Diplomarbeit

Framework-Evolution

am Beispiel eines Frameworks zur Steuerung von Tk-Anlagen

vorgelegt von

Jan Willamowius
Rückertstr. 27
22089 Hamburg
Matrikel-Nr. 4175391

September 1997

Erstbetreuer: Prof. Dr. Heinz Züllighoven

Zweitbetreuer: Dr. Walter Bischofberger

Diese Diplomarbeit wurde am Fachbereich Informatik der Universität Hamburg zur teilweisen Erfüllung der Anforderungen zur Erlangung des Titels Diplom-Informatiker eingereicht.

Erklärung

Hiermit versichere ich, diese Arbeit selbständig und unter ausschließlicher Zuhilfenahme der in der Arbeit aufgeführten Hilfsmittel erstellt zu haben.

Hamburg, den 05.09.1997

Jan Willamowius

Rückertstr. 27
D-22089 Hamburg
EMail: jan@janhh.shnet.org
Matrikel-Nr. 4175391

Betreuung

Prof. Dr. Heinz Züllighoven (Erstbetreuer)
Dr. Walter Bischofberger (Zweitbetreuer)

Prof. Dr. Heinz Züllighoven

Fachbereich Informatik
Arbeitsbereich Softwaretechnik
Universität Hamburg
Vogt-Kölln-Straße 30
D-22527 Hamburg
Deutschland

Dr. Walter Bischofberger

TakeFive Software AG
Eidmattstr. 51
CH-8052 Zürich
Schweiz

Inhaltsverzeichnis

1 Einleitung 1

1.1 Ausgangspunkte und Aufbau der Arbeit 1

1.2 Verwendete Notationen 2

1.3 Danksagung 3

2 Frameworks 5

2.1 Definition und Terminologie von Frameworks 5

2.1.1 Hot Spots und Frozen Spots 6

2.2 Techniken zur Framework-Konstruktion 7

2.2.1 Vererbung und abstrakte Klassen 7

2.2.2 Parametrisierung 9

2.2.3 Schnittstellen-Gestaltung 10

2.2.3.1 Kunden- und Spezialisierungs-Schnittstelle 10

2.2.3.2 Trennung von Parametern und Einstellungen 11

2.2.3.3 Dynamische Bindung 12

2.3 Kategorien von Frameworks 14

2.4 Die Beziehung zwischen Entwurfsmustern und Frameworks 17

2.5 Vorgehensweise bei der Framework-Entwicklung 18

2.6 Framework-Evolution 19

2.6.1 Gründe für Framework-Evolution 20

2.6.2 Gerichtete und ungerichtete Evolution 21

2.6.3 Typische Merkmale der Framework-Evolution 22

2.6.4 Auswirkungen der Framework-Evolution 24

3 Anwendungsgebiet: Telefonie 27

3.1 Anwendungsgebiete für Telefonie-Systeme 27

3.2 Das Micrologica Communication Center (MCC) 28

3.3 Hardware-Konfigurationen für Computer-Telefonie 29

3.4 Telefonie-APIs 30

3.4.1 Kommunikation zwischen Anwendung und Tk-Anlage 32

3.4.2 Das Half-Call-Modell 33

3.5 Probleme der Computer-Telefonie 35

3.5.1 Vielfältige Hardware-Landschaft 35

3.5.2 Keine exklusive Steuerung bei LAN-zentrierten Konfigurationen 36

3.5.3 Verlorene Events 36

3.5.4 Unzureichende Standardisierung 36

3.6 Ausgangspunkt zur Framework-Entwicklung 37

4 Das Micrologica-Telefonie-Framework ... 39
4.1 Bisherige Architektur ... 39
4.1.1 Probleme der bisherigen Architektur ... 40
4.2 Ziele der Framework-Entwicklung ... 41
4.3 Aufbau des Frameworks ... 41
4.3.1 Die Bereiche im Überblick ... 43
4.3.2 Aufträge an die Tk-Anlage: Requests ... 44
4.4 Benutzung des Telefonie-Frameworks ... 46
4.5 Vorgehensweise bei der Framework-Entwicklung ... 47
5 Erfahrungen aus der Evolution des Micrologica-Telefonie-Frameworks ... 49
5.1 Unnötige Compilezeit-Abhängigkeiten ... 49
5.1.1 Arten von Abhängigkeiten ... 49
5.1.2 Reduktion der Abhängigkeiten ... 50
5.1.3 Bessere Generierungswerkzeuge ... 52
5.2 Umstellung des Meta-Object-Protocol ... 52
5.2.1 Umgang mit kommenden Spracheigenschaften ... 54
5.2.2 Implementation eines Meta-Object-Protocol ... 55
5.2.3 Designrichtlinien für die Verwendung des Meta-Object-Protocol ... 58
5.3 Evolution der Zustandsmodellierung ... 59
5.3.1 Aufzählungstypen ... 63
5.3.2 Eine wiederverwendbare Singleton-Implementation ... 63
5.4 Entwicklung der Event-Normalisierung ... 66
5.5 Evolution der Kern-Abstraktion ... 69
5.6 Zusammenfassung ... 71
6 Framework-Partitionierung ... 73
6.1 Die Partitionen des Telefonie-Frameworks ... 73
6.2 Abgrenzung des Anwendungsbereichs für ein Framework ... 77
6.2.1 Das Design-Zentrum des Telefonie-Frameworks ... 78
6.3 Notwendigkeit der Framework-Strukturierung ... 79
6.3.1 Framework-Partitionierung ... 80
6.3.2 Framework-Strukturierung ... 82
6.4 Konstruktionstechniken für Partitionen ... 84
6.5 Partitionierungskriterien ... 87
6.6 Zusammenfassung ... 89
7 Fazit und Ausblick ... 91
Anhang A: Literatur ... 93
Anhang B: Glossar der Telefonie-Begriffe ... 101

Abbildungsverzeichnis

Abbildung 1: Notationen für Klassen ... 2
Abbildung 2: Beziehungen zwischen Klassen ... 2
Abbildung 3: Interaktionsdiagramm ... 3
Abbildung 4: Namenspräfixe ... 3
Abbildung 5: Spezialisierung einer abstrakten Klasse ... 8
Abbildung 6: Anpassung einer Framework-Klasse durch Parametrisierung ... 9
Abbildung 7: Kunden- und Spezialisierungs-Schnittstelle ... 11
Abbildung 8: Getrennte Übergabe von Parametern und Einstellungen ... 12
Abbildung 9: Lose Kopplung durch Entwurfsmuster ... 18
Abbildung 10: Vergleich der Spezialisierungs-Techniken ... 22
Abbildung 11: Entwicklung der Quelltext-Größe bei ET++ [Gamma 92, S. 161] ... 23
Abbildung 12: LAN- bzw. Server-zentrierte Konfiguration [Burton 95] ... 29
Abbildung 13: PC-zentrierte Konfiguration [Burton 95] ... 30
Abbildung 14: Position des Tk-Anlagen-Treibers ... 31
Abbildung 15: Das Half-Call-Modell einer einfachen Verbindung ... 33
Abbildung 16: Das Modell einer Konferenzschaltung ... 34
Abbildung 17: Das Modell einer Rückfrage ... 34
Abbildung 18: Meine Modellierung des Half-Call-Modells ... 35
Abbildung 19: Zustandsmodell für Connections [Novell 95, S. 3-13] ... 37
Abbildung 20: Telefonie-Prozesse ... 40
Abbildung 21: Die Bereiche des Micrologica-Telefonie-Frameworks ... 42
Abbildung 22: MakeCall-Klassen ... 45
Abbildung 23: Typische Verwendung des Frameworks ... 46
Abbildung 24: Reduzierung der Abhängigkeiten ... 50
Abbildung 25: Abhängige Dateien ... 51
Abbildung 26: Vererbungshierarchie der Requests ... 52
Abbildung 27: Vererbungshierarchie der Events ... 53
Abbildung 28: RTTI-Makros für Compiler ohne RTTI ... 57
Abbildung 29: Einfügen der RTTI-Informationen in Anwendungsklassen ... 57
Abbildung 30: Verwendung der RTTI-Informationen zur Laufzeit ... 57
Abbildung 31: RTTI-Makros für Compiler mit RTTI ... 58
Abbildung 32: Zustandsmodellierung mit Micrologica-FSM-Klassen ... 60
Abbildung 33: Zustandsmodellierung mit State-Muster ... 61
Abbildung 34: Singleton-Implementierung als Template ... 65
Abbildung 35: Vererbungshierarchie mit Singleton-Template ... 66

Abbildung 36: Event-Normalisierung durch den Tk-Anlagen-Treiber 67
Abbildung 37: Erste Version der Event-Normalisierung 68
Abbildung 38: Zweite Version der Event-Normalisierung 69
Abbildung 39: Die ersten Entwicklungsstufen des Designs 70
Abbildung 40: Implementierte Versionen 71
Abbildung 41: Die Partitionen des Telefonie-Frameworks 75
Abbildung 42: Unterstrukturierung der softwaretechnischen Basis 76
Abbildung 43: Unterstrukturierung der Telefonie-Basis-Objekte 76
Abbildung 44: Funktion des Fassade 78
Abbildung 45: Zu feste Kopplung der virtuellen Tk-Anlage 78
Abbildung 46: Lose Kopplung der virtuellen Tk-Anlage 79
Abbildung 47: Bildung der Framework-Struktur durch vertikale Partitionierung 82
Abbildung 48: Horizontale Partitionierung der Telefonie-Basis-Objekte 83
Abbildung 49: Eskalierung 85
Abbildung 50: Degradierung 85
Abbildung 51: Redundanz 86
Abbildung 52: Mögliche Partitionierung eines umfangreicheren Telefonie-Frameworks 88

Definitionsverzeichnis

Definition 1: Software-Design ... 6
Definition 2: Framework (vorläufige Fassung) ... 6
Definition 3: Hot Spot ... 6
Definition 4: Frozen Spot ... 7
Definition 5: Abstrakte Klasse ... 7
Definition 6: Application-Framework ... 14
Definition 7: GUI-Framework ... 14
Definition 8: Fachliches Framework ... 15
Definition 9: Softwaretechnisches Framework ... 15
Definition 10: White-Box-Framework ... 16
Definition 11: Black-Box-Framework ... 16
Definition 12: Vererbungsbasiertes Framework ... 16
Definition 13: Kompositionsbasiertes Framework ... 16
Definition 14: Entwurfsmuster ... 17
Definition 15: Framework-Evolution ... 20
Definition 16: Gerichtete Evolution ... 21
Definition 17: Ungerichtete Evolution ... 21
Definition 18: Telefonie ... 27
Definition 19: Request ... 32
Definition 20: Event ... 33
Definition 21: Event-Normalisierung ... 37
Definition 22: Fachliche Abhängigkeiten ... 49
Definition 23: Compilezeit-Abhängigkeit ... 50
Definition 24: Offen-Geschlossen-Prinzip ... 63
Definition 25: Design-Zentrum ... 77
Definition 26: Framework-Partition ... 81
Definition 27: Framework (endgültige Fassung) ... 81
Definition 28: Framework-Struktur ... 82
Definition 29: Horizontale Partitionierung ... 83
Definition 30: Vertikale Partitionierung ... 83
Definition 31: Stufenbildung ... 83
Definition 32: Eskalierung ... 84
Definition 33: Degradierung ... 85
Definition 34: Redundanz ... 86

1 Einleitung

Der Wechsel allein ist das Beständige.
ARTHUR SCHOPENHAUER

Das Brockhaus-Lexikon definiert Evolution als eine "langsam, kontinuierlich fortschreitende Entwicklung". Auch Software-Systeme sind einer kontinuierlichen Fortentwicklung unterworfen. Für einzelne Anwendungen ist es eine akzeptierte Tatsache, daß sie nicht einmal geradlinig entwickelt und fortan unverändert eingesetzt werden können. Doch wie stellt sich die Situation für Frameworks dar ? Können sie eine solide und vor allem stabile Basis für Anwendungen sein, oder sind auch Frameworks einer Evolution unterworfen ? Und wenn sie einer Evolution unterworfen sind, wie kann man mit ihr umgehen ?

1.1 Ausgangspunkte und Aufbau der Arbeit

Im Rahmen einer Kooperation zwischen dem Softwaretechnik-Center des Fachbereichs Informatik der Universität Hamburg und der Firma Micrologica Computersysteme GmbH, Bargteheide, hatte ich Gelegenheit, die langjährige Erfahrung von Micrologica im Telefonie-Bereich in ein Framework zur Steuerung von Telekommunikationsanlagen (Tk-Anlagen) umzusetzen.

Anhand der Entwicklung dieses Frameworks habe ich auch viele Erfahrungen gemacht, die sich auf die Evolution von Frameworks beziehen. Da die Evolution von Frameworks noch ein offenes Forschungsthema ist, möchte ich in dieser Arbeit die Erfahrungen und meine Schlußfolgerungen daraus vorstellen. Die wenigen Publikationen, die es in diesem Bereich gibt, habe ich als "Absicherung" verwendet, um zum einen einen breiteren Blickwinkel zu bekommen, auch auf Aspekte, die bei meiner konkreten Arbeit nicht aufgetreten sind. Andererseits dienen sie mir als Indiz für die Übertragbarkeit meiner Erfahrungen auf andere Frameworks.

Besonders beeinflußt wurde diese Arbeit von dem Artikel von Roberts und Johnson, in dem sie in einer Mustersprache beschreiben, welche Evolutionsschritte ein reifendes Framework in einer idealisierten Evolution durchmachen kann (siehe [RobJohn 96]). Hierauf aufbauend möchte ich den Gründen nachspüren, die bei der Framework-Konstruktion die Basis für Art und Umfang der Evolution legen.

In **Kapitel 2** gebe ich zunächst einen Überblick über die Begriffe und Techniken, die bei der Konstruktion von Frameworks verwendet werden und den Begriff der Framework-Evolution definieren. Dieses Kapitel soll die begrifflichen Grundlagen für die weitere Diskussion legen.

Dann beschreibe ich in **Kapitel 3** den Anwendungsbereich der Telefonie-Systeme und erläutere, welche Probleme mit dem erstellten Framework gelöst werden sollen. Die hier vorgestellten Eigenheiten der Telefonie werden in den weiteren Kapiteln immer wieder als Ursachen für diverse Evolutionsschritte eine Rolle spielen. Im Anschluß daran wird in **Kapitel 4** ein grober Überblick über das erstellte Framework gegeben. Um das Verständnis dieser Kapitel zu erleichtern, enthält **Anhang B** ein Glossar der verwendeten Telefonie-Begriffe.

Einige Evolutionsschritte des Telefonie-Frameworks werden in **Kapitel 5** exemplarisch vorgestellt und ihre Ursachen und Auswirkungen diskutiert. Aus diesen Erfahrungen werde ich dort, wo es möglich ist, Folgerungen ableiten, die auch auf andere Frameworks übertragbar sind.

Eines der wichtigsten Ergebnisse dieser Arbeit wird in **Kapitel 6** vorgestellt. Ausgehend von den Erfahrungen mit der Evolution des Frameworks bin ich zu einer Strukturierungs-Technik für Frameworks gekommen, der "Partitionierung". Diese Technik ähnelt dem "Layering" von Frameworks, wie es beispielsweise in [BGKLRZ 97] vorgeschlagen wird, ist aber durch konkrete Konstruktionstechniken und Strukturierungsrichtlinien angereichert.

Kapitel 7 zieht ein Resümee der Arbeit und gibt einen Ausblick auf zukünftige Entwicklungen.

In **Anhang A** ist die verwendet Literatur aufgeführt.

1.2 Verwendete Notationen

Für Klassendiagramme wird eine an [GHJV 95] angelehnte Notation verwendet:

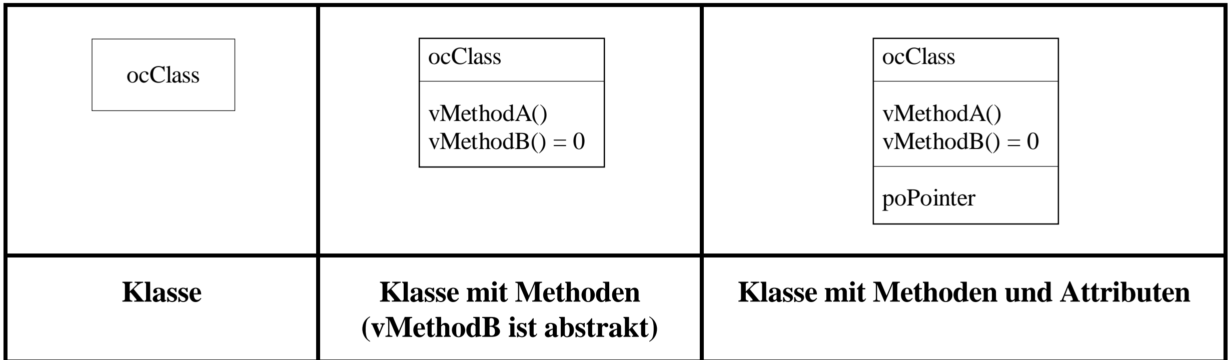

Abbildung 1: Notationen für Klassen

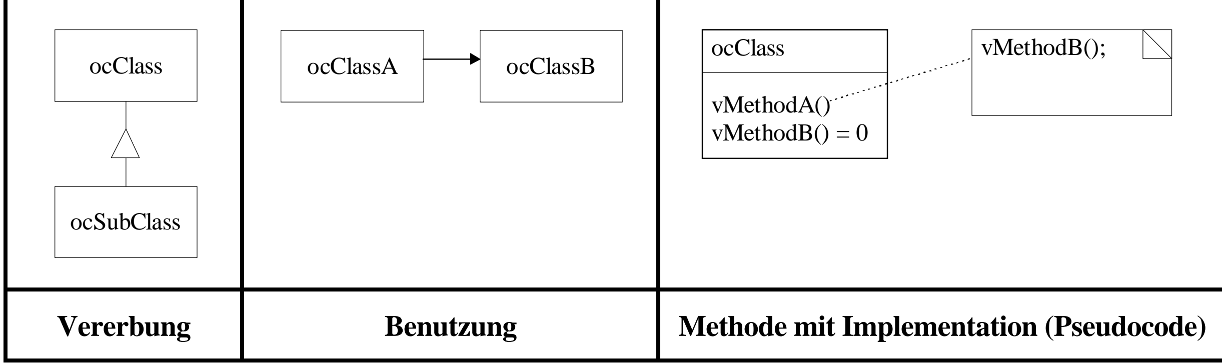

Abbildung 2: Beziehungen zwischen Klassen

Zur Darstellung der Kommunikation zwischen Prozessen werden einfache Interaktionsdiagramme in der Art von Abbildung 3 verwendet. Da es sich um (gepufferte) asynchrone Kommunikation handelt, muß nicht dargestellt werden, wo der Kontrollfluß verläuft - er wird durch diese Kommunikation nur indirekt beeinflußt.

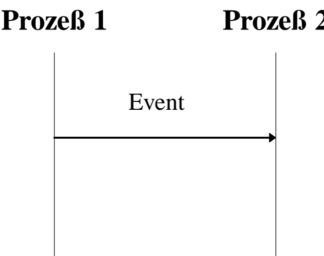

Abbildung 3: Interaktionsdiagramm

Die meisten Beispiele mit Quelltext in dieser Arbeit folgen der bei Micrologica durch Qualitätsrichtlinien [Koschek 95] vorgeschriebenen Form der ungarischen Notation. Es werden folgende Präfixe für Namen verwendet (auszugsweise):

oc	Klassentyp (object class)
o	Exemplar einer Klasse (object)
p	Pointer
v	Void
i	Integer
b	Bool

Abbildung 4: Namenspräfixe

1.3 Danksagung

Ich möchte mich bedanken bei der Firma Micrologica Computersysteme GmbH und allen Mitarbeitern, die mich durch viele Gespräche und ihre konstruktive Zusammenarbeit bei der Erstellung des Telefonie-Frameworks unterstützt haben.

Mein weiterer Dank gilt Prof. Heinz Züllighoven und Dr. Walter Bischofberger für die Betreuung dieser Diplomarbeit. Insbesondere danke ich auch Wolfgang Strunk für unzählige erhellende Diskussionen und Thomas Pfohe für die Mitarbeit bei der Erstellung des Telefonie-Frameworks.

2 Frameworks

Dieses Kapitel soll die begriffliche Grundlage legen für die spätere Beschreibung des Micrologica-Telefonie-Frameworks zum einen, und die Diskussion der Framework-Evolution zum anderen.

Der Grundgedanke von Frameworks besteht darin, Implementationen für den größten Teil einer Kategorie von Anwendungen bereitzustellen ("generische Anwendungen"). Für eine neue Anwendung werden lediglich die Teile hinzugefügt, die diese Anwendung von anderen dieser Kategorie unterscheiden.

Ein Ziel dieses Vorgehens ist es, möglichst schnell eine Reihe ähnlicher Anwendungen mit dem Framework zu erstellen. Alle Anwendungen, die auf demselben Framework basieren, verwenden sowohl das Design als auch die Implementierung des Frameworks gemeinsam, wodurch Fehlerkorrekturen an zentraler Stelle im Framework allen abgeleiteten Anwendungen zugute kommen, so daß eine höhere Qualität der Software erreicht werden kann.

Dieses Kapitel soll einen Überblick über die Begriffswelt und Technik von Frameworks in der objektorientierten Programmierung geben.

2.1 Definition und Terminologie von Frameworks

Unter einem Framework verstehe ich in Übereinstimmung mit [BKGSZ 95] eine Konfiguration von Klassen, die den nötigen Ablauf zur Bewältigung einer Reihe verwandter Problemstellungen beinhaltet und vergegenständlicht. Es stellt eine generische Lösung für einen Anwendungsbereich dar, die für die jeweiligen Anwendungsfälle spezialisiert werden muß.

In [GHJV 95, S. 26f] werden Frameworks folgendermaßen definiert:

> "The framework dictates the architecture of your application. It will define the overall structure, its partitioning into classes and objects, the key responsibilities thereof, how the classes and objects collaborate, and the thread of control. A framework predefines these design parameters, so that you, the application designer/implementer, can concentrate on the specifics of your application. [...] Frameworks thus emphasize *design reuse* over code reuse, ...".

Entscheidend ist hier bei der Begriff *Design*. Auf fachlicher Ebene meint Design den Vorgang der Modellbildung. Dieses fachliche Modell des Anwendungsbereichs ist der Kern des Frameworks. Auf technischer Ebene spiegelt sich die Modellbildung in den modellierten Klassen bzw. Subsystemen und deren Beziehungen untereinander wider.

Das Zitat macht deutlich, wie stark der Einfluß des Frameworks auf das Design einer speziellen Anwendung ist: Es diktiert das Design in "seinem Bereich". Alle Anwendungen, die mit dem Framework erstellt werden, müssen sich am fachlichen Modell des Frameworks orientieren und können es nur in Grenzen adaptieren. Wie wir später sehen werden, gibt es verschiedene Kategorien von Frameworks, die jeweils einen Teil einer Anwendung oder auch die gesamte Anwendung als "ihren Bereich" betrachten.

Definition 1: Software-Design

Der Prozeß der fachlichen und technischen Modellbildung für ein Software-System wird als Software-Design (im folgenden kurz Design) bezeichnet. Das fachliche Design spiegelt sich technisch in der Aufteilung des Systems in Klassen und Subsysteme und deren Beziehungen bzw. Interaktion untereinander wider.

Durch diese Vorgabe des Designs wird eine Wiederverwendung eines Designs zusammen mit seiner Implementation möglich. Gerade die Wiederverwendung auf Designebene war bisher mit Klassenbibliotheken nicht möglich und stellt den entscheidenden Vorteil bei der framework-basierten Softwareentwicklung dar.

Im Gegensatz zu Frameworks sind Klassenbibliotheken Sammlungen von allgemein verwendbaren Klassen, z.B. Containerklassen, Strings, komplexe Zahlen etc. Diese Klassen sind dazu gedacht, einzeln verwendet zu werden. Sie stellen kein Design zur Verfügung.

Definition 2: Framework (vorläufige Fassung)[1]

Ein Framework stellt ein generisches Design zu einer Reihe verwandter Probleme zur Verfügung. Es enthält bereits einen Teil der Implementation, muß für konkrete Anwendungen aber noch spezialisiert bzw. parametrisiert werden. Die Anwendung verwendet das Framework als Ganzes, so daß auch der Kontrollfluß der Anwendung vom Framework vorgegeben wird.

Für den Begriff "Framework" gibt es auch die deutsche Übersetzung "*Rahmenwerk*". Ich werde jedoch durchgängig den englischen Begriff Framework verwenden, der sich inzwischen auch im Deutschen eingebürgert hat.

2.1.1 Hot Spots und Frozen Spots

Wenn ein Framework den allgemeinen Teil einer Gruppe von Anwendungen verkörpert, so stellt sich die Frage, wie man herausfindet, was denn "allgemein" und was "speziell" ist an dieser Gruppe von Anwendungen.

Unglücklicherweise ist dies eine der schwierigsten Fragen bei der Framework-Entwicklung, die sich nur mit sehr viel Wissen über den Anwendungsbereich und mit einem evolutionären Vorgehen beantworten läßt.

Wolfgang Pree (siehe [Pree 94a], [Pree 94b]) hat zwei Begriffe geprägt, die bei der Diskussion dieser Frage hilfreich sind: Die Teile des Anwendungsbereichs, die von Anwendung zu Anwendung variieren und daher im Framework flexibel gehalten werden müssen, bezeichnet er als *hot spots.* An den hot spots werden häufig Änderungen erwartet.

Definition 3: Hot Spot

Ein Bereich eines Frameworks, der nicht allgemein für alle Anwendungen des Frameworks definiert werden kann, wird als hot spot bezeichnet.

---

[1] Diese Definition wird in Kapitel 6.3.1 noch um zusätzliche Kriterien erweitert.

Im Gegensatz zu den hot spots werden die Teile des Anwendungsbereichs, die bei allen Anwendungen gleich sind und daher im Framework fest implementiert werden können, nach Pree als *frozen spots* bezeichnet.

Definition 4: Frozen Spot

Ein Bereich eines Frameworks, der in allen auf dem Framework basierenden Anwendungen gleich ist, wird als frozen spot bezeichnet.

Diese Begriffe sind wichtig bei der Konstruktion eines Frameworks, um auszudrücken, welche Bereiche adaptierbar sein müssen und welche festliegende Zusammenhänge implementieren. Sie liefern jedoch nur das Vokabular und nicht die Antwort. In Kapitel 3.5 werde ich beschreiben, wie die hot spots und frozen spots des Micrologica-Telefonie-Frameworks identifiziert wurden.

2.2 Techniken zur Framework-Konstruktion

Zur Konstruktion von Frameworks werden eine Reihe von Techniken der objektorientierten Konstruktion eingesetzt. Die wichtigsten dieser Techniken möchte ich im folgenden kurz vorstellen und ihre Terminologie beschreiben.

2.2.1 Vererbung und abstrakte Klassen

Da ein Framework das allgemeine Verhalten einer Gruppe von Anwendungen beschreibt, muß es für den konkreten Anwendungsfall spezialisiert werden. Die am häufigsten eingesetzte Technik hierzu ist die Vererbung.

Eine Art, Vererbung einzusetzen, ist, durch eine gemeinsame Oberklasse das Interface einer Klasse (bzw. Gruppe von Klassen) zu spezifizieren. Dadurch, daß von der Oberklasse keine Exemplare (Objekte) erzeugt werden können, wird die Funktion der abstrakten Klasse als Definition einer Schnittstelle bzw. eines Protokolls deutlich gemacht. Dies schließt nicht aus, daß in ihr bereits einige Operationen implementiert sind, die von allen Unterklassen verwendet werden.

Definition 5: Abstrakte Klasse

Eine unvollständig implementierte Klasse, von der keine Exemplare erzeugt werden können, wird als abstrakte Klasse bezeichnet.

Nach [JohnRusso 91] und [GHJV 95, S. 325ff] kann man 4 Arten von Operationen bei abstrakten Klassen unterscheiden:

- *Abstrakte Operationen* werden nicht implementiert. Dies ist Aufgabe der jeweiligen Unterklassen. Die abstrakten Operationen spezifizieren aber das Interface aller Unterklassen. Sie fungieren als Platzhalter.

- *Hook-Operationen* haben eine sehr ähnliche Funktion wie die abstrakten Operationen. Sie haben jedoch ein Default-Verhalten, so daß sie von den Unterklassen nicht zwingend überladen werden müssen.

- *Basis-Operationen* hingegen sind Operationen, die bereits in der abstrakten Klasse vollständig implementiert werden können.

- *Template-Operationen* implementieren einen Algorithmus basierend auf anderen Operationen. Der Algorithmus wird vollständig angegeben, zur Ablauffähigkeit fehlen ihm aber die Implementationen der abstrakten Operationen. [2]

Bevor abstrakte Klassen in einer Anwendung verwendet werden können, müssen sie spezialisiert werden. Hierzu müssen mindestens die abstrakten Operationen in der spezialisierten Klasse implementiert werden.

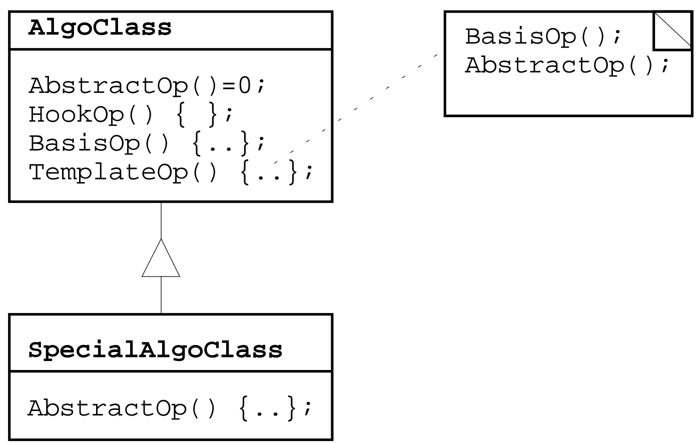

Abbildung 5: Spezialisierung einer abstrakten Klasse

In C++ ist eine Klasse abstrakt (auch dt. aufgeschoben, engl. deferred), wenn eine ihrer Optionen abstrakt ist. In einigen objektorientierten Sprachen (z.B. Eiffel, Java) können Klassen auch direkt als abstrakt markiert werden. Die sog. "Interfaces" in Java dürfen keine Implementationen enthalten. Auf diese Weise wird die Aufgabe der Protokoll-Definition von der Definition eines gemeinsamen Verhaltens getrennt.

In Sprachen, die keine explizite Deklaration von Interfaces unterstützen, wie z.B. C++, sollte auf andere Weise sichergestellt werden, daß von der Klasse keine Exemplare erzeugt werden können. In C++ ist dies dadurch zu erreichen, daß der Konstruktor der Klasse als 'protected' deklariert wird und somit nicht zugreifbar ist. Diese Technik ist insbesondere bei Klassen wichtig, die konzeptionell abstrakt sind, die aber keine abstrakten Operationen, sondern nur Hook-Operationen besitzen. Anderenfalls kann der Compiler das Erzeugen von Exemplaren nicht verhindern, da Hook-Operationen ja eine leere Default-Implementation besitzen.

Die Verwendung von abstrakten Oberklassen führt zu Systemen mit einer loseren Kopplung als bei direkten Beziehungen zwischen den Klassen. Kollaborierende Klassen beziehen sich so immer nur auf ein Protokoll (Interface), nicht aber auf eine konkrete Implementierung. Daher

[2] Template-Operationen sollten auf keinen Fall mit C++ Templates ("generischen Klassen") verwechselt werden!

können einzelne Klassen leichter ausgetauscht werden durch solche, die das gleiche Protokoll beherrschen, aber eine andere Implementierung besitzen.

Eine typische Eigenschaft von Frameworks ist der invertierte Kontrollfluß: Bei traditioneller Anwendungsentwicklung ruft üblicherweise die Anwendung den Bibliothekscode und bestimmt somit auch den überwiegenden Teil des Kontrollflusses. Frameworks hingegen verwenden oft abstrakte Klassen und sehen abstrakte Operationen vor, in die der Anwendungscode eingefügt wird. Die Anwendung wird dann zu passender Zeit vom Framework aufgerufen. Dadurch liegt die Ablaufsteuerung nicht mehr in der Anwendung, sondern wird vom Framework definiert (vgl. [BKGSZ 95]). Oft wird auf diese Weise die gesamte Ereignisschleife gekapselt.

Die Verwendung des invertierten Kontrollflusses wird auch als *Hollywood-Prinzip* (nach dem Motto "don't call us, we call you") bezeichnet (siehe [Vlissides 96a]).

2.2.2 Parametrisierung

Eine Alternative zur Anpassung von Framework-Klassen durch Vererbung ist die Parametrisierung. Statt eine abstrakte Methode vorzusehen, die von der Unterklasse implementiert werden muß, wird die Übergabe eines Parameter-Objekts an die Framework-Klasse vorgesehen, an das die Ausführung der Operation delegiert wird.

Für die Parameter-Objekte muß eine Oberklasse definiert werden, die die Signatur der spezialisierbaren Operation festlegt. Diese Operation ist meist eine abstrakte Operation, es ist aber auch der Einsatz einer Hook-Operation möglich, so daß eine Spezialisierung entfallen kann.

Abbildung 6 zeigt die notwendige Klassenhierarchie bei der Spezialisierung durch Parameter-Objekte.

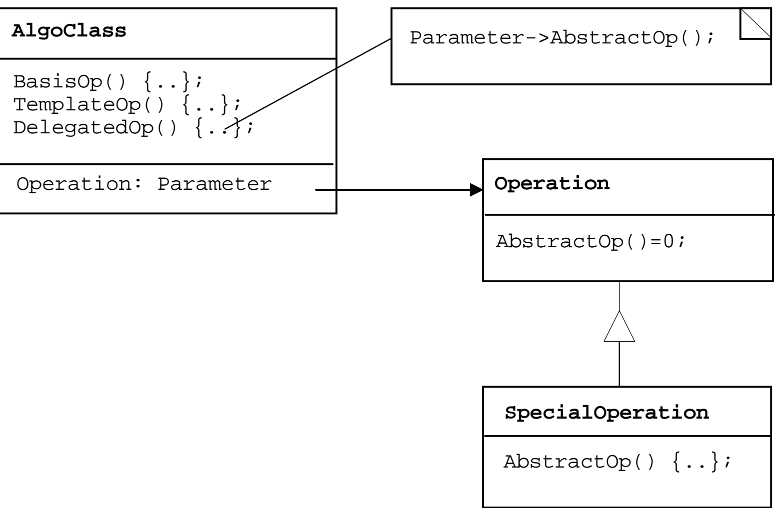

Abbildung 6: Anpassung einer Framework-Klasse durch Parametrisierung

Die Parametrisierung führt üblicherweise zu einer geringeren Kopplung als die Vererbung, da keine Operationen der Oberklasse aufgerufen werden müssen und die zu implementierende

Schnittstelle meist einen geringeren Umfang hat. Andererseits ist die Spezialisierung durch Parametrisierung auf die von vornherein festgelegten Modifikationsmöglichkeiten beschränkt.

Ein weiterer wichtiger Vorteil der Parametrisierung ist die Tatsache, daß erst zur Laufzeit entschieden werden muß, welches Parameter-Objekt verwendet wird, während Vererbung zur Compilezeit stattfindet.

2.2.3 Schnittstellen-Gestaltung

Ein wichtiger Aspekt der Framework-Konstruktion besteht in der Schnittstellen-Gestaltung der Framework-Klassen. Zum einen sollte die intendierte Verwendung der Operationen für den Framework-Benutzer verdeutlicht werden. Dies kann durch eine explizite Unterscheidung in eine Kunden-Schnittstelle und einen Spezialisierungs-Schnittstelle geschehen (siehe Kapitel 2.2.3.1). Andererseits müssen auch die Auswirkungen der Schnittstellen-Gestaltung auf die Adaptierbarkeit des Frameworks berücksichtigt werden, die in Kapitel 2.2.3.2 und 2.2.3.3 diskutiert werden.

2.2.3.1 Kunden- und Spezialisierungs-Schnittstelle

Da viele Klassen in einem Framework ausdrücklich entwickelt wurden, um spezialisiert zu werden, kann man bei ihnen entsprechende Schnittstellen-Teile unterscheiden: die Kunden-Schnittstelle und die Spezialisierungs-Schnittstelle (vgl. [Andert 95], [Cotter 95]).

Die *Kunden-Schnittstelle* (engl. client API) ist der Teil der Schnittstelle, der von anderen Klassen benutzt wird, um Dienste in Anspruch zu nehmen. Sie entspricht dem öffentlichen Teil der Schnittstelle, den alle Klassen bieten, die keine speziellen Vorkehrungen zu ihrer Spezialisierung treffen.

Die *Spezialisierungs-Schnittstelle* (engl. subclassing API, oder inheritance interface) hingegen besteht aus den Operationen, die zur Spezialisierung der Klasse vorgesehen sind. Dies sind die abstrakten Operationen, aber auch andere Operationen, die dynamisch gebunden sind und redefiniert werden können bzw. sollen.

Diese Klassifizierung ist jedoch nicht exklusiv: In der Praxis geschieht es oft, daß eine Operation sowohl zur Kunden-Schnittstelle als auch zur Spezialisierungs-Schnittstelle gehört, d.h. eine Methode spezialisiert wird, die von Kunden der Oberklasse direkt aufgerufen wird (vgl. [Cotter 95, S. 61]).

Beim Design von Framework-Klassen ist es einerseits wichtig, diese Aufteilung dem Anwendungsprogrammierer deutlich zu machen und andererseits keine zu restriktiven Einschränkungen bezüglich der Spezialisierung (subclassing) zu machen.

In C++ sollte die Unterscheidung zwischen Kunden- und Spezialisierungs-Schnittstelle u.a. durch die Zugriffsberechtigung (private / protected / public) deutlich gemacht werden (siehe Abbildung 7). Darüber hinaus gibt die dynamische bzw. statische Bindung einzelner Methoden einen eindeutigen Hinweis, wo Spezialisierungen vorgesehen sind. Da sich die beiden Schnitt-stellen jedoch überschneiden können, ist es wichtig, durch Kommentare und Dokumentation

die vorgesehene Verwendung der Methoden deutlich zu machen und sich nicht auf die Sprachmittel von C++ zu beschränken.

```
class ocSimpleMakeCallRequest
{
    // Kunden-Schnittstelle
    public:
    void vStatusUpdate();                   // statische Bindung - darf
                                            // nicht überladen werden

    // Spezialisierungs-Schnittstelle
    public:
    virtual void vCheckDevices();           // dynamische Bindung -
                                            // darf überladen werden

    // Spezialisierungs-Schnittstelle
    protected:
    virtual void vStartMakeCall() = 0;      // muß überladen werden

    // weder Kunden- noch Spezialisierungs-Schnittstelle
    private:
    void vInternalUpdate();
};
```

Abbildung 7: Kunden- und Spezialisierungs-Schnittstelle

Ein Black-Box-Framework wird daher eine sehr viel schmalere Spezialisierungs-Schnittstelle haben als ein White-Box-Framework. Eine Klasse mit schmaler Schnittstelle ist einfacher zu spezialisieren. Anderseits schränkt eine schmale Spezialisierungs-Schnittstelle die Umgangsmöglichkeiten mit unerwarteten Änderungen deutlich ein. Daher muß in jedem konkreten Anwendungskontext eine neue Abwägung zwischen Flexibilität und Einfachheit der Benutzung stattfinden. In [WeiGamMar 89] wird dies als das *Narrow Inheritance Interface Principle* bezeichnet. Wichtiger als eine schmale Spezialisierungs-Schnittstelle, die die Spezialisierungs-Möglichkeiten ausdrückt, ist es die Zahl der Operationen zu minimieren, die von einer Unterklasse überschrieben werden müssen. Eine breite Spezialisierungs-Schnittstelle, die zum größten Teil aus Hook-Operationen besteht, die nicht überladen werden müssen, ist nur wenig komplexer als eine Schnittstelle ohne die Hook-Operationen.

2.2.3.2 Trennung von Parametern und Einstellungen

Ein weiterer Gesichtspunkt bei der Gestaltung der Klassenschnittstelle ist die Kunden-Schnittstelle und hier besonders die Übergabe von Argumenten an die einzelnen Methoden: Je weniger Methoden aufgerufen werden müssen, um eine bestimmte Aufgabe zu erledigen, desto einfacher ist eine Klasse zu verwenden. Andererseits sind wenige Methodenaufrufe mit vielen Argumenten unübersichtlich und erfordern Änderungen bei allen Kunden der Klasse, falls die Argumentenlisten später verändert werden.

Um die Evolutionsfähigkeit einer Klasse zu gewährleisten, sollte unterschieden werden zwischen den *Parametern*, mit denen die eigentliche Verarbeitung durchgeführt wird, und den *Einstellungen*, die die Art und Weise der Verarbeitung beeinflussen. Während die Parameter erfahrungsgemäß nur selten variieren, werden die Einstellungen sehr oft nachträglich noch erweitert. Daher sollten alle Parameter zusammen einer Methode übergeben werden, nachdem

vorher die Einstellungen jeweils einzeln gesetzt worden sind. Dieses Vorgehen führt zu einer übersichtlichen Übergabe von Argumenten, da für Einstellungen häufig sinnvolle Default-Werte existieren. Es muß allerdings sichergestellt werden, daß alle notwendigen Einstellungen vorgenommen wurden, bevor die Verarbeitung beginnt. Dies kann z.B. durch ein Vertragsmodell mit Zusicherungen geprüft werden.

Während beispielsweise die anzurufende Telefonnummer ein unbedingt benötigter Parameter für einen Verbindungsaufbau ist, wäre die Wartezeit auf ein Freizeichen eher eine Einstellung, die dann auch getrennt übergeben werden sollte. Die beiden Varianten sind in Abbildung 8 einander gegenüber gestellt.

```
class ocConnectionMaker_1
{
public:
    virtual void vConnect(ocString oDestination, int iTimeout);
};


class ocConnectionMaker_2
{
public:
    virtual void vSetTimeout(int iDuration);
    virtual void vConnect(ocString oDestination);
};
```

Abbildung 8: Getrennte Übergabe von Parametern und Einstellungen

Mit dieser Aufteilung läßt sich die Funktionalität einer Klasse nachträglich verändern, und nur diejenigen Kunden müssen angepaßt werden, die diese neue Funktionalität auch tatsächlich nutzen (vgl. [Meyer 90]).

Diese Technik ist nicht an die Verwendung von objektorientierten Methoden gebunden und kann auch bei imperativen Sprachen mit Gewinn eingesetzt werden (vgl. Diskussion in [Meyer 82] über FORTRAN-Bibliotheken).

2.2.3.3 Dynamische Bindung

Für die lose Kopplung von Klassen ist es oft wichtig, Objekte polymorph nur unter ihrer Oberklasse zu verwenden. Dieses Vorgehen macht ein dynamisches Binden erforderlich, da die passende Methode der Unterklasse erst zur Laufzeit bestimmt werden kann.

Die redefinierten Methoden müssen jedoch die Semantik der Methode der Oberklasse einhalten und dürfen auch den Kontrollfluß innerhalb des Frameworks nicht stören. Daher sollte explizit dokumentiert werden, welche Methoden der Framework-Klassen von einer redefinierten Methode weiterhin aufgerufen werden müssen, um den Kontrollfluß des Frameworks nicht zu unterbrechen. Die Einhaltung dieser Bedingung muß ggf. durch Zusicherungen überprüft werden.

Eine dynamische Bindung von Methoden findet in C++ jedoch nicht automatisch statt. Der Programmierer muß dynamisch zu bindende Methoden ausdrücklich als *virtual* kennzeichnen.

Dies wirft die Frage auf, welche Methoden einer Klasse als virtual zu deklarieren sind und welche nicht.

Eine Klasse mit virtuellen Funktionen wird durch die dynamische Bindung anpaßbar, da man Unterklassen von ihr bilden kann, die dann polymorph verwendet werden können. Bei statisch gebundenen Methoden ist dies nicht möglich, da zur Laufzeit die Methode der Oberklasse ausgeführt werden würde.

Da der Framework-Entwickler nur Vermutungen über die zukünftige Verwendung seines Frameworks haben kann, sollte er nicht unnötig Anpassungen seiner Framework-Klassen unterbinden. Damit ein Anwendungsprogrammierer die Möglichkeit hat, die Methode der Framework-Klassen zu spezialisieren, sollten diese grundsätzlich als "virtual" deklariert sein, wenn es keine zwingenden Gründe gibt, warum diese Methode auf keinen Fall überladen werden darf. Dies ist bei Frameworks insbesondere deshalb wichtig, da die Anwendungsprogrammierer oft keinen Zugriff auf den Quelltext des eingesetzten Frameworks haben und daher eine solche Änderung meist nicht nachträglich durchführen können.

Da erst zur Laufzeit entschieden werden kann, welche Methode aufgerufen werden soll, ist für den Methodenaufruf ein gewisser Overhead notwendig. Anstatt direkt auf eine Methode zu zeigen, verweist der Zeiger auf eine dynamisch gebundene Methode, auf die sog. *virtual table* (oder kurz VTable), in der die passende Methode gesucht wird.

Der hierdurch entstehende Performance-Verlust sollte jedoch nicht überbewertet werden, da es sich um wenige Zeiger-Dereferenzierungen handelt, die nur einen kleinen Teil der insgesamt ausgeführten Operationen ausmachen. In [DriHölz 96] wird das Ergebnis diverser Simulationen unter Berücksichtigung verschiedener Prozessor-Architekturen vorgestellt. Der Performance-Verlust für die Verwendung von virtuellen Methoden im Vergleich zu direkten Funktionsaufrufen wird dort mit durchschnittlich 5-10% quantifiziert. Das Fazit der Autoren ist:

> "Given that better optimizing compilers are possible, it hardly seems appropriate for programmers to compromise the structure of their programs to avoid dispatch."

Auch in [Stroustrup 94b, S. 147] wird argumentiert, daß der durch dynamische Bindung entstehende Mehraufwand zu einem guten Teil wegoptimiert werden könne, sobald fortgeschrittenere C++ Compiler vorliegen. Einige alternative Optimierungstechniken zur Verwendung in Compilern werden in [BaconSweeney 96] diskutiert.

Ein Abweichen von diesem Vorgehen sollte erst erwogen werden, wenn tatsächlich Performance-Probleme auftreten und diese durch Profiling auf das dynamische Binden zurückgeführt werden können.[3]

Es wird deutlich, daß die Ausrichtung auf maximale Performanz beim Design der Sprache C++ in diesem Bereich zu Voreinstellungen für die Bindung von Methoden geführt hat, die einer leichten Evolution zuwiderlaufen. Eine dynamische Bindung als Voreinstellung und das explizite Anfordern von statischer Bindung wäre meiner Ansicht nach sinnvoller gewesen.

[3] "Premature optimization is the root of all evil." Donald Knuth (zitiert nach [McConnell 93], S. 681)

Mit der vorgeschlagenen Vorgehensweise alle Methoden dynamisch zu binden, solange kein gegenteiliger Bedarf besteht, wird auch einer speziellen Art von Speicherlecks in C++ vorgebeugt, die sonst recht schwer zu finden ist: Wenn Oberklassen von Klassen mit virtuellen Funktionen keinen virtuellen Destruktor haben, werden beim Freigeben des Speichers durch den delete-Operator nicht immer alle Destruktoren aufgerufen [Stroustrup 92, S. 230f].

Da zum Zeitpunkt der Entwicklung einer Oberklasse kaum bekannt ist, ob es einmal eine Unterklasse geben wird, die dynamisch gebundene Methoden benötigen wird, ist dieses Problem nur zu lösen, indem alle Klassen mit virtuellen Methoden auch einen virtuellen Destruktor haben. Ansonsten würde man schon aus technischen Gesichtspunkten das Spezialisieren dieser Klassen unmöglich machen. Der virtuelle Destruktor kann ohne weiteres leer sein, so daß ein guter Compiler ihn ggf. auch wegoptimieren kann.

2.3 Kategorien von Frameworks

Ein Framework stellt das Design und eine (Teil-)Implementation für ein spezielles Gebiet zur Verfügung. Anhand des modellierten Gebiets kann man Frameworks in Kategorien unterteilen. Für das jeweilige Gebiet diktieren die Frameworks das Design der Anwendung.

Definition 6: Application-Framework

Ein Framework, das zum Ziel hat, ein ablauffähiges Programm zu erzeugen, wird *Application-Framework* genannt. Es enthält das Design und den Kontrollfluß für eine vollständig lauffähige Anwendung.

Häufig anzutreffen sind die *GUI-Frameworks*, die das Look-and-Feel einer graphischen Oberfläche implementieren und den gesamten Anwendungskontrollfluß (und meist auch ein leeres Fenster und Standardmenü) enthalten. Da das Erstellen von GUI-Anwendungen recht aufwendig ist, stellen die GUI-Frameworks den größten Teil der heute kommerziell verfügbaren Frameworks dar. Oft sind GUI-Frameworks gleichzeitig auch Application-Frameworks, da sie Zugriff auf die Ereignisschleife benötigen.

ET++ ist ein bekanntes Application-Framework für C++ (vgl. [Gamma 92], [WeiGamMar 89]). Es diente als Grundlage für GUI-Anwendungen wie z.B. die C++-Entwicklungsumgebung SNiFF+ (siehe [Bischofberger 92], [Pfeiffer 97]), sowie auch als Basis für spezialisiertere Frameworks.

Definition 7: GUI-Framework

Ein Framework, das die graphische Interaktion eines Benutzers mit einer Anwendung modelliert, wird als GUI-Framework bezeichnet.

Ein Beispiel für ein GUI-Framework ist das MFC-Framework. Zur Erleichterung der Programmierung von MS-Windows hat Microsoft 1991 mit der Entwicklung eines Frameworks zur Erstellung von Windows-Anwendungen begonnen (vgl. [Wingo 95]). Das MFC-Framework wurde in C++ realisiert und ist heute Basis eines Großteils der aktuellen Windows-Anwendungen.

Application-Frameworks müssen aber keinesfalls immer GUI-Frameworks sein. Es gibt auch diverse Kategorien von Anwendungen ohne GUI, für die ein Application-Framework erstellt

werden kann. Das Micrologica Telefonie-Framework (siehe Kapitel 4) ist ein Beispiel dafür. Es definiert den gesamten Kontrollfluß, der nötig ist, um die Treibersoftware für eine Telekommunikationsanlage zu erstellen.

Frameworks, die kein Design für eine ablauffähige Anwendung enthalten, können weiter unterschieden werden in *fachliche Frameworks* und *softwaretechnische Frameworks. Fachliche Frameworks* geben ein Design für ein bestimmtes Feld aus der Anwendungswelt vor, während *softwaretechnische Frameworks* von dem Anwendungsgebiet unabhängige Basismechanismen bereitstellen, die in diversen unterschiedlichen Kontexten eingesetzt werden können (z.B. die Implementierung eines Meta-Object-Protocol).

> **Definition 8: Fachliches Framework**
>
> Ein Framework, das einen (Teil-)Bereich der fachlichen Anwendung modelliert, bezeichnet man als fachliches Framework.
>
> **Definition 9: Softwaretechnisches Framework**
>
> Ein Framework, das eine (implementations-)technisch notwendige Rahmenarchitektur zur Verfügung stellt, bezeichnet man als softwaretechnisches Framework.

In der Praxis werden oft diverse Bereiche innerhalb eines Frameworks unterstützt. So enthalten beispielsweise fast alle fachlichen Frameworks ein gewisses Maß an softwaretechnischer Unterstützung, die auch den Anwendungen zur Verfügung gestellt wird.

Ein Beispiel für die Kombinierbarkeit von Frameworks ist das Multimedia-Framework MET++, das auf der Basis des GUI-/Application-Frameworks ET++ entstanden ist (siehe [Ackermann 96]).

Eine andere Kategorisierung von Frameworks orientiert sich an der Art, in der sie zur Erstellung konkreter Anwendung verwendet werden:

Bei einem *White-Box-Framework* ist die Art der Spezialisierung kaum vorgegeben. Der Anwendungsprogrammierer kann die Klasse bei Bedarf durch Vererbung spezialisieren, muß aber beim Redefinieren darauf achten, nicht den Kontrollfluß innerhalb des Frameworks zu stören. Die redefinierte Methode muß ggf. die ursprüngliche Methode der Oberklasse aufrufen, damit das Framework seine Arbeit erledigen kann. Die "gemeinsamen Invarianten" [Johnson 93], also das implizite Kommunikationsprotokoll innerhalb der Klassen, muß gewahrt bleiben. Daher ist es für die Spezialisierung der Klassen eines White-Box-Frameworks unerläßlich, sich über die interne Realisierung der Framework-Klassen bewußt zu sein. Daher der Name White-Box-Framework.

Zur der Verwendung eines *Black-Box-Frameworks* ist weit weniger Wissen über den internen Aufbau des Frameworks notwendig, da die Art der Spezialisierung der Framework-Klassen z.B. durch Parameter-Objekte schon vorgegeben ist. Es ist offensichtlich, daß ein Black-Box-Framework daher einfacher zu benutzen ist.

Ein Black-Box-Framework kann eine Sammlung von austauschbaren Teilimplementierungen zur Verfügung stellen, die zur Parametrisierung verwendet werden können. Im Idealfall werden Anwendungen nur durch die Komposition von existierenden Klassen erstellt. Dies ist jedoch

eine idealisierte Vorstellung, da ein Framework-Entwickler nie alle Anwendungsfälle seines Frameworks vorhersehen wird, so daß der Anwender meist doch einige Klassen selbst erstellen muß.

Die zur Spezialisierung notwendigen Parameter-Objekte können entweder direkt von Framework-Klassen erzeugt werden, oder der Anwendungsprogrammierer muß zunächst eine Klasse spezialisieren, von der er dann die Parameter-Objekte erzeugt. Die Auffassungen in der Literatur gehen auseinander, ob ein Framework, das keine oder nur wenige fertige Klassen für Parameter-Objekte enthält, bereits ein Black-Box-Framework darstellt. Ich bin der Meinung, daß man es als solches bezeichnen sollte, wenn zur Erstellung der Klassen für Parameter-Objekte kein oder nur sehr wenig Wissen über den internen Aufbau des Frameworks notwendig ist.

Man könnte sogar sagen, daß die Unterscheidung zwischen Black-Box und White-Box letztlich durch den Anwendungs-Programmierer getroffen wird. Denn er kann ein Black-Box-Framework auch an nicht zur Spezialisierung vorgesehenen Stellen in der Art eines White-Box-Frameworks spezialisieren, sofern ihm die interne Arbeitsweise des Frameworks hinreichend bekannt ist.

Definition 10: White-Box-Framework

Ein Framework, zu dessen Verwendung Wissen über den internen Aufbau notwendig ist, wird als White-Box-Framework bezeichnet.

Definition 11: Black-Box-Framework

Ein Framework, zu dessen Verwendung kein bzw. sehr wenig Wissen über den internen Aufbau notwendig ist, wird als Black-Box-Framework bezeichnet.

Mir ist es wichtig, mit den Begriffen Black-Box und White-Box zu charakterisieren, inwieweit zur Anwendung des Frameworks ein Wissen über den internen Aufbau notwendig ist - unabhängig davon, wie dies realisiert ist. Um die Realisierung der Spezialisierung von Framework-Klassen zu klassifizieren, erscheint mir die Begriffsbildung aus [Taligent 95] geeignet. Dort wird zwischen vererbungsbasierten (engl. inheritance based) und kompositionsbasierten (engl. composition based) Frameworks unterschieden. In den meisten Fällen sind White-Box-Frameworks vererbungsbasiert und Black-Box-Frameworks kompositionsbasiert, aber man sollte diese Kategorien nicht miteinander gleichsetzen, da die eine die Art der Verwendung und die andere die Art der technischen Realisierung beschreibt.

Definition 12: Vererbungsbasiertes Framework

Ein Framework, das zur Spezialisierung durch Vererbung gedacht ist, wird als vererbungsbasiertes Framework bezeichnet.

Definition 13: Kompositionsbasiertes Framework

Ein Framework, das zur Spezialisierung durch Parametrisierung gedacht ist, wird als kompositionsbasiertes Framework bezeichnet.

In Kapitel 2.6.3 (Abbildung 10) ist beispielhaft die Implementierung einer Dienstanforderung an eine Tk-Anlage mit vererbungsbasierter und mit kompositionsbasierter Spezialisierung

gegenübergestellt. In Kapitel 2.6.3 diskutiere ich auch, warum Black-Box-Frameworks sehr häufig aus intensiv eingesetzten White-Box-Framework, erstellt werden.

Es ist wichtig zu betonen, daß die Begriffe White-Box- und Black-Box-Framework zwei Extreme darstellen. In der Praxis werden Frameworks oft teilweise zur White-Box- und teilweise zur Black-Box-Verwendung geeignet sein. Auch umfassen Frameworks oft sowohl vererbungsbasierte als auch kompositionsbasierte Teile.

2.4 Die Beziehung zwischen Entwurfsmustern und Frameworks

Frameworks werden oft mit Hilfe von Entwurfsmustern (engl. design pattern) entwickelt bzw. dokumentiert, da beide Ansätze ähnliche Intentionen haben: Sie drücken gemachte Erfahrungen aus und vergegenständlichen sie. Dadurch transportieren beide Ansätze eine Design-Idee. Frameworks enthalten zusätzlich jedoch auch eine Implementation dieser Design-Idee.

Beide Ansätze unterstützen auch die Adaptierbarkeit von Designs: Entwurfsmuster werden eingesetzt, damit sich bestimmte Teile eines Designs unabhängig von anderen Bereichen entwickeln können ("Design for Change"). Dies ist eine der Grundvoraussetzungen für ein robustes Design und damit für die Wiederverwendbarkeit, da sich mit hoher Wahrscheinlichkeit die Anforderungen zumindest in Teilbereichen verschieben werden. Die Adaptierbarkeit ist auch ein wichtiges Problem bei der Framework-Entwicklung zur Realisierung der hot spots. Abbildung 9 listet einige bekannte Entwurfsmuster auf, auf die ein Framework-Entwickler zurückgreifen kann (vgl. [GHJV 95, S. 30], [MätBisch 96], [RobJohn 96]). Im Unterschied zur üblichen Darstellung in Entwurfsmuster-Katalogen sind die Muster hier anhand der veränderlichen Eigenschaft kategorisiert.

Der Einsatz von Entwurfsmustern in Frameworks hat den zusätzlichen Vorteil, daß das Design leichter kommunizierbar ist, da die Entwurfsmuster ein Vokabular zur Beschreibung der zugrundeliegenden Design-Ideen zur Verfügung stellen. Das Framework wird dadurch leichter erlernbar.[4]

Definition 14: Entwurfsmuster

Ein Entwurfsmuster vermittelt die Lösung eines Design-Problems in einem speziellen Kontext. Es erklärt die Design-Idee, ist aber nicht an eine konkrete Implementation gebunden.

Es ist auch denkbar, Frameworks zu erstellen, die Implementationen für diverse Entwurfsmuster zur Verfügung stellen. Hierbei ist jedoch zu bedenken, daß die z.B. in [GHJV 95] angegebenen Beispiel-Implementationen keineswegs überall eingesetzt werden können (vgl. Kapitel 5.3.2) und der Transport der Design-Idee bei Entwurfsmustern klar im Vordergrund stehen sollte.

[4] Hierbei liegt die Annahme zugrunde, daß die zukünftigen Framework-Benutzer mit den wichtigsten Entwurfsmustern vertraut sind.

Veränderliche Eigenschaft	**Entwurfsmuster**
Algorithmus	Strategy, Visitor
Traversions-Algorithmus	Iterator
Schritte eines Algorithmus	Template-Method
Aktionen	Command
Reaktion auf Änderungen	Beobachter
Interaktion zwischen Objekten	Mediator
Erzeugte Objekte	Fabrik-Methode, abstrakte Fabrik, Prototype, Builder
Systemkonfiguration	abstrakte Fabrik
Implementationen	Brücke
Objekt-Schnittstelle	Adapter
Objekt-Verhalten	Decorator, State
Realisierung eines Subsystems	Facade

Abbildung 9: Lose Kopplung durch Entwurfsmuster

2.5 Vorgehensweise bei der Framework-Entwicklung

Nach Definition 2 stellt ein Framework ein generisches Design für Anwendungen zur Verfügung. Dieses Design muß die Abstraktionen des Anwendungsgebietes modellieren, und die hot spots müssen hinreichend flexibel spezialisiert werden können. Um ein Framework entsprechend konstruieren zu können, ist sehr viel Anwendungserfahrung notwendig. Bei der Framework-Entwicklung ist es daher wichtig, zunächst mehrere Anwendungen zu erstellen, damit genügend Anwendungserfahrung gesammelt werden kann.

Auf der Basis der im jeweiligen Anwendungsgebiet gesammelten Erfahrung können dann die gemeinsamen Abstraktionen gesucht und analysiert werden. Aus ihnen muß sich die Struktur des zu erstellenden Frameworks ergeben.

Aus den erstellten Anwendungen können dann die als wiederverwendbar erkannten Teile in ein Framework übernommen werden. Es ist hilfreich, aber nicht zwingend notwendig, daß die Anwendungen objektorientiert erstellt wurden. Hauptsächlich kommt es darauf an, daß allgemein verwendbare Abstraktionen gefunden werden und die hot spots identifiziert werden können. Daher ist die Beteiligung von Fachleuten des Anwendungsbereiches bei der Erstellung dieser Anwendungen extrem wichtig. (Dies ist eigentlich bei jeder Anwendungsentwicklung wichtig. Fehler in der fachlichen Modellierung wiegen bei der Framework-Entwicklung jedoch ungleich schwerer.)

Um Redundanzen zu vermeiden, sollte jedes fachliche Konzept im Framework genau einmal repräsentiert sein. Bei der Konstruktion des Frameworks sollte daher der Schwerpunkt auf einem fachlich sauberen Design liegen und nicht unbedingt auf maximaler Wiederverwendung

der vorliegenden Implementationen. Es muß bedacht werden, daß das Framework das Design aller noch mit ihm zu erstellenden Anwendungen maßgeblich beeinflußt.

Die Framework-Entwicklung sollte nicht auf dem kritischen Pfad einer Projektentwicklung stattfinden, um einen zu großen Zeitdruck zu vermeiden. Eine tragfähige Grundstruktur ist wichtig, um spätere Änderungen am Framework soweit wie möglich zu vermeiden. Denn sobald Anwendungen auf dem Framework basieren, sind sie potentiell von Änderungen in der Framework-Struktur betroffen. Je früher mit dem Framework Anwendungen erstellt werden, desto größer ist die Gefahr, ihnen das Design eines noch nicht ausgereiften Frameworks aufzuzwingen. Daher sollte die Framework-Entwicklung nicht in Projekten stattfinden, die einem hohen externen Zeitdruck unterliegen.

Ein Beispiel für die Folgen einer Framework-Entwicklung ohne konkrete Anwendungen berichtet Scot Wingo in [Wingo 95]: Die erste (unveröffentlichte) Version der Microsoft Foundation Classes wurde "am grünen Tisch" entwickelt mit dem Ziel, eine möglichst weitgehende Abstraktion vom Windows-API zu erreichen. Nachdem die Entwickler aber massive Probleme hatten, mit dem resultierenden Framework Anwendungen zu erstellen, wurde das Framework völlig neu entwickelt und nur eine dünne Schicht an Abstraktionen über das Windows-API gelegt. Diese Version wurde als MFC 1.0 veröffentlicht, und erst später wurden schrittweise höhere Abstraktionen eingeführt, nachdem man sie zuvor in Anwendungen erprobt hatte.

2.6 Framework-Evolution

Frameworks werden erstellt, um eine ganze Reihe von Anwendungen auf ihnen aufzubauen. Diese Anwendungen benötigen eine stabile Basis. Brown und Forte folgern daher in [BrownForte 96] :

> "The success of a framework depends totally on the reusability and flexibility of the objects that it [the framework] encapsulates. When building a framework, the public interface of classes within the framework must be well defined and virtually static to ensure that changes will not adversely affect systems that have been built using the framework. This means that an object designer needs to consider all the future possible uses of the object."

Ich möchte darlegen, warum dieses Vorhaben auf lange Sicht zum Scheitern verurteilt ist: Kaum ein Softwareprodukt wird einmal geradlinig entwickelt und dann nie mehr verändert. Die Veränderung im Laufe der Zeit, die Evolution, ist auch bei Frameworks eher die Regel als die Ausnahme.

Wie bereits in der Einleitung erwähnt, verwende ich den Begriff Evolution im Sinne einer "langsam, kontinuierlich fortschreitenden Entwicklung" [Brockhaus 88, S. 688]. In dieser Bedeutung möchte ich den Begriff auf die Fortentwicklung von Frameworks über ihre initiale Fertigstellung hinaus übertragen. Lange Zeit war der Evolutionsbegriff auf die biologische Evolution beschränkt. Heute wird er im übertragenen Sinn auch in vielen nicht-biologischen Disziplinen verwendet. Die Merkmale der biologischen Evolution treffen dabei häufig nur teilweise bzw. in einem metaphorischen Sinn zu (vgl. [BI 80, S. 610]). Auf die speziellere

Bedeutung des Evolutionsbegriffs in der Biologie (aber auch in der Kosmogonie[5] und Philosophie) gehe ich daher nicht weiter ein.

Definition 15: Framework-Evolution

Die kontinuierliche Veränderung eines Frameworks im Laufe seines gesamten Lebenszyklus wird als Framework-Evolution bezeichnet.

Da die Framework-Evolution (wie im folgenden gezeigt wird) in sachlichen Zwängen begründet ist, kann man sich ihr als Software-Entwickler nicht widersetzen: Man muß ihr Wesen kennen und kann präventive Maßnahmen ergreifen, aber die Zukunft bleibt etwas nicht Vorhersehbares, so daß man auch immer wieder mit unerwarteten Veränderungen umgehen muß.

Ich möchte in diesem Abschnitt darlegen, warum gerade Frameworks einem ständigen Drang zur Veränderung ausgesetzt sind und warum die Auswirkungen von Veränderungen in einem Framework ungleich weitreichender sind als in anderen Software-Produkten.

Die Ausführungen sollen als Motivation dienen für die Diskussion im nächsten Kapitel über präventive Maßnahmen, um die Auswirkungen von Änderungen bereits frühzeitig in den Entwicklungszyklus einzubeziehen.

Zunächst möchte ich der Frage nachgehen, warum Frameworks überhaupt einer Evolution unterworfen sind.

2.6.1 Gründe für Framework-Evolution

In Kapitel 2.5 habe ich dargelegt, daß ein Framework aus Anwendungen entwickelt werden muß und nicht "am grünen Tisch" entworfen werden kann. Daher liegen ihm mehr Erfahrungen zugrunde als einer einzelnen Anwendung. Man könnte meinen, daß Frameworks dadurch eine gewisse Robustheit gegenüber Änderungen besitzen. Dieser Effekt wird jedoch kompensiert durch das Änderungspotential der Summe der auf dem Framework basierenden Anwendungen. Sobald die Anwendungen, aus denen das Framework abstrahiert wurde, auf die Benutzung des Frameworks umgestellt sind und sich dann weiter entwickeln, multipliziert sich auch das Potential für Änderungen im Framework. Diese Änderungen können dann entweder innerhalb der Anwendungen abgehandelt oder in das Framework integriert werden.

Je weniger Anwendungen man als Basis für die Framework-Entwicklung einsetzt, um die Änderungen in der frühen Entwicklungsphase zu reduzieren, desto größer wird das Potential für gravierende Änderungen im weiteren Lebenszyklus des Frameworks.

Aber auch nach der initialen Entwicklung gibt es einen starken Druck für Veränderungen in Frameworks:

Da ein Framework gemachte Erfahrung verkörpert und die gefundenen Lösungen für alle Anwendungen einer Domäne verfügbar machen soll, besteht der Drang, später gemachte Erfahrungen, z.T. erst durch den Einsatz des Frameworks machbare Erfahrungen, ebenfalls in

---

[5] Lehre von der Entstehung der Himmelskörper (siehe [Brockhaus 88])

das Framework zu integrieren. Dies ist die gleiche Motivation, die ursprünglich zur Entwicklung der ersten Framework-Version geführt hat.

Auch wenn man sich entschließt, die Funktionalität eines Frameworks nicht zu erweitern bzw. zu verändern, so müssen immer noch Fehlerkorrekturen durchgeführt haben, die ebenfalls weitreichende Auswirkungen auf das Framework und seine Struktur haben können.

2.6.2 Gerichtete und ungerichtete Evolution

Die Evolution eines Frameworks hängt maßgeblich davon ab, ob und wie sich die Anforderungen an dieses Framework ändern. Die veränderten Anforderungen bestimmen die Richtung der Evolution. Die Anpaßbarkeit an die neuen Anforderungen bestimmt, ob das Framework mit der Entwicklung schritthalten kann.

Es sind sowohl interne als auch externe Faktoren, die zu veränderten Anforderungen führen. Eine wichtige Quelle ist z.B. der Lernprozeß aller an der Entwicklung beteiligten Personen. Dieser Lernprozeß endet nicht nach der Erstellung der ersten Beispielanwendungen, sondern er geht weiter und begleitet den gesamten Lebenszyklus des Frameworks.

Einige der zu erwartenden bzw. sich verändernden Anforderungen lassen sich bei einer gründlichen Analyse antizipieren und in das Design des Frameworks einbeziehen. Diese erwarteten Änderungen führen zu einer sog. gerichteten Evolution (engl. directed evolution) [MätBisch 96]. Ein Beispiel aus dem Telefonie-Framework ist die Erwartung, daß ständig neue Tk-Anlagen auf den Markt kommen werden, über deren Fähigkeiten jetzt noch keine Aussage möglich ist. Problematischer sind jedoch die unerwarteten Änderungen. Sie führen zu einer sog. ungerichteten Evolution (engl. undirected evolution).

Definition 16: Gerichtete Evolution

Veränderungen, die an erwarteten Stellen bzw. aus erwarteten Gründen stattfinden, werden als gerichtete Evolution bezeichnet.

Definition 17: Ungerichtete Evolution

Veränderungen, die an unerwarteten Stellen stattfinden, werden als ungerichtete Evolution bezeichnet.

Der Umfang der ungerichteten Evolution hängt ganz entscheidend davon ab, wieweit sich ein Framework für Gebiete "zuständig fühlt", für die es nicht entworfen wurde. Wenn der Bereich, den ein Framework abdeckt, nur schwammig definiert ist, kann es leicht passieren, daß neue Anforderungen entstehen, die bei der ursprünglichen Entwicklung nicht berücksichtigt wurden, während ein Framework mit einem klar umrissenen Zuständigkeitsbereich viel leichter Anforderungen als "nicht in seinem Zuständigkeitsbereich" verweigern kann.

Dies ist jedoch ein Gratwanderung: Eine extreme Einschränkung bedeutet, nur Unterstützung für eine einzige Anwendung zu liefern. Diese Unterstützung kann dann sehr spezifisch ausfallen und ist kaum Änderungsanforderungen ausgesetzt, die unerfüllbar sind. Dies ist jedoch nicht die Intention von Frameworks. Frameworks sollen immer eine Gruppe von Anwendungen unterstützen. Dies darf jedoch nicht dazu führen, daß das Framework für "alles und nichts" zuständig ist. Der Preis für eine zu weit gefaßte Zuständigkeit des Frameworks ist eine ständig

variierende Basis für alle auf dem Framework basierenden Anwendungen, da das Framework ständig mit unerwarteten Änderung zu kämpfen hat und nur eine sehr generische Unterstützung für jede spezielle Anwendung geboten werden kann.

2.6.3 Typische Merkmale der Framework-Evolution

Es hat sich gezeigt, daß die meisten Frameworks einige typische Merkmale bei ihrer Evolution gemeinsam haben.

Üblicherweise ist die erste Version eines Frameworks vererbungsbasiert (vgl. [JohnFoote 88], [RobJohn 96]). Dies liegt darin begründet, daß die Framework-Klassen aus Anwendungsklassen hervorgegangen sind, aus denen das abstrakte Konzept extrahiert wurde. Die speziellen Teile sind jetzt in Unterklassen zu finden.

Gerade in der frühen Framework-Entwicklung sind die gefundenen Abstraktionen noch nicht so allgemein, daß man kaum Änderungen an ihnen vornehmen muß, um neue Anwendungen zu erstellen. Daher ist meistens eine Spezialisierung per Vererbung nötig, um umfassende Änderungen an der Framework-Klasse vorzunehmen. Dies führt zu White-Box-Frameworks.

Durch den mehrfachen Einsatz des Frameworks wächst der Erfahrungsschatz, um robustere Abstraktionen zur Verfügung zu stellen. Dadurch wird eine schrittweise Entwicklung des Frameworks zu einem Black-Box-Framework möglich. Sobald die typischen Modifikationen bekannt sind, die von den Anwendungen benötigt werden, können diese in den Framework-Klassen z.B. durch Parametrisierung vorgesehen werden. Abbildung 10 zeigt beispielhaft die Umformung einer Klasse von vererbungsbasierter Spezialisierung zur kompositionsbasierten Spezialisierung.

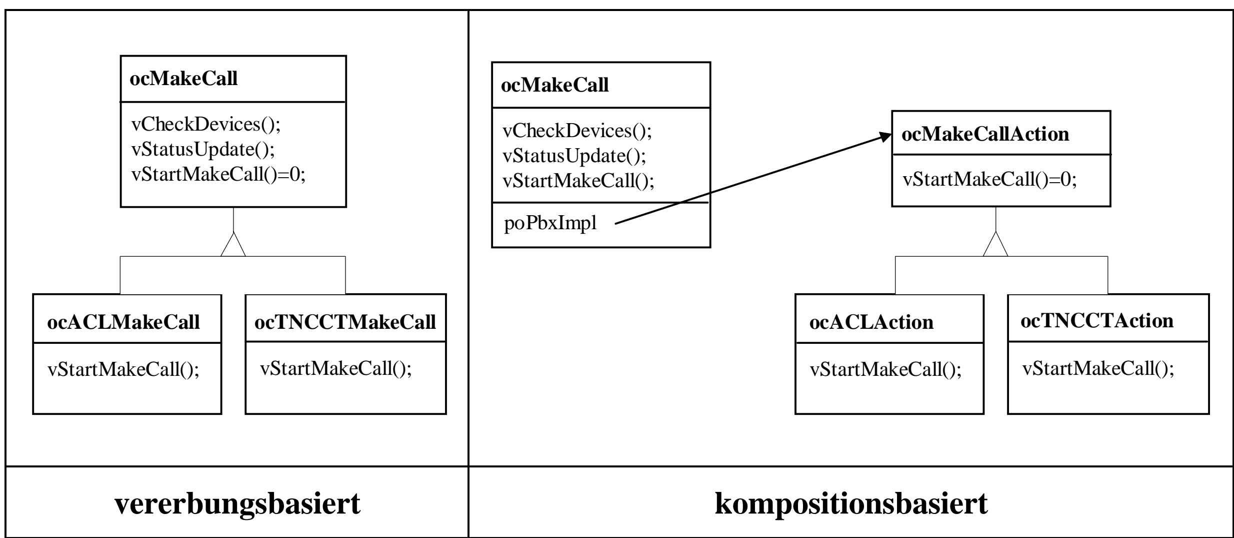

Abbildung 10: Vergleich der Spezialisierungs-Techniken

Zum einen erleichtern Black-Box-Frameworks die Arbeit des Anwendungsprogrammierers, und zum anderen geben sie die Möglichkeit, eine losere Kopplung zwischen Framework-Klassen und Anwendungs-Klassen zu erreichen. Die Erstellung eines Black-Box-Frameworks

setzt allerdings ein viel umfassenderes Wissen über den Anwendungsbereich und mehr Erfahrung beim Framework-Entwickler voraus, die erst im Laufe eines Lernprozesses erreicht werden kann.

Die Oberklasse von Parameter-Objekten ist meist deutlich einfacher als die von White-Box-Klassen, die spezialisiert werden müssen. Die Parameter-Klassen müssen eine viel schlankere Schnittstelle implementieren, so daß es dem Anwendungsprogrammierer leichter fallen wird, neue Parameter-Klassen zu programmieren als die Spezialisierung einer White-Box-Klasse zu erstellen.

Je mehr Anwender das Framework verwenden, desto schwieriger ist meist die Kommunikation der Anwender mit den Framework-Entwicklern. Wenn ein Framework die Entwickler-Organisation verläßt, verschlechtert sich die Kommunikation meist weiter. Bei entsprechend weiter Verbreitung lohnt es sich, einen größeren Aufwand innerhalb des Frameworks zu betreiben, um unsachgemäße Benutzung zu verhindern. Diese Möglichkeit ist bei Black-Box-Frameworks leichter gegeben, da genauer bekannt ist, an welchen Stellen Anwendungs-operationen aufgerufen werden. Vor und nach diesen Operationen kann dann zur Laufzeit (in Grenzen) geprüft werden, ob diese alle ihre Pflichten erfüllt haben. Auch das Einhalten von Reihenfolge-Bedingungen etc. wird bei Black-Box-Frameworks in das Framework verlagert und dem Anwendungsprogrammierer abgenommen.

Ein Anzeichen dafür, daß adäquate Abstraktionen gefunden sind, ist *das Zurückgehen der Quelltext-Größe*, da mehr Code wiederverwendet werden kann. Das Zurückgehen der Quell-text-Größe ist eine typische Erscheinung eines "gereiften" Frameworks, sie sollte jedoch nicht durch maximale Code-Wiederverwendung erreicht werden. Wichtiger ist, daß die fachlichen Abstraktionen stimmig sind und möglichst wenig Redundanz der fachlichen Konzepte existiert. Dies geht dann normalerweise auch mit weniger Redundanz im Code einher.

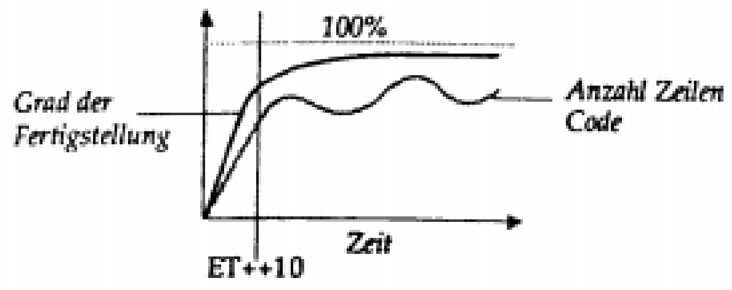

Abbildung 11: Entwicklung der Quelltext-Größe bei ET++ [Gamma 92, S. 161]

Als Beispiel für die zurückgehende Quelltext-Größe sei hier das ET++ Framework angeführt (siehe Abbildung 11). Im Micrologica-Telefonie-Framework ist dieses Phänomen bisher nicht so deutlich zu erkennen gewesen, da es zum einen mit Anforderungen nach zusätzlicher Funktionalität (z.B. Event-Normalisierung) überlagert wurde und da zum anderen das Framework noch nicht lange genug im praktischen Einsatz ist.

Bei Verfügbarkeit einer großen Anzahl von fertigen Parameter-Objekten für ein Black-Box-Framework und einem weitreichenden Einsatz des Frameworks kann an die Werkzeug-unterstützung beim Umgang mit dem Framework gedacht werden. In diesem Stadium wären

graphische Werkzeuge zur Komposition von Anwendungen aus vorgefertigten Teilen denkbar (vgl. [RobJohn 96]). In diesem Endstadium wäre dann keine Programmierung mehr notwendig, sondern nur noch die Konfiguration fertiger Komponenten. Dies wird jedoch nur für sehr wenige, eng umgrenzte Bereiche möglich sein.

Es ist wichtig festzustellen, daß keineswegs alle Frameworks alle diese Stadien durchlaufen. Oft verbleiben sie in Zyklen auf dem Weg zwischen White-Box- und Black-Box-Framework. Aber die generelle Tendenz ist bei fast allen Frameworks zu beobachten. Damit ist auch keinerlei Wertung verbunden - bei vielen Frameworks macht es keinen Sinn, sie zu Black-Box-Frameworks auszubauen, weil sich z.B. die Anforderungen zu schnell ändern oder die zu erstellenden Anwendungen nicht homogen genug sind.

2.6.4 Auswirkungen der Framework-Evolution

Im Gegensatz zur Evolution von Anwendungen, die nur die jeweilige Anwendung betrifft, sind Frameworks explizit dazu gedacht, in einer ganzen Reihe von Anwendungen verwendet zu werden. Daher werden auch viele Anwendungen potentiell von der Evolution des Frameworks tangiert.

Zum einen hat dies den positiven Effekt, daß z.B. Fehlerkorrekturen im Framework automatisch in alle betroffenen Anwendungen übernommen werden, sobald eine korrigierte Version des Frameworks zur Verfügung steht. Zum anderen darf aber nicht übersehen werden, daß das Framework das Design aller auf ihm basierenden Anwendungen bestimmt und somit alle Anwendungen den Änderungen im Framework folgen müssen.

Gamma et. al. drücken es in [GHJV 95, S. 27] sehr präzise aus, wenn sie schreiben

> "..., because applications are so dependent on the framework for their design, they are particularly sensitive to changes in framework interfaces. As a framework evolves, applications have to evolve with it."

Wenn nun im Framework Änderungen am Design vorgenommen werden, kann dies im ungünstigsten Fall bedeuten, daß in allen abgeleiteten Anwendungen ein massiver Änderungsaufwand entsteht. Es kann eine Art "Domino-Effekt" entstehen, bei dem sich Veränderungen tief in die mit dem Framework erstellten Anwendungen hinein ziehen, wenn sich z.B. die im Framework definierten Kommunikationsprotokolle ändern.

Andererseits sind auch alle mit dem Framework erstellten Anwendungen einer Evolution unterworfen. Dabei entstehen oft auch neue Anforderungen an das Framework. Da das Framework den Anforderungen einer Reihe von Anwendungen gerecht werden muß und oft zeitlich lange vor den Anwendungen erstellt wird, kann der Fall eintreten, daß das Framework den neuen Anforderungen nicht ohne massive Änderungen gerecht werden kann.

Aufgrund der Tatsache, daß sich Änderungen im Framework nicht nur in eine, sondern in viele Anwendungen auswirken, müssen Modifikationen wesentlich behutsamer durchgeführt werden als an normalen Anwendungen. In einigen Fällen wird man sich ggf. gegen Änderungen entscheiden, weil die damit verbundenen Anpassungskosten zu hoch erscheinen.

3 Anwendungsgebiet: Telefonie

Das folgende Kapitel soll einen Überblick über das Anwendungsgebiet der Computer-Telefonie geben. Zunächst werden die Einsatzbereiche für Telefonie-Systeme vorgestellt und danach das Telefonie-System der Fa. Micrologica, an dessen Entwicklung ich im Rahmen dieser Diplomarbeit maßgeblich mitgewirkt habe. Anhand der verschiedenen Hardware-Konfigurationen und Programmierschnittstellen werden die Probleme der Computer-Telefonie erläutert, die zur Entscheidung geführt haben, ein Framework für die Tk-Anlagen-Steuerung zu entwickeln.

Der Begriff *Telefonie* bezeichnet alle Arten von Systemen zur Unterstützung des Telefonierens.[6] Heutzutage wird diese Unterstützung meist durch den Einsatz von EDV erreicht, daher spricht man häufig auch von *CTI* (engl. computer telephony integration).

Definition 18: Telefonie

Systeme, deren Ziel die Unterstützung des Telefonierens ist, werden als Telefonie-Systeme bezeichnet.

Unter dem Begriff Telefonie werden oft auch neuartige Dienste wie Voicemail und Spracherkennung o.ä. gefaßt. Ich möchte mich bei der Diskussion hier auf das reine Vermitteln von Telefonverbindungen beschränken. Welche Art von Anwendung über diese Verbindungen betrieben wird, ist dabei sekundär.

3.1 Anwendungsgebiete für Telefonie-Systeme

Für Telefonie-Systeme gibt es drei Haupteinsatzbereiche: die schwerpunktmäßige Bearbeitung von eingehenden Gesprächen (sog. Inbound-Bereich), das schwerpunktmäßige Führen von ausgehenden Gesprächen (sog. Outbound-Bereich) und die Erleichterung von ein- und ausgehenden Gesprächen für Mitarbeiter, die für ihre eigentliche Tätigkeit nur gelegentlich telefonieren. Je nach Anwendungsfall und eingesetztem Telefonie-System sind fast beliebige Mischformen denkbar.

Von einem *Call-Center* spricht man, wenn die dort arbeitenden Personen mind. 90% ihrer Arbeit mit dem Telefon verrichten. Typische Anwendungsgebiete für Call-Center sind Bestellannahmen von Versandhäusern, Auskunfts- oder Hotline-Dienste und Direktmarketing. Es sind jedoch auch hoch qualifiziertere Tätigkeiten denkbar, die ebensoviel mit dem Telefon arbeiten und eine ähnlich mächtige Telefonie-Unterstützung benötigen.

Ausgehend von einer automatischen Anrufverteilung (engl. automatic call distribution, ACD), wie sie in gängigen Telefonanlagen z.B. für Sammelanschlüsse enthalten oder auch als etwas flexiblere Softwarelösung zu erwerben ist, geht die Entwicklung immer mehr zum Incoming-Call-Management (ICM), das eine sehr viel intelligentere Verteilung der Anrufe beinhaltet. Mit einer flexiblen Anrufverteilung läßt sich oft die Einrichtung von großen Call-Centern zugunsten von kleineren, dezentralen Call-Centern vermeiden, bzw. es können Spezialisten direkt an ihrem Arbeitsplatz erreicht werden, die nur gelegentlich z.B. mit Hotline-Aufgaben betraut werden sollen.

---

[6] Websters Dictionary spricht gar von "the science of communication by telephone".

Ein wesentliches Ziel von Telefonie-Systemen ist es, die Produktivität der telefonierenden Mitarbeiter zu steigern. Wenn sich ohne Unterstützung eines Telefonie-Systems ca. 5-8 ausgehende Gespräche pro Stunde aufbauen und führen lassen, kann sich diese Zahl bei Einsatz eines sog. "Power Dialers" auf ca. 15-23 Verbindungen pro Stunde erhöhen (vgl. [Schneegans 96]). Der Grund für die deutlich höhere Gesprächszahl liegt zum einen am schnelleren Auffinden und Wählen der nächsten Nummer, mehr aber noch daran, daß der Mitarbeiter nur noch Gespräche mit erreichbaren Kunden führen muß. Alle besetzten bzw. nicht erreichbaren Verbindungen werden vom System gar nicht erst zum Mitarbeiter durchgestellt.

Bei eingehenden Anrufen ist eine Steuerung der Zahl der Gespräche durch das System naturbedingt nicht möglich. Da es z.B. nach Fernsehspots zu extremen Gesprächsvolumen kommen kann, gilt es, hier Lösungen zu finden, um möglichst viele Anrufe bedienen zu können, damit kein Kundenkontakt verloren geht. Dies kann z.B. durch das Zuschalten weiterer Call-Center bzw. das Anbieten von Rückrufen geschehen, wenn kein Mitarbeiter erreichbar ist (sog. *Recall-Management*). Alternativ kann der Kunde auch mit einem Spracherkennungssystem (engl. interactive voice response, IVR) bedient werden, was jedoch nicht von allen Kundenkreisen genutzt wird.

Beim Kontakt mit bekannten Kunden ist eine deutliche Serviceverbesserung dadurch zu erreichen, daß das Telefonie-System die Telefonnummer des Anrufers bereitstellt (sofern diese z.B. bei ISDN-Teilnehmern zur Verfügung steht) und dadurch der Anruf zu einem für diesen Kunden zuständigen Mitarbeiter geleitet wird bzw. beim Mitarbeiter sofort eine Maske mit allen notwendigen Kundeninformationen erscheint, sobald sein Telefon klingelt.

3.2 Das Micrologica Communication Center (MCC)

Das Micrologica Communication Center (MCC) ist eine Telefonielösung, die einen großen Teil der Anwendungsbereiche von Telefonie-Systemen abdecken kann. Es werden sowohl der Inbound- als auch der Outbound-Bereich unterstützt. Mehrere MCC-Systeme können standortübergreifend zu einem sog. "virtuellen Call-Center" zusammengeschaltet werden (vgl. [Kuhn 96]).

Um eine möglichst gleichmäßige Auslastung der Mitarbeiter zu erreichen, wird eine flexible Zuordnung der Mitarbeiter zwischen Inbound- und Outbound-Bereich unterstützt (sog. *Call-Blending*). Bei Überlastung der jeweils anderen Gruppe werden nach vordefinierten Regeln Mitarbeiter der jeweils anderen Gruppe mit einbezogen (vgl. [van Dahle 95]).

Das MCC arbeitet mit einer LAN-zentrierten Telefonie-Konfiguration (siehe Kapitel 3.3) und kann daher ohne weitere Hardwareanschaffungen eingesetzt werden, sobald ein CTI-Link (Verbindung eines Servers im LAN zur Tk-Anlage) vorhanden ist. Für jeden Tk-Anlagen-Typ wird jedoch ein spezieller Treiber im MCC benötigt. Da es sehr viele Tk-Anlagen mit unterschiedlicher Ansteuerung auf dem Markt gibt, muß eine ganze Reihe dieser Treiber, die (historisch bedingt) als TSERV bezeichnet werden, erstellt werden.

Um je nach Installationsgröße skalierbar zu sein, ist das MCC in eine Reihe von kommunizierenden Prozessen aufgeteilt, die entweder alle auf einem Server oder verteilt auf mehrere Server arbeiten können. Zusätzlich gehört auf dem PC eines jeden Mitarbeiters (sog. Agenten) ein "Agentenmonitor" zum MCC-System, mit dem sich der Mitarbeiter beim System

an- und abmeldet. Zusätzlich gibt es noch eine Reihe von Administrations- und Statistik-Komponenten, auf die ich an dieser Stelle aber nicht weiter eingehen möchte.

Aufgrund seiner modularen und daher skalierbaren Architektur eignet sich das MCC insbesondere für große bis sehr große Installationen. Bei der Börseneinführung der deutschen Telekom wurden beispielsweise in drei mit dem MCC ausgestatteten Call-Centern in Kooperation mit dem Intelligenten Netz der Telekom zusammen bis zu 70.000 Anrufe pro Stunde verarbeitet (vgl. [Welt 96], [Kuhn 97b]).

3.3 Hardware-Konfigurationen für Computer-Telefonie

Man kann Konfigurationen von Telefonie-Systemen danach klassifizieren, wie die Kopplung von Computer und Telefon bewerkstelligt wird. In [Burton 95] wird u.a. unterschieden in

- LAN- bzw. Server-zentriert

  Die PCs sind wie üblich in einem LAN miteinander verbunden, und nur ein Server wird über einen sog. *CTI-Link* mit der Tk-Anlage verbunden. Ein vorhandenes LAN kann verwendet werden, ebenso meist auch die vorhandene Tk-Anlage. Dadurch ist diese Lösung oft mit geringen Investitionen in neue Hardware verbunden (nur für den CTI-Link).

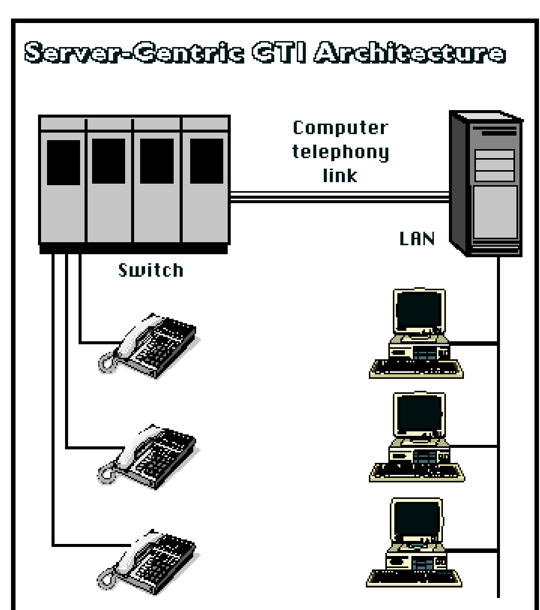

Abbildung 12: LAN- bzw. Server-zentrierte Konfiguration [Burton 95]

Da der Server immer "für andere" steuernd eingreift, spricht man hier von 3rd-Party Call Control. Im Gegensatz dazu spricht man bei Steuerung "für sich selbst" von 1st-Party Call Control. Dieser Ansatz wird sowohl vom Micrologica Communication Center als auch z.B. von Novell mit dem TSAPI-Standard (siehe Kapitel 3.4) verfolgt.

Die Telefone sind weiterhin direkt mit der Tk-Anlage verbunden und können somit nach wie vor auch unabhängig vom PC benutzt werden.

- PC-zentriert

  Jeder PC ist direkt mit einer Tk-Anlage verbunden. Die Telefone werden direkt an den PC angeschlossen. Der PC hat hierdurch einen exklusiven Zugriff auf sein Telefon. Durch diese direkte Kopplung ist auch eine Manipulation der über die Telefonleitung übertragenen Informationen durch den PC möglich. Der PC-zentrierte Ansatz ist jedoch mit höheren Kosten für die notwendige Spezialhardware verbunden.

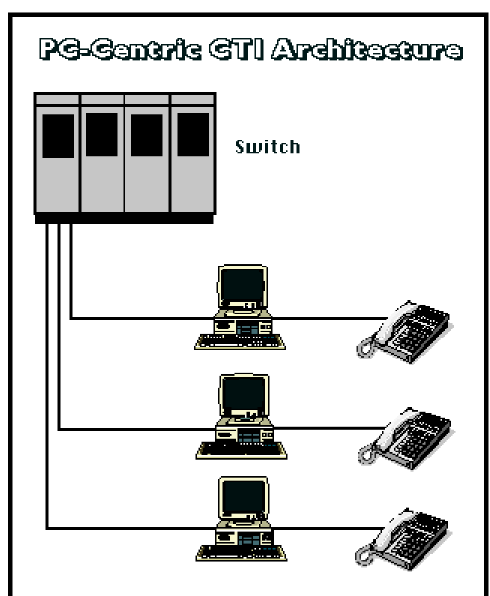

Abbildung 13: PC-zentrierte Konfiguration [Burton 95]

Im praktischen Einsatz wird die Hardware-Konfiguration je nach Einsatzumfeld ausgewählt. Bei großen Installationen überwiegen die LAN-zentrierten Konfigurationen, da sie besser skaliert und zentral administriert werden können.

3.4 Telefonie-APIs

Seit der Einführung der Computer-Telefonie besteht Bedarf an Tk-Anlagen-unabhängiger Steuerung und entsprechenden Standards, um Anwendungen und Hardware unterschiedlicher Hersteller zusammen zu verwenden.

Die ECMA (European Computer-Manufacturers Association) hat im **CSTA Standard** (engl. computer supported telecommunications applications standard) eine Reihe von Telefonie-Diensten definiert und ein konzeptionelles Modell zur Darstellung der beteiligten "Objekte" bereitgestellt, das sog. Half-Call-Modell (siehe Kapitel 3.4.1). Der CSTA Standard verwendet durchgängig 3rd-Party Call Control.

Dieser CSTA-Standard ist jedoch nicht so detailliert, daß er zur Erstellung von Anwendungen bzw. für die Programmierung der Firmware von Tk-Anlagen ausreicht. Er definiert nur das Protokoll, nicht das konkrete API (application programming interface). Die Firmen Novell und AT&T haben 1993 mit **TSAPI** (telephony services API) ein CSTA-konformes API definiert,

anhand dessen Telefonie-Anwendungen hardwareunabhängig erstellt werden können [Cronin 96].

TSAPI definiert Funktionsaufrufe und Datenstrukturen zur Übergabe von Parametern bzw. Events.[7] Neben der Spezifikation des API umfaßt TSAPI auch eine Reihe von Bibliotheken und Modulen zur Installation auf einem Telefonie-Server. Mitte 1996 waren für ca. 30 Tk-Anlagen TSAPI-Treiber verfügbar [Greenfield 96].

Das zweite wichtige Telefonie-API ist **TAPI** (telephony API) von Microsoft / Intel, das ebenfalls 1993 auf den Markt kam und heute einigen Versionen von MS-Windows bzw. Windows NT beiliegt und auch nur für Microsoft-Betriebssysteme verfügbar ist [Udell 94]. Im Gegensatz zum CSTA-Standard wird hier eine Mischung aus 1st-Party und 3rd-Party Call Control und ein anderes konzeptionelles Modell verwendet [Nixon 96].

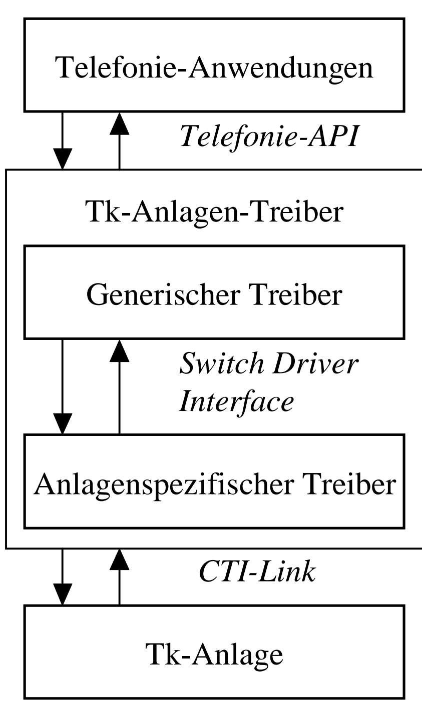

Abbildung 14: Position des Tk-Anlagen-Treibers

Um den Herstellern von Tk-Anlagen die Erstellung eines Tk-Anlagen-Treibers für ein spezielles API zu erleichtern, bieten sowohl TSAPI als auch TAPI einen generischen Treiber an, der aber noch um einen anlagenspezifischen Treiber erweitert werden muß. Abbildung 14 veranschaulicht die Plazierung und den Aufbau eines Tk-Anlagen-Treibers.

Der generische Treiber des API-Definierers (Novell bzw. Microsoft) enthält die zentrale Koordination bzw. den hauptsächlichen Kontrollfluß. Der anlagenspezifische Treiber kommuniziert mit dem generischen Treiber über das sog. *Switch Driver Interface*.

[7] "Events" enthalten im Telefonie-Bereich üblicherweise auch Kontext-Informationen über den eingetretenen Zustandswechsel, so daß sie nicht mit programmiertechnischen "Events" verwechselt werden sollten, die meist nur die Information enthalten, daß sich etwas verändert hat.

Nicht alle Tk-Anlagen unterstützen alle Dienste des jeweiligen Telefonie-APIs. Daher sind im API sondierende Funktionen enthalten, über die der aktuelle Treiber befragt werden kann, welche Dienste unterstützt werden und welche nicht.

Beiden APIs ist gemeinsam, daß sie lediglich eine Reihe von Funktionen zur Verfügung stellen. Bei der Benutzung dieser Funktionen sind vom Anwendungsprogrammierer viele Reihenfolge-bedingungen einzuhalten, die er der Dokumentation entnehmen muß. Die Telefonie-APIs stellen kein Framework im Sinn von Definition 2 zur Verfügung.

Aufgrund ihrer unterschiedlichen Herangehensweise bei der Call Control (1st-Party bzw. 3rd-Party Call Control) eignen sich die verschiedenen Telefonie-APIs unterschiedlich gut für die verschiedenen Konfigurationstypen. Während sich 1st-Party Call Control besonders für PC-zentrierte Konfigurationen eignet, wird in LAN-zentrierten Konfigurationen meist 3rd-Party Call Control verwendet.

3.4.1 Kommunikation zwischen Anwendung und Tk-Anlage

Um eine Tk-Anlage zu steuern, ruft die Anwendung eine oder mehrere Funktionen des jeweiligen Telefonie-APIs auf und erhält vom Anlagen-Treiber Zustandsmeldung über den aktuellen Anlagenzustand.

Auf einer konzeptionellen Ebene werden die Aufträge an die Tk-Anlage (z.B. "baue eine Verbindung zwischen Telefon 123 und 456 auf") als *Requests* bezeichnet. Um dem Tk-Anlagen-Treiber ein Request mitzuteilen, sind je nach API ein oder mehrere Funktionsaufrufe notwendig. Meist muß auch der aktuelle Zustand der Tk-Anlage verfolgt werden, um festzu-stellen, wann der nächste Funktionsaufruf möglich ist.

Um beispielsweise eine Verbindung zwischen zwei Telefonen aufzubauen, kann folgender Ablauf notwendig sein:

1. Funktion aufrufen, um die Verbindung von der Tk-Anlage zum ersten Telefon herzustellen
2. warten, bis die Verbindung hergestellt ist
3. Funktion aufrufen, um die Verbindung von der Tk-Anlage zum zweiten Telefon herzustellen
4. warten, bis die Verbindung hergestellt ist
5. Funktion aufrufen, um beide Verbindungen miteinander zu verknüpfen
6. warten, bis die Verknüpfung durchgeführt ist

Erst nach Erledigung des letzten Schritts ist das Request erfolgreich verarbeitet.

Definition 19: Request

Ein Auftrag an eine Tk-Anlage wird als Request bezeichnet. Diese Aufträge sind meist auf einem höheren Niveau, als die Tk-Anlage in einem Schritt abarbeiten kann.

Um den Zustand der Tk-Anlage zu verfolgen, registriert sich die Anwendung für bestimmte Ereignismeldungen. Diese Ereignismeldungen werden als Events bezeichnet. Die Events sind asynchron8 und beinhalten alle notwendigen Kontext-Informationen über den neuen Zustand, denn im Gegensatz zu computerinternen Events sind sie meist mit einer nicht vernachlässigbaren Zeitverzögerung verbunden, so daß eine Nachfrage nach mehr Details nicht möglich ist. Der Zustand könnte sich bereits schon wieder verändert haben. Insbesondere bei LAN-zentrierten Konfigurationen ergibt sich eine beträchtliche Zeitverzögerung. Diese Tatsache muß von einem Tk-Anlagen-Treiber berücksichtigt werden.

Definition 20: Event

Ein Event ist eine asynchrone Ereignismeldung über Zustandsänderungen in der Tk-Anlage. Es enthält alle verfügbaren Informationen über den neuen Zustand.

3.4.2 Das Half-Call-Modell

Das Half-Call-Modell ist ein konzeptionelles Modell, um die an einer Telefonverbindung beteiligten "Objekte" unabhängig von einer konkreten Tk-Anlage darzustellen. Es ist Teil des CSTA-Standard. Es besteht aus den Basisobjekten Device, Call und Connection. Devices symbolisieren dabei "Telefongerätschaften" aller Art - vom handelsüblichen Telefonapparat bis zum Sammelanschluß. Man kann Devices weiter unterteilen in physikalische Devices (z.B. Telefonen) und logische Devices (z.B. Sammelanschlüssen). Das Call-Objekt drückt die Kommunikationsbeziehung zwischen Devices aus. Die Verbindung von einem Device zu einem Call wird immer über eine Connection realisiert. Abbildung 15 zeigt eine einfache 1-zu-1 Verbindung zwischen zwei Devices.

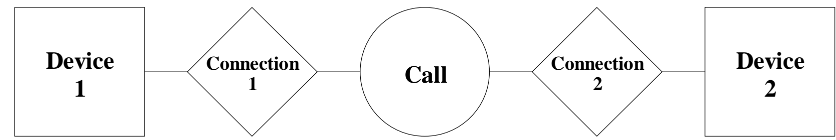

Abbildung 15: Das Half-Call-Modell einer einfachen Verbindung

Während das Connection-Objekt immer eine 1-zu-1-Beziehung zwischen einem Device und einem Call darstellt, kann ein Call, beispielsweise bei einer Konferenzschaltung, eine ganze Reihe von Connections zu anderen Devices haben (siehe Abbildung 16).

[8] d.h. die Tk-Anlage arbeitet weiter, unabhängig davon, ob der Empfänger das Event bereits empfangen hat oder nicht

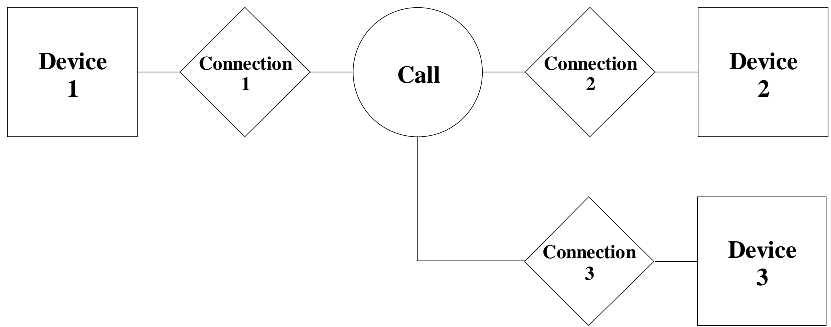

Abbildung 16: Das Modell einer Konferenzschaltung

Andererseits kann ein Device mit mehreren Calls verbunden sein, z.B. wenn ein Teilnehmer während des Gesprächs in Rückfrage geht und eine neue Verbindung zu einem anderen Device aufbaut, während die alte Verbindung weiter gehalten wird (siehe Abbildung 17).

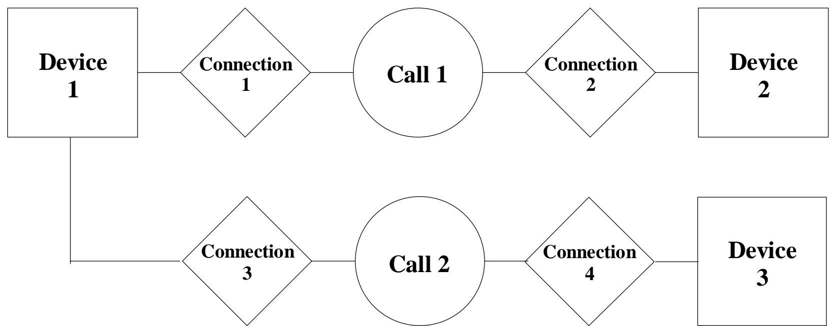

Abbildung 17: Das Modell einer Rückfrage

Alle Basisobjekte im Half-Call-Modell haben einen Zustand:

Der *Zustand eines Device* ergibt sich aus der Summe seiner Connection-Zustände. Ebenso ergibt sich der *Zustand eines Call* aus der Summe seiner Connection-Zustände. Der *Zustand einer Connection* spiegelt den Verbindungszustand im Bezug auf genau ein Device und einen Call wieder.

Häufig werden Verbindungen über Tk-Anlagen-Grenzen hinaus geführt. Dadurch ist der Zugriff auf den Zustand der nicht-lokalen Devices nur sehr begrenzt möglich. Ich unterscheide daher bei meiner Modellierung zwischen lokalen Devices, deren Zustand man immer in Erfahrung bringen kann, und anderen Devices (OtherDevices), über deren Zustand möglicherweise nur begrenzte Informationen vorliegen.[9]

[9] Um Verwechslungen zu verhindern, stehen Devices und OtherDevices bei mir auch in keinerlei Vererbungsbeziehung und sind damit ausdrücklich nicht typkompatibel.

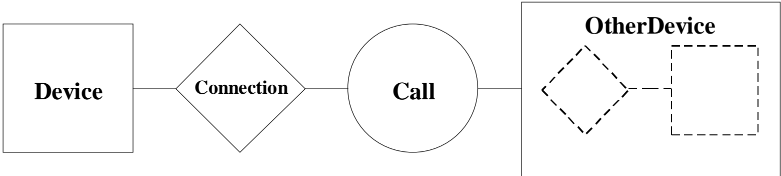

Abbildung 18: Meine Modellierung des Half-Call-Modells

Um den Zustand der Tk-Anlage verfolgen zu können, gibt es die Möglichkeit des Monitoring: Man fordert von der Tk-Anlage Informationen über die Zustandsänderungen der Basisobjekte (Device, Connection und Call) an. (Man sagt "ein Monitorpunkt wird gesetzt".) Dies ist für Devices oder für Calls möglich. Daraufhin wird man über alle Zustandsänderungen der mit dem beobachteten Objekt in Zusammenhang stehenden Connections informiert - die Zustände der Objekte selbst ergeben sich aus den Zuständen der Connections.

Der Name „Half-Call-Modell“ bezieht sich darauf, daß sich alle von der Tk-Anlage gemeldeten Zustandsänderungen immer auf das beobachtete ("gemonitorte") Device beziehen. Bei einer normalen Verbindung zwischen zwei Devices werden (sofern beide Devices beobachtet werden) also zwei „Stränge“ in der Art von Abbildung 18 gehalten, wobei jeweils ein Device als OtherDevice repräsentiert ist. Ein „Strang“ ist also die Hälfte der über einen Call vorliegenden Informationen.

3.5 Probleme der Computer-Telefonie

Im folgenden sollen nur kurz einige Kernprobleme bei der Steuerung von Tk-Anlagen aufgezählt werden. Sie sind bei Micrologica beim Umgang mit der bisher eingesetzten Software erkannt worden und haben als Grundlage für die Definition von hot spots in unserem neu erstellten Framework gedient.

3.5.1 Vielfältige Hardware-Landschaft

Das größte Problem für Computer-Telefonie-Anwendungen ist die Hardwareansteuerung, da es eine große Zahl von Tk-Anlagen (und noch mehr Modelle) auf dem Markt gibt und sich diese in Art, Anzahl und Implementation ihrer Fähigkeiten z.T. deutlich unterscheiden. Um nicht auf eine kleine Zahl unterstützter Modell festgelegt zu sein, ist eine hardwareunabhängige Programmierung von großer Wichtigkeit.

Diese Situation könnte sich in der Zukunft einerseits durch die stärkere Verbreitung von CSTA-konformen Hardware-Implementierungen etwas entschärfen, aber es ist zu erwarten, daß dies durch die starke Dynamik des Marktes mit dem Erscheinen von neuen Tk-Anlagen bzw. Modellen kompensiert wird.

3.5.2 Keine exklusive Steuerung bei LAN-zentrierten Konfigurationen

Durch die Verwendung einer LAN-zentrierten Hardware-Konfiguration kommt es dazu, daß weder die Telefonie-Anwendung noch der Tk-Anlagen-Treiber eine exklusive Kontrolle über den Zustand und das Verhalten der Tk-Anlage haben. Ein Treiber kann bei keinem Kommando an die Tk-Anlage sicher sein, daß diese es ausführen kann, da sich ihr Zustand bereits verändert haben könnte.

Der Treiber ist auf den Event-Strom mit Zustandsmeldungen angewiesen, um ein internes Modell des realen Zustands zu führen und immer wieder zu aktualisieren. Je höher das Anrufvolumen ist und je langsamer der CTI-Link arbeitet, desto größer wird die zeitliche Verzögerung, mit der der Treiber sein internes Modell aktualisieren kann, und um so höher ist die Wahrscheinlichkeit, daß der Treiber Kommandos abschickt, die nicht bearbeitet werden können.

Diese Tatsache muß im Programmiermodell für Telefonie-Anwendungen unbedingt berücksichtigt werden. Einerseits geschieht dies dadurch, daß die Events immer alle vorhandenen Informationen beinhalten - eine Nachfrage ist ja nicht möglich. Andererseits kann der Tk-Anlagen-Treiber bei keinem Kommando an die Tk-Anlage davon ausgehen, daß es auf jeden Fall erfolgreich verarbeitet werden kann, sondern muß immer eine intensive Fehlerüberprüfung vorsehen.

Zur Illustration sei auf Abbildung 12 (LAN- bzw. Server-zentrierte Konfiguration [Burton 95]) verwiesen: Man überlege sich, wie ein Tk-Anlagen-Treiber reagieren sollte, der versucht, ein Gespräch auf einen Telefonapparat zu vermitteln, den der Benutzer gerade abgehoben hat, wobei das Event über das Abheben den Treiber noch nicht erreicht hat.

3.5.3 Verlorene Events

Die Erfahrung beim Umgang mit Tk-Anlagen hat außerdem gezeigt, daß diese unter starker Last (großes zu vermittelndes Anrufvolumen) z.T. Events verlieren. Bei einem Tk-Anlagen-Treiber, der dies nicht berücksichtigt, divergiert in solchen Situationen sehr schnell sein internes Modell vom realen Tk-Anlagen Zustand.

Bei Kenntnis der verwendeten Tk-Anlage kann jedoch oft mit einer guten Wahrscheinlichkeit gesagt werden, welches Event verloren gegangen sein könnte, so daß dieses fehlende Event vom Tk-Anlagen-Treiber simuliert werden kann.

3.5.4 Unzureichende Standardisierung

Die mehrjährige Erfahrung bei Micrologica zeigt, daß die bisherigen Standards nicht eindeutig genug sind, um eine hardwareunabhängige Programmierung zu ermöglichen. So müssen bisher z.T. auch für Tk-Anlagen, für die ein TSAPI-Treiber vorliegt, unterschiedliche Programmversionen erstellt werden.

Folgende Eigenschaften von TSAPI haben sich dabei als besonders problematisch erwiesen:

- Aufgrund der unterschiedlichen Implementationen der Zustände bzw. Zustandsübergänge von Connections in den Tk-Anlagen definiert TSAPI kein verbindliches Modell der Connection-Zustände. So gibt es keine abschließende Liste aller möglichen Zustände, und auch die möglichen Zustandsübergänge sind nicht definiert. Es wird lediglich ein Beispiel eines möglichen Modells angegeben (siehe Abbildung 19), das von Anlagen-Herstellern jedoch beliebig verändert werden kann.

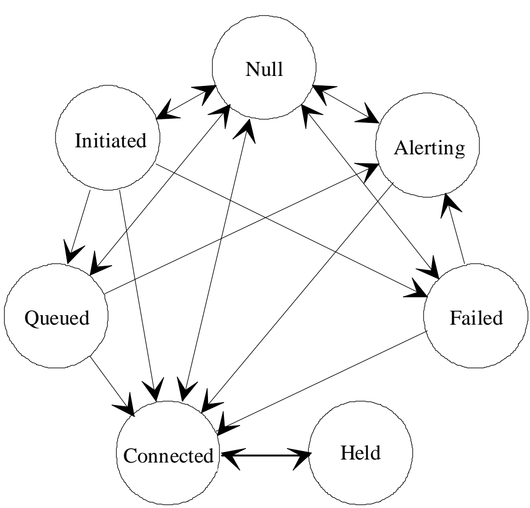

Abbildung 19: Zustandsmodell für Connections [Novell 95, S. 3-13]

- Es sind zwar alle nötigen Event-Typen zur Ablaufverfolgung definiert, ihr Auftreten bzw. die Reihenfolge des Auftretens ist jedoch nicht garantiert. Es wäre wünschenswert, wenn der Tk-Anlagen-Treiber diese Unterschiede zwischen den Tk-Anlagen ausgleichen würde und fehlende Events soweit möglich simulieren könnte. Dieser Vorgang wird im Telefonie-Bereich als *Event-Normalisierung* bezeichnet.

Definition 21: Event-Normalisierung

Die Konvertierung des anlagenspezifischen Event-Stroms in eine anlagenunabhängige Reihenfolge und Darstellung wird als Event-Normalisierung bezeichnet. Es dient dem Ausgleich herstellerspezifischer Anlagen-Eigenschaften.

3.6 Ausgangspunkt zur Framework-Entwicklung

Trotz aller Probleme bei der hardwareunabhängigen Programmierung hat sich andererseits gezeigt, daß für die vom MCC benötigten Funktionen der Tk-Anlagen durchaus eine hardwareunabhängige Programmierung möglich sein müßte. Diese Erfahrungen haben uns als Grundlage gedient für die Entwicklung eines Frameworks für Tk-Anlagen-Treiber, des "Micrologica-Telefonie-Frameworks". Dieses Framework soll es ermöglichen, mit vertretbarem Aufwand Treiber für alle zu unterstützenden Tk-Anlagen zu erstellen.

Im folgenden Kapitel wird dieses Framework und seine Entstehung vorgestellt. Die in diesem Kapitel aufgezeigten Grundlagen und Probleme der Telefonie finden sich dort wieder als Kriterien zur Framework-Gestaltung, aber auch als Faktoren, die zu einer Evolution des Frameworks geführt haben.

4 Das Micrologica-Telefonie-Framework

Dieses Kapitel beschreibt das im Rahmen dieser Diplomarbeit erstellte Framework zur Steuerung von Tk-Anlagen.

Wie in Kapitel 3.2 beschrieben, benötigt das Micrologica Communication Center (MCC) als Unterbau eine Reihe von Treibern für die verschiedenen Tk-Anlagen. Diese Treiber sind jeweils als eigene Prozesse realisiert. Die direkte Steuerung einer Tk-Anlage erfolgt durch den sog. TSERV-Prozeß.

Micrologica hat bereits eine Reihe von Tk-Anlagen-Treibern in C programmiert und in den vergangenen Jahren eine beachtliche Erfahrung mit der Ansteuerung von Tk-Anlagen gesammelt. Mit der steigenden Zahl der zu unterstützenden Tk-Anlagen und der Erkenntnis, daß die realisierten Treiber große Ähnlichkeiten aufweisen, stieg der Bedarf an einem Konzept zur Erstellung von TSERV-Prozessen, das eine Code-Duplizierung vermeidet und es ermöglicht, schnell und effizient Treiber für neue Tk-Anlagen zu erstellen.

Aus diesem Grund wurde beschlossen, ein objektorientiertes Framework für TSERV-Prozesse in C++ [Stroustrup 92] zu entwickeln, um die Erfahrungen im Telefonie-Bereich als Rahmenarchitektur bereitzustellen. Bei der Framework-Entwicklung konnten wir auf die bereits gefundenen Abstraktionen aufbauen und diese objektorientiert umsetzen.

4.1 Bisherige Architektur

Das Micrologica Communication Center besteht aus einer ganzen Reihe von Prozessen, die mittels Interprozeßkommunikation (IPC) miteinander in Verbindung stehen. Schichtenartig bauen Prozesse mit höheren Abstraktionen auf darunterliegenden Prozessen auf. Abbildung 20 zeigt einen Ausschnitt der beteiligten Prozesse. Bei den Kommunikationsbeziehungen der Prozesse in der Abbildung wird zwischen dem Kommunikationsmedium (Transport der Information) und dem über dieses Medium realisierten Kommunikationsprotokoll (Ablauf der Kommunikation) unterschieden.

Der hier mit **MCC** bezeichnete Prozeß besteht, je nach Konfiguration, aus einer Reihe von Einzel-Prozessen, die die eigentliche Telefonie-Anwendung aus Sicht des Anwenders realisieren. Dies ist bei weitem der umfangreichste Teil der gesamten Konfiguration.

Der **ACD_TC**-Prozeß bietet dem MCC eine vereinfachte und auf seine Bedürfnisse zugeschnittene (proprietäre) Schnittstelle zu den Tk-Anlagen. Weiterhin ist er für die Koordination von Tk-Anlagen-übergreifende Vermittlung von Anrufen zuständig, d.h. die Arbeit mit mehreren Tk-Anlagen gleichzeitig. Der ACD_TC-Prozeß kommuniziert mit dem MCC über das proprietäre ACD_TOP-Protokoll.

Der **TSERV**-Prozeß übersetzt die Requests des Telefonie-Protokolls (TSAPI) in Tk-Anlagen-spezifische Befehle (siehe Kapitel 3.4) und steuert mit diesen die Anlage. Er benötigt hierzu ein Modell des aktuellen Zustandes der Tk-Anlage, die er steuert. Es gibt von diesem Prozeß verschiedene Ausprägungen, die jeweils an eine bestimmte Tk-Anlage angepaßt sind. Diese Prozesse wurden auf Basis des Micrologica-Telefonie-Frameworks neu implementiert.

Um den TSERV-Prozeß beliebig im LAN plazieren zu können, steuert dieser nicht direkt die Tk-Anlage, sondern schickt die vorbereiteten Befehle an den (sehr einfachen) **TM-Driver**-Prozeß, der sie an die Tk-Anlage durchreicht. Hierdurch wird der TSERV auch von der direkten Hardwaresteuerung befreit, die im TM-Driver gekapselt ist.

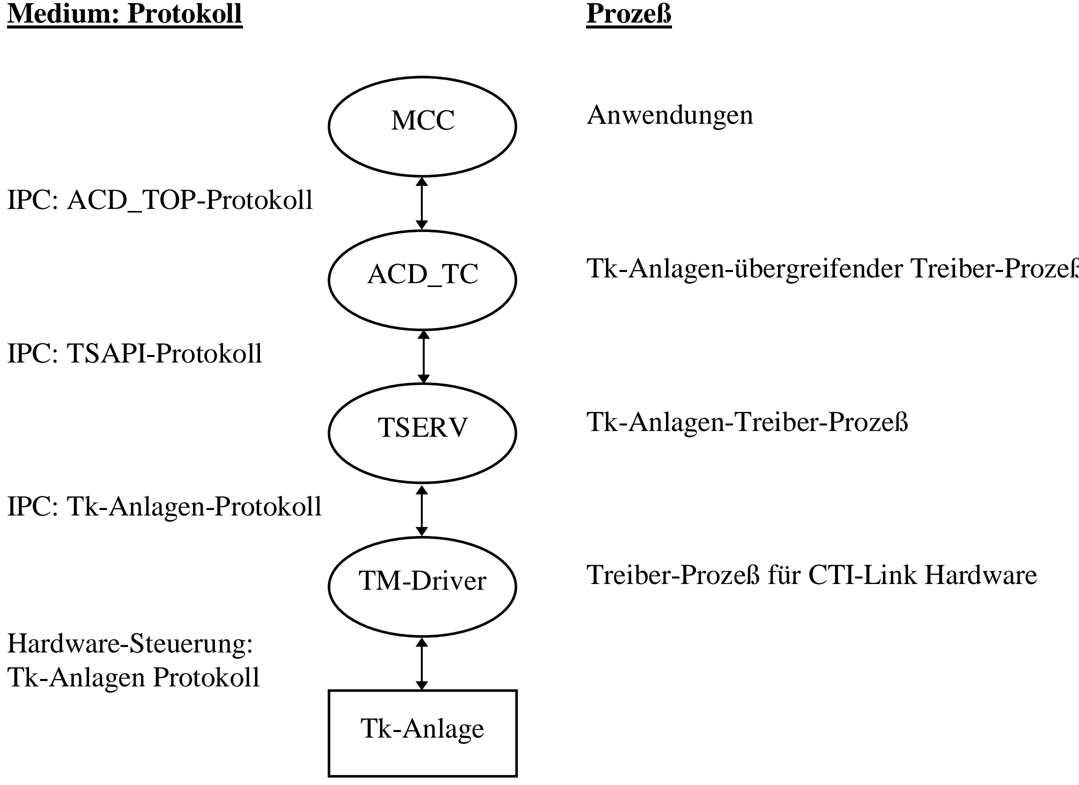

Abbildung 20: Telefonie-Prozesse

Diese Architektur ist sehr flexibel skalierbar, da alle Prozesse per Interprozeßkommunikation (IPC) miteinander kommunizieren und je nach zu erwartender Last alle auf einer Workstation oder alle auf unterschiedlichen Workstations im LAN ablaufen können.

4.1.1 Probleme der bisherigen Architektur

Es hat sich als sehr aufwendig herausgestellt, daß sowohl vom ACD_TC- als auch vom TSERV-Prozeß für jeden neu zu unterstützenden Tk-Anlagentyp eine neue Ausprägung erstellt werden mußte bzw. existierende Varianten modifiziert werden mußten. Die Erstellung neuer TM-Driver war weniger aufwendig.

Diese Prozesse waren bisher in der Programmiersprache C realisiert und wurden angepaßt, indem der Quelltext kopiert und für den neuen Tk-Anlagentyp modifiziert wurde.

Mit der Tatsache, daß der TSERV-Prozeß für jeden Tk-Anlagentyp modifiziert werden müßte, war gerechnet worden. Es stellte sich jedoch heraus, daß sich die verschiedenen Tk-Anlagentypen innerhalb der Grenzen des Telefonie-Protokolls so stark unterschieden (siehe Kapitel 3.5.4), daß jeweils angepaßte ACD_TC-Prozesse nötig waren.

4.2 Ziele der Framework-Entwicklung

Die Motivation zur Entwicklung eines Frameworks lag in dem Wunsch, größere Teile des TSERV-Prozesses bei der Erstellung neuer Ausprägungen wieder zu verwenden und eine Code-Duplizierung mit dem damit verbundenen erhöhten Wartungsaufwand zu vermeiden. Zusätzlich sollte nach Möglichkeiten gesucht werden, das Verhalten aller unterstützten Tk-Anlagen soweit zu vereinheitlichen, daß nur noch eine Ausprägung des ACD_TC-Prozesses benötigt wird, die unabhängig ist von der aktuell verwendeten Tk-Anlage.

Die langjährige Erfahrung im Telefonie-Bereich, die sich einzelne Mitarbeiter erworben haben, sollte in Form einer Rahmenarchitektur allen, auch neuen, Mitarbeitern zur Verfügung gestellt werden.

Als Fernziel wird es für möglich gehalten, daß einige der für den TSERV erstellten Abstraktionen auch in anderen Treiberprozessen verwendet werden können. Diese Möglichkeit habe ich im Rahmen meiner Diplomarbeit jedoch nicht untersucht. Durch die Prozeßarchitektur des MCC-Systems war es möglich, zunächst nur die TSERV-Prozesse auf ein objektorientiertes Framework umzustellen. Die Umstellung anderer Prozesse des MCC-Systems in ähnlicher Weise wird z.Z. erwogen.

Konkret sollten durch die Framework-Entwicklung folgende Ziele vorrangig erreicht werden:

- schnelle Erstellung von TSERV-Prozessen für weitere Tk-Anlagentypen
- Kapselung des unterschiedlichen Verhaltens der Anlagentypen im TSERV-Prozeß
- strengere Protokollspezifikation für die Telefonie-Protokolle, um eigene Anwendungsentwicklung zu erleichtern (siehe Kapitel 3.5.4)
- Möglichkeit zur Unterstützung mehrerer Telefonie-Protokolle, da die Entwicklung des Marktes bzw. der Kundenwünsche nach speziellen Protokollen nicht absehbar ist

Wie diese Zielsetzung umgesetzt wurde, soll der nächste Abschnitt anhand der gewählten Architektur verdeutlichen.

4.3 Aufbau des Frameworks

Das Design des Frameworks ist einerseits geprägt von dem technischen Anwendungsbereich und andererseits von den bisher bei Micrologica gemachten Erfahrungen mit der Entwicklung von Telefonie-Systemen. Da die Anwender des Frameworks Softwareentwickler sind, die oft bereits einiges Hintergrundwissen im Telefonie-Bereich haben, haben wir uns nicht gescheut, auf den bekannten technischen Abstraktionen aufzubauen.

Die im Framework enthaltenen fachlichen Abstraktionen sind nur zu einem geringen Teil bei der Entwicklung des Frameworks entstanden. Fast immer sind sie das Ergebnis der bei der konventionellen Programmierung in C gemachten Erfahrungen. Wir haben die dort gefundenen Abstraktionen „lediglich“ objektorientiert modelliert und in Form eines Frameworks zur Spezialisierung für konkrete Anwendungsfälle zur Verfügung gestellt.

Bei der Modellierung der hot spots des Frameworks haben wir uns auf die Erfahrung der Mitarbeiter verlassen, die im Telefonie-Bereich über langjährige Erfahrung verfügen.

Intern wird der vom Framework modellierte TSERV-Prozeß nicht in Schichten, sondern in Bereichen[10] von fachlich zusammengehörigen Klassen strukturiert. Das Framework ist in 5 voneinander streng getrennte Bereiche gegliedert:

- Ein abstraktes Modell eine Tk-Anlage, die sog. „Virtuelle Tk-Anlage“
- Das Telefonie-Protokoll
- Das Tk-Anlagen-Protokoll
- Die Telefonie-Basis-Objekte
- Die softwaretechnische Basis

Diese Bereiche sind streng voneinander getrennt und bauen aufeinander auf. Diese Struktur ist in Abbildung 21 dargestellt.

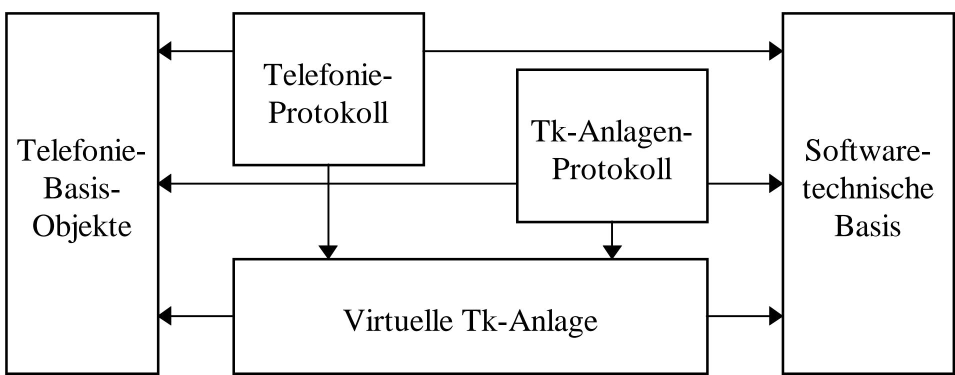

Abbildung 21: Die Bereiche des Micrologica-Telefonie-Frameworks

Die Pfeile zeigen an, in welcher Richtung die einzelnen Bereiche aufeinander aufbauen. Insbesondere im Fall von bidirektionalen Abhängigkeiten sind die Bereiche lediglich über allgemeine Oberklassen wie z.B. "Nachrichtenempfänger" miteinander gekoppelt, um ihre Unabhängigkeit voneinander zu gewährleisten.

Implementierungen der Bereiche sind einzeln austauschbar. Dadurch lassen sich große Teile der Implementation in unterschiedlichen Ausprägungen der TSERV-Prozesse wiederverwenden. Somit wird die Entwicklungszeit für neue Ausprägungen drastisch reduziert, da oft nur ein Bereich neu implementiert werden muß. Im günstigsten Fall liegen bereits Implementationen für alle Bereiche im Framework vor, die nur noch kombiniert werden müssen. Auf der Ebene dieser Bereiche ist das Framework als Black-Box-Framework angelegt. Die interne Realisierung der Bereiche ist jedoch eher ein White-Box-Framework.

---

[10] In Kapitel 6 werde ich eine detaillierte Begriffsbildung zur Aufteilung von Frameworks vornehmen. Hier möchte ich zunächst allgemein von „Bereichen“ eines Frameworks sprechen.

Es wird angenommen, daß die Implementation des Tk-Anlagen-Protokolls die weitaus häufigste Veränderung ist, die für neue Ausprägungen des TSERVs notwendig sein wird. Die Zahl der existierenden Telefonie-Protokolle ist dagegen sehr gering (vgl. Kapitel 3.4).

4.3.1 Die Bereiche im Überblick

An dieser Stelle möchte ich einen Überblick über die Aufgabenverteilung der einzelnen Bereiche geben. Eine detaillierte Beschreibung würde den Rahmen dieser Arbeit sprengen. Auf eine Reihe interessanter Aspekte werde ich jedoch in Kapitel 5 noch genauer eingehen. Zum Verständnis der folgenden Diskussionen ist jedoch ein grobes Verständnis des Zusammenspiels der Bereiche notwendig.

Die *Virtuelle Tk-Anlage* verkörpert eine abstrakte Tk-Anlage, die ein Modell der real zu steuernden Tk-Anlage enthält. Aufträge an die Tk-Anlage, sog. Requests, werden ihr zur Ausführung übergeben. Die Requests selbst sind Teil der von uns modellierten Telefonie-Basis-Objekte. Die Virtuelle Tk-Anlage ihrerseits verschickt Meldungen über Zustandsänderungen in der Tk-Anlage (genauer: in ihrem Modell der Tk-Anlage), sog. Events[11]. Dieser Bereich wird durch eine einzige Klasse (ocVirtualPbx) nach außen vertreten. Alle Implementationsdetails dieses Bereichs sind hinter dieser Fassade verborgen.

Der Bereich *Telefonie-Protokoll* enthält Konverter, die die Requests an die Tk-Anlage bzw. die Events von der Tk-Anlage zwischen einer internen Darstellung und der Darstellung des jeweiligen Protokolls konvertieren. Um ein anderes Telefonie-Protokoll zu unterstützen, würde dieser Bereich komplett ausgetauscht und durch eine, mit den Telefonie-Basis-Objekten erstellte, neue Implementierung ersetzt werden.

Analog zum Telefonie-Protokoll enthält der Bereich *Tk-Anlagen-Protokoll* Konverter, um Events von der Tk-Anlage in die intern verwendete Darstellung zu konvertieren. Weiterhin enthält dieser Bereich spezielle Ableitungen der Requests, um diese auf der jeweiligen Tk-Anlage ausführen zu können.

Die *Telefonie-Basis-Objekte* modellieren die Grundelemente, mit denen ein Tk-Anlagen-Treiber umgehen muß. So sind u.a. die "Objekte" des Half-Call-Modells (Call, Device, Connection) vorhanden, aber auch ein Modell für Aufträge an die Tk-Anlage, sog. *Requests*. Die Requests enthalten alle zu einem Auftrag gehörigen Daten sowie eine Zustandsmaschine (engl. finite state machine, FSM), die den Stand der Bearbeitung und die in jedem Zustand notwendigen Aktionen modelliert.

Der Bereich *Softwaretechnische Basis* ist nicht telefonie-spezifisch. Er enthält z.B. Containerklassen, ein Meta-Object-Protocol oder eine generische Implementierung des Singleton-Musters (siehe Kapitel 5.3.2).

Für die Erstellung von TSERV-Prozessen ist die Abtrennung der Telefonie-Basis-Objekte sowie der softwaretechnischen Basis nicht notwendig. Sie werden höchstwahrscheinlich immer in dieser Form verwendet werden und könnten daher Teil der Virtuellen Tk-Anlage sein. Es wäre jedoch denkbar, daß sie in anderen Prozessen getrennt von ihr verwendbar sind.

[11] siehe Kapitel 3.4.1 zur Definition von Events bei Telefonie-Systemen

4.3.2 Aufträge an die Tk-Anlage: Requests

Die konkrete Steuerung der Tk-Anlage erfolgt durch Requests (siehe Definition 19). Ein zentrales Problem bei der Aufteilung des TSERV-Prozesses in einzeln austauschbare Bereiche war die Behandlung der Requests. Mit ihnen muß in mehreren Bereichen umgegangen werden: Die Aufträge gehen in einem vom Telefonie-Protokoll abhängigen Format beim TSERV ein, und die Abarbeitung wird von der Virtuellen Tk-Anlage gesteuert, wobei die Befehle an die Tk-Anlage vom jeweiligen Anlagentyp abhängen. Es mußte also eine Modellierung gefunden werden, die die Unabhängigkeit der einzelnen Bereiche wahrt.

Im Telefonie-Bereich ist es üblich, die Abläufe von Requests mit Zustandsmaschinen (engl. Finite State Machines, FSMs) zu modellieren, da die Abläufe vom aktuellen Zustand der beteiligten Geräte (bzw. der Objekte im Half-Call-Modell) abhängen. Die von uns modellierten Requests enthalten daher neben den zur Steuerung nötigen Daten (z.B. anzurufende Telefonnummer) eine FSM, die den Ablauf des Requests steuert.

Die gefundene Lösung zur Modellierung von Requests verwendet das State-Muster aus [GHJV 95, S. 305] zur Modellierung der FSMs. Sie erreicht über eine mehrstufige Vererbungshierarchie die Unabhängigkeit der einzelnen Bereiche. Dabei sind die unterschiedlichen Hierarchieebenen verschiedenen Bereichen zugeordnet. Abbildung 22 zeigt diese Vererbungshierarchie am Beispiel eines MakeCall-Requests. Dabei hat jedes Request entsprechend dem State-Muster einen Zeiger auf ein Zustandsobjekt vom Typ RequestState. Dieses Zustandsobjekt ist typischerweise ein Singleton. Von der Klasse RequestState werden dann die konkreten Zustandsklassen abgeleitet. Beim Erreichen eines neuen Zustands wird automatisch die Methode ActionOnEnter ausgeführt. Alle eintreffenden Events werden zur Verarbeitung an das aktuelle Zustandsobjekt delegiert.

Eine wesentliche Erweiterung des State-Musters besteht darin, daß bei unserer Modellierung sowohl die Request-Klassen als auch Zustandsklassen über mehrere Vererbungsstufen spezialisiert werden. Hierdurch wird es möglich, zum einen die Gemeinsamkeiten aller Requests abzubilden, z.B. daß sie alle am Ende der Verarbeitung entweder erfolgreich erledigt sind (DoneState) oder fehlschlugen (FailedState). Andererseits wird Vererbung eingesetzt, um die allgemeinen Teile von den anlagenspezifischen Teilen zu trennen.

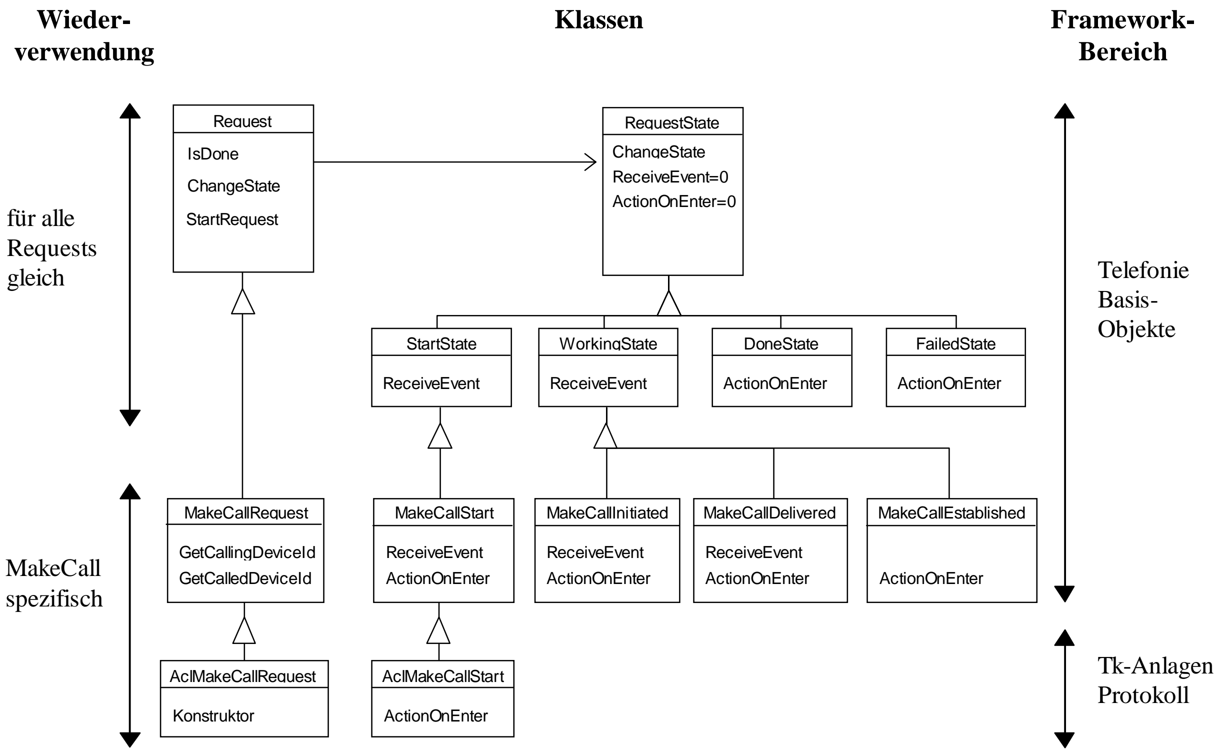

Abbildung 22: MakeCall-Klassen

Um die Erzeugung der Tk-Anlagen-spezifischen Requests vom Telefonie-Protokoll unabhängig zu gestalten, werden die Requests im Telefonie-Protokoll-Bereich immer über eine abstrakte Fabrik-Klasse [GHJV 95, S. 87] erzeugt.

4.4 Benutzung des Telefonie-Frameworks

Der Anwendungsentwickler, der einen neuen Tk-Anlagen-Treiber implementieren möchte, leitet eine Klasse von der Klasse `ocTservProzess` ab und muß die aufgeschobene Methode `vConfig()` implementieren. Danach kann der Prozeß mit der Methodenfolge `vInit()`, `vRun()`, `vDone()` gestartet werden (siehe Abbildung 23). Die Tatsache, daß die einzige in dieser Klasse implementierte Methode nicht direkt aufgerufen wird, ist ein typisches Beispiel für den invertierten Kontrollfluß in diesem Framework.

```
class ocNewTserv : public ocTservProcess
{
public:
    virtual void vConfig(void)[12]
    {
        // Erzeugen der Tk-Anlagen-spezifischen Objekte
        poPbxEventConverter = new ocAclEventConverter;
        poRequestFactory = new ocAclRequestFactory;

        // Erzeugen der Telefonie-Protokoll-spezifischen Objekte
        poTelephonyEventConverter = new ocTsapiEventConverter;
        poTelephonyRequestConverter = new ocTsapiRequestConverter;

        // Konfiguration
        vSetPbxEventConverter(poPbxEventConverter);
        vSetRequestFactory(poRequestFactory);
        vSetTelephonyEventConverter(poTelephonyxEventConverter);
        vSetTelephonyRequestConverter(poTelephonyRequestConverter);
    };
};

int main()
{
    ocNewTserv oTserv;

    oTserv.vInit();
    oTserv.vRun();
    oTserv.vDone();
    return 0;
}
```

Abbildung 23: Typische Verwendung des Frameworks

Wenn es zu allen verwendeten Bereichen bereits passende Implementationen gibt, beschränkt sich die Arbeit des Anwendungsentwicklers lediglich auf das Konfigurieren und Parametrisieren dieser Implementationen.

Für eine neue Tk-Anlage muß der Bereich des Tk-Anlagen-Protokolls neu implementiert werden. Er besteht zum einen aus einem Event-Konverter, der die anlagenspezifischen Events in ein internes Format umwandelt, und zum anderen aus für die konkrete Anlage spezialisierten Requests. Die Requests werden durch Vererbung aus generischen Requests aus den Telefonie-

[12] Sowohl die Implementation der Klasse ocNewTserv in derselben Datei wie die main()-Routine als auch die inline-Implementation der Methode vConfig() dienen lediglich der übersichtlichen Darstellung in dieser Arbeit und werden sonst nicht empfohlen.

Basis-Objekten abgeleitet. Außerdem muß eine Unterklasse der Fabrik-Klasse gebildet werden, die die anlagenspezifischen Requests erzeugt.

4.5 Vorgehensweise bei der Framework-Entwicklung

Die Entwicklung des Telefonie-Frameworks begann mit einer Reihe von Interviews mit den bisher an der Entwicklung beteiligten Personen. In diesen Interviews wurden sog. "Kommunikations-Szenarien" erstellt, die die Kommunikation zwischen den einzelnen Prozessen des MCC-Systems beschreiben.

Das Design des bestehenden Systems wurde zunächst dokumentiert, wobei die Dokumente in mehreren Autor-Kritiker-Zyklen überarbeitet wurden. Bei der Besprechung unserer Entwicklungsdokumente war es sehr hilfreich, daß unsere Gesprächspartner ebenfalls Software-Entwickler waren und so leicht eine "gemeinsame Sprache" gefunden werden konnte. Sie hatten intensive Kenntnisse im Telefonie-Bereich. Da sie nicht aktiv am Design des Frameworks, sondern nur als Interviewpartner zum Erfahrungsaustausch an der Entwicklung beteiligt waren, gab es keine Probleme mit evtl. nicht vorhandenen Kenntnissen in objektorientierten Vorgehensweisen.

Bei der Erstellung dieser Szenarien wurde viel Wert auf das Warum der einzelnen Kommunikationsschritte gelegt. Dies hat sich beim Design der Framework-Struktur als sehr hilfreich erwiesen. Da der Fokus nicht auf der Beschreibung des Kommunikationsprotokolls lag (das auch zur Disposition gestanden hätte, wären gravierende Probleme gefunden worden), wurden die Kommunikations-Szenarien nicht formalisiert (z.B. als Interaktionsdiagramme) erstellt, sondern in Prosa (Freitext) formuliert.

Parallel zur Erstellung der Kommunikations-Szenarien wurde ein Glossar der bei Micrologica üblichen Terminologie im Telefonie-Bereich angelegt. Dies half, die einzelnen Begriffe schärfer gegeneinander abzugrenzen und die Grundlage für gemeinsames Verständnis aller Beteiligten zu legen. Weiterhin konnte das entstandene Glossar auch anderweitig zur Einarbeitung von neuen Mitarbeitern, die z.T. nicht einmal direkt an Entwicklungen im Telefonie-Bereich beteiligt sind, eingesetzt werden.

Die bisherige Prozeßstruktur wurde in Diagrammen visualisiert. Bei diesem Vorgehen wurden bereits kleinere Ineffizienzen in der Kommunikation zwischen den Prozessen erkannt, die ohne größeren Aufwand im bestehenden System behoben werden konnten, so daß z.B. die erzeugte Netzlast im LAN reduziert wurde.

Das erste Design des Telefonie-Frameworks wurde mit Hilfe von CRC-Karten [Beck 89] erstellt. Die Verwendung von CRC-Karten hat sich als sehr nützlich erwiesen, um das dynamische Verhalten des zu entwickelnden Systems zu entwerfen. Insbesondere die Flexibilität des Ansatzes und die Möglichkeit, mit mehreren Personen gleichzeitig an dem Entwurf arbeiten zu können, war hilfreich und hat zu einem gemeinsamen Verständnis beigetragen.

5 Erfahrungen aus der Evolution des Micrologica-Telefonie-Frameworks

Die Beobachtungen und Erfahrungen aus der Entwicklung des Micrologica-Telefonie-Frameworks dienen als Grundlage meiner Überlegungen zur Framework-Evolution. Ich möchte daher in diesem Kapitel die wichtigsten Schritte in der bisherigen Evolution dieses Frameworks darstellen und einige allgemeine Folgerungen für das Design und die Konstruktion von Frameworks ableiten.

5.1 Unnötige Compilezeit-Abhängigkeiten

Nachdem das Framework eine gewisse Größe erreicht hatte, stellte sich heraus, daß viele Änderungen dadurch erschwert wurden, daß sie fast immer eine aufwendige Neucompilierung erforderten. Dabei war der resultierende Compilierungsaufwand weitgehend unabhängig von Art und Ort der Änderung - es wurde fast immer das gesamte Framework und die (zu diesem Zeitpunkt einzige) Anwendung neu übersetzt.

Eine Analyse der Abhängigkeiten der einzelnen Dateien ergab, daß tatsächlich fast alle auf irgendeine Weise direkt oder indirekt voneinander abhingen. Daraufhin wurde die Art der Abhängigkeiten untersucht. Es stellte sich heraus, daß auch diverse Dateien voneinander abhingen, die auf fachlicher Ebene voneinander völlig unabhängig waren. Nachdem diese Abhängigkeiten entfernt bzw. auf ein Minimum reduziert waren, sank der durchschnittliche Compilierungsaufwand je Änderung auf etwa die Hälfte, so daß die zeitliche Auswirkung von Änderungen deutlich geringer wurde.

5.1.1 Arten von Abhängigkeiten

Die Abhängigkeiten zwischen Framework-Komponenten bestimmen die Auswirkungen von Änderungen. Man kann dabei zwischen zwei Arten von Abhängigkeiten unterscheiden: fachlich motivierte und technisch motivierte. Die fachlichen Abhängigkeiten spiegeln Abhängigkeiten innerhalb der modellierten Domäne wider. Änderungen an fachlichen Konzepten, auf denen andere Konzepte basieren, führen auch dort zu Konsequenzen.

> **Definition 22: Fachliche Abhängigkeiten**
>
> Abhängigkeiten, die "in der Natur" der modellierten Domäne liegen, bezeichnen wir als fachliche Abhängigkeiten.

Bei der Implementation werden die fachlichen Konzepte mit den in der jeweilig verwendeten Programmiersprache zur Verfügung stehenden Mitteln ausgedrückt. Dies hat Konsequenzen auf die bestehenden Abhängigkeiten. In C++ sind sprachbedingt auch einige Implementationsdetails (private/protected) in der Klassenschnittstelle zu sehen:

- Benutzung anderer Klassen (#include)
- private Klassenvariablen
- protected Klassenvariablen

Diese Abhängigkeiten in den Header-Dateien (.hpp) führen dazu, daß das System häufiger compiliert werden muß als fachlich nötig. Übliche Generierungswerkzeuge wie z.B. make (siehe [Feldman 79]) prüfen lediglich das Dateidatum, ob sich abhängige Dateien verändert haben - nicht aber inhaltlich, ob eine Neucompilierung tatsächlich nötig ist.

Definition 23: Compilezeit-Abhängigkeit

Die Abhängigkeiten einzelner Compilierungseinheiten untereinander, die bei Änderung einer Einheit ebenfalls die Neucompilierung einer anderen Einheit erzwingen, werden als Compilezeit-Abhängigkeiten bezeichnet.

5.1.2 Reduktion der Abhängigkeiten

Die Compilezeit-Abhängigkeiten gehen oft weit über die fachlichen Abhängigkeiten hinaus. Daher sollten die Abhängigkeiten der Header-Dateien untereinander auf das absolut notwendige Minimum beschränkt werden, um auf dem Framework basierende Anwendungen soweit wie möglich vor unnötigen Neucompilierungen zu schützen. Oftmals reicht es aus, die Include-Dateien von der Header-Datei in die Implementations-Datei (.cpp) zu verlagern.

<table>
<tr><th>gängige Praxis</th><th>weniger Abhängigkeiten</th></tr>
<tr><td><pre>#include "MakeCall.hpp"
#include "AnswerCall.hpp"
//...

class ocRequestFactory {
public:
  ocMakeCall oNewMakeCall(...) =0;
  ocAnswerCall oNewAnswerCall(...) =0;
  // ...
};</pre></td><td><pre>class ocMakeCall;
class ocAnswerCall;
// ...

class ocRequestFactory {
public:
  ocMakeCall oNewMakeCall(...) =0;
  ocAnswerCall oNewAnswerCall(...) =0;
  // ...
};</pre></td></tr>
</table>

Abbildung 24: Reduzierung der Abhängigkeiten

Das Beispiel in Abbildung 24 zeigt unnötige Abhängigkeiten: Ein Klient, der die Request-Fabrik zur Erzeugung von MakeCalls benutzt, würde bei der ersten Variante auch bei Änderungen an der Schnittstelle des AnswerCall neu compiliert werden. Bei der zweiten Variante müssen nur diejenigen Compilierungseinheiten neu übersetzt werden, die selbst AnswerCall.hpp einbinden, wie z.B. die Implementations-Datei der Request-Fabrik.

Die volle Tragweite dieser Umstellung (und der analogen Umstellung der konkreten ACL-RequestFactory) wird erst deutlich, wenn man die Zahl der Abhängigkeiten für einige Dateien gegenüberstellt und dabei die indirekten Abhängigkeiten mitzählt:

Datei	**vorher** abhängig von n Dateien bei 10 Request-Typen	**nachher** abhängig von n Dateien bei 10 Request-Typen
RequestFactory.hpp	10	0
ACL-RequestFactory.hpp	21	1
ACL-RequestFactory.cpp	22	22
TSAPI-RequestKonverter.hpp (benutzt die RequestFactory)	11	1

Abbildung 25: Abhängige Dateien

Obwohl die Kopplung zwischen den Partitionen Telefonie-Protokoll (hier: TSAPI) und den Telefonie-Basis-Objekten über die abstrakte Oberklasse RequestFactory erfolgt, hätte vorher eine Änderung an den einzelnen Request-Typen zu einer Neucompilation des Request-Konverters geführt. Die Kapselung der Partition wurde jetzt somit deutlich verbessert. Es wird aber auch deutlich, daß sich die fachlichen Abhängigkeiten natürlich nicht reduzieren lassen, so daß die Implementierung der konkreten Fabrik weiterhin alle konkreten Request-Klassen kennen muß.[13]

Es gibt in C++ nur 3 Fälle, in denen Header-Dateien in einer Klassendeklaration eingebunden werden sollten [Lakos 96, S. 379]:

1. Subtyp-Beziehung: die Klassen erben voneinander
2. Enthält-Beziehung: die Klasse enthält ein Exemplar (keinen Pointer) einer anderen Klasse (oder wird als Template-Parameter verwendet)
3. Inline-Methoden: andere Klassen werden von Inline-Implementationen benötigt

Bei allen anderen Verwendungen (z.B. als Rückgabewert oder Parameter) ist das Einbinden der Header-Datei nicht notwendig.

Inline-Methoden sollten ohnehin vermieden werden, da Änderungen an ihrer Implementation unweigerlich zur Neucompilierung des Systems inklusive alle Framework-Klienten führt.[14] Selbstverständlich sollte eine Header-Datei nur die Deklaration einer (!) Klasse enthalten.

Eine Nichtbeachtung dieses Prinzips kann bei großen Projekten dazu führen, daß alleine der Aufwand für die notwendigen Neucompilationen Änderungen unmöglich bzw. unpraktikabel macht.

---

[13] Dies ließe sich z.B. durch ein anderes Erzeugungsverfahren (z.B. Product-Trader alias "späte Erzeugung") verhindern.

[14] Leider ist es bei aktuellen Compilern für Templates oft notwendig, die Implementation ebenfalls in die Header-Datei einzubinden, so daß Template-Methoden quasi zu Inline-Methoden werden.

Ein anderes Beispiel macht deutlich, daß leicht auch fachlich unabhängige Partitionen durch eine unbedachte Implementation miteinander verbunden sein können: Eine frühe Version der verwendeten String-Klasse verwendete zur Fehlerausgabe direkt Methoden aus dem IPC-Framework, was dazu führte, daß alle Partitionen, die Strings verwendeten, nur zusammen mit dem IPC-Framework getestet werden konnten. Die Aufhebung dieser Kopplung hat zwar zu einer kleinen Redundanz geführt, aber die Trennung der Partitionen wieder hergestellt.

5.1.3 Bessere Generierungswerkzeuge

Wie bereits erwähnt, betrachtet das gängige Generierungswerkzeug make lediglich das Dateidatum, um festzustellen, ob eine Neucompilation notwendig ist. Dies führt dazu, daß im Extremfall das Hinzufügen eines Leerzeichens oder eines Kommentars zur Neucompilation des gesamten Systems führt.

Hier könnte Abhilfe geschaffen werden durch ein Generierungswerkzeug, das zu jeder Datei eine Kopie erzeugt, die von Kommentaren befreit und mit einer Standard-Formatierung versehen wird. Nur wenn sich der Inhalt dieser Datei ändert, müßte eine Neucompilation der abhängigen Dateien ausgelöst werden. (Zur Erzeugung des Systems bzw. zur weiteren Bearbeitung würde selbstverständlich weiterhin die Originaldatei verwendet.)

Gerade das nachträgliche Einfügen von Kommentaren sollte nicht durch eine aufwendige Neucompilation "bestraft" werden.

5.2 Umstellung des Meta-Object-Protocol

Die Requests (an die Tk-Anlage) sind in einer Vererbungshierarchie organisiert (siehe Abbildung 26). Über weite Strecken werden sie im System nur unter ihrer Oberklasse behandelt. Diese polymorphe Verwendung ist ein wichtiges Mittel, die einzelnen Bereiche zu entkoppeln. An einigen Stellen müssen sie jedoch unter ihrer vollen Schnittstelle bekannt sein, so daß ein sog. "down-cast" nötig ist, also eine Typkonvertierung von einer Oberklasse auf eine Unterklasse abhängig vom dynamischen Typ des Objekts.

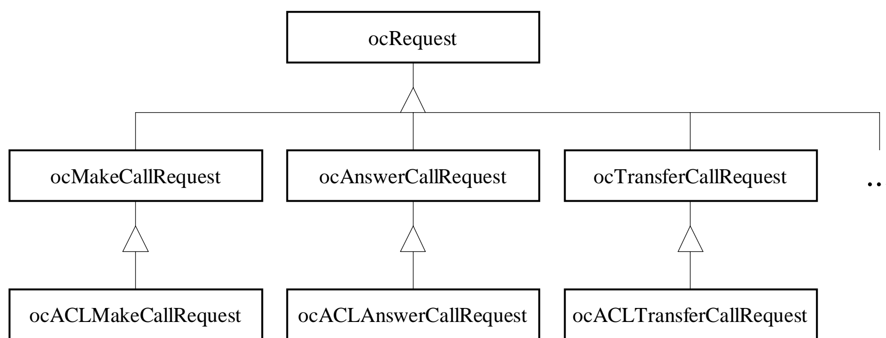

Abbildung 26: Vererbungshierarchie der Requests

Zunächst wurde es als eine Ausnahme betrachtet, daß die dynamische Typinformation benötigt wird. Da die zu diesem Zeitpunkt verwendeten C++-Compiler keine Informationen über den dynamischen Typ zur Verfügung stellten, wurde in der Oberklasse aller Requests eine sondierende Funktion abstrakt implementiert, über die man eine String-Repräsentation des

dynamischen Typs bekommen kann. Mit dieser konnte dann bei Bedarf der dynamische Typ ermittelt werden.

Im Laufe der Erweiterung des Frameworks ergab sich jedoch noch an diversen anderen Stellen ein Bedarf nach dynamischen Typinformationen, so daß ein allgemein verfügbarer Mechanismus benötigt wurde. So können beispielsweise Events (siehe Abbildung 27) in der virtuellen Tk-Anlage unter ihrer Oberklasse verwaltet und den Objekten des Half-Call-Modells zugeordnet werden - zur Auswertung muß jedoch ihr konkreter Typ bekannt sein.

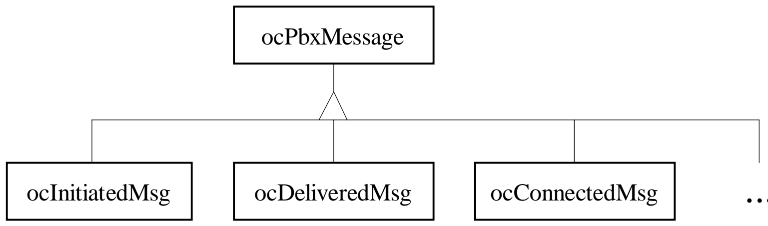

Abbildung 27: Vererbungshierarchie der Events

Dieser Bedarf nach Meta-Informationen tritt offenbar recht häufig auf, so daß fast alle größeren Frameworks für Programmiersprachen ohne Zugriff auf Meta-Informationen einen solchen Mechanismus bereitstellen (z.B. ET++, Microsoft MFC etc.). Die Notwendigkeit, mit dem dynamischen Typ von Objekten zu arbeiten, resultiert aus dem Entwurf lose gekoppelter Systeme. Solange Objekte zwischen lose gekoppelten Bereichen nur in einer Richtung ausgetauscht werden, läßt sich das gewünschte Verhalten der polymorphen Objekte meist durch virtuelle Methoden erreichen. Sobald jedoch Objekte polymorph in beiden Richtungen ausgetauscht werden, entsteht sehr häufig der Bedarf, auf den konkreten Typ der Objekte zuzugreifen.

Allgemein werden Informationen über den Typ eines Objekts zur Laufzeit als Meta-Object-Protocol bezeichnet. Nach [Siberski 95] umfaßt dieser Oberbegriff vier Bereiche:

1. Abfragen über die Klassenzugehörigkeit (Runtime Type Information)
2. Abfragen über die Vererbungsbeziehung
3. Abfragen über die Repräsentation des Objektzustandes
4. Abfragen über die Schnittstelle

Im Telefonie-Framework wird (zumindest bisher) lediglich der erste Bereich benötigt.

Die verwendeten Compiler stellten überhaupt keine MOP-Unterstützung zur Verfügung. Es war jedoch abzusehen, daß der in Arbeit befindliche ANSI-Standard für C++ mindestens Runtime Type Information (RTTI) beinhalten würde, so daß in einiger Zeit die von uns benötigte Funktionalität in den meisten C++-Compilern enthalten sein würde (siehe [Stroustrup 94b, Seite 390ff], [Meyers 96, Seite 12ff]).

Es stellten sich somit hauptsächlich drei Fragen:

1. Sollte das Framework auf einen allgemeinen MOP-Mechanismus umgestellt werden, obwohl sich die Umstellung auf alle Kundenklassen auswirken würde ?
2. Wie geht man mit den zu erwartenden Änderungen im Sprachstandard um ? Sollte man versuchen jetzt eine Hilfskonstruktion zu implementieren, die in absehbarer Zeit überholt sein wird ?
3. Wie kann die benötigte Untermenge der MOP-Funktionalität am einfachsten implementiert werden ?

Die Beantwortung der ersten Frage fiel am leichtesten. Da zu diesem Zeitpunkt kaum Kundenklassen betroffen waren, war es einfach, sich für eine Umstellung zu entscheiden, insbesondere deshalb, weil mit einem weiteren Wachstum der betroffenen Vererbungshierarchien gerechnet wurde.

Der Diskussion der anderen beiden Fragen widmen sich die beiden folgenden Abschnitte.

5.2.1 Umgang mit kommenden Spracheigenschaften

Ganz allgemein stellt sich die Frage, wie mit absehbaren Änderungen in der Implementationssprache umgegangen werden soll. Im Bezug auf C++ betrifft dies nicht nur das Meta-Object-Protocol, sondern auch Behälterklassen, String-Klasse oder Wahrheitswerte, deren Implementation vom ANSI-Standard geregelt wird.

Der laufende Standardisierungsprozeß von C++ ist noch nicht abgeschlossen. Der Abschluß des Standardisierungsverfahrens wird für 1998 erwartet. Es gibt jedoch bereits jetzt eine Reihe von Veröffentlichungen, die den zukünftigen ANSI/ISO-Standard recht genau beschreiben ([Ellis 90], [Stroustrup 94a bzw. b.], [Musser 96]).

Zur Zeit ist der Sprachumfang der verfügbaren C++-Compiler sehr unterschiedlich. Viele Compiler unterstützen erst eine Untermenge des geplanten Standards. Sowohl für den Entwickler von Anwendungen als auch für den Entwickler eines Frameworks stellt sich die Frage, wie er mit den neuen Eigenschaften von C++ umgeht:

a) Wenn er sie intensiv verwendet, besteht die Gefahr, daß sein Code nicht mehr auf Compiler portabel ist, die diese Eigenschaft noch nicht unterstützen.

b) Wenn er sich bemüht, sie nicht zu nutzen, muß er evtl. umfangreiche Hilfskonstruktionen erstellen, um die benötigte Funktionalität zu erhalten.

Variante a) ist noch am ehesten zu akzeptieren, da im Laufe des Jahres 1997 wohl die meisten wichtigen Hersteller von C++-Compilern alle neuen Eigenschaften implementiert haben werden.

Es stellt sich jedoch die Frage, wie mit noch nicht verfügbaren Eigenschaften umgegangen werden soll.

Wichtigster Gesichtspunkt war es, Portabilität zu erreichen und nicht zu Änderungen am Quelltext gezwungen zu werden, sobald neue Compiler-Versionen erscheinen. Zum anderen sollte aber nicht zu viel Arbeit in die Implementation von Funktionalität gesteckt werden, die es in ein paar Monaten evtl. "frei Haus" gibt.

Dies hat zu zwei Designgrundsätzen für die Implementation zu erwartender Spracheigenschaften geführt:

1. *Die Verwendung der Nachimplementierungen sollte den standardisierten Eigenschaften semantisch so ähnlich wie möglich sein.*

2. *Die Benennung muß sich lexikalisch unbedingt von der standardisierten Benennung unterscheiden, um nicht mit ihr in Konflikt zu kommen.*

Bei den Container-Klassen haben wir uns für die vom Compiler mitgelieferte Bibliothek entschieden, für die aus einem anderen Projekt bereits eine portable Implementation für andere Plattformen vorlag. Zu einem späteren Zeitpunkt wäre es möglich, das API der Klassenbibliothek mit Hilfe der STL-Klassen aus dem C++-Standard zu implementieren.

Für die Realisierung eines Meta-Object-Protocols haben wir uns für den Einsatz einer Reihe von Makros entschieden, die in der Verwendung den RTTI-Konstrukten des C++-Standard entsprechen, jedoch nur eine Untermenge der Funktionalität implementieren.

Der hier vorgeschlagene Weg geht davon aus, daß die aktuelle Standardisierung der Sprache C++ kurz vor dem Abschluß steht. Die vorgestellte Methode, damit umzugehen, ist jedoch allgemeiner anwendbar, sofern Informationen über zukünftige Veränderungen der Sprache bzw. Implementationsumgebung vorliegen.

5.2.2 Implementation eines Meta-Object-Protocol

Wie bereits angedeutet, wurden diese Informationen zuerst explizit mit einer Abfragemethode in den jeweiligen Oberklassen realisiert, die von jeder Unterklasse überschrieben werden mußte. Dies ist allerdings ein für den Programmierer recht aufwendiges Verfahren und insbesondere bei mehrstufigen Vererbungshierarchien auch sehr fehlerträchtig. (Da die erste Vererbungsstufe bereits die abstrakte Methode der Oberklasse redefiniert, findet bei weiteren Vererbungsstufen keine Warnung durch den Compiler mehr statt, wenn die Redefinition vergessen wird.)

Das Abbilden der Typinformationen auf Strings hatte den zusätzlichen Nachteil, daß zur Compilezeit keine Prüfung auf die richtige Schreibweise des Klassennamens erfolgen konnte. Erst zur Laufzeit wurde der dynamische Typ nicht bzw. falsch erkannt, weil ein Schreibfehler vorlag. Gerade bei einer so mechanischen Tätigkeit wie dem Übernehmen des Klassennamens in eine Klassen-Variable treten Schreibfehler jedoch verstärkt auf.

Um diesen Problemen zu begegnen, mußte also eine Lösung gefunden werden, die folgenden Ansprüchen genügte:

- möglichst einfache Anwendung durch den Programmierer
- Prüfung der Schreibweise von Klassennamen zur Compilezeit
- Warnung zur Compilezeit, wenn das Einfügen des RTTI-Mechanismus vergessen wird
- es braucht nur der dynamische Typ ermittelt zu werden, nicht seine Stellung in der Vererbungshierarchie (andere Abfragen werden nicht benötigt)
- kein zu großer Implementierungs-Aufwand, da in Kürze die meisten Compiler RTTI unterstützen werden
- leichter Übergang auf den RTTI-Mechanismus des C++ Standard in späteren Versionen des Frameworks

Zur Implementierung eines Meta-Object-Protocol in C++ gibt es grundsätzlich drei Möglichkeiten (vgl. [Siberski 95]):

1. Integration in die Sprache bzw. in den Compiler
2. Verwendung eines Precompilers
3. Verwendung von Makros

Die erste Variante wird in Zukunft sicher vorherrschen, ist aber ohne den Zugang zum Quelltext des Compilers unmöglich und selbst dann nur mit extrem hohen Aufwand zu bewerkstelligen. Auch die Verwendung eines Precompilers ist mit einem hohen Erstellungsaufwand für den Precompiler verbunden.

Daher wird meist eine Lösung durch Makros vorgezogen (z.B. in ET++ oder in den Microsoft Foundation Classes). Die Verwendung von Makros arbeitet meist so, daß je ein Makro in die Klassen-Deklaration und die Klassen-Implementation eingefügt wird (vgl. [Schmidberger 95], [Gamma 92, S. 22f]).

In Übereinstimmung mit unseren Design-Grundsätzen für zu erwartende Spracheigenschaften (vgl. Kapitel 5.2.1) haben wir ein ähnliches Verfahren gewählt, das jedoch mit nur einem Makro in der Klassen-Deklaration auskommt:

- In jede Klassen-Deklaration, die Typinformationen enthalten soll, wird das Makro RTTI(Klassenname) eingefügt.
- In die Oberklassen sollte statt des Makros RTTI(Klassenname) das Makro RTTIBASE(Klassenname) eingefügt werden. Dies ermöglicht eine bessere Prüfung, ob eine Unterklasse den RTTI-Makro vergißt.
- Zur Laufzeit kann der dynamische Typ eines Objekts mit dem Makro TYPEID(Objekt) abgefragt und mit dem Ergebnis von CLASSID(Klassenname) verglichen werden.

Diese Realisierung der Typabfrage ähnelt stark dem kommenden C++ Standard. Die folgende Implementierung erfüllt die erwähnten Ansprüche. Für die Implementierung der Makros werden bereits jetzt zwei Implementierungen zur Verfügung gestellt: Eine für Compiler mit RTTI-Unterstützung und eine für Compiler ohne RTTI-Unterstützung.

```
#define RTTIBASE(ClassName)   virtual String oGetType(void) = 0;
#define RTTI(ClassName)       static void __foo(void) {}; \
                              virtual String __oGetType(void) const \
                              { static String __oType = #ClassName; \
                              ClassName::__foo(); return __oType; };
#define TYPEID(obj)           (obj).__oGetType()
#define CLASSID(type)         (type::__foo(), #type)
```

Abbildung 28: RTTI-Makros für Compiler ohne RTTI

Das Makro RTTIBASE fügt in die Klasse eine abstrakte Methode oGetType ein, die eine String-Repräsentation der Klasse (bei uns den Klassennamen) liefert. Alle Unterklassen müssen das Makro RTTI verwenden, das eine Implementation für oGetType in die Klasse einfügt (siehe Abbildung 29). Da Makros einen reinen Textersetzungsmechanismus verwenden, kann zur Compilezeit weder geprüft werden, ob der Parameter des RTTI-Makros der Name der jeweiligen Klasse ist bzw. ob es sich überhaupt um einen Klassennamen handelt. Um sicherzustellen, daß es sich bei dem Parameter zumindest um einen in diesem Kontext bekannten Klassennamen handelt, erzeugt das Makro (ansonsten unnötigen) Code[15], der auf den Parameter als Klasse zugreift. So wird zur Compilezeit ein Syntaxfehler provoziert, wenn der Parameter z.B. durch einen Tippfehler nicht stimmt.

```
class ocAnwendungsklasse {
public:
    RTTI(ocAnwendungsklasse);       // einzige einzufügende Zeile

    virtual void vMethode(void);
    // ...
};
```

Abbildung 29: Einfügen der RTTI-Informationen in Anwendungsklassen

Um von der internen Realisierung zu abstrahieren, werden die Makros TYPEID verwendet, um den dynamischen Typ von Objekten zu erfragen. Da der Inhalt der String-Repräsentation einer Klasse nicht bekannt ist (es wird lediglich garantiert, daß er für alle Objekte der Klasse gleich ist und sich von allen anderen Klassen unterscheidet), kann die String-Repräsentation einer Klasse mit dem Makro CLASSID aus dem Klassennamen erzeugt werden.

```
if ( TYPEID(oObject) == CLASSID(ocClass) )
{
      // der dynamische Typ entspricht der Klasse ocClass
}
else {
      // das Objekt hat einen anderen dynamischen Typ
}
```

Abbildung 30: Verwendung der RTTI-Informationen zur Laufzeit

[15] dieser überflüssige Code kann jedoch vom Compiler oft wegoptimiert werden

Es ist zu erwarten, daß der C++-Standard demnächst verabschiedet wird und in Zukunft alle Compiler den RTTI-Mechanismus beherrschen werden. Aus Effizienzgründen wird es dann sinnvoll sein diesen zu verwenden und nicht mehr mit eigenen String-Repräsentationen zu arbeiten. Um dann eine erneute Umstellung zu vermeiden ist bereits jetzt eine zweite Version des Makros erstellt worden, die den Mechanismus des Compilers verwendet (siehe Abbildung 31). Sobald das Framework mit einem Compiler mit RTTI-Unterstützung eingesetzt wird, wird zunächst nur die Implementierung der Makros ausgetauscht. Die Anwendungen sind hiervon völlig unberührt. (Bei einem Compilerwechsel müssen sie ohnehin neu compiliert werden.)

```
#define RTTIBASE(ClassName)          /* */
#define RTTI(ClassName)              /* */

#define TYPEID(obj)                  typeid(obj)
#define CLASSID(type)                typeid(type)
```

Abbildung 31: RTTI-Makros für Compiler mit RTTI

Bei einem größeren Versionswechsel wäre es dann irgendwann in der Zukunft denkbar, völlig auf die Makros zu verzichten und direkt den RTTI-Mechanismus des Compilers zu verwenden.

Die hier vorgestellte Implementation ist einfacher einzusetzen als andere makro-basierte Implementierungen, da sie sich auf die im Kontext des Telefonie-Frameworks benötigte Funktionalität beschränkt. Gleichzeitig wird sie die Evolution des Frameworks durch die Orientierung am kommenden C++-Standard erleichtern.

5.2.3 Designrichtlinien für die Verwendung des Meta-Object-Protocol

Man kann in vielen Systemen, die auf eine lose Kopplung achten, beobachten, daß immer wieder Fälle auftreten, wo der Zugriff auf den dynamischen Typ notwendig ist (s.o.). Dennoch bedeutet eine Typumwandlung einen schweren Eingriff in das Typsystem und sollte daher mit Vorsicht eingesetzt werden.

Der unüberlegte Einsatz eines MOP kann sogar die Bemühungen um eine lose Kopplung behindern. Sobald z.B. Klassen unter ihrer Oberklasse übergeben werden, vor ihrer Verwendung aber immer eine Typumwandlung auf den dynamischen Typ erfolgt, liegen oft ebenso viele Annahmen über den konkreten Typ vor wie bei einer direkten Übergabe.

Man muß sich immer die Frage stellen, ob der Bedarf an Meta-Informationen nicht das Resultat eines schlechten Designs ist und der gleiche Effekt auch eleganter durch die Redefinition von Methoden der Oberklasse gelöst werden kann, die polymorph aufgerufen werden können.

Für den Einsatz des Meta-Object-Protocols für die Übergabe von Objekten zwischen zwei lose zu koppelnden Bereichen haben wir uns daher folgende Designrichtlinien zu eigen gemacht:

- So weit wie möglich sollte die Kommunikation zwischen unabhängigen Bereichen über abstrakte Klassen erfolgen.
- Die abstrakten Oberklassen sollten ausschließlich fachlich motiviert sein.
- Der Zugriff auf die konkrete Klasse mittels MOP darf nur für zurückgegebene Objekte verwendet werden, anderenfalls besteht keine lose Kopplung, wenn in beiden Bereichen immer der konkrete Typ bekannt sein muß.

Insbesondere die dritte Richtlinie ist wichtig: Sobald der dynamische Typ benötigt wird, ist dies ein Zeichen dafür, daß Annahmen gemacht werden, die über die Garantien der Oberklasse hinaus gehen! In einigen wenigen Fällen mag dies gerechtfertigt sein. Die Implikationen für die Kopplung dürfen dabei jedoch nicht übersehen werden.

Insbesondere besteht die Gefahr, daß im Laufe der Evolution eines Frameworks durch "kleine Erweiterungen hier und da", die durch den Zugriff auf den dynamischen Typ gelöst werden, ursprünglich unabhängige Bereiche plötzlich eng gekoppelt sind, da Annahmen über den konkreten Typ gemacht werden. Diese Degeneration des ursprünglichen Designs ist äußerst schwer zu erkennen, da die Parameterübergabe ja weiterhin ausschließlich unter der Oberklasse erfolgt.

5.3 Evolution der Zustandsmodellierung

Wie in Kapitel 3.4 dargelegt, muß die virtuelle Tk-Anlage ein Modell des aktuellen Zustands der realen Tk-Anlagen haben. Hierzu verwenden wir das in Kapitel 3.4.1 beschriebene Half-Call-Modell.

Über den Zustand der Tk-Anlage hinaus muß auch noch der Abarbeitungszustand der an die Tk-Anlage geschickten Requests verfolgt werden. Da die Requests aus mehreren Schritten bestehen können, ist es wichtig, die erfolgreiche Erledigung der einzelnen Schritte z.B. zum Verbindungsaufbau zu verfolgen, um danach den nächsten Schritt einzuleiten bzw. Erfolg oder Mißerfolg des Requests zu signalisieren.

Der Zustand und die notwendigen Arbeitsschritte werden im Telefoniebereich üblicherweise durch endliche Automaten (engl. finite state machines, FSM) modelliert. Es stellt sich daher die Frage, wie eine solche FSM objektorientiert zu modellieren ist. Hinzu kommt der Wunsch, allgemeine Abläufe spezialisieren zu können (z.B. für eine neue Tk-Anlage), ohne alle Zustände und Zustandsübergänge neu implementieren zu müssen.

Bei Micrologica gibt es bereits seit langem eine Bibliothek mit C-Funktionen, die die Modellierung von FSMs unterstützen. Darauf basierend wurden für die High-Level Telefonie-Prozesse des MCC einige C++-Klassen entwickelt, die FSMs kapseln sollen und auch eine Vererbung ermöglichen [Thießen 96].

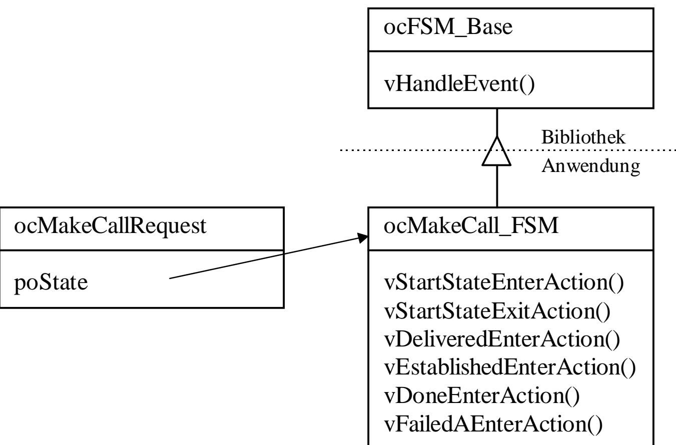

Abbildung 32: Zustandsmodellierung mit Micrologica-FSM-Klassen

Diese Klassen wurden zunächst zur Modellierung der Requests verwendet. Beim Einsatz dieser FSM-Klassen traten jedoch einige Probleme auf:

- Die Zustände können nicht benannt werden, sondern müssen fortlaufend aufsteigend durchnumeriert sein. (Es können nur die Aktionen beim Erreichen und Verlassen des Zustands benannt werden.)
- Um eine FSM in diesem Modell implementieren zu können, müssen vorher der Automat aufgezeichnet und die Zustände numeriert werden.
- Die beim Erreichen bzw. Verlassen eines Zustands auszuführenden Methoden werden als Methoden-Zeiger übergeben. Es findet keine Überprüfung statt, ob alle Methoden vom selben Objekt sind.

Aus der Arbeit mit diesen FSM-Klassen und den daraus resultierenden Problemen haben wir folgende Forderungen an eine benutzerfreundlichere und leichter änderbare FSM-Implementierung aufgestellt:

- Die Zustände sollten symbolisch benannt werden können.
- Die FSM sollte auch im nachhinein leicht erweiterbar sein.
- Die Spezialisierung von Zuständen sollte einfach möglich sein, da sie sehr häufig benötigt wird.
- Die Spezialisierung von Übergängen soll möglich sein, wird aber wahrscheinlich selten benötigt.

In der Literatur werden diverse Möglichkeiten diskutiert, wie FSMs objektorientiert modelliert werden können: State-Muster (vgl. [GHJV 95, S. 305]), tabellenorientierte Ansätze (siehe z.B. [Stevenson 95], [Ackroyd 95]) oder eine Erweiterung der sog. "reification technique (RT)" (siehe [SaneCamp 95]).

Wir haben uns für eine Umstellung auf das State-Muster entschieden, da die Mächtigkeit für uns ausreichend ist und jeder Zustand als eine Klasse modelliert ist, die später durch Vererbung auch spezialisiert werden kann. Durch diese Zustandsklassen wird ein benannter Kontext für die in dem jeweiligen Zustand auszuführenden Aktionen gebildet, was bei der Modellierung sehr hilfreich war und auch das "Hineindenken" in diesen Zustand bei späteren Änderungen erleichtert hat.

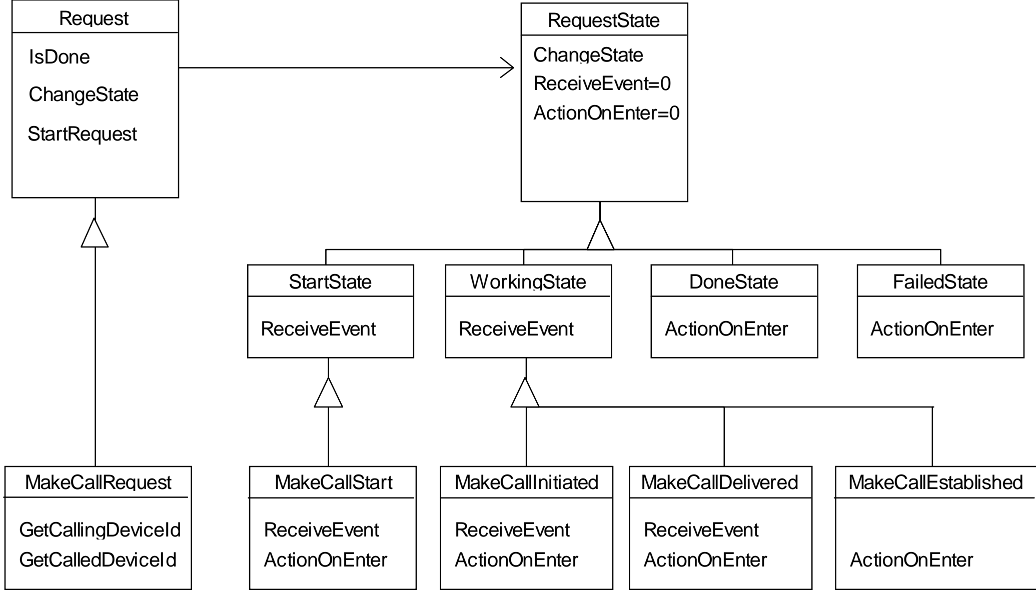

Abbildung 33: Zustandsmodellierung mit State-Muster

Bei der Spezialisierung der Zustände trat das Problem auf, daß die Zustände global als Singleton-Objekte vorliegen. Da diese über den Klassennamen referenziert und dann als Singleton-Objekt von dieser Klasse erzeugt werden, bestand keine Möglichkeit, einen referenzierten Zustand durch Vererbung zu spezialisieren.

Der erste Lösungsansatz bestand darin, eine Singleton-Registry einzuführen, wie in [GHJV 95, S. 130ff] vorgeschlagen. Um einen Zustand zu spezialisieren, meldete sich ein neuer Zustand unter dem Namen des ursprünglichen Zustands bei der Singleton-Registry an und überschrieb so den alten Zustand. Diese Lösung hatte jedoch eine Reihe gravierender Nachteile:

- die Spezialisierung der Zustände ist global, d.h. falls ein Zustand aus verschiedenen Kontexten heraus erreicht wird, wird immer der spezialisierte Zustand erreicht, was gelegentlich zu Verwirrung führte

- die so entstehenden Automaten sind kaum mehr verständlich, da das Überschreiben der Zustände erst zur Laufzeit stattfindet und man leicht den Überblick verliert, welcher Zustand tatsächlich noch in Benutzung ist und welcher überschrieben wird und daher nie erreicht werden kann

- die Realisierung der endlichen Automaten wird vermischt mit dem Singleton-Mechanismus, der auch an anderen Stellen benötigt wird

Die Probleme der Unübersichtlichkeit scheinen im Ansatz der Spezialisierung von Automaten zu liegen, die prinzipiell problematisch ist. Überhaupt ist die Spezialisierung, Vererbung oder auch Einbettung von endlichen Automaten ein noch keineswegs intensiv betrachtetes Gebiet und birgt eine sehr hohe Komplexität. Eine vertiefende Betrachtung der Probleme und Lösungsansätze ist in [SaneCamp 95] zu finden. Das dort vorgestellte Verfahren und die dazu notwendige Modellierung sind jedoch recht aufwendig.

Wir haben uns aufgrund der hohen Komplexität dieses Themas auf die von uns unbedingt benötigte Funktionalität beschränkt und keinen allgemeinen Ansatz zur Spezialisierung von endlichen Automaten gesucht. Dabei sind wir zu folgendem Modell gekommen:

- Automaten können nur als Ganzes spezialisiert werden
- Zustände werden als Singleton-Objekte repräsentiert, die über ihren Klassennamen referenziert werden, aber es ist keine Spezialisierung des Zustands möglich, d.h. es wird immer auch tatsächlich der über den Namen referenzierte Zustand erreicht
- um einen Automaten zu spezialisieren, wird sein Start-Zustand spezialisiert
- der spezialisierte Zustand kann Zustandsübergänge in andere spezialisierte Zustände haben, oder in die Zustände des Original-Automaten. Sobald ein Zustand des Original-Automaten erreicht ist, verhält sich der Automat wieder wie der ursprüngliche Automat (spezialisierte Zustände können dann nicht mehr erreicht werden)
- jeder spezialisierte Zustand ruft immer die Methoden seines Ober-Zustands auf

Dieses Modell hat sich für die von uns benötigten Automaten als praktikabel herausgestellt, da üblicherweise nur sehr wenige Zustände nach dem Start-Zustand spezialisiert werden mußten. Danach konnten meist die ursprünglichen Zustände weiterverwendet werden. Diese Form der Anpaßbarkeit hat genügt, um die Automaten hinreichend leicht verändern zu können, und hat damit ausgereicht, um die Hauptprobleme der Micrologica-FSM-Klassen zu beheben.

Eine weitere Erfahrung mit der Implementierung von Automaten mit dem State-Muster betrifft die enorme Anzahl der benötigten Klassen. Um 10 Request-Typen für 1 Tk-Anlage zu implementieren, wurden ca. 60 Zustands-Klassen benötigt. Für die Spezialisierung der Request-Typen je weiterer Tk-Anlage würden noch einmal ca. 20 zusätzliche Klassen benötigt. Alle diese Zustands-Klassen sind jedoch sehr einfach, so daß der Overhead der Klassendeklaration im Vergleich zum Umfang der Methoden beachtlich groß ist. Mit den Micrologica-FSM Klassen wären bei einer vergleichbaren Implementierung nur ca. 20-30 Klassen nötig gewesen. Die Frage, wie man endliche Automaten in objektorientierten Systemen implementieren kann, scheint daher noch nicht befriedigend gelöst.

Die Problematik der Spezialisierung von Automaten hat sich als deutlich komplexer herausgestellt, als zunächst angenommen. Die gefundene Lösung ist weit davon entfernt, allgemeingültig zu sein, aber sie hat das konkrete Problem gelöst. Der Umgang mit derartigen Automaten ist mit Sicherheit ein interessantes Forschungsthema, falls es gelingt, die Komplexität im Umgang mit den Automaten in den Griff zu bekommen.

5.3.1 Aufzählungstypen

Eine andere, häufig verwendete Technik, um eine Menge von Zuständen zu repräsentieren, ist die Definition eines Aufzählungstyps, der alle möglichen Zustände enthält. Für die Zustandsmodellierung des Half-Call-Modells wurde zunächst ihre Verwendung erwogen, dann aber aus Gründen der mangelhaften Erweiterbarkeit verworfen.

Die Modellierung von Zuständen durch Aufzählungstypen macht den möglichen Wertebereich deutlich und erleichtert damit die Lesbarkeit des Quelltextes. Dennoch hat sie einen gravierenden Nachteil: der Wertebereich ist im nachhinein kaum änderbar. Das bedeutet, daß eine Evolution eines Frameworks, das Aufzählungstypen verwendet, nur durch die Modifikation des Framework-Quelltextes möglich ist - nicht aber durch Spezialisierung.

Die Ursache liegt darin, daß Aufzählungstypen in C++ keine Klassen sind und daher im nachhinein nicht mehr erweitert werden können, ohne den ursprünglichen Quelltext zu verändern. Sie verletzen damit das *Offen-Geschlossen-Prinzip* (engl. open-closed principle) [Meyer 88].

Definition 24: Offen-Geschlossen-Prinzip

Das Offen-Geschlossen-Prinzip besagt, daß eine wiederverwendbare Komponente sowohl offen für Erweiterungen sein soll als auch abgeschlossen, d.h. in einem stabilen Zustand, so daß sie (für den Anwendungsprogrammierer unerreichbar) z.B. in einer Bibliothek abgelegt werden kann.

Da ein Framework oft in compilierter Form vorliegt, ist eine Änderung der Definition eines Aufzählungstyps durch den Anwendungsprogrammierer nicht möglich. Die Anforderungen verschiedener Anwendungsprogrammierer können auch durchaus gegenläufig sein, so daß eine zentrale Änderung im Framework oft nicht sinnvoll ist.

Als Ersatz für Aufzählungstypen sind in C++ Defines und Konstanten möglich. Bei beiden können im nachhinein Werte hinzugefügt werden, sie haben jedoch den Nachteil, daß keine Überprüfung stattfindet, ob der betreffende Wert nicht an anderer Stelle bereits mit anderer Bedeutung verwendet wird.

In Fällen, wo nur eine geringe Zahl von Zuständen benötigt wird, bietet sich das Ausweichen auf eigene Klassen für die Zustände an. Diese können dann polymorph (siehe z.B. State-Muster [GHJV 95, S. 305ff]) oder im Zusammenhang mit einem Meta-Object-Protocol eingesetzt werden.

5.3.2 Eine wiederverwendbare Singleton-Implementation

Da die Zustandsmodellierung bei den Requests in großem Stil das State-Muster einsetzt, standen wir vor dem Problem, Vererbungshierarchien von State-Objekten zu realisieren. Diese State-Objekte sind zustandslos, da sie nur das Verhalten in einem Zustand repräsentieren, und werden daher als Singletons modelliert. Bei der Realisierung anhand der Beispiel-Implementation des Singleton-Musters aus [GHJV, S. 128ff] wurde deutlich, daß diese Implementation nicht wiederverwendbar ist und gravierende Einschränkungen aufweist.

Ich werde zunächst auf die Defizite der Beispiel-Implementation eingehen und danach aufzeigen, wie man durch generative Programmierung (vgl. [Eisenecker 96]) zu einer wiederverwendbaren Implementation kommen kann. Über die folgende Singleton-Implementierung hinaus lassen sich sicher noch diverse andere Implementationsprobleme mit generativer Programmierung sehr elegant lösen.

Das Singleton-Muster soll ausdrücken, daß es von einer bestimmten Klasse im laufenden System nur maximal ein Objekt geben kann bzw. soll. Die übliche Implementation in C++ (vgl. [GHJV 95], S. 128ff) verwendet eine static-Variable in der Singleton-Klasse, die einen Zeiger auf das einzige erzeugbare Exemplar hält. Wenn dieser Zeiger noch nicht gesetzt ist, wird ein neues Exemplar erzeugt, sonst wird immer ein Zeiger auf das bereits erzeugte Exemplar zurückgeliefert.

Wenn man in einer Anwendung mehrere Klassen zu Singletons machen möchte und z.B. Vererbungshierarchien von Singletons hat, treten gleich eine ganze Reihe von Problemen auf:

- Unterklassen einer Singleton-Klasse erben die Singleton-Eigenschaft nicht. Die Initialisierung der static-Variable muß immer in der konkreten Klasse geschehen, da in C++ static-Methoden nicht dynamisch gebunden werden können. Die Unterklasse muß die Initialisierung daher explizit selbst vornehmen. Daher wird in [GHJV 95, S. 130ff] für solche Fälle mit Vererbung eine Implementierung mit einer globalen Registry vorgeschlagen. Da sich alle Objekte dort registrieren müssen, muß es von jeder Klasse ein static-Objekt geben.

- Da alle Klassen static instanziiert werden müssen, dürfen die Konstruktoren nicht protected sein. In der ursprünglichen Implementation konnte so verhindert werden, daß auf andere Weise als vorgesehen Objekte der Klasse erzeugt werden. Bei einer Registry-Implementierung ist dies nicht möglich, und die Singleton-Eigenschaft kann nicht mehr vom Compiler überprüft werden!

- Die Instanziierung des static-Klassenattributs darf nur in einer Übersetzungseinheit vorkommen (d.h. darf nicht in der Header-Datei erfolgen, sondern muß in einer Implementations-Datei geschehen), damit es nicht doppelt definiert ist. Diese Distanz im Quelltext erhöht die Gefahr, die Instanziierung zu vergessen. Wenn die static-Instanziierung der Klasse vergessen wird, führt dies jedoch erst zur Laufzeit zu Fehlern.

Von diesen Implementierungs-Problemen inspiriert, stelle ich einige Forderungen an eine Singleton-Implementierung in C++ auf:

- Der Konstruktor einer Singleton-Klasse sollte protected (bzw. private) sein, um irrtümliche Instanziierungen zu verhindern.

- Man sollte von Singleton-Klassen erben und diese Unterklassen ebenfalls zu Singletons machen können.

- Der Anwendungsprogrammierer sollte sich um möglichst wenige technischen Details der Singleton-Implementierung (z.B. static-Instanziierungen) kümmern müssen.

In unserem Framework werden z.B. für Events und in der Zustandsmodellierung größere Singleton-Hierarchien verwendet. Wir setzen dafür eine modifizierte Version des von Douglas C. Schmidt in [Vlissides 96b] vorgeschlagenen Singleton-Template ein, bei der im Gegensatz zum Vorschlag von Vlissides nicht die zum Singleton zu machende Klasse vom Singleton-Template erbt, sondern umgekehrt.

```
template <class TYPE>
class ocSingleton : public TYPE
{
protected:
    ocSingleton<TYPE>() { };
    ocSingleton<TYPE>(const ocSingleton<TYPE> &) { };

public:
    static ocSingleton<TYPE> * poInstance(void);
};

template <class TYPE>
ocSingleton<TYPE> * ocSingleton<TYPE>::poInstance(void)
{
    static ocSingleton<TYPE> * _poinstance = 0;
    if (_poinstance == 0) {
        _poinstance = new ocSingleton<TYPE>;
    }
    return _poinstance;
}
```

Abbildung 34: Singleton-Implementierung als Template

Diese Implementierung erfüllt die drei postulierten Forderungen und erzeugt den konkreten Typ erst zur Compilezeit aus dem Template und der Anwendungsklasse. Dadurch können sich sowohl die Template-Implementation als auch die Anwendungsklasse getrennt voneinander entwickeln.

Der Anwendungsprogrammierer stellt lediglich sicher, daß der Konstruktor und Copy-Konstruktor seiner Klasse protected ist. (Der Zuweisungsoperator braucht nicht geschützt zu werden, da ohnehin kein Speicher für Objekte seiner Klasse alloziert werden kann.)

Dadurch entsteht folgende Vererbungsstruktur:

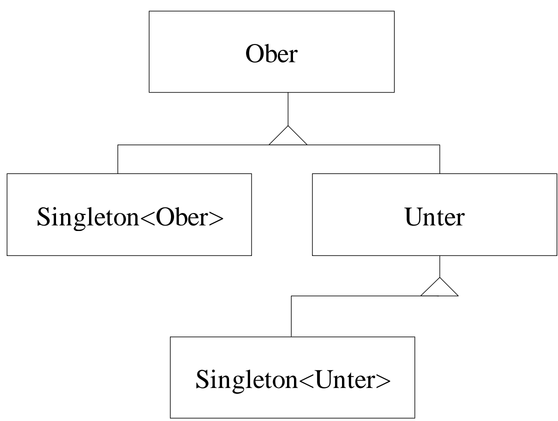

Abbildung 35: Vererbungshierarchie mit Singleton-Template

Da bei dieser Implementierung die Konstruktoren wieder geschützt sein können, können die Klassen selbst nicht irrtümlich verwendet werden, sondern nur in Verbindung mit dem Singleton-Template.

Da die per Template erzeugten Singletons nun aber nicht mehr Unterklassen voneinander sind, dürfen nur Variablen der Anwendungsklasse deklariert werden, wenn von Polymorphie Gebrauch gemacht werden soll. Dies entspricht aber auch der Anwendungssemantik. Das Template tritt nur beim Zugriff auf Singleton-Exemplare in Erscheinung.

Zusätzlich verwendet unser Singleton-Template eine static-Variable innerhalb der Erzeugungsmethode anstatt in der Klasse. Daher wird die bei allen [GHJV 95] Implementationen notwendige Initialisierung der static-Variable dem Programmierer abgenommen und kann im Template geschehen, da in C++ nur static-Variablen von Methoden, nicht aber von Klassen direkt bei ihrer Deklaration initialisiert werden können.

Ein wesentlicher Vorteil bei einer Template-Implementierung ist, daß gemäß dem Prinzip der "Separation of Concern" die Schnittstelle der Anwendungsklasse und ihre Implementierung getrennt sind von der Singleton-Implementierung. Auf diese Weise wird die Wiederverwendbarkeit und die Evolutionsfähigkeit erhöht.

5.4 Entwicklung der Event-Normalisierung

Die im Anwendungsbereich begründete Notwendigkeit, eine Event-Normalisierung (siehe Definition 21) durchzuführen, wurde bereits früh erkannt. Wie in Kapitel 3.5.4 beschrieben, reicht die Spezifikation der gängigen Telefonie-Protokolle nicht aus, um Tk-Anlagen-unabhängige Telefonie-Anwendungen zu entwickeln.

Um diesem Problem zu begegnen, verwendet das Telefonie-Framework eine sehr viel detailliertere Protokollbeschreibung. Bei der Entwicklung dieser Protokollbeschreibung wurde einerseits darauf geachtet, daß sie nicht im Widerspruch zu den üblichen Standards steht, andererseits jedoch in den für das MCC wichtigen Bereichen ausreicht, um den Anwendungs-

entwicklern genügend Zusicherungen über das Verhalten der Tk-Anlage zu geben, so daß sie völlig anlagenunabhängig entwickeln können.

Somit verhalten sich die Micrologica-TSERV-Prozesse völlig standard-konform, so daß Programme von Fremdanbietern verwendet werden können, und andererseits bieten sie soviel Zusicherungen, wie sonst oft nur mit einem proprietären Protokoll zu erreichen sind.

Um dieses Verhalten zu realisieren, ist es notwendig, in der virtuellen Tk-Anlage ein sehr detailliertes Modell des Tk-Anlagen-Zustands zu halten, so daß zusätzliche Events generiert und überflüssige Events verschluckt werden können. Die Interaktionsdiagramme in Abbildung 36 zeigen, wie unterschiedliche Eventfolgen verschiedener Tk-Anlagen normalisiert werden, so daß der Anwendung ein fast identischer Event-Strom zur Verfügung gestellt werden kann. Lediglich das Timing der Events ist leicht abweichend.

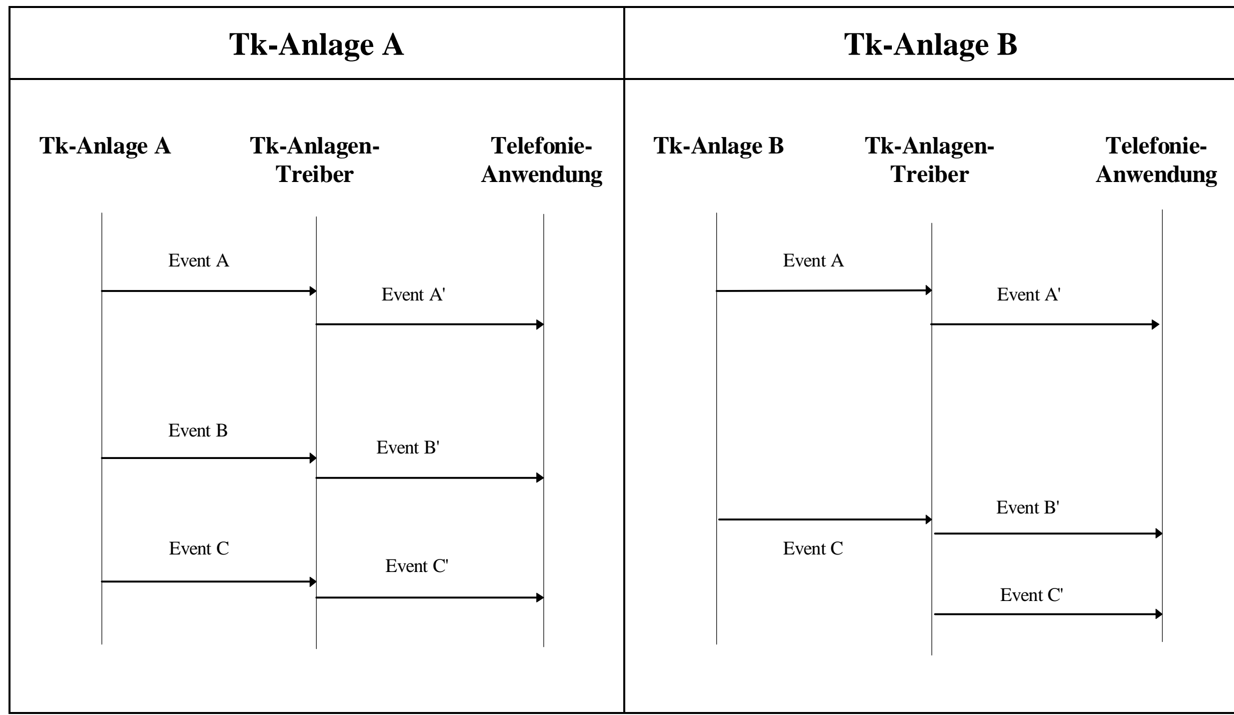

Abbildung 36: Event-Normalisierung durch den Tk-Anlagen-Treiber

In der ersten Version des Frameworks wurden die Events der Tk-Anlage dem jeweiligen Request zugeordnet, das sie ausgelöst hat. Dieses Request bzw. die darin enthaltene FSM bot den nötigen Kontext, um fehlende Events zu ergänzen und die richtige Reihenfolge herzustellen. Abbildung 37 zeigt den (vereinfachten) Ablauf einer Event-Normalisierung, bei der aus einem Event A einer konkreten Tk-Anlage zwei Events generiert werden.

Dabei wird folgender Ablauf verwendet (Formatkonvertierungen werden zur Vereinfachung außer acht gelassen):

Ein Event von der Tk-Anlage wird vom Event-Konverter an die virtuelle Tk-Anlage übergeben. Diese ordnet das Event dem Request-Objekt zu, das das Event ausgelöst hat. Entsprechend dem State-Muster übergibt das Request-Objekt das Event an seinen aktuellen

Zustand. Anhand des aktuellen Zustands werden dann mehrere Events generiert. Die normalisierten Events werden dann vom Telefonie-Protokoll-Konverter an die Anwendungen gemeldet.

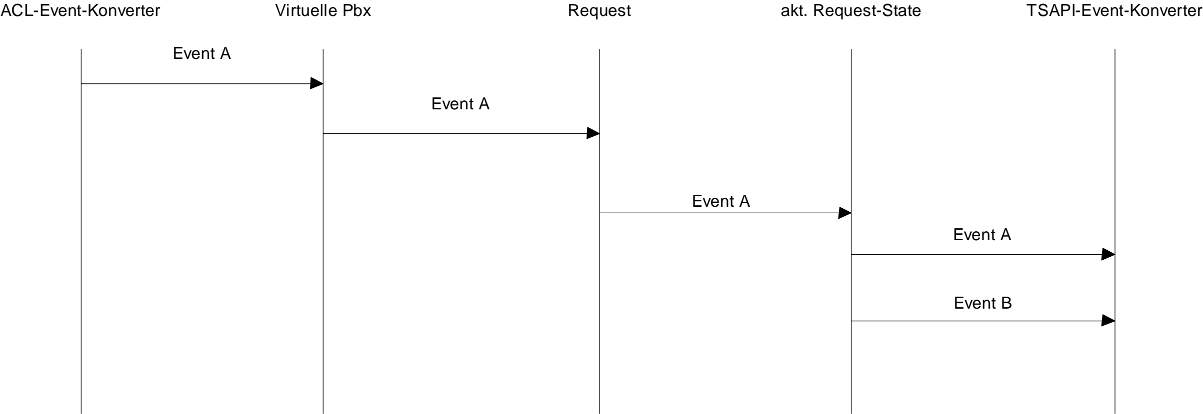

Abbildung 37: Erste Version der Event-Normalisierung

Erst einige Zeit nachdem dieser Ablauf implementiert und auch getestet war, fiel ein Denkfehler auf: Auf diese Weise lassen sich nur dann alle Events normalisieren, wenn alle Aktionen der Tk-Anlage durch Requests veranlaßt wurden. Bei den durchgeführten Tests ist dies auch stets der Fall gewesen, aber das Framework sollte auch in einer LAN-zentrierten Konfiguration eingesetzt werden, bei der keineswegs eine exklusive Steuerung der Tk-Anlage gewährleistet ist (vgl. Kapitel 3.5.2). Daher kann nicht davon ausgegangen werden, daß alle eingehenden Events von Requests des Telefonie-Frameworks ausgelöst wurden; sie können auch von anderen Benutzern der Tk-Anlage verursacht worden sein.

Um dieses Problem zu lösen, ist es notwendig, ein vollständiges Modell aller Aktivitäten der Tk-Anlage im Tk-Anlagen-Treiber vorzuhalten - unabhängig davon, wer sie mit welcher Intention ausgelöst hat. Daher wurde eine zusätzliche Instanz, der sog. Call-Manager, eingefügt, die den aktuellen Zustand der Tk-Anlage in Half-Call-Modellierung (siehe Kapitel 3.4.1) vorhält (sofern sie ihr über Events bekannt werden).

Die Normalisierung der Events wird nun von einem sog. Event-Normalisierer durchgeführt, der auf das Modell des Call-Managers zurückgreift und daraus die zu erzeugenden Events ableitet. Diese Art der Event-Normalisierung ist deutlich komplexer, insbesondere deshalb, weil die Intention nicht bekannt ist, warum bestimmte Verbindungen aufgebaut werden. Vorher war durch den Typ des Requests bekannt, was erreicht werden sollte. Die folgende Abbildung zeigt den neuen Ablauf:

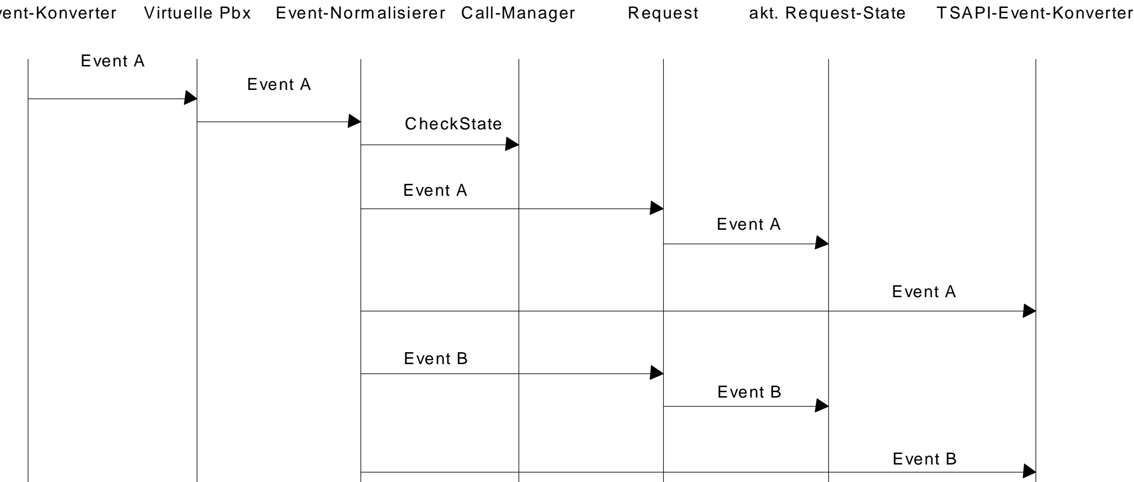

Abbildung 38: Zweite Version der Event-Normalisierung

Bei der Implementation des neuen Ablaufs hat es sich als sehr positiv erwiesen, daß der Bereich der virtuellen Tk-Anlage hinter einer Fassade verborgen war. Auf diese Weise sind die anderen Bereiche von dieser unerwarteten Änderung völlig unberührt geblieben. Es mußte lediglich der interne Ablauf in der virtuellen Tk-Anlage geändert werden.

Im nachhinein betrachtet, könnte man natürlich argumentieren, daß man die Unzulänglichkeit der ersten Implementation vorher hätte erkennen können und sollen. Andererseits ist diese Änderung das Ergebnis des Lernprozesses der an der Entwicklung Beteiligten, und man wird nie alle Probleme bei der ersten Analyse erkennen können. Daher macht dieses Beispiel deutlich, wie wichtig ein flexibles Design ist, wenn es darum geht, mit späteren Änderungen umzugehen.

Es ist bei der Framework-Entwicklung durchaus üblich, daß neue, wichtige Erfahrungen in dem jeweiligen Anwendungsbereich gemacht werden, da sie oftmals Anlaß für eine unabhängige, neutrale Betrachtung des Anwendungsgebietes bietet.

Obwohl sie in der Literatur kaum anzutreffen ist, möchte ich die These aufstellen, daß die Event-Normalisierung eine allgemein verwendbare Technik zur Hardware-Steuerung ist. Eine ähnliche Technik wird offenbar auch im "CallPath" Telefonie-System von IBM eingesetzt (siehe [Salamone 96]), über deren Implementation mir leider keine Publikationen bekannt sind. Es wäre ein lohnenswertes Unterfangen zu untersuchen, ob sich diese Technik weiter verallgemeinern läßt.

5.5 Evolution der Kern-Abstraktion

Die folgenden Abbildungen sollen die gedankliche Entwicklung der ersten Framework-Version verdeutlichen: Sie sind in dieser Form nicht implementiert worden und stellen nur die Evolution des Designs dar und nicht die Evolution einer Implementierung.

Der Grundgedanke, der bei Micrologica schon lange existierte, war, daß man den TSERV-Prozeß als Protokoll-Konverter betrachten müßte. Er konvertiert ein Protokoll A in ein Protokoll B, genauer: Er konvertiert ein Telefonie-Protokoll in ein Tk-Anlagen-Protokoll.

1. Stufe	**2. Stufe**	**3. Stufe**	**4. Stufe**
A + B	A B	A B	A B
Vermischung der Protokolle	Trennung der Protokolle	Austauschbare Protokoll-Komponenten	Protokoll-Komponenten als Adapter auf gemeinsamem Kern

Abbildung 39: Die ersten Entwicklungsstufen des Designs

Bei Micrologica waren daher Darstellungen in der Art von Stufe 2 (Abbildung 39) gebräuchlich, obwohl sich die vorhandene Implementation eher auf Stufe 1 befand. Die bisherigen TSERV-Implementationen haben beide Protokolle untrennbar miteinander verbunden. Sie sind jedoch bereits mit dem Gedanken der Protokoll-Konvertierung entwickelt worden.

Es hat sich für interne Diskussionen als sehr hilfreich erwiesen, auf diese Darstellung zurückzugreifen und die Design-Idee des neuen Frameworks graphisch als Evolutionsschritte aus der ursprünglichen Darstellung abzuleiten. Dieses Vorgehen vermittelte das Gefühl von Kontinuität und hob den Wert des existierenden fachlichen Konzepts hervor.

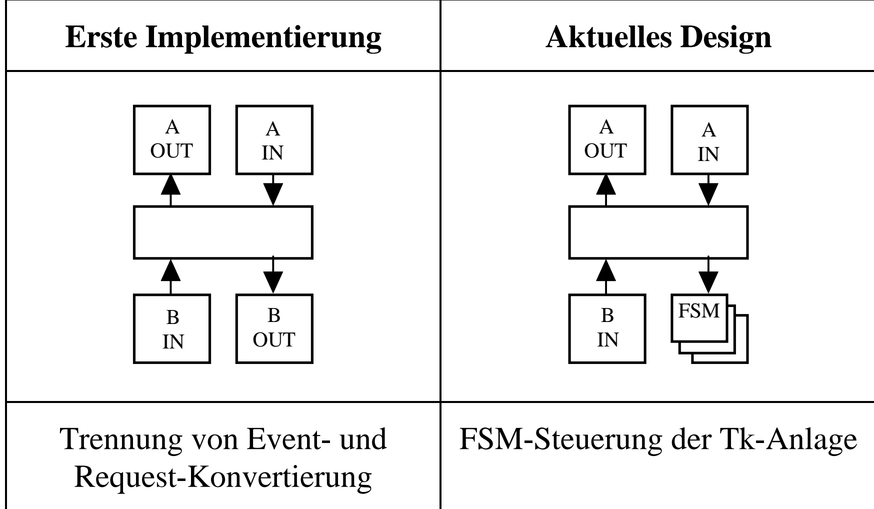

Abbildung 40: Implementierte Versionen

Bei der Implementierung stellte sich sehr schnell heraus, daß die Protokoll-Konverter zwei weitgehend getrennt Funktionen erfüllen: die Konvertierung von Events und die Konvertierung von Requests. Diese Teile sind dann auch getrennt implementiert worden.

Bei der Implementierung der Tk-Anlagen-Steuerung wurde jedoch festgestellt, daß diese nicht durch einen simplen Konverter zu bewerkstelligen ist, da sie in mehreren Tk-Anlagen-spezifischen Schritten abläuft, die Zugriff auf das interne Modell der Tk-Anlage benötigen (siehe Kapitel 3.4.1). Um diese Schritte einzeln spezialisieren zu können, wurde daher die Ablaufsteuerung im nächsten Evolutionsschritt mit Zustandsmaschinen modelliert. Die Trennung dieser Zustandsmaschinen in allgemeine und Tk-Anlagen-spezifische Teile ist bereits bei der Erklärung der Requests in Kapitel 4.3.2 dargestellt worden.

5.6 Zusammenfassung

Die in diesem Kapitel vorgestellten Entwicklungsschritte haben zu dem Konzept von lose gekoppelten Framework-Bereichen geführt, das in Kapitel 6 noch ausführlich diskutiert wird. Es hat sich gezeigt, daß die explizite Bildung von "Framework-Bereichen" dazu beigetragen hat, daß die Auswirkungen von Änderungen weitgehend gekapselt werden konnten.

Andererseits ist die Evolution des Telefonie-Frameworks natürlich auch ein Resultat des Einsatzkontextes. Der intensive Kontakt zu den Telefonie-Spezialisten hat sich ausgezahlt. Es sind wenige gravierende Änderungen aufgetreten, die die Grundannahmen, auf denen das Framework basiert, ins Wanken hätten bringen können.

6 Framework-Partitionierung

Der wohl wichtigste Einfluß auf die Evolution eines Frameworks ist unserer Ansicht nach die scharfe Abgrenzung des unterstützten Modells. Es muß von Anfang an explizit gemacht werden, auf welchen Annahmen das Framework basiert und was der Kern seines fachlichen Modells ist. Nur so ist eine Diskussion mit allen Beteiligten möglich, um soviel Erfahrung auf dem Anwendungsgebiet einzubringen wie möglich.

Die Abgrenzung des Aufgabenbereiches findet zum einen für das Framework als Ganzes statt. Zum anderen haben wir es aber auch als sehr hilfreich erfahren, das Framework in Teilbereiche aufzuteilen, für die ebenfalls genaue Aussagen getroffen werden, welche Abstraktionen sie repräsentieren sollen und auf welche Konzepte sie sich stützen.

Mit der Aufteilung einher geht die Strukturierung des Frameworks. Bei der Entwicklung des Micrologica-Telefonie-Frameworks haben wir eine Strukturierungs-Technik entwickelt, die wir als Partitionierung bezeichnen. Dadurch, daß einzelne fachliche Konzepte voneinander getrennt werden, werden Design-Entscheidungen über ihre Realisierung und ihre Beziehungen zueinander explizit gemacht und stehen der konstruktiven Kritik offen. Zusätzlich bringt die Strukturierung von Frameworks ähnliche Vorteile für die objektorientierte Programmierung wie die Modulbildung für die imperative Programmierung: Sie ermöglicht eine Reduzierung der Komplexität, unterstützt das Information Hiding und erlaubt es, Abstraktionsstufen auszudrücken.

Ich möchte daher zunächst noch einmal die Partitionen des Telefonie-Frameworks vorstellen und wie wir zu ihnen gekommen sind. Daraus abgeleitet stelle ich dann das Konzept eines Design-Zentrums vor, das bei der Abgrenzung des Aufgabenbereichs des Frameworks hilft.

Die Technik der Partitionierung wird dann allgemein diskutiert, und es werden Konstruktionstechniken erörtert, die helfen, unabhängige Partitionen zu bilden.

6.1 Die Partitionen des Telefonie-Frameworks

In Kapitel 4.3 habe ich bereits erwähnt, daß das Telefonie-Framework in einzelne "Bereiche" unterteilt ist. An dieser Stelle möchte ich nun auf die Gründe eingehen, die zu dieser Partitionierung geführt haben und anhand welcher Kriterien sie vorgenommen wurde.

Um zu Strukturierungskriterien zu kommen, haben wir uns an den in der Analyse erkannten Problemen der Telefonie orientiert (siehe Kapitel 3.5). Bei der Modellierung haben wir uns von den Gegenständen einer realen Telefonie-Konfiguration leiten lassen. Wir haben uns bemüht, das im Framework realisierte Modell den bisher verwendeten Begriffen und Konzepten so ähnlich wie möglich zu gestalten. Das zentrale Element ist daher eine "virtuelle" Tk-Anlage. Die verwendeten Protokolle, deren ständiger Wechsel erwartet wird, sind in eigenen Partitionen gekapselt. Die Funktion dieser Partitionen ist eindeutig festgelegt, und auch die Interaktion mit anderen Partitionen kann abstrakt beschrieben werden. Die konkrete Realisierung variiert. Diese Modellierung vergegenständlicht die Dinge, mit denen die Telefonie-Spezialisten bereits seit langem gearbeitet haben. Es war daher auch leicht möglich, über sie zu diskutieren und Mißverständnisse zu erkennen, bevor das vollständige Modell implementiert und lauffähig war. Auch bei der Wahl der Bezeichnungen haben wir uns (von

wenigen in der bisherigen Praxis existierenden Zweideutigkeiten abgesehen) an der bestehenden Fachsprache orientiert.

Obwohl bei Micrologica eine ganze Reihe von Telefonie-Anwendungen erstellt werden, haben wir das Einsatzgebiet des Frameworks auf die Erstellung von Tk-Anlagen-Treibern eingeschränkt. Die zunächst beabsichtigte Erstellung von generischen Komponenten, die in beliebigen Telefonie-Anwendungen verwendbar sind, wurde verworfen, da es sich zeigte, daß es nur bei einer klaren Festlegung des Einsatzkontextes möglich war, konkrete Aufgabenverteilungen und Kommunikationsbeziehungen festzulegen.

Bei der Entwicklung weiterer Frameworks werden möglicherweise Komponenten dieses Frameworks wiederverwendet oder verallgemeinert. Dies setzt aber auch wieder eine konkrete Abgrenzung des Einsatzkontextes voraus.

Über die Festlegung auf Tk-Anlagen-Treiber hinaus haben wir ausdrücklich einige Grundannahmen gemacht, die den Einsatzkontext weiter einschränken:

- es wird erwartet, daß in Zukunft immer mehr Tk-Anlagen mit CSTA-konformen oder zumindest CSTA-ähnlichen Telefonie-Konzepten arbeiten werden; die Kommunikation mit solchen Anlagen sollte daher einfach zu bewältigen sein, für anders geartete Anlagen wird eine aufwendigere Protokolladaption akzeptiert

- die Interprozeßkommunikation wird höchstwahrscheinlich auch in Zukunft über die PSI-Middleware stattfinden, so daß es hier nicht zwingend notwendig, ist eine völlig abstrakte Schnittstelle zu definieren

- die wichtigste Zielplattform ist Novell Netware, um aber eine möglichst weitgehende Plattformunabhängigkeit zu erreichen, werden nur portable C++ Funktionen verwendet. Alle weiteren Plattformabhängigkeiten sollen von der PSI-Middleware gekapselt werden.

Die Partitionierung erlaubt es, diese Designentscheidungen jeweils in einer eigenen Partition zu kapseln. Die Beziehungen der Partitionen drücken die Hierarchie der fachlichen Abstraktionen aus.

Abbildung 41 zeigt die vollständige Partitionierung des Telefonie-Frameworks sowie die Abhängigkeiten der Partitionen untereinander.

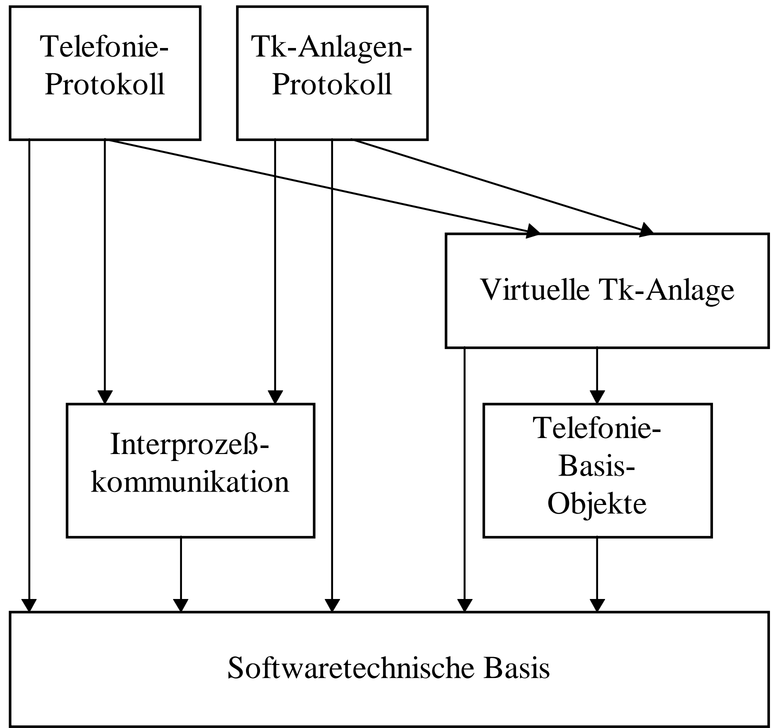

Abbildung 41: Die Partitionen des Telefonie-Frameworks

Zum einen sind durch diese Partitionierung die fachlichen Bereiche voneinander getrennt. Zum anderen wurde darüber hinaus jedoch ein besonderes Augenmerk auf die Abhängigkeiten der Partitionen voneinander gelegt. Sie sind stufenartig strukturiert und sind nur von Partitionen auf tieferen Stufen abhängig. Gegenseitige Beziehungen der Partitionen konnten dadurch beseitigt werden, daß die Kommunikation über Abstraktionen stattfindet, die auf tieferen Stufen angesiedelt sind.

Die Partitionen gruppieren die Klassen des Frameworks. Die logische Schnittstelle einer Partition ist jedoch reduziert und stellt nicht einfach alle Klassen der Partition zur Verfügung. Sie ist aufgeteilt in einen öffentlichen und einen privaten Teil. Die verwendete Programmiersprache C++ stellt jedoch nur unzureichende Mittel zur Verfügung, um diese Konstruktion auszudrücken. Daher wird die Trennung zwischen dem öffentlichen und dem privaten Teil der Partition bisher nur implizit per Konvention bzw. in der Dokumentation ausgedrückt. Für spätere Versionen wird angestrebt, die Klassendeklarationen der privaten Klassen soweit wie möglich den Benutzern der Partition vorzuenthalten.

Die Partition der virtuellen Tk-Anlage besteht beispielsweise aus ca. 15 Klassen. Davon ist jedoch nur eine Klasse öffentlich. Die interne Realisierung wird entsprechend dem Fassade-Muster verborgen. Bei anderen Partitionen besteht die öffentliche Schnittstelle aus mehreren Klassen, aber fast nie sind alle Klassen öffentlich.

Abbildung 42: Unterstrukturierung der softwaretechnischen Basis

Die Partitionen sind z.T. weiter unterstrukturiert. Die softwaretechnische Basis (siehe Abbildung 42) besteht u.a. aus Klassen-Teams, die Behälter oder ein Meta-Object-Protocol realisieren. Ebenso bestehen die Telefonie-Basis-Objekte (siehe Abbildung 43) aus einer Reihe von Basisabstraktionen aus dem Telefoniebereich: den Objekten des Half-Call-Modells (siehe Kapitel 3.4.2), den Events der Tk-Anlage und den Requests an die Tk-Anlage (siehe Kapitel 3.4.1).

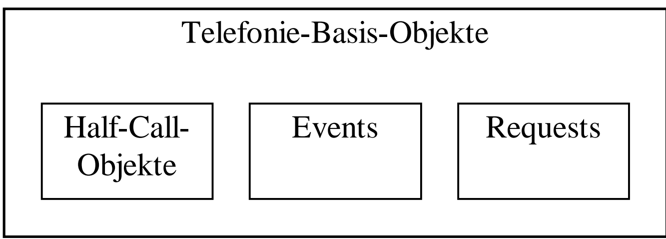

Abbildung 43: Unterstrukturierung der Telefonie-Basis-Objekte

Analog zur angestrebten Unabhängigkeit der Partitionen wird auch eine Trennung der unterschiedlichen Abstraktionen unterhalb der Partitionen angestrebt. Im nachhinein wird deutlich: Die Kriterien zur Partitionsabgrenzung sind keineswegs starr. Sie hängen von der Größe des zu strukturierenden Frameworks ab und von der Komplexität der einzelnen fachlichen Bereiche. Innerhalb eines größeren, umfassenderen Frameworks könnten die jetzigen Partitionen die Unterstruktur einer größeren Partition bilden.

Die von uns gewählte Partitionierung ist nicht zufällig robust gegenüber Änderungsanforderungen: Da sie sich an den realen Telefonie-Komponenten orientiert, konnte die langjährige Erfahrung der Micrologica-Mitarbeiter darauf übertragen werden. Die Erfahrungen über die Interaktion dieser Komponenten und die potentiellen Reibungspunkte waren zumindest leichter auf unser Modell übertragbar als auf ein völlig neues Modell, das stärker von den realen Gegenständen abweicht.

Diese Aufteilung hat sich als äußerst förderlich für die Evolutionsfähigkeit des Frameworks erwiesen. Da die Partitionen weitgehend voneinander entkoppelt sind, waren die meisten bisherigen Evolutionsschritte lokal innerhalb einer Partition.

6.2 Abgrenzung des Anwendungsbereichs für ein Framework

Um ein Framework auf seine Evolution vorzubereiten, muß man sich festlegen, mit welchen Arten von Änderungen man bereit wäre zu leben und was die Kern-Abstraktionen sind, auf die sich das Framework verläßt.

Dies mag zunächst selbstverständlich erscheinen. In der Praxis handelt es sich dabei um eine schwierige Gratwanderung: Bei einer exakten Festlegung des zu unterstützenden Bereichs kann die Unterstützung sehr spezifisch ausfallen. Je weiter der Bereich gewählt wird, desto generischer und allgemeiner wird die Unterstützung.

Andererseits sind Frameworks dazu gedacht, eine Reihe von Anwendungen zu unterstützen, so daß der abzudeckende Bereich nicht zu eng gewählt werden darf, denn schließlich muß der ökonomische Aufwand einer Framework-Entwicklung gerechtfertigt werden. Je mehr Anwendungen mit dem Framework realisiert werden können, desto leichter ist dies zu erreichen.

Im Telefonie-Framework wurde beispielsweise zunächst bewußt auf die Unterstützung anderer Bereiche außerhalb der konkreten Tk-Anlagen-Steuerung verzichtet, um diesen Bereich mit sehr konkreten Abstraktionen zu unterstützen.

Eine Entscheidung über den Fokus des Frameworks kann nur im konkreten Fall gefällt werden. Es hat sich jedoch als hilfreich erwiesen, die getroffenen Annahmen über die Art der zu unterstützenden Anwendungen explizit zu machen. Einem Außenstehenden (auch in der gleichen Firma) ist es oft nicht leicht ersichtlich, ob das Framework sein konkretes Problem unterstützen kann. Daher sollte dokumentiert sein, welche Grundannahmen vorliegen und was anpaßbar ist.

In [MätBisch 96] wird der Begriff *Design-Zentrum* verwendet, um die einem Framework zugrundeliegende Kern-Abstraktion zu beschreiben.

Definition 25: Design-Zentrum

Die zentrale Design-Abstraktion eines Frameworks wird als Design-Zentrum (engl. design center) bezeichnet. Es verkörpert die wichtigsten Entwurfsentscheidungen des Frameworks.

Bei der Evolution eines Frameworks muß das Design-Zentrum immer intakt bleiben. Sobald das Design-Zentrum eines Frameworks verändert werden muß, besteht die Gefahr von massiven Änderungen in allen auf dem Framework basierenden Anwendungen, da sich diese auf die vom Design-Zentrum verkörperte Grundannahme verlassen haben.

Bei der Konstruktion eines Frameworks kann Vorsorge getroffen werden, um das Design-Zentrum möglichst frei zu halten von den Änderungen in seiner Umgebung. Durch eine lose Kopplung mit seiner Umgebung werden "Sollbruchstellen" angelegt. Es bietet sich an, das Design-Zentrum in einer eigenen Partition zu realisieren.

Für die Evolution eines Frameworks bedeutet lose Kopplung eine Begrenzung der Auswirkungen von Änderungen. Zum einen ist die Schnittstelle zwischen den Partitionen schmaler, so daß bei vielen Änderungen keine anderen Partitionen tangiert werden. Zum anderen erlaubt

eine lose Kopplung den Austausch von Partitionen, z.B. weil sie aufgrund der veränderten Anforderungen völlig neu geschrieben werden mußten.

6.2.1 Das Design-Zentrum des Telefonie-Frameworks

Im Telefonie-Framework repräsentiert die Partition "virtuelle Tk-Anlage" das Design-Zentrum. Die Annahme, daß alle Tk-Anlagen von Requests gesteuert werden und die Anlagen über Events ihren Zustand melden, ist für unser Modell zentral. Auch die Kommunikation der virtuellen Tk-Anlage mit ihrer Umwelt ist Teil des Design-Zentrums, denn sie entspricht unseren Annahmen über die Zusammenarbeit der virtuellen Tk-Anlage mit "Nachrichten-empfängern", über die mit den Anwendungen und der realen Tk-Anlage kommuniziert wird.

Um eine lose Kopplung des Design-Zentrums zu erreichen, wird die gesamte Partition von einer Fassaden-Klasse `ocVirtualPbx` repräsentiert (vergl. [GHJV 95], S. 185ff).

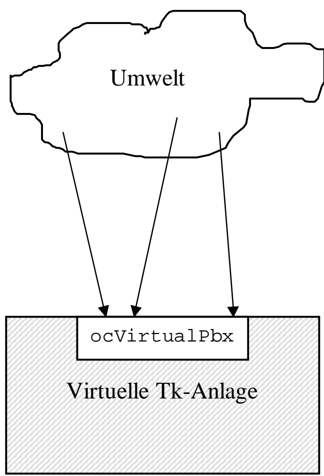

Abbildung 44: Funktion des Fassade

Über den Einsatz einer Fassaden-Klasse hinaus ist aber auch die Gestaltung der Schnittstelle dieser Fassaden-Klasse wichtig: In einer ersten Modellierung hatte sie etwa folgende Schnittstelle:

```
class ocVirtualPbx {
public:
      void vHandleMake(ocMakeCall * poMakeCall);
      void vHandleAnswerCall(ocAnswerCall * poMakeCall);
      void vHandleTransferCall(ocTransferCall * poMakeCall);
      // ...
};
```

Abbildung 45: Zu feste Kopplung der virtuellen Tk-Anlage

Diese Art der Modellierung hat zur Folge, daß für jeden neuen Request-Typ die Schnittstelle der virtuellen Tk-Anlage angepaßt werden müßte. Im Hinblick auf die schnell fortschreitende Entwicklung auf dem Markt für Tk-Anlagen wäre diese Art der Evolution sicherlich sehr unerfreulich.

Daher wurden die Requests in einer mehrstufigen Vererbungshierarchie implementiert (siehe Abschnitt 4.3.2). Diese Hierarchie macht der virtuellen Tk-Anlage in einer Oberklasse ocRequest genau diejenigen Eigenschaften zugänglich, die sie benötigt (hauptsächlich: Starten der Abarbeitung).

```
class ocVirtualPbx {
public:
 void vHandleRequest(ocRequest * poReq);

 // ...
};
```

ocRequest

ocMake Call | ocAnswer Call | ocTransfer Call

Abbildung 46: Lose Kopplung der virtuellen Tk-Anlage

Auf diese Weise können weitere Request-Typen hinzukommen, ohne daß die virtuelle Tk-Anlage geändert werden muß. Andererseits wäre die virtuelle Tk-Anlage auch leicht auswechselbar, wenn aus unerwarteten Gründen die bisherige Implementation nicht mehr den Anforderungen genügt. Die neue Implementation müßte lediglich diese minimale Schnittstelle weiter unterstützen.

Die lose Kopplung hat aber auch noch eine weitere, indirekte Folge: Man stelle sich vor, daß die virtuelle Tk-Anlage das Request nach der Abarbeitung zurückgibt. Sie könnte dies nur unter der ihr bekannten (Ober-)Klasse tun. Es wäre das Problem des Empfängers, den dynamischen Typ des ihm übergebenen Objekts herauszufinden, wenn er auf speziellere Eigenschaften zugreifen möchte. Auf die Implikationen des hierfür notwendigen Meta-Object-Protocol wurde bereits in Kapitel 5.2 ausführlich eingegangen.

6.3 Notwendigkeit der Framework-Strukturierung

Nachdem in diesem Kapitel die Partitionierung bisher nur auf das Telefonie-Framework bezogen wurde, möchte ich nun allgemeiner die grundsätzliche Notwendigkeit diskutieren, Frameworks zu strukturieren. Danach werde ich den bisher informell verwendeten Begriff der Framework-Partition definieren und Entwurfskriterien zur Partitionierung erörtern.

Ein wichtiger Einflußfaktor für die Framework-Evolution resultiert aus der internen Aufteilung des Frameworks. Denn obwohl ein Framework zunächst als geschlossene Einheit erscheint, gibt es gewichtige Gründe, Frameworks weiter zu untergliedern:

- Begrenzung der Auswirkung von Änderungen

  Um mit ungerichteter Evolution umgehen zu können, ist es sehr hilfreich, "Sollbruchstellen" zu haben, die die Auswirkungen von Änderungen begrenzen. Dies war im Telefonie-Framework eine der Hauptintentionen zur Partitionierung.

- Handhabbarkeit

  Wie bei allen nicht-trivialen Softwaresystemen ist es auch bei der Entwicklung und Pflege von Frameworks allgemein sehr hilfreich, wenn sie hinreichend fein strukturiert sind. Große Frameworks werden hierdurch überhaupt erst handhabbar.

- Kombinierbarkeit von verschiedenen Frameworks

  Oft haben Framework-Anwender das Bedürfnis, die Funktionalität von mehreren unabhängig voneinander entwickelten Frameworks zu benutzen. Neben anderen Kriterien ist eine klare Abgrenzung der einzelnen Funktionsbereiche innerhalb des Frameworks eine Grundvoraussetzung hierfür. Dies wird für das Telefonie-Framework erst dann relevant, wenn es z.B. zusammen mit einem GUI-Framework eingesetzt werden soll.

- Reduzierung des Overhead

  Je mächtiger ein Framework ist, desto größer ist auch die Wahrscheinlichkeit, daß es Funktionalität enthält, die von vielen Anwendungen nicht benötigt wird. Es wäre wünschenswert, diese Anwendungen nicht unnötig aufzublähen. Daher sollten einzelne Bereiche des Frameworks einzeln verwendbar sein, ohne daß ihre Referenzen zu anderen Teilen des Frameworks dazu führen, daß das gesamte Framework eingebunden werden muß.

Die Ähnlichkeit dieser Forderungen zur Modularisierung bei der imperativen Programmierung ist offensichtlich. Die meisten objektorientierten Sprachen bieten jedoch kein Ausdrucksmittel, um eine Modularisierung oberhalb der Ebene von Klassen auszudrücken.

Hierzu möchte ich im folgenden zwei Aspekte erörtern:

1. die Aufteilung in einzelne Bereiche: die Partitionierung

2. die Beziehungen der Bereiche untereinander: die Strukturierung

6.3.1 Framework-Partitionierung

Bei der Beschreibung des Micrologica-Telefonie-Frameworks habe ich allgemein von "Bereichen" des Frameworks gesprochen. Als Bezeichnung für die "Bereiche" eines Frameworks möchte ich den Begriff "Partitionen" etablieren, wie zu Beginn dieses Kapitels angedeutet.

Der Begriff "Partition" ist weniger mißverständlich als der Begriff "Layer", der in der Literatur des öfteren in ähnlicher Bedeutung verwendet wird (z.B. [BGKLRZ 97], [GHJV 95, S. 28]). Seit Dijkstra bezeichnen Layer eine strenge Schichtenarchitektur, bei der Kommunikation ausschließlich mit angrenzenden Schichten möglich ist (vgl. [Dijkstra 68]). Auch das bekannte ISO 7-Schichten-Modell beinhaltet beispielsweise diese Annahme.

Diese strenge Trennung ist bei Frameworks meist nicht zu erreichen und auch nicht notwendig. Die höhere Flexibilität gegenüber einer reinen Schichten-Architektur ist gerade eine der Stärken von Frameworks. Eine reine Schichten-Architektur hat kein Ausdrucksmittel, um schichtenübergreifende Gemeinsamkeiten auszudrücken (vgl. [Campbell 92]). Um eine Austauschbarkeit zu gewährleisten, sollten die Partitionen untereinander jedoch lose gekoppelt sein.

Definition 26: Framework-Partition

Eine Framework-Partition enthält das fachliche und technische Design eines Teilbereiches des Framework-Designs. Sie dient der Modularisierung des Frameworks. Ähnlich wie ein Framework wird sie von Klienten als Ganzes verwendet, bezieht sich aber auf andere Partitionen und ist nicht allein einsetzbar.

Es stellt sich jedoch die Frage, worin sich eine Partition von einem Framework unterscheidet. Tatsächlich wird in der Literatur oft von Frameworks gesprochen, die wieder aus diversen Frameworks bestehen. Diese Begriffsbildung führt jedoch zu einer inflationären Verwendung des Begriffs Framework, so daß kaum noch ein Unterschied zum Klassen-Team besteht. Ganz wesentlich für ein Framework ist aber das fachlich abgeschlossene Modell. Während Partitionen nur ein Teilmodell enthalten und nur in Kombination mit anderen Partitionen einsetzbar werden, sind Frameworks eigenständig verwendbar! Gelegentlich enthalten Partitionen so wenig vom fachlichen Modell, daß sie einzeln verwendbar sind. Dann haben sie allerdings nicht die Funktion eines Frameworks, sondern eher die Funktion einer Bibliothek. Im Telefonie-Framework betrifft dies beispielsweise die softwaretechnische Partition.

Die ursprüngliche Definition von Frameworks muß zur Abgrenzung daher wie folgt erweitert werden:

Definition 27: Framework (endgültige Fassung)

Ein Framework stellt ein generisches Design zu einer Reihe verwandter Probleme zur Verfügung. Es enthält bereits einen Teil der Implementation, muß für konkrete Anwendungen aber noch spezialisiert bzw. parametrisiert werden. Die Anwendung verwendet das Framework als Ganzes, so daß auch der Kontrollfluß der Anwendung vom Framework vorgegeben wird.
Das Framework enthält ein fachlich abgeschlossenes Modell und ist eigenständig verwendbar. Es benötigt zusätzlich höchstens eine softwaretechnische Unterstützung.

Die Framework-Partitionierung weist viele Parallelen zur Modularisierung bei imperativer Programmierung auf. Den meisten objektorientierten Sprachen fehlt es jedoch an Ausdrucksmitteln für eine Modularisierung in größeren Einheiten als Klassen. Dies ist besonders deshalb schmerzlich, weil es daher kaum Möglichkeiten gibt, technisch die Schnittstelle einzuschränken, die eine Partition nach außen anbietet. Wünschenswert wäre es, zwischen öffentlichen Schnittstellen-Klassen und privaten Implementierungs-Klassen unterscheiden zu können.

Das Cluster-Konzept von Eiffel unterstützt die Modularisierung hauptsächlich auf konzeptioneller Ebene. Die Cluster werden auf Verzeichnisse im Dateisystem abgebildet. Eine Möglichkeit, um die Schnittstelle eines Clusters zu definieren, gibt es nicht.

C++ hat überhaupt keine Mittel, um die Partitionierung auszudrücken. Sie muß daher mit Programmierkonventionen ausgedrückt werden. Die Einschränkung der Partitionsschnittstelle kann nur durch Kommentare und sonstige Dokumentation erfolgen.

Eine Ausnahme bildet Java: Hier gibt es das Package-Konzept, das es ermöglicht, jede Klasse einem Package zuzuweisen und zu definieren, ob die Klasse nur innerhalb dieses Packages

sichtbar sein soll oder ob sie auch von außen zugreifbar ist. Auf diese Weise ist es möglich, die Schnittstelle von Packages durch die Sprachsyntax auszudrücken und abzusichern.

In Programmiersprachen, die diese Möglichkeit nicht bieten, sollte trotzdem die Unterteilung in öffentliche und private Klassen beibehalten werden. Diese muß dann durch Programmierkonventionen unterstützt werden.

6.3.2 Framework-Strukturierung

Ein besonderes Augenmerk sollte den Beziehungen der Framework-Partitionen untereinander gewidmet werden. Dabei sind sowohl Benutzungs- als auch Vererbungsbeziehungen zu beachten.

Diese Beziehungen möchte ich als Framework-Strukturierung bezeichnen.

Definition 28: Framework-Struktur

Die Beziehungen der Framework-Partitionen untereinander bezeichnet man als Framework-Struktur.

Um die Evolutionsfähigkeit eines Frameworks zu gewährleisten, sollte eine Framework-Struktur angestrebt werden, bei der die Partitionen stufenartig aufeinander aufbauen. Durch diese Stufenbildung (engl. levelization) wird erreicht, daß keine zyklischen Beziehungen zwischen den Partitionen entstehen, die die getrennte Weiterentwicklung behindern würden.

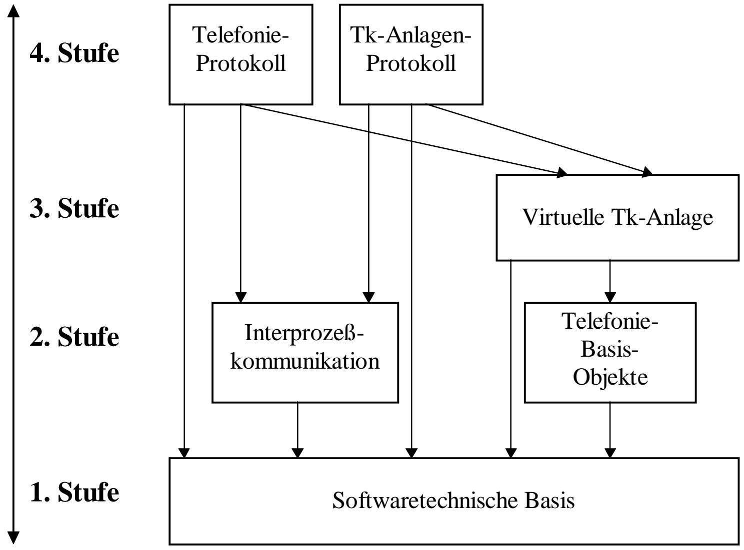

Abbildung 47: Bildung der Framework-Struktur durch vertikale Partitionierung

Um eine Stufenbildung zu ermöglichen, müssen die Partitionen entsprechend gebildet werden. Abbildung 47 zeigt die vertikale Partitionierung des Micrologica-Telefonie-Frameworks. Zur Partitionierung wurden Konzepte unterschiedlicher Abstraktionsniveaus verschiedenen Stufen zugeordnet. Fachlich unabhängige Abstraktionen wurden immer getrennt, auch wenn sie sich auf ein und derselben Stufe befinden.

So wurden das Telefonie-Protokoll und das Tk-Anlagen-Protokoll auf eine höhere Stufe gestellt als die Telefonie-Abstraktionen ("Telefonie-Basis-Objekte" genannt), auf denen beide basieren (vertikale Partitionierung).

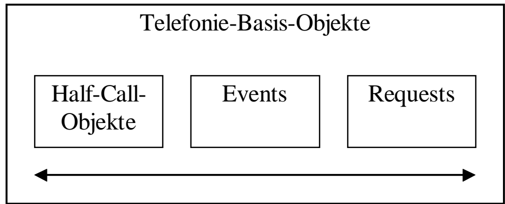

Abbildung 48: Horizontale Partitionierung der Telefonie-Basis-Objekte

Auch innerhalb von Partitionen lassen sich weitere Partitionen finden. Abbildung 48 zeigt die horizontale Partitionierung der Telefonie-Basis-Objekte. Diese Konzepte stehen auf gleichem Abstraktionsniveau nebeneinander und sind nicht voneinander abhängig.

Definition 29: Horizontale Partitionierung

Die Aufteilung in Partitionen ähnlichen Abstraktionsgrads, die nicht aufeinander aufbauen, wird als horizontale Partitionierung bezeichnet.

Definition 30: Vertikale Partitionierung

Die Aufteilung in Partitionen, die aufeinander aufbauen, wird als vertikale Partitionierung bezeichnet.

Zur Darstellung der Abstraktionen einer Partition dürfen nur Konzepte aus der Partition selbst und aus darunter liegenden Stufen verwendet werden, nicht aber aus anderen Partitionen auf derselben Stufe oder aus höheren Stufen.

Definition 31: Stufenbildung

Unter Stufenbildung verstehen wir die Zuordnung der Framework-Partitionen zu aufeinander aufbauenden Gruppen. Abhängigkeiten dürfen jeweils nur zu Partitionen aus darunter befindlichen Stufen existieren.

Auf fachlicher Ebene sind die vertikale Partitionierung und Stufenbildung als Bildung von Abstraktionsebenen zu interpretieren, während auf technischer Ebene die Kopplung des Systems im Vordergrund steht. Es ist wichtig, diese Stufenbildung nicht nur im Design, sondern auch in der Implementation konsequent durchzuhalten. Auf die bei der technischen Realisierung auftretenden Probleme und den Umgang mit ungewollten Beziehungen zwischen Partitionen geht der folgende Abschnitt ein.

6.4 Konstruktionstechniken für Partitionen

Die Framework-Partitionierung entsteht, wie das ganze Framework, in einem evolutionären Prozeß. Dabei kommt es im Laufe des Entwicklungszyklus, z.B. durch Erweiterungen der Funktionalität oder Aufsplittung von Partitionen, oft dazu, daß Partitionen mit gegenseitigen Abhängigkeiten entstehen. Dadurch werden die aufgestellten Kriterien zur Framework-Strukturierung verletzt.

Die gegenseitigen Abhängigkeiten behindern die Stufenbildung und führen zu einer starken Kopplung innerhalb des Frameworks. In diesem Abschnitt möchte ich daher einige Konstruktionstechniken vorstellen, um diese gegenseitigen Abhängigkeiten aufzulösen.

Die Bildung von Partitionen stellt ein enges Zusammenspiel von fachlichen und strukturell begründeten Gesichtspunkten dar. Die Abgrenzung der Partitionen muß (siehe oben) nach fachlichen Gesichtspunkten durchgeführt werden. Andererseits gibt es strenge strukturelle Anforderungen an die Abhängigkeit der Partitionen untereinander.

Man muß sich, insbesondere wenn es wie hier um die Auflösung von Abhängigkeiten geht, vergegenwärtigen, daß Software-Design ein gestaltgebender Vorgang ist. Es gibt nicht *das* fachliche Modell, sondern es gibt viele mögliche Modelle. Welches Modell gewählt wird, hängt von vielen Faktoren ab - unter anderem auch von strukturellen Gesichtspunkten des aus dem fachlichen Modell folgenden technischen Modells.

Wenn man vor dem Problem steht, Abhängigkeiten zwischen Partitionen aufzulösen, kann es daher hilfreich sein, sich zu vergegenwärtigen, welche strukturellen Umformungsmöglichkeiten bestehen, um die Abhängigkeiten aufzuheben und welche Folgen diese für das fachliche Modell haben. Erst dann kann man sich entscheiden, welche Modellierung zu wählen ist.

Ich möchte nun einige Techniken vorstellen, die von Lakos für die Modulbildung vorgeschlagen wurden (vgl. [Lakos 96, Kapitel 5]), und diese auf die Entkopplung von Partitionen übertragen. Um die Diskussion anschaulich zu halten, werde ich im folgenden nur die Abhängigkeiten zwischen Klassen betrachten - die Abhängigkeit von Partitionen folgt aus den Abhängigkeiten der enthaltenen Klassen.

> **Definition 32: Eskalierung**
>
> Das Anheben gegenseitig verwendeter Funktionalität bei Klassen mit zyklischen Beziehungen wird als Eskalierung (engl. escalation) bezeichnet. Der von der jeweils anderen Klasse benötigte Teil wird extrahiert und auf einer höheren Stufe angesiedelt.

Abbildung 49 zeigt zwei voneinander abhängige Klassen (die Pfeile bedeuten hier eine allgemeine "ist angewiesen auf"-Beziehung) und wie sie mittels Eskalierung voneinander getrennt werden könnten.

Um eine Eskalierung durchzuführen, werden die Klassen in einen unabhängigen (x_{frei} und y_{frei}) und einen von der anderen Klasse abhängigen Teil (x_{abh} und y_{abh}) aufgespalten. Falls nach der Aufteilung x_{abh} und y_{abh} weiterhin voneinander abhängen, so können sie zu einer neuen Klasse z zusammengefaßt werden, wenn sich dies fachlich anbietet. Anderenfalls müssen sie analog zum ersten Schritt erneut aufgeteilt werden. Falls man bei diesem Vorgehen zwei gegenseitig abhängige Klassen behält, ist die Eskalation gescheitert bzw. nicht möglich.

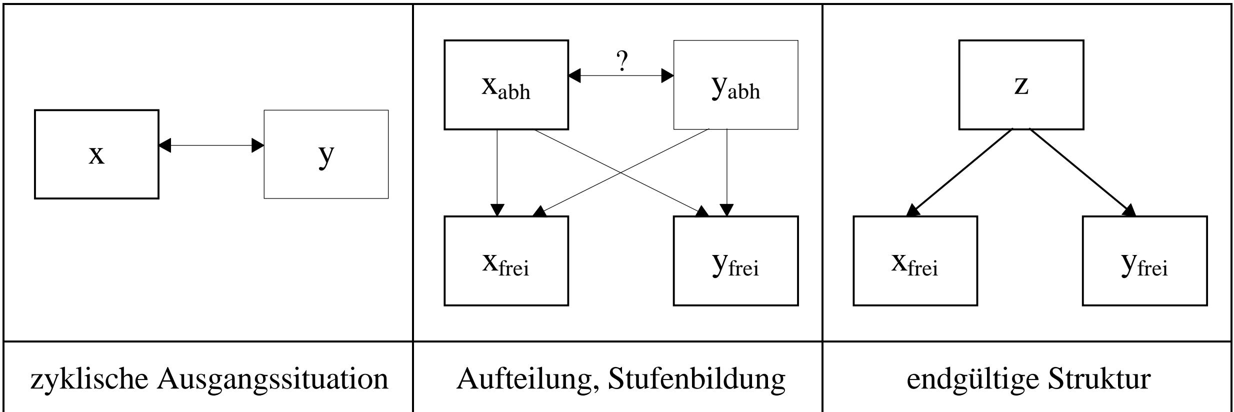

Abbildung 49: Eskalierung

Die Technik der Eskalierung bietet sich an, wenn nur ein kleiner Teil der Funktionalität der einen Klasse auf die andere Klasse angewiesen ist. Wenn x und y beispielsweise Klassen sind, die jeweils einen Konstruktor oder Konvertierungs-Operator füreinander haben, aber sonst voneinander unabhängig sind, bietet es sich an, die Konvertierung aus den Klassen zu entfernen und als neue Klasse z zu implementieren.

Definition 33: Degradierung

Das Absenken gegenseitig benutzter Funktionalität bei Klassen mit zyklischen Beziehungen wird als Degradierung (engl. demotion) bezeichnet. Der von der jeweils anderen Klasse benötigte Teil wird extrahiert und auf einer niedrigeren Stufe angesiedelt.

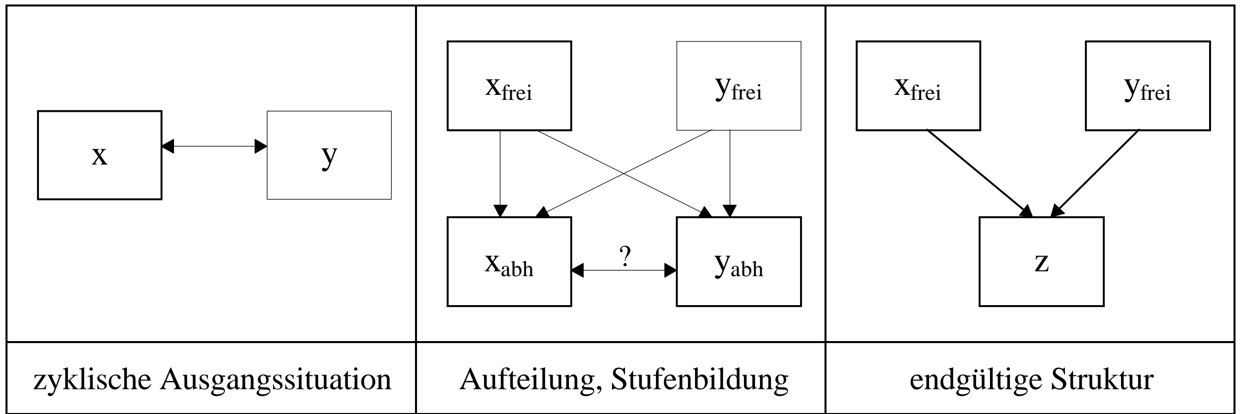

Abbildung 50: Degradierung

Zur Degradierung werden die Klassen analog zur Eskalierung aufgespalten. Sollten weiterhin Abhängigkeiten zwischen x_{abh} und y_{abh} bestehen, könnten beide zusammengefaßt oder erneut aufgespalten werden.

Die Technik der Degradierung bietet sich beispielsweise an, um gemeinsame Kommunikationsprotokolle oder Basisfunktionalität aus Klassen zu extrahieren.

Definition 34: Redundanz

Die Mehrfach-Implementierung eines (möglichst kleinen) Teils der Funktionalität wird als Redundanz bezeichnet. Sie dient dazu, die gegenseitige Abhängigkeit zwischen Klassen aufzubrechen.

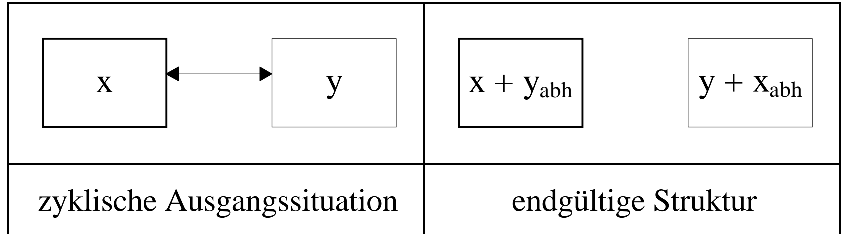

Abbildung 51: Redundanz

Eine redundante Implementierung trennt die Klassen völlig, indem der von der jeweils anderen Klasse benötigte Teil doppelt vorhanden ist. Hier wird zwar eine optimale Trennung erreicht, jedoch zum Preis eines verdoppelten Wartungsaufwandes, falls am redundanten Teil Änderungen notwendig werden. Diese Technik sollte daher mit Vorsicht und nur dann eingesetzt werden, wenn sehr kleine Teile redundant implementiert werden müssen.

Während man zyklische Abhängigkeiten zwischen Klassen-Teams im Ausnahmefall noch akzeptieren kann, sind sie zwischen Partitionen auf jeden Fall zu vermeiden.

Bei dieser Betrachtung der Framework-Strukturierung darf man die fachliche Motivation nicht aus den Augen verlieren und Klassen nur danach in Partitionen einteilen, wie ihre strukturellen Abhängigkeiten beschaffen sind. Wenn sich in einem Framework keine azyklische Partitionierung finden läßt, die den fachlichen Gegebenheiten entspricht, so ist dies ein starkes Indiz, daß die fachliche Modellierung überarbeitet werden sollte.

Die Wahl, welche Technik zum Aufbrechen von zyklischen Beziehungen verwendet wird (Eskalierung, Degradierung oder Redundanz), muß durch die fachliche Modellierung entschieden werden. Diese Techniken dürfen nicht Selbstzweck sein um zyklen- und sinn-freie Strukturen zu erzeugen, sondern müssen die fachliche Modellierung unterstützen.

Die Unabhängigkeit der Partitionen ist, über die hier dargestellten Gründe der Evolutionsfähigkeit hinaus, auch für andere Eigenschaften des Frameworks wichtig. Sie ermöglicht beispielsweise das getrennte Testen einzelner Partitionen oder erleichtert die arbeitsteilige Implementation des Frameworks. Für den Framework-Anwender wird durch die Dekomposition in kleinere Einheiten die Erlernbarkeit gefördert.

6.5 Partitionierungskriterien

Es gibt einige Ansätze für die Aufteilung von objektorientierten Systemen. Beispielsweise bietet die 3-Ebenen-Architektur (engl. three-tier architecture) Kriterien, um Anwendungen in drei Komponenten zu zergliedern: Front-End (Darstellung und Interaktion), fachliches Modell und Datenspeicherung (siehe [Hirschfeld 97]). Diese Aufteilung ist jedoch rein technisch motiviert, also unabhängig von der Komplexität der jeweiligen Bereiche in einer konkreten Anwendung. Sie skaliert nicht mit dem Umfang der Anwendung, sondern bietet immer nur drei Ebenen. Für die Strukturierung werden keine Kriterien angegeben.

Meiner Ansicht nach hängen die Kriterien für die Partitionsbildung sehr stark vom zu unterstützenden Anwendungsbereich ab und sollten aus dem fachlichen Modell abgeleitet werden. Ein weiteres Beispiel soll dies verdeutlichen:

Im Telefonie-Framework spielte bisher die Hardware-Konfiguration eine entscheidende Rolle und mit ihr technische Abwägungen über die zu unterstützenden Protokolle und Hardware. Dies spiegelt sich auch in der von uns gewählten Partitionierung wider. Sobald direkt Aufgabenbereiche von Benutzern unterstützt werden[16], böte sich eine Strukturierung anhand dieser Aufgabenbereiche an.

Wenn beispielsweise das Telefonie-Framework um weitere Funktionsbereiche erweitert wird, böte es sich an, die Framework-Partitionen an den Haupteinsatzbereichen von Telefonie-Systemen (Inbound- und Outbound-Geschäft) zu orientieren. Abbildung 52 zeigt eine mögliche Partitionierung eines umfassenderen Frameworks. Die bisher realisierte Funktionalität des Tk-Anlagen-Treibers wäre dann eine Partition unter vielen.

[16] Diese Aufgabenbereiche werden bereits heute unterstützt. Sie sind jedoch nicht frameworkbasiert realisiert.

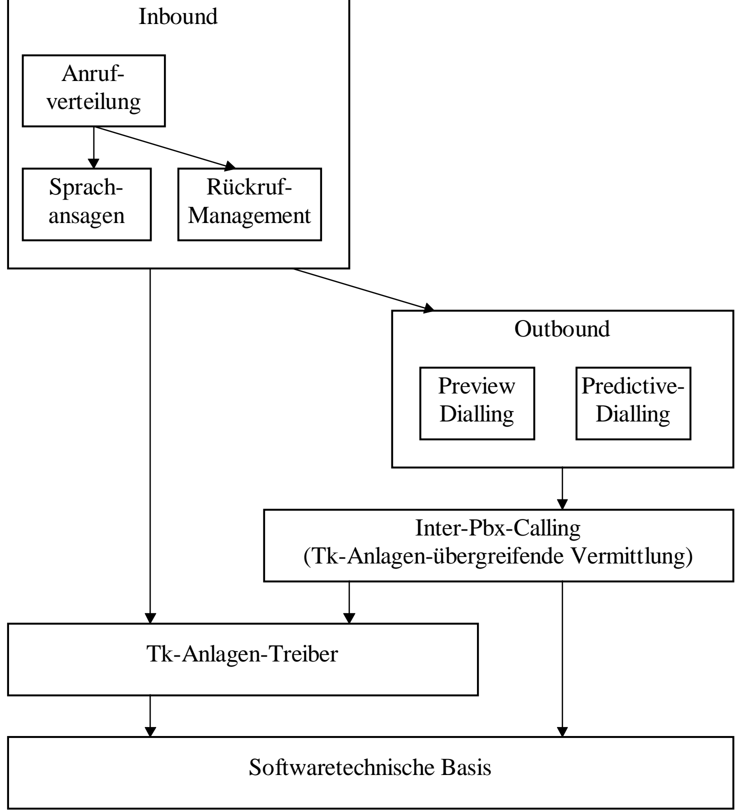

Abbildung 52: Mögliche Partitionierung eines umfangreicheren Telefonie-Frameworks

Ein Beispiel für ein Framework (das sog. GEBOS-Framework), dessen Aufteilung sich an den zu unterstützenden Aufgabenbereichen orientiert, findet sich in [BGKLRZ 97]. Dort wird das Framework anhand der Aufgabenbereiche im Bankgeschäft strukturiert.

Eine andere mögliche Art der Partitionsbildung könnte sich an der Organisationsstruktur des einsetzenden Unternehmens orientieren. Die in dieser Arbeit besprochenen Telefonie-Systeme sind Standardsoftware und können daher nur nach Funktionsbereichen strukturiert werden, denn der konkrete Einsatzkontext ist unbekannt. Eine auf einen speziellen Kunden zugeschnittene Lösung könnte aber durchaus dessen Organisationsstruktur berücksichtigen. Beispielsweise könnte die Partitionierung auch die Art der zu unterstützenden Arbeitsplätze widerspiegeln. Die Unterstützung für einen Kapitalanlageberater würde sicher anders aussehen als für den Zeitschriftenvertrieb per Telefon. Eine an der Organisationsstruktur orientierte Partitionierung könnte sich von der in Abbildung 52 vorgestellten Partitionierung beispielsweise durch die fehlende Trennung in Inbound- und Outbound-Geschäft auszeichnen, wenn diese in der betrieblichen Praxis nicht vorhanden ist.

6.6 Zusammenfassung

Die Framework-Partitionierung basiert auf vielen in der Softwaretechnik bekannten Ansätzen und Methoden. Sie kombiniert unter der Bezeichnung "Layering" bekannte Techniken zur Schichtenbildung mit Konstruktionskriterien und -techniken, die u.a. aus der modularen Programmierung übertragen wurden.

Entscheidend ist jedoch die Folge der Partitionierung für die Evolution von Frameworks: Durch die explizite Modellierung einzelner Bereiche werden die Designentscheidungen verdeutlicht und können früh überprüft werden. Die Orientierung an den bereits bestehenden fachlichen Konzepten macht den Entwurf kommunizierbar - auch für Nicht-Software-Entwickler. Die Stufenbildung und lose Kopplung der Partitionen macht einzelne Design-entscheidungen auch im nachhinein leichter revidierbar.

7 Fazit und Ausblick

Dieses Kapitel zieht ein Resümee der Arbeit und gibt einen Ausblick auf offene Fragen, die in weiterführenden Arbeiten betrachtet werden sollten.

Es wurde dargelegt, daß die Evolution von Frameworks ein unvermeidlicher Vorgang ist. Er ist zu einem wesentlichen Teil in der Evolution des Anwendungsbereichs und dem Verständnis desselben begründet. Die exemplarisch aufgezeigten Evolutionsschritte des Telefonie-Frameworks machen dies deutlich. Das Wissen um die typischen Merkmale der Framework-Evolution legt den Grundstein zur Beurteilung der Folgen von Designentscheidungen bei der Framework-Entwicklung.

Es wurde jedoch auch deutlich, daß man auf vielen Ebenen mit der Evolution umgehen kann und daß wir als Software-Entwickler sehr wohl einen Einfluß auf die Auswirkungen und Konsequenzen der Evolution haben. Das Spektrum der Einflußmöglichkeiten reicht von Methoden, um die bei der Analyse bereits gemachte Erfahrungen besser in das Framework-Design zu integrieren, bis zu speziellen Implementationstechniken, die die Konsequenzen von unerwarteten Modifikationen mindern.

Irving und Eichmann stellen in ihrer Arbeit fest, daß das Streben nach Wiederverwendbarkeit ein Streben nach Adaptierbarkeit bedeutet: "anything built to be reusable is effectively built to be adapted" (siehe [Irving 96]). In dieser Arbeit über Framework-Evolution geht es mir ähnlich. Ich stelle fest, daß Anpaßbarkeit eine wichtige Grundvoraussetzung für die Evolutionierbarkeit eines Frameworks ist. Die Evolutionsschritte des Telefonie-Frameworks belegen die Wichtigkeit von Implementationstechniken, die spätere Änderungen und Erweiterungen nicht behindern. Mindestens ebenso wichtig ist jedoch eine Modellbildung im Framework, die es ermöglicht, so viele Erfahrungen aus der bisherigen Systementwicklung wie möglich in das Framework aufzunehmen und auf diese Weise ein möglichst stabiles Design zu erreichen.

Die Aufteilung von Frameworks in einzeln evolutionierende Partitionen ist der wichtigste Beitrag, den diese Arbeit zur Diskussion über adäquates Framework-Design liefert. Auch wenn die Gedanken zur Modularisierung von Softwaresystemen keineswegs neu sind, ist eine Rückbesinnung auf sie auch in der objektorientierten Programmierung durchaus fruchtbar und bereichernd.

Die vorgestellte Ausrichtung an den bisher üblichen Begriffen und Konzepten des Anwendungsbereichs als Partitionierungskriterien ist übertragbar auf andere Anwendungsbereiche, für die Frameworks erstellt werden sollen. Die Erfahrungen mit dem Telefonie-Framework zeigen, daß es für die Evolution eines Frameworks äußerst förderlich ist, wenn das fachliche Modell, das es verkörpert, dem existierenden Modell möglichst ähnlich ist. Es lassen sich so viele Erfahrungen in die Frameworkentwicklung einbringen, die auf ein völlig neues Modell nicht anwendbar gewesen wären.

Das im Rahmen dieser Arbeit entwickelte Framework hat sich in der Praxis bewährt und hat gezeigt, wie vorteilhaft sich die Partitionierung auf die Evolution eines Frameworks auswirkt. Die gestellten Ziele der Frameworkentwicklung konnten voll erreicht werden. Zur Zeit wird die Umstellung weiterer Prozesse des Micrologica Communication Center auf Framework-Technik erwogen.

Die Frage nach dem Umgang mit der Evolution von Frameworks ist mit dieser Arbeit sicherlich nicht endgültig beantwortet. Die Betrachtung beschränkte sich hier auf den Ein-Prozeßraum. Aufgrund der steigenden Komplexität der heutigen Systeme geht der Trend jedoch immer stärker zu verteilten Systemen. Dadurch stellt sich für weitere Arbeiten die Frage, wie man über Prozeßgrenzen hinaus evolutionsfähige Systeme erstellen kann. Dies wird in Zukunft zu untersuchen sein. In [MätBisch 96] werden bereits Vorschläge zur losen Kopplung von Prozessen gemacht. Es wäre die Frage zu klären, ob man ähnliche Partitionierungskriterien verwenden kann, um zu einer Aufteilung in mehrere Prozesse zu kommen.

Einer umfassenderen Betrachtung bedarf auch die Berücksichtigung der Evolution im Entwicklungsprozeß. Die Erörterung der damit verbundenen Management-Aspekte geht über die Möglichkeiten dieser Arbeit hinaus und bleibt weiteren Arbeiten vorbehalten.

Anhang A: Literatur

[Ackermann 96] Ackermann, Ph., "Developing Object-Oriented Multimedia Software", dpunkt-Verlag, 1996

[Ackroyd 95] Ackroyd, M., "Object-oriented design of a finite state machine", JOOP, June 1995

[Andert 95] Andert, G., "Frameworks in Taligent's CommonPoint", in [Lewis 95], 1995

[AT&T 95] "DEFINITY System Programmer's Guide Release 2.1", AT&T, 1995

[BaconSweeney 96] Bacon, D. F., Sweeney, P. F., "Fast Static Analysis of C++ Virtual Function Calls", OOPSLA '96 Conference Proceedings, 1996

[Beck 89] Beck, K., Cunningham, W., "A Laboratory For Teaching Object-Oriented Thinking", OOPSLA '89 Conference Proceedings, SIGPLAN Notices, Vol. 24, No. 10, November 1989

[BGZ 95] Bürkle, U., Gryczan, G., Züllighoven, H., "Object-Oriented System Development in a Banking Project: Methodology, Experience and Conclusions", Human-Computer Interaction, Special Issue: Empirical Studies of Object-Oriented Design, Vol. 10, No. 2 & 3, 1995, S. 293-336

[BGKLRZ 97] Bäumer, D., Gryczan, G., Knoll, R., Lilienthal, C., Riehle, D., Züllighoven, H., "Framework Development for Large Systems", Communications of the ACM, Oktober 1997

[BI 80] "Enzyklopädie Philosophie und Wissenschaftstheorie"; Bibliographisches Institut, Band 1, 1980

[BirEgg 93] Birrer, A., Eggenschwiler, T., "Frameworks in the Financial engineering Domain: An Experience Report", Proceedings of ECOOP '93, 1993

[BirEgg 94] Birrer, A., Eggenschwiler, T., "Leveraging Corporate Software Development", Proceedings of the UBILAB '94 Conference, Universitätsverlag Konstanz, 1994

[BirBischEgg 95] Birrer, A., Bischofberger, W. R., Eggenschwiler, T., "Wiederverwendung durch Frameworktechnik - vom Mythos zur Realität", OBJEKTspektrum September / Oktober 1995

[Bischofberger 92] Bischofberger, W. R., "Sniff - A Pragmatic Approach to a C++ Programming Environment", Proceedings of the USENIX C++ Conference, Portland, Oregon, August 1992

[Bischofberger 95] Bischofberger, W. R., "Frameworkbasierte Softwareentwicklung" Proceedings of OOP '95, 1995

[BKGSZ 95] Bäumer, D., Knoll, R., Gryczan, G., Strunk, W., Züllighoven, H., "Objektorientierte Entwicklung anwendungsspezifischer Rahmenwerke", OBJEKTspektrum November / Dezember 1995

[BKGZ 96] Bäumer, D., Knoll, R., Gryczan, G., Züllighoven, H., "Large Scale Object-Oriented Software-Development in a Banking Environment - An Experience Report", Proceedings of ECOOP '96, Cointe, P. (Herausgeber), Springer-Verlag, 1996, S. 73-90

[Brockhaus 88] "Brockhaus Enzyklopädie in 42 Bänden", F.A. Brockhaus, 19. Auflage, Band 6, 1988

[Brooks 87] Brooks, F., "No Silver Bullet - Essence and Accidents of Software Engineering", IEEE Computer, April 1987

[BrownForte 96] Brown, G., Forte, P., "Building reusable classes for frameworks", Journal of Object-Oriented Programming, Vol. 9, No. 7, S. 49-54, 1996

[Burton 95] Burton, J., ”Standard Issue”, Byte September 1995, S. 201-208

[Buschmann 96] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., "A System of Patterns", John Wiley & Sons, 1996

[Campbell 92] Campbell, R. H., Islam, N., Madany, P., "Choices, Frameworks and Refinement", Computing Systems, Vol. 5, No. 3, 1992

[Casais 91] Casais, E., "Managing Evolution in Object Oriented Environments: An Algorithmic Approach", Ph.D. thesis, University of Geneva, 1991

[Chapman 95] Chapman, M., Berndt, H., Gatti, N., "Software Architecture for the future information market", Computer Communications, Vol. 18, No. 11, S. 825-837, 1995

[Chen 94] Chen, D. J., Chen, David T. K., "An experimental study of using reusable software design frameworks to achieve software reuse", JOOP, Vol. 7, No. 2, 1994

[Coplin 92] Coplin, J. O., "Advanced C++ - Programming Styles and Idioms", Addison-Wesley, 1992

[Coplin 95] Coplin, J. O. "A Generative Development-Process Pattern Language", in [CopSchm 95], 1995

[CopSchm 95] Coplin, J. O., Schmidt, D. C., Herausgeber, "Pattern Languages of Program Design", Addison-Wesley, 1995

[Cotter 95] Cotter, S., Potel, M., ”Inside Taligent Technology”, Addison-Wesley, 1995

[Cox 86] Cox, B. J., "Object Oriented Programming - An Evolutionary Approach", Addison-Wesley, 1986

[Christner 92] Christner, J., "Abiturwissen Evolution", Klett-Verlag, 1992

[Cronin 95] Cronin, P., Choy, H., "TSAPI - A Technical Tutorial", Novell Inc., 1995

[Cronin 96] Cronin, P., "An Introduction to TSAPI and Network Telephony", IEEE Communications Magazine, April 1996, Vol. 34, No. 4, S. 48-54

[Dijkstra 68] Dijkstra, E. W., "The structure of the 'T.H.E.' multiprogramming system", Communications of the ACM, 18(8), 1968, S. 453-457

[DriHölz 96] Driesen, K., Hölzle, U., "The Direct Cost of Virtual Function Calls in C++", OOPSLA '96 Conference Proceedings, 1996

[ECTF 96] "S.100 Media Services C Language APIs, Revision 1.0", ECTF, 1996

[Ellis 90] Ellis, M. A., Stroustrup, B., "The Annotated C++ Reference Manual", Addison-Wesley, 1990

[Eisenecker 96] Eisenecker, U., "Generatives Programmieren in C++", OBJEKT-spektrum November / Dezember 1996

[Feldman 79] Feldman, S. E., "Make: A Program for Maintaining Computer Programs", Software Practice and Experience, 1979, S. 255-265

[Firesmith 93] Firesmith, D. G., "Frameworks: The golden path to object Nirvana", JOOP, Vol. 6, No. 6, 1993, S. 6-8

[Foote 88] Foote, B., "Designing to Facilitate Change with Object-Oriented Frameworks", master's thesis, University of Illinois, 1988

[FootOpd 95] Foote, B., Opdyke, W. F., "Lifecycle and Refactoring Patterns That Support Evolution and Reuse", in [CopSchm 95], 1995

[FooteYoder 96] Foote, B., Yoder, J., "Attracting Reuse", PLoP '96 submission, 1996

[Gamma 92] Gamma, E., "Objektorientierte Software-Entwicklung am Beispiel von ET++ - Design-Muster, Klassenbibliothek, Werkzeuge", Springer-Verlag, 1992

[Gamma 96] Gamma, E., "The Extension Objects Pattern", PLoP '96 submission, 1996

[Garlan 95] Garlan, D., Allen, R., Ockerbloom, J., "Architectural Mismatch or Why it's hard to build software out of existing parts", Proceedings of the Seventeenth International Conference on Software Engineering, 1995

[GHJV 95] Gamma, E., Helm, R., Johnson, R., Vlissides, J., "Design Patterns. Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995

[Greenfield 96] Greenfield, D., "CTI Specifications: Squaring off", Data Communications, Mai 1996

[Griss 94] Griss, M. L., Favaro, J., Walton, P., "Managerial and organizational issues - starting and running a software reuse program", in [Schäfer 94], 1994

[Hansson 95] Hansson, A., "Evolution of intelligent network concepts", Computer Communications, Vol. 18, No. 11, S. 793-801, 1995

[Heuser 95] Heuser, L., Hehl, K., "Entwicklung eines Anrufverteilsystems mit Hilfe von DC++", OBJEKTspektrum März / April 1995, S. 61-66

[Hirschfeld 97] Hirschfeld, R., "Three-Tier Distribution Architecture", in "Collected papers from PLoP '96 and EuroPLoP '96 Conferences", Washington University Department of Computer Science, technical report wucs-97-07, 1997

[Hüni 95] Hüni, H., Johnson, R., Engel, R., "A Framework for Network Protocol Software", SIGPLAN Notices, Vol. 30, No. 10, S. 358-369, 1995

[Hüni 97] Hüni, H., Keller, B., "Ein OO-Framework für Netzwerkprotokoll-Software", OBJEKTspektrum Januar / Februar 1997, S. 51-56

[Irving 96] Irving, C. W., Eichmann, D., "Patterns and Design Adaptability", PLoP '96 submission, 1996

[JohnFoote 88] Johnson, R. E., Foote, B., "Designing reusable classes", JOOP, Vol. 1, No. 2, 1988, S. 22-35

[JohnRuso 91] Johnson, R. E., Russo, V. F., "Reusing Object-Oriented Designs", University of Illinois, technical report UIUCDCS 91-1696, 1991

[Johnson 93] Johnson, R. E., "How to Design Frameworks", Tutorial Notes, OOPSLA '93, 1993

[Karlson 95] Karlson, E., Herausgeber, "Software Reuse - A Holistic Approach", John Wiley & Sons, 1996

[Kilbert 93] Kilbert, K., Gryczan, G., Züllighoven, H., "Anwendungsorientierte Softwareentwicklung", Vieweg, 1993

[Koschek 95] Koschek, H., "Micrologica Programmierrichtlinien für C++", Version 1.0-3, Micrologica GmbH, interne Arbeitsanweisung QS-07-10, 1995

[Kuhn 96] Kuhn, K., "Call Center für die Börseneinführung der Telekom-Aktie", Midrange MAGAZIN, November 96, S. 70ff

[Kuhn 97a] Kuhn, K., "Deutsche Telekom setzt auf Call Center: Das Telefon wiederentdeckt", Gateway 1/97, S. 62

[Kuhn 97b] Kuhn, K., "Deutsche Telekom AG: Das Call Center für die Telekom-Aktie", ISDN-Report, 1/97, S. 23ff

[Lakos 96] Lakos, J., "Large Scale C++ Software Design", Addison-Wesley, 1996

[Lewis 95] Lewis, T., Herausgeber, "Object Oriented Application Frameworks", Manning Publication Co., 1995

[MätBisch 96] Mätzel, K.-U., Bischofberger, W., "Evolution of Object Systems - or How to tackle the Slippage Problem in object systems", Proceedings of the Ubilab Conference '96, Universitätsverlag Konstanz, 1996

[MätBisch 97] Mätzel, K.-U., Bischofberger, W., "Designing Object Systems for Evolution", TAPOS '97 Conference Proceedings, Vol. 3, No. 4, 1997

[Martin 95] Martin, R., "Discovering Patterns in Existing Applications", in [CopSchm 95], 1995

[McConnell 93] McConnell, S., "Code Complete - A Practical Handbook of Software Construction", Microsoft Press, 1993

[Meyer 82] Meyer, B., "Principles of Package Design", Communications of the ACM, Vol. 25, 1982, S. 419-428

[Meyer 88] Meyer, B., "Object-oriented Software Construction", Prentice Hall, 1988

[Meyer 90] Meyer, B., "Lessons from the design of the Eiffel libraries", Communications of the ACM, Vol. 33, No. 9, September 1990

[Meyer 91] Meyer, B., "Eiffel: The Language", Prentice Hall, 1991

[Meyer 94] Meyer, B., "Reusable Software", Prentice Hall, 1994

[Meyers 96] Meyers, S., "More Effective C++", Addison-Wesley, 1996

[Moore 96] Moore, I., "Automatic Inheritance Hierarchy Restructuring and Method Refactoring", OOPSLA '96 Conference Proceedings, 1996

[Musser 96] Musser, D. R., Saini, A., "STL Tutorial and Reference Guide", Addison-Wesley, 1996

[Nixon 96] Nixon, T., ”Design Considerations for Computer-Telephony Application Programming Interfaces and Related Components”, IEEE Communications Magazine, April 1996, Vol. 34, No. 4, S. 43-47

[Novell 95] "Telephony Services Application Programming Interface (TSAPI) Release 2", NetWare SDK, Novell Inc., 1995

[Opdyke 92] Opdyke, W. F., "Refactoring Object-Oriented Frameworks", Ph.D. thesis, also technical report No. UIUCDCS-R-92-1759, University of Illinois, 1992

[Pfeiffer 97] Pfeiffer, A., "SNiFF+: eine einheitliche Arbeitsumgebung für große Softwareprojekte", OBJEKTspektrum März / April 1997, S. 30ff

[Pree 94a] Pree, W., "Design Patterns for Object-Oriented Software Development", Addison-Wesley, 1994

[Pree 94b] Pree, W., ”Meta Patterns - A Means for Capturing the Essentials of Reusable Object-Oriented Design”, Proceedings of ECOOP ’94, S. 150-162, 1994

[RobJohn 96] Roberts, D., Johnson, R., "Evolve Frameworks into Domain-Specific Languages", PLoP 96 submission, 1996

[Rosenstein 95] Rosenstein, L., "MacApp: First Commercially Successful Framework", in [Lewis 95], 1995

[Salamone 96] Salamone, S., "Opening PBX Doors", BYTE, March 1996

[SaneCamp 95] Sane, A., Campbell, R., "Object-Oriented State machines: Subclassing, Composition, Delegation and Genericity", Proceedings of OOPSLA '95, 1995

[Schäfer 94] Schäfer, W., Prieto-diaz, R., Matsumoto, M. (Herausgeber), "Software Reusability", Ellis Horwood, 1994

[Scharenberg 91] Scharenberg, M. E., Dunsmore, H. E., "Evolution of classes and objects during object-oriented design and programming", JOOP, Vol. 3, No. 5, 1991

[Schmidberger 95] Schmidberger, R., "Metaklassen und Laufzeitinformationen für C++", OBJEKTspektrum Juni / Juli 1995, S. 84ff

[Schmidt 93] Schmidt, D. C., ”Reactor. An Object Behavior Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching.”, C++ Report, Vol. 5, No. 2, Februar 1993, S. 1-12

[Schmidt 95a] Schmidt, D. C., "Using design patterns to develop reusable object-oriented communication software", Communications of the ACM, Vol. 38, No. 10, S. 65-74, 1995

[Schmidt 95b] Schmidt, D. C., Stephenson, P., "Experience using design patterns to evolve communication software across diverse OS platforms", ECOOP '95 Conference Proceedings, S. 399-423, Springer-Verlag, 1995

[Schneegans 96] Schneegans, M., "Kosten/Nutzen-Analyse von Call-Centern: Mehr Gewinn am Telefon", Gateway 10/96, S. 50

[Schneegans 97] Schneegans, M., "Gleiche Daten für alle - Computertelefonie im Call Center", Gateway 3/97, S. 142-143

[Siberski 95] Siberski, W., "Meta-Konzepte in objektorientierten Sprachen - unter besonderer Berücksichtigung von C++", Diplomarbeit, Universität Hamburg, Fachbereich Informatik, 1995

[Stevenson 95] Stevenson, R., "Whatever happened to finite state machines", Object Magazine, November / December 1995

[Stroustrup 92] Stroustrup, B., "Die C++ Programmiersprache", 2. überarbeitete Auflage, Addison-Wesley, 1992

[Stroustrup 94a] Stroustrup, B., "The Design and Evolution of C++", Addison-Wesley, 1994

[Stroustrup 94b] Stroustrup, B., "Design und Entwicklung von C++", Addison-Wesley, 1994 (deutsche Übersetzung von [Stroustrup 94a])

[Sullivan 94] Sullivan, K. J., "Mediators: Easing the Design and Evolution of Integrated Systems", Technical Report 94-08-01, Department of Computer Science and Engineering, University of Washington, 1994

[Taligent 93] "Leveraging Object-Oriented Frameworks", Taligent Inc., 1993

[Taligent 94] "Building Object-Oriented Frameworks", Taligent Inc., 1994

[Taligent 95] Taligent Inc., "The Power of Frameworks", Addison-Wesley, 1995

[Thießen 96] Thießen, S., "FSM-Klassen - Dokumentation, Version 1.0-1", Micrologica GmbH, interne Dokumentation, 1996

[Udell 94] Udell, J., "Computer Telephony", BYTE, Juli 1994

[van Dahle 95] van Dahle, H., "Neue Call Center-Technik für Direktmarketing-Agentur", TeleTalk 11/95, S. 20

[Vlissides 96a] Vlissides, J., "Pattern Hatching - Protection, Part I: The Hollywood Principle", C++-Report, Februar 1996

[Vlissides 96b] Vlissides, J., "Pattern Hatching - To Kill a Singleton", C++-Report, June 1996

[Wegner 90] Wegner, P., "Concepts and Paradigms of Object-Oriented Programming", OOPS Messenger, No. 1, 1990, S. 8-87

[WeiGamMar 89] Weinand, A., Gamma, E., Marty, R., "Design and Implementation of ET++, a Seamless Object-Oriented Application Framework", Structured Programming, Vol. 10, No. 2, Springer-Verlag, 1989

[Welt 96] "Micrologica kanalisiert die Anruf-Flut", Die Welt, 14. Oktober 1996

[Wilson 90] Wilson, D. A., Rosenstein, L. S., Shafer, D., "Programming with MacApp", Addison-Wesley, 1990

[Wingo 95] Wingo, S., "The Microsoft Foundation Classes - Past, present, and future", C++ Report, September 1995, S. 43-45

[Wirfs-Brook 90] Wirfs-Brook, R., Wilkerson, B., Wiener, L., "Designing Object-Oriented Software", Prentice Hall, 1990

[Zweig 90] Zweig, J. M., Johnson, R. E., "The conduit: a communication abstraction in C++", Usenix C++ Conference Proceedings, 1990

Anhang B: Glossar der Telefonie-Begriffe

ACD	Automatic Call Distribution; Die Verteilung eingehender Anrufe auf Sachbearbeiter nach festgelegten Regeln.
ACD_TC	Prozeß, der eine vereinfachte Telefonie-Schnittstelle für das MCC zur Verfügung stellt.
ACL	Kommandosprache der Hicom-Tk-Anlage von Siemens
ACS	API Control Services; Teil des Telefonie-Protokolls TSAPI, das für die Verwaltung von Streams zuständig ist
Agent	funktionaler Teil des MCC, welcher die Dienstleistungen für Sachbearbeiter eines Anrufs zur Verfügung stellt
Call	Kommunikationsbeziehung zwischen 2 bis n Devices. Beim Verbindungsaufbau kann zeitweise auch nur 1 Device beteiligt sein.
Connection	Verbindung zwischen einem Call und einem Device (1:1-Beziehung)
CSTA	a.) Oberbegriff: Computer-Supported Telecommunications Applications b.) Standard für Telefonie-API
CTI-Link	Verbindung einer Tk-Anlage zu einem Computer (Hardware)
CTI	Computer Telephony Integration; Integration von Computer und Telefon
Device	bezeichnet z.B. ein Telefon oder eine Sammelnummer
Dialer-Dienst	Erzeugt einen ausgehenden Anruf und läßt diesen nach erfolgreichem Aufbau (!) einem freien Agenten zuteilen.
dVPU	Digital Voice Processing Unit; spezielle Baugruppe zur Sprachsteuerung
Event	asynchrone Ereignismeldung über Zustandsänderung, enthält alle nötigen Informationen über den neuen Zustand
FSM	Finite State Machine (dt. Endlicher Automat)
Half-Call	konzeptionelles Modell, das den Zustand einer Tk-Anlage beschreibt (CSTA-Standard)
Hicom	Tk-Anlage von Siemens
ICM	Incoming Call Management: umfassendes Management eingehender Anrufe
Inbound-Dienst	Verarbeitung eingehender Anrufe
ISDN	Integrated Services Digital Network; "digitales Telefonnetz"

MakeCall	Request zum Aufbau einer Telefonverbindung zwischen zwei Devices
MCC	Micrologica Communication Center. High-Level Telefonie-Anwendung der Micrologica GmbH
Monitorpunkt	a.) Ein Device, für das Zustandsänderungen gemeldet werden sollen. Um Meldungen über Zustandsänderungen anzufordern, "setzt" man einen Monitorpunkt. b.) Die zu einem Device gelieferte Information über Zustandsänderungen.
Outbound-Dienst	Dienst, der eine Verbindung zu einem bestimmten Agenten herstellt und dann einen ausgehenden Anruf erzeugt und ihn an den Agenten vermittelt. (Zum Wählen an Telefonanlage ohne CTI-Link.)
PBX	Tk-Anlage (engl. Private Branch Exchange)
Power Dialing	Anwählen möglichst vieler Nummern und Verteilen der zustande-kommenden Verbindungen auf freie Agenten. Meist wird es in Verbindung mit Predictive Dialing eingesetzt. Evtl. zuviel aufgebaute Verbindungen werden stillschweigend abgebaut („silent dropping“)
Predictive Dialing	Aufbau von Verbindungen in der Erwartung, daß bald Agenten frei werden
Preview Dialing	Wählen einer Telefonnummer aus einer Anwendung
Request	Auftrag an eine Tk-Anlage
SEP	Service Entry Point; Die Telefonnummer, die symbolisch für einen nach außen angebotenen Dienst steht. Auf dieser Nummer eingehende Anrufe werden dann weiter verteilt.
TAPI	Telefonie-API von Microsoft / Intel
Telefonie	Vermittlung von Telefonleitungen bzw. jegliche Art von Systemen zur Unterstützung des Telefonierens
Tk-Anlage	Telekommunikationsanlage; z.B. eine Telefonanlage
TNCCT	Kommandosprache für Integral-Tk-Anlage von Telenorma
TransferCall	Request zum Transferieren einer Telefonverbindung von einem Device auf ein anderes
TSAPI	Telefonie-API von Novell / AT&T
TSERV	Treiber-Prozeß zur Steuerung von Tk-Anlagen