Scripting für CORBA

Diplomarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Michael Held

1999

Leiter der Arbeit:
Prof. Dr. Oscar Nierstrasz
Institut für Informatik und angewandte Mathematik

Die Adresse des Autors:

Software Composition Group
Institut für Informatik und angewandte Mathematik - Universität Bern
Neubrückstrasse 10
CH-3012 Bern
Email: held@iam.unibe.ch
WWW: http://www.iam.unibe.ch/~held

oder

Michael Held
Römerstrasse 16
CH-2563 Ipsach

Zusammenfassung

Verteilte Applikationen stellen hohe Anforderungen an die Software-Hersteller, da diese Applikationen sehr schnell sehr komplex werden, unzureichende Entwicklungswerkzeuge haben und einen hohen Administrationsaufwand verursachen. Zur Unterstützung werden heutzutage oft Middleware-Technologien eingesetzt, wie zum Beispiel DCOM, RMI oder CORBA. In dieser Arbeit wird CORBA analysiert und mit Hilfe eines in dieser Arbeit erstellten Verbesserungsansatzes und einer Skript-Sprache zu einem Komponenten Framework verbessert.

CORBA ist eine Middleware für verteilte, heterogene Applikationen und verspricht die Unabhängigkeit von Programmiersprache sowie eine transparente Verteilung von Objekten. CORBA hat einige Schwachstellen, wie zum Beispiel die mangelhafte Flexibilität, die mit hohem Aufwand verbundene Erweiterbarkeit oder das aufwendige Testen. In der Informationstechnologie gibt es heute viele, schnelle Veränderungen, was ein hohes Mass an Flexibilität und Erweiterbar- beziehungsweise Veränderbarkeit der Software erfordert.

In dieser Arbeit wird gezeigt, dass eine Middleware, wie CORBA, durch einen einfachen Verbesserungsansatz, der *Komponenten-Proxy-Methode*, wesentlich verbessert werden kann. Mit Hilfe dieser Methode und der verwendeten Skript-Sprache Python wird ein Komponenten Framework erstellt, das die Verwendung von verteilten Komponenten wesentlich flexibler macht. Durch dieses Framework wird eine höhere Abstraktion der Applikationen erreicht, da die Applikationslogik von den Komponenten getrennt werden kann und die Komponenten durch *Scripting* verbunden werden. Dadurch entstehen allgemeinere Komponenten, die einfacher wiederverwendet werden können. Die Komponenten-Proxy-Methode fördert zudem die Einsatzmöglichkeiten von CORBA. Es werden Ansätze für einen *travelling agent* und einen *intelligent proxy* gezeigt.

Persönliche Bemerkungen

> So eine Arbeit wird eigentlich nie fertig, man muss sie für fertig erklären, wenn man nach Zeit und Umständen das Möglichste getan hat.
> *Goethe, Italienische Reise*

An dieser Stelle möchte ich allen danken, die mich in irgendeiner Form bei dieser Arbeit unterstützt haben. Nachfolgenden Personen gilt ein ganz besonderer Dank; ich möchte sie namentlich erwähnen.

Als erstes danke ich meinem Betreuer **Jean-Guy Schneider** für seine kompetente Unterstützung. Er war jederzeit bereit, mit mir Probleme oder Unklarheiten zu diskutieren. Sein Rat sowie seine *Paper*-Tips brachten mir stets neue, hilfreiche Anregungen.

Danken möchte ich insbesondere **Prof. Dr. Oscar Nierstrasz** für seine Unterstützung, ohne die diese Arbeit nicht möglich gewesen wäre.

Ein Dank geht an die ganze **Software Composition Group**, die ich jeder Zeit um Rat fragen konnte. In dieser Arbeitsgruppe herrscht eine offene, kreative und kritische Arbeitsatmosphäre.

Daniel Kühni und **Nissim Buchs** danke ich für die kritische Durchsicht der ganzen Diplomarbeit und die hilfreichen Gespräche, die wir in der Mensa führen konnten. **Franz Achermann** und **Tobias Röthlisberger** möchte ich für den *Review* der Arbeit danken.

Schliesslich gilt mein ganz persönlicher Dank **meinem Vater** für die finanzielle Unterstützung und meiner Freundin **Käthi Urbanek** für ihre Geduld, das Verständnis und ihre Unterstützung.

Inhaltsverzeichnis

Zusammenfassung **i**

Persönliche Bemerkungen **ii**

1 Einleitung **1**
1.1 Motivation . . . 1
1.2 Ziele . . . 2
1.3 Aufbau der Arbeit . . . 3

2 Begriffe und Konzepte **4**
2.1 Verteilte Systeme . . . 4
2.2 Middleware . . . 5
2.2.1 DCOM . . . 6
2.2.2 RMI . . . 7
2.2.3 CORBA . . . 7
2.3 Software Komposition . . . 8
2.3.1 Komponenten . . . 8
2.3.2 Komponenten Framework . . . 9
2.4 Legacy Komponenten . . . 9

3 Skript-Sprachen **12**
3.1 Einleitung . . . 12
3.2 Eigenschaften . . . 14
3.3 Ein Überblick . . . 15
3.4 Python . . . 18

4 CORBA **23**
4.1 Einleitung . . . 23
4.2 Architektur . . . 25
4.2.1 OMA Guide (*Object Management Architecture Guide*) . . . 26
4.2.2 Aufbau eines CORBA ORBs . . . 29
4.3 Eigenschaften . . . 31
4.3.1 IDL (Interface Definition Language) . . . 31
4.3.2 Sprachanbindung (Language mapping) . . . 31

4.3.3 Interoperabilität . . . 33
4.3.4 Dynamischer Methodenaufruf . . . 34
4.4 “Pricing”-Applikation - ein Beispiel . . . 36

5 Problemanalyse **41**
5.1 Probleme von CORBA . . . 41
5.2 Einschränkung der Problemstellung . . . 45

6 Lösungsansatz der CORBA-Probleme **47**
6.1 Überblick . . . 47
6.2 Die Methode des Komponenten-Proxy . . . 48

7 Komponenten-Proxy **51**
7.1 Die Methode . . . 51
7.1.1 *NameServer* . . . 53
7.1.2 Das Modul *corba_proxy* . . . 55
7.1.3 Verwendung . . . 57
7.2 Komponenten Framework für CORBA . . . 58
7.2.1 Software Architektur . . . 58
7.2.2 Python - die Skript-Sprache . . . 61
7.2.3 *Glue - Wrapping* von *legacy components* . . . 63
7.3 Zusätzliche Möglichkeiten . . . 66
7.3.1 Intelligent Proxy . . . 66
7.3.2 Agent (*Mobile Agent*) . . . 69
7.4 Grenzen . . . 74
7.4.1 Kein dynamischer Methodenaufruf . . . 74
7.4.2 “Halb-dynamischer” Methodenaufruf . . . 74
7.4.3 Abbildung von Python-Komponenten nach CORBA . . . 75

8 Fallstudie: FAMOOS **77**
8.1 Das FAMOOS-Projekt . . . 77
8.2 Einleitung . . . 79
8.3 Aufgabe . . . 80
8.4 *Wrapping* der beiden bestehenden Komponenten . . . 81
8.4.1 Adaption der *sniff2cdif*-Komponente . . . 81
8.4.2 Adaption der FAMOOS-*Darstellung*s-Komponente . . . 82
8.5 Erweiterungen von FAMOOS . . . 83
8.5.1 *GUI* der *FAMOOS-Darstellung* . . . 83
8.5.2 *TableGenerator* . . . 84
8.6 Verwendungsbeispiele . . . 85
8.6.1 Interaktives *Debugging* von Komponenten . . . 85
8.6.2 Applikationslogik - zwei Beispiele . . . 85
8.6.3 Agenten . . . 87
8.6.4 Komponenten-Erweiterung mit dem *Intelligent Proxy* . . . 88

8.7 Diskussion 89

9 Schlussbemerkungen **90**
9.1 Erreichte Ziele 90
9.2 Offene Probleme 91

A Arbeitsumgebung **93**

B Technische Details von CORBA **95**
B.1 Ausnahmen (Exception) beim VisiBroker 95
B.2 Das *low level*-Verhalten eines ORBs 96
B.3 IOR/IIOP - Interoperabilität 96
B.4 CORBA 3.0 - der neue Standard 97
B.5 Fnorb für Python 98
B.6 Skript-Sprachen-Anbindung an CORBA 99

C CORBA Komponenten-Bibliothek **100**
C.1 Komponenten-Proxy-Methode 101
C.1.1 *corba_proxy* - Modul 101
C.1.2 NameServer 101
C.1.3 *Intelligent Proxy* 102
C.1.4 *Mobile Agent* 102
C.2 CORBA-Komponenten 102
C.2.1 Hallo, Welt! 102
C.2.2 Counter-Komponente 102
C.2.3 *Pricing* 103
C.2.4 Das *ZeitServer*-Beispiel 103
C.3 FAMOOS 103
C.3.1 *sniff2cdif*-Komponente 103
C.3.2 FAMOOS-Darstellung 104
C.3.3 *TableGenerator* 105
C.3.4 Skripts 105
C.4 Weitere CORBA-Beispiele 106
C.4.1 Komplexe Zahlen 106
C.4.2 DII 107

Literaturverzeichnis **108**

Abbildungsverzeichnis

2.1 Eine Middleware verbindet Client und Server 5
2.2 Komponente als black box 8
2.3 Ein ideales Komponenten Framework 10

3.1 *Scripting* vs. *System Programming Languages* von John Ousterhout 14
3.2 Python, die Skript- bzw. System-Programmiersprache 19
3.3 Ein Python-Skript: Suchen und ersetzen von Wörtern in einer Datei. 21

4.1 2-Tier Architektur 24
4.2 3-Tier Architektur 24
4.3 n-Tier Architektur 25
4.4 CORBA liegt zwischen Applikation und Betriebssystem 26
4.5 OMA-Referenzmodell 27
4.6 Methodenaufruf über den ORB 28
4.7 Die Struktur eines ORBs 30
4.8 IDL-Beispiel: Komplexe Zahlen 33
4.9 Interoperabilität 35

6.1 Eine Abstraktion durch ein Modul 49
6.2 Ein Komponenten-Proxy leitet die Anfragen zur Original-Komponente weiter. . . 50

7.1 Methode der Komponenten-Proxy 52
7.2 Mechanismus der Komponenten-Proxy-Methode 53
7.3 Architektur des Komponenten Frameworks 58
7.4 Hierarchische Struktur einer n-Tier Applikation 60
7.5 Test von CORBA-Komponenten 62
7.6 Grafische Benutzeroberfläche einer Passwortabfrage 63
7.7 Adaption einer Komponente durch einen Wrapper 64
7.8 Transformation von inkompatiblen Datentypen in der Applikationslogik 65
7.9 Analyse Tool mit Hilfe des Intelligent Proxy 67
7.10 UML Klassendiagramm des Intelligent Proxy 68
7.11 *Sending Agent* 70
7.12 *Travelling Agent* 70
7.13 Aufbau des *Travelling Agent* 72
7.14 Kein Python-Proxy für CORBA 75

8.1 Famoos *Reengineering*-Lebenszyklus 78
8.2 Famoos Komponenten 79
8.3 Famoos Szenario 81
8.4 Grafische Benutzeroberfläche für ein FAMOOS-Modell 84

B.1 Ausnahmebehandlung im VisiBroker für Java 95
B.2 Interoperabilität mit IIOP 97

C.1 Eingabe Applet vom Komplex Rechner 106
C.2 Grafische Darstellung der Resultate des Komplex Rechners 107

Tabellenverzeichnis

3.1 Wichtige Programmier-Paradigmen 13
3.2 Python, Perl und Tcl im Vergleich 16

4.1 Auflistung aller CORBA Dienste 29
4.2 Die IDL-Spezifikation 32
4.3 IDL nach Java Mapping 34

B.1 Überblick der CORBA Spezifikationen 98

Kapitel 1

Einleitung

Das Gebiet der Informationstechnologie ist einem dauernden Veränderungsprozess ausgesetzt. Laufend werden neue Technologien entwickelt und nur kurze Zeit später sind diese in Form eines Produktes auf dem Markt erhältlich. Der Markt der Informationstechnologie ist sehr gross, was einen enormen Konkurrenzkampf zur Folge hat. Als Folge dieser schnellen Veränderungen und des grossen Konkurrenzkampfes entstehen die unterschiedlichsten Normen, sowie eine Vielfalt an Standards und Protokollen. Aus dieser Vielfalt an Normen resultieren inkompatible Sofwarekomponenten, was zu grossen Kommunikationsproblemen führen kann.

Ein Ansatz zur Lösung des Kommunikationsproblems besteht in der Definition eines globalen Standards, der die Interoperabilität garantieren würde, so dass unterschiedliche Softwarekomponenten mit den unterschiedlichsten Normen sich verständigen könnten. Ein Ziel der idealen Informationstechnologie ist daher ein universeller Standard, der alle Hard- und Softwarekomponenten umfassen könnte.

Leider sind wir heute von der idealen Informationstechnologie weit entfernt. Es werden aber immer wieder Schritte unternommen, um inkompatible Formate miteinander zu verbinden. CORBA ist ein Standard, der Interoperabilität zwischen heterogenen, verteilten Rechnern mit unterschiedlichen Programmiersprachen ermöglicht.

1.1 Motivation

CORBA ist eine Middleware für verteilte, heterogene Client/Server-Applikationen und verspricht die Unabhängigkeit von Programmiersprache und eine transparente Verteilung von Objekten. Diese Eigenschaften machen CORBA zu einem Standard, der Interoperabilität zwischen verschiedenen Hardware- und Softwarekomponenten ermöglicht. Aus der praktischen Verwendung von CORBA ergeben sich aber die folgenden Probleme (vgl. Problemanalyse in Kapitel 5):

- Mangelnde Flexibilität: Der Standard von CORBA 2.0 basiert auf einem Objektmodell (vgl. [OMG97b]). Die Erfahrung hat gezeigt, dass die objektorientierte Technologie keine Garantie für flexible und anpassbare Software-Entwicklung ist (vgl. [SN98a]).

- Aufwendiges Testen: Das Testen einer einzelnen Komponente ist aufwendig, das heisst, um einen Server zu testen, muss ein vollständiger CORBA-Client geschrieben werden. Zudem gibt es noch keine *Debugger* und Analyse-Tools für CORBA (vgl. [BR97]).

- Ungenügende Erweiterbarkeit: CORBA-Objekte besitzen aufgrund der Implementation genau eine Schnittstelle (vgl. [OMG97b]). Diese Eigenschaft birgt die Schwierigkeit Komponenten zu verändern (vgl. [Maf97]), das heisst, CORBA-Objekte müssen für jede zusätzliche Funktionalität neu angepasst werden (vgl. [BR97]).

- Mangelhaftes Abstraktionsniveau: Grosse Applikation tendieren zu grosser Komplexität (vgl. [Maf97]). Oft wird daher die Applikationslogik auf verschiedene Komponenten verteilt, was zu einer *Maintenance Nightmare* führen kann.

- Wiederverwendungsproblem: Da grössere CORBA-Applikationen meist eine hohe Komplexität erlangen und CORBA nicht sehr fehlertolerant ist (vgl. [EKM97]), müssen diese Client/Server-Applikationen oft genau aufeinander abgestimmt werden. CORBA-Komponenten von solch aufeinander abgestimmten Applikationen können daher kaum wiederverwendet werden. Es stellt sich zudem die Frage, auf welche Weise *legacy component*s in CORBA integriert werden können.

1.2 Ziele

In dieser Arbeit wird versucht, CORBA in Form eines Komponenten Frameworks zu verwenden, in welchem die einzelnen Komponenten mit einer Skript-Sprache zusammengesetzt (connected) werden. Das Ziel dieser Arbeit ist, einen vertieften Einblick in die Themengebiete *Scripting* und CORBA zu erhalten. Es sollten die folgenden Teilziele erreicht werden:

- Einarbeitung in die Themengebiete: Skript-Sprachen und CORBA

- Erstellung einer Problemanalyse von CORBA

- Ausarbeitung eines Verbesserungsansatzes mit Hilfe einer Skript-Sprache

- Validierung des Ansatzes anhand einer Komponenten-Bibliothek

Mit dem in dieser Arbeit ausgearbeiteten Verbesserungsansatz sollen die folgenden Verbesserungen zu den oben erwähnten CORBA-Problemen erzielt werden:

- Die Komponenten-Proxy-Methode (vgl. Abschnitt 7.1) ermöglicht *Scripting* von CORBA-Komponenten ohne CORBA-Kenntnisse. Auf diese Weise kann zum Beispiel ein interaktives *debugging* und Testen von CORBA-Komponenten betrieben werden.

- Eine höhere Flexibilität wird durch ein Komponenten Framework (vgl. Abschnitt 2.3.2) erreicht. Das Komponenten Framework nimmt eine klare Trennung zwischen Komponenten und Applikationslogik vor, was ein besseres Abstraktionsniveau zur Folge hat.

- Eine verbesserte Erweiterbarkeit der CORBA-Komponenten kann durch einen eleganten Ansatz, den *Intelligent Proxy* (vgl. Absatz 7.3.1), realisiert werden.
- Die Softwarewiederverwendung wird durch das Komponenten Framework gefördert, da die Applikationslogik von den CORBA-Komponenten getrennt wird. Bei der Komponenten-Entwicklung kann daher mehr Wert auf die Wiederverwendbarkeit gelegt werden. Zudem können *legacy component*s mit dem *Glue* des Komponenten Frameworks integriert werden.

Die vorliegende Arbeit zeigt, dass mit einem relativ einfachen Verbesserungsansatz eine sehr grosse Wirkung erzielt werden kann, wie zum Beispiel der Zugriff auf CORBA-Komponenten ohne CORBA-Kenntnisse oder die Realisierung eines Komponenten Frameworks. Mit Hilfe von diesem Verbesserungsansatz werden zwei nicht triviale zusätzliche Möglichkeiten entwickelt, die CORBA ganz neue Dimensionen verleihen: *Intelligent Proxy* und Agent (*Mobile Agent* vgl. Absatz 7.3.2). Die Entwicklung des *Intelligent Proxy* und des *Mobile Agent* kann als positives Ergebnis betrachtet werden.

1.3 Aufbau der Arbeit

Dieser Abschnitt gibt dem Leser einen kurzen Überblick über die vorliegende Arbeit, die wie folgt in vier Einheiten unterteilt wird:

- Grundlagen (Kapitel 2 bis 4):
  In Kapitel 2 werden die für diese Arbeit wichtigen Begriffe und Konzepte erläutert, welche für das Verständnis der ganzen Arbeit benötigt werden. Das 3. Kapitel stellt das Themengebiet der Skript-Sprachen vor. Besondere Aufmerksamkeit gilt der Skript-Sprache Python, da in dieser Arbeit eine Python-Anbindung an CORBA verwendet wird. Im 4. Kapitel wird die Middleware CORBA eingeführt. Die Architektur und die Verwendung von CORBA wird in Form einer Einführung vorgestellt.
- Problem-Bearbeitung (Kapitel 5 und 6):
  Das 5. Kapitel untersucht anhand ausgewählter Literatur die Probleme von CORBA. Der Problemkreis von CORBA wurde in dieser Arbeit aus zeitlichen Gründen eingeschränkt. Das Kapitel 6 beschreibt einen Lösungsansatz, wobei mit dieser Studie vergleichbare Ansätze zuvor kurz vorgestellt werden.
- Methode (Kapitel 7 und 8):
  Die in dieser Arbeit erstellte Komponenten-Proxy-Methode wird im Kapitel 7 erläutert. Es wird ein Komponenten Framework beschrieben, das die Verwendung von CORBA-Komponenten vereinfacht. Dieser Ansatz erweitert CORBA mit zusätzlichen Möglichkeiten wie zum Beispiel dem *Intelligent Proxy*. Im Kapitel 8 wird anhand einer Fallstudie im Forschungsprojekt FAMOOS (vgl. [DT98]) die Methode verwendet und gezeigt, wie das Komponenten Framework eingesetzt werden kann.
- Ergebnisse (Kapitel 9):
  Im letzten Kapitel wird eine Zusammenfassung der Resultate erstellt. Anhand der Ergebnisse werden offene Probleme und mögliche Fragestellungen für weitere Arbeiten beschrieben.

Kapitel 2

Begriffe und Konzepte

In diesem Kapitel werden die für diese Arbeit wichtigen Begriffe und Konzepte vorgestellt. CORBA ist eine Middleware, die das Entwickeln von verteilten Systemen erleichtert. Mit CORBA erstellt man Client/Server-Applikationen. Ein Server ist eine Komponente und Komponenten können durch Software-Komposition zu einer Applikation zusammengesetzt werden. Ein Komponenten Framework erleichtert die Software-Komposition. Der *Glue* des Frameworks ermöglicht das Einbinden von *legacy components.*

2.1 Verteilte Systeme

Am Anfang der Computertechnologie waren die informationsverarbeitenden Maschinen sehr gross und extrem teuer. Diese Situation änderte sich ab Mitte der 80er Jahre durch die folgenden zwei technologischen Fortschritte:

- Leistungsstarke Mikroprozessoren:
  Nach und nach kamen billige und sehr leistungsstarke Mikroprozessoren auf den Markt. Die Leistung eines solchen Rechners entsprach bald einmal einem mittleren Mainframe.

- Hochgeschwindigkeits-Computernetzwerke:
  Der Fortschritt in der Hochgeschwindigkeits-Netzwerktechnologie ermöglicht das Verbinden von mehreren Rechnern mit hohen Datenübertragungsraten. Man unterscheidet zwischen Lokal Area Networks (LAN) und Wide Area Networks (WAN).

Diese Fortschritte erlauben es, eine grosse Anzahl von billigen Mikrocomputern durch ein Hochgeschwindigkeits-Netzwerk zu verbinden. Stattet man ein solches Netzwerk mit einem geeigneten Betriebssystem aus, dann handelt es sich um ein verteiltes System.
Tanenbaum definiert ein verteiltes System wie folgt (vgl. [Tan95]):

Definition 1 *Ein verteiltes System ist eine Sammlung voneinander unabhängiger Computer, die dem Benutzer des Systems den Eindruck vermitteln, es handle sich um einen einzigen Computer.*

Dabei sind zwei Aspekte wichtig. Erstens werden die Computer als autonom betrachtet; zweitens soll dem Benutzer das Gefühl vermittelt werden, als handle es sich bei dem System um einen einzigen Rechner.

2.2 Middleware

Es hat sich in der Praxis gezeigt, dass verteilte Client/Server-Applikationen sehr schnell sehr komplex werden. Ein Grund liegt in dem grossen Spektrum von Entwurfs- und Entwicklungsalternativen. Daher wird immer häufiger Hilfssoftware eingesetzt, die die Arbeit der Entwickler erleichtert und vereinfacht. Bei der Entwicklung von verteilten Informationssystemen kommt Middleware-Software eine Schlüsselrolle zu (vgl. [BR97]).

Für Middleware gibt es noch kein einheitliches Verständnis (vgl. [Tre97]). Man kann sich aber unter Middleware eine zusätzliche Abstraktionsschicht vorstellen, die zwischen einer Client- oder einer Server-Komponente und dem darunter liegendem Betriebssystem liegt. Die zusätzliche Abstraktionsschicht übernimmt die Kommunikation zwischen Client und Server (vgl. Abbildung 2.1).

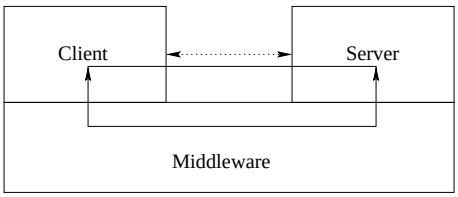

Abbildung 2.1: Eine Middleware verbindet Client und Server

Gemäss Bernstein (vgl. [Ber96]) beinhaltet Middleware hauptsächlich Kommunikationskomponenten, die durch Dienste und Funktionalitäten verwirklicht werden. Dabei unterscheidet Bernstein zwischen den folgenden Middleware-Diensten:

- Kommunikationsdienst
- System-Dienst
- Informations-Dienst
- Ablaufkontroll-Dienst
- Präsentations-Dienst
- Berechnungs-Dienst

Tresch (vgl. [Tre96]) bezeichnet die klassischen Aufgaben von Middleware als Kommunikation zwischen Client und Server sowie den Zugriff auf verteilte Datenbanken:

- Kommunikations-Middleware
  Für die Kommunikation zwischen Client und Server gibt es grundsätzlich zwei Mechanismen: Erstens ermöglicht der *Remote Procedure Call* (RPC) das Ausführen einer Prozedur auf einem entfernten Rechner. Und zweitens beschreibt das *Message Queueing* die asynchrone Kommunikation (Mailbox-ähnlich).

- SQL-Middleware für den Zugriff auf verteilte Daten

Middleware schirmt die Applikation von der Heterogenität der Betriebssysteme und der Netzwerkprotokolle ab. Es werden Standardschnittstellen und Dienste bereitgestellt. Dadurch werden folgende Ziele erreicht:

- Verbinden von verteilten Objekten

- Integration von heterogenen Applikationen

Özsu (vgl: [ODV94]) unterteilt die aktuelle Middleware-Entwicklung aufbauend auf den Grundfunktionalitäten der Kommunikation, Datenverwaltung und Ablaufkontrolle in die folgenden Formen von Middleware:

- Kommunikations-Middleware für verteilte Objekte

- Multimedia-Anfrageschnittstellen

- Data Warehousing

- Agenten und Koordinatoren

- Tansaktions-Monitore (TP-Monitore)

- Middleware mit Anwendungslogik

Die heute sehr bekannten Beispiele von Middleware sind CORBA, DCOM und RMI. Nachfolgend werden diese drei Vertreter kurz vorgestellt.

2.2.1 DCOM

DCOM ist ein Produkt von Microsoft (vgl. [Mic98]). DCOM (Distributed Component Object Model) ist eine Erweiterung von COM (Component Object Model) für verteilte Objekte.

COM (beziehungsweise zukünftig COM+) bietet eine Infrastruktur und Kommunikationsmechanismen für die Entwicklung von Objekten. COM vereinfacht die Applikationsentwicklung, indem ein grosser Teil des Verwaltungsaufwandes der Objekte übernommen wird. Mit Hilfe von COM kann man zum Beispiel in einer Anwendung auf einfache Art und Weise ein Name oder eine Adresse aus einer Datenbank anfordern.

COM definiert einem binären Standard wie Klassen, beziehungsweise Objekte verwaltet und deren Methoden aufgerufen werden. Dadurch ist es möglich Objekte zusammenarbeiten zu lassen,

die in verschieden Programmiersprachen geschrieben sind.

Die Weiterentwicklung DCOM (für Distributed COM) bietet eine Plattform für verteilte Anwendungen, das heisst, DCOM ist eine Entwicklungsunterstützung für Applikationen mit Objekten, welche auf verschiedenen Rechnern leben. Zur Zeit unterstützt DCOM die folgenden Programmiersprachen: Visual C++, Visual Basic und Visual J++[1]. DCOM läuft vorallem auf Windows- und Mac-Rechnern. Die Portierung von DCOM auf andere Plattformen wie Sun Solaris (SPARC) soll gemäss einiger Experten noch Schwierigkeiten bereiten (vgl. [Wit97]). Der Software AG[2] hingegen sind keine solche Schwierigkeiten bekannt.

Viele der Prinzipien und Funktionalitäten von DCOM sind sehr ähnlich zu denjenigen von CORBA (vgl. [Pud97]). DCOM kennt hingegen keine Interoperabilitätsprobleme, da die Kontrolle der Implementierung in der Hand von Microsoft liegt und dadurch eine Abhängigkeit zu einem Entwickler besteht, dies obwohl Microsoft DCOM an die *Open Group*[3] übergeben hat. DCOM spezifiziert ein binäres Interface, das von der Objektimplementierung getrennt wird. Der Quellcode von Objektinterfaces ist nicht offen, das heisst eine Wiederverwendung erfolgt nur über Aggregation auf Binärebene.

2.2.2 RMI

RMI (Abkürzung für Remote Methode Invocation) wurde für den Methodenaufruf von verteilten Objekten entwickelt (vgl. [Sun97]). Das verteilte Objektmodell von RMI wurde in Java integriert. Diese Middleware unterstützt nur Client/Server-Applikationen, die in der Programmiersprache Java geschrieben werden.

RMI ist ab der Version 1.1 vom *Java Developer Kit* (JDK) ein Teil von Java und besteht aus den folgenden drei Java-Paketen: *java.rmi*, *java.rmi.server* und *java.rmi.registry*.

Die Server-Objekte publizieren ihre Java-Interfaces bei der *Registry*, damit die Clients über dieses Interface auf den Server zugreifen können.

RMI besitzt einen *Naming Service* für das Auffinden von Server-Objekten. Mit Hilfe dieses Dienstes können Server durch einen Namen referenziert werden. Für die verteilten Objekte gibt es einen Mechanismus, der den Speicher von nicht mehr verwendeten Objekten freigeben kann (garbage collection).

2.2.3 CORBA

CORBA (Common Object Request Broker Architecture) ist eine Spezifikation (vgl. [OMG97b]) einer sprachunabhängigen Middleware für offene, verteilte und objektorientierte Client/Server-

[1] vgl. $http://msdn.microsoft.com/vstudio/sp/default.asp$
[2] vgl. $http://www.softwareag.com/corporat/solutions/entirex/entirex.htm$
[3] vgl. http://www.activex.org: Die *Active Group* ist eine Tochter von *Open Group*.

Applikationen, deren Interfaces auf ASCII-basieren. CORBA wird in Kapitel 4 ausführlich besprochen.

2.3 Software Komposition

Auf Grund der vielen verschiedenen Hardwarekomponenten und Betriebssystemen diversifiziert sich auch die Software stark. Bei der Softwareentwicklung wird eine Applikation jeweils auf die entsprechende Hardware beziehungsweise das verwendete Betriebssystem zugeschnitten. Für die entwickelten Softwarekomponenten erschwert sich dadurch die Wiederverwendung in einer anderen Umgebung.

Es wird oft versucht, Komponenten mit unterschiedlichen Architekturen auf eine geeignete Art zusammenzusetzen um ein zusammenhängendes Softwaresystem zu erlangen. Dieses Zusammensetzen von beliebigen Softwarekomponenten zu einer Applikation nennt man Software Komposition. In dieser Disziplin hat sich ein neues Programmier-Paradigma herauskristallisiert: der komponentenorientierte Stil. Dieser Stil lässt sich mit der einfachen Formel 2.1 beschreiben.

$$Programm = Komponenten + Skripts \tag{2.1}$$

2.3.1 Komponenten

> Components are designed to be composed.
> *Oscar Nierstrasz and Jean-Guy Schneider* [SN98b]

Eine Komponente kann man sich als “black box” vorstellen. Die “black box” ist ein Programm, von dem man weder die Grösse noch dessen Inhalt kennen muss. Eine Komponente besitzt aber drei wichtige Merkmale, die in der Abbildung 2.2 ersichtlich sind:

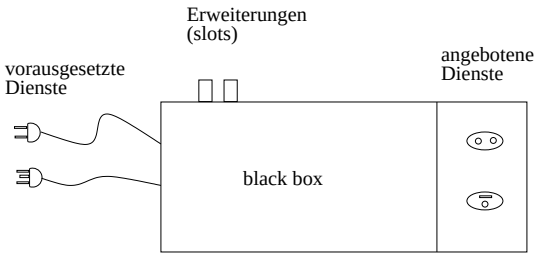

Abbildung 2.2: Komponente als black box

- Angebotene Dienste: Es werden Dienste über ein Interface angeboten. Diese Dienste können von anderen Komponenten benutzt werden.
- Vorausgesetzte Dienste: Eine Komponente kann gewisse Voraussetzungen besitzen, das heisst, es dürfen Dienste von anderen Komponenten vorausgesetzt werden. Sind diese Voraussetzungen nicht erfüllt, so kann die Komponente nicht verwendet werden.

- Erweiterungen: Es soll die Möglichkeit bestehen, die Komponenten mit neuen Fähigkeiten zu erweitern (plug-in).

Zur Veranschaulichung einer Komponente kann man sich zum Beispiel ein Transistorradio vorstellen. Das Radio ist dann eine “black box”, die mit einigen Knöpfen versehen ist. Diese Knöpfe entsprechen den Diensten, die angeboten werden. Damit mit einem Transistorradio überhaupt Musik gehört werden kann, muss das Radio mit Strom versorgt sein. Das heisst, als vorausgesetzter Dienst haben wir zum Beispiel Batterien, die das Radio mit Energie speisen. Erweitern könnte man des Radio zum Beispiel mit externen Lautsprechern.

2.3.2 Komponenten Framework

Ein Komponenten Framework beschreibt, wie Komponenten und Applikationen mit Komponenten erstellt werden. Ein Framework definiert die Bauweise der Komponenten, damit sie in eine vorgegebene Architektur passen. Eine Applikation besitzt meist eine Menge von Komponenten. Zusammengesetzt werden die Komponenten dann in einem Skript. Die Abbildung 2.3 zeigt ein ideales Bild des Komponenten Frameworks, das aus den folgenden vier Elementen zusammengesetzt ist:

- Die Architektur eines Komponenten Frameworks beschreibt, dass eine Applikation aus zwei Bausteinen besteht: Komponenten und Konnektoren (vgl. [GS96]). Komponenten werden durch Skripts verbunden (connected). Es wird zudem erläutert, wie die Komponenten gebaut, beziehungsweise verwendet werden. Die Applikationslogik (Konnektoren) befindet sich in einem Skript.
- Mit Koordination werden Abstraktionen bezeichnet, die nebenläufige Prozesse synchronisieren und deren Interaktion steuern, wie zum Beispiel bei UNIX die Mechanismen von *piping* und *filtering.*
- Der *Glue* adaptiert bestehende, veraltete Komponenten (legacy applications), die nicht zu diesem Framework gehören. Diese Adaption wird oft mit einem sogenannten *Wrapper* (vgl. Adapter Pattern in [GHJV95]) realisiert.
- Skripts fügen dann schliesslich die bereits aufeinander abgestimmten Komponenten zusammen.

2.4 Legacy Komponenten

Im letzten Abschnitt wurde ein Komponenten Framework beschrieben. Der *Glue* des Komponenten Frameworks wird für die Adaption von *legacy component*s verwendet.

Unter einer *legacy component* (*legacy* engl. Vermächtnis) versteht man eine bestehende Software-Komponente, die vererbt wurde und die nützlich und wertvoll ist. Eine solche Komponente kann meistens aus Kompatibilitätsgründen nicht direkt in eine aktuelle Applikation übernommen

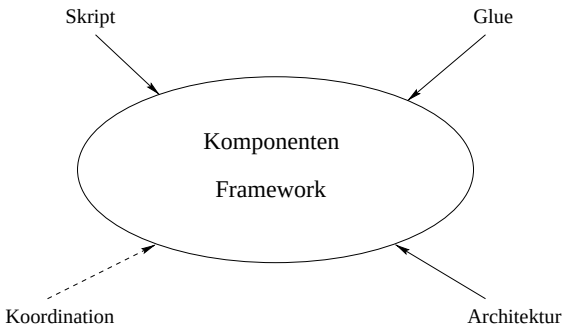

Abbildung 2.3: Ein ideales Komponenten Framework

werden. Eine *legacy componet* muss daher oft verändert oder umgewandelt werden.

*Legacy component*s weisen meist eines der folgenden typischen Probleme (vgl. [Nie98]) auf:

- Der Entwickler der Software-Komponente arbeitet nicht mehr in derselben Firma.
- Die Komponente wurde in einer anderen Programmiersprache geschrieben.
- Teile der verwendeten Methoden sind veraltet oder sind nicht mehr korrekt.
- An dem System wurden viele Veränderungen beziehungsweise Erweiterungen gemacht.
- Es gibt meist keine oder nur eine mangelhafte Dokumentation.
- Die Komponente läuft nur auf einer anderen Plattform.

Viele Firmen haben ein grosses Problem, wenn sie zum Beispiel ihre Geschäftsprozesse verändern. Probleme entstehen bei der Anpassung der Software. Es stellt sich hier die folgende Frage: Muss die ganze Software von Grund auf neu geschrieben werden, oder kann man die bestehende Altsoftware anpassen und Teile wiederverwenden?

Es wird oft entschieden, dass alles von Grund auf neu implementiert wird. Diese Entscheidung hat den Vorteil, dass die Anwendungen danach dem neusten technologischen Stand entsprechen. Das ist zum Beispiel so, wenn ein Informationssystem an das Jahr 2000-Problem angepasst werden muss. Ein weiterer Vorteil besteht darin, dass man so eine saubere, gut strukturierte Lösung bekommt. Es ist oft der Fall, dass eine bestehende Applikation dauernd verbessert und verändert wird. Solche Applikationen sind meist so komplex, dass sie für die Wartung (zu) teuer und nicht flexibel genug werden. Nachfolgend werden Nachteile einer Neuentwicklung aufgezählt:

- Es entstehen hohe Kosten.
- Alle Kinderkrankheiten der neuentwickelten Software müssen wieder eliminiert werden.
- Es kann Jahre dauern, bis das System entwickelt und danach auch zuverlässig ist.

Solche Nachteile bewegen viele Firmen dazu, bestehende Altsoftware wieder zu verwenden. Zudem haben sich die bestehenden Komponenten schon über Jahre bewährt und sind meist fehlerfrei. Es gibt aber auch Systeme, bei denen es in Anbetracht ihrer Komplexität riskant ist, diese von Grund auf neu zu entwickeln (vgl. “Pattern for legacy Integration” in [MM97]). Solche Projekte werden meist zu teuer und/oder laufen Gefahr zu scheitern.

Oft wird die bestehende Altsoftware angepasst und wiederverwendet. Zur gleichen Zeit beginnt man die Software von Grund auf neu zu entwickeln. Nach und nach werden neue Komponenten in das System integriert und die alten ersetzt. Dieser Vorgang dauert so lange, bis am Schluss das ganze System erneuert ist.

Eine *legacy application* stellt einen gewissen Wert oder Nutzen dar, daher will man diese Komponenten in einer Entwicklung integrieren. Die Integration von *legacy applications* ist nicht einfach und erfordert eine Umwandlung (Adaption) der bestehenden Altsoftware. Diese Umwandlung von *legacy applications* heisst Wrapping (Verpackung, Umhüllung). Das Ziel vom *Wrapping* ist die Wiederverwendung von bestehender Altsoftware.

Es ist nicht einfach eine Applikation aus bestehenden Komponenten zu bauen. Diese Problematik ist unter *Architectural Mismatch* bekannt (vgl. [GAO95]). Um Altsoftware in einem neuen Software-System zu verwenden, sind beispielsweise die folgende Schritte notwendig (vgl. [Sne96]):

- Analyse des Altsystems
- Spezifizikation der Schnittstellen (zum Beispiel als CORBA-IDL)
- Veränderung der Altkomponente, so dass diese einzeln aufgerufen werden kann
- Entwicklung eines Wrappers, der die Aufrufe der Altkomponente umsetzt

Kapitel 3

Skript-Sprachen

> Increases in computer speed and changes in the application mix are making scripting languages more and more important for applications of the future.
> *John K. Ousterhout*

Skript-Sprachen, *Glue*, Architektur und Koordination sind die vier Elemente, die ein ideales Komponenten Framework bilden (vgl. Absatz 2.3.2). Skript-Sprachen werden für die komponentenorientierte Programmierung benötigt.

In diesem Kapitel werden zu Beginn die wichtigsten Eigenschaften der Skript-Sprachen erläutert. Anschliessend werden wichtige Vertreter von Skript-Sprachen kurz vorgestellt. Am Ende dieses Kapitels werden die wichtigsten Merkmale von Python eingeführt, da Python die in dieser Arbeit verwendete Skript-Sprache ist.

3.1 Einleitung

Skript-Sprachen, wie zum Beispiel AXLE, AWK, Sed, Unix Shell und HyperTalk (vgl. [Nie96]), gibt es schon seit längerer Zeit. Die Liste der Skript-Sprachen ist gross. Dieser Typ von Programmiersprache konnte sich lange Zeit nicht so richtig durchsetzen, da die Rechnerleistung für die interpretierten Skript-Sprachen zu langsam war. Erst in den letzten fünf Jahren erfreut sich die *scripting-community* einem regelrechten Aufschwung. Diese Trendwende lässt sich auf verschiedene Faktoren zurückführen (vgl. [Ous97]):

- Die Hardware wird immer billiger und deren Leistung immer grösser. Das hat beispielsweise zur Folge, dass der verfügbare Speicher kein Flaschenhals mehr darstellt.
- Heutzutage haben die grafischen Oberflächen (GUI) einen enorm wichtigen Stellenwert. Mit Hilfe von Tk kann auf eine einfache Art und Weise eine grafische Benutzeroberfäche erstellt werden.
- Das Internet boomt in einem Ausmass, das vor zehn Jahren die kühnsten Optimisten nicht prognostiziert hätten. Für Server-seitige Web-Anwendungen werden hauptsächlich Skript-

Sprachen zum Erstellen von CGIs verwendet. Java-Applets werden im Gegensatz dazu oft auf der Client-Seite eingesetzt.

- Aus der Komponentenvielfalt entwickelt sich ein neues Programmierparadigma: der komponentenorientierte Programmierstil (vgl. Tabelle 3.1 aus [Nie96]).

Stil	Programm
Imperativ	Programm := Algorithmus + Daten
Funktional	Programm := Funktion ∘ Funktion
Logisch	Programm := Fakten + Regeln
Objektorientiert	Programm := Objekte + Nachrichten
Komponentenorientiert	Programm := Komponenten + Skripts

Tabelle 3.1: Wichtige Programmier-Paradigmen

Skript-Sprachen werden immer häufiger eingesetzt. Aber was sind Skript-Sprachen genau? Es ist schwierig, eine Skript-Sprache exakt zu definieren. Die Ansichten über die Definition der Skript-Sprachen sind nicht einheitlich. Nachfolgend werden zwei wichtige Definitionen dargestellt.

Definition 2 (Clemens Szyperski) *Unlike mainstream component programming, scripts do not introduce new components, but simply wire existing ones. Scripts can be seen as introducing new behaviour but no new states.*

Aus Definition 2 gehen zwei wichtige Aspekte hervor. Erstens macht Clemens Szyperski die Unterscheidung zwischen Komponenten-Programmiersprachen (auch *system programming* oder traditionelle Programmiersprachen genannt) und Skript-Sprachen. Zweitens haben Skript-Sprachen den Zweck, Komponenten zu verbinden und nicht etwa um Neue zu erstellen.

Auch John K. Ousterhout unterscheidet wie Clemens Szyperski zwischen diesen zwei Sprachtypen. Die Grafik 3.1 aus “Scripting: Higher Level Programming for the 21st Century” (vgl. [Ous97]) zeigt diese zwei Typen. Skript-Sprachen sind meist schwach typisierte Sprachen. Man nennt sie auch dynamisch typisiert, da der Datentyp der Objekte beziehungsweise der Methoden erst zur Laufzeit verifiziert wird. Im Gegensatz dazu werden die traditionellen Sprachen von einem Compiler unter anderem auf Datenintegrität geprüft und sind daher stark typisiert.

Definition 3 (Jean-Guy Schneider, Oscar Nierstrasz) *Scripting languages are designed for configuring and connecting components.*

In Definition 3 wird hauptsächlich der Zweck der Skript-Sprachen hervorgehoben. Skript-Sprachen adaptieren und verbinden Komponenten, welche meist aus anderen Programmiersprachen stammen.

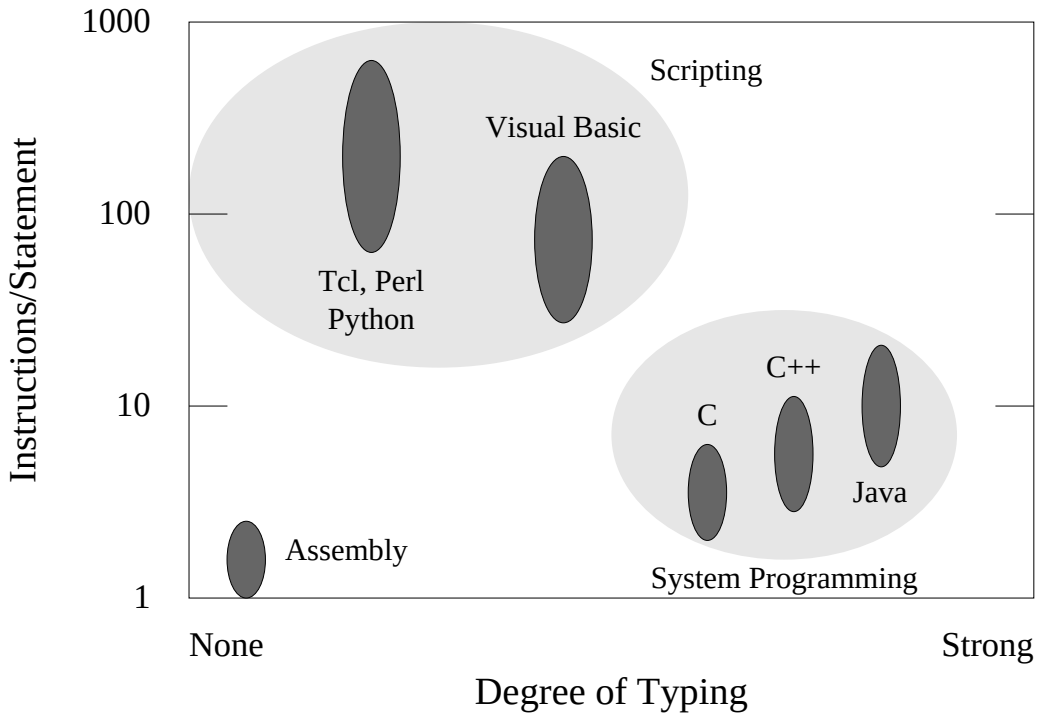

Abbildung 3.1: *Scripting* vs. *System Programming Languages* von John Ousterhout

Jede Programmiersprache kann im Prinzip zum Verbinden von Komponenten verwendet werden. Wie unterscheiden sich nun traditionelle Sprachen von Skript-Sprachen?

3.2 Eigenschaften

Die Erfahrung hat gezeigt, dass eine Programmiersprache erst dann als Skript-Sprache bezeichnet werden kann, wenn sie auch zu diesem Zweck verwendet wird. Skript-Sprachen besitzen meistens die nachfolgenden typischen Eigenschaften (vgl. [SN98b]):

- erweiterbar: Skript-Sprachen sind auf zwei Arten erweiterbar: Einerseits ist eine Erweiterung der Skript-Sprache leicht möglich, das heisst benutzerdefinierte Prozeduren oder Funktionen (meist in C) lassen sich als Modul oder sogar in den Sprachkern aufnehmen. Dies ist vor allem wichtig, wenn die Geschwindigkeit eine Rolle spielt. Andererseits lassen sich Skripts zu Komponenten erweitern, so dass sie in einem Komponenten Framework verwendet werden können.

- adaptieren und verbinden Komponenten: Skript-Sprachen wurden entwickelt zum Adaptieren und Verbinden von Komponenten (*connecting components*).

- Komponenten Framework: Eine Skript-Sprache ist ein wichtiger Bestandteil eines Komponenten Frameworks.

- in anderen Programmiersprachen integrierbar: Skript-Sprachen lassen sich in anderen Programmiersprachen wie zum Beispiel C oder Java integrieren. Man nennt die Skript-Sprachen dann auch *embeddable.* Auf diese Weise können die traditionellen Programmiersprachen von den Fähigkeiten der Skript-Sprachen profitieren.

- interpretiert oder dynamisch compiliert: Die Programme müssen nach der Implementierung nicht selbst compiliert und gelinkt werden wie das in den traditionellen Sprachen der Fall ist. Ein Skript kann sofort gestartet werden, dabei wird es einerseits zeilenweise interpretiert oder andererseits zur Laufzeit compiliert. Die Ausführungsgeschwindigkeit ist daher nicht ganz so schnell wie zum Beispiel bei einem C-Programm.

- portabel: Programmcode (ein Skript) lässt sich ohne Veränderung auf einem anderen Rechnertyp ausführen, da das Programm nicht in einen Maschinencode umgewandelt wird. Skripts werden erst während der Ausführung interpretiert oder dynamisch compiliert.

- dynamisch typisiert: Es ist keine Variablendeklaration erforderlich. Eine Variable wird definiert, indem man ihr einen Wert zuweist. Bei den meisten Skript-Sprachen ist es so, dass die gleiche Variable zum Beispiel einmal Zahl und einmal String als Wert haben kann.

- frei verfügbar: Der Gebrauch dieser Programmiersprachen ist meist kostenlos. Das bedeutet natürlich nicht, dass Skript-Sprachen nichts Wert sind. Im Gegenteil; die Entwicklung einer Skript-Sprache ist eine seriöse Arbeit. Viele Programmierer rund um die Welt decken Fehler auf und machen Verbesserungsvorschläge, da die Sprachen frei verfügbar sind. Die meisten Skript-Sprachen sind während vielen Jahren intensiv entwickelt worden.

Ein Skript zeichnet sich meistens durch seine kompakte, flexible Art aus. Skripts werden hauptsächlich in den folgenden Anwendungsbereichen verwendet:

- dynamische Web-Anwendungen
- Testen von Applikationen
- Systemadministrationsaufgaben
- Entwicklung von Prototypen (vor allem grafische Benutzeroberflächen)
- Softwareentwicklung mit Hilfe eines Komponenten Frameworks

3.3 Ein Überblick

Die ersten Skript-Sprachen waren “shell languages” und “command languages”. Diese Sprachen wurden vor allem für Betriebssysteme und spezielle veränderbare Applikationen verwendet. Beispielsweise wurde 1971 die Unix-Shell als Benutzerinterface (Shell) und als Skript-Sprache entwickelt. Man verwendete diese Sprachen um eine Reihe von Befehlen abzuarbeiten, ohne dass eine Interaktion mit dem Benutzer stattfand. Die abzuarbeitenden Befehle schrieb man in Dateien, damit die gleiche Befehlssequenz mehrere Male ausgeführt werden konnte. Diese Dateien

nennt man auch “batch files”. In DOS-Systemen erkennt man diese Dateitypen an der Dateiendung “.bat”. Ein Beispiel ist die Datei “autoexec.bat”. Diese Datei wird beim Bootvorgang automatisch ausgeführt.

In der Folge wurden sogenannte “text processing languages” wie zum Beispiel AWK, Perl oder Sed benötigt. Dieser Typ von Programmiersprache ist auf die Verarbeitung von Texten und Dateien spezialisiert. “text processing languages” können auch sehr grosse Dateien bearbeiten.

Heute fasst man alle Programmiersprachen mit solchen Fähigkeiten zusammen und nennt sie “Skript-Sprachen”. Es gibt viele Vertreter von Skript-Sprachen. Am meisten verwendet und am bekanntesten sind: AppleScript, VB, Tcl/Tk, Perl und Python (vgl. [LS97]).

Im Artikel “Script im Griff” von Cameron Laird und Kathryn Soraiz (vgl. [LS98b]) werden einige Script-Sprachen im Überblick dargestellt. Die Tabelle 3.2 basiert auf diesem Artikel und zeigt zusammenfassend die wichtigsten Unterscheidungsmerkmale von Tcl/Tk, Perl und Python. Nachfolgend werden diese Skript-Sprachen kurz vorgestellt.

	Python	Perl	Tcl/Tk
Paradigma	Objektorientiert	Imperativ	Imperativ
Konzept/Motto	Everything is an object.	There is more than one way to do it.	Everything is a string.
verteilte Programmierung	ja	ja	ja
Hauptzweck	Administration	Textprocessing	GUI
Vorteil	sauber, portierbare Sprache	verbreitet, dominant in CGI	einfache Syntax, erweiterbar
Nachteil	Fachkenntnisse noch klein	Maintenance weniger beachtet	umständlich für einfache Arithmetik

Tabelle 3.2: Python, Perl und Tcl im Vergleich

Perl

Perl (Practical Extraction and Report Language) ist eine weit verbreitete Skript-Sprache. Perl wird vor allem in der Unix-Welt verwendet. Sie wurde mit grosser Sorgfalt entwickelt und ist daher eine sehr sichere Sprache (vgl. [WCS96]). In der letzten Zeit wurde Perl hauptsächlich für die Erstellung von CGIs (Common Gateway Interface) verwendet. “There is more than one way to do it” ist das Motto von Perl.

Mit Perl kann man sehr einfach Texte, Dateien und Prozesse manipulieren. Perl ist optimiert um Informationen aus Dateien zu extrahieren. Ein sogenanntes *pattern matching* erlaubt ein sehr schnelles durcharbeiten von grossen Mengen von Daten. Dateien können manipuliert werden wie zum Beispiel Umbenennen von Dateinamen oder Ändern von Zugriffsrechten. Perl erlaubt das

Erzeugen, Steuern und Vernichten von Prozessen. Sie eignet sich für die meisten Systemadministrationsarbeiten und vor allem für *Textprocessing.*

Die Länge eines Datentyps ist nicht auf irgendeine Zweierpotenz limitiert, wie das in manchen Programmiersprachen der Fall ist. Variablennamen, Strings und Arrays dürfen unbeschränkt lange werden, das heisst bis man keinen Speicher mehr hat. Binäre Datentypen bieten keine Probleme. Perl-Programme sind sicher, da sie unsichere Datentypen durch einen "tracing"-Mechanismus eruieren können. Dieser Mechanismus beugt vielen Sicherheitslöchern vor.

Diese Programmiersprache läuft auf den gängigsten Betriebssystemen und Hardware-Architekturen. Perl ist eine dynamisch compilierte Sprache, das heisst, dass Perlprogramme (Skripts) erst zur Laufzeit compiliert werden und nicht explizit compiliert werden müssen. Sie ist eine prozedurale Sprache, die ab der Perl Version 5 auch die objektorientierte Programmierung erlaubt.

Perl kombiniert die besten Fähigkeiten von C, sed, awk, und sh. Die Ausdruck-Syntax ist sehr ähnlich wie die Ausdruck-Syntax von C.

Tcl/Tk

Im Jahre 1988 begann John K. Ousterhout mit der Entwicklung von Tcl/Tk (vgl. [Ous94]). Tcl/Tk besteht aus zwei Software-Paketen:

- Tcl (genannt "tickel") ist eine interpretierte, stringbasierte Skript-Sprache.
- Tk erweitert Tcl um grafische Benutzeroberflächen.

Bei der Entwicklung von Tcl (Abkürzung für Tool Command Language) wurde vor allem darauf geachtet eine wiederverwendbare Befehlssprache zu schreiben. Der Kern von Tcl besteht aus einer Bibliothek von C-Prozeduren. Sie ist eine leicht erweiterbare Sprache, die hauptsächlich als Kontroll- und Erweiterungssprache benutzt wird. Anwendungen werden aus einer Kombination von bestehenden und einigen neuen Funktionen geschrieben. Tcl verfolgt das Motto: "Everything is a string.". Tcl eignet sich für Stringmanipulationen und "pattern matching" mit Hilfe von regulären Ausdrücken.

Tcl wird aber hauptsächlich wegen seines X Window Toolkits Tk (sprich "tee-kay") verwendet. Tk ist eine grafische Erweiterung von Tcl. Sie ermöglicht auf eine einfache Weise, grafische Benutzerinterfaces zu schreiben. Auch Tk besteht wie Tcl nur aus einigen grundlegenden Befehlen. Mit diesen Funktionen kann man sehr einfach grafische Objekte erzeugen, welche Widgets genannt werden. Diese Widgets lassen sich durch einen Geometrie-Manager gestalten, damit man auf einfache Weise das Layout erhält. Tk erlaubt grafische Benutzerschnittstellen zu erstellen und diese mit einer Tcl-Applikation zu kombinieren.

3.4 Python

Python[1] wurde von Guido van Rossum während den Weihnachtsferien 1989/1990 entwickelt (vgl. [vR96]). Python wurde zum Testen des neu entwickelten, verteilten Betriebssystems Amoeba[2] verwendet. Bei der Programmierung von Python wurde besonderen Wert auf die Erweiterbarkeit der Sprache und deren Programme gelegt.

In Python verhalten sich Funktionen, Klassen, Methoden und Module wie Daten, das heisst man kann sie auch als Argumente an Funktionen übergeben. Man nennt das auch "first class values".

Python ist eine interpretierte, objektorientierte Programmiersprache (vgl. [Lut96] und [vLF97]). Mehrfachvererbung (multiple inheritance) wird unterstützt, das heisst eine Klasse kann mehrere Basisklassen haben. Die Syntax von Python ist kurz und sehr aussagekräftig. Der Programmcode ist immer schön formatiert, da in Python die Blockstruktur durch das Einrücken gebildet wird. Auf einer Zeile steht normalerweise nur ein Kommando, das hat zur Folge, dass die Programme viel besser lesbar sind. Python hat eine automatische Speicherverwaltung.

Der Pythoninterpreter ist sehr fehlertolerant, das heisst, er unterstützt einen Ausnahmebehandlungsmechanismus mit *try*- und *except*-Anweisungen. Fehler in Programmen können auf diese Weise sanft abgefangen werden.

Es gibt verschiedene Möglichkeiten die Argumente einer Methode zu definieren. Man unterscheidet wie folgt:

- Argumente können als optional deklariert werden, das heisst man kann Standardwerte für die Argumente (default arguments) angeben. Beim Methodenaufruf können die Argumente mit Standardwerten auch weggelassen werden. `def add(a=1, b=1):` könnte zum Beispiel der Kopf einer Methodendefinition sein, die zwei Zahlen addiert.

- Eine Methode kann auch mit variabler Anzahl Argumente deklariert sein. Die Anzahl der Argumente wird beim Methodenaufruf bestimmt. `def sum(*args):` könnte zum Beispiel der Kopf einer Methodendefinition sein, die eine Liste von Zahlen zusammen summiert.

- Die Argumente können auch als ungeordnete Menge definiert werden. Beim Methodenaufruf braucht man dann sogenannte Schlüsselwortparameter, da die Reihenfolge der Argumente keine Rolle spielt. `def reg(**sw):` könnte zum Beispiel der Kopf einer Methodendefinition sein. Mit `reg(name= "Held", grösse=182)` könnte eine Person registriert werden.

[1] Der Name Python kommt von der britischen Comedy-Truppe: Monty Pythons Flying Circus.

[2] Amoeba entstand 1981 an der Vrije Universität von Amsterdam, Niederlande, in einem Forschungsprojekt im Bereich der verteilten und parallelen Berechnung unter der Leitung von Andrew S. Tanenbaum.

- Es gibt natürlich auch ganz “normale” Argumente, wie bei anderen Programmiersprachen auch.

In Python gibt es zwei dynamische Konstrukte:

- Die *exec*-Anweisung ermöglicht, Anweisungen zur Laufzeit auszuführen, wie zum Beispiel:

```
>>> exec "print 7"
7
```

- Um Ausdrücke dynamisch auszuwerten, verwendet man die Funktion *eval*. Ein möglicher Ausdruck sieht wie folgt aus:

```
>>> eval("1+2")
3
```

Python ist nicht nur eine Skript-Sprache, denn sie eignet sich auch ausgezeichnet für das Erstellen von grossen Applikationen. Die Grafik 3.2 von Cameron Laird und Kathryn Soryiz (vgl. [LS98a]) zeigt diese Ausprägung von Python sehr schön.

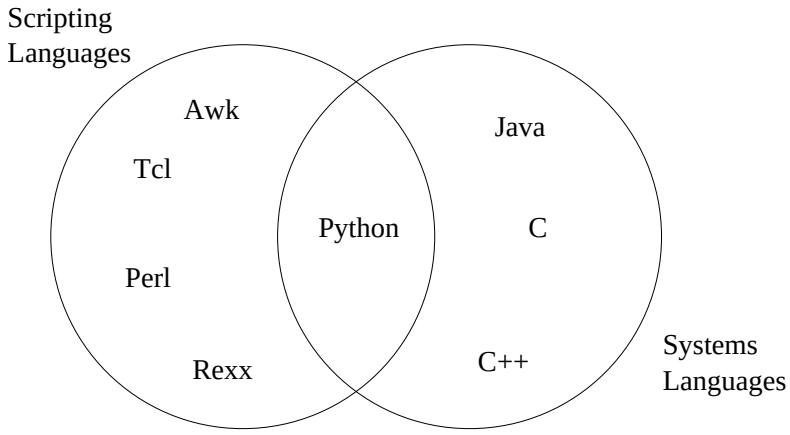

Abbildung 3.2: Python, die Skript- bzw. System-Programmiersprache

Einige der wichtigsten Merkmale von Python als Skript-Sprache sind nachfolgend aufgezählt:

- interpretiert, bzw. compiliert zu Byte-Code
- automatische Speicherverwaltung
- dynamisch typisiert

Eigenschaften von Python als traditionelle Programmiersprache sind:

- Objekt-orientiert, Mehrfachvererbung, Polymorphie
- Sprachanbindungen (language mapping) an CORBA (ILU, Fnorb) und COM/ActiveX (PythonWin)

Python kann sowohl im interaktiven Modus als auch in einem Skript als Programm ausgeführt werden. Die interaktive Arbeitsweise sieht wie folgt aus:

```
Python 1.5.1 (#14, Oct  7 1998, 11:33:24)  [GCC 2.7.2] on sunos5
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>> print "Hallo Welt!"
Hello World!
>>> def inc(x):
...   return x+1
...
>>> map(inc, range(5, 33, 3))
[6, 9, 12, 15, 18, 21, 24, 27, 30, 33]
>>
```

Ein Pythonprogramm (auch Skript genannt) wird in einer Datei gespeichert. Auf diese Weise kann dieselbe Befehlsfolge mehrere Male ausgeführt werden. Kleine Skripts eignen sich vor allem zum Testen von anderen Programmen. Python erzeugt beim ersten Programmdurchlauf einen Plattform unabhängigen Byte-Code. Die Abbildung 3.3 zeigt ein Beispiel eines Pythonprogrammes, das das Suchen und Ersetzen eines Strings in einer angegebenen Datei ermöglicht.

Es ist ein grosser Pluspunkt von Python, dass umfangreiche Bibliotheken vorhanden sind. Es wurde bei der Entwicklung von Python darauf geachtet, dass der Sprachkern möglichst klein gehalten wurde. Mit Hilfe von Modulen gelang es die eigentliche Sprache zu entlasten. Die Bibliothek enthält vor allem die folgenden Modul-Gruppen:

- Strings, zum Beispiel *string, regex* oder *regsub*
- Betriebssystem, etwa *os, time* oder *socket*
- UNIX: *posix* oder *pwd*
- Internet: *cgi, urllib* oder *ftplib*
- Multimedia, wie zum Beispiel *audioop, jpeg* oder *imageop*
- GUI: *Tkinter, stdwin* oder *rect*

Python besitzt in Pythonwin[3] eine DCOM-Integration. Diese Integration befindet sich aber noch in einem experimentellen Status (vgl. [vLF97]). In Zukunft wird es sehr wichtig sein, Komponenten von verschiedenen Autoren in einem System zu integrieren. Solche Aufgaben werden

[3]Pythonwin ist eine Version von Python. Es ist ein reines Windows-Programm, das von Mark Hammond unter Visual C++ entwickelt wurde.

```
# search and replace a string in a file
# Standard/built-in modules.
import string, sys, os
# Function
def search_replace(x, y, z):
  # x: text to scan
  # y: string that has to be replaced
  # z: replacing string
  l = string.splitfields(x, y)
  return string.joinfields(l, z)
# Main
if __name__ == '__main__':
  # is there a file and two strings?
  if (len(sys.argv[1:])< 3):      # no
    print "usage:  search_replace.py <file> <search> <replace>"
  elif (len(sys.argv[1:]) == 3):  # yes
    filename = sys.argv[1]
    search   = sys.argv[2]
    replace  = sys.argv[3]
    # is 'filename' really a file?
    if (os.path.isdir(filename)==0 and os.path.isfile(filename)==1):
      f = open(filename)
      text = f.read()
      f.close()
      print search_replace(text, search ,replace)
    else:
      print "The first argument must be a file!"
  else:                           # no
    print "usage:  search_replace.py <file> <search> <replace>"
```

Abbildung 3.3: Ein Python-Skript: Suchen und ersetzen von Wörtern in einer Datei.

mit verteilter Programmierung gelöst und dazu kann Middleware wie DCOM oder CORBA verwendet werden.

Etwas besser sieht es mit der Middleware CORBA aus. Mit dem ILU (Inter Language Unification) von der Firma Xerox (vgl. [JS97]) und dem Fnorb der Universität von Queensland aus Australien (vgl. [Chi98]) besitzt Python zwei recht zuverlässige ORBs. Leider ist die Anbindung (language mapping) von IDL nach Python noch kein Standard. In "Python Language Mapping" reichten Martin Löwis, Bill Janssen und Martin Chilvers (vgl. [CJvL98]) einen Vorschlag für diese Abbildung der OMG ein (Stand der Skript-Sprachen-Anbindung an CORBA vgl. im Anhang B.6).

Für das Internet sind vor allem sogenannte CGI-Skripts (Common Gateway Interface) von grosser Bedeutung. Mit Python kann man sehr einfach solche Skripts schreiben (vgl. [WvRA96]). CGI ist eine Schnittstelle zwischen einem Web-Server und einem externen Programm. Der Web-Server startet das externe Programm (CGI-Skript) und übergibt ihm die Umgebungsvariablen. Python enthält in seiner Standardbibliothek ein Modul *cgi*. Dieses Modul vereinfacht das Lesen und Verarbeiten der Umgebungsvariablen. Nach der Verarbeitung erzeugt das Programm eine HTML-

Antwort.

Es hat sich in der Vergangenheit gezeigt, dass die Benutzer lieber mit grafischen Oberflächen arbeiten als mit kommandozeilenorientierten Programmen. Python bietet verschiedene Möglichkeiten ein GUI (graphical user interface) zu erstellen. Grundsätzlich gibt es zwei Lager. Erstens gibt es Module, die auf der *Microsoft Foundation Class* (MFC) und nur auf Microsoft-Betriebssystemen laufen. Und zweitens hat man verschiedene Module die auf Tcl/Tk basieren wie zum Beispiel das Modul *Tkinter*. Mit diesem Modul kann man sehr einfach eine grafische Benutzeroberfläche erstellen.

Zusammenfassend werden anschliessend die wichtigsten Eigenschaften von Python herausgehoben, die in einem Komponenten Framework benötigt werden:

- Komponenten können einfach zusammengestellt und verbunden werden (composition).
- Python eignet sich um Komponenten zu adaptieren (*Gluing*).
- Es können sehr einfach grafische Benutzeroberflächen erstellt werden.
- Python eignet sich für das Testen von Komponenten.
- Komponenten können dynamisch ausgeführt werden.

Kapitel 4

CORBA

> Communication software and distributed services for next generation applications must be reliable, efficient, flexible, and reusable. These requirements motivate the use of the Common Object Request Broker Architecture (CORBA). *Silvano Maffeis and Douglas C. Schmidt*

4.1 Einleitung

Als man noch mit Mainframe-Systemen arbeitete, war jeder Benutzer über ein Terminal mit einem zentralen Rechner verbunden. Mit stetig steigender Anzahl von Benutzern und zunehmender Komplexität der Anwendungen wurden Rechner mit einer immer grösseren Leistung verwendet. Mit der Zeit wurden für gewisse Arbeiten immer mehr preisgünstige, leistungsstarke Mikrorechner (PC) eingesetzt. Das führte dazu, dass man alltägliche Arbeiten, wie zum Beispiel Briefe schreiben oder Tabellen erstellen, auf dem PC erledigte und für Hauptgeschäftsanwendungen mittels Terminal auf den Mainframe zugriff.

Durch die Entwicklung von Netzwerktechniken entstand die Möglichkeit, Rechner auch über grosse Entfernung miteinander zu verbinden. Einerseits können nun PC's mit PC's und anderseits auch PC's mit Mainframes verbunden werden. Für den Zugriff auf einen entfernten Rechner wurden daraufhin sogenannte Client/Server-Architekturen entwickelt. Der Grund für diesen Wandel liegt in den vielen Vorteilen die verteilte Anwendungen im Gegensatz zu den Mainframes bringen (vgl.[Tan95]).

Am Anfang der Client/Server-Ära entwickelte man Systeme, bei denen der Client einen direkten Zugriff auf den Server hatte. Man nannte sie auch 2-*Tier*[1] Architekturen (vgl. Abbildung 4.1). Der Client enthielt die ganze Anwendungs- und Präsentationslogik. Der Server verarbeitete lediglich spezifische Anfragen und führte zum Beispiel eine Datenbankabfrage oder eine spezielle Berechnung durch. Auf Grund dieses Ungleichgewichtes zwischen Client und Server entstand auch der Name "Fat Client". Diese 2-Tier Architekturen hatten folgende Nachteile:

[1] engl. für Stufe, das heisst zweistufig

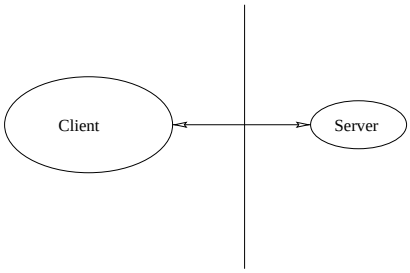

Abbildung 4.1: 2-Tier Architektur

- inhärente Komplexität der Entwicklung verteilter Anwendungen
- kleinere Leistung von Servern, was oft zu Leistungsengpässen führte
- Sicherheitsprobleme
- grosse Netzwerkbelastung

In einer weiteren Stufe baute man 3-Tier Client/Server-Systeme. Das bedeutete für die Clients, dass sie nicht mehr direkt auf den Server zugreifen konnten, da eine weitere Schicht, eine sogenannte Applikationslogik, dazwischen gelegt wurde. Der Client bestand nur noch aus der Präsentationslogik und wurde demzufolge “Thin Client” genannt. Die Abbildung 4.2 verdeutlicht diese neue Architektur.

Seit einiger Zeit baut man n-Tier Architekturen (vgl. Abbildung 4.3). In grösseren Systemen drängt es sich auf, dass die Applikationslogik nicht nur aus einem riesigen Gebilde besteht. Eine grössere Anzahl kleinerer Dienste entstanden aus der grossen Applikationslogik. Ein solcher modularer Dienst kann unabhängig von anderen Diensten entwickelt werden. Die Gesamtheit aller dieser Dienste repräsentiert die Applikationslogik.

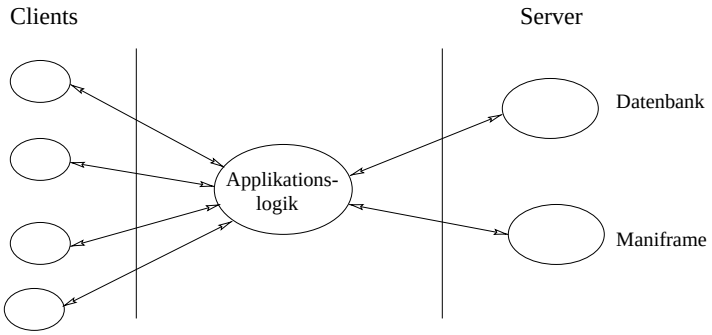

Abbildung 4.2: 3-Tier Architektur

n-Tier Architekturen versprechen die nachfolgenden Verbesserungen:

- bessere Lastverteilung
- geringere Netzwerkbelastung

Diese n-Tier Architekturen haben aber nicht nur Vorteile, hier einige Nachteile:

- hoher Administrationsaufwand
- unzureichende Entwicklungs- und Wartungswerkzeuge

n-Tier Architekturen werden durch die Middleware-Technologie unterstützt. Die heute bekanntesten Typen von Middleware sind CORBA, COM und RMI (vgl. Abschnitt 2.2 auf Seite 5).

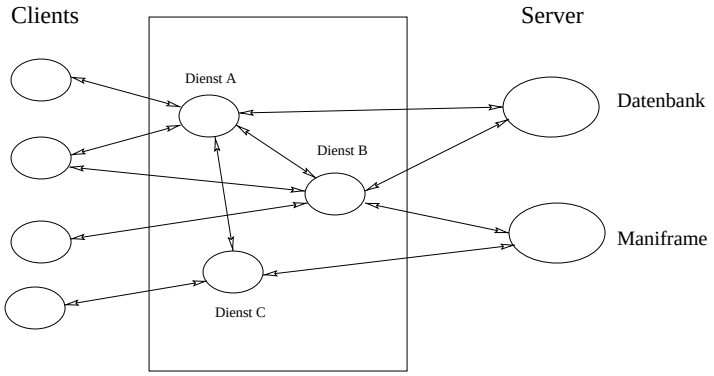

Abbildung 4.3: n-Tier Architektur

4.2 Architektur

CORBA ist die Abkürzung für *Common Object Request Broker Architecture*. Ein Standard entsteht in der Informatik normalerweise aus einem Produkt (meist einer einzelnen Firma), das sich bewährt hat. Ganz anders ist es mit CORBA. Diese Architektur wurde interessanterweise von einem Konsortium, der Object Management Group (kurz OMG), und nicht von einer einzelnen Firma definiert. Die OMG wurde 1989 gegründet und hat heute über 750 Hersteller und Firmen als Mitglieder. CORBA ist keine Implementierung sondern lediglich eine Spezifikation. Die Implementierung dieser Spezifikation liegt bei den einzelnen ORB-Herstellern (Object Request Broker). Die einzelnen ORBs werden von unterschiedlichen Herstellern entwickelt und sind daher nicht hundertprozentig kompatibel. Die OMG verfolgt die folgenden zwei Ziele (vgl. [OMG97b] und [VD97]):

- Förderung der objektorientierten Technologie in der Software Entwicklung.
- Entwicklung eines gemeinsamen architektonischen Frameworks für verteilte, objektorientierte Applikationen basierend auf einer Interface Spezifikation.

Die *Common Object Request Broker Architecture* (CORBA) definierte einen ORB-Standard gemäss dem OMA-Referenzmodell. Die OMG veröffentlichte 1991 die erste CORBA-Spezifikation CORBA 1.1. CORBA 2.0 wurde im Jahre 1995 veröffentlicht und versprach Interoperabilität zwischen den verschiedenen ORBs (Hersteller übergreifend).

Die Abbildung 4.4 zeigt die Beziehung zwischen Betriebssystem, Programmiersprache und CORBA. Durch die Middleware CORBA werden einer Programmiersprache Teile der Kommunikation mit dem Betriebssystem vereinfacht oder ganz abgenommen.

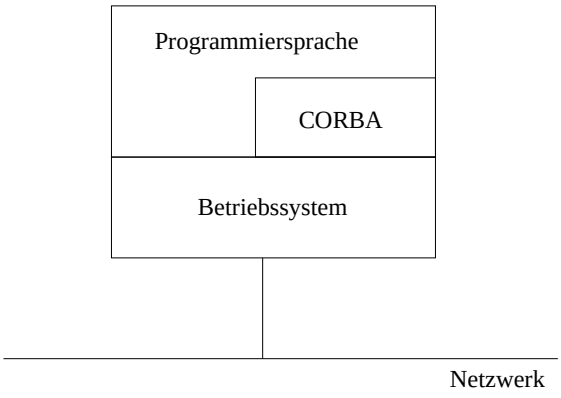

Abbildung 4.4: CORBA liegt zwischen Applikation und Betriebssystem

4.2.1 OMA Guide (*Object Management Architecture Guide*)

In der OMA wird ein Middleware-Referenzmodell beschrieben. Dieses Referenzmodell für verteilte objektorientierte Applikationen wurde von der OMG erstellt und enthält eine umfassende Beschreibung, wie dieses Softwaresystem aussehen soll.

Die Abbildung 4.5 stellt die Architektur der OMA dar. Die OMA (vgl.[OH97] und [OMG97b]) besteht aus den folgenden Elementen:

- Im Zentrum steht der *Object Request Broker* (ORB). Er ermöglicht eine transparente Kommunikation zwischen den Objekten.
- *Object Services* definieren eine Sammlung von Diensten, welche in Form von Schnittstellen beschrieben sind. Diese Dienste können unabhängig vom Anwendungsgebiet von allen Objekten benutzt werden.
- Die *Common Facilities* definieren spezialisierte und anwendungsspezifische Funktionalitäten, wie zum Beispiel das *Distributed Document* oder die *Time Facilities.*
- Die *Domain Interfaces* sind eine Sammlung von Diensten. Sie wurden für bestimmte Anwendungsgebiete definiert, wie zum Beispiel für Produktionsprozesse oder das Gesundheitswesen.

- *Application Interface* sind die Produkte der Anwender. Die Application Objects sind die für eine verteilte Anwendung entstandenen Objekte.

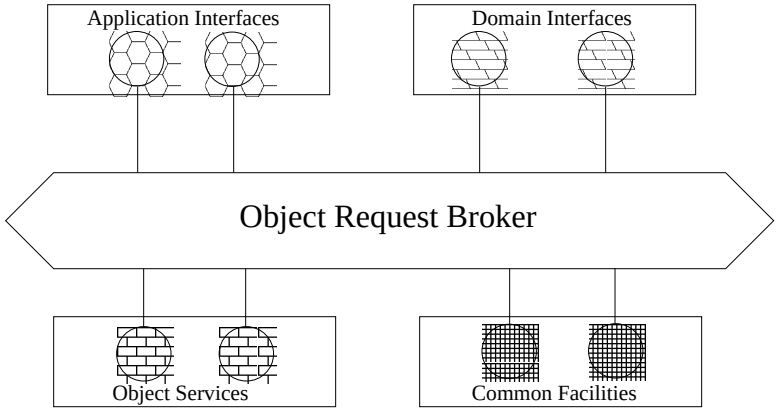

Abbildung 4.5: OMA-Referenzmodell

Im folgenden werden die einzelnen Elemente des OMA-Referenzmodells noch genau erläutert.

ORB (Object Request Broker)

Ein ORB ist ein logischer Objektbus, der die einzelnen Komponenten über mehrere Rechner hinweg verbindet. Der ORB ermöglicht zwischen CORBA-Objekten (Komponenten mit einem Anschluss an einen ORB) eine transparente Kommunikation. Das Client-Objekt braucht weder zu wissen, wo sich das Server-Objekt befindet, noch muss ein Port angegeben werden, um mit dem Server-Objekt zu kommunizieren. Für den Client spielt es keine Rolle, ob sich der CORBA-Server lokal oder auf einem entfernten Rechner befindet, der ORB übernimmt die Kommunikationsaufgaben.

Die Trennung von Interface und Implementierung und die sprachneutralen Datentypen ermöglichen es CORBA, sprach-, betriebssystem- und hardwareunabhängig zu sein. Die Abbildung 4.6 zeigt wie zum Beispiel ein Client, in Java programmiert, Methoden eines C++-Servers verwendet. Der ORB verbindet die zwei Komponenten und übernimmt die Kommunikation.

Ein CORBA ORB ermöglicht sowohl einen statischen als auch einen dynamischen Methodenaufruf. Statische Methodenaufrufe werden vor der Kompilation definiert und können nach der Kompilation nicht mehr verändert werden. Durch die Möglichkeit eine Methode dynamisch zu rufen, erhält man maximale Flexibilität, in dem man die Methode erst zur Laufzeit bindet (late binding) oder vor dem Gebrauch eines Objektes die Funktionalität erforscht (run-time discovering).

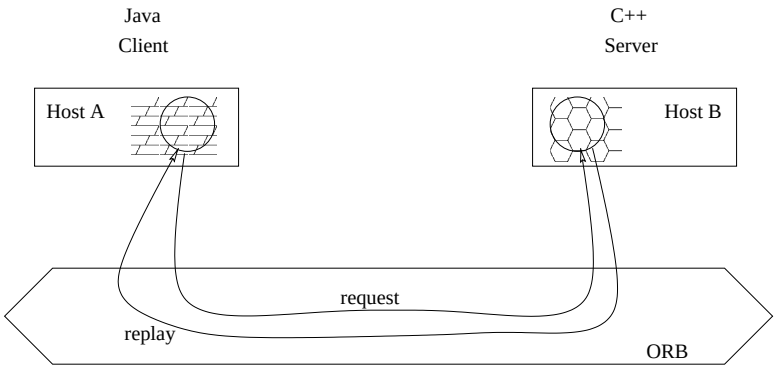

Abbildung 4.6: Methodenaufruf über den ORB

Common Object Services

Der CORBA Standard stellt einen Rahmen für die Realisierung von verteilten Applikationen zur Verfügung. Die OMG definierte die *Common Object Services* (auch Dienste oder CORBA Services genannt) um die Software-Entwicklung zu erleichtern. Diese Dienste bieten den Software-Entwicklern ergänzende Möglichkeiten wie zum Beispiel das Verwalten von Objekten oder Daten. Alles in allem wird es fünfzehn verschiedene CORBA Services geben (vgl. Tabelle 4.1), aber leider sind noch nicht alle Dienste spezifiziert und daher nur wenig implementiert. Nachfolgend werden einige Dienste kurz beschrieben (für eine ausführliche Einführung vgl. zum Beispiel [EFA96]):

- Der *Naming Service* ermöglicht das Referenzieren von Objekten mit einem eindeutigen Namen innerhalb eines verteilten Namensraumes. Die Reichweite dieses Telefonbuchähnlichen Dienstes ist beschränkt auf den jeweiligen ORB. Der Namensdienst ist einer der wenigen CORBA Dienste, die schon fast in jedem ORB vorhanden sind.
- Der *Event Service* ermöglicht die Entkoppelung der Kommunikation zwischen Client und Server. Über einen Ereignis-Kanal werden Ereignisse gesammelt und anschliessend wieder verteilt, so dass die beteiligten Komponenten nichts voneinander wissen müssen.
- Der *Lifecycle Service* besteht aus Standardschnittstellen, die definieren, wie Objekte erzeugt, gelöscht, kopiert oder auch bewegt werden.
- Der *Transaction Service* ermöglicht die Entwicklung von zuverlässigen, verteilten Applikationen. In diesem Dienst werden sogenannte ACID[2]-Transaktionen definiert.

Common Facilities

Die *Common Facilities*, früher auch *Horizontal Framework* genannt, definieren eine Sammlung von Schnittstellenbeschreibungen eines Frameworks. Diese Sammlung von endanwendungsorientierten Diensten kann in einem breiten Bereich von Anwendungen eingesetzt werden. Die meisten

[2] ACID steht für Atomicity, Consistency, Isolation und Durability.

Dienst	Erklärung
Naming Service	Identifikation von Objekten durch einen Namen
Event Service	Sammeln und Verteilen von verteilten Ereignissen
Life Cycle Service	Standard Operationen für Komponenten
Persistence Service	Speicherung von Komponenten
Relationship Service	Gruppierung von Komponenten
Externalization Service	Mechanismen von Komponenten für Dateninput/-output
Transaction Service	Koordination von Komponenten
Concurrency Control Service	Koordination von nebenläufigen Prozessen
Licensing Service	Gebrauchs- und Berechnungskontrolle (Messinstrument)
Query Service	SQL-ähnliche Komponentenabfrage
Properties Service	Dynamische Zuordnung von Eigenschaften
Security Service	Zugriffskontrolle
Time Service	Synchronisation der Uhrzeit
Collections Service	Erzeugung und Bearbeitung einer Sammlung von Komponenten
Trader Service	Publikation von Diensten

Tabelle 4.1: Auflistung aller CORBA Dienste

der einzelnen Dienste sind konfigurierbar. *Common Facilities* bieten zum Beispiel ein Dokumentenmanagement für verteilte Dokumente oder eine Prozessablaufsteuerung.

Domain Interfaces

Die *Domain Interfaces* bieten vor allem branchenspezifische Dienste an, wie zum Beispiel in den Bereichen des Internets, der Telekommunikation oder des Finanzwesens.

Früher nannte man diese Dienste auch *Vertical Framework*, da sie hauptsächlich für bestimmte Anwendungsgebiete definiert wurden. 1996 wurden diese eher branchenspezifischen Dienste von den *Common Facilities* getrennt.

4.2.2 Aufbau eines CORBA ORBs

Der ORB wurde bisher immer nur als eine Einheit dargestellt, wie in Abbildung 4.5 auf Seite 27 ersichtlich. Der CORBA ORB setzt sich bei genauer Betrachtung aber aus mehreren Bestandteilen zusammen gemäss Abbildung 4.7.

Der ORB erbringt die Kommunikation zwischen Client und Server. Diese Kommunikation erfolgt mit Hilfe der verschiedenen Schnittstellen, die die einzelnen Bestandteile anbieten und einem ORB-Agent (daemon process). Die Schnittstellen *Dynamic Invocation Interface, Dynamic Skeleton Invocation, Basic Object Adapter* und der ORB Kern liegen als vorgefertigte Komponenten in einer Bibliothek vor. Der Stub und das Skeleton werden aus dem IDL-Interface (siehe unten)

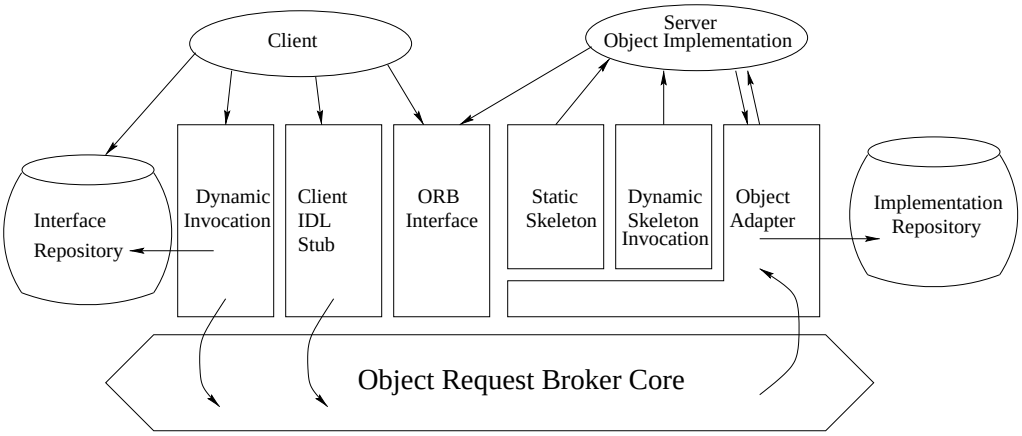

Abbildung 4.7: Die Struktur eines ORBs

durch einen Compiler automatisch erzeugt.
Nachfolgend werden die Aufgaben der einzelnen Bestandteile eines ORBs erläutert:

- Der ORB Kern (Object Request Broker Core) bietet vielseitige Dienste für den Client und den Server. Er hilft zum Beispiel den *Basic Object Adapter* (BOA) zu initialisieren oder eine Objektreferenz in einen String umzuwandeln (`object_to_string()`-Methode) (vgl. Absatz 4.3.4).

- Das *Implementation Repository* speichert die Programme, zum Beispiel die Java-Byte-Code Dateien. Assoziationen zwischen Objekten und Server werden mit “idle” oder “running” vermerkt. Das *Implementation Repository* ist noch nicht spezifiziert, das heisst es ist von ORB zu ORB verschieden. Der *Basic Object Adapter* wurde in CORBA 2.0 definiert. Seine Hauptaufgabe ist die Aktivierung und Deaktivierung von Serverobjekten.

- Der *Stub* wird automatisch aus dem IDL-Interface generiert. Der Client kann mit Hilfe eines Stubs einen statischen Methodenaufruf erzeugen. Der Stub übernimmt das *Marshaling* (vgl. Absatz 4.3.2) bei einem Methodenaufruf.

- Das *Dynamic Invocation Interface* (DII) ermöglicht einen dynamischen Methodenaufruf, das heisst es wird kein Stub benötigt. Das DII definiert eine Pseudo-IDL-Schnittstelle. Diese Schnittstelle erlaubt ein Serverobjekt zu referenzieren, sowie Methoden und deren Argumente zu setzen. Diese Informationen werden durch den Aufruf einer DII-Methode (invoke) zum Serverobjekt übertragen.

- Das *Skeleton* wird wie der *Stub* automatisch aus dem IDL-Interface erzeugt. Es liegt zwischen BOA und CORBA-Server. Das Skeleton transformiert die empfangenen Datenpakete in den entsprechenden Methodenaufruf (Unmarshaling).

- Die *Dynamic Skeleton Invocation* nimmt Anfragen des DII entgegen und erzeugt einen dynamischen Operationsaufruf beim Serverobjekt.

- Im *Interface Repository* werden die Interfaces der Serverobjekte abgespeichert. Ohne diese Informationen könnten die Methoden nicht dynamisch aufgerufen werden.

4.3 Eigenschaften

4.3.1 IDL (Interface Definition Language)

CORBA ist weder an eine bestimmte Programmiersprache noch an eine Hardware noch an ein bestimmtes Betriebssystem gebunden. Diese Unabhängigkeit wird durch die Trennung von Interface und Implementierung erreicht. Jedes CORBA-Server-Objekt hat eine klar definierte Schnittstellenbeschreibung. Diese Schnittstellen werden in einer Interface Definition Language (IDL) deklariert.

Die IDL ist eine rein deklarative Sprache, welche ausschliesslich zur Beschreibung der Objektschnittstellen verwendet wird. Die IDL wurde von der OMG spezifiziert und hat grosse Ähnlichkeit zu der Syntax von C++.

Die Tabelle 4.2 gibt einen Überblick über die OMG-IDL Spezifikation. Die IDL-Spezifikation kann in die folgenden vier Aspekte aufgeteilt werden:

- Datentypen werden zusammengesetzt aus den Basistypen, den zusammengesetzten Typen und den Templatetypen.
- Attribute und Methoden definieren die Aktionen, die ein Server-Objekt zur Verfügung stellt. Ein Attribut ist eine einfache Möglichkeit, zwei Methoden (get und set) zu erzeugen, die auf eine einzelne Variable zugreifen.
- Ein Interface definiert die Schnittstelle zu einem Server-Objekt. Eine Schnittstelle kann von einer anderen Schnittstelle erweitert werden.
- Ein Modul fasst Schnittstellen und Deklarationen zusammen und wird in einen eigenen Namensraum (scope) abgelegt.

Die Abbildung 4.8 zeigt ein Modul `Math` mit dem IDL-Interface `ComplexIF`. Das Modul beschreibt das Interface in einem eigenen Namensraum, damit es keine Namenskonflikte gibt.

4.3.2 Sprachanbindung (Language mapping)

CORBA-Server-Objekte definieren ihre Schnittstellen durch die Interface Definition Language. Jede Programmiersprache benötigt für seine Sprachkonstrukte eine Abbildung (language mapping) in die entsprechenden Typen der IDL. Die Tabelle 4.3 auf Seite 34 zeigt als Beispiel die Abbildung von IDL nach Java. Man verwendet diese Abbildung, um aus einem Interface den entsprechenden Programmcode zu erzeugen. Diese Programmcode werden durch einen Compiler, zum Beispiel einen IDL nach Java Compiler (idl2java), automatisch erzeugt. Der Programmcode übernimmt nun die ganze Kommunikation zwischen Client und Server, wie zum Beispiel Kommunikationsaufbau, Umwandlung der Datenstrukturen in einen Byte-Stream (Marshaling) oder

	OMG-IDL	Erläuterung
Basistypen	short	16-bit 2er Komplement (ohne Vorzeichen) Integer
	unsigned short	16-bit 2er Komplement (mit Vorzeichen) Integer
	long	32-bit 2er Komplement (ohne Vorzeichen) Integer
	unsigned long	32-bit 2er Komplement (mit Vorzeichen) Integer
	float	16-bit IEEE floating point number
	double	32-bit IEEE floating point number
	octet	8-bit uninterpreted type
	char	ISO Latin-1 Character
	boolean	Boolean mit TRUE und FALSE
	any	irgendein IDL-Typ
Zusammengesetzte Typen (Constructet types)	enums	Aufzähltyp mit Integerwerten
	struct	für die Deklaration neuer Datentypen
	discriminated union	Deklaration eines Auswahltyps
Templatetypen	sequence	geordnete Kollektion; unbestimmte Anzahl Elemente
	string	String mit variabler Länge (fixed at run-time)
	array	Ein Array mit fester Länge und bestimmten Typ
Interface	interface	Definiert Schnittstelle eines Objekts
	Inheritance specification	Erben eines Interfaces
	type declaration	Deklaration neuer Datentypen
	constant declaration	Konstantendeklaration
	exception declaration	Deklaration von Ausnahmen
	attribute declaration	Attributdeklaration
	operation declaration	Methodendeklaration
Module	module	Definition eines eigenen Namensraumes (scope)
	type declaration	Deklaration neuer Datentypen
	constant declaration	Konstantendeklaration
	exception declaration	Deklaration von Ausnahmen
	interface declarationen	Interfaces
	module declarationen	Schachtelung von Modulen

Tabelle 4.2: Die IDL-Spezifikation

```
// IDL- Interface of complex numbers

// Declaration of a module:
module Math{

  // Declaration of an exception
  exception ZeroDivisionError{
    string name;
  };

  // Declaration of a datastructure
  struct Complex{
    float re;
    float im;
  };

  // Declaration of the interface:
  interface ComplexIF{
    Complex add(Complex z1, Complex z2);
    ...
    Complex div(Complex z1, Complex z2)
      raises ZeroDivisionError;
  };
};
```

Abbildung 4.8: IDL-Beispiel: Komplexe Zahlen

Methodenaufruf.

4.3.3 Interoperabilität

Der ORB selbst ist das Kommunikationsmedium zwischen Anwendungen. Er ist auch für die Interoperabilität von Programmiersprachen und Plattform unabhängiger Anwendungen zuständig. Benutzt man aber ORBs von verschiedenen Herstellern, so werden weitere Konzepte benötigt. Die Interoperabilität zwischen verschiedenen ORBs basiert auf den zwei folgenden Elementen (vgl. Abbildung 4.9 und [SAT98]):

- Das Internet-Inter-ORB-Protocol (IIOP) definiert ein Kommunikationsprotokoll gemäss dem General-Inter-ORB-Protocol (GIOP). Die OMG definierte erst in CORBA 2.0 das abstrakte Protokoll GIOP.

- Die Inter-ORB Reference (IOR) ermöglicht eine weltweit eindeutige Referenzierung der CORBA-Objekte. Diese eindeutige Referenz enthält die Internetadresse des Objektes, eine Port-Nummer und eine ORB-abhängige Objektidentifikation.

	OMG-IDL	Java (VisiBroker)
Basistypen	long	int
	unsigned long	int
	short	short
	unsigned short	short
	float	float
	double	double
	octet	byte
	char	char
	boolean(TRUE, FALSE)	boolean(true, false)
	any	Klasse CORBA.any(org.omg.CORBA.Any)
Constructet types	enums	Klasse mit static final enum Wert
	struct	Klasse mit Instanzvariable und Konstruktor
	discriminated union	Klasse mit get und set Methoden für die Felder
Templatetypen	sequence	array
	string	java.lang.String
	array	array
Interface	interface	interface
	exception	Klasse mit Instanzvariable, erweitert CORBA.UserException
Module	module	packages

Tabelle 4.3: IDL nach Java Mapping

4.3.4 Dynamischer Methodenaufruf

Ein CORBA-Client kann mit Hilfe des *Dynamic Invocation Interface* (DII) dynamisch Methoden aufrufen, das heisst, der Client kann auch ohne das compilierte Interface, den Stub, auskommen. Der Client erzeugt einen *Request,* der dann dynamisch ausgewertet wird. Dieser *Request* enthält die Informationen über die Objektreferenz, den Methodennamen sowie Datentypen und Werte der Argumente. Der Client kann diese Informationen durch das *Interface Repository* erlangen.

Im nachfolgenden Programmcode wird exemplarisch ein dynamischer Methodenaufruf eines CORBA-Client in Java mit dem VisiBroker aufgezeigt:

```
...
// get object reference with a stringified object reference (ior)
CORBA.Object obj = orb.string_to_object(ior);

// get the interface definition from Interface Repository
CORBA.InterfaceDef int_def = obj._get_interface();
```

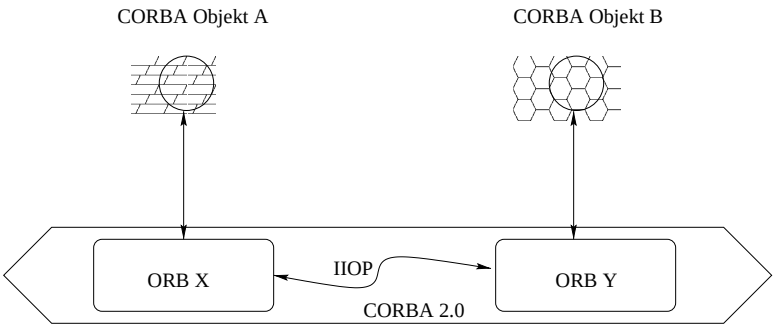

Abbildung 4.9: Interoperabilität

```
// get full interface description
CORBA.InterfaceDef_.FullInterfaceDescription full_int_def =
      int_def.describe_interface();

// using the DII to make an invocation
// create a support object
DiiAnySupport dii_any_support = new DiiAnySupport();

// create and initialize result
NVList result_list = orb.create_list(0);
result_list.add_value("", dii_any_support.TC2Any(
      full_if_desc.operations[0].result, ParameterMode._PARAM_OUT ),0 );

// create and initialize an argumentlist for the request
NVList arg_list = orb.create_list(0);
arg_list.add_value(full_if_desc.operations[0].parameters[i].name,
      dii_any_support.TC2Any(full_if_desc.operations[0].parameters[i].type
      ...);

// create request
CORBA.Request request = obj._create_request(...);

// invoke request
request.invoke();

// get result
System.out.println("result:\n    " + request.result().value() );
```

An diesem Programmcode fällt auf, dass ein Client bei einem dynamischen Methodenaufruf - genau gleich wie beim statischen Aufruf (vgl. nächster Abschnitt) - erst den ORB initialisiert und danach mit Hilfe der `string_to_object`-Methode den Server referenziert. Danach sind die zwei Verfahren verschieden. Der Client braucht nun die Unterstützung des *Interface Repository* um einen *Request* zu machen. Zwei Listen werden für den Methodenaufruf benötigt: Eine Resultat- und eine Argumentliste.

Um einen dynamischen Methodenaufruf zu machen, wird das *Interface Repository* verwendet. Das *Interface Repository* ist seit CORBA 2.0 standardisiert. Die ORB-Hersteller sollten eigentlich

das *Interface Repository* alle einheitlich implementiert haben. Somit sollte ein dynamischer ORB-übergreifender Zugriff auf die CORBA-Komponenten möglich sein.

4.4 “Pricing”-Applikation - ein Beispiel

Anhand eines Beispiels soll die Anwendung von CORBA dargestellt werden. Es wurde bewusst ein einfaches Beispiel gewählt, um die eigentlichen Mechanismen der Erstellung einer CORBA-Applikation mit statischen Methodenaufrufen hervorzuheben.

Der *PricingServer* (vgl. Anhang C) ist eine in Java geschriebene Client/Server-Applikation. Clients können beim Server Preisinformationen über bestimmte Artikel erhalten. Ist der Artikel nicht in der Serverdatenbank enthalten, erscheint die Meldung “UnknownArticle”.
Diese CORBA-Applikation besteht aus den folgenden Dateien:

- Pricing.idl : Definition des Interfaces des Pricing Objektes
- Pricing.java : Implementation der Pricing-Klasse
- PricingServer.java : CORBA Server
- PricingClient.java : Implementation des CORBA-Client

Je nach Hersteller gibt es verschiedene Vorgehensmöglichkeiten, eine CORBA-Applikation zu erstellen. Der VisiBroker (vgl. [Vis97]) zum Beispiel bietet neben dem “idl2java”-Compiler noch zwei weitere Hilfsmittel. Erstens übersetzt ein sogenannter “Caffeine”-Compiler (java2iiop) Java nach IDL, das heisst, das Interface kann in Java geschrieben werden und der Compiler übersetzt anschliessend dieses Java-Interface in ein IDL-Interface. Zweitens erzeugt ein weiteres Hilfsmittel (java2idl) aus einer bestehenden Applikation den IDL-Code. Diese Zusätze sind aber vom Hersteller angebotene Erweiterungen und kommen nicht bei jedem ORB vor. Normalerweise erstellt man eine CORBA-Applikation wie folgt:

1. Erstellen eines IDL-Interfaces
2. Compilation des IDL-Codes zu Stub und Skeleton
3. Implementierung des Servers
4. Implementierung des Clients
5. Compilation des Client beziehungsweise des Servers
6. Server starten
7. anschliessend den Client starten

Der IDL-Interface-Code des “PricingServer”-Beispiels sieht wie folgt aus:

```
// Pricing.idl; IDL-Interface for a pricing object
// namespace "Store"
module Store{
  // exception for non existing articles
  exception UnknownArticle{};
  interface Pricing{
    float cost(in string article)
      raises (UnknownArticle);
  };
}; //End module
```

Dieses Interface definiert das Modul `Store`, das für dieses Interface einen eigenen Namensraum kreiert. Das Modul besteht aus einer Ausnahmen-Deklaration (exception) und dem Interface `Pricing`. Das Interface enthält eine Methode, welche eine Ausnahme generieren kann.

Nach der Interfaceerstellung folgt deren Compilation. "idl2java" heisst der Compiler vom VisiBroker (VisiBroker for Java). Die Compilation erzeugt den Stub und das Skeleton, sowie einige weitere Klassen, Module und Interfaces. In unserem Beispiel geht das folgendermassen:

```
%idl2java Pricing.idl
Creating: Store
Creating: Store/UnknownArticle.java
Creating: Store/UnknownArticleHolder.java
Creating: Store/UnknownArticleHelper.java
Creating: Store/Pricing.java
Creating: Store/PricingHolder.java
Creating: Store/PricingHelper.java
Creating: Store/_st_Pricing.java
Creating: Store/_sk_Pricing.java
Creating: Store/_PricingImplBase.java
Creating: Store/PricingOperations.java
Creating: Store/_tie_Pricing.java
Creating: Store/_example_Pricing.java
```

Der Stub (in unserem Beispiel `Store/_st_Pricing.java`) ist für den Client zuständig. Er zerlegt Methodenaufrufe des Clients in geeignete Datenpakete (Marshaling) und leitet sie weiter an den ORB. Die Datenpakete gelangen durch den ORB zum Server. Auf der Seite des Servers ist ein Skeleton (in unserem Beispiel `Store/_PricingImplBase.java`). Das Skeleton übernimmt die Datenpakete und setzt sie wieder zusammen. Der Server führt nun die aufgerufene Methode mit den entsprechenden Argumente aus. `Store/Pricing.java` und `Store/UnknownArticle.java` sind zwei generierte Interfaces, die vom Stub beziehungsweise dem Skeleton benötigt werden.

Die Erstellung eines CORBA-Servers erfolgt immer in zwei Schritten. Erstens implementiert man die im Interface definierte Klasse (hier Pricing.java). Zweitens erstellt man einen Server, der das Pricing-Objekt der CORBA-Welt zur Verfügung stellt.
Der Code der Pricing-Klasse sieht wie folgt aus:

```
// Pricing.java; Java-Implementation of the Pricing class
```

```
class Pricing extends Store._PricingImplBase {
  /** Construct a persistently named object. */
  public Pricing(String name){
    super(name);
  }
  /** Construct a transient object. */
  public Pricing(){
    super();
  }
  public float cost(java.lang.String article) throws Store.UnknownArticle {
    // the implementation in this methode is just a dummy
    // normally here would be a real database access
    if (article.equals("apple")){
      return (float)1.2;
    }
    else if (article.equals("banana")){
      return (float)0.33;
    }
    else {
      // if it is not an apple or a banana raise an exception
      throw new Store.UnknownArticle();
    }
  }
}
```

Die eigentliche Klasse implementiert das definierte Interface und entspricht im wesentlichen einer ganz normalen Java-Klasse. Es gilt zu beachten, dass die Klasse `Pricing` von der Klasse `Store._PricingImplBase` erbt. Die Oberklasse `Store._PricingImplBase` ist das vom IDL-Compiler erzeugte Skeleton. Die `Pricing`-Klasse hat zwei Konstruktoren, die dann die Konstruktoren der Oberklasse aufrufen. Der erste Konstruktor hat einen Parameter, das erlaubt diesem Objekt einen Namen zugeben (vgl. *Naming Service* zum Beispiel in [Sie96]). Es folgt der Code des `PricingServer`s:

```
// PricingServer.java

public class PricingServer {
  public static void main(String[] args){
    try{
      // Initialize the ORB (Object Request Broker).
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init();

      // Initialize the BOA (Basic Object Adapter)
      org.omg.CORBA.BOA boa = orb.BOA_init();

      // Create an Object "FruitStore"
      Pricing fruitstore = new Pricing("FruitStore");

      // Export the newly createD object.
      boa.obj_is_ready(fruitstore);
      System.out.println(fruitstore + " is ready.");
```

```
      // Wait for incoming requests
      boa.impl_is_ready();
    }
    catch (org.omg.CORBA.SystemException e) {
      System.err.println("ERROR: " + e);
    }
  } // End main
}
```

Die Server-Klasse enthält eine `main`-Methode, damit der Server selbstständig gestartet werden kann. Diese Methode ist `static`, damit ein ORB referenziert werden kann und sie erlaubt das Übergeben von Argumenten. Das Innere der `main`-Methode wird von einem `try-catch`-Block umhüllt, damit die CORBA-Ausnahmen gefangen werden können. Zuerst wird ein ORB mit `ORB.init()` referenziert. Mit dieser Referenz kann nun auch der BOA (Basic Object Adapter) initialisiert werden. Ein Objekt der Klasse `Pricing` wird erzeugt, das anschliessend vom BOA aktiviert und für die CORBA-Welt registriert wird.
Schliesslich noch den Code des Clients:

```
// PricingClient.java
public class PricingClient{
  public static void main(String[] args) {
    try{
      // Initialize the ORB.
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init();
      // Locate the "FruitStore" object
      Store.Pricing price_maker = Store.PricingHelper.bind(orb, "FruitStore");
      try{
        // invoke the method "cost" of the object "FruitStore"
        System.out.println("An apple costs SFr." + price_maker.cost("apple"));
      }
      catch(Store.UnknownArticle e){
        System.out.println("\nThis fruit is not in the range!") ;
      };
      try{
        System.out.print("A pineapple costs SFr."+price_maker.cost("pineapple"));
      }
      catch(Store.UnknownArticle e){
        System.out.println("\nSorry, but this fruit is not in the range!") ;
      };
    }
    catch (org.omg.CORBA.SystemException e) {
      System.out.println("ERROR : " + e) ;
    }
  } // end main
}
```

Die `PricingClient`-Klasse hat eine ähnliche Struktur wie die `PricingServer`-Klasse. Der ORB wird referenziert, damit danach das Server-Objekt lokalisiert werden kann. Diese Lokalisation benutzt den *Naming Service,* das heisst dieses Objekt konnte durch den Namen '`FruitStore` gefunden werden. Der Aufruf der Methode `cost` muss mit einer Ausnahmebehandlung

umgeben werden, da diese Methode eine Ausnahme erzeugen kann.

Der Client und der Server werden nun compiliert. Der Java-Compiler erzeugt einen Byte-Code, der auf einer “Java Virtual Machine” ausgeführt werden kann. Die ausführbare Applikation kann nun gestartet werden.

Zuerst wird der Server gestartet und anschliessend der Client. Die Ausgabe des Servers sieht folgendermassen aus:

```
%java PricingServer
Pricing[Server,oid=PersistentId[repId=IDL:Store/Pricing:1.0,
objectName=FruitStore]] is ready.
UnknownArticle raised!
```

und die des Clients:

```
%java PricingClient
An apple costs SFr.1.2
A pineapple costs SFr.
Sorry, but this fruit is not in the range!
```

Es wurde absichtlich ein einfaches Beispiel gewählt. Das Beispiel verwendet nur einen ORB mit einer Programmiersprache, damit das Vorgehen anschaulich gezeigt werden kann.

Interoperabilität

Um das FruitStore-Objekt über verschiedene ORBs hinweg zu verwenden, sind die folgenden zusätzlichen Schritte nötig:

- In der `PricingServer`-Klasse muss man eine Referenz des `FruitStore`-Objektes erzeugen. Diese Referenz wird zum Übertragen in einen herstellerunabhängigen String umgewandelt. Der String wird IOR (Inter-ORB Reference) genannt.
- Die IOR muss irgendwie zum Client gelangen. Dies kann zum Beispiel in Form einer URL-Adresse oder via eine Datei geschehen.
- Der Client liest die IOR und wandelt diesen wieder in eine Objektreferenz um.

Kapitel 5

Problemanalyse

Die nachfolgende Analyse erörtert die Problematik der Middleware CORBA. Auf einen Vergleich mit DCOM oder RMI wird verzichtet, da dies den Rahmen dieser Arbeit sprengen würde.

Diese Untersuchung wird allgemein gehalten und geht nicht auf die unterschiedlichen CORBA ORBs ein, da die Hersteller der einzelnen ORBs sich an den vorgegebenen Standard halten. Zudem bieten einzelne Hersteller diverse Erweiterungen, die nicht dem Standard entsprechen. Daher ist es sinnvoll, sich hauptsächlich auf die Spezifikation von CORBA 2.0 zu beziehen.

CORBA ist eine Middleware, die Programmiersprachenunabhängigkeit verspricht. Diese Technologie eignet sich für die Programmierung von verteilten, heterogenen Applikationen. Zudem verspricht man sich mit CORBA eine transparente Verteilung der Serverobjekte. Eine objektorientierte Architektur bildet die Basis von CORBA. In diesem Kapitel werden die versprochenen Eigenschften von CORBA überprüft und die Schwachstellen aufgezeigt.

5.1 Probleme von CORBA

Ein erstes “Hello, World!” geht in den meisten Fällen ohne grosse Probleme. Leider ist das nicht so mit CORBA. Beim Erlernen von CORBA fällt sofort auf, dass gewisse Kenntnisse erforderlich sind, um das erste “Hello, World!” schreiben zu können. Der Weg zu diesem Einstiegsprogramm ist mit diversen Schwierigkeiten verbunden, welche teilweise gar nichts mit der Programmierung des Clients beziehungsweise des Servers zu tun haben. Meist ist nicht genau ersichtlich, wo sich die Fehlerquelle befindet, wie zum Beispiel die falsche Version des Betriebssystems oder ein Fehler in einem CORBA-Service.

Diverse Kritiker haben sich schon mit den Problemen von CORBA befasst. Sie deckten beim praktischen Gebrauch verschiedene Mängel dieser Middleware auf. Im allgemeinen wird dieser Standard als sehr gut bewertet. Aber je nach Benutzungsfeld können zusätzliche Bedürfnisse an CORBA entstehen. Nachfolgend werden Meinungen ausgewählter Kritiker aufgelistet.

In “Client/Server Programming with Java und CORBA” listen Robert Orfali und Dan Harkey (vgl. [OH97]) folgende Schwachpunkte von CORBA auf:

- Die Spezifikationen von CORBA sind nur schlecht oder mangelhaft dokumentiert.
- Für viele Programmiersprachen fehlen die IDL-Mapping-Standards. Für Python zum Beispiel ist das Mapping (IDL-to-Python) noch nicht definitiv festgesetzt.
- Wie werden Server skaliert? Es gibt keinen Mechanismus, der eine grosse Anzahl von Servern verwalten kann. Ein weiteres ungelöstes Problem stellt die Lastverteilung dar. Hier handelt es sich um die Frage, wie ein stark belasteter Server die Arbeit auf mehrere Prozesse verteilen kann.
- Bei der Lösung von *Mission-critical requirements* wird der Anwender nicht von CORBA unterstützt.
- Die Kommunikation zwischen Client und Server kommt nur dann zustande, wenn beide gleichzeitig aktiv sind (synchrone Kommunikation). CORBA hat keine *asynchrone message queue*, welche vor allem für heterogene Systeme von grossem Vorteil wäre. Mit dem asynchronen Modus könnten sowohl Client als auch Server voneinander unabhängig arbeiten.

Walter Bischofberger und Dirk Riehle (vgl. [BR97]) unterteilen die Problematik von CORBA in die drei Gesichtspunkte: Objektmodell, Software Architektur und pragmatische Überlegungen. Sie untersuchen, ob CORBA zur Entwicklung von Anwendungssystemen taugt.

Objektmodell:

- In der Spezifikation werden nur die einfachsten Aspekte von Objekten definiert. Es gibt nur wenig gemeinsame Standardfunktionalitäten für alle Objekte. Eine Grundfunktionalität von der jedes Objekt erben muss, existiert nicht.
- Es fehlt ein allgemeines Zugriffsprotokoll für den Objektzustand, was für das Erstellen von Debuggern erforderlich wäre.
- CORBA hat keine Schnittstelle mit generischer Funktionalität. Der Anwender muss dadurch existierende Implementationen für jede Einführung neuer Funktionalität unnötigerweise neu anpassen und erweitern.
- CORBA basiert fix auf einem Objektmodell, dadurch wird eine Evolution von Objektsystemen verunmöglicht.

Softwarearchitektur:

- CORBA definiert eine Menge von Diensten und Facilities. Wie man aber mit diesen Grundelementen umgehen soll, erklärt CORBA kaum. So ist es schwierig, mit Hilfe der Dienste und Facilities ein Framework für Client/Server Architekturen zu bauen.
- Wie schon erwähnt, ist CORBA eine Objekt-Architektur. Seit einiger Zeit hat sich gezeigt, dass eine Komponentenarchitektur viel geeigneter für das Erstellen von Softwarearchitekturen ist. Die OMG hat dieses Problem erkannt und wird darauf mit dem CORBA 3.0 Standard reagieren.

Pragmatische Überlegungen:

- Aus einem grosstechnischen, konsensorientierten Standardisierungsprozess wie bei CORBA resultieren normalerweise relativ komplexe, schwer verständliche Standardgebilde.
- Die OMG ist ein Konsortium von über 750 Mitgliedern, das hat zur Folge, dass die Standardisierung nur schleppend voran kommt. Einige der ORB-Anbieter bauen daher ihre eigenen adhoc Erweiterungen, um den Kundenwünschen gerecht zu werden. So entsteht eine Diskrepanz zwischen den einzelnen ORBs und der Spezifikation des CORBA Standards.
- CORBA ist ein offener Standard, dadurch ist es jedermann möglich, seinen eigenen ORB zu bauen. Daher gibt es viele verschiedene ORB-Anbieter und es herrscht zwischen den Anbietern ein grosser Konkurrenzkampf. Der Kunde muss bei seiner ORB-Evaluation jeweils noch die Überlebenschancen der ORB Hersteller einberechnen.

Aus dem Artikel von Bernd Eichner, David Kamber und Stephan Murer (vgl. [EKM97]) und dem Referat von Stefan Murer an der Object Expo (vgl. [Mur97]) gehen die folgenden Problempunkte hervor:

- Die CORBA Services sind noch nicht fertig spezifiziert. Die Entwickler brauchen diese Dienste aber um ihre Applikationen einfacher entwickeln zu können. Andererseits sind bereits standardisierte Dienste sind noch nicht ausgereift.
- CORBA mangelt es an Fehlertoleranz (fault-tolerance). In einigen Fällen treten unerwartete Probleme auf.
- CORBA generiert Fehlermeldungen, die es nicht erlauben, den Ort des Fehlers zu lokalisieren. Beispielsweise bricht eine Applikation ab und meldet:
  COMPLETION_MAYBE[1]

[1] Diese Fehlermeldung besagt einerseits, dass ein Fehler aufgetreten ist, aber andererseits lässt sich der Fehlerort nicht genau lokalisieren.

Douglas C. Schmidt et al. (vgl. [SGHP97]) bemängelt vor allem das Echtzeitverhalten von CORBA. Er vermisst Echtzeitgarantien, wie sie in vielen Anwendungsapplikationen, wie zum Beispiel in der Luftfahrt oder der Telekommunikation, unerlässlich sind. Diese Echtzeitprobleme entspringen aus den nachfolgenden Defiziten:

- CORBA fehlt eine Spezifikation über die Dienstqualität (Quality of Service). Es gibt auch keine Unterstützung für die Durchführung einer End zu End Dienstqualität. Für die Abarbeitungsreihenfolge der Clients existieren zum Beispiel keine Prioritäten.
- Die Echtzeitprogrammierung (real-time features) wird von CORBA nicht unterstützt.
- Ein weiteres Problem besteht darin, dass die meisten ORBs nicht effizient arbeiten, da CORBA nicht auf Geschwindigkeit optimiert wurde. Um eine bessere Performance zu erzielen, müssten die verschiedensten Strategien wie zum Beispiel die interne *Message Buffering Strategy* oder auch diverse Algorithmen verbessert werden.

Silvano Maffeis (vgl. [Maf97] und [MS97]) macht vorallem auf die Mängel aufmerksam, die durch den intensiven Gebrauch von CORBA auftreten. Das sind:

- Es gibt keine absolute Zuverlässigkeit (reliablility). In einem 7x24-System[2] ist CORBA nicht geeignet, da unerwartete Fehler auftreten können.
- Grössere Applikationen mit CORBA tendieren sehr schnell sehr komplex zu werden. Ein solches System kann fast nicht mehr gewartet werden. Silvano Maffeis spricht von einem “maintenance nightmare”.
- CORBA hat Erweiterungsprobleme. Verändert man ein IDL-Interface, so kann es sein, dass mehrere Clients eine Anpassung benötigen. CORBA ist nicht geeignet für eine langsame, stufenweise Applikationsentwicklung im Sinne eines *Spiralenmodell*s (vgl. [Jen97]).

Zusammenfassend kann gesagt werden, dass vor allem folgende Punkte die Schwachstellen von CORBA ausmachen:

- Fehlende Erweiterbarkeit
- Mangelnde Flexibilität
- Ungenügende Fehlertoleranz
- Lastverteilungsproblem
- Unvollständigkeit der Services
- Keine Dienstqualitätsspezifikation
- fehlendes Echtzeitverhalten

[2] In einem 7x24-System darf es keine Unterbrüche geben, d.h. es muss 7 mal 24 Stunden zur Verfügung stehen.

5.2 Einschränkung der Problemstellung

Im letzten Abschnitt wurde anhand von ausgewählter Literatur eine eingehende Problemanalyse von CORBA erstellt. In diesem Abschnitt wird basierend auf dieser Problemanalyse die Problemstellung für die vorliegende Arbeit reduziert, da sonst der zeitliche Rahmen dieser Arbeit gesprengt würde.

Die Fachgruppe *Software Composition Group* der Universität Bern, unter der Leitung von Prof. Dr. O. Nierstrasz, befasst sich unter anderem mit dem Zusammenstellen von Software-Komponenten (component composition). Unter dem Begriff "Software" versteht man Applikationen, Methoden oder Komponenten, welche auf irgendeiner Art von Rechner laufen.

Im Sinne der Forschungsgruppe interessiert die Verwendbarkeit sowie die Verbesserung von Entwicklungshilfsmittel. Nachfolgend werden die in dieser Arbeit zu lösenden Probleme von CORBA aufgelistet:

Mangelnde Flexibilität

Der Standard von CORBA 2.0 basiert auf einer Objekt Architektur (vgl. [OMG97b]). Softwaresysteme sind dauernden Veränderungen ausgesetzt. Die Erfahrung hat gezeigt, dass die objektorientierte Technologie keine Garantie für flexible und anpassbare Software-Entwicklung ist (vgl. [SN98a]) und dass Objekt Architekturen sich nicht einfach verändern lassen. Bei grossen Applikationen wird die Applikationslogik oft auf mehrere Komponenten verteilt, das heisst Client und Server sind meist aufeinander abgestimmt.

Es ist relativ aufwendig, eine CORBA-Komponente zu verwenden, da einerseits die ganzen CORBA-Kenntnisse benötigt werden und andererseits verschiedene Informationen von der CORBA-Komponente bekannt sein müssen, um überhaupt ein CORBA-Objekt zu erzeugen.

Aufwendiges Testen

CORBA-Applikationen haben ein sogenanntes *all or nothing*-Problem. Das heisst eine Server-Komponente kann nicht alleine existieren. Um ein lauffähiges Programm zu erhalten, muss zu jedem CORBA-Server ein entsprechender Code für den CORBA-Client geschrieben werden.

Das Testen einer einzelnen Komponente ist daher aufwendig, das heisst, um einen Server zu testen, muss ein vollständiger CORBA-Client geschrieben werden und der CORBA-Client muss in die Middleware CORBA eingebunden werden. Zudem gibt es noch keine *Debugger* und Analyse-Tools für CORBA (vgl. [BR97]).

Ungenügende Erweiterbarkeit

CORBA-Objekte sind "starre Gebilde", da sie implementorisch gesehen nur genau eine Schnittstelle (vgl. [OMG97b]) besitzen. Diese Eigenschaft birgt die Schwierigkeit Komponenten zu verändern (vgl. [Maf97]), das heisst, CORBA-Objekte müssen für jede zusätzliche Funktionalität neu

angepasst werden (vgl. [BR97]).

Das Erweitern einer bestehenden CORBA-Applikation kann ausserdem erschwert sein, wenn sich die Applikationslogik auf einer tiefen Abstraktionsebene befindet. Es ist oftmals einfacher, eine neue CORBA-Applikation zu schreiben, da die Einarbeitung in eine schon Bestehende zu viel Zeit benötigt.

Zudem ist es schwierig, bestehende Nicht-CORBA-Komponenten einzubinden, da dies von CORBA nur mangelhaft unterstützt wird.

Mangelhaftes Abstraktionsniveau

Grosse Applikationen bestehen meist aus mehreren Clients und Servern, und diese tendieren sehr schnell sehr komplex zu werden (vgl. [Maf97]). Ausserdem wird oft die Applikationslogik auf verschiedene Komponenten verteilt, so dass es leicht zu einem tiefen Abstraktionsniveau führen kann. Eine auf verschiedene Komponenten verteilte Applikationslogik erschwert das Verständnis der Implementierung und kann zu einer *Maintenance Nightmare* führen.

Wiederverwendungsproblem

Um eine CORBA Komponente wiederzuverwenden, benötigt man CORBA Kenntnisse, denn ohne CORBA-Kenntnisse kann keine Komponente eingebunden werden. Es stellt sich auch die Frage, auf welche Weise *legacy component*s in CORBA integriert werden können.

Da grössere CORBA-Applikationen meist eine hohe Komplexität erlangen und CORBA nicht sehr fehlertolerant ist (vgl. [EKM97]), werden diese Client/Server-Applikationen oft genau aufeinander abgestimmt. CORBA-Komponenten von speziell aufeinander abgestimmten Applikationen können daher kaum wiederverwendet werden. Zudem werden normalerweise nur Komponenten wiederverwendet, wenn sie mit grosser Sorgfalt getestet wurden, was ja mit CORBA schwierig ist.

Kapitel 6

Lösungsansatz der CORBA-Probleme

Wie aus dem letzten Kapitel ersichtlich wurde, existieren mit CORBA noch einige ungelöste Probleme. CORBA verspricht ein objektorientiertes Framework zu sein. Aber die Standardisierung ist noch nicht beendet und kommt nur langsam voran. Das bedeutet für die Entwickler, dass sie das objektorientierte Framework nur eingeschränkt verwenden können.

Dieses Kapitel gibt zuerst einen Einblick, wie man die Probleme von CORBA lösen könnte. Anschliessend wird der Ansatz der Komponenten-Proxy erläutert, der in dieser Arbeit eingeführt wird. Dieser Ansatz soll die Möglichkeit bieten, aus CORBA eine Komponenten Architektur zu machen. Durch den Einsatz einer Skript-Sprache kann ein flexibleres und leichter veränderbares Komponenten Framework für CORBA erstellt werden.

6.1 Überblick

Die vier Firmen IBM, Netscape, Oracle und Sunsoft forderten die OMG auf (vgl. [INOS97]), den Standard schneller voran zu treiben. Es wird zudem verlangt, dass CORBA unbedingt eine Skript-Sprache benötigt. Skript-Sprachen sollen als Entwicklungsumgebung für kleine Applikationen (rapid application development) zum Einsatz kommen. Zur Laufzeit kann man Skript-Sprachen zum Schreiben von neuen Skripts und deren anschliessenden Ausführung durch eine “eval”-Funktion gebrauchen. Diese Sprachen bieten die Möglichkeit, aus CORBA ein Komponenten Framework zu machen.

CORBA basiert auf einer Objekt Architektur. Um aus CORBA ein Komponenten Framework zu machen, benötigt CORBA unter anderem eine Anbindung an eine Skript-Sprache (vgl. Absatz 2.3.2). Die OMG hat das Fehlen einer Skript-Sprache erkannt und veröffentlichte ein *Request for Proposal* (RFP vgl. [TC98]). Einige Mitglieder-Firmen haben darauf reagiert und haben ihre Vorschläge eingereicht. Der Standard der Anbindung einer Skript-Sprache an CORBA befindet sich aber noch in Bearbeitung (vgl. Anhang B.6).

Mittlerweile gibt es schon einige experimentelle ORBs, die eine Anbindung an eine Skript-

Sprache realisiert haben. Es ist zu beachten, dass diese Anbindungen an die IDL noch nicht standardisiert sind. Nachfolgend werden einige Vertreter von ORBs aufgezählt, die eine Skript-Sprache anbinden:

- *MICO* für Tcl (vgl. [RPP97])
- *Inter-Language Unification* für Python (vgl. [JS97])
- *Fnorb* für Python (vgl. [Chi98])
- *COPE* für Perl (vgl. [Res98])

Eine Skript-Sprachen-Anbindung an einen ORB ermöglicht der entsprechenden Skript-Sprache ihre Vorteile auf CORBA zu übertragen.

Das Entwickeln von CORBA-Applikationen ist recht aufwendig und benötigt viel Erfahrung. Um eine CORBA-Komponente mit einer Skript-Sprache zu erstellen, ist das Verfahren genau gleich wie zum Beispiel das Entwickeln einer CORBA-Komponente mit Java. Das bedeutet, um eine CORBA-Komponente (Client oder Server) in einer Skript-Sprache zu implementieren, benötigt man das ganze Wissen über CORBA. Aus diesem Grund wird aber der Einsatz einer Skript-Sprache im eigentlichen Sinn sehr stark eingeschränkt. Ein schnelles Verbinden oder Testen einzelner Komponenten wird zu einer aufwendigen Angelegenheit, das heisst, Vorteile einer Script-Sprache gehen verloren.

An der Universität von Lille in Frankreich arbeitet eine Forschungsgruppe an einem CORBA-Projekt (vgl. [MGRG98]), das GOODE (Generic Object-Oriented Dynamic Environment) heisst. In diesem Projekt wird gezeigt, wie die Skript-Sprache *CorbaScript* dynamisch auf CORBA-Komponenten zugreifen kann. Man erreicht diesen dynamischen Zugriff auf CORBA-Objekte durch zusätzliche Dienste (CORBA tools). Diese Dienste sind von dieser Forschungsgruppe erstellte Erweiterungen eines ORBs. Der Interpreter von *CorbaScript* prüft die Datentypen zur Laufzeit, indem er mit dem *Interface Repository* von CORBA Kontakt aufnimmt.

In “Scripting COM components in Haskell” von Simon Peyton Jones und Erik Meijer (vgl. [JM98]) wird gezeigt, wie man eine funktionale Programmiersprache an Microsoft’s *Component Object Model* (COM) anbindet. Mit Hilfe der “higher-order” Feature der funktionalen Programmiersprache können Komponenten auf eine ganz spezielle Art zusammengesetzt werden, wie zum Beispiel mit “lazy evaluation”. Diese zusätzlichen Eigenschaften sollen die Sicherheit sowie die Aussagekraft der Applikationen fördern.

6.2 Die Methode des Komponenten-Proxy

Im Vergleich zu den im oberen Abschnitt genannten Lösungsansätzen ist der Ansatz der vorliegenden Arbeit nicht an einen einzelnen ORB gebunden. Der hier verwendete Lösungsansatz basiert

auf einer sogenannten “Komponenten-Proxy”-Methode, welche eine ORB-unabhängige Integration von CORBA-Komponenten ermöglicht. Der Zweck eines Proxy wird im Buch “Design Pattern” (vgl. [GHJV95]) wie folgt beschrieben:

> Ein Proxy kontrolliert den Zugriff auf ein Objekt mit Hilfe eines vorgelagerten Stellvertreter-Objekts.

Die Methode des Komponenten-Proxy führt ein weiteres Abstraktionsniveau ein. Diese Abstraktion wird als zusätzliche Schicht, in Form eines Moduls zwischen die Skript-Sprache und CORBA gelegt (vgl. Abbildung 6.1), was zu einer ORB-Unabhängigkeit führt. Die CORBA-Komponenten werden dadurch von der Skript-Sprache abgekapselt, können jedoch mit Hilfe dieses Moduls als sogenannte Stellvertreter-Objekte (Komponenten-Proxy) erzeugt werden. Diese Komponenten-Proxies können dann von der Skript-Sprache verwendet, als wären sie ganz “normale” Objekte. Ein Proxy-Objekt kann entweder mit der Objektreferenz oder mit seinem Namen erzeugt werden. Damit ein Stellvertreter-Objekt durch seinen Namen instanziert werden kann, muss ein globaler Namensdienst (*NameServer*) erstellt werden.

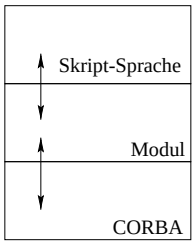

Abbildung 6.1: Eine Abstraktion durch ein Modul

Ein Objekt des Komponenten-Proxy befindet sich in der “Welt” der Skript-Sprachen (Remote-Proxy) und agiert genau gleich wie die Original-Komponente mit dem Unterschied, dass die Aktionen nicht im Proxy bearbeitet werden, sondern weitergeleitet und vom Originalserver ausgeführt werden (vgl. Abbildung 6.2). Der CORBA-Server sendet das Resultat über die eingebaute Abstraktion an den Proxy zurück. Der Anwendungsprogrammierer bemerkt von diesem Weiterleiten nichts, das heisst, die Komponente muss nicht speziell behandelt werden.

Die Komponenten-Proxy-Methode bringt folgende Vorteile:

- Es wird eine ORB-Unabhängigkeit erreicht, da diese Methode keine ORB spezifischen Veränderungen vornimmt.
- CORBA-Komponenten können ohne spezifisch Kenntnisse von CORBA verwendet werden.
- Es kann eine einzelne CORBA-Server-Komponente verwendet werden, ohne dass ein vollständiger Client dazu geschrieben werden muss, da das durch das Modul erzeugte Stellvertreter-Objekt wie jedes andere Objekt verwendet werden kann.

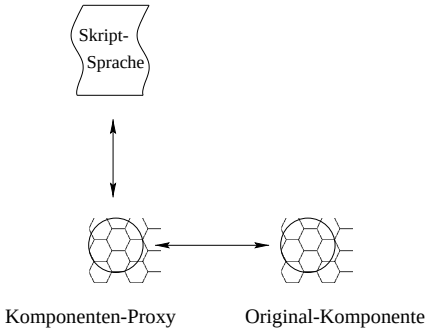

Abbildung 6.2: Ein Komponenten-Proxy leitet die Anfragen zur Original-Komponente weiter.

- Eine Skript-Sprache hat mit Hilfe des Moduls eine transparente Sicht auf die bestehenden CORBA-Komponenten.
- Das einfache Verwenden von CORBA-Komponenten fördert die Wiederverwendbarkeit von bestehenden und das Erstellen von allgemein verwendbaren Komponenten.

Aus der ORB-Unabhängigkeit ergibt sich aber ein Nachteil. Die CORBA-Komponenten müssen sich entweder beim globalen *NameServer* anmelden oder sonst irgendwie ihre Referenz publik machen.

Kapitel 7

Komponenten-Proxy

In diesem Kapitel wird gezeigt, wie man ein Komponenten Framework für CORBA erstellt. Zuerst wird mit Hilfe der in dieser Arbeit entwickelten Methode “Komponenten-Proxy” der Grundstein für das Komponenten Framework gelegt. Mit dieser Methode erreicht man eine Komponenten Architektur, das heisst, Komponenten und deren Steuerung werden getrennt. Danach wird der Aufbau eines Frameworks basierend auf dieser Komponenten Architektur erklärt. Es werden anhand von zwei Ansätzen die neuen Möglichkeiten aufgezeigt. Abschliessend werden die Grenzen dieses Ansatzes diskutiert.

In dieser Arbeit wird ein ORB mit einer Anbindung an Python verwendet. Dieser ORB heisst *Fnorb* und wurde an der Universität von Queensland in Brisbane (Australien) entwickelt (vgl. [Chi98]). Der *Fnorb* wurde gewählt, weil er zum Zeitpunkt der Evaluation der am weitesten entwickelte ORB für eine Skript-Sprache war. Zudem unterstützte er schon damals die Mechanismen für die Interoperabilität. Der *Fnorb* ist aber immer noch ein experimenteller ORB. Man hätte aber für die Komponenten-Proxy-Methode genausogut einem anderen ORB verwenden können.

7.1 Die Methode

Die “Komponenten-Proxy”-Methode ermöglicht das automatische Erstellen eines Stellvertreter-Objektes (Komponenten-Proxy) von einem CORBA-Server. Diese CORBA-Server können sich irgendwo im Rechnernetz befinden und sie sind in einer beliebigen Programmiersprache mit einer IDL-Anbindung geschrieben.

Die Abbildung 7.1 zeigt den Aufbau dieser Methode. Die Methode ermöglicht Python den Zugriff auf eine CORBA-Komponente. Dieser Zugriff erfolgt mit Hilfe eines Stellvertreter-Objekts (Komponenten-Proxy). Die Kommunikation verläuft zwischen Stellvertreter und CORBA-Komponente über CORBA ORBs.

Die Methode wird mit Hilfe der folgenden zwei Erweiterungen realisiert:

- Das Modul *corba_proxy* ermöglicht, ein Stellvertreter-Objekt zu erzeugt.

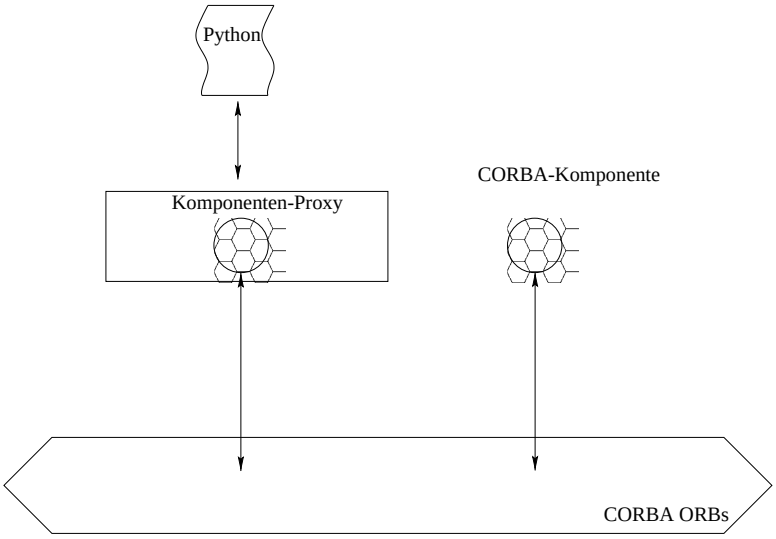

Abbildung 7.1: Methode der Komponenten-Proxy

- Der globale *NameServer* ist ein Namensdienst, bei dem sich die CORBA-Komponenten anmelden können. Dieser Dienst wird in Form eines CORBA-Servers realisiert.

Diese zwei Erweiterungen nehmen keine Veränderungen am ORB-Kern oder am sonstigen CORBA-Standard vor. Die Methode ist daher nicht an einen einzelnen ORB gebunden und kann auf die CORBA-Komponenten von verschiedenen ORBs zugreifen. Da diese Methode keine Veränderungen an CORBA vornimmt, ist es leicht vorstellbar, dass dieser Ansatz auf andere Middleware-Architekturen übertragen werden kann.

Die Abbildung 7.2 zeigt den detailierten Ablauf dieser Methode. Ein CORBA-Server bietet seine Dienste an, die von Python verwendet werden möchten. Das Kommunikationsmedium ist die Summe aller verwendeten CORBA ORBs. Es dürfen beispielsweise der *Fnorb* und der *VisiBroker* (vgl. [Vis97]) verwendet werden. Die Entscheidung, welche ORBs ausgewählt werden sollen, ist abhängig von der verwendeten Programmiersprache. Nicht jeder ORB unterstützt jede Programmiersprache.

Der globale *NameServer* und das Modul *corba_proxy* ermöglichen Python einen einfacheren Zugang zu CORBA-Komponenten. Der globale *NameServer* ist eine zusätzliche CORBA-Komponente, die Informationen über alle vorhandenen Server sammelt. Das Modul liegt zwischen Python und dem Fnorb, und generiert mit Hilfe des *NameServer* Stellvertreter-Objekte. Ein Stellvertreter-Objekt wird wie folgt erzeugt:

- Der globale *NameServer* wird als erstes gestartet.

- Die CORBA-Komponente muss sich nach seiner Erzeugung beim *NameServer* anmelden (register).

- Durch ein Python-Skript oder die Python-Shell wird das Modul *corba_proxy* importiert, um ein Stellvertreter-Objekt des CORBA-Servers zu erstellen.

- Das Modul *corba_proxy* verbindet sich mit dem globalen *NameServer* um die Referenz des CORBA-Servers zu bekommen (ask). Ist der Server registriert, so erstellt das Modul ein Komponenten-Proxy-Objekt, das mit dem CORBA-Server verbunden wird (connect).

- Von Python aus können nun die Dienste des Servers via den Komponenten-Proxy verwendet werden.

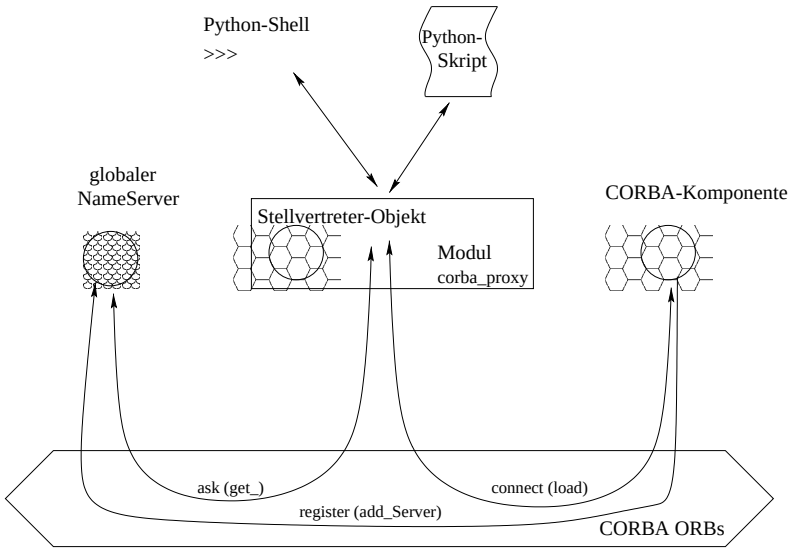

Abbildung 7.2: Mechanismus der Komponenten-Proxy-Methode

7.1.1 *NameServer*

Der globale *NameServer* ist eine CORBA-Komponente, die für die Komponenten-Proxy-Methode zusätzlich erstellt wurde. Diese Komponente ist ein unterstützender Dienst, der das Auffinden von CORBA-Servern vereinfacht.

Die Aufgaben des *NameServer*s lassen sich in zwei Bereiche unterteilen. Einerseits können sich CORBA-Server hier registrieren, damit potentielle Benutzer sie auch auffinden. Andererseits benötigt das Modul Informationen über die Server um einen Komponenten-Proxy zu erstellen.

Der *NameServer* ist ein Dienst, der mit dem CORBA *NamingService* vergleichbar ist. Im Unterschied zum CORBA *NamingService* ist der *NameServer* nicht in einem ORB integriert, sondern eine “normale” CORBA-Komponente. Im *NameServer* werden die CORBA-Server auch nicht automatisch registriert, das muss von jedem einzelnen selbst gemacht werden. Bei der Anmeldung

registriert eine Komponente zum Beispiel den Namen, den Stub-Namen und die IOR (Interoperable Object Reference).

Die Dienste des *NameServer* sind durch das nachfolgende IDL-Interface definiert:

```
// NameServer.idl; IDL-Interface of the global NameServer

module NameServer{
  // define an exception
  exception UNKNOWN_SERVER{};

  interface NameServerIF{
    short number_available_servers();
    // maximum index is one less then number_available_servers(),
    // because counting starts with 0
    string get_Server_Name(in short index) raises (UNKNOWN_SERVER);
    string get_Stub_Name(in string name);
    string get_IOR(in string name);
    string get_IDL(in string name);
    // is this server available
    boolean is_avail(in string name);

    void add_Server(in string name, in string IOR, in string Stub_Name,
                    in string IDL);
    void remove_Server(in string name) raises (UNKNOWN_SERVER);
  };
};
```

Für das Interface des globalen *NameServer* wird ein gleichnamiger Namensraum erzeugt, damit es keine Konflikte mit anderen CORBA-Servern gibt. Es wurde eine Ausnahme (`UNKNOWN_SERVER`) definiert, damit der Zugriff auf nicht registrierte Servernamen eine Ausnahme erzeugen kann. Um aber solche Ausnahmen zu vermeiden, gibt es eine Methode `is_avail`, die überprüft, ob ein entsprechender Servername existiert. Zur Vereinfachung wurde das Interface der Methoden `get_Stub_Name`, `get_IOR` und `get_IDL` ohne Ausnahmebehandlung definiert.

Ein Server muss sich mit der `add_Server`-Methode registrieren. Bei der Registration kann er seinen Namen, die IOR, den Stub und sein IDL-Interface angeben. Normalerweise wird bei der Registration der Name des Servers, die Zeichen-Folge oder der Dateiname der IOR und der Name des Stub angegeben. Die Registration sieht zum Beispiel in Java mit dem *VisiBroker* wie folgt aus:

```
// register the Famoos Model Server at the NameServer

// the IOR of the NameServer is saved in a file
DataInputStream file=new DataInputStream(new FileInputStream("nameserver.ref"));
String stringifiedIOR = new String(file.readLine());

// connect to the NameServer
org.omg.CORBA.Object obj = orb.string_to_object(stringifiedIOR);
NameServer.NameServerIF ns = NameServer.NameServerIFHelper.narrow(obj);

// register at the NameServer: name, ior, stub, idl
ns.add_Server("FM", "/home/held/public_html/IOR/FMserver.ref", "FM", "");
```

Das IDL-Interface wird in der Methode `add_Server` nur dann verwendet, wenn noch kein Stub vorhanden ist. In diesem Fall besteht die Möglichkeit, den Stub zur Laufzeit mit Hilfe des IDL-Interfaces zu erstellen (vgl. “halb-dynamischer”-Methodenaufruf in Absatz 7.4.2).

Der globale *NameServer* befindet sich immer an einem fest definierten Ort, damit jedermann zu jeder Zeit diesen Dienst benutzen kann. Diese Informationen können zum Beispiel auf dem Web publiziert werden.

Für den statischen Methodenaufruf bedeutet das, dass sowohl die in einen String umgewandelte IOR als auch der Stub des CORBA-Servers bekannt sein müssen. Ein dynamischer Methodenaufruf benötigt dagegen nur die Referenz des Servers (IOR) vom *NameServer.*

7.1.2 Das Modul *corba_proxy*

Es wurde ein Modul *corba_proxy* entwickelt, das eine weitere Abstraktion ermöglicht. Das Modul bildet eine Schicht zwischen CORBA und Python. Die Abbildung 6.1 auf Seite 49 zeigt, wie die Kommunikation mit der Komponenten-Proxy-Methode von CORBA zur Skript-Sprache und umgekehrt verläuft.

CORBA ermöglicht sowohl einen statischen als auch einen dynamischen Methodenaufruf innerhalb eines ORBs. In dieser Arbeit wird gezeigt, wie die Komponenten-Proxy-Methode den statischen Methodenaufruf unterstützt.

Statischer Methodenaufruf

Das erstellte Modul basiert auf statischen Methodenaufrufen, das heisst Methodenaufrufe benötigen Stub und Skeleton. Stub und Skeleton werden automatisch aus dem IDL-Interface des CORBA-Servers generiert. Bei einem Methodenaufruf übernehmen sie das Absenden und Entgegennehmen der Datenpakete (Marshaling und Unmarshaling).

Im nachfolgenden Programmcode wird ein grosser Teil des Moduls *corba_proxy* gezeigt. Zur Vereinfachung wurde ein Teil der Methode `load` weggelassen. Im vorliegenden Teil dieser Methode sieht man, wie ein Stellvertreter-Objekt mit Hilfe des *NameServers* erzeugt werden kann. Der weggelassene Code würde das Stellvertreter-Objekt durch seinen Stub-Name und seine IOR auf analoge Weise erzeugen. Es folgt das Listing des *corba_proxy*-Moduls (vgl. auch im Anhang C):

```
# corba_proxy.py; a Python module to generate proxy-objects
# Standard/built-in modules.
import sys, string, os, urllib
# Fnorb modules (a CORBA 2.0 ORB).
from Fnorb.orb import BOA, CORBA

class GeneralProxy:
  """This is the general proxy class """
  def __init__(self, *args):
```

```
    print "GeneralProxy init"
    # Initialise the ORB.
    self.orb = CORBA.ORB_init(sys.argv, CORBA.ORB_ID)
  def load(self, *args):
    if (len(args)==0):
      print "usage: load(<stub>, <ior>) or load(<name>)"
    elif (len(args)==1):
      # load the object by name, the nameserver must be running
      # import the Stub of the nameserver
      import NameServer
      # get the name
      name = args[0]
      # connect to the nameserver (to simplify: local reference)
      f = open('nameserver.ref', 'r')
      stringified_ior = f.read()
      f.close()
      # Convert the stringified IOR into an active object reference.
      ns = self.orb.string_to_object(stringified_ior)
      if ns.is_avail(name):
        # get the stub and the ior of the named object
        stub = ns.get_Stub_Name(name)
        ior  = ns.get_IOR(name)
        if os.path.isfile(ior):
          f=open(ior, "r")
          ior = f.read()
          f.close
        try:
          # load the stub
          __import__(stub)
        except:
          print "ERROR:     bad stub"
          sys.exit()
        try:
          print "create the proxy-object"
          return self.orb.string_to_object(ior)
        except:
          print "ERROR:     bad ior"
          sys.exit()
      else:
        print "NameServer doesn't know this name."
    elif (len(args)==2):
      # locate the object with the stub and the IOR (URL or filename)
      ...
    else:
      print "usage: load(<stub>, <ior>) or load(<name>)"
```

Das erstellte Modul *corba_proxy* besteht aus einer Klasse `GeneralProxy`. Diese Klasse hat einen Konstruktor und eine Methode `load`. Der Konstruktor initialisiert den ORB. Die Methode `load` erzeugt ein Stellvertreter-Objekt, das alle Methodenaufrufe dem CORBA-Server weiter leitet.

Die `load`-Methode kann auf zwei verschiedene Arten verwendet werden:

- Einerseits kann die Methode mit einem Argument aufgerufen werden. Das Argument entspricht dem Namen des Servers. Der globale *NameServer* wird kontaktiert[1], damit man von diesem Server den Stub und die IOR in Erfahrung bringt. Mit Hilfe von Stub und IOR wird dann das Stellvertreter-Objekt erzeugt.
- Andererseits kann ein Komponenten-Proxy auch direkt, das heisst ohne Rückfrage beim *NameServer* erstellt werden. Auf diese Weise benötigt man das Wissen, wie der Stub (absoluter Pfad) heisst und wo die Datei mit der IOR abgespeichert ist. Die IOR kann auch hier als URL angegeben werden.

7.1.3 Verwendung

Normalerweise besteht eine CORBA-Applikation aus Clients und Servern. Eine solche Applikation wird mit dem Ziel programmiert, ein spezifisches Problem zu lösen. Das bedeutet, dass die Komponenten dieser Applikation oft genau aufeinander abgestimmt sind (in einander verzahnt) und dadurch meist nur für genau ein Problem verwendet werden können.

Mit Hilfe der in dieser Arbeit erstellten Komponenten-Proxy-Methode können CORBA-Server sehr einfach verwendet werden, das heisst die Methode bringt eine Erleichterung bei der Erstellung der Clients. Die interaktive Verwendung einer CORBA-Komponente (vergleiche dazu einen vollständigen Client `PricingClient` auf Seite 39) sieht zum Beispiel für die “Hello”-Komponente (vgl. im Anhang C.2.1) wie folgt aus:

```
>>> import corba_proxy
>>> gp = corba_proxy.GeneralProxy()
>>> say = gp.load("Hello")
>>> print say.hello()
Hello World!
```

In diesem Beispielcode wird erst das erstellte Modul *corba_proxy* geladen. Danach wird die Klasse `GeneralProxy` instanziert, die für die Erstellung der Stellvertreter-Objekte zuständig ist. Mit der Methode `load` des `GeneralProxy`-Objektes kann ein Stellvertreter-Objekt erzeugt werden. Im obigen Beispiel wurde ein Komponenten-Proxy des “Hello”-Servers erstellt. Mit dem Stellvertreter-Objekt `say` kann nun die Methode `hello` des Servers ausgeführt werden.

Damit die CORBA-Komponenten auf diese Weise verwendet werden können, sind die folgenden zwei Schritte erforderlich:

- Der globale *NameServer* muss einmal gestartet werden und wird immer aktiv bleiben. Dieser *NameServer* ist eine CORBA-Komponente, die in Python geschrieben wurde. Es ist ein zusätzlicher Dienst, der mit “nameserver.py” gestartet wird.
- Je nach Verwendung werden die CORBA-Server gestartet.

[1] Zur Vereinfachung befindet sich die Referenz des *NameServer* in einer Datei, abgespeichert im aktuellen Verzeichnis. Die Referenz könnte zum Beispiel auch mit einer globalen bekannten URL auf dem Web oder via Email publiziert werden.

7.2 Komponenten Framework für CORBA

Im letzten Abschnitt wurde die Komponenten-Proxy-Methode vorgestellt. Wie in Abschnitt 2.3.2 beschrieben, besteht ein ideales Komponenten Framework aus einer Architektur, Skripts, *Glue* und Koordination. In diesen Abschnitt wird ein Komponenten Framework zu dieser Methode eingeführt. In diesem Komponenten Framework wird beschrieben, wie man eine Applikation mit Hilfe der Komponenten-Proxy-Methode erstellt. Es werden vor allem die Elemente des Komponenten Framework erläutert, die im Zusammenhang mit der Komponenten-Proxy-Methode oder CORBA ein anderes Verhalten haben.

7.2.1 Software Architektur

In diesem Komponenten Framework wird in einer Applikation zwischen zwei Softwarebausteinen unterschieden. Einerseits hat man die Komponenten, das sind in dieser Arbeit CORBA-Server. Andererseits gibt es eine Applikationslogik (Clients), die in einem Skript geschrieben sind. Die Kommunikation zwischen verteilten Komponenten übernehmen die CORBA ORBs.

Die Abbildung 7.3 zeigt diese Trennung von Komponenten und Applikationslogik (connector). Eine Applikation besteht aus mehreren unabhängigen Servern, die durch ein Skript verbunden und gesteuert werden.

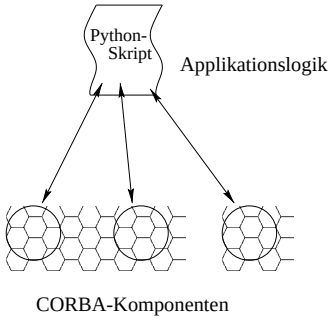

Abbildung 7.3: Architektur des Komponenten Frameworks

CORBA-Server - die Komponenten

In diesem Framework sind CORBA-Server die Softwarekomponenten, das heisst, es können nur CORBA-Komponenten verwendet werden. Die Komponenten dürfen in irgendeiner Programmiersprache mit einer CORBA-Anbindung geschrieben werden.

Damit man ein wiederverwendbares System erhält, muss darauf geachtet werden, dass die CORBA-Komponenten eine unabhängige Einheit darstellen. Das heisst, ein Server sollte nicht fest mit einem anderen Server verbunden sein. Steht ein Server in Abhängigkeit mit einem anderen Server, so muss die Verbindungsaufnahme im Skript explizit gesetzt werden können.

Für die Erstellung von CORBA-Komponenten gibt es grundsätzlich drei Möglichkeiten:

- Eine Komponente wird von Grund auf neu erstellt.
- Man verwendet einen bereits bestehenden CORBA-Server.
- Man bindet auch bereits existierende Nicht-CORBA-Komponenten in das Framework ein.

Die Erstellung eines neuen CORBA-Servers verläuft normalerweise in zwei Schritten:

- Beschreibung des IDL-Interfaces
- Implementation des Servers

Jeder CORBA-Server wird durch sein IDL-Interface beschrieben. Das Interface beschreibt die Dienste, die der Server anbietet. Bei der Implementierung der CORBA-Komponente muss darauf geachtet werden, dass die implementierten Dienste als unabhängige Einheiten definiert werden. Das heisst für eine Komponente, dass sie nicht direkt mit einer anderen Komponente verbunden sein darf (die Verbindung wird vom Skript erstellt.). Unabhängige Dienste können einfacher erstellt und später besser wiederverwendet werden. Jeder Server muss sich beim *NameServer* anmelden. Mit der Methode `object_to_string` wird die Referenz des Objektes als String abgelegt und dem globalen *NameServer* gemeldet.

Verwendet man einen bereits existierenden CORBA-Server, sind zwei Schritte erforderlich. Erstens muss die Komponente auf ihre Unabhängigkeit hin untersucht werden. Zweitens muss der bereits existierende Server sich beim *NameServer* anmelden.

Komponenten wie die des `komplexe Zahlen`-Beispiels (vgl. C.4.1) haben das Problem, dass gewisse Resultate an eine weitere Komponente, den Display-Server, weitergegeben werden. Diese Komponente muss erst unabhängig gemacht werden, und das heisst der Quellcode muss verändert werden.

Alle Software Komponenten, die ohne die Middleware CORBA programmiert sind, können als sogenannte Nicht-CORBA-Komponenten zusammengefasst werden. Eine Nicht-CORBA-Komponente kann aus der Sicht des Komponenten Frameworks als *legacy application* (vgl. Abschnitt 2.4) bezeichnet werden. Wie man solche Komponenten in der Komponenten Architektur einbindet, wird im Abschnitt *Glue* (vgl. Abschnitt 7.2.3) beschrieben.

Applikationslogik - die Steuerung

Die Applikationslogik ist der zweite Baustein der Komponenten Architektur. In der Applikationslogik werden die CORBA-Komponenten verwendet und mit anderen CORBA-Komponenten in Verbindung gebracht. In der Applikationslogik befindet sich die Logik und Steuerung einer Applikation.

Dieser Teil einer Applikation wird in diesem Komponenten Framework mit der Skript-Sprache Python geschrieben. Python verwendet die Komponenten-Proxy-Methode um die einzelnen CORBA-Server zu referenzieren und anschliessend koordinieren zu können. Die Komponenten werden entweder in einem Skript oder von der Python-Shell aus benutzt.

Wie in Absatz 4.1 auf Seite 23 beschrieben, kann eine n-Tier Applikation aus mehreren Diensten bestehen. Ein Dienst besteht in diesem Komponenten Framework aus mehreren verschiedenen CORBA-Komponenten und einer eigenen Applikationslogik (oder auch Dienstlogik). Die Applikationslogik steuert die einzelnen CORBA-Komponenten oder die verschiedenen Dienste und bildet somit eine hierarchische Applikationsstruktur. Die Abbildung 7.4 zeigt wie eine n-Tier Applikation aufgebaut sein kann.

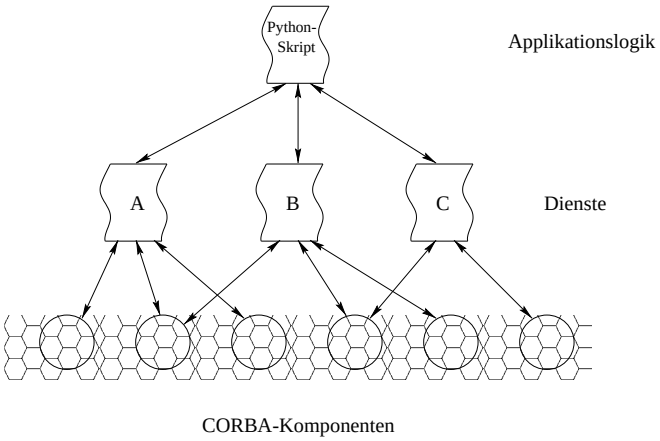

Abbildung 7.4: Hierarchische Struktur einer n-Tier Applikation

Die Trennung zwischen Applikationslogik und Servern fördert das Erstellen von kleinen, generellen und flexiblen Komponenten. Diese Server Komponenten können dadurch später viel einfacher wiederverwendet werden.

Die Abtrennung der Applikationslogik von den CORBA-Komponenten bringt eine bessere Abstraktion. Diese Abstraktion fördert das Verständnis der Applikation oder des Dienstes und vereinfacht das Auffinden von Fehlern, da einerseits die Komponenten einzeln getestet werden können, und andererseits die ganze Steuerung zentralisiert ist. Dieser Ansatz macht verteilte Applikationen einfacher und verständlicher.

Eine Veränderung eines Servers bei gleichbleibendem Interface und gleicher Semantik (Versioningproblem) hat keinen Einfluss auf die Applikationslogik. Das heisst ein sehr wichtiger Teil der Applikation muss bei einem Server *update* nicht erneuert werden.

Bei nachträglichen Abänderungen oder Erweiterungen einer Applikation können bestehende Server-Komponenten einfach wiederverwendet werden und es muss hauptsächlich die Steuerung

im Skript angepasst werden. Solche Beispiele werden in der FAMOOS-Fallstudie gezeigt (vgl. Kapitel 8).

7.2.2 Python - die Skript-Sprache

Die Komponenten-Proxy-Methode ermöglicht Python den Zugriff auf eine CORBA-Komponente ohne dass der Benutzer Kenntnisse von CORBA benötigt. Dieser Zugriff auf die CORBA-Komponenten erfolgt über ein Stellvertreter-Objekt. Eine CORBA-Applikation kann auf diese Weise mit Hilfe eines Python-Skripts erstellt werden. Python eignet sich gut zum Schreiben von kleinen, flexiblen Skripts, die Stellvertreter-Objekte steuern und verbinden. Python hat viele Eigenschaften (vgl. Abschnitt 3.4), die für eine fehlerfreie Entwicklung und praktische Verwendung einer Applikation eingesetzt werden können, wie zum Beispiel das Erstellen einer grafischen Benutzeroberfläche oder eines Testskripts.

Ein Stellvertreter-Objekt wird in drei Schritten erstellt; das sieht im Beispiel der *FruitStore*-Komponente (vgl. Beispiel C.2.3) wie folgt aus:

```
import corba_proxy
gp = corba_proxy.GeneralProxy()
fruitstore = gp.load("FruitStore")
```

Zuerst wird das Modul `corba_proxy` geladen. Dann muss das GeneralProxy-Objekt instanziert werden. Und drittens wird mit der Methode `load` das Stellvertreter-Objekt erzeugt.

Die Skript-Sprache Python hat in diesem Komponenten Framework hauptsächlich die folgenden wichtigen Aufgaben: Einerseits ist die Skript-Sprache Python in diesem Framework dazu da, die einzelnen CORBA-Komponenten zu verbinden und zu steuern, das heisst die Applikationslogik befindet sich in einem Skript. Andererseits ist das Testen von Komponenten eine kontrollierende Aufgabe, die die Komponenten abschliessend testet, bevor sie zum Einsatz kommen. Das Testen von einzelnen Komponenten kann mit Hilfe von Python durchgeführt werden. Abschliessend erfolgt eine Interaktion mit dem Benutzer meist über eine grafische Benutzeroberfläche. Eine grafische Benutzeroberfläche kann in einem Python-Skript erstellt werden.

Testen von Komponenten

Bevor Software-Komponenten verwendet werden, sollten sie getestet werden, denn testen ist ein Prozess, der das Vertrauen der Benutzer aufbaut (vgl. [Sch94]). Unter Testen wird das Prüfen von Software-Komponenten durch Ablauf in einer definierten Umgebung verstanden.

Man kann einerseits die Komponenten interaktiv debuggen, das heisst man erstellt ein Stellvertreter-Objekt (wie in Absatz 7.1.3 beschrieben) und sucht nach Fehlern. Oder es wird andererseits ein Skript geschrieben, dass dann verschiedene Tests durchführt.

Die Abbildung 7.5 zeigt die verschiedenen Testszenarien, die sich wie folgt unterscheiden lassen:

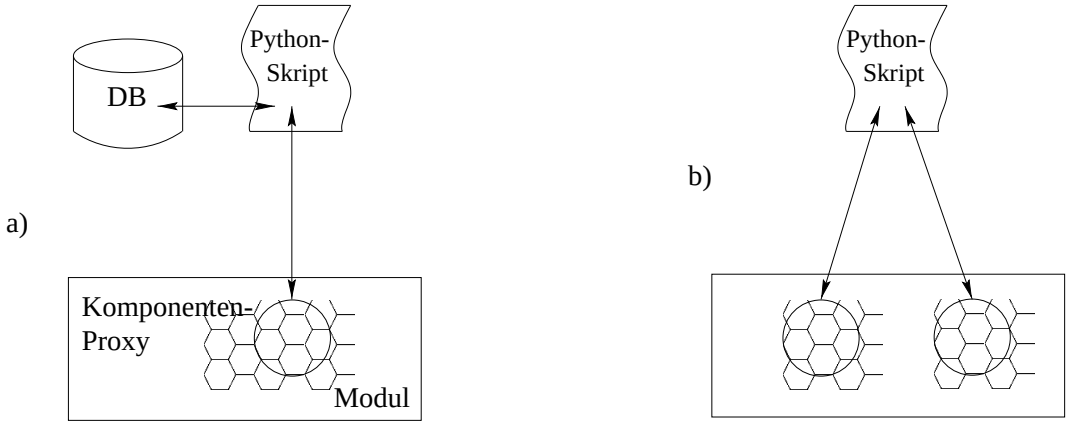

Abbildung 7.5: Test von CORBA-Komponenten

a) Für eine CORBA-Komponente wird ein Test-Skript erstellt, damit kann diese Komponente während der Entwicklung regelmässig auf ihre Korrektheit hin geprüft werden (*regression test*). Eine Komponente wird mit Testdaten und Testresultaten aus einer Datenbank getestet. Korrekturen und nachträgliche Erweiterungen können zu neuen Fehlern führen. Mit Hilfe des Regressionstests wird geprüft, ob die Komponente auch nach der Veränderung noch korrekt ist.

b) Eine alte Komponente wird durch eine neue ersetzt. Das Skript führt bei beiden Komponenten dieselben Operationen aus, und vergleicht anschliessend die Resultate.

GUI (Graphical User Interface)

Ein wichtiger Aspekt einer Applikation stellt die grafische Benutzeroberfläche dar. Es gibt Benutzer, die Applikationen nur auf Grund ihrer praktischen grafischen Oberflächen kaufen und verwenden. GUIs können mit Python geschrieben werden, da mit Python auf eine sehr einfache Art eine grafische Benutzeroberfläche erstellt werden kann. Python kennt mindestens vier verschiedene Möglichkeiten um ein GUI zu erstellen (vgl. [vLF97]).

Die umfangreichste und bekannteste Möglichkeit bietet das *Tkinter*-Modul. Das *Tkinter*-Modul wird von einem *Geometrie-Manager* unterstützt, der die Positionen der *Widgets*[2] in einem Fenster berechnet.

Die Abbildung 7.6 zeigt als Beispiel eine kleine Interaktion mit einem Benutzer für eine Passwortabfrage, die in Python geschrieben wurde. Natürlich können auf diese Weise auch komfortable Menüsteuerungen für grosse Applikation programmiert werden.

[2]Unter einem *Widget* wird ein Bereich in einem Fenster (GUI) verstanden, der ein bestimmtes Aussehen und eine wohl definierte Funktionalität hat. Ein Knopf (Button) oder eine Dialogbox sind Beispiele von Widgets.

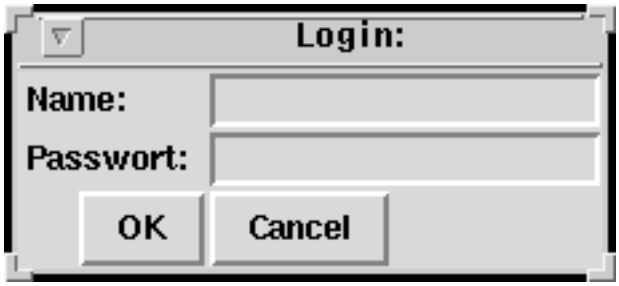

Abbildung 7.6: Grafische Benutzeroberfläche einer Passwortabfrage

7.2.3 *Glue - Wrapping* von *legacy components*

Ein sogenannter *Glue* (engl. leimen) wird verwendet, um Komponenten, die nicht zur vorgegebenen Architektur gehören, anzupassen. Man kann zwischen zwei Typen von *Glue* unterscheiden (vgl. [SN98a]):

- Adaption
- Transformation

Adaption von Komponenten an die Architektur

Damit die Komponenten-Proxy-Methode verwendet werden kann, müssen CORBA-Komponenten vorliegen. Damit eine bereits bestehende Nicht-CORBA-Komponente auch als Komponenten-Proxy verwendet werden kann, muss man diese zuerst "CORBAifizieren". *Wrapping* (Einhüllen) ist die Technik für diese Anpassung (vgl. die Design Pattern *Bridge* in [GHJV95] und *Object Wrapper* in [MM97]). Durch das *Wrapping* werden die Komponenten eingehüllt und mit derselben Funktionalität aber in einem anderen Umfeld wiederverwendet.

Eine Adaption beschreibt, wie eine nicht zur Architektur gehörende Komponente (*legacy application*) zum Framework adaptiert wird (vgl. Abbildung 7.7).

Bestehende Applikationen können in irgendeiner Programmiersprache implementiert sein. Es stellt sich nun das Problem, wie man diese Komponenten zu einer CORBA-Komponente umwandelt. Man unterscheidet zwischen den folgenden Typen von sogenannten Nicht-CORBA-Komponenten:

- Komponenten einer Sprache mit CORBA-Anbindung:
  Eine Softwarekomponente die zum Beispiel in C++ geschrieben ist, kann nicht a priori in eine CORBA-Komponente umgewandelt werden. Es stellt sich die Frage: Sind die in der Komponente verwendeten Signaturen konform zu IDL-Datentypen? Wenn ja, dann kann man direkt aus der Komponente eine CORBA-Komponente erstellen und wie in Absatz 7.2.1 "CORBA-Server - die Komponenten" beschrieben, in das Framework integrieren.

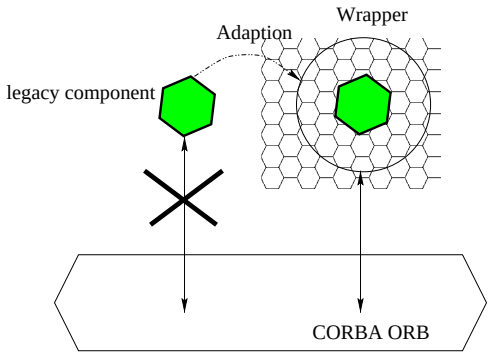

Abbildung 7.7: Adaption einer Komponente durch einen Wrapper

Der Ursprung dieser Problematik liegt in den Sprachanbindungen an CORBA. Es werden alle IDL-Datentypen auf Datentypen einer Programmiersprache abgebildet aber nicht umgekehrt, das heisst, dass gewisse Programmiersprachen Datentypen haben, die von CORBA nicht unterstützt sind. Zum Beispiel kann ein Array von variablem Datentyp, wie man es von Python her kennt, nicht in einen IDL-Datentyp umgewandelt werden.

Softwarekomponenten mit nicht unterstützten Signaturen müssen in einer anderen Form repräsentiert werden. Eine Möglichkeit dies zu erreichen wäre zum Beispiel das Schreiben einer neuen Methode, welche die inkompatible Methode aufruft, aber danach eine IDL-konforme Signatur hat.

- Komponenten einer Sprache ohne CORBA-Anbindung:
  Für Applikationen, die in einer Sprache ohne IDL-Anbindung geschrieben wurden, muss man auf jeden Fall einen *Wrapper* bauen. Das heisst die bestehende Komponente wird umwickelt und mit einem neuen Interface versehen. Diese Umwicklung kann zum Beispiel auf die folgenden zwei verschiedenen Arten realisiert werden:
  - Eine Komponente wird durch eine Spracherweiterung adaptiert.
  - Das *Wrapping* einer Applikation kann mit Hilfe von Parameter-Optionen erfolgen. Zum Beispiel können Systemoperationen (wie beispielsweise ein `sort`-Befehl in UNIX) durch einen Kommandoaufruf und den entsprechenden System-Parametern gewrappt werden. Und dann durch ein CORBA-Server repräsentieren.

- binäre Komponenten:
  Software, die nur in binärer Form vorliegt und es keinen Sourcecode dafür gibt. Solche Software-Komponenten können meistens nicht an CORBA adaptiert werden, da für das *Wrapping* oft auch der Sourcecode leicht angepasst werden muss.

Transformation

Eine Transformation formt inkompatible Datenformate um. Die CORBA-Komponenten definieren im IDL-Interface, wie ihre Methoden verwendet werden können. Im Interface wird für jede

Methode die Signatur angegeben. Die Abbildung 7.8 zeigt schematisch, wie zwei Komponenten auf Grund ihrer inkompatiblen Datentypen nicht verbunden werden können. Mit Hilfe einer Umwandlung in der Applikationslogik können diese Komponenten aber miteinander kommunizieren.

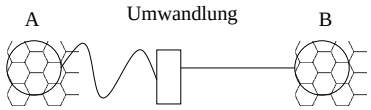

Abbildung 7.8: Transformation von inkompatiblen Datentypen in der Applikationslogik

Der *Time-Server* und der *Display-Server* (vgl. im Anhang C.2.4) sind zum Beispiel CORBA-Komponenten, die zueinander inkompatibel sind. Das Interface des *Time-Server*s sieht wie folgt aus:

```
// Interface of a Time-Server
module Time{
  interface TimeIF{
    short get_hour();
    short get_minute();
  };
};
```

Das Interface `TimeIF` hat zwei Methoden `get_hour` und `get_minute`, die die aktuelle Stunde beziehungsweise die Minuten in Form eines `short` liefert. Die CORBA-Komponente des *Time-Server*s wurde in diesem Beispiel so gewählt, dass sie die Zeit einer anderen Zeitzone[3] liefert.

Der *Display-Server* hat das folgende Interface:

```
// Interface of a Display-Method
interface DisplayIF{
  void display_time(in string time);
};
```

Die Transformation erfolgt im nachfolgenden Skript:

```
import corba_proxy
gp = corba_proxy.GeneralProxy()
time  = gp.load("Time")
display = gp.load("Display")
h = time.get_hour()
m = time.get_minute()
m = (m - 30)%60
h = (h-2)%24
display.display_time("Es ist " + str(h)+":"+str(m)+" Uhr.")
```

[3] Genauer gesagt, musste die Komponente gar nicht speziell angepasst werden, da das JDK 1.1.6 dieses Fehlverhalten hat.

In dem Code des aufgelisteten Skripts werden zwei Transformationen gemacht. Erstens liefert der *TimeServer* die Zeit einer anderen Zeitzone, das heisst die Uhrzeit muss um die Zeitverschiebung angepasst werden. Zweitens muss dem *Display-Server* ein String übergeben werden. Die Uhrzeit muss daher von einem `short` in einen String umgewandelt werden (Typenumwandlung, *casting*).

7.3 Zusätzliche Möglichkeiten

Die Komponenten-Proxy-Methode vereinfacht die Verwendung von CORBA-Servern. Das Komponenten Framework hilft Applikationen zu bauen, die aus einfachen, flexiblen Komponenten bestehen. Die Methode soll den Einsatz von Python in CORBA fördern.

Der *Agent* und der *Intelligent Proxy* sind zwei mögliche Anwendungsbereiche, bei denen man diese Methode einsetzen kann.

7.3.1 Intelligent Proxy

Die in diesem Kapitel vorgestellte Methode *corba_proxy* ermöglicht das Erzeugen von Stellvertreter-Objekten, die jeweils die Methodenaufrufe an den Original-Server weiterleiten. Die Tätigkeit der Stellvertreter-Objekte besteht lediglich im Weiterleiten von Methodenaufrufen. Stellvertreter-Objekte haben im Allgemeinen keine weiteren Funktionalitäten.

Die Abbildung 7.9 zeigt den Ansatz, wie Komponenten-Proxies mit einer gewissen Intelligenz versehen werden könnten. Der Ansatz besteht darin, dass die Stellvertreter-Objekte in Verbindung mit einem zusätzlichen Programm und/oder einer Datenbank stehen. Mit Hilfe der Erweiterung des Komponenten-Proxy könnten zum Beispiel die folgenden Verbesserungen realisiert werden:

- Es besteht die Möglichkeit zusätzliche Sicherheitsvorkehrungen zu treffen. Man kann sich beispielsweise eine Passwort-Abfrage vorstellen. Ein Benutzer kann nur einen Komponenten-Proxy benutzen, wenn er das richtige Passwort eintippt. Es ist auch denkbar, einem Benutzer nur partielle Zugriffsrechte zu erteilen. Jedem Benutzer könnte man so ein persönliches Zugriffsprofil geben.

- Der Komponenten-Proxy speichert ein lokales Abbild der Serverdaten. Auf diese Weise könnte das Stellvertreter-Objekt einfache Abfragen gleich selbst beantworten. Dieses lokale Beantworten von Anfragen hat einen positiven Einfluss auf die *Performance* sowie die Netzwerkbelastung. Die Gewährleistung der Datenintegrität stellt hingegen einen Schwachpunkt dar. Es müsste zum Beispiel ein *Polling* betrieben werden, um sich beim Server zu erkundigen, ob die Daten noch aktuell sind.

- Das Stellvertreter-Objekt sendet den Methodenaufruf nicht nur an den Server, sondern auch noch an eine weitere Instanz. Ein Analyse-Tool liesse sich zum Beispiel auf diese Weise auf den Komponenten-Proxy aufsetzen (vgl. [MK98]).

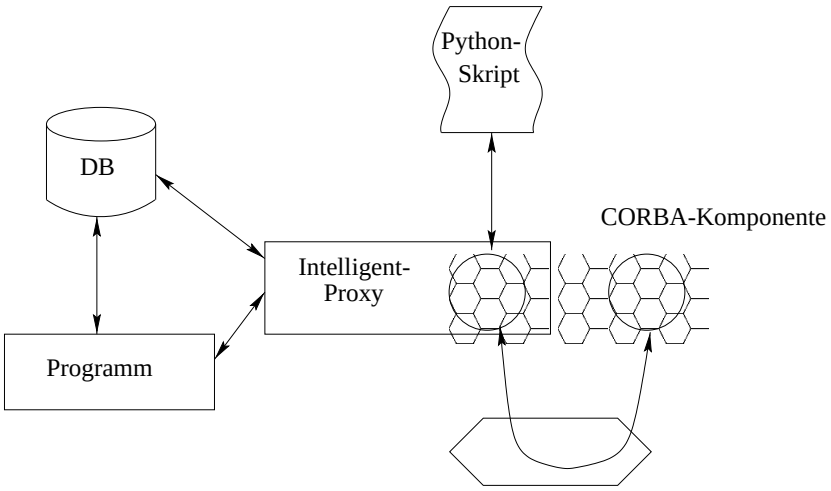

Abbildung 7.9: Analyse Tool mit Hilfe des Intelligent Proxy

Die Erweiterung der Komponenten-Proxy-Methode kann zum Beispiel durch eine Klasse *Intelligent_Proxy* realisiert werden. Die Klasse *Intelligent_Proxy* erbt von drei Klassen wie aus der Abbildung 7.10 ersichtlich ist. Die *Intelligent_Proxy*-Klasse ist ein sogenanntes *mixin* (vgl. [Gri98]), das heisst, die Klasse besitzt keine vollständige Funktionalität, wird aber zur Kombination mit weiteren Klassen für die Mehrfachvererbung verwendet.

Wichtig für die Erweiterung der *Intelligent_Proxy*-Klasse ist die Klasse `Proxy`, welche auf dem *Proxy Design Pattern* in Python (vgl. [Ves97]) basiert. Für den nachfolgenden Code der Klasse *Intelligent_Proxy* ist die pythonspezifische Abarbeitungsreihenfolge (von links nach rechts) der geerbten Klassen von Bedeutung (vgl. [vR96]):

```
class Intelligent_Proxy(Extension, Restriction, Proxy):
  def __init__(self, subject):
    Proxy.__init__(self, subject)

# based on the Python's Proxy-Pattern
class Proxy:
  def __init__(self, subject):
    self.__subject = subject
  def __getattr__(self, name):
    logfile(name)                          # for an analyse tool
    return getattr(self.__subject, name)
```

Die Klasse des `Intelligent_Proxy` ermöglicht Stellvertreter-Objekten die folgenden neuen Möglichkeiten:

- Erweiterung von CORBA-Komponenten
- Einschränkung von CORBA-Komponenten

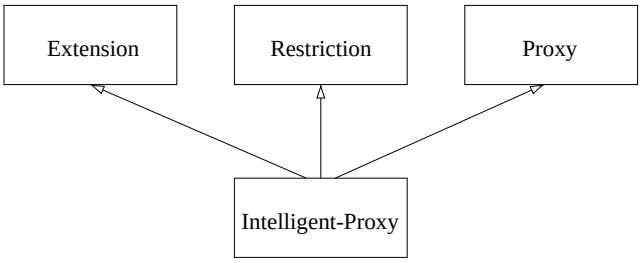

Abbildung 7.10: UML Klassendiagramm des Intelligent Proxy

- Ansatz für ein Analyse-Tool

Die folgende Klasse `Extension` ist eine Superklasse der Klasse `Intelligent_Proxy` und ermöglicht eine Erweiterung von bestehenden CORBA-Komponenten.

```
# This is an extention of the Hello-Server
class Extension:
  def good_morning(self):                  # extend the component with a method
    return "good morning!!\n"
```

In der oben definierten Klasse `Extension` wird die in Java geschriebene *Hello*-Komponente (vgl. im Anhang C.2.1) um die Methode `good_morning` erweitert.

Auch die Klasse `Restriction` ist eine Oberklasse von der Klasse `Intelligent_Proxy` und erlaubt es bestehende CORBA-Komponenten einzuschränken.

```
class Restriction:
  def say_hello(self):
    pass
```

Die Klasse `Restriction` macht die Methode der *Hello*-Komponente unzugreifbar, das heisst die Methode `say_hello` wird überschrieben durch die Klasse `Restriction`.

Der Ansatz für ein Analyse-Tool wird in der Klasse `Proxy` gelegt. In dieser Klasse wird die Methode `logfile` aufgerufen und die sieht in der einfachsten Form wie folgt aus:

```
def logfile(name):
  print "log", name
```

Die Verwendung sieht zum Beispiel wie folgt aus:

```
>>> gp = corba_proxy.GeneralProxy()
GeneralProxy init
>>> helloserver = gp.load("Hello")
create Object
>>> print helloserver.say_hello()
Hello world !!
```

```
>>> ip = Intelligent_Proxy(helloserver)
>>> print helloserver.say_hello()
Hello world !!
>>> print ip.say_hello()
None
>>> print ip.good_morning()
good morning!!
```

In dieser beispielhaften Verwendung wird ein Stellvertreter-Objekt `helloserver` von dem CORBA-Server *Hello* erzeugt. Diese Komponente wird anschliessend mit der Klasse Intelligent_-Proxy, wie oben definiert, um die Methode `good_morning` erweitert und um die Methode `say_hello` eingeschränkt und heisst nun `ip`.

7.3.2 Agent (*Mobile Agent*)

Unter einem *Mobile Agent* kann man sich dasselbe vorstellen, wie unter einem Handelsreisenden: Jemand, der von Ort zu Ort reist, um dort jeweils gewisse Geschäfte (Tätigkeiten) zu erledigen und kommt mit den gewünschten Resultaten zurück (für eine genaue Definition vgl. zum Beispiel [FG96] oder [Mil96]). In einem lokalen Netzwerk könnte ein solcher Agent zum Beispiel die folgenden Aufgaben erledigen und dadurch eine geringere Netzwerkbelastung erreichen:

- Laufende Programme eines bestimmten Benutzers suchen
- Programme, die zu ausgeloggten Users gehören, finden.
- Dateien auf "/var/tmp", die älter als 30 Tage sind, suchen.
- Suchen von Dateien im lokalen Filesystem, die $>$10MB (zum Beispiel überlaufende Log-Files) sind.

Mit Hilfe der Komponenten-Proxy-Methode kann ein *Mobile Agent* erstellt werden. Ein CORBA-Server kann einen Agenten entgegennehmen, ihn seine Tätigkeit ausführen lassen und weiterleiten. Ein Agent führt bei einem entsprechenden Server seine Tätigkeit (Skript) aus und geht je nach Auftrag zum nächsten Server oder zurück. Man kann die folgenden zwei verschiedenen Typen von *Mobile Agent* unterscheiden:

- Der Agent besucht nur eine Maschine und bringt anschliessend die Resultate zurück. Danach geht er zum nächsten Rechner. In Abbildung 7.11 wird dieser Agent *Sending Agent* genannt, da er pro Auftrag nur zu einem Kunden gesandt wird und anschliessend wieder zurückkommt.
- Ein *Travelling Agent* wird von einem Auftraggeber (Python-Skript) von Maschine zu Maschine geschickt, bis er alle möglichen Kunden (Rechner) besucht hat (vgl. Abbildung 7.12). Bei der Rückmeldung der Resultate kann man zwischen einer direkten (am aktuellen Ort kommt eine direkte Meldung zum Auftraggeber) und einer stufenweisen Rückmeldung an den jeweiligen Aufrufer (die Resultate gehen durch jeden bisher besuchten Rechner zurück zum Auftraggeber) unterscheiden.

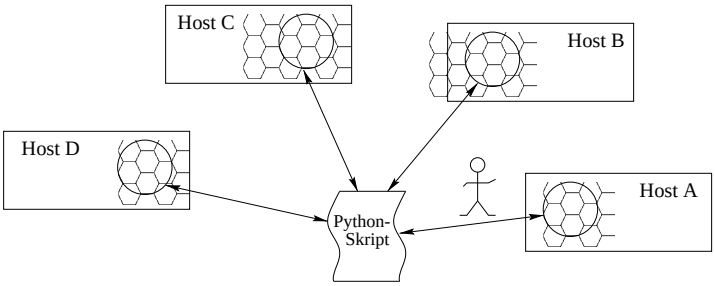

Abbildung 7.11: *Sending Agent*

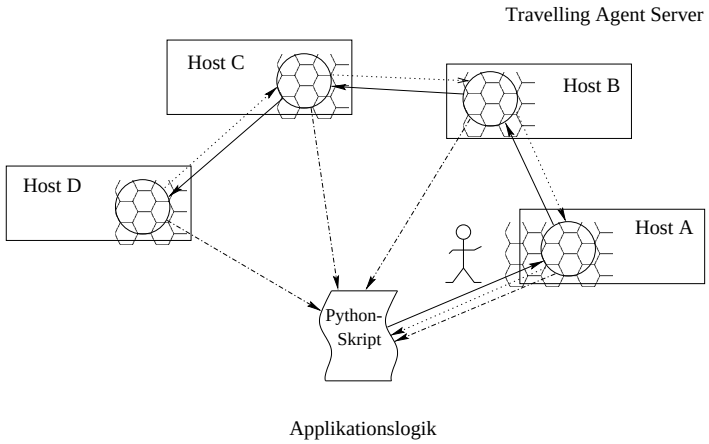

Abbildung 7.12: *Travelling Agent*

Travelling Agent - ein Beispiel

In diesem Absatz wird als Beispiel ein *Travelling Agent* beschrieben, der von Rechner zu Rechner geht, um zu schauen, ob sich im "/var/tmp"-Verzeichnis die gesuchte Datei befindet. Die verteilte Applikation besteht aus einer Applikationslogik und mehreren *Travelling Agent* Servern. Die Applikationslogik erzeugt einen Agenten, der mit einem Auftrag von Server zu Server geht um gewisse Tätigkeiten auszuführen.

Der Agent enthält eine Agentenlogik (eigene Applikationslogik), damit er selbstständig durch das lokale Netz navigieren kann. Mit Hilfe dieser Agentenlogik kann der Agent einen Auftrag in einem Server erledigen und die Resultate sammeln.

Das IDL-Interface der *Travelling Agent*-Server sieht wie folgt aus:

```
// IDL Interface for the travelling agent server

module Agent {
  exception Invalid_Script{};
  interface AgentIF {
    string my_script(in string script) raises (Invalid_Script);
```

```
  };
};
```

Das Interface besteht aus einer Methode `my_script`, die eine Ausnahme generieren kann. In dieser Methode wird der *Travelling Agent* als Argument in Form eines Skripts zum Server gelangen. Der *Travelling Agent* Server wird dann den Agenten mit `exec` starten. Gibt es bei der Ausführung einen Fehler, so wird eine Ausnahme erzeugt.

In der Applikationslogik hat es eine Methode `agent`, die den *Travelling Agent* generiert und auf den Weg schickt. Über das Argument (`to_do`) kann man dem *Travelling Agent* den Auftrag erteilen. Der Auftrag besteht aus einem Python-Skript und muss eine Variable `result` haben, wo das Resultat des Auftrages abgelegt wird. Nachfolgend wird die Schlüsselmethode `agent` der Applikationslogik gezeigt:

```
# application logic; build the travelling agent
# the argument "to_do" contains the job (python-script) for the travelling agent

  def agent(self, to_do):
  # build the head of the travelling agent
    head = """
import corba_proxy, string
# servers to visit
servers = string.splitfields(serverlist, ",")
if len(servers):                # is there a server
  nextserver = servers[0]  # take the first
  servers.remove(servers[0])
  serverlist = ""
  for s in servers:
    serverlist = serverlist + s + ","
else:
  nextserver = ""
serverlist = serverlist[:-1]
"""

# build the tail of the travelling agent
    tail = """
if len(nextserver):
  #  the data of the travelling agent contains tree informations
  # separated by a "<:--:>"
  data = home+ "<:--:>"+serverlist+"<:--:>"+ result
  # the travelling agent has a script and data separated by a "<-|||->"
  TA = script + separator + data
  # connect to the next TAServer
  gp = corba_proxy.GeneralProxy()
  ta = gp.load(nextserver)
  # send the travelling server to do the job
  ta.my_script(TA)
"""

    # the script (logic of the agent) is a string containing
    # a head, a script (argument) and a tail
```

```
script = head + to_do + tail
separator = "<-|||->"
# connect to the NameServer
gp = corba_proxy.GeneralProxy()
nameserver=gp.load("NameServer", "www.iam.unibe.ch/~held/IOR/nameserver.ref")
# initialize
serverlist = ""
firstserver = ""
# where to go?
for i in range(nameserver.number_available_servers()):
  server = nameserver.get_Server_Name(i)
  if string.count(server, "TA_Server"):
    if len(firstserver):
      serverlist = serverlist + server + ", "
    else:
      firstserver = server
serverlist = serverlist[:-1]
# where is the application logic
machine = posix.uname()[1]
print "home: ", machine
 #  home-host <:--:> serverlist <:--:> resultstring
data = machine + "<:--:>" + serverlist + "<:--:>0"
# the travelling agent has a script and data
ta = script + separator + data
# send the script and the data to the next TAServer
if len(firstserver):
  gp = corba_proxy.GeneralProxy()
  ta_home = gp.load(firstserver)
  ta_home.my_script(ta)
else:
  print "run a TAServer.py"
```

Die Methode `agent` erstellt einen *Travelling Agent* und schickt ihn auf den Weg. Der *Travelling Agent* besteht aus einer Agentenlogik `script` und Daten `data` (vgl. Abbildung 7.13).

Die Daten werden durch das folgende Protokoll übertragen:

```
data = home+ "<:--:>"+serverlist+"<:--:>"+ result
```

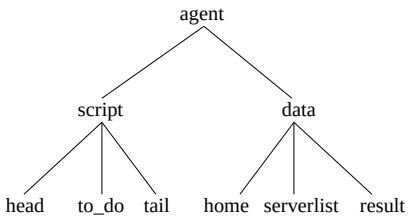

Abbildung 7.13: Aufbau des *Travelling Agent*

Jeder *Travelling Agent* speichert in `home` die Adresse seiner Applikationslogik, damit die Resultate zurückgesendet werden können. Die `serverlist` enthält die Namen aller Server, die der

Agent noch besuchen muss. In `result` befinden sich alle bisher gesammelten Resultate (ein Art Quittungen).

Das Skript, die Agentenlogik, geht immer mit dem *Travelling Agent* mit und wird in drei Teile aufgeteilt und sieht wie folgt aus:

```
script = head + to_do + tail
```

Im `head` werden die aktuellen Module geladen, sowie der nächste zu besuchende Server bestimmt. `to_do` ist ebenfalls ein Skript, das schon als Argument der Methode `agent` der Applikationslogik übergeben wurde und das die eigentliche Aufgabe des Agenten beinhaltet. In `tail` wird der *Travelling Agent* wieder zusammen gebaut und zum nächsten *Travelling Agent* Server geschickt.

Der *Travelling Agent* besucht auf seiner "Reise" Server, die wie folgt aussehen:

```
# travelling agent server

  def my_script(self, ta):
    local_name_space = {'__name__':'__main__'}
    script, data = string.splitfields(ta, "<-|||->")
    home, serverlist, result = string.splitfields(data, "<:--:>")

    # pass data to the script(here: separator, script, ...)
    local_name_space["separator"] = "<-|||->"
    local_name_space["script"] = script
    local_name_space["home"] = home
    local_name_space["serverlist"] = serverlist[:-1]
    local_name_space["result"] = result

    # start the script
    exec(script)in local_name_space, local_name_space

    # send the result home
    gp = corba_proxy.GeneralProxy()
    ta_home = gp.load("TAHomeServer")
    if len(local_name_space["result"]):
      res = local_name_space["result"]
    else:
      res = "No results"

    machine = posix.uname()[1]
    # return the results to the home town
    ta_home.collect_result("Results from " + machine + ":\n" + res)
```

Im obigen Server werden gemäss dem in diesem Beispiel verwendeten Protokoll Agentenlogik und Daten, sowie die einzelnen Informationen eruiert. Diese werden dann über gemeinsame Variablen dem Skript zur Verfügung gestellt. Das Skript wird dynamisch mit `exec` gestartet. Abschliessend wird das Resultat (wieder über eine gemeinsame Variable `result`) der Applikationslogik zurückgegeben.

7.4 Grenzen

In diesem Abschnitt werden Grenzen der Komponenten-Proxy-Methode aufgezeigt. Es wird erläutert, wieso mit diesem Modul *corba_proxy* im Moment noch kein dynamischer Methodenaufruf möglich ist. Es ist aber möglich einen sogenannten "halb-dynamischen" Methodenaufruf zu generieren. Python-Komponenten können nicht automatisch zu CORBA-Komponenten umgewandelt werden.

7.4.1 Kein dynamischer Methodenaufruf

Ein CORBA-Client kann mit Hilfe des *Dynamic Invocation Interface* dynamisch Methoden aufrufen, das heisst, der Client kann auch ohne das compilierte Interface, den Stub, auskommen (vgl. 4.3.4). Um einen dynamischen Methodenaufruf zu erzeugen, wird das *Interface Repository* verwendet. Das *Interface Repository* ist seit CORBA 2.0 standardisiert. Die ORB-Hersteller sollten eigentlich alle das *Interface Repository* einheitlich implementiert haben. Somit sollte ein dynamischer ORB-übergreifender Zugriff auf die CORBA-Komponenten möglich sein.

Bei den durchgeführten Testversuchen konnte aber festgestellt werden, dass ein solcher Zugriff im Moment noch nicht möglich ist, da das Interface des referenzierten Objektes nicht ermittelt werden kann. Genauer gesagt, konnte der Zugriff auf das entfernte *Interface Repository* nicht realisiert werden. Diese unglückliche Situation wurde auf Anfrage vom Entwickler des *Fnorb*s bestätigt und als generelles CORBA 2.0 Problem beschrieben.

Der Versuch von Python über den *Fnorb* und den *VisiBroker* auf einen CORBA-Server zuzugreifen, endete beim folgenden Programmschritt:

```
full_int_desc = intdef.describe_interface()
==> break down.
```

Das hat zur Folge, dass ein ORB-unabhängiger, dynamischer Ansatz (wie die Komponenten-Proxy-Methode) ohne einen Eingriff auf einen ORB verunmöglicht wird.

Im globalen *NameServer* kann das IDL-Interface abgespeichert werden. Somit könnte man die Interfacebeschreibung der Server auch aus dem *NameServer* erlangen. Dieser Ansatz würde aber einen Nachbau eines *Interface Repository* bedeuten.

7.4.2 "Halb-dynamischer" Methodenaufruf

Es hat sich gezeigt, dass ein Methodenaufruf mit der Komponenten-Proxy-Methode nur dann zustande kommt, wenn neben der IOR auch der durch den Compiler erzeugte Stub des Servers vorliegt. Ist der Stub noch nicht vorhanden, so bietet der globale *NameServer* eine zusätzliche Möglichkeit, indem er neben dem Servername, Stub und IOR auch noch das IDL-Interface speichern kann. Das ermöglicht dem Client zur Laufzeit den Compiler zu starten und sich den Stub zu erzeugen. Anschliessend kann das Server-Objekt referenziert werden und statisch eine Methode

aufrufen.

Als Beispiel wird nachfolgend der Stub des *Counter Server* (vgl. im Anhang C.2.2) dynamisch generiert:

```
# dynamic generation of a stub
import fnidl
idl_file = ns.get_IDL("CountServer")
print "IDL-file: ", idl_file
print "Generating the stub. Wait..."
fnidl.main(["", idl_file])
```

Zur Vereinfachung wird die Datei des IDL-Interfaces lokal abgelegt. Der *NameServer* muss somit nur den Dateinamen kennen. Danach wird mit dem Modul “fnidl” (Compiler von IDL nach Python) der Stub erzeugt.

7.4.3 Abbildung von Python-Komponenten nach CORBA

Die Abbildung 7.14 zeigt, dass CORBA-Komponenten mit Hilfe der Komponenten-Proxy-Methode nach Python abgebildet werden können (vgl. Abschnitt 7.1). Die Methode *corba_proxy* erstellt von einer CORBA-Komponente ein Stellvertreter-Objekt, auf das dann Python zugreifen kann. Der umgekehrte Ansatz bietet verschiedene Probleme. Dieser Absatz soll Probleme aufzeigen, die ein automatisches Abbilden von Python-Objekten nach CORBA hat.

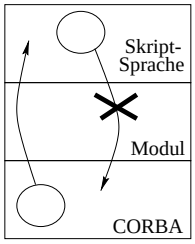

Abbildung 7.14: Kein Python-Proxy für CORBA

Um Python-Objekte automatisch nach CORBA abbilden zu können, muss man aus einem Objekt das Interface bestimmen. Für Python-Programme ist das aber nicht so einfach möglich, da die Sprache dynamisch typisiert ist. Bei einer Methode werden die Argumente nicht deklariert und werden nur zur Laufzeit bestimmt. Die Signatur einer Methode kann daher oft nur durch den Kontext bestimmt werden. Die folgende Methode `out` ist zum Beispiel erst zur Laufzeit eindeutig bestimmt:

```
def out(x):
  print x
```

Ist das Argument `x` der Methode `out` ein Integer oder aber ein String? Um zu bestimmen, was das Argument `x` für einen Datentyp hat, müsste man den Programmcode analysieren. Es kann

aber auch sein, dass der Datentyp von `x` nicht eindeutig bestimmt werden kann, wenn zum Beispiel eine Interaktion mit dem Benutzer vorliegt. In Python ist es auch möglich, dass das Attribut `x` einmal ein String und danach wieder als Integer aufgerufen werden kann.

Python hat verschiedene Möglichkeiten die Argumente einer Methode zu definieren. Die Argumente einer Methode können in Python zusätzlich auch optional, von variabler Anzahl oder ungeordnet definiert werden (vgl. 3.4). Nachfolgend ist ein Beispiel einer Methode definiert, die mit variabler Anzahl Argumenten gerufen werden kann:

```
def sum(*x):
  s = 0
  for i in x:
    s = s + i
  return s
```

Die Methode `sum` kann zum Beispiel mit `sum(2, 3)` oder aber mit `sum(1, 2, 3)` gerufen werden. CORBA unterstützt in seiner IDL keine solche Interfaces.

Ein Array kann in Python Elemente von verschiedenen Datentypen haben. Das folgende Array ist beispielsweise in Python erlaubt:

```
a = [2, "da"]
```

Das oben gezeigte Array `a` enthält einen Integer “2” und einen String “da”. In CORBA ist ein solches Array unmöglich, da ein Array nur von einem Datentyp sein darf.

Kapitel 8

Fallstudie: FAMOOS

Ziel der Fallstudie ist es, die in dem letzten Kapitel entwickelte Methode *corba_proxy* und dessen Komponenten Framework zu validieren. Bestehende Komponenten werden durch den *Glue* des Komponenten Framework adaptiert.

8.1 Das FAMOOS-Projekt

Die *Software Composition Group*[1] (SCG) arbeitet unter anderem an einem Forschungsprojekt FAMOOS. FAMOOS ist die Abkürzung für *Framework-based Approach for Mastering Object-Oriented Software Evolution* und ist ein ESPRIT-Projekt[2]. Es handelt sich um ein drei Jahre dauerndes Forschungsprojekt, welches am 1. September 1996 begonnen hat. Dieses Forschungsprojekt hat das folgende Ziel:

Existierende Software soll durch eine geeignete *Reengineering* flexibler gemacht werden.

Im FAMOOS-Forschungsprojekt werden Methoden für das *Reengineering* entwickelt. Mit Hilfe der neuen *Reengineering*-Methoden sollten veraltete objektorientierte Applikationen in ein Framework umgewandelt werden.

Die Abbildung 8.1 (von [DT98])) zeigt den FAMOOS *Reengineering*-Lebenszyklus, der in die folgenden sechs Schritte unterteilt wurde:

(1) Analyse der Anforderungen: In den Anforderungen werden die Ziele für das Reengineering bestimmt.

(2) Erfassen des Modells: Die Einarbeitung in das Modell erfolgt über das Verständnis von Applikation und deren Dokumentation.

[1] Die *Software Composition Group* ist die Forschungsgruppe von Prof. Dr. O. Nierstrasz an der Universität Bern.

[2] ESPRIT-Projekte sind Forschungs- und Entwicklungsaktivitäten auf dem Gebiet der Informationstechnologie. FAMOOS ist das ESPRIT-Projekt mit der Nummer 21'975.

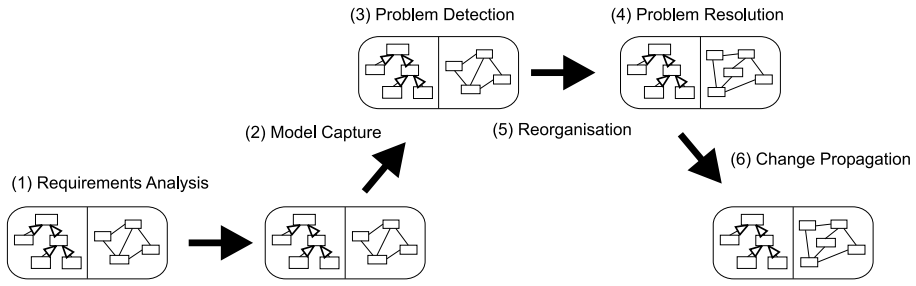

Abbildung 8.1: Famoos *Reengineering*-Lebenszyklus

(3) Problemanalyse: Mit der Analyse der Probleme untersucht man Flexibilitäts- und Qualitätsaspekte.

(4) Problemlösung: Der Lösungsansatz definiert die Wahl der neuen Software Architektur.

(5) Reorganisation: Die alte Applikation wird durch eine Umwandlung zu einem neuen Modell verändert.

(6) Verbreitung der Veränderung: Das neue System wird bei den Kunden eingeführt, damit von der neuen Version profitiert werden kann.

Für eine gezielte Forschung wurden die folgenden drei Forschungsgebiete bestimmt:

- Metriken & Heuristiken
- Modularisierung (*Grouping*)
- Reorganisation

Die internationalen FAMOOS-Partner[3] erstellen in den einzelnen Forschungsgebieten ihre eigenen Prototypen von Applikationen für die geplanten Experimente. Die Applikationen können in verschiedenen Programmiersprachen implementiert werden. Diese Applikationen werden in einer Bibliothek als Basis für weitere Experimente zur Verfügung gestellt.

Damit sich die Applikationen verständigen können, wurde ein Modell für den Datenaustausch entwickelt. Dieses Modell heisst *FAMOOS Information Exchange Model* (vgl. [DT98]) und macht eine sprachunabhängige Repräsentation eines objektorientierten Source-Codes. Die Repräsentation bietet genug Informationen für *Reengineering*-Aufgaben.

Für den effektiven Datenaustausch wird das einheitliche Format CDIF (*Common Data Interchange Format*) verwendet, das heisst, eine Applikation kann in ein Modell abgebildet werden und wird durch CDIF festgehalten. CDIF ist ein industrieller Standard (vgl. [CDI98]), der als flacher ASCII-Strom repräsentiert wird. CDIF eignet sich gut für die Forschung, da dieser ASCII-Strom

[3]Die Forschungspartner sind Nokia ($www.nokia.com$), Daimler-Benz ($www.daimler-benz.com$), FZI ($www.fzi.de$), und Sema Group ($www.sema.es$).

ein verständlicher Code darstellt und dadurch eine Fehlersuche vereinfacht.

8.2 Einleitung

Für die vorliegende Arbeit ist FAMOOS geeignet, da in diesem Forschungsprojekt verschiedene voneinander unabhängige Komponenten entwickelt werden. Die Abbildung 8.2 gibt einen Überblick des aktuellen Standes der Forschung in diesem Projekt. Die bereits erstellten und verwendeten Komponenten werden nachfolgend kurz beschrieben:

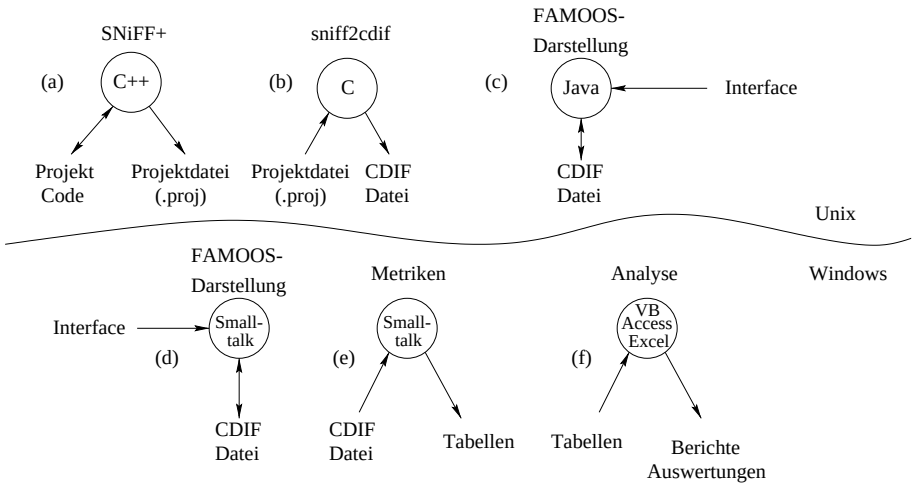

Abbildung 8.2: Famoos Komponenten

(a) *SNiFF+* ist eine sprachunabhängige Entwicklungsumgebung von der Firma *TakeFive software* und ist auf dem Internet unter der folgenden URL zu finden: *http://www.takefive.com/sniff/*. Mit *SNiFF+* wird der Quellcode einer Applikation geladen, das ermöglicht dann dem Entwickler ein komfortables Arbeiten mit einer grafischen Benutzeroberfläche. Ein Browser erlaubt verschiedene Ansichten wie zum Beispiel das Anzeigen aller Unterklassen einer Klasse und stellt Werkzeuge zur Verfügung. *SNiFF+* speichert alle Informationen zu einer Applikation in einer Datei. Diese Datei heisst Projekt-Datei und hat die Endung `.proj`.

(b) Die in C geschriebene Komponente *sniff2cdif* liest die von *SNiFF+* gespeicherten Informationen aus einer Projekt-Datei und wandelt diese in eine CDIF-Repräsentation um. Die CDIF-Repräsentation wird in einer Datei abgelegt.

(c) Die *FAMOOS-Darstellungs*-Komponente ist eine Implementierung des *FAMOOS Information Exchange Models*, welche objektorientierten Source-Code über ein Interface repräsentiert. Diese Komponente wurde in Java geschrieben. Die *FAMOOS-Darstellungs*-Komponente liest aus einer CDIF-Datei die Informationen eines Programms und erstellt intern ein

Modell. Das Modell kann über ein Interface abgefragt oder auch verändert werden. Die *FAMOOS-Darstellung* kann eine veränderte Modell-Repräsentation wieder in eine Datei schreiben.

(d) Diese *FAMOOS-Darstellungs*-Komponente ist analog zu (c), aber in Smalltalk geschrieben.

(e) Die in Smalltalk geschriebene Komponente *Metriken* liest den CDIF-Code aus einer Datei und erzeugt Tabellen auf Grund von vorgegebenen Metriken.

(f) Die Komponente *Analyse* liest die Tabellen aus einer Datei in VisualBasic, Access oder Excel ein und erzeugt für die verschiedenen Metriken Berichte und Auswertungen.

Die Komponenten *SNiFF+*, *sniff2cdif* und eine FAMOOS-*Darstellung* wurden auf dem Betriebssystem Unix entwickelt. Die Komponenten der anderen FAMOOS-*Darstellung*, der *Metriken* und der *Analyse* sind Komponenten, die auf einem Windows-Betriebssystem laufen. Hierzu ist zu sagen, dass die in Java und Smalltalk implementierten Komponenten auch auf anderen Betriebssystemen laufen.

8.3 Aufgabe

In dieser Fallstudie wurden aus dem FAMOOS-Forschungsprojekt zwei Komponenten ausgewählt und mit Hilfe von dem im letzten Kapitel beschriebenen Framework verwendet. Es sind die zwei folgenden Komponenten:

- *sniff2cdif* (in C programmiert)
- die in Java geschriebene FAMOOS-*Darstellung*

Diese Komponenten wurden ausgewählt, da mit ihnen die Basis für die metrischen Untersuchungen gelegt wird und die Mechanismen des Komponenten-Framework (vgl. Kapitel 7.2) gezeigt werden. Beide Komponenten stellen aus der Sicht der Komponenten-Proxy-Methode *legacy component*s dar, da sie benötigt werden und keine CORBA-Komponenten sind. Die FAMOOS-*Darstellung* wurde in Java geschrieben und Java ist eine Programmiersprache, für die es eine CORBA-Anbindung gibt. Für die Programmiersprache C gibt es auch eine CORBA-Anbindung, doch in dieser Arbeit wird die Annahme getroffen, dass die *sniff2cdif*-Komponente eine *legacy component* von einer Programmiersprache ohne CORBA-Anbindung sei. Auch für eine solche Komponente soll ein *Wrapping*-Mechanismus gezeigt werden.

Nachdem die Komponenten *sniff2cdif* und FAMOOS-*Darstellung* zu CORBA-Komponenten umgewandelt worden sind, werden zwei Erweiterungen zu diesen Komponenten erstellt. Einerseits wird für die FAMOOS-*Darstellung* eine grafische Benutzeroberfläche in Python entwickelt. Diese Benutzeroberfläche soll bestehende Modelle veranschaulichen oder verändern können oder aber ein Modell von Grund auf neu erstellen. Andererseits wird eine zusätzliche CORBA-Komponente auch in Python erstellt, welche die Tabellen für die metrischen Analysen erzeugt.

Die Abbildung 8.3 zeigt das Ziel dieser Fallstudie. Die zwei FAMOOS-Komponenten werden mit *Wrappern* an das Komponenten-Framework der Komponenten-Proxy-Methode angepasst. Der *TableGenerator* ist eine zusätzliche CORBA-Komponente, die Tabellen für die *Analyse*-Komponente erstellt. Die grafische Benutzeroberfläche wird mit einem Python-Skript realisiert. Mit Python kann auf alle drei Komponenten über Komponenten-Proxies zugegriffen werden.

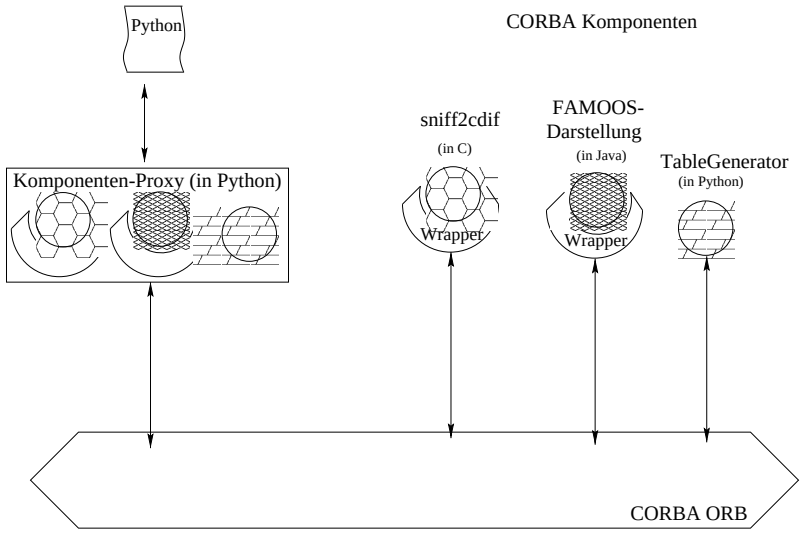

Abbildung 8.3: Famoos Szenario

Anhand von exemplarischen Beispielen werden die neuen Verwendungsmöglichkeiten im FAMOOS Forschungsprojekt aufgezeigt wie zum Beispiel das programmiersprachenunabhängige Verbinden der nun verteilten FAMOOS-Komponenten.

8.4 *Wrapping* der beiden bestehenden Komponenten

Damit die bestehenden FAMOOS-Komponenten mit der Komponenten-Proxy-Methode verwendet werden können, müssen CORBA-Komponenten vorliegen. Nachfolgend wird gezeigt, wie die Komponenten *sniff2cdif* und *FAMOOS-Darstellung* zu CORBA-Komponenten umgewandelt werden. Dieser Umwandlungsvorgang wurde im Komponenten Framework mit *Gluing* bezeichnet.

8.4.1 Adaption der *sniff2cdif*-Komponente

Die Komponente *sniff2cdif* wurde in der Programmiersprache C geschrieben. In dieser Fallstudie wurde die Aufgabe gestellt, dass diese Komponente nicht in einen in C geschriebenen CORBA-Server umgewandelt wird. Daher wird versucht die Komponente möglichst geschickt zu umhüllen und einer Sprache mit CORBA-Anbindung zur Verfügung zu stellen, dass eine CORBA-Komponente erzeugt werden kann.

sniff2cdif wurde in C so angepasst, dass diese Komponente von einer *Shell* gestartet werden kann. Die Methoden des *sniff2cdif*-Interfaces wurden mit Hilfe von Parametern repräsentiert. Ein *Shell*-Aufruf der umhüllten *sniff2cdif*-Komponente sieht zum Beispiel folgendermassen aus:

```
sniff2cdif -level 2 +sourceAnchor  -o ziel.cdif quell.proj
```

Mit Python wurde nun ein CORBA-Server erstellt, der dasselbe Interface wie die *sniff2cdif*-Komponente hat. Über Shell-Kommandos wird die gewrappte *sniff2cdif*-Komponente aktiviert und deren Dienste verwendet.

Die *sniff2cdif*-Komponente hätte zum Beispiel auch auf die folgenden Arten gewrapped werden können:

- SWIG (vgl. [Bea97]) ist ein Codeerzeugungswerkzeug, das C-Code einer Skript-Sprache wie Perl, Python oder Tcl durch ein Interface zur Verfügung stellen kann.
- Erweiterung einer Programmiersprache
- In C einen CORBA-Wrapper erstellen.

8.4.2 Adaption der FAMOOS-*Darstellungs*-Komponente

Die FAMOOS-*Darstellungs*-Komponente wurde in Java implementiert. Die Implementierung des *FAMOOS Information Exchange Model*s besteht aus ca. zwanzig Klassen (vgl. im Anhang C.3.2). Das *Wrapping* der FAMOOS-*Darstellungs*-Komponente in eine CORBA-Komponente erfolgt in drei Schritten. In einem ersten Schritt muss ein IDL-Interface zu dieser Komponente definiert werden und das sieht wie folgt aus:

```
// FamoosModel-Interface

module FM {
  exception Invalid_Connection{};

  // Define an Enummeration-Type (it's a set)
  interface enumString {
    boolean hasMoreElements();
    string  nextElement();        // returns a uniqueName/Propertyname
  };


  interface FamoosModelIF {
    // load and save a model represented in CDIF
    void loadVal(in string part);
    string saveVal();

    // model the model
    void createEntity(in string uniqueName, in string kind);
    void removeEntity(in string uniqueName);
    string getKind(in string uniqueName);
```

```
    ...
  };
};
```

Es wurde ein Interface *enumString* erstellt, damit kann auf Modelle mit einer Menge von Elementen zugegriffen werden. Das *interface FamoosModelIF* ermöglicht das Laden und das Abspeichern eines Modells. Die weiteren Methoden erlauben die Bearbeitung des Modells.

Im nächsten Schritt mussten zwei Klassen in Java implementiert werden, die das Serverinterface mit der FAMOOS-*Darstellungs*-Komponente verbinden. In einem dritten und letzten Schritt wurde das definierte Interface als CORBA-Server bereitgestellt.

Es gibt ORBs, die von einer bestehenden Nicht-CORBA Komponente das IDL-Interface erzeugen können. Beim VisiBroker zum Beispiel ist diese Möglichkeit unter *Caffeine* bekannt (vgl. [Vis97]). In dieser Fallstudie wird auf diese Möglichkeit verzichtet, da ein exemplarisches *Wrapping* von Hand gezeigt werden wollte.

8.5 Erweiterungen von FAMOOS

Die beiden in dieser Arbeit verwendeten FAMOOS-Komponenten werden mit einem *GUI* zu der *FAMOOS-Darstellung* und einem *TableGenerator* für die *Analyse*-Komponente erweitert.

8.5.1 *GUI* der *FAMOOS-Darstellung*

Zur Veranschaulichung der im *FAMOOS Information Exchange Model* gespeicherten Modelle wurde eine zusätzliche grafische Benutzeroberfläche (GUI) erstellt.

Die Abbildung 8.4 zeigt die in Python erstellte Benutzeroberfläche (vgl. im Anhang C.3.4). Diese grafische Benutzeroberfläche kann in drei horizontal getrennte Regionen aufgeteilt werden; es entsteht ein Kopf-, ein Mittel- und ein Fussteil.

Der Kopfteil enthält eine Menüsteuerung und diese besteht aus drei Menüs: *File*, *Connection* und *Help*. Das Menü *Connection* ermöglicht den Verbindungsaufbau zu einer Famoos-*Darstellungs*-Komponente. Die Verbindung kann auch wieder durch *Connection* abgebrochen werden, damit man sich mit einer anderen Famoos-*Darstellungs*-Komponente verbinden kann. Das Menü *File* ermöglicht das Laden, das Speichern, das Löschen sowie das Erstellen eines neuen Modells. Die eben erwähnten Möglichkeiten haben die Voraussetzung, dass eine Verbindung zu einer Famoos-*Darstellungs*-Komponente besteht. Unter dem Menü *File* kann das GUI mittels *Quit* beendet werden.

Die grafische Benutzeroberfläche erlaubt in ihrem Mittelteil das Anzeigen sowie die Eingabe von Informationen. In der Abbildung 8.4 kann man sehen, wie von einem Modell alle *Properties*[4]

[4]Ein *Property* ist eine Einheit aus dem FAMOOS-*Information Exchange Model*.

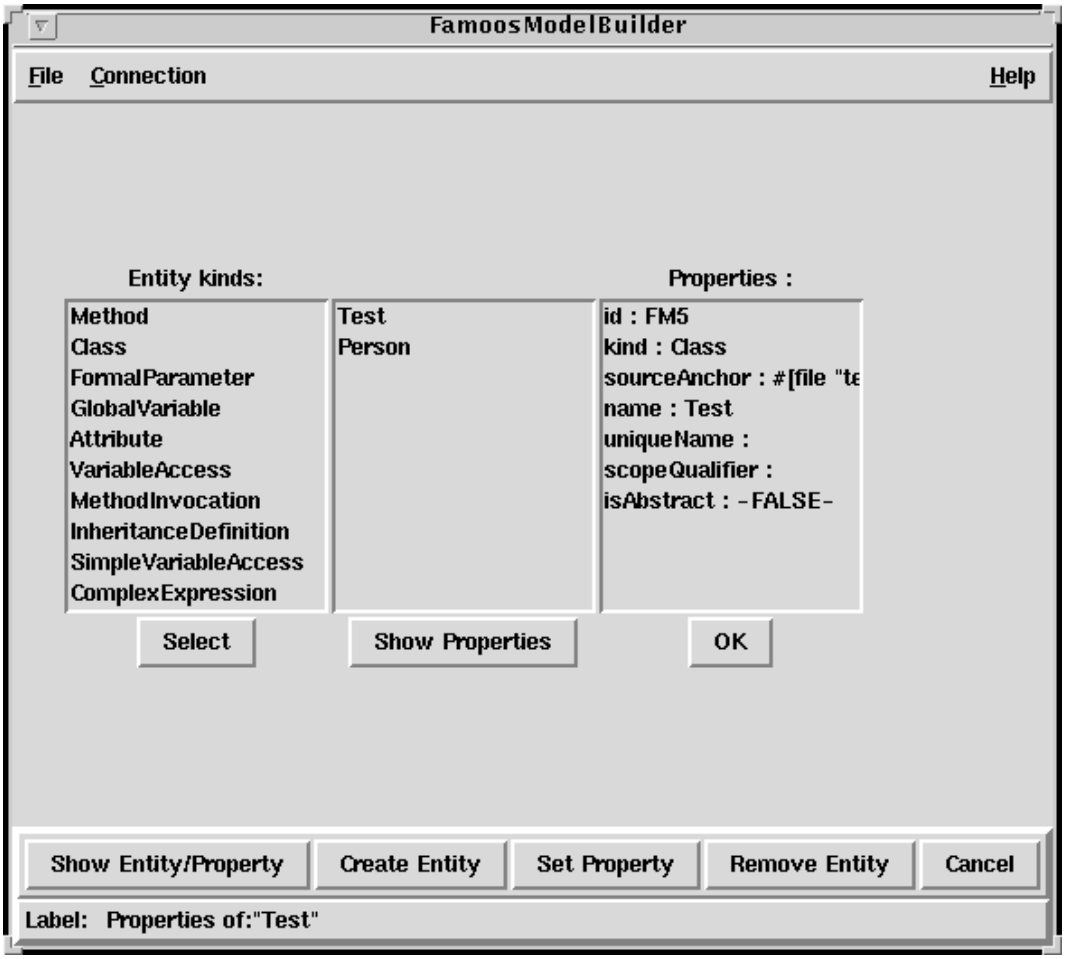

Abbildung 8.4: Grafische Benutzeroberfläche für ein FAMOOS-Modell

der Klasse `Test` (irgendein Beispiel-Modell) angezeigt werden.

Der Fussteil der grafischen Oberfläche enthält Knöpfe, damit die Informationen von einer Software in Form eines Modells angeschaut, verändert oder auch erweitert werden können. Mit dem Knopf *Create Entity* kann zum Beispiel ein Modell um eine Methode erweitert werden. *Remove Entity* erlaubt das Löschen einer Entität.

8.5.2 *TableGenerator*

Der *TableGenerator* ist eine zusätzliche CORBA-Komponente, die es erlaubt die Tabellen für die *Analyse*-Komponente von FAMOOS zu erstellen. Diese Komponente wurde erstellt, da eine neue Version des *FAMOOS Information Exchange Model* definiert wurde, und die in Smalltalk geschriebene *Metriken*-Komponente nicht mehr auf dem neusten Stand war. Das Interface des *TableGenerator* sieht wie folgt aus:

```
// TabelGenerator is aquivalent to the Analysis, FAMOOS
module Metric{
  interface MetricIF{
    void connect2FamoosModel(in string fmname);
    string Association_Table();
    string Attribute_Table();
    string Class_Table();
    string Entity_Table();
    string Inheritance_Table();
    string Method_Table();
    string MethodInvocation_Table();
  };
};
```

Der *TableGenerator* ist eine in Python geschriebene CORBA-Komponente. Das Interface wurde `MetricIF` genannt, da diese Komponente dasselbe Interface hat, wie die *Metriken*-Komponente im FAMOOS-Forschungsprojekt. Diese Komponente kann als eine Neuimplementierung einer bereits bestehenden Komponente angesehen werden, die aber nicht in die neue Architektur passt.

8.6 Verwendungsbeispiele

8.6.1 Interaktives *Debugging* von Komponenten

Mit der Komponenten-Proxy-Methode können Komponenten von Python interaktiv gesteuert werden. Diese gibt die Möglichkeit Komponenten interaktiv zu *debuggen*. So können bei der Entwicklung von Komponenten schon frühzeitig Fehler lokalisiert werden. Von dieser Möglichkeit wurde auch im FAMOOS-Forschungsprojekt Gebrauch gemacht und das sieht im Ansatz etwa wie folgt aus:

```
>>> import corba_proxy
>>> gp = corba_proxy.GeneralProxy()
GeneralProxy init
>>> famo = gp.load("FM")
create Object
>>> famo.loadRef("test.cdif")
>>> print x.hasMoreElements()
1
>>> print x.nextElement()
Person.set_name (self, arg)
```

8.6.2 Applikationslogik - zwei Beispiele

Metrische Tabelle

Das nachfolgende Skript, eine Applikationslogik, verbindet die drei FAMOOS CORBA-Komponenten. Diese Applikationslogik erstellt aus einer Projekt-Datei eine metrische Tabelle.

```
# applicationlogic: generating a metric out of a software-component
import corba_proxy
```

```
gp= corba_proxy.GeneralProxy()

sniff2cdif = gp.load("sniff2cdif")
file = open("Test.proj", "r")
proj_stream = file.read()
file.close()
cdif = sniff2cdif.readSniffSymbolTable(proj_stream, "")

famoos_model = gp.load("FM")
famoos_model.loadVal(cdif)

metric = gp.load("metric")
metric.connect2FamoosModel("FM")
association_table = metric.Association_Table()

print association_table
```

Dieser Python-Code importiert das Komponenten-Proxy-Modul `corba_proxy`, damit die Stellvertreterobjekte der drei CORBA-Komponenten erstellt werden können. Danach wird aus diesem Modul die Klasse GeneralProxy instanziert. Mit Hilfe des `gp`-Objektes werden die Stellvertreterobjekte `sniff2cdif`, `famoos_model` und `metric` erzeugt. Durch das Objekt `sniff2-cdif` wird das Software-Projekt `Test.proj` in einen CDIF-Code `cdif` umgewandelt. Das Stellvertreterobjekt `famoos_model` der FAMOOS-*Darstellung*s-Komponente lädt das Modell, das durch den cdif-Code repräsentiert wird. `metric`, der *TableGenerator*, wird erst mit einem FAMOOS-*Darstellung*s-Server verbunden und erstellt anschliessend eine metrische Tabelle von allen Assoziationen, die in diesem Modell vorkommen.

Subset-Tester

Der *Subset-Tester* ist ein Python-Skript, das testet, ob ein FAMOOS-Modell eine Untermenge von einem anderen FAMOOS-Modell ist. Dieses Skript kann in Informatik-Projekten verwendet werden, um festzustellen, ob alle Signaturen der Spezifikation entsprechen. In der Designphase werden viele Definitionen gemacht. In dieser Phase des Projektes werden hauptsächlich die Schnittstellen von Komponenten definiert. Diese Definitionen können zum Beispiel mit Hilfe des GUI der FAMOOS-*Darstellung*s-Komponente in einem FAMOOS-Modell *M1* dargestellt werden.

Für ein Projekt ist es wichtig, dass sich jeder Programmierer an die Definitionen des Interfaces hält. Mit Hilfe der FAMOOS-Komponenten kann während der Implementierungsphase aus dem vorhandenen Code ein zweites FAMOOS-Modell *M2* erstellt werden. Mit dem folgenden Skript kann überprüft werden, ob *M1* eine Teilmenge von *M2* ist. Das heisst, man kann recht schnell feststellen, ob sich jeder Programmierer an die Vorgaben gehalten hat.

```
# subset-tester: is cdif1 a subset of cdif2?
import corba_proxy, os

def is_subset(cdif1, cdif2):
  gp  = corba_proxy.GeneralProxy()
  famo1 = gp.load("FM1")         # 1. proxy
```

```
if os.path.isfile(cdif1):
  famo1.clear()
  famo1.loadRef(cdif1)         # load CDIF from a file
else:
  famo1.loadVal(cdif1)         # load CDIF-stream
famo2 = gp.load("FM2")         # 2. proxy
if os.path.isfile(cdif2):
  famo2.clear()
  famo2.loadRef(cdif2)
else:
  famo2.loadVal(cdif2)
x = famo1.enumEntities()
i = 0                          # i is an internal counter
difference = ""
# for all Elements
while x.hasMoreElements():     # repeat until there are no more elements
  i = i + 1
  print i, " ",                # counter to show that something is going on.
  uN = x.nextElement()
  bObject = famo2.isEntity(uN) # is uN of M1 in M2?
  if not(bObject):
    difference = difference + "Object: " + uN + "\n"
  y = famo1.enumProperties(uN)
  j = 0
  # for all Properties
  while y.hasMoreElements() and bObject:
    j = j + 1
    nextkind = y.nextElement()
    kindvalue = famo1.getProperty(uN, nextkind)
    if not(kindvalue == famo2.getProperty(uN, nextkind)):
      difference = difference + "Propery in :" + uN + " : " \
      + nextkind + " : " + kindvalue +"\n"

if len(difference):
  print "the difference is:\n" + difference[:-1]
else:
  print "It is a subset!"
```

`famo1` und `famo2` sind Stellvertreterobjekte der FAMOOS-*Darstellung*s-Server `FM1` und `FM2`. Mit der `loadRef`-Methode wird ein Modell geladen. In `famo1` wird Modell `M1` und in `famo2` `M2` geladen. Eine doppelte `while`-Schleife überprüft, ob sowohl alle Elemente als auch alle Merkmale (*Properties*) der Elemente von `M1` auch in `M2` vorkommen. Alle Abweichungen werden durch die Variable `difference` festgehalten und am Ende des Vergleiches ausgegeben.

8.6.3 Agenten

Im FAMOOS Forschungsprojekt könnte ein Agent, wie in Abschnitt 7.3.2 eingeführt, eingesetzt werden, um festzustellen, ob und wo sich im Netzwerk CDIF-Dateien befinden. Die Aufgabe (Skript) des *Sending-Agent* würde wie folgt aussehen:

```
# script for the sending agent
```

```
def walker(arg, dir, path):
  global result
  for val in path:
    if string.find(val, search_file) >= 0:
      result = result + machine + "::" + dir + val + "\n"


machine = posix.uname()[1]
starting_path = "/var/tmp/held/"
search_file = ".cdif"
result = ""
os.path.walk(starting_path, walker, None)
```

Das eben beschriebene Skript des *Sending-Agent* wird auf den Rechnern ausgeführt, zu denen es gesendet wird. Die Methode `walker` durchsucht alle Unterverzeichnisse von einem vorgegebenen Verzeichnis (`starting_path`) um Dateien mit dem Teilstring ".cdif" zu finden. Das `result` enthält den Namen des Rechners und den genauen Ort der gefundenen Datei.

8.6.4 Komponenten-Erweiterung mit dem *Intelligent Proxy*

In diesem Beispiel wird versucht die *TableGenerator*-Komponente um eine Methode zu erweitern. Für diese Aufgabe wird der Ansatz des *Intelligent Proxy* (vgl. Abschnitt "*Intelligent Proxy*" im Kapitel 7.3.1) verwendet. Der Code der Erweiterung sieht wie folgt aus:

```
# This is an extension of the TableGenerator-component
class Extension:
  def version(self):
    return "FAMOOS Version 1.1"

class intelligent_proxy(Extension, proxy):
  def __init__(self, subject):
    print "__init__intelligent_proxy"
    proxy.__init__(self, subject)
```

Die Klasse `intelligent_proxy` erbt von den Klassen `Extension` und `proxy`. In der Klasse `Extension` können alle Erweiterungen (Methoden, Attribute) definiert werden, die zu einem Objekt `subject` gemacht werden sollen. Die Klasse `proxy` leitet alle Methodenaufrufe an das im Argument (`subject`) übergebene Objekt weiter. Die Verwendung dieser Komponenten-Erweiterung kann etwa wie folgt aussehen:

```
>>> gp = corba_proxy.GeneralProxy()
>>> tg = gp.load("Tables")
>>> extended_tg = intelligent_proxy(tg)
>>> print extended_tg.version()
```

8.7 Diskussion

Das Forschungsprojekt FAMOOS ist ein geeignetes Fallbeispiel, um das in dieser Arbeit erstellte Komponenten Framework zu validieren. In FAMOOS existieren einige von einander unabhängige Komponenten, die sich über den Austausch von Dateien verständigen. Eine der FAMOOS-Komponenten (FAMOOS-*Darstellung*) wurde sogar zwei Mal erstellt. Zudem wurde während dieser Fallstudie eine grundlegende Veränderung am *FAMOOS Information Exchange Model* vorgenommen, so dass Komponenten zum Teil von Grund auf neu implementiert werden mussten. Das Interface dieser Komponenten bleibt hingegen unverändert.

In dieser Fallstudie wird ein exemplarisches *Gluing* der *sniff2cdif* - und der FAMOOS-*Darstellung*s-Komponente vorgenommen, damit diese zwei Komponenten im Komponenten Framework der vorliegenden Arbeit verwendet werden können. Das *Gluing* bedeutet für die zwei Komponenten, dass sie nun zu rechnerunabhängigen Servern geworden sind. Die Verständigung zwischen den Komponenten erfolgt nicht mehr durch den Austausch einer Datei, sondern direkt über Methodenaufrufe.

Die Veränderung des *FAMOOS Information Exchange Model*s zeigt sich als eine sehr interessante Situation. Durch diese Veränderung mussten die aus der Sicht des Komponenten Framework *legacy component*s neu implementiert werden. Da diese Fallstudie unabhängig von dem Forschungsprojekt gemacht wurde, entstanden bei der Neuimplementierung keine CORBA-Komponenten, sondern wieder neue Nicht-CORBA-Komponenten. Diese neuen Komponenten haben dasselbe Interface wie vorher, das heisst die *Wrapper* bleiben unverändert. Das hat den Vorteil, dass das Komponenten Framework als *Debugging*- und Test-Umgebung eingesetzt werden kann. Auf diese Weise konnten einige Fehler behoben werden.

Das GUI zu der FAMOOS-*Darstellung*-Komponente und die *TableGenerator*-Komponente erweisen sich als praktische Erweiterungen. Mit der grafischen Benutzeroberfläche ist es sehr einfach Modelle anzuschauen, Modelle zu erweitern, zu verändern oder neu zu erstellen. Der *TableGenerator* kann als Neuimplementierung der in Smalltalk geschriebenen und durch die Veränderung veralteten *Metriken*-Komponente angesehen werden. Dieser in Python implementierte *TableGenerator* erzeugt Tabellen für die *Analyse*-Komponente.

Kapitel 9

Schlussbemerkungen

9.1 Erreichte Ziele

In dieser Arbeit wurden die Themengebiete *Scripting* und CORBA behandelt. Die in Abschnitt 1.2 gesteckten Ziele konnten weitgehend erreicht werden. Aus zeitlichen Gründen wurden einige Themen etwas weniger intensiv als andere behandelt. Dies darf für das jeweilige Thema jedoch nicht wertend betrachtet werden.

Aus einer eingehenden Studie ausgewählter Literatur wurden die Grundlagen für diese Arbeit erarbeitet (vgl. Kapitel 2 bis 4). Daraus resultierte im Kapitel 5 eine Problemanalyse von CORBA.

Basierend auf der in dieser Arbeit entwickelten Komponenten-Proxy-Methode konnte gezeigt werden, wie CORBA-Komponenten in einem Komponenten Framework verwendet werden können. Die Methode der Komponenten-Proxy ist ein ORB-unabhängiger Ansatz, der an keinen spezifischen ORB gebunden ist und vom Ansatz her kann auch in anderen Middleware-Technologien eingesetzt werden. Mit Hilfe dieser Methode können CORBA-Komponenten verwendet werden, ohne dass CORBA-Kenntnisse nötig sind (vgl. Abschnitt 7.1).

Die Methode der Komponenten-Proxy basiert auf statischen CORBA-Methodenaufrufen und bringt die folgenden Verbesserungen von CORBA:

- Flexibilität: Durch das Komponenten Framework können Komponenten wie Bausteine zusammengesetzt (*connect*) werden und werden dadurch flexibler (vgl. Abschnitt 7.2).

- Softwarewiederverwendung: Komponenten können vorallem aus zwei Gründen leicht wiederverwendet werden. Einerseits fördert das Komponenten Framework die Entwicklung von kleinen, unabhängigen Komponenten und andererseits können *legacy component*s mit dem *Glue* in das Komponenten Framework integriert werden (vgl. Absatz 7.2.3).

- Erweiterbarkeit: Ein bestehendes System kann mit Hilfe des *intelligent_proxy* leichter erweitert oder verändert werden (vgl. Absatz 7.3.1).

- Testen: CORBA-Komponenten haben durch den Einsatz einer Skript-Sprache gute Testmöglichkeiten. Interaktives Debugging ist mit Hilfe der Skript-Sprache schon während der Entwicklung möglich (vgl. Absatz 7.2.2).
- Wartbarkeit: Das Komponenten Framework kann zur Entwicklung von Software führen, die besser gewartet werden kann (vgl. Absatz 7.2.1).
- Abstraktionsniveau: Durch den Einsatz des Komponenten Frameworks werden vor allem kleine flexible Komponenten entwickelt, die zum Zusammenbauen von Applikationen mit Skripts verwendet werden. Durch diese Komponenten Architektur (Komponenten und Skripts) erlangen CORBA-Applikationen ein höheres Abstraktionsniveau.

In der Fallstudie FAMOOS (vgl. Kapitel 8) wurde die Komponenten-Proxy-Methode validiert. Anhand dieser Validierung konnte festgestellt werden, dass sich die entwickelte Methode für den vorgesehenen Verwendungszweck eignet. In unterschiedlichen Programmiersprachen implementierte Komponenten wurden mit Hilfe von *Wrapping*-Techniken zu CORBA-Komponenten umgewandelt. Die adaptierten Komponenten wurden durch das Komponenten Framework verwendet. Es hat sich durch die Verwendung gezeigt, dass das noch recht unerforschte *Wrapping* (vgl. [Sne96]) der Komponenten eine ziemlich schwierige Aufgabe ist. Dieser Aufgabe kann aber dank dem *Glue* des Komponenten Framework eine weitere Alternative zur Verfügung gestellt werden.

Im Anhang C hat es eine Übersicht zu der in dieser Arbeit erstellten Software. Die Softwareanforderungen für die Verwendung sowie der Ablageort der CORBA-Komponenten und der Komponenten-Proxy-Methode wird ebenfalls im Anhang C beschrieben.

9.2 Offene Probleme

Aus zeitlichen Gründen konnten die folgenden Fragen nur am Rande oder gar nicht behandelt werden. Sie können aber Ziele für weiterführende Arbeiten sein:

- Welche verschiedenen *Gluing*-Techniken (*Wrapping*) gibt es?
- Wie sieht das Zeitverhalten der in dieser Arbeit erstellten Methode im Vergleich zu anderen Ansätzen aus?
- Ist der Overhead, der durch die Stellvertreterobjekte entsteht, relevant?
- Wird es mit dem neuen CORBA 3.0 Standard möglich sein, die Methode der Komponenten-Proxy auch mit dynamischen CORBA-Methodenaufrufen zu realisieren?
- Wie können CORBA-Komponenten geeignet typisiert werden, um einen optimalen Einsatz der Komponenten in einem Komponenten Framework zu gewährleisten?
- Kann eine Qualität der Applikationen garantiert werden?
- Wie kann die Last in einer solch verteilten Applikation verteilt werden? Was für Mechanismen können verwendet werden?

- Durch den Erweiterungsansatz *intelligent-proxy* eröffnen sich diverse Anwendungsgebiete. Wie wird zum Beispiel ein Analyse-Tool für CORBA erstellt?
- Eignet sich der in Abschnitt 7.3.2 erstellte Ansatz von Agent für die Aufgaben eines Systemadministrators? Was gibt es dabei für Sicherheitsprobleme? Können selbständige Agenten im Internet erzeugt werden?

Anhang A

Arbeitsumgebung

Hardware

Als Arbeitsumgebung wurden `SUN-Workstations`[1] der Fachgruppe SCG verwendet. Die Ausstattung der Rechner ist relativ einheitlich. Der in dieser Arbeit am meisten verwendete Rechner (Lursa) ist eine SPARCstation-4 (SUN) mit 32 MB RAM.

Betriebssystem

Auf den SUN-Workstations ist das Sun `Solaris 2.5.1`[2] (UNIX) installiert. UNIX ist ein Multitasking- und Multiuser-Betriebssystem.

Software

Für die vorliegende Arbeit wurden folgende Hilfsmittel verwendet:

- Programmiersprachen:
  Die objektorientierte Skript-Sprache `Python` wurde für das Komponenten Framework verwendet. Für die Komponenten-Erstellung wurde die objektorientierte, robuste und interpretierte Programmiersprache `Java` eingesetzt.
- Middleware:
  Der `VisiBroker` für Java von Visigenic und der `Fnorb`[3] für Python sind die verwendeten CORBA 2.0 ORBs.
- Editor:
  Als Editor wurde der `xemacs` (Version 19.14) verwendet und die Arbeit wurde mit LaTeX (teTeX) geschrieben.
- Grafiken:
  Einerseits wurden die meisten Abbildungen mit dem Unix-Tool `xfig` erstellt. `xfig` ist ein

[1] Mehrzahl, da die Client/Server-Applikationen meist auf verschiedenen Rechnern ausgeführt wurden.
[2] Solaris 2.5 oder höher gilt als Voraussetzung für den VisiBroker 3.0!
[3] Der Fnorb ist ab der Version 0.7 ganz unabhängig.

Menü-gesteuertes Grafik-Tool, um Zeichnungen unter dem X Window System zu erstellen. Andererseits wurden GUIs von Applikationen mit dem Hilfsmittel `snapshot` in einer Abbildung festgehalten.

Anhang B

Technische Details von CORBA

B.1 Ausnahmen (Exception) beim VisiBroker

Die CORBA-Ausnahmen werden aus den Java-Ausnahmen (java.lang.Exception) abgeleitet. Die Abbildung B.1 aus [VD97] zeigt die Zusammenhänge zwischen den Java- und den CORBA-Ausnahmen. Die CORBA-*Exceptions* bestehen aus den *System-Exceptions* und den benutzerdefinierten Exceptions. Die System-Ausnahmen enthalten Deklarationen von 26 verschiedenen Standard-Ausnahmen (Pre-defined SystemExceptions) für Ausnahmen von Netzwerk, ORB und System-Fehlern. Alle *Pre-defined System Exceptions* können kollektiv als *CORBA.SystemException* abgefangen werden. Eine Methode erzeugt nur dann benutzerdefinierte Ausnahmen, wenn diese auch in der Signatur definiert sind.

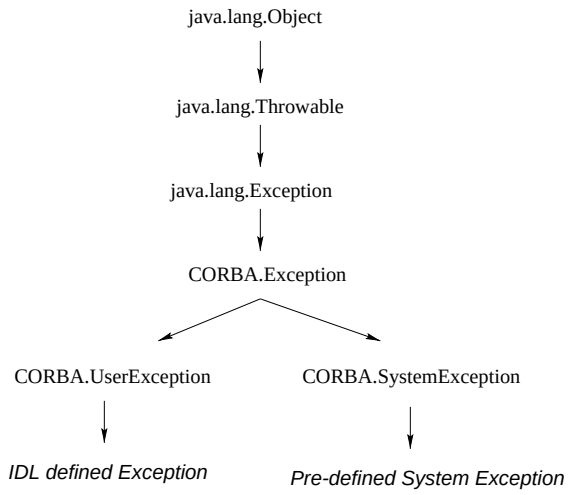

Abbildung B.1: Ausnahmebehandlung im VisiBroker für Java

B.2 Das *low level*-Verhalten eines ORBs

Die Kommunikation zwischen verteilten Client/Server-Applikationen wird vom CORBA ORB übernommen, das heisst, die Entwickler brauchen nichts von Sockets, Ports oder Protokollen zu wissen, da diese Arbeit vom ORB übernommen wird. Daraus ergibt sich der Nachteil: Der Entwickler weiss nicht genau, wie nun die Kommunikation abläuft.

Daher ist es von Interesse, wie der ORB diese Kommunikation realisiert. Als erstes wurde das Handbuch des VisiBrokers (vgl. [Vis97]) konsultiert. Anschliessend wurde anhand eines Experimentes der *low level*-Ablauf der Kommunikation eruiert.

Das VisiBroker *Reference Manual* besagt:

1) Der Client macht einen UDP-Broadcast um den *osagent* zu finden.
2) Der *osagent* teilt dem Client mit, wo er den Server finden kann.
3) Ein TCP-Verbindung wird zwischen Client und Server aufgebaut.

Das *snoop*-Experiment

Um die Kommunikationsabläufe herauszufinden, wurde zuerst ein UNIX-Prozess *snoop*[1] gestartet. *snoop* ist ein Prozess, der die Aktivität auf dem Netzwerk überwacht. Danach wurden *osagent*, Server und Client nacheinander auf drei verschiedenen Rechnern gestartet. Während dieser Zeit wurden die einzelnen Datenpackete mit Hilfe von *snoop* in ein Logfile geschrieben. Die Auswertung der Logdatei ergab die folgenden Resultate:

1) Der Client macht wirklich einen UDP-*broadcast.*
2) Der Server startet danach eine TCP-Verbindung mit dem Client.

Die Kontaktaufnahme zum *osagent* konnte nicht festgestellt werden, das liegt wahrscheinlich an der Tatsache, dass das eine interne Optimierung vom VisiBroker ist.

B.3 IOR/IIOP - Interoperabilität

Die Kommunikation zwischen zwei verschiedenen ORBs nennt man Interoperabilität und verläuft über IOR/IIOP. Die IOR ist die Objektreferenz und IIOP das Protokoll, welches die Interoperabilität von CORBA ermöglicht.

IOR (Interoperable Object Reference)

Die IOR ist der Standard für die Objekt Referenzen in CORBA. Dank der IOR ist eine Inter-ORB-Kommunikation möglich. Wird eine IOR in einen String umgewandelt, so sieht sie wie folgt aus:

[1] Captures packets from the network and displays their contents (UNIX manual).

```
IOR:000000000000001449444c3a54696d652f54696d6549463a312e3000000000020000000000000
00460001000000000000d3133302e39322e36352e33300000c6340000002a00504d43000000000000
001449444c3a54696d652f54696d6549463a312e300000000006436c6f636b000000000000010000
00240000000000000001000000010000001400000000000000000000000000010109000000000000
00000000000000000000000001010900000000
```

In diesem Beispiel ist der IOR-String 358 Zeichen (Byte) lang, das kann aber von Objekt zu Objekt variieren. In dieser sogenannten *stringified* IOR sind unter anderem Adresse und Port des Server-Objektes enthalten.

IIOP (Internet Inter-ORB Protocol)

Für die Interoperabilität spezifizierte die OMG GIOP (General Inter-ORB Protocol). GIOP definiert die Übertragungstypen für die ORB-zu-ORB-Kommunikation. Das GIOP ist nicht dasselbe wie ein RPC und baut auf einem ORB auf (vgl. [Sie96]). Das GIOP erfüllt die folgenden Anforderungen:

- Verbindungsorientiertes Protokoll
- Zuverlässige Übertragung
- Meldung bei Verbindungsunterbruch
- Modell für die Initialisierung einer Verbindung

Das IIOP ist eine Definition der GIOP Spezifikation (vgl. Abbildung B.2 aus [Sie96]). Das Protokoll IIOP basiert auf TCP/IP-Verbindungen. IIOP definiert eine Menge von Regeln für das Formatieren von Daten genannt CDR. CDR (Common Data Representation) definiert alle IDL Datentypen, *type code* und zusammengesetzte Typen für die Datenrepräsentation. CDR wird zudem für das *Bridging* benötigt. Neben den CDR Regeln werden im IIOP einige Übertragungsarten festgesetzt.

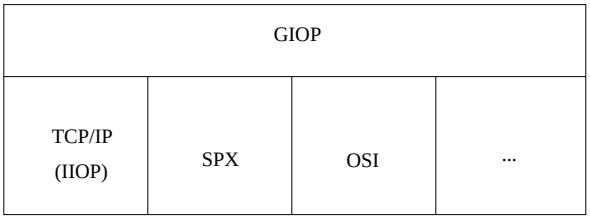

Abbildung B.2: Interoperabilität mit IIOP

B.4 CORBA 3.0 - der neue Standard

Die meisten heute verwendeten ORBs unterstützen den CORBA 2.0 Standard. Doch der Standard wird laufend weiter entwickelt. Die Tabelle B.1 zeigt die zeitliche Abfolge der einzelnen Versionen von CORBA.

Standard	Zeit
CORBA 1.0	Oktober 1991
CORBA 1.1	Februar 1992
CORBA 1.2	Dezember 1993
CORBA 2.0	August 1996
CORBA 2.1	August 1997
CORBA 2.2	Februar 1998
CORBA 2.3	Spätsommer 1998
CORBA 3.0	Mitte 1999

Tabelle B.1: Überblick der CORBA Spezifikationen

In einem *Pre-release* wurden die zu erwartenden Erneuerungen des CORBA 3.0 Standards veröffentlicht (vgl. [OMG98]). Nachfolgend sind die wichtigsten Neuerungen dieses Standards aufgelistet:

- *Multiple Interface* erlaubt das Erstellen von Mehrfach-Interfaces von einem Objekt. Das hat den Vorteil, dass so die Sichtbarkeit sowie die Zugriffskontrolle geregelt werden können.
- Eine Methode wird die Unterstützung von *Objects-by-Value* anbieten. *Objects-by-Value* soll schneller und intuitiver sein.
- Die OMG definiert mit CORBA 3.0 ein Framework für wiederverwendbare Objekte in einer Komponenten Architektur. In dieser Spezifikation sind auch JavaBeans enthalten.
- CORBA 3.0 wird eine Spezifikation einer Skript-Sprache enthalten. Diese Skript-Sprache soll zum Zusammensetzen (compose) von CORBA-Komponenten zu einer Applikation verwendet werden.
- Spezifikationen wie *Realtime*-CORBA 1.0 und *Asynchronous Messaging* werden im Bereich der Service Qualität eingesetzt. Die *Event*- und *Life Cycle Service*s werden in CORBA-Objekten eingeschlossen, das heisst, Clients können dynamisch auf Methoden, Fähigkeiten und *Events* der Objekte zugreifen.
- CORBA 3.0 definiert ein Interface für Firewalls, damit können IIOP kontrolliert kommunizieren.

B.5 Fnorb für Python

Der Fnorb (vgl. [Chi98]) ist ein noch experimenteller CORBA 2.0 ORB, der fast vollständig in Python implementiert[2] wurde. Der Fnorb wurde als ein Teilprojekt von Hector[3] am *Cooperative*

[2]Das hatte den grossen Vorteil, dass eine Fehlerkorrektur (Bug fix) eigenhändig durchgeführt werden konnte. Die Klassen des ORBs mussten weder neu compiliert noch gelinkt werden, der ORB war sofort wieder benutzbar.

[3]vgl. $http://www.dstc.edu.au/Hector/$

Research Centre für *Distributed Systems Technology* an der Universität von Queensland in Brisbane (Australien) entwickelt. Die IDL nach Python Anbindung ist leider noch nicht standardisiert (vgl. [CJvL98]) und befindet sich in Bearbeitung (vgl. nächster Abschnitt).

B.6 Skript-Sprachen-Anbindung an CORBA

Die OMG hat das Fehlen einer Skript-Sprache im CORBA 2.0 Standard realisiert und hat deshalb ein RFP (Request for Proposal) veröffentlicht (vgl. [OMG97a]). Im Moment ist der Status: *Revised submissions have been received.* Über die aktuelle Situation kann man sich unter der folgenden Adresse informieren:
$www.omg.org/techprocess/meetings/schedule/CORBA_Scripting_Language_RFP.html$
Die Liste der *Revised Submission* enthält folgende Skript-Sprachen:

- CorbaScript (vgl. $http://corbaweb.lifl.fr/CorbaScript/$)
- Tcl
- JavaScript (von Netscape)
- Python

Anhang C

CORBA Komponenten-Bibliothek

Dieses Kapitel gibt eine Übersicht der in dieser Arbeit erstellten Komponentenbeispielen, die mit Hilfe der Komponenten-Proxy-Methode verwendet werden können. Die erstellten Implementierungen befinden sich noch in einem experimentellen Stadium, das heisst, es existiert wohl ein lauffähiges System (Prototyp), das aber absichtlich einfach gehalten wurde.

In diesem Kapitel werden keine technischen Anweisungen zur Verwendung der Komponenten gegeben, da es beim Sourcecode jeweils noch eine *README*-Datei hat. Sämtlicher Code der nachfolgenden Komponenten ist unter der nachfolgenden Adresse verfügbar:

lokal: $/home/scg/held/KomponentenBeispiele$ oder

$http://www.iam.unibe.ch/\sim held/KomponentenBeispiele$

Um die vorliegenden Komponentenbeispiele zu verwenden, müssen die nachfolgenden Systemanforderungen der Software erfüllt sein:

- Der `Fnorb` und `Python` müssen installiert sein, um die in dieser Arbeit implementierte Komponenten-Proxy-Methode zu verwenden.

- Der `VisiBroker` für Java und `Java` sollten auf dem System vorhanden sein, um die CORBA-Komponenten zu benutzen. Es kann aber auch sonst ein ORB mit `eigenen Komponenten` verwendet werden. Achtung: Eigene Komponenten müssen sich auch beim *NameServer* anmelden, so wie es im Abschnitt 7.1 beschrieben ist.

Nachfolgend werden die Elemente der Komponenten-Proxy-Methode beschrieben. Danach werden die CORBA-Komponenten sowie die FAMOOS-Komponenten vorgestellt, die mit der Komponenten-Proxy-Methode verwendet werden können. Im letzten Abschnitt gibt es zwei weitere CORBA-Beispiele.

C.1 Komponenten-Proxy-Methode

Dieser Abschnitt enthält kurze Beschreibungen zu der in Kapitel 7 aufgestellten Methode “Komponenten-Proxy-Methode”.

C.1.1 *corba_proxy* - Modul

Das Modul *corba_proxy* wurde in Python geschrieben und besteht aus der Klasse `GeneralProxy`. Der Zweck dieses Moduls ist das automatische Erstellen eines Stellvertreter-Objektes (Komponenten-Proxy) von einem CORBA-Server. Das Modul ermöglicht Python den Zugriff auf CORBA-Komponenten mittels eines Stellvertreter-Objekts. Die Kommunikation verläuft zwischen Stellvertreter und CORBA-Komponente über CORBA ORBs. Das Modul *corba_proxy* wird wie folgt verwendet:

- Modul *corba_proxy* importieren.
- Klasse `GeneralProxy` instanzieren.
- CORBA-Objekte können mit der Methode `load` geladen werden.

C.1.2 NameServer

Der globale `NameServer` ist eine CORBA-Komponente, die für die Komponenten-Proxy-Methode erstellt wurde. Die Aufgaben des globalen `NameServer`s lassen sich in zwei Bereiche unterteilen. Einerseits können sich CORBA-Server beim *NameServer* registrieren, damit potentielle Benutzer sie finden. Andererseits können CORBA-Server mit Hilfe dieses Dienstes ORB-unabhängig durch einen Namen referenziert werden. Das Modul `corba_proxy` benötigt Informationen über die Server, um einen Komponenten-Proxy zu erstellen. Das IDL-Interface sieht wie folgt aus:

```
// NameServer.idl
module NameServer{
  exception UNKNOWN_SERVER{};

  interface NameServerIF{
    short number_available_servers();
    string get_Server_Name(in short index) raises (UNKNOWN_SERVER);
    string get_Stub_Name(in string name);
    string get_IOR(in string name);
    string get_IDL(in string name);
    boolean is_avail(in string name);

    void add_Server(in string name, in string IOR, in string Stub_Name,
                    in string IDL);
    void remove_Server(in string name) raises (UNKNOWN_SERVER);
  };
};
```

C.1.3 *Intelligent Proxy*

Der *Intelligent Proxy* ist ein zusätzlicher Ansatz mit dem Stellvertreter-Objekte flexibler gemacht werden können. Dieser Ansatz ermöglicht den Stellvertreter-Objekten vor allem drei neue Möglichkeiten: Eine CORBA-Komponente kann erstens um zusätzliche Fähigkeiten erweitert werden, wie zum Beispiel unterschiedliche Zugriffsrechte oder lokale Informationen (Geschwindigkeit). Zweitens können Interfaces aus Sicherheitsüberlegungen eingeschränkt werden. Drittens bietet dieser Ansatz die Voraussetzung für ein Analyse-Tool (vgl. Absatz 7.3.1).

C.1.4 *Mobile Agent*

Mit Hilfe der Komponenten-Proxy-Methode kann ein *Mobile Agent* erstellt werden. Ein *Mobile Agent* kann in einem Rechnernetz unabhängige Aufgaben ausführen. In dieser Arbeit wurden ein *Travelling Agent* und ein *Sending Agent* implementiert (vgl. Absatz 7.3.2).

C.2 CORBA-Komponenten

Die Komponenten in diesem Abschnitt können nur gestartet werden, wenn auch der *NameServer* läuft, da sie sich automatisch bei ihm anmelden. Diese Annahme wurde zur Vereinfachung getroffen.

Die nachfolgenden CORBA-Komponenten sind in `Java` implementiert worden. Beim Starten dieser CORBA-Server kann durch einen optionalen Parameter explizit der Name dieser Komponente angegeben werden (zum Beispiel mit: `java Hello myHello`).

C.2.1 Hallo, Welt!

Die `Hallo, Welt!`-Komponente ist ein CORBA-Server, der nur eine Methode hat. Diese Methode liefert als Rückgabewert “Hello, World!”. Der Server hat folgendes Interface:

```
// HelloWorld.idl; Hello World Interface
module HelloWorld{
  interface Discussion{
    string say_hello();
  };
};
```

C.2.2 Counter-Komponente

Der `Counter` (Zähler) ist ein CORBA-Server, der aus einem Attribut `sum` und einer Methode `increment` besteht. Das Interface sieht wie folgt aus:

```
// Count.idl
module Counter {
  interface Count {
    attribute long sum;
    long increment();
```

```
  };
};
```

C.2.3 *Pricing*

Der `Pricing`-Server wurde im Abschnitt 4.4 ausführlich besprochen.

C.2.4 Das *ZeitServer*-Beispiel

Das `ZeitServer`-Beispiel besteht aus zwei CORBA-Komponenten. Die erste Komponente `Time` erzeugt die aktuelle Zeit, die über die Methode `get_hour` und `get_minute` abgefragt werden kann. Hier ist noch zu erwähnen, dass diese aktuelle Zeit leider nicht mit der lokalen Zeit übereinstimmt (falsche Zeitzone; Java-Fehler in JDK 1.1.6). Der `Time`-Server hat das folgende Interface:

```
// Clock.idl
module Time{
  interface TimeIF{
    short get_hour();
    short get_minute();
  };
};
```

Die zweite Komponente `Display` ist ein Output-Server. Mit der Methode `display_time` kann die Zeit in Form eines Strings ausgegeben werden. Die `Display`-Komponente hat folgende Struktur:

```
// Display.idl
module Display{
  interface DisplayIF{
    void display_time(in string time);
  };
};
```

C.3 FAMOOS

Dieser Abschnitt enthält die Komponenten, die in der Fallstudie (vgl. Kapitel 8) erstellt und verwendet wurden.

C.3.1 *sniff2cdif*-Komponente

sniff2cdif ist eine in C geschriebene Komponente, die aus einem von SNiFF++ gespeicherten Software-Projekt eine CDIF-Repräsentation erstellt. Die *sniff2cdif*-Komponente wurde in einen CORBA-Server umgewandelt. Diese Umwandlung wurde durch eine Adaption realisiert. Das Interface der adaptierten *sniff2cdif*-Komponente sieht wie folgt aus:

```
// sniff_API.idl
module sniff_API{
  interface sniff_APIIF{
```

```
    void sniffAPIisAvailable();
    void setLevel(in short level);
    short getLevel();
    void showVariableAccess();
    void hideVariableAccess();
    void showMethodInvocation();
    void hideMethodInvocation();

    void showSourceAnchor();
    void hideSourceAnchor();
    void showFullInheritanceDefinition();
    void hideFullInheritanceDefinition();
    string readSniffSymbolTable(in string proj, in string output_file);
  };
};
```

C.3.2 FAMOOS-Darstellung

Die FAMOOS-Darstellungs-Komponente wurde in einer Diplomarbeit (vgl. [Fra99]) erstellt. In der vorliegenden Arbeit wurde ein Wrapper um diese Komponente gelegt und zu einem CORBA-Server umgewandelt. Das Wrapping erfolgte in drei Schritten:

- Definition des Interface (vgl. FamoosModel.idl)
- Interface mit der bestehenden Komponente verbinden
- CORBA-Server bereitstellen

```
// FamoosModel.idl
module FM {
  exception Invalid_Connection{};

  interface enumString {
    boolean hasMoreElements();
    string  nextElement();       //returns uniqueName/Propertyname
  };

  interface FamoosModelIF {
    void loadRef(in string href);
    void loadVal(in string part);
    void saveRef(in string href);
    string saveVal();
    void createEntity(in string uniqueName, in string kind);
    void removeEntity(in string uniqueName);
    enumString enumEntities();   // List of uniqueNames
    string getKind(in string uniqueName);
    string getId(in string uniqueName);  //CDIFid
    void setProperty(in string uniqueName, in string property, in string value);
    // getProperty: value of Property
    string getProperty(in string uniqueName, in string property);
    // enumString: List of all Propertynames of an uniqueNameEntiety
    enumString enumProperties(in string uniqueName);
    void connection_tester() raises (Invalid_Connection);
```

```
    void clear();
  };
};
```

C.3.3 *TableGenerator*

Der *TableGenerator* ist ein CORBA-Server, der in Python implementiert wurde. Dieser Server muss immer mit einer FAMOOS-Darstellungs-Komponente verbunden werden (connect2Famoos-Model). Der *TableGenerator* ermöglicht es, aus einem Software-Modell Tabellen für metrische Analysen zu erstellen. Nachfolgend ist das Interface des *TableGenerator* zu sehen:

```
// Metric.idl; Tabelgenerator for the Metrics
module Metric{
  interface MetricIF{
    void connect2FamoosModel(in string fmname);
    string Association_Table();
    string Attribute_Table();
    string Class_Table();
    string Entity_Table();
    string Inheritance_Table();
    string Method_Table();
    string MethodInvocation_Table();
  };
};
```

C.3.4 Skripts

ModelBuilder

Der *ModelBuilder* ist eine in Python geschriebene grafische Benutzeroberfläche und wurde zur Visualisierung der Famoos-Darstellungs-Komponente erstellt. Mit dem *ModelBuilder* kann ein im *FAMOOS information Exchange Model* gespeichertes Software-Modell angezeigt, erweitert und gelöscht werden. Es können aber auch Modelle von Grund auf neu erstellt werden.

Subset-Tester

Der *Subset*-Tester ist ein Python-Skript, das mit Hilfe von zwei *FAMOOS-Darstellungs*-Komponenten arbeitet. Das Skript testet, ob ein Software-Modell eine Teilmenge eines anderen Software-Modelles ist.

Metrische Tabelle

Das Skript der metrischen Tabelle ist eine einfache Applikationslogik, die drei Stellvertreter-Objekte miteinander verbindet. Das Skript erstellt aus einer Software-Projekt-Datei eine metrische Tabelle, die dann für weitere Analysen verwendet werden kann.

Finde CDIF-Dateien

Die Aufgabe eines in dieser Arbeit eingeführten *Mobile Agent* ist ein Skript. Ein Skript (Aufgabe) kommt jeweils auf einem Rechner zur Ausführung und übergibt anschliessend dem Agenten die Resultate. Das Skript “Finde CDIF-Dateien” wird eingesetzt, um herauszufinden, ob und wo sich im Netzwerk CDIF-Dateien befinden.

C.4 Weitere CORBA-Beispiele

C.4.1 Komplexe Zahlen

Das Beispiel der `komplexen Zahlen` ist eine CORBA-Applikation, die in einer frühen Phase dieser Arbeit erstellt wurde. Es handelt sich dabei um ein Beispiel einer CORBA-Applikation mit einer auf verschiedene Komponenten verteilten Applikationslogik. Komponenten sind genau aufeinander abgestimmt. `Komplexe Zahlen` besteht aus drei Bestandteilen:

- Ein CORBA-Client `KeyBoard` erlaubt die Eingabe von komplexen Zahlen über ein Applet (vgl. Abbildung C.1).
- Ein CORBA-Server `Math` übernimmt die Berechnung der komplexen Operationen. Die Komponente `Math` ist aber zugleich auch wieder Client vom `ShowServer`.
- Der CORBA-Server `ShowServer` ist für die grafische Darstellung des Resultats verantwortlich (vgl. Abbildung C.2).

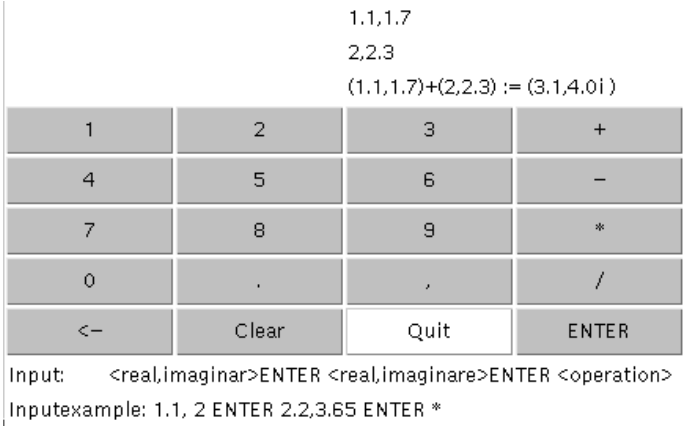

Abbildung C.1: Eingabe Applet vom Komplex Rechner

Anhand dieser Applikation kann gezeigt werden, dass die Komponenten einer CORBA-Applikation nicht unbedingt einfach und wiederverwendbar sein müssen. Das in dieser Arbeit erstellte Komponenten Framework hilft dem Entwickler einfachere Applikationen zu erstellen.

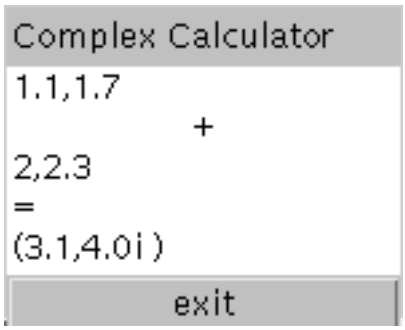

Abbildung C.2: Grafische Darstellung der Resultate des Komplex Rechners

C.4.2 DII

DII ist eine CORBA-Applikation, die dynamische Methodenaufrufe macht. In dieser Applikation werden weder Stub noch Skeleton verwendet. Der Client erzeugt einen *Request*, der dann dynamisch ausgewertet wird. Diese Beispiel-Applikation kann für weiterführende Arbeiten sehr wichtig sein.

Literaturverzeichnis

[Bea97] David M. Beazley. *SWIG Users Manual.* Department of Computer Science University of Utah, June 1997.

[Ber96] Philippe A. Bernstein. Middleware: A Model for Distributed System Services. *Communications of the ACM*, February 1996.

[BR97] Walter Bischofberger and Dirk Riehle. Löst CORBA wirklich alle unsere Probleme? *OBJEKTspektrum, SIGS Publications*, July 1997.

[CDI98] CDIF, http://www.cdif.org/. *CDIF standards* , 1998.

[Chi98] Martin Chilvers. *Fnorb 0.7.1, Reference Manual.* CRC for Distributed Systems Technology, University of Queensland, Brisbane (Australia), http://www.dstc.edu.au/AU/staff/martin-chilvers/Fnorb/, June 1998.

[CJvL98] Martin Chilvers, Bill Janssen, and Martin von Löwis. Python Language Mapping. http://www.python.org/sigs/do-sig/corbamap.html, February 1998.

[DT98] Serge Demeyer and Sander Tichelaar. Definition of the FAMOOS Information Exchange Model - Version 1.1. Technical report, University of Bern, SCG, http://www.iam.unibe.ch/∼famoos/InfoExchFormat/, July 1998.

[EFA96] Wolfgang Emmerich, Fabrizio Ferrandina, and Jim Arlow. Die CORBAServices der OMG - ein Überblick. *OBJEKTspektrum, SIGS Publications*, 6, 1996.

[EKM97] Bernd Eichner, David Kamber, and Stephan Murer. CORBA: Principles and practical experiences. *Informatik/Informatique - Zeitschrift der schweizerischen Informatikerganisation*, April 1997.

[FG96] Stan Franklin and Art Graesser. Is it an Agent, or just a Programm? A Taxonomy for Autonomous Agents. Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages; Springer-Verlag, Institute for Intelligent Systems University of Memphis, 1996.

[Fra99] Fredi Frank. The Associative Documentation Model. Master's thesis, University of Bern, SCG, 1999.

[GAO95] David Garlan, Robert Allen, and John Ockerbloom. Architectural Mismatch: Why Reuse Is So Hard. *IEEE Software*, 12(6):17–26, November 1995.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[Gri98] Frank Griffel. *Componentware: Konzepte und Techniken eines Softwareparadigmas.* dpunkt.verlag, 1998.

[GS96] David Garlan and Mary Shaw. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, April 1996.

[INOS97] IBM, Netscape, Oracle, and Sunsoft. CORBA Component Imperatives. Technical report, OMG, ftp://www.omg.org/pub/docs/orbos/97-05-25.ps, May 1997.

[Jen97] Bruno Jenny. *Projektmanagement in der Wirtschaftsinformatik.* vdf Hochschulverlag AG an der ETH Zürich, 1997.

[JM98] Simon Peyton Jones and Erik Meijer. Scripting COM components in Haskell. Technical report, University of Amsterdam, June 1998.

[JS97] Bill Jansson and Mike Spreizer. *Inter-Language Unification 2.0 Reference Manual.* Xerox Corporation, ftp://ftp.parc.xerox.com/pub/ilu/ilu.html, May 1997.

[LS97] Cameron Laird and Kathryn Soraiz. Choosing a scripting language. *SunWorld*, October 1997.

[LS98a] Cameron Laird and Kathryn Soraiz. Getting started with Python. *SunWorld*, February 1998.

[LS98b] Cameron Laird and Kathryn Soraiz. Scripts im Griff. *BYTE Deutschland*, July 1998.

[Lut96] Mark Lutz. *Programming Python: Object-Oriented Scripting.* O'Reilly & Associates, October 1996.

[Maf97] Silvano Maffeis. Wenn CORBA-Objekte abstürzen. In *Object Expo, Tutoriumsbeilagen*, October 1997.

[MGRG98] Philippe Merle, Christophe Gransart, Jean-François Roos, and Jean-Marc Geib. CorbaScript: A Dedicated CORBA Scripting Language. Technical report, Laboratoire d'Informatique Fondamentale de Lille, France, September 1998.

[Mic98] Microsoft, http://www.microsoft.com/com/dcom.asp. *Distributed Component Object Model (DCOM), Specification*, January 1998.

[Mil96] Damien Miller. Three types of intellignet agents, their uses, and their future. *NetscapeWorld*, November 1996.

[MK98] Julia Miller and Alexander Kotopoulis. Überwachung und Diagnose von verteilten CORBA-Anwendungen. *OBJEKTspektrum, SIGS Publications,* November/Dezember 1998.

[MM97] Thomas J. Mowbray and Raphael C. Malveau. *CORBA Design Patterns.* Wiley, 1997.

[MS97] Silvano Maffeis and Douglas C. Schmidt. Constructing Reliable Distributed Communication Systems with CORBA. *IEEE Communications Magazine,* 14(2), February 1997.

[Mur97] Stefan Murer. Java und CORBA: Erfahrungen bei Credit Suisse. In *Object Expo, Tutoriumsbeilagen,* October 1997.

[Nie96] Oscar Nierstrasz. Programming Languages. Lecture Notes, University of Bern, SCG, April 1996.

[Nie98] Oscar Nierstrasz. Object-Oriented Software Reengineering. Lecture Notes, Universität Bern, SCG, November 1998.

[ODV94] Tamer M. Özsu, Umeshwar Dayal, and Patrick Valduriez. *Distributed Object Management.* Morgan Kaufmann, 1994.

[OH97] Robert Orfali and Dan Harkey. *Client/Server Programming with Java and CORBA.* Wiley, 1997.

[OMG97a] OMG. CORBA Scripting Language RFP, Final Version. Technical report, OMG, ftp://www.omg.org/pub/docs/orbos/97-06-13.ps, June 1997.

[OMG97b] OMG, Object Management Group, ftp://ftp.omg.org/pub/docs/formal/97-02-25.ps.gz. *CORBA 2.0: Architecture and Specification,* February 1997.

[OMG98] OMG. History of CORBA. Technical report, OMG, ftp://www.omg.org/archives/experts/msg00991.html, fall 1998.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[Ous97] John K. Ousterhout. Scripting: Higher Level Programming for the 21st Century. White Paper, Sun Microsystems, May 1997.

[Pud97] Arno Puder. Verteilte Objekte: DCOM versus CORBA. *iX, Heise Verlag,* August 1997.

[Res98] Lunatech Research. *COPE, Reference Manual.* Lunatech Research, Rotterdam, Netherland, http://www.lunatech.com/research/corba/cope/, June 1998.

[RPP97] Kay Römer, Arno Puder, and Frank Pilhofer. *MICO, Reference Manual.* Universität Frankfurt, http://www.vsb.informatik.uni-frankfurt.de/∼mico/, February 1997.

[SAT98] Martin Schader, Markus Aleksy, and Christoph Tapper. Interoperabilität verschiedener "Object Request Broker" nach CORBA 2.0-Standard. *OBJEKTspektrum, SIGS Publications*, Mai/Juni 1998.

[Sch94] Arnold Scheuing. *Verifikation und Validierung von Software*. Vorlesungsunterlagen, Universität Bern, Mai 1994.

[SGHP97] Douglas C. Schmidt, Aniruddha Gokhale, Timothy H. Harrison, and Guru Parulkar. A High-performance Endsystem Architecture for Real-time CORBA. *IEEE Communications Magazine*, 14(2), February 1997.

[Sie96] Jon Siegel. *CORBA Fundamentals and Programming.* Wiley, 1996.

[SN98a] Jean-Guy Schneider and Oscar Nierstrasz. Components, Scripts and Glue. Submitted for Publication, August 1998.

[SN98b] Jean-Guy Schneider and Oscar Nierstrasz. Scripting: Higher-level Programming for Component-based Systems. In *OOPSLA-Tutorial*, October 1998.

[Sne96] Harry M. Sneed. Die Einbindung alter Host-Software in eine Client/Server-Architektur. *OBJEKTspektrum, SIGS Publications*, 4:36–43, 1996.

[Sun97] Sun, http://java.sun.com/products/jdk/1.1/docs/guide/rmi/spec/rmiTOC.doc.html. *Remote Method Invocation Specification,* 1997.

[Tan95] Andrew S. Tanenbaum. *Verteilte Betriebssysteme.* Prentice Hall, 1995.

[TC98] OMG TC. CORBA Scripting Language RFP. Technical report, OMG, ftp://www.omg.org/library/schedule/CORBA_Scripting_Language_RFP.htm, August 1998.

[Tre96] Markus Tresch. Schlüsseltechnologie zur Entwicklung verteilter Informationssysteme. *Informatik Spektrum*, 19:249–256, October 1996.

[Tre97] Markus Tresch. Middleware: Versuch einer Einordnung. *Informatik/Informatique - Zeitschrift der schweizerischen Informatikerganisation*, Februar 1997.

[VD97] Andreas Vogel and Keith Duddy. *Java Programming with CORBA.* Wiley, 1997.

[Ves97] Savikko Vespe. Design Patterns in Python. In *Proceedings of 6th International Python Conference*, www.python.org/workshops/1997-10/proceedings/, October 1997. Software Systems Laboratory Tampere University of Technology, Finland.

[Vis97] Visigenic Software, Inc., http://www.visigenic.com. *VisiBroker Reference Manual, Version 3.0*, September 1997.

[vLF97] Martin von Löwis and Nils Fischbeck. *Das Python-Buch.* Addison-Wesley, 1997.

[vR96] Guido van Rossum. *Python Reference Manual.* Corporation for National Research Initiatives (CNRI), http://www.python.org/doc/ref/, October 1996.

[WCS96] Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl.* O'Reilly & Associates, 2nd edition, September 1996.

[Wit97] Matthias Withopf. Portierung: Microsofts DCOM für Unix. *iX, Heise Verlag*, August 1997.

[WvRA96] Aaron Watters, Guido van Rossum, and James Ahlstrom. *Internet Programming with Python.* MIS Press, October 1996.