalb der Applikation durchgeführt. Ist eine protokollspezifische Mixin-Schicht für eine Kollaboration vorhanden, wird diese bei der Instantiierung einer Middleware verwendet. In Abbildung 5.36 ist in zwei Code-Beispielen dargestellt, wie die Auswahl der Protokollimplementierung durch `#include`-Anweisungen gesteuert werden kann (jeweils Zeile 3).

```
1  // ein Familienmitglied mit gSOAP
2  #include "general/mxMessageLayer.h"
3  #include "protocols/gsoap/mxConnectionLayer.h"
4
5  typedef mxConnectionLayer<mxMessageLayer> myFamilyMember;
```

```
1  // "dasselbe" Familienmitglied mit dem Protokoll "Foo"
2  #include "general/mxMessageLayer.h"
3  #include "protocols/foo/mxConnectionLayer.h"
4
5  typedef mxConnectionLayer<mxMessageLayer> myFamilyMember;
```

Abbildung 5.36: Konfiguration der Protokollimplementierung

Die `#include`-Anweisungen können bei der Imlementierung einer Applikation durch den Entwickler von Hand oder durch ein Werkzeug eingefügt werden.

Kapitel 6

Zusammenfassung, Bewertung und Ausblick

6.1 Zusammenfassung und Bewertung

Zielstellung dieser Arbeit waren Analyse, Entwurf und Implementierung von Teilaspekten einer Middlewarefamilie für mobile Informationssysteme. Um die Anforderungen nach Wiederverwendbarkeit, Flexibilität, Erweiterbarkeit und Konfigurierbarkeit zu erfüllen, wurden verschiedene Konzepte aus dem Bereich der Softwaretechnik eingesetzt.

Im Verlauf der Arbeit entstand die Grundlage einer Middlewarefamilie, die diesen Anforderungen nachvollziehbar gerecht wird. Wichtige Kommunikationsmerkmale der Zielsysteme sind in austauschbaren Kollaborationen gekapselt, so daß die Erzeugung maßgeschneiderte Systeme einfach ist. Die Konfigurierung der Familienmitglieder ist mit und ohne Werkzeugunterstützung möglich. Weitere Merkmale können leicht in zusätzlichen Kollaborationen gekapselt und in die Kollaborationenhierarchie eingefügt werden. Für Applikationen oder funktionale Erweiterungen steht eine Schnittstelle bereit, die Funktionsdienste, Klassenfunktionsdienste und Objektdienste bereitstellen kann.

Mit der Arbeit ist am Beispiel gezeigt worden, daß die Softwarekonzepte, insbesondere der *Kollaborationenentwurf* und die *Mixin-Layer-Implementierung*, geeignet sind, den Entwurf und die Implementierung einer Schichtenarchitektur unter den o.g. Prämissen zu unterstützen.

Die Ergebnisse der Arbeit sollen nun mit aktuellen softwaretechnischen Fragestellungen in Verbindung gebracht werden, um dem Leser eine qualitative Bewertung der Arbeit zu ermöglichen.

Modularisierung von Merkmalen

Smaragdakis und Batory stellen in [SB02] fest, daß die Modularisierung von Merkmalen Probleme bei der Konfigurierung maßgeschneiderter Software aufwirft, die mit grob- oder feingranularer Modulbildung nicht gelöst werden können. Je größer die Module sind,

desto schwerer lassen sich überflüssige Merkmale aus einem Systemen heraushalten. Verkleinerte Module erschweren den Konfigurationsvorgang. Im Bereich eingebetteter und kleiner Betriebssysteme wie z.B. PURE [BGP+99] werden daher komplexe Konfigurationswerkzeuge wie CONSUL verwendet, um den Konfigurationsprozeß automatisiert zu unterstützen [BSSP03]. Mit CONSUL werden Merkmale modularisiert, indem beim Konfigurationsprozeß eine Merkmalsauswahl auf eine Komponentenauswahl und Komponentenkonfigurierung abgebildet wird. In [SB02] wird statt dessen vorgeschlagen, Merkmale direkt in Kollaborationen zu modularisieren, um den Konfigurationsprozeß zu vereinfachen.

In dieser Arbeit wurde gezeigt, daß diese Art der Modularisierung von Merkmalen möglich ist und sich in der praktischen Anwendung bewährt hat. Die Merkmale der Zielsysteme, die in Kapitel 3 analysiert wurden, konnten als Kollaborationen formuliert und mit der Mixin-Layer-Technik implementiert werden. Ein Teil der Merkmale wie *Synchronisation* und *Verbindung* sind in einer Kollaboration gekapselt worden. Andere Merkmale wie *Client* und *Server* werden durch einfache Kombination von Kollaborationen erzeugt. Auch das *Serviceparadigma*, daß viele andere Merkmale des Systems tangiert, kann durch einfache Kombination konfiguriert werden.

Einige Kollaborationen müssen vor ihrer Verwendung durch den Applikationsentwickler von Hand oder mit Werkzeugunterstützung angepaßt werden, um eine maßgeschneiderte Variante einsetzen zu können. In vielen Fällen gibt es jedoch eine überschaubare Anzahl von Varianten, in denen eine Kollaboration vorliegen kann. Diese Varianten können als *Standardkonfigurationen* archiviert werden, um sie bei der Erstellung eines Familienmitgliedes wiederverwenden zu können. Nur die Integration von Datentypen in die Parameterschicht ist mit Handarbeit verbunden oder auf Werkzeugunterstützung angewiesen.

Crosscutting Concerns

Das Merkmal *Protokoll* ist ein relativ grobgranularer Crosscutting Concern. Eine aspektorientierte Kapselung dieses Merkmals ist mit aspektorientierter Programmierung noch nicht möglich, da *AspectC++* noch keine Templates unterstützt. Dies soll aber später nachgeholt werden.

Die Modularisierung des Merkmals wird gelöst, indem Kollaborationen, die von diesem Merkmal tangiert werden, als protokollspezifische Kollaborationen erstellt werden. Damit ist die Konfigurierung des Protokolls Aufgabe des Applikationsentwicklers, der ein Familienmitglied durch Kombination von Kollaborationen erstellt.

Der Crosscutting Concern *Protokoll* wird damit im weitesten Sinne über *Template*-Programmierung modularisiert. Diese Implementierungsvariante wird von Spinczyk in [SGSP02] S. 28 ff. als “Notlösung” vorgeschlagen, wenn keine programmiersprachliche AOP-Unterstützung zur Verfügung steht. Tangieren mehrere Crosscutting Concerns eine Klasse gleichzeitig, wird die Implementierung ohne AOP-Unterstützung mehr und mehr erschwert [SGSP02]. Insbesondere die Implementierung sehr feingranularer Crosscutting

Concerns ist mit dieser Methode schwer vorstellbar. In dieser Arbeit wurde am Beispiel gezeigt, daß diese Implementierungsvariante mit Hilfe von Mixin-Schichten für *einen* Crosscutting Concern möglich ist.

Jedoch wird mit dem bestehenden Entwurf bereits eine gute Grundlage für eine spätere *aspektorientierte* Implementierung des Merkmals *Protokoll* gelegt. Durch die Trennung protokollspezifischer von protokoll-unspezifischen Kollaborationen sind die Middleware-Komponenten im bestehenden Entwurf bereits von den zukünftigen Protokoll-Aspekten getrennt. Später können die protokollspezifischen Merkmale in die protokoll-unspezifischen Kollaborationen eingewebt werden.

Weitere noch nicht implementierte Crosscutting Concerns wie Dienstgüte-Mechanismen, die im mobilen Bereich eine wichtige Rolle spielen, sollen später in Aspekte gekapselt werden.

Anforderungen an eine Middleware-Architektur für kleine, mobile Systeme

Die wichtigsten "fachlichen Anforderungen" an eine Middleware-Architektur für kleine, mobile Systeme sind nach [AP03a] *statische und dynamische Konfigurierbarkeit*.

Mit *statischer Konfigurierbarkeit* wird das Ziel verfolgt, maßgeschneiderte Systeme für kleine Geräte mit geringen Ressourcen zu erstellen. Eine *maßgeschneiderte* Middleware beansprucht nur die Ressourcen (Speicher, Batterie, Bandbreite etc.), die sie benötigt. Dieses Ziel kann mit einer Kollaborationen-Architektur erreicht werden. Ein maßgeschneidertes Familienmitglied wird erstellt, indem die Kollaborationen ausgewählt werden, die die benötigen Merkmale implementieren. Falls es erforderlich ist, werden diese Kollaborationen konfiguriert und anschließend miteinander verbunden. Das Resultat ist eine maßgeschneidere Middleware. (Später kann dieses Ziel durch aspektorientierte Programmierung unterstützt werden.)

Dynamische Rekonfigurierbarkeit ist erforderlich, um eine Middleware an eine hochdynamische Umgebung, wie es im mobilen Bereich der Fall ist, zur Laufzeit anzupassen. Wechselnde Bandbreite, wechselnde Netze etc. sowie dynamisch ausgehandelte Dienstgüten machen es erforderlich, ein System zur Laufzeit ständig zu rekonfigurieren. Dieses Ziel wird durch den *Strategy Pattern* unterstützt. Die vorgestellte Strategie-Implementierung ermöglicht die Wiederverwendbarkeit von Strategien bei unterschiedlichem Bindungsverhalten. Eine Kontext wird entweder mit einer festen Strategie ausgestattet, um Ressourcen zu sparen und die Strategiemethoden effizient aufzurufen, oder er kann seine Strategie zur Laufzeit austauschen, wobei geringere Effizienz und ein höherer Speicherverbrauch hingenommen werden. Merkmale, die als Strategien implementiert werden, sind so entweder fest in die Architektur integriert oder zur Laufzeit austauschbar. (In späteren Erweiterung soll die dynamische Rekonfigurierbarkeit durch dynamisches AOP erhöht werden, indem Merkmale zur Laufzeit in die Middleware gewebt werden.)

Verteiltes Informationssystem

Die Middleware ist als Grundlage für ein verteiltes Informationssystem geeignet. Auf Basis von Middleware-Servern werden Informationen über Funktions-, Klassenfunktions- oder Objektdienste bereitgestellt. Später können Dienste um *Dienstgüten* erweitert werden. Client-Applikationen können über Dienstaufrufe Anfragen an entfernte Datenbank-Server stellen. Auch dezentrale Informationssysteme von kleinen, mobilen Geräten werden unterstützt. Dazu werden die Geräte mit Server- und Client-Funktionalität ausgestattet und können so als *Peers* auftreten. Um der geringen Ausstattung der Geräte mit Ressourcen gerecht zu werden, werden die einzelnen *Peers* maßgeschneidert.

6.2 Ausblick

Abschließend wird ein kurzer Ausblick auf zukünftige Erweiterungen der Middleware gegeben, die weitere Forschungsarbeiten notwendig machen.

- Aspektorientierte Implementierung
  Mit Hilfe aspektorientierter Programmierung sollen Crosscutting Concerns, wie z.B. die Protokollimplementierung, zukünftig in die Middleware-Schichten eingewoben werden, um die Erstellung von Kollaborationen und damit die Konfiguration von Familienmitgliedern zu vereinfachen. Dazu muß *AspectC++* auch parametrisierten Komponentencode unterstützen.

- Integration weiterer Merkmale in den unteren Schichten
  In den unteren Schichten der Middleware-Familie können weitere Merkmale entworfen und implementiert werden. Beispielsweise ist in einem verteilten System das Server-Merkmal *MultiThread* insbesondere für stationäre Datenbankserver geeignet. Ein Merkmal, um auf *entfernte Klassen-* oder *Objektattribute* direkt zuzugreifen unterstützt die Implementierung verteilter Klassen und Objekte.

- Dienstgüte
  Nach [AP03a] werden in höheren Schichten der Middleware zentrale Dienstgüte-Mechanismen bereitgestellt. Zur *Dienstgüte-Infrastruktur* gehören nach [Bec01] eine zentrale *Ressourcenverwaltung*, eine zentrale *Kostenabrechnung* u.a. Um Dienstgüte bereitzustellen, werden Dienstgüte-Mechanismen in Aspekten implementiert und beim Konfigurationsvorgang in die Middleware gewoben.

- Familie von Nachrichtensystemen
  Weitere Protokollimplementierungen können für die Middleware bereitgestellt werden. Es ist jedoch davon auszugehen, daß keines der vorhandenen Nachrichtensysteme für die Middleware vollständig geeignet ist, z.B. weil einzelne Merkmale nicht unterstützt werden. Die Nachrichtensysteme müssen leider als Monolith verwendet werden und lassen sich nicht in ihrem Funktionsumfang einschränken. Um die

Middleware noch besser konfigurieren zu können, und um Dienstgütemechanismen leichter zu integrieren, sollte eine Familie von Nachrichtensystemen erstellt werden, die unterschiedliche Protokolle, Verbindungsarten, Synchronisationsstrategien usw. implementiert, und statische und dynamische Konfigurierung unterstützt.

- Reflektion & Selbstadaption
  Eine Laufzeitumgebung, die *Reflektion & Selbstadaption* bereitstellt, ermöglicht eine leichtere dynamische Rekonfigurierung der Middleware und unterstützt insbesondere Dienstgüte-Mechanismen.

- Java-Implementierung
  Trotz seiner geringen Performance besitzt Java im mobilen Bereich einige Vorteile gegenüber C++. In einigen Jahren besitzen möglicherweise auch kleine Geräte genügend Leistung, um Java-Programme in einer akzeptablen Geschwindigkeit auszuführen. Insbesondere ist Java-Bytecode unabhängig von der Hardware auf jeder *Virtuellen Java-Maschine* lauffähig. Clients könnten auf diese Weise ad-hoc eine benötigte Funktion zur Laufzeit von einem entfernten Server über das Netzwerk laden. Ebenso wären z.B. *Peers* in der Lage, mobile Applikationen, die sofort lauffähig sind, innerhalb einer Gruppe auszutauschen. Auf diese Weise können sich "on demand" mobile verteilte (Informations-)Systeme bilden.

Literaturverzeichnis

[ABC+02] D. Ayala, C. Browne, V. Chopra, P. Sarangand K. Apshankar, and T. McAllister. *Professional Open Source Web Services.* Wrox Press Ltd., 2002.

[ABE01] F. Akkai, A. Bader, and T. Elrad. Dynamic Weaving for Building Reconfigurable Software Systems. In *OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Tampa Bay, Florida, October 2001.

[AFM97] O. Agesen, S. Freund, and J. Mitchell. Adding type parametrization to the java language. In *OOPSLA 1997*, pages 49–65, 1997.

[AP03a] S. Apel and M. Plack. Komponenten einer Middleware-Plattform für mobile Informationssysteme. In Hagen Höpfner, Gunter Saake, and Eike Schallehn, editors, *Tagungsband zum 15. GI-Workshop Grundlagen von Datenbanken 10.-13. Juni 2003, Preprint Nr. 06/2003*, pages 93–97, Tangermünde, Sachsen-Anhalt, Deutschland, May 2003. Fakultät für Informatik, Universität Magdeburg.

[AP03b] S. Apel and M. Plack. Vergleich von Technologien als Grundlage einer Middleware für mobile Informationssysteme. In Britta König-Ries, editor, *Proceedings of the Workshop Scalability, Persistence, Transactions - Database Mechanisms for Mobile Applications 10.-11. April*, Karlsruhe, Deutschland, 2003.

[BCB+02] G. S. Blair, G. Coulson, L. Blair, H. Duran-Limon, P. Grace, R. Moreira, and N. Parlavantzas. Reflection, self-awareness and self-healing in openorb. In *Proc. of the first workshop on Self-healing systems*, pages 9–14. ACM Press, 2002.

[Bec01] C. R. Becker. *Dienstgüte-Management in verteilten Objektsystemen.* PhD thesis, Johann Wolfgang Goethe - Universität in Frankfurt am Main, 2001.

[BGP+99] D. Beuche, A. Guerrouat, H. Papajewski, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. The PURE Family of Object-Oriented Operating

Systems for Deeply Embedded Systems. In *Proc. of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distribted Computing Systems (ISORC'99)*, Saint Malo, France, May 2-5, 1999.

[BK96] N. Brown and C. Kindel. Distributed component object model protocol – dcom/1.0, November 11 1996. draft-brown-dcom-v1-spec-01.txt.

[BMSP+00] D. Beuche, R. Meyer, W. Schröder-Preikschat, O. Spinczyk, and Ute Spinczyk. Streamlined pure systems. In *Proceedings of the Third ECOOP Workshop on Object-Orientation in Operating Systems (ECOOP-OOOSWS'2000)*, Sophia Antipolis/Cannes, Frankreich, June 12-13 2000.

[BO92] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, 1992.

[BSSP03] D. Beuche, O. Spinczyk, and W. Schröder-Preikschat. Variability Management with Feature Models. In *Proc. of the Software Variability Management Workshop 2003 (SVM 03)*, Groningen, The Netherlandes, February 2003. Technical Report RUG Groningen.

[Cap02] L. Capra. Exploiting Reflection to build Context-Aware Mobile Computing Middleware. Mphil/phd transfer report, University College London, Department of Computer Science, 2002.

[CBM+02] L. Capra, G. S. Blair, C. Mascolo, W. Emmerich, and P. Grace. Exploiting Reflection in Mobile Computing Middleware. *ACM Mobile Computing and Communications Review*, 6(4):34–44, October 2002.

[COR03] Corba 3 release information, 2003. http://www.omg.org/technology/-corba/corba3releaseinfo.htm.

[CS99] D. Chalmers and M. Sloman. A survey of Quality of Service in mobile computing environments. *IEEE Communications Surveys and Tutorials*, 2(2), 1999.

[DH00] J. L. Diaz-Herrera. Domain Engineering, 2000. http://citeseer.nj.nec.com/-diaz-herrera00domain.html.

[Emm00] W. Emmerich. Software engineering and middleware: a roadmap. In *Proceedings of the conference on The future of Software engineering (ICSE 2000) - Future of SE Track*, pages 117–129, Limerick, Ireland, 2000. ACM Press.

[FKV00] D. Fritsch, D. Klinec, and S. Volz. Nexus - Positioning and Data Management Concepts for Location Aware Applications. In *Proc. of the 2nd International Symposium on Telegeoprocessing*, pages 171–184, Nice-Sophia-Antipolos, France, 2000.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GS96] L. Gilman and R. Schreiber. *Distributed Computing with IBM MQSeries*. John Wiley & Sons, 1996.

[Hab76] A. N. Habermann. Modularization and hierarchy in a family of operating systems. In *Communications of the ACM*, pages 266–272, 1976.

[Hal96] Carl L. Hall. *Building ClientServer Applications Using TUXEDO*. John Wiley & Sons, March 1996.

[Har02] M. Harsu. A survey on domain engineering. Technical Report 31, Institute of Software Systems, Tampere Univeristy of Technology, 12 2002. http://practise.cs.tut.fi/pub/papers/domeng.pdf.

[HBG+01] F. J. Hauck, U. Becker, M. Geier, E. Meier, U. Rastofer, and M. Steckermeier. AspectIX: a quality-aware, object-based middleware architecture. In *Proc. of the 3rd IFIP Int. Conf. on Distributed Applications and Interoperable Systems - DAIS*, Krakow, Poland, Sep. 17-19, 2001.

[HBS99] M. Hapner, R. Burridge, and R. Sharma. Java message service specification. technical report. Technical report, Sun Microsystems, Nov 1999. http://java.sun.com/products/jms.

[Hud94] E. S. Hudders. *CICS: A Guide to Internal Structure*. John Wiley & Sons, 1994.

[KCH+90] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, cmu/sei-90-tr-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, November 1990.

[KdRB91] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[KHH+01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[KLM+97] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, 1997.

[KRL+00] F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, L. Claudio M., and R. H. Campbell. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, New York, April 3-7, 2000. ACM/IFIP.

[Led99] T. Ledoux. OpenCorba: A Reflective Open Broker. *Lecture Notes in Computer Science*, 1616, 1999.

[MCE02] C. Mascolo, L. Capra, and W. Emmerich. Mobile Computing Middleware. *NETWORKING Tutorials*, 2002. http://citeseer.nj.nec.com/596660.html.

[OTN00] Oracle Oracle Technology Network. Oracle9i application server wireless, 2000. http://technet.oracle.com/products/iaswe/content.html.

[Par76] D. L. Parnas. On the design and development of program families. *IEEE Transactions On Software Engineering*, SE-2(1), March 1976.

[PGA02] A. Popovici, T. Gross, and G. Alonso. Dynamic Weaving for Aspect Oriented Programming. In *1st International Conference on Aspect-Oriented Software Development (AOSD)*, Enschede, The Netherlands, April 2002.

[PMR99] G. P. Picco, A. L. Murphy, and G.-C. Roman. LIME: Linda Meets Mobility. In D. Garlan, editor, *Proc. of the 21st International Conference on Software Engineering (ICSE'99)*, pages 368–377, Los Angeles, CA, USA, May 1999. ACM Press. Also available as Technical Report WUCS-98-21, July 1998, Washington University in St. Louis, MO, USA.

[SB98] Y. Smaragdakis and D. Batory. Implementing Layered Designs with Mixin Layers. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 550–570. Springer-Verlag LNCS 1445, 1998.

[SB02] Y. Smaragdakis and D. Batory. Mixin layers: An object-oriented implementation technique for refinements and collabroation-based designs. *ACM Transactions on Software Engineering and Methodology*, 11(2):215–255, April 2002.

[SGSP02] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An Aspect-Oriented Extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, Sydney, Australia, February 18–21 2002. IEEE Computer Society.

[SLM98] D. C. Schmidt, D. L. Levine, and S. Mungee. The design of the TAO real-time object request broker. *Computer Communications*, 21(4):294–324, 1998.

[Smi82] B. C. Smith. *Reflection and Semantics in a Procedural Programming Language*. PhD thesis, MIT, 1982.

[SOA03] SOAP Version 1.2 W3C Recommendation 24 June 2003, 2003. http://www.w3.org/TR/soap12-part0/.

[Sri95] R. Srinivasan. RFC 1831: RPC: Remote Procedure Call Protocol Specification Version 2, August 1995. Status: PROPOSED STANDARD, ftp://ftp.internic.net/rfc/rfc1831.txt, ftp://ftp.math.utah.edu/pub/rfc/rfc1831.txt.

[Str00] B. Stroustrup. *Die C++ Programmiersprache*. Addison-Wesley, 4 edition, 2000.

[SUN99] Java Remote Method Invocation API Specification, 1999. http://java.sun.com/j2se/1.3/docs/guide/rmi/spec/rmiTOC.html.

[Vos97] G. Vossen. The CORBA specification for cooperation in heterogeneous information systems. In P. Kandzia and M. Klusch, editors, *Proceedings ot the First International Workshop on Cooperative Information Agents*, volume 1202 of *LNAI*, pages 101–115, Berlin, 1997. Springer.

[Wal98] J. Waldo. Javaspaces specification 1.0. Technical report, Sun Microsystems, March 1998.

[Wei93] M. Weiser. Hot topics: Ubiquitous computing. *IEEE Computer*, October 1993.

[WMLF99] P. Wyckoff, S.W. McLaughry, T.J. Lehman, and D.A. Ford. T spaces. *IBM Systems Journal*, 37(3), 1999.

[ZBS97] J. A. Zinky, D. E. Bakken, and R. E. Schantz. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems*, 3(1), 1997.

Selbständigkeitserklärung

Hiermit versichere ich, daß ich die vorliegende Diplomarbeit selbständig und nur unter Verwendung erlaubter Hilfsmittel angefertigt habe.

Helge Sichting
Magdeburg, den 14. Januar 2004