Georg-August-Universität
Göttingen
Zentrum für Informatik

ISSN 1612-6793
Nummer ZFI-BM-2004-06

Bachelorarbeit

im Studiengang "Angewandte Informatik"

Eine Implementation des CORBA Property Service

Benjamin Zeiß

am Insitut für

Mathematik

Bachelor- und Masterarbeiten
des Zentrums für Informatik
an der Georg-August-Universität Göttingen

16. März 2004

Georg-August-Universität Göttingen
Zentrum für Informatik

Lotzestraße 16-18
37083 Göttingen
Germany

Tel. +49 (5 51) 39-1 44 02
Fax +49 (5 51) 39-1 44 03
Email office@informatik.uni-goettingen.de
WWW www.informatik.uni-goettingen.de

Ich erkläre hiermit, daß ich die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Göttingen, den 16. März 2004

Bachelorarbeit

Eine Implementation des CORBA Property Service

Benjamin Zeiß

16. März 2004

Betreut durch Prof. Dr. Robert Switzer
Mathematisches Institut
Georg-August-Universität Göttingen

CORBA ist eine von der OMG entwickelte objekt-orientierte Middelware, die Protokolle und Dienste definiert und damit eine Plattform für verteilte Anwendungen in heterogenen Umgebungen schafft. Der *Property Service* gehört zu den CORBAservices, welche wiederverwendbare und domänen-unabhängige Dienste sind, die die Integration und Interoperation verteilter Objekte unterstützen. Er erweitert die CORBA Dienste um die Möglichkeit beliebige CORBA-Objekte mit dynamischen Attributen, bestehend aus <Name,CORBA-Any>-Tupeln, zu versehen. Die Umsetzung der Spezifikation in eine funktionierende Eiffel Implementation mit MICO/E als Grundlage und den damit verbundenen Problematiken werden in dieser Arbeit beschrieben.

CORBA is an object-oriented middleware developed by the OMG and provides a platform for distributed applications in heterogenous environments by defining protocols and services. The *Property Service* belongs to the class of CORBAservices which are reusable and domain-independent services that support the integration and interoperation between distributed objects. It extends CORBA services with the possibility to attach dynamic attributes consisting of <Name,CORBA-Any> tupels to any CORBA object. This thesis describes the specification's realization into a fully working Eiffel implementation with MICO/E as foundation and the problems involved.

Inhaltsverzeichnis

1 Einleitung

Die rasante Entwicklung von Computernetzwerken ist eine der interessantesten Phänomene unserer Zeit. Den festen Platz, den das wohl prominenteste Netzwerk, das Internet, in der modernen Gesellschaft eingenommen hat, verdankt es zu einem Grossteil Menschen, die sich um Architekturen bemüht haben, die das Leben für Programmierer von Netzwerkanwendungen leichter gemacht haben. Auch wenn die *Common Object Request Broker Architecture* nicht etwa für den Erfolg von E-Mail oder dem World Wide Web verantwortlich ist, so handelt es sich dennoch um eine konsequente Weiterentwicklung der Technologien, die sich in den 90er Jahren bereits manifestiert haben. Monolithische Einzelplatzanwendungen passten immer weniger in das Bild von Netzwerken. Es entstand die Notwendigkeit für eine Plattform, die heterogene Systeme als dezentrale Softwarekomponenten im Sinne der objekt-orientierten Programmierung verbindet. Die OMG (*Object Management Group*) kam als Gruppe vieler namhafter Hersteller zusammen, um für diese Forderung eine Lösung als offenen Standard zu erarbeiten. Diese sollte der vergleichsweise grossen Komplexität, die ein verteiltes Softwaresystem mit sich bringt, ein gewisses Mass an Transparenz verleihen, so dass als Folge die Aufgabe der Softwareentwicklung für verteilten Netzwerkapplikationen wieder zu bewältigen ist. Das Ergebnis ist ein Katalog von Spezifikationen, der mit dem Akronym CORBA umschrieben wird und in vielen Bereichen kommerzieller Branchenanwendungen nicht mehr wegzudenken ist.

Neben z.B. einer einheitlichen Schnittstellenbeschreibung gehören zu CORBA auch Spezifikationen für eine Reihe von Diensten, die teilweise für die Funktionalität von CORBA essentiell sind, aber andererseits auch Dienste, die das Leben der Anwendungsprogrammierer in dem Sinne vereinfachen, als dass sie einen Wekzeugkasten darstellen auf den man bei Bedarf zurückgreifen kann. Der *Property Service*, den diese Ausarbei-

tung zum Inhalt hat, fällt in genau diesen Bereich der sogenannten *CORBAservices*. Es handelt sich um ein Werkzeug, das einer CORBA Anwendung ermöglicht, Objekte, die durch eine sonst weitgehend statische Schnittstelle beschrieben werden, mit dynamischen Attributen zu versehen. Motiviert von erhöhter Flexibilität im Umgang mit Objektattributen und Unabhängigkeit von der statisch erzeugten Schnittstellenbeschreibung erschien Mitte 2000 die Version 1.0 der Spezifikation.

Diese Arbeit gliedert sich in 5 Kapitel. Kapitel 2 befasst sich mit den CORBA Grundlagen, die für das Verständnis des Propery Service notwendig sind. Wegen des Umfanges von CORBA kann im Rahmen dieser Arbeit nur ein kleiner Teil von dem vorgestellt werden, was wirklich Teil von CORBA ist. Der Schwerpunkt liegt in der grundlegenden Beschreibung der Architektur, die sich aus Software-Bus, Schnittstellendefinition und Schnittstellen-Compiler sowie Kommunikationsprotokolle zusammensetzt.

Kapitel 3 stellt den Property Service in seiner Funktionalität vor. Es werden die einzelnen Schnittstellen, die Umsetzung von Zugriffsrechten, Exceptions, die Funktionsweise der Iteratoren sowie die sogenannten Factorymethoden des Property Service besprochen.

Den wichtigsten und umfangreichsten Teil der Arbeit stellt Kapitel 4 dar. Von den ersten Schritten der Quelltexterzeugung aus der Schnittstellenbeschreibung, werden die wesentlichen Teile der Implementierung erläutert. Dazu gehört die dem Dienst zugrunde liegende Datenstruktur, aber auch z.B. die Erörterung von bestimmten Problematiken wie etwa die Umsetzung von bestimmten durch die Spezifikation festgelegten Nebenbedingungen.

Zuletzt bildet Kapitel 5 den Abschluss der Arbeit und fasst die wesentlichen Eckpunkte zusammen. Weiterhin wird herausgearbeitet, was an der Implementierung noch verbesserunswürdig ist.

2 CORBA Grundlagen

Die *Common Object Request Broker Architecture* (CORBA) ist ein Standard der OMG (Object Management Group), der eine Umgebung beschreibt, in der Software-Objekte transparent für den Anwendungsprogrammierer über Netzwerke genutzt werden können. Das Kernstück dieses Standards ist der sogenannte ORB (*Object Request Broker*), der für Nachrichtenzustellungen zwischen CORBA-Objekten zuständig ist und die Komplexität eines verteilten Systems vor seinen Nutzern versteckt. Aus Anwendersicht kann man sich den ORB als einen Software-Bus vorstellen, über den Kommunikationsdaten fliessen, insbesondere z.B. Methodenaufrufe und deren Antworten von über diesen Bus verbundenen Objekten. Dabei besitzt jedes CORBA-Objekt eine Schnittstelle, die durch den ebenfalls von der OMG spezifizierten IDL (*Interface Definition Language*) Standard beschrieben wird. Das IDL-Interface, das eine C++ ähnliche Syntax verwendet, wird von IDL-Compilern in Quelltexte einer (fast) beliebigen Programmiersprache umgewandelt, welcher von dem Anwendungsprogrammierer sowohl auf Clientseite (*Stub*) als auch auf Serverseite (*Skeleton*) in den Applikationsquelltext eingebunden und für Aufrufe bzw. zur Implementierung von Funktionen auf verteilten Objekten genutzt wird. Dabei bedient sich der IDL-Compiler sogenannter *Language Mappings*, die die IDL-Syntax auf eine konkrete Programmiersprache abbildet. Neben der Sprachunabhängigkeit ist auch die Platformunabhängigkeit eine sehr wichtige Eigenschaft des CORBA-Standards. CORBA-Objekte können nicht nur über ein Netz beliebig verteilt sein, sondern auch auf verschiedenen Rechnerarchitekturen Verwendung finden. Die Interoperabilität zwischen den Objekten wird durch den standardisierten ORB und durch das Kommunikationsmodell von CORBA gewährleistet, welches mit GIOP (*General InterORB Protocol*) und IIOP (*Internet InterORB Protocol*) festlegt wie Objekte von ORB zu ORB miteinander kommunizieren. Im Folgenden

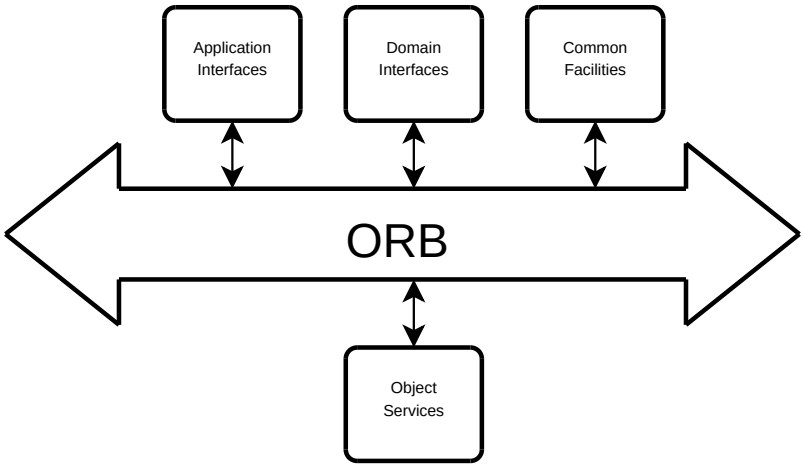

Abbildung 2.1: *Object Management Architecture*

werden die einzelnen Bestandteile näher erläutert.

2.1 OMA

CORBA selbst ist ein Teil der *Object Management Architecture* (OMA). Es stellt ein Framework dar, das sich aus dem ORB als Kernteil, Object Services (*CORBAservices*), Common Facilities (*CORBAfacilities*), Domain Interfaces und Application Objects zusammensetzt. Dabei ist der ORB primär für die Kommunikation zwischen CORBA-Objekten zuständig. Die restlichen Komponenten der OMA nutzen den ORB zur Kommunikation. Die Application Objects sind CORBA-Objekte, die spezifisch für eine Applikation entwickelt werden, während die anderen drei OMA-Teile Domain Interfaces, Common Facilities und Object Services wiederverwendbare Dienste implementieren, die auch durch die OMG spezifiziert werden. Näheres dazu in 2.6.

2.2 Der ORB-Core

Das Kernstück von CORBA ist der ORB, welcher die Grundlage für die Kommunikation von CORBA-Objekten untereinander ist. Er ist verantwortlich für die Lieferung von

Anfragen zu Objekten und für die Zustellung der entsprechenden Antworten. Dabei ist ausschlaggebend, dass der ORB diese Aufgabe für den Anwendungsprogrammierer transparent erledigt. Nutzer einer CORBA-Implementierung werden sich nicht mit Netzwerkprogrammierung und Protokollen (z.B. TCP/IP) auseinandersetzen, sondern lediglich auf die Funktionalität der Architektur zurückgreifen, die alle komplizierten Details vor ihm versteckt. So weiss der Client auch nicht direkt wo sich sein Ziel-Objekt befindet. Es könnte sich sowohl auf dem selben Rechner als auch im anderen Teil der Welt befinden. Die Standortsdaten befinden sich in Objektreferenzen (IORs), die aber auf Applikationsebene nicht näher inspiziert werden. Ebensowenig interessiert den Client wie das Ziel-Objekt implementiert ist. Hardware, Programmier- oder Skriptsprache können für jedes CORBA-Objekt beliebig gewählt werden, solange die Interoperabilität zu den ORB-Versionen bei Server und Client gewährleistet ist. Der Client verwendet zum Nutzen von CORBA-Objekten nur die aus den IDL-Dateien erzeugten *Stubs*, die auf seine Sprache bzw. Plattform angepasst sind. Ist ein Ziel-Objekt nicht in einem ausgeführten Zustand, so kümmert sich der ORB um die Aktivierung des Objektes. Der Client muss daher nichts über den Zustand von seinem Ziel-Objekt wissen. Insgesamt erlaubt der ORB also, dass sich der Applikationsprogrammierer mehr um die eigentlichen Probleme seiner Software kümmern kann und sich in Bezug auf die Verteiltheit seiner Anwendung komplett auf seine CORBA-Implementierung verlassen kann.

Der grobe Ablauf eines Methodenaufrufes in der CORBA-Umgebung beginnt damit, dass der Client zunächst sein Ziel-Objekt identifizieren muss. Zu diesem Zweck benötigt er eine Objektreferenz, die immer parallel mit einem neuen CORBA-Objekt erzeugt wird und auch immer nur auf dieses verweist. Eine solche Referenz erhält der Client, indem er entweder ein neues CORBA-Objekt erzeugen lässt (über sogenannte Factory-Objekte) und dessen Referenz erhält, eine Anfrage an einen Directory Service stellt oder aus einem String zurückkonvertiert. Directory Services sind Indizes, die in etwa einem Telefonbuch entsprechen. In Abschnitt 2.6 werden *CORBAservices* näher behandelt. Weiterhin lassen sich Objektreferenzen auch in einen String und zurück umwandeln. Auf diesem Wege könnten Objektreferenzen sogar per E-Mail geschickt werden. Alternativ könnte eine solche String-Referenz auch z.B. in einer Datenbank abgelegt werden.

Dadurch, dass Objekte nur durch das Schicken von Requests erzeugt werden, stellt sich sehr schnell die Frage wie ein CORBA-Programm seine erste Objektreferenz bekommt (Bootstrapping). Meistens ist die erste Referenz, die jedes CORBA-Programm erhält, die des Naming Services dessen Standort durch Übergabeparamter dem Programm mitgeteilt wird.

2.3 IDL

Die *Interface Definition Language* ist eine deklarative Sprache, in der die Schnittstellen der CORBA-Objekte festgelegt werden. Durch die Schnittstellendefinition in einer von Programmiersprachen unabhängigen Syntax wird die Sprachunabhänigigkeit von CORBA erst ermöglicht, so dass beispielsweise Methodenaufrufe von einem in Java implementierten Client auf ein in Eiffel implementiertes CORBA-Objekt kein Problem darstellen. Die von der OMG spezifizierten *Language Mappings* sorgen dafür, dass die Quelltexterzeugung aus der IDL-Schnittstellendefinition in Programmiersprachen einheitlich ist. Das Ergebnis des *Language Mappings* sind u.a. sogenannte *Stubs* und *Skeletons* in der Programmiersprache der Wahl, mit deren Hilfe entsprechende Implementierungen vorgenommen werden können. Das erfordert einen IDL-Compiler, der die *Language Mappings* von IDL in andere Sprachen (z.B. C, C++, Python, Eiffel) umsetzt. Die IDL-Sprachelemente sind ähnlich zu den aus Programmiersprachen bekannten (*long*, *double*, *boolean*, *string*, *struct*, *union* etc.). Listing 2.1 zeigt ein einfaches Beispiel einer Schnittstellendefinition, die grundlegende Eigenschaften eines Bankkontos darstellt. Die IDL-Syntax wird in dieser Arbeit nicht näher abgehandelt.

```
1  interface Account
2  {
3      void deposit( in unsigned long amount );
4      void withdraw( in unsigned long amount );
5      long balance();
6  };
```

Listing 2.1: *IDL Beispiel*

2.4 GIOP/IIOP

Bei CORBA läuft die Kommunikation wie bereits erwähnt über Objektreferenzen (sog. IORs=Interoperable Object References). Über diese können Programme (*Clients*) Methodenaufrufe auf verteilten Objekten (*Server*) durchführen. Es bleibt anzumerken, dass man aus Sicht einzelner Operationen im CORBA-Modell durchaus von *Clients* und *Server* reden kann. Allerdings ist diese Beziehung bei genauerem Blick auf das Kommunikationsmodell nicht ganz richtig, da CORBA-Objekte auch Callbacks zum Client machen können, was zur Folge hat, dass die Rollen vertauscht werden. Insofern wäre es im Allgemeinen korrekter von *Peers* zu sprechen. Wie schon zuvor erwähnt läuft die Kommunikation unter den Objekten unter Aufsicht des ORB. In dem Internet Umfeld kommunizieren ORBs mit dem TCP/IP-Protokoll IIOP (*Internet InterORB Protocol*). IIOP basiert auf dem *General InterORB Protocol* (GIOP), das ein grundlegendes Protokoll für verbindungs-orientierte Nachrichtenkommunikation zwischen ORBs spezifiziert. IIOP definiert wie GIOP auf dem TCP/IP-Layer implementiert wird und ist momentan die populärste Art ORBs miteinander zu verbinden. Andere Implementierungen, etwa GIOP über Firewire (IEEE 1394), sind denkbar oder bereits spezifiziert. Die Objektreferenzen müssen dabei je nach Implementierung verschiedene Informationen tragen - jeweils die Informationen, die notwendig sind, um das Objekt bei dem genutzten Kommunikationsmodell zu finden und zu nutzen. Bei IIOP enthält eine IOR Hostnamen und Portnummer. Durch die genaue Spezifikation des Kommunikationsmodelles durch die OMG ist es möglich, ORBs verschiedener Hersteller miteinander interagieren zu lassen.

2.5 Stubs und Skeletons

Stubs und *Skeletons* entstehen, wenn man Quelltext mit einem IDL-Compiler (z.B. *eifidl* bei Eiffel) erzeugt. Häufig redet man auch von *Client Stubs* und *Server Skeletons*. Der *Client Stub* wird in den Client-Teil des Programmes mit reinkompiliert und ermöglicht den Zugriff auf die Serverkomponente, die durch die IDL-Datei spezifiziert wird. Tatsächlich handelt es sich dabei um ein sogenanntes Proxy-Objekt, welches für den *Client* die Schnittstelle der IDL-Datei repräsentiert, aber nicht die Server-Implementierung

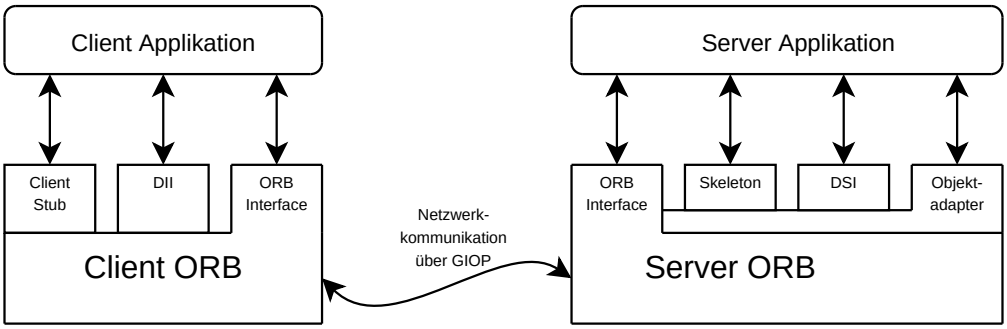

Abbildung 2.2: *Vereinfachtes Schema von CORBA*

beinhaltet. Stattdessen wird ein Aufruf auf das Proxy-Objekt in eine Nachricht verpackt (*Marshalling*) und mit Hilfe der Objektreferenz über den ORB an das korrekte Ziel-Objekt geschickt, der die Implementierung enthält. Analog besteht das *Server Skeleton* aus Quelltext, welcher z.B. das *Demarshalling* der Nachrichten übernimmt. Die eigentliche Server-Funktionalität wird in den Servant-Klassen implementiert. Diese werden ebenfalls vom IDL-Compiler erzeugt und stellen die eigentliche Umsetzung der Schnittstellendefinition dar. Daher sind die Methodenrümpfe von Servant-Klassen direkt nach Erzeugung auch leer. Der Programmierer muss diese mit sinnvollen Inhalten füllen.

2.6 CORBAservices, CORBAfacilities

Als Bestandteil der OMA kommt den *CORBAservices* und *CORBAfacilities* die Bedeutung wiederverwendbarer Dienste zu, die nichts spezifisches für eine spezielle Applikation beinhalten. Bei den *CORBAservices* handelt es sich um allgemeingültige Dienste, unabhängig jeder Domäne, die viele verteilte Programme benutzen. So ist der wichtigste *CORBAservice* wohl der Naming Service, der ein Verzeichnis für bei ihm angemeldete CORBA-Objekte ist und quasi das Telefonbuch darstellt. Die Komponente ist insbesondere wegen der von CORBA geforderten Ortstransparenz wichtig. Wird der Rechner mit einem CORBA-Dienst an einen anderen Ort gebracht, so hat dies praktisch keine Auswirkung auf die Komponenten, die mit ihm arbeiten. Der verlagerte CORBA-Dienst

meldet sich bei dem Naming Service wieder an und kann sofort von den anderen Client-Komponenten gefunden werden, sobald sie neu in dem Naming Service nachsehen und ihre Objektreferenz aktualisieren. Ein weiterer wichtiger Service ist der *Trading Service*, welcher umgekehrt die Gelben Seiten darstellt. In diesem Verzeichnis werden Objekte anhand ihrer Eigenschaften eingetragen. Der *Property Service* ist ebenso ein *CORBAservice* und wird in Kapitel 3 genau beschrieben.

CORBAfacilities sind Dienste, die viele Applikationen gebrauchen können. Sie sind nicht so fundamental wie *CORBAservices*. Ein Beispiel für eine *CORBAfacility* ist die *Internationalization and Time Facility* [11]. Sie behandelt die kulturell unterschiedlichen Repräsentationen etwa von Zeit- oder Geldschreibweise. Weiterhin gibt es vertikale *CORBAfacilities*, die Domänen-spezifische Dienste für bestimmte Industriezweige enthalten. Beispiel für solche Dienste sind *Bibliographic Query Service* [8] oder *Clinical Observations Access Service* (COAS) [9].

2.7 Weitere CORBA-Komponenten

Weitere wichtige Bestandteile der CORBA-Kernes sind das *Interface Repository* (IR), das *Dynamic Invocation Interface* (DII), das *Dynamic Skeleton Interface* (DSI) und die Objektadapter *Basic Object Adapter* (BOA) sowie der *Portable Object Adapter* (POA). Da diese aber zum Verständis des *Property Service* nicht so wichtig sind, wird an dieser Stelle auf [3] oder [19] für Details verwiesen.

3 Der Property Service

Der *Property Service* gehört in der OMA zu den Object Services und zählt somit zu der Klasse der Domänen-unabhänigen Dienste, die allgemein für CORBA-Applikationen brauchbar sind. Seine Aufgabe ist, CORBA-Objekte dynamisch um Attribute erweiterbar zu machen. Im Normalfall stehen Attribute von CORBA-Objekten in der Schnittstellendefinition der IDL-Datei. Gelegentlich kann es aber nützlich sein, ein Objekt während der Laufzeit mit Eigenschaften zu versehen, die ausserhalb der a priori festgelegten und statischen IDL-Definition liegen. Das könnte z.B. dann sinnvoll sein, wenn der Nutzer eines Objektes Informationen an ein Objekt heften will, wenn er keine Möglichkeit hat, das Objekt selbst abzuändern, weil es sich nicht in seinem Zuständigkeitesbereich befindet. Abbildungen 3.2 und 3.3 stellen das UML-Interfacediagramm des *Property Service* dar.

3.1 Property und PropertySet

Kernstück des *Property Service* ist das PropertySet-Interface. Ein *PropertySet* ist eine Menge von Properties, wobei eine *Property* ein Zwei-Tupel, bestehend aus *property_name* und *property_value*, ist.

Property
property_name
property_value

Dabei ist *property_name* ein String, der den Namen der *Property* beinhaltet und *property_value* enthält einen Wert vom Typ CORBA IDL-Typ *any*. Die typischen Operationen auf einem PropertySet sind *define_property*, *get_number_of_properties*,

get_property_value, *delete_property* und *is_property_defined*. Dazu gibt es entsprechend analoge Operationen, die zur Gruppenverarbeitung mehrerer oder aller Properties (Batch-Verarbeitung) gedacht sind. Hier kommt der *Properties*-Datentyp zum Einsatz, welcher eine Sequenz des Typs *Property* ist. Im Gegensatz zu normalen Attributen einer IDL-Spezifikation können Properties neu angelegt und gelöscht werden. Der Client kann Informationen zu dem Inhalt eines PropertySets durch die Anzahl der Properties oder eine Liste der Property-Namen erhalten. Das um die Exceptions gekürzte IDL-Interface von *PropertySet* ist in Listing 3.1 abgedruckt.

3.2 PropertyDef/PropertySetDef und PropertyModes

PropertySetDef ist eine Ableitung von *PropertySet* und erweitert es um die Möglichkeit mit Zugriffsrechten zu arbeiten, welche durch das Attribut *PropertyMode* genau be-

```
1  interface PropertySet
2  {
3      /* Support for defining and modifying properties */
4      void define_property (in PropertyName property_name, in any property_value)
5          raises (...);
6      void define_properties (in Properties nproperties) raises (...);
7
8      /* Support for Getting Properties and their Names */
9      unsigned long get_number_of_properties ();
10     void get_all_property_names (in unsigned long how_many, out PropertyNames property_names,
11                                  out PropertyNamesIterator rest);
12     any get_property_value (in PropertyName property_name) raises (...);
13     boolean get_properties (in PropertyNames property_names, out Properties nproperties);
14     void get_all_properties (in unsigned long how_many, out Properties nproperties,
15                              out PropertiesIterator rest );
16
17     /* Support for Deleting Properties */
18     void delete_property (in PropertyName property_name) raises (...);
19     void delete_properties (in PropertyNames property_names) raises (...);
20     boolean delete_all_properties ();
21
22     /* Support for Existence Check */
23     boolean is_property_defined (in PropertyName property_name) raises (...);
24 };
```

Listing 3.1: *IDL-Interface PropertySet*

```
1  interface PropertySetDef : PropertySet
2  {
3      /* Support for retrieval of PropertySet constraints */
4      void get_allowed_property_types (out PropertyTypes property_types);
5      void get_allowed_properties (out PropertyDefs property_defs);
6
7      /* Support for defining and modifiying poperties */
8      void define_property_with_mode (in PropertyName property_name, in any property_value,
9                                      in PropertyModeType property_mode) raises (...);
10     void define_properties_with_modes (in PropertyDefs property_defs) raises (...);
11
12     /* Support for Getting and Setting Property Modes */
13     PropertyModeType get_property_mode (in PropertyName property_name) raises (...);
14     boolean get_property_modes (in PropertyNames property_names,
15                                 out PropertyModes property_modes);
16     void set_property_mode (in PropertyName property_name, in PropertyModeType property_mode)
17         raises (...);
18     void set_property_modes (in PropertyModes property_modes) raises (...);
19 };
```

Listing 3.2: *IDL-Interface PropertySetDef*

schrieben wird. Zu diesem Zweck wird die CORBA-Struktur *Property* um *PropertyMode* erweitert und trägt den Namen *PropertyDef*. Im Folgenden wird der Begriff *PropertySets* in dem Sinne gebraucht, dass damit nicht strikt mehrere Elemente vom IDL-Interface *PropertySet* gemeint sind, sondern sowohl *PropertySet* als auch *PropertySetDef*. Anderenfalls wird dies explizit angegeben.

Property Mode Types	
normal	Lese-, Lösch-und Schreibzugriff
read_only	Lösch- und Lesezugriff
fixed_normal	Lese- und Schreibzugriff
fixed_readonly	nur Lesezugriff
undefined	zeigt *PropertyNotFound* bei einem Batch-Lesezugriff an

Die Operationen in dem Interface *PropertySetDef* bieten zusätzlich zu den von *PropertySet* geerbten Funktionalitäten Möglichkeiten zum Setzen bzw. Erfragen von Properties mit Modes. Ebenso wie schon beim PropertySet werden auch hier Gruppen-Operationen unterstützt (z.B. *define_properties_with_modes*). Listing 3.2 zeigt das wie-

der um die Exceptions gekürzte IDL Interface von PropertySetDef.

3.3 Exceptions

Insbesondere auch durch die *PropertyModes* und Nebenbedingungen der Factories (siehe Abschnitt 3.5), die in *PropertySetDef* eine Rolle spielen, ist es für einen *Property Service* unerlässlich, dass eine konsequente und detaillierte Fehlerbehandlung existiert. Daher werden von der OMG in der *Property Service* Spezifikation entsprechende Exceptions vorgegegeben.

Exceptions
ConstraintNotSupported
InvalidProeprtyName
ConflictingProperty
PropertyNotFound
UnsupportedTypecode
UnsupportedProperty
UnsupportedMode
FixedProperty
ReadOnlyProperty

Es wird unterschieden zwischen Exceptions, die sich auf eine einzelne Property beziehen (z.B. *define_property*) oder auf eine Gruppe von Properties (z.B. *define_properties*). Im Fall der einzelnen Property wird sofort eine Exception geworfen, sobald der Fehler auftritt. Für letzteren Fall gibt es die Exception *MultipleExceptions*, welche eine PropertyException-Liste beinhaltet. In einer *PropertyException* wird immer ein Tupel <ExceptionReason,PropertyName> gespeichert, so dass bei einer Gruppenoperation immer ein Bezug zwischen Exception und dem Namen der Property hergestellt werden kann. Erst wenn die Gruppenoperation abgeschlossen ist, wird die MultipleException geworfen, sofern die Liste nicht leer ist. Dadurch wird die Gruppenoperation nie durch eine Exception unterbrochen.

3.4 Iterator

Einige Operationen auf einem *PropertySet* oder *PropertySetDef*, die dafür gedacht sind, eine Liste von Properties oder Propertynamen zurückzugeben, sind so definiert, dass die Grösse der Liste als Argument mit übergeben wird. Die Rückgabewerte dieser Operationen sind eine Property-Liste (Typ *Properties*) der angegebenen Grösse und eine Referenz auf einen *PropertiesIterator* oder *PropertyNamesIterator*, falls nicht alle Ergebnisse in die Liste gepasst haben. Mit Hilfe des Iterators kann durch den Rest des Ergebnisses mit den Operationen *next_one* und *next_n* iteriert werden. Die *reset* Methode setzt den Iterator wieder zurück an die erste Position und *destroy* zerstört ihn. Im Gegensatz zu dem bekannten Iterator-Entwurfsmuster aus [1] handelt es sich hier allerdings nicht um polymorphe Iteratoren. Statt dessen sind die beiden Iterator Interfaces aus dem Property Service konkret und werden von dem PropertySet bzw. PropertySetDef direkt zurückgegeben.

3.5 PropertySetFactory/PropertySetDefFactory

In der Praxis gibt es zwei verschiedene Möglichkeiten PropertySets zu nutzen:

- Ableitung eines Interfaces von PropertySet bzw. PropertySetDef
- Erzeugung von PropertySets durch Factory-Interfaces

Die erste Möglichkeit ist simpel. Das Interface des CORBA-Objektes, das ein *PropertySet* bereitstellen soll, erbt IDL-seitig von *PropertySet* oder *PropertySetDef* und bietet somit automatisch die Funktionalitäten eines solchen.

Ist dies nicht möglich, weil der Programmierer keinen Einfluss auf das CORBA-Objekt hat, an welches er gerne Properties anhängen möchte oder er wünscht Nebenbedingungen zu definieren (z.B. welche CORBA-Typcodes die Properties in einem *PropertySet* haben dürfen), so kann er auf die *PropertySetFactory* bzw. *PropertySetDefFactory* zurückgreifen. Implementierungsseitig werden die Factories durch einen Daemon erzeugt, in dessen Prozess beliebig viele *PropertySets* unabhängig von Objekten

gespeichert werden können. Wie die Beziehung zwischen PropertySets und Objekten hergestellt wird, gehört nicht zu der *Property Service* Spezifikation [12] und wird deshalb hier auch nicht genauer erörtert. Eine gute Möglichkeit dafür ist der *Relationship Service* [13]. Er dient der Verwaltung von Beziehungen zwischen Objekten.

Beide Factories bieten die Operationen *create_propertyset*, *create_constrained_propertyset* und *create_initial_propertyset* an. Die create_propertyset-Operation in der *PropertySetFactory* erzeugt ein einfaches neues PropertySet. Welchen *PropertyModeTypes* (read_only, fixed_read_only, etc.) die Properties unterliegen, legt [12] nicht fest. Die create_constrained_propertyset-Operation kann ein beschränktes *PropertySet* erzeugen, welches lediglich angegebene CORBA-Typecodes zulässt und nur die übergebenen Properties enthalten darf. Die Methode *create_initial_propertyset* kann ein *PropertySet* mit initialen Properties erzeugen.

Analog sehen die Operationen auf einem *PropertySetDef* in der *PropertySetDefFactory* aus. Nur werden hier *PropertySetDefs* erzeugt und *PropertyDefs* an die Operationen für initiale Properties übergeben.

Es handelt sich beim *Property Service* nicht nur um einen *CORBAservice* im herkömmlichen Sinne wie z.B. der *Naming Service* oder *Trading Service*. Vielmehr bietet er neben dem Daemon, der mit dem Angebot der Factories eher den konventionellen Teil eines *CORBAservices* ausmacht, auch eine Toolkit-artige Funktionalität, da in vielen Fällen wohl die Wahl auf Ableitung von *PropertySet* oder *PropertySetDef* fällt. Der Grund dafür liegt ganz einfach in der Tatsache, dass es nicht immer sinnvoll ist, dass mit einem speziellen Objekt verknüpfte Attribute in einem anderen Prozess existieren.

3.6 Ein Anwendungsszenario

Ein interessanter Anwendungsfall für die praktische Nutzung des *Property Service* findet sich in der OMG Audio/Visual Stream Spezifikation [10], die ein Architekturmodell und IDL-Interfaces zum Bauen von verteilten Multimedia-Streaming Frameworks bietet. Eine Schlüsselkomponente von diesem ist die Multimedia Device Factory (*MMDevice*), welches das Verhalten eines Multimediagerätes abstrahiert. Ein solches Gerät kann sowohl physisch (z.B. Lautsprecher) als auch logisch (z.B. ein Video, welches von

Festplatte gelesen wird) sein. Das *MMDevice* kapselt die gerätespezifischen Parameter eines Multimediagerätes. Diese Parameter unterscheiden sich je nach Gerät. So ist für ein Gerät zur Videoausgabe z.B. das Videokodierungsformat (z.B. MPEG2) und die Frame-Rate interessant während ein Audioausgabe-Gerät Audioformat und Samplerate geeigneter sind. Weil die mit dem *MMDevice* verbundenen Attribute je nach Gerät so verschieden sein können, können dieses Attribute nicht zur statischen Schnittstellendefinition gehören, sondern müssen dynamisch sein. Der *Property Service* bietet sich für diesen Zweck an, da er genau diese Flexibilität bietet. Im Falle des *MMDevice* wird der *Property Service* durch Ableitung genutzt. Abb. 3.1 zeigt an einem Beispiel wie die dynamischen Attributezuweisungen bei einem *MMDevice* aussehen könnten.

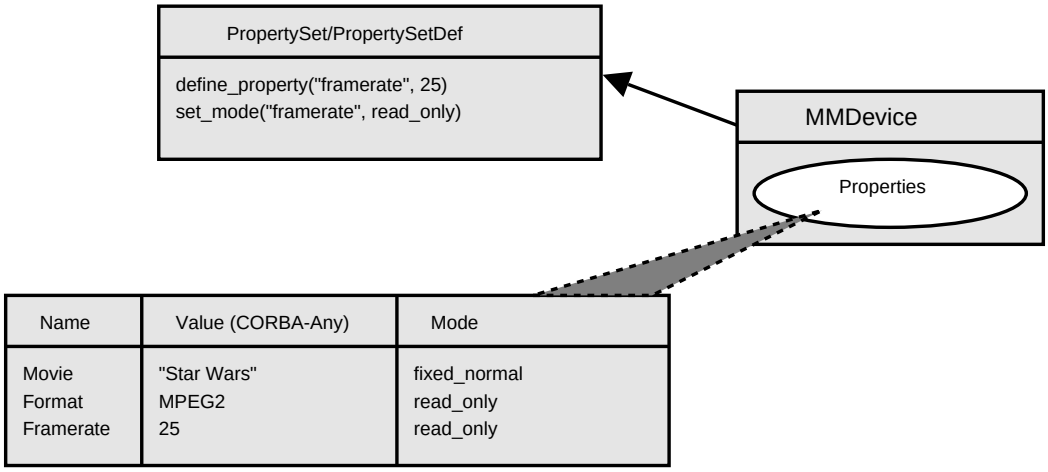

Abbildung 3.1: *MMDevice benutzt den Property Service*

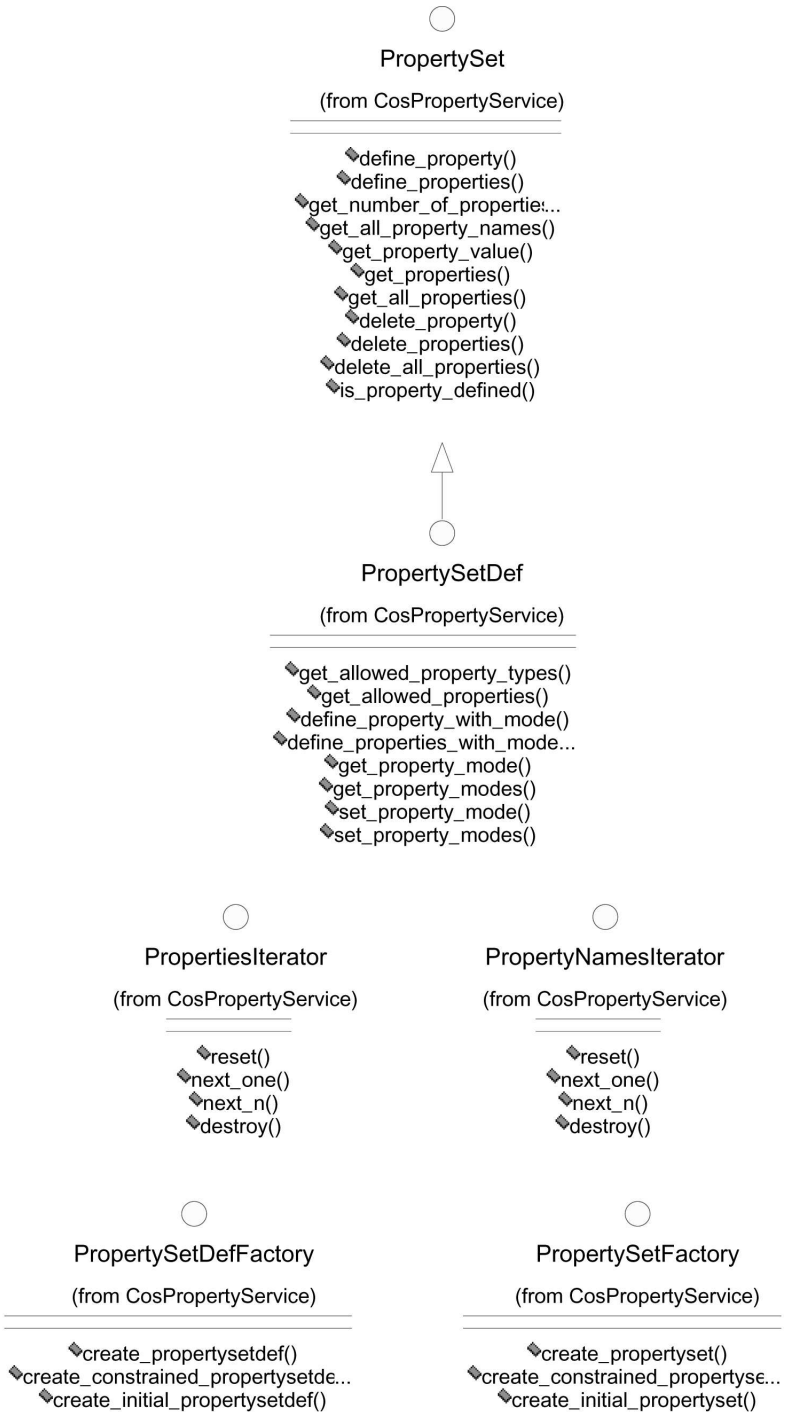

Abbildung 3.2: *CORBA-Interfaces des Property Service*

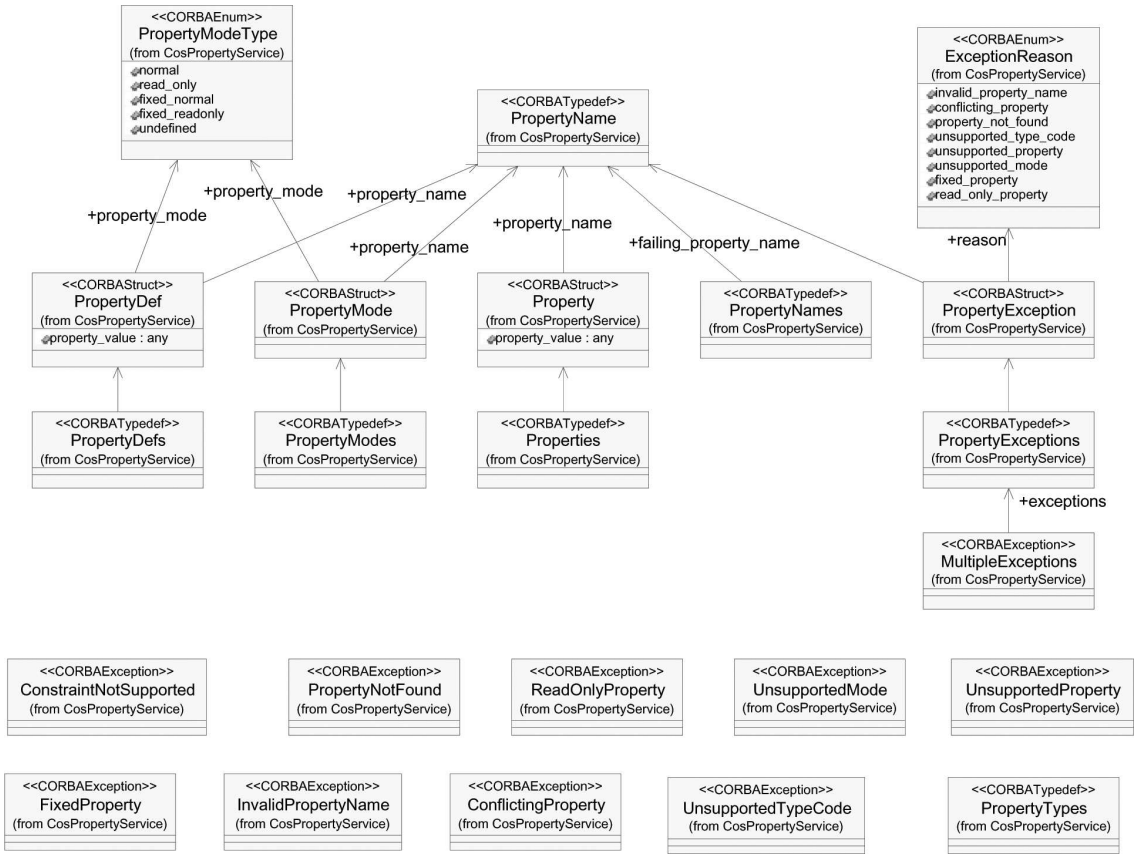

Abbildung 3.3: *Datentypen des Property Service*

4 Implementierung

Die Implementierung des *Property Service* basiert auf dem MICO/E ORB, der in der Arbeitsgruppe um Prof. Dr. R. Switzer am Göttinger Mathematischen Institut entwickelt wurde. MICO/E ist eine Eiffel Implementierung der CORBA-Spezifikation, die auf der Ausarbeitung von Kay Römers Diplomarbeit [4] basiert. Unter [15] ist auch die Implementierung des hier beschriebenen Property Service im Quelltext zu finden.

4.1 Orientierung

Dadurch, dass mit den IDL-Interfaces und der Spezifikation des *Property Service* schon sehr viele Details vorgegeben sind, finden sich die interessanten Implementierungsfragen weniger in der Schnittstelle selbst, sondern z.B. in der Wahl der intern verwendeten Datenstrukturen. Zu implementieren sind die sechs Interfaces des Property Service:

- PropertySetFactory
- PropertySetDefFactory
- PropertySet
- PropertySetDef
- PropertyNamesIterator
- PropertiesIterator

Der IDL-Compiler (Im Falle von MICO/E *eifidl*) erzeugt aus diesen Interfaces Klassen mit leeren Methodenrümpfen in dem Verzeichnis *servant*. Diese gilt es mit sinnvollen

Programmcode zu füllen. Die Erzeugung von Exceptions und Datentypen erledigt der IDL-Compiler alleine, so dass diese sich nach der Compilierung in einem Verzeichnis *base* befinden und direkt ohne weiteres benutzt werden können. Weiterhin werden die Verzeichnisse *skeleton*, *stub* und *proxy* erzeugt. Dem Namen entsprechend beinhalten die ersten beiden Verzeichnisse die Skeleton- und Stub-Klassen wie in Kapitel 2 beschrieben. Die Proxy-Klassen sind spezielle Stub-Klassen, die für den Fall optimiert sind, wenn sich Client und Server im selben Prozess befinden. Stub-, Skeleton- und Proxyklassen enthalten aber nur die Infrastruktur für die Client-Server Kommunikation und müssen für die Implementierung des *Property Services* nicht näher betrachtet werden. Man verlässt sich einfach darauf, dass diese korrekt funktionieren.

Zur Implementierung der Interfaces kommt ein kleiner Daemon hinzu, der die PropertySetFactory und PropertySetDefFactory zur Vefügung stellt. Das Grundgerüst von diesem Daemon wird allerdings nicht von dem IDL-Compiler erzeugt. Wie wir in Teil 4.7 sehen werden, hält sich der Aufwand für dessen Realisierung allerdings in Grenzen.

4.2 Implementierungsentwurf

Während vieles durch die Spezifikation vorgegeben ist, so muss der CORBAservice-Entwickler dennoch einige Implementierungsentscheidungen treffen, die ihm nicht von der OMG abgenommen werden. Bei dem *Property Service* drehen sich die zentralen Fragen bei der Implementierung um folgende Themen:

- Welche Datenstruktur wird zur Speicherung der Properties gewählt?
- Wie sieht das Zusammenspiel der Servant-Klassen von PropertySet und PropertySetDef aus?
- Wie werden *PropertyModes* und Nebenbedingungen (Constraints) gehandhabt?
- Wie bewegt man die *Property Service* Iteratoren und den Iterator des ADT_DICTIONARY zur Kooperation?
- Wie wird der Daemon implementiert, der den Zugriff auf *PropertySetFactory* und *PropertySetDefFactory* ermöglicht?

- Wie wird die Implementierung getestet?

Die wohl mit Abstand wichtigste Entscheidung, die bei der Implementierung des *Property Service* zu treffen ist, ist die Wahl der Datenstruktur, in der die Properties gespeichert werden sollen. Dabei muss man sich genau überlegen wie der Service in der Praxis gebraucht werden könnte und inwieweit er deshalb auf Performance oder Speicherbelegung optimiert sein soll. Das Auffinden von Properties anhand ihres Namens ist der zeitkritische Teil in dieser Implementierung. Dieses Problem ist in der Informatik bestens erforscht und anhand von Standardliteratur wie [17] lässt sich für jeden Anspruch ein geeigneter Algorithmus auswählen. Die Datenstruktur, die zur Service-internen Speicherung der Properties für die MICO/E Implementierung gewählt wurde, ist ein sogenanntes *Dictionary*, welches auf dem Prinzip des Hashings beruht.

4.3 Hashing

Hashing basiert auf einer arithmetischen Transformation $h(k) \longrightarrow i$, die Schlüssel k durch eine Hashing-Funktion h auf Adressen i in einer Tabelle der Grösse N abbildet. Mit dem Wissen, dass i Werte zwischen 1 und N annehmen kann, lässt sich i als Tabellenindex für das zu speichernde Element mit Schlüssel k verwenden. Ist $h(k_1) = h(k_2)$ mit $k_1 \neq k_2$ für in die Tabelle einzufügende Schlüssel k_1 und k_2, so redet man von einer *Kollision*.

Zentrale Bedeutung bei der Abbildung spielt die Hash-Funktion. Idealerweise ist eine solche *bijektiv*, was bedeutet, dass alle zu hashende Schlüssel ohne Kollision auf unterschiedliche Indizes der Tabelle abgebildet werden, ohne dass Indizes unbesetzt bleiben. In der Praxis treten aber Kollisionen auf, weshalb die Funktion nicht *injektiv* ist. Sie ist auch nicht *surjektiv*, weil nicht zwingend alle Indizes der Tabelle verwendet werden bzw. die Tabelle nicht immer genau so gross wie die Anzahl der einzufügenden Elemente ist. Beispiel 4.3.1 zeigt eine einfache Hashfunktion für Integer-Schlüssel.

Beispiel 4.3.1

$$h(k) = k \ mod \ N \tag{4.1}$$

Die Qualität der Hash-Funktion hängt von folgenden Faktoren ab:

- Kollisionswahrscheinlichkeit
- Komplexität von h
- Grösse der Tabelle in Relation zu der Anzahl der einzufügenden Elemente (*Belegungsfaktor*)

Der zweite wichtige Bestandteil bei einer hashing-basierten Suche ist die *Kollisionsbeseitigung*, in der die Fälle behandelt werden, in denen auf denselben Wert i abgebildet wird. Im einfachsten Fall wird das Tabellenelement am Index i das erste Element einer verketteten Liste sein, welche alle kollidierenden Elemente enthält (*geschlossenes Hashing*, Abb. 4.1). Es gibt hier aber auch Methoden mit kürzeren Suchzeiten.

Dadurch, dass die Hash-Funktion direkt den Tabellenindex liefert, ist eine Hashing-Tabelle im Prinzip sehr performant (unter Voraussetzung, dass die Hash-Funktion nicht zu komplex ist). Allerdings ist die Performance in der Praxis von der Grösse der Tabelle und somit der Kollisionsvermeidung abhängig. Insofern stellt sie eher den Kompromiss zwischen Zeit- und Platzbedarf dar. Wählt man die Tabelle gross, so werden Kollisionen unwahrscheinlicher. Jedoch steigt der Platzbedarf der Tabelle dadurch entsprechend an.

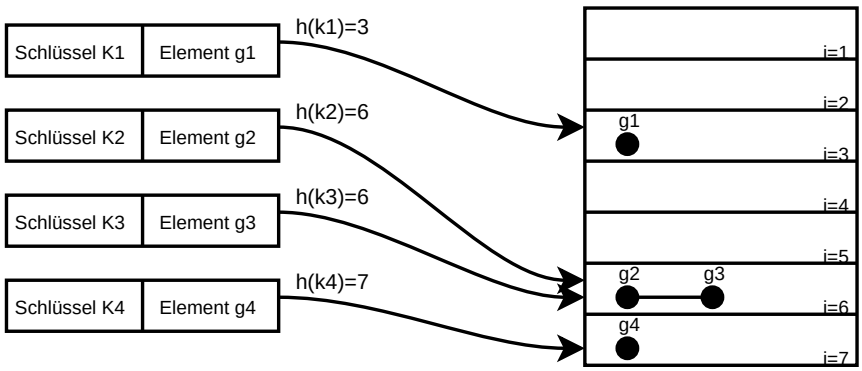

Abbildung 4.1: *geschlossenes Hashing*

Der sogenannte *Belegungsfaktor* α einer Hash-Tabelle ist ein Indikator, der aussagt wie gut die Auslastung der Tabelle ist.

Definition 4.3.1 (Belegungsfaktor) *Der Belegungsfaktor α für eine Hash-Tabelle der Grösse m, in welcher n Elemente gepspeichert sind, ist definiert durch*

$$\alpha = \frac{n}{m} \tag{4.2}$$

.

Ist $n < m$ bzw. $\alpha < 1$, d.h. werden insgesamt höchstens so viele Elemente gespeichert wie die Tabelle gross ist, so ist bei gleichmässiger Verteilung die mittlere Zugriffszeit auf ein Element in der Hash-Tabelle $O(1)$.

4.4 ADT_DICTIONARY

Das in der MICO/E Implementierung des *Property Service* genutzte *ADT_DICTIONARY*, das Teil der ADT-Sammlung von MICO/E ist, ist im Prinzip ein durchsuchbarer Container mit <Schlüssel,Element>-Tupeln. Suchen, Einfügen und Löschen sind die wichtigen Operationen auf einem Dictionary. Es sind grundsätzlich Einträge mit gleichem Schlüssel erlaubt. Eine effiziente Möglichkeit Dictionaries zu implementieren sind Hash-Tables, was auch im Falle des ADT_DICTIONARY zutrifft. Genauer gesagt kommt hier ein geschlossenes Hashing zum Einsatz, wie unter 4.3 besprochen. Im Gegensatz zur Hash-Table ist das Dictionary nicht auf eine bestimmte Tabellengrösse fixiert. Vielmehr wird hier der Belegungsfaktor begrenzt und die Tabelle bei Bedarf vergrössert (*Rehashing*), so dass immer eine konstante Zugriffszeit $O(1)$ gewährleistet werden kann. Diese Belegungsfaktor-Grenze, die zur Tabellenvergrösserung führt, kann vom Programmierer selbst bestimmt werden. Dazu erbt es von der Klasse ADT_TRAVERSABLE, was Iteration über die Hash-Table mit einem Iterator-Objekt (ADT_ITERATOR) ermöglicht. Eine *amortisierende Analyse* beweist, dass sich die Kosten eines Rehashs kaum bemerkbar machen und asymptotisch die Kosten für eine *put*-Operation im Mittel $O(1)$ bleiben.

Das Dictionary stellt als Datenstruktur einen Zeit- und Platzkompromiss für den *Property Service* dar. Ein Dictionaryeintrag bildet sich fast 1:1 auf die Struktur einer *Property* ab. Da man ohnehin nicht voraussagen kann wie der Service genau von Applikationsprogrammierern im Endeffekt verwendet wird, ist eine Entscheidung für eine Zeit oder Platz begünstigende Datenstruktur eher für einen Spezialfall denkbar. Für eine möglichst allgemein brauchbare Implementierung ist ein Mittelweg vorzuziehen. Die vorhandenen Open-Source Implementierungen des *Property Service* sind diesbezüglich geteilter Meinung: Die MICO [5] Implementierung nutzt einen einfachen *vector* der *Standard Template Library* (STL) zur Speicherung von Properties und ist somit wenig effizient bei einer grösseren Menge an Properties. Die OpenORB [6] Implementierung dagegen verwendet die Java Klasse *Hashtable*. Diese Klasse bietet prinzipiell die Funktionalität an, den Belegungsfaktor zu berücksichtigen, um bei Bedarf zu rehashen. Die OpenORB Implementierung macht jedoch keinen Gebrauch davon.

4.5 PropertySet/PropertySetDef und Nebenbedinungen

Ein zweiter wichtiger Aspekt bei der Implementierung ist das Zusammenspiel zwischen den Servant Implementierung von *PropertySet* und deren Ableitung *PropertySetDef*. Das *PropertySet* bietet die grundlegenden Operationen mit einer Property, während das *PropertySetDef* zusätzlich *PropertyModes* unterstützt, mit denen Zugriffsrechte für jede einzelne Property festgelegt werden können. Da *PropertySetDef* von *PropertySet* erbt, sollte es sich um eine Spezialisierung handeln, die möglichst viel Funktionalität aus geerbten Methoden verwendet. Auf der anderen Seite sollte das *PropertySet* auch nicht zuviel unnötige Komplexität beinhalten, die vom Grundgedanken der Interface-Definition zu stark abweicht bzw. eigentlich in die Spezialisierung gehört.

Dennoch hat sich bei der Implementierung des *Property Service* ergeben, dass letzterer Gedanke der Einfachheit der Implementierung weichen sollte. Von der OMG sind zwei Datentypen zur Speicherung einer Property vorgesehen: *Property* und *PropertyDef*. In der vorliegenden Implementierung kommt nur der Datentyp *PropertyDef* zur eigentlichen Speicherung der Properties zum Einsatz. Das hat zwar den Nachteil, dass dies unsauber im Sinne der strikten logischen Trennung von *PropertySet* und *Property-*

```
1  property_list                : ADT_DICTIONARY[
2                                   COSPROPERTYSERVICE_PROPERTYDEF, STRING]
3  property_type_constraints    : COSPROPERTYSERVICE_PROPERTYTYPES
4  property_constraints         : COSPROPERTYSERVICE_PROPERTYDEFS
```

Listing 4.1: *Die klassenlokalen Objekte vom PropertySet*

SetDef ist, jedoch macht dies den Quelltext weniger aufwendig zu lesen, da statt zwei Datentypen nur einer verwendet wird. Es müssen zusätzlich keine unnötigen Methodenreimplementierungen oder umständliche Workarounds stattfinden, um den anderen Datentyp in der Ableitung zu benutzen. Da mit einer umfassenden Überarbeitung des *Property Service* in absehbarer Zeit nicht zu rechnen ist, kann dies durchaus als legitime Lösung betrachtet werden. Diesen Weg wählte im übrigen auch Leif Jakobsmeier, der die Implementierung des *Property Service* für MICO [5] geschrieben hat.

Es gibt drei klassenlokale Objekte, die vom *PropertySet* und *PropertySetDef* intern zur Verwaltung der Daten verwendet werden (Listing 4.1). Die **property_list** ist das zuvor erwähnte Dictionary, welches sowohl für das *PropertySet* als auch das *PropertySetDef* die Properties in Form von PropertyDef-Datentypen verknüpft mit einem String speichert. Hier werden die Daten gespeichert, um die es sich in diesem Dienst eigentlich handelt.

Das Objekt **property_type_constraints** enthält eine Liste von CORBA-Typecodes, die für das *PropertySet* zugelassen sind. Im Grunde spielen diese als Nebenbedingung nur bei *PropertySets* und *PropertySetDefs* eine Rolle, die durch eine Factory (*PropertySetFactory* oder *PropertySetDefFactory*) mit den Methoden *create_constrained_propertyset* bzw. *create_constrained_propertysetdef* erzeugt wurden. Die Interfaces *PropertySet* und *PropertySetDef* allein bieten keine Möglichkeit diese Nebenbedingungen zu definieren. Der Weg über Factories ist also unumgänglich falls man die oben beschriebene Nebenbedingung nutzen möchte.

Gleiches gilt auch für das letzte der drei klassenlokalen Objekte mit dem Namen **property_constraints**, welches eine Liste von *PropertyDefs* speichert. Die ebenfalls nur in Factories vorhandene Nebenbedingung *allowed_properties* ist in der Spezifikation des *Property Service* leider nur sehr mangelhaft beschrieben. Die Umsetzung in

der vorliegenden MICO/E Implementierung geht davon aus, dass die in den Methoden *create_constrained_propertyset* bzw. *create_constrained_propertysetdef* übergebenen Argumente *allowed_properties* lediglich eine Liste von Properties enthält, in der zwar ein Property- oder PropertyDef-Tupel vorhanden ist, aber das Value-Objekt keine bedeutenden Informationen enthält. Ausschlaggebend für die Nebenbedingung ist vielmehr nur sein CORBA-Typ, der Name und gegebenenfalls der *PropertyModeType*. So wird in der MICO/E Implementierung jeglicher Inhalt des Value-Objektes verworfen und nicht genutzt. Liegt ein *PropertySet* mit einer solchen Einschränkung vor, so kann z.B. nur dann eine Property definiert oder überschrieben werden, wenn in der *property_constraints*-Liste eine Property mit genau diesem Namen, Typ und PropertyModeType vorhanden ist. Zugegebenermassen ist diese Art der Implementierung nicht sonderlich effizient, weil der eigentliche Inhalt vom Value-Objekt verworfen wird und im ungünstigen Fall der Anwender des *Property Service* in diesem viele Daten speichert, die hinterher überflüssig im Speicher des Serverprozesses liegen. Allerdings ist dieser Mangel direkt aus der Spezifikation abgeleitet, wo diese Angelegenheit auch durch Übergabe einer Liste von Properties gelöst wurde anstatt eines eigenen <Name,Typecode> bzw. <Name,Typecode,ModeType>-Tupels. So werden schon bei dem Methodenaufruf zum Erzeugen des *PropertySets* (z.B. create_constrained_propertyset) mehr Daten geschickt als wirklich notwendig sind. Es bleibt daher in der Verantwortung des Anwendungsprogrammierers in der *allowed_properties*-Liste minimal grosse Objekte zu übergeben.

Die Spezialisierung *PropertySetDef* enthält zusätzlich die Funktionalität der *PropertyModes*. Da ein Grossteil des PropertyMode-Handlings bereits in der Klasse *PropertySet* implementiert ist, fällt diese etwas einfacher aus. Im Grunde werden hier nur die für die Klasse *PropertySetDef* neu definierten Methoden implementiert. Jedoch fallen auch diese Methoden kürzer aus, da viele Teile aus dem *PropertySet* wiederverwendet werden können. Ein Beispiel für die beschriebene Vereinfachung ist die Methode *define_property_with_mode*, in der praktisch nur eine neue Exception *Unsupported Mode* ausgewertet und der Rest durch eine Methode in der Superklasse erledigt wird. Listing 4.2 zeigt diese Methode.

```
1  define_property_with_mode (property_name  : STRING;
2                             property_value : CORBA_ANY;
3                             property_mode  :
4                               COSPROPERTYSERVICE_PROPERTYMODETYPE) is
5  do
6      if property_mode.value = CosPropertyService_undefined then
7          raise_unsupportedmode_ex ("define_property_with_mode")
8      end
9      define_prop (property_name, property_value, property_mode)
10 end
```

Listing 4.2: *Vereinfachung der Methoden in PropertySetDef*

4.6 PropertiesIterator/PropertyNamesIterator

Ein weiterer interessanter Aspekt der Implementierung des *Property Service* ist das Zusammenwirken des *PropertyNamesIterator* bzw. *PropertiesIterator* mit dem ADT_ITERATOR des intern benutzten Dictionaries. Wie bereits in Kapitel 3 erwähnt, handelt es sich beim *Property Service* im Gegensatz zum ADT_ITERATOR der ADT Bibliothek von MICO/E nicht um polymorphe Iteratoren. In der ADT Bibliothek gibt es den externen Iterator vom Typ ADT_ITERATOR, der über die Datenstruktur einer Klasse iteriert, die von ADT_TRAVERSABLE abgeleitet ist. Weil die Iteratoren des *Property Service* inkompatibel zu denen der ADT Bibliothek sind, wurde eine relativ naheliegende Lösung gewählt: Wird ein *Property Service* Iterator erzeugt, so bekommt er in der Erzeugungsprozedur Referenzen auf das ADT_DICTIONARY und einen ADT_ITERATOR übergeben, welcher auf das erste Element des Restes zeigt. Listing 4.3 zeigt anhand der Methode *get_all_property_names* wie eine typische Methode der PropertySets aussieht, die einen Iterator auf einen Rest zurückgibt. Zeilen 9-19 erzeugen eine Liste von Propertynamen, indem vom Anfang des Dictionaries bis *how_many* mit dem ADT_ITERATOR *it* iteriert und jeweils der Propertyname in diese eingehängt wird (Zeile 16). Ist die Liste in Zeile 20 fertig erzeugt, so steht der ADT_ITERATOR *it* auf dem ersten Element des potentiellen Restes. Daher wird in Zeilen 22-25 entsprechend ein *PropertyNamesIterator* mit den oben beschriebenen Argumenten erzeugt.

```
1  get_all_property_names (how_many       : INTEGER;
2                          property_names : COSPROPERTYSERVICE_PROPERTYNAMES;
3                          rest           :  CORBA_REF[
4                            COSPROPERTYSERVICE_PROPERTYNAMESITERATOR]) is
5  local
6      it    : ADT_ITERATOR
7      i     : INTEGER
8      pnit  : COSPROPERTYSERVICE_PROPERTYNAMESITERATOR_IMPL
9  do
10     -- fill property_names
11     from
12         it := property_list . iterator
13         property_names.make_default
14     until
15         it . finished or else i >= how_many
16     loop
17         property_names.append (property_list.key (it))
18         it . forth
19         i := i + 1
20     end
21
22     -- put left property_names in rest
23     if not it . finished then
24         create pnit.make (it, property_list)
25         rest .make_with_item (pnit.this)
26     end
27 end
```

Listing 4.3: *Beispiel zum Umgang mit Iteratoren*

4.7 Der Property Service Daemon

Wie bereits in Kapitel 3 beschrieben, gibt es zwei Möglichkeiten den Property Service zu nutzen: Durch Ableitung eines Interfaces von *PropertySet* bzw. *PropertySetDef* oder durch Erzeugung von *Propertysets* mit einer Factory. Um eine solche Factory über den ORB verfügbar zu machen, ist es notwendig, einen sogenannten Daemon zu schreiben. Es handelt sich dabei um nichts anderes als ein simples CORBA-Programm, das kaum komplexer ist als die Demoprogramme, die MICO/E beiliegen. Es erledigt im Grunde nur folgende Aufgaben:

- ORB und POA Standardinitialisierung

- Erzeugung der fertig implementierten *PropertySetFactory* und *PropertySetDefFactory* Objekte
- Anmeldung der *PropertySetFactory* und *PropertySetDefFactory* Objekte beim Naming Service
- POA Aktivierung und Einsetzen der ORB Event Loops

Listing 4.4 zeigt den Kernteil des *eifpsfd* (Eiffel Property Service Factory Daemon), welcher die oben beschriebene Aufgaben erledigt. Zeilen 12-15 sind Standardformulierungen, die man in praktisch jedem CORBA-Programm wiederfindet. Anhand der Argumente in der Kommandozeile wird der ORB und POA-Manager initialisiert. Zeilen 18-19 erzeugen jeweils die Factory-Objekte während sie in Zeilen 22-23 beim *Naming Service* angemeldet werden. Die *advertise_object* Methode ist selbstgeschrieben und hier nicht mit abgedruckt. Zeilen 27-29 aktivieren den POA und setzen den ORB in seinen Event Loop bis ein Userabbruch erfolgt.

Der Client muss zur Benutzung des *eifpsfd* lediglich eine Referenz auf die Factory-Objekte von dem *Naming Service* anfordern und diese benutzen, um neue PropertySets zu erzeugen.

4.8 Testlauf

Da die Funktionsfähigkeit des *Property Service* nicht dem Zufall überlassen sein sollte, ist als Nebenprodukt ein relativ umfangsreiches Testprogramm entstanden, das halbwegs systematisch die Funktionalitäten des *Property Service* durchtestet. Es ist eher zum Testen der grundlegenden Fähigkeiten des Dienstes ausgelegt und versucht nicht auf Effizienz, Belastung oder Fehlerfreiheit zu testen. Zu diesem Zweck wurde ein minimales IDL-Interface entworfen, welches von PropertySet erbt und nur eine *SayHello* Methode anbietet. Dieses ist notwendig, um die Funktionalität eines *PropertySets* anhand einer Ableitung zu testen. Zum Test der *Property Service* Factories ist dieses nicht notwendig. Listing 4.5 zeigt dieses Interface.

Der dazu passende Server unterscheidet sich kaum von der bereits in Listing 4.4 vorgestellten Standardformulierung. Die Implementierung des Interfaces ist auch minimal gehalten. In der *SayHello* Methode gibt der Server lediglich selbst einen typischen "Hello World"-String aus. Letztendlich soll hier nur die geerbte Funktionalität getestet werden, weshalb der Interface-Inhalt an sich unwichtig ist.

```
1  make is
2  local
3      err      : BOOLEAN
4      obj      : CORBA_OBJECT
5      poa      : PORTABLESERVER_POA
6      mgr      : PORTABLESERVER_POAMANAGER
7      pf_obj   : COSPROPERTYSERVICE_PROPERTYSETFACTORY_IMPL
8      pdf_obj : COSPROPERTYSERVICE_PROPERTYSETDEFFACTORY_IMPL
9  do
10     if not err then
11         -- Initialize the ORB and POA
12         orb := ORB_init (argument_array, "mico-local-orb")
13         obj := orb. resolve_initial_references ("RootPOA")
14         poa := PortableServer_POA_narrow (obj)
15         mgr := poa.the_POAManager
16
17         -- Create an implementation object
18         create pf_obj.make
19         create pdf_obj.make
20
21         -- Now make our object available to clients
22         advertise_object (pf_obj.this, "PropertySetFactory")
23         advertise_object (pdf_obj.this, "PropertySetDefFactory")
24
25         -- Activate the POA and put the
26         -- ORB in its event loop.
27         mgr.activate
28         orb.run
29         poa.destroy (true, true)
30     else
31         -- ...
32 rescue
33     -- ...
34 end
```

Listing 4.4: *Der Kernteil des eifpsfd*

```
1  #include <CosPropertyService.idl>
2
3  interface  demo : CosPropertyService::PropertySet {
4      void SayHello();
5  };
```

Listing 4.5: *Test Interface (demo.idl)*

Der Testclient dagegen ist schon bedeutend aufwendiger und gliedert sich in drei Teile:

1. Test der *PropertySetFactory*
2. Test der *PropertySetDefFactory*
3. Test der im Interface *demo* geerbten Funktionalität

Jeder dieser drei Tests untergliedert sich wiederum in detailliertere Tests. Dabei wird Test 1 in folgende Schritte aufgeteilt:

- Erzeugen eines *PropertySets* und constrained *PropertySets*
- Einfügen von Properties verschiedener Typen
- Testen der get-Methoden
- Testen der Delete-Methoden

Den Erfolg bzw. Misserfolg und Exceptions kann man über die Standardausgabe verfolgen. Test 2 verhält sich analog zu Test 1. Jedoch werden hier Properties mit allen vorhandenen Modes erzeugt, so dass man sich von dem erwarteten Verhalten bzgl. der Zugriffseinschränkung überzeugen kann.

Zuletzt wird in Test 3 das Verhalten des *PropertySets* getestet, indem seine Funktionalität durch Ableitung gebraucht wird. Die Tests entsprechen genau denen von Test 1, bis auf die Tatsache, dass hier kein neues PropertySet erzeugt wird. Der Test, ob ein von *PropertySetDef* abgeleitetes Interface ebenso fehlerfrei funktioniert, fehlt. An dieser Stelle wird einfach davon ausgegangen, dass dieses problemlos funktionieren wird,

solange der Test mit Ableitung von *PropertySet* sowie die Tests auf *PropertySets* und *PropertySetDefs*, die durch Factories erzeugt wurden, einwandfrei funktionieren.

Obwohl der Testlauf halb systematisch ist, so muss man jedoch ganz klar erkennen, dass er nicht hinreichend ist, um auf tatsächliche Korrektheit der Implementierung zu testen. Für eine zuverlässigere Aussage über die Lauffähigkeit der Implementierung wäre eine komplette Testumgebung, die z.B. auf *getest* des GOBO Eiffel Projektes [2] aufsetzt, wünschenswert.

5 Zusammenfassung

Diese Bachelorarbeit hat sich mit der Implementierung des OMG spezifizierten *Property Service* beschäftigt. Es wurden in Kapitel 2 die Grundlagen von CORBA erarbeitet, um dem Leser ein oberflächliches Verständnis der dem Dienst zugrundlegenden Technologie zu vermitteln. CORBA ist bekannt für seinen Umfang und die steile Lernkurve für Anwender, die sich erstmals mit der Thematik befassen. Deshalb ist dieses Kapitel kurz gefasst und beschränkt sich nur auf das Wesentliche.

In Kapitel 3 wurde der *Property Service* in seiner Funktionalität umrissen. Dazu gehören die kleinsten Einheiten *Property* und *PropertyDef* sowie deren Container *PropertySet* und *PropertySetDef*, wobei *PropertyDef* und *PropertySetDef* auszeichnet, dass sie die zusätzliche Funktionalität der *PropertyModes* besitzen, die grundlegende Zugriffsrechte ermöglichen. Weiterhin wurden in Bezug auf die Fehlerbehandlung die *Property Service* Exceptions und deren Bedeutung besprochen. Über die Factory Objekte, *PropertySetFactory* und *PropertySetDefFactory*, kann der Nutzer des *Property Service* PropertySets erzeugen und deren Referenzen beziehen.

Kapitel 4 befasst sich mit der eigentlichen Implementierung des *Property Service* mit MICO/E als Grundlage. Nach einer Orientierung in der vom IDL-Compiler erzeugten Umgebung wurde Hashing und das Dictonary als Datenstruktur besprochen und erörtert, warum es für den Zweck des *Property Service* geeignet ist. Bei dem Zusammenspiel zwischen *PropertySet* und *PropertySetDef* wurde ein Kompromiss eingegangen, der die Komplexität in die Implementierung des Interfaces *PropertySet* verschiebt, was eine relativ günstige Realisierung des Interfaces *PropertySetDef* zur Folge hat. Weiterhin wurde die interne Handhabung der *PropertyModes* und Nebenbedingungen erläutert. Dabei kam eine kleine Schwäche der OMG Spezifikation des *Property Service* zum Vorschein, die durch Übergabe von unnötig viel Daten in den Übergabeargumenten der Metho-

den *create_constrained_propertyset* und *create_constrained_propertysetdef* begründet ist. Das Zusammenspiel der inkompatiblen Iteratoren stellte sich als relativ einfach heraus. Das Problem wurde durch eine Durchreichung von Objektreferenzen bei Erzeugung der *Property Service* Iteratoren gelöst. Die Implementierung des *eifpsfd* - dem Property Service Daemon, der neue PropertySets erzeugt und deren Referezen zurückgibt erwies sich als problemlos. Seine Komplexität liegt lediglich in der Implementierung der Interfaces *PropertySetFactory* und *PropertySetDefFactory*. Das Testprogramm, das als Nebenprodukt des *Property Service* entstanden ist, zeigt eine besonders einfache Art und Weise, halbwegs systematisch die Methoden der Interfaces zu testen. Es wurde allerdings betont, dass es sich nicht um eine ordentliche Testumgebung handelt, die bis zu einem gewissen Grad Funktionalität gewährleisten kann.

Verbesserungswürdig in der Implementierung ist sicherlich die Testumgebung, die man noch systematischer und ausführlicher gestalten könnte und auch verbessern sollte, wenn man einen produktiven Einsatz des Dienstes gewährleisten möchte. Weiterhin wurde die Möglichkeit angesprochen, dass bei Erzeugung eines *PropertySets* mit Nebenbedingungen eine Liste von Properties gespeichert wird, an denen Value-Objekte hängen, die unbenutzt verworfen werden. Um hier vor einem falschen Gebrauch zu schützen, wäre es sinnvoll, serverseitig nur ein Tupel bestehend aus Name, CORBA-Typecode und gegebenenfalls PropertyMode zu speichern. Zuletzt ist die vorliegende Implementierung mit der gewählten Datenstruktur des Dictionary ein Kompromiss zwischen Speicherverbrauch und Performance. Diese mag nicht für jeden Zweck effizient genug sein. Es wäre denkbar, verschiedene Implementierungen des *Property Service* für verschiedene Zwecke zu realisieren. Ob sich solch ein Aufwand lohnt, hängt sicherlich von der Rolle ab, die der Dienst in der realen Anwendung spielt.

Danksagung

An erster Stelle gilt der Dank Herrn Prof. Dr. Robert Switzer, der mich betreut und diese Arbeit ermöglicht hat. Ganz besonders hervorzuheben sind weiterhin Paul Lettich und Roman Asper für die tolle Teamarbeit bei der Implementierung. Weiterhin danke ich meinem Zweitkorrektor Herrn Prof. Dr. Grabowski und meinen Korrekturlesern Gisa Zeiß, Daniel Zeiß und Remko Ricanek. Zuletzt möchte ich dem gesamten MICO/E Team meinen Respekt für die beachtliche Arbeit und Leistung ausdrücken, die in MICO/E steckt und die Grundlage für diese Arbeit darstellt.

Abkürzungsverzeichnis

ADT	Abstract Data Type
BOA	Basic Object Adapter
CORBA	Common Object Request Broker
DAEMON	Disk And Execution Monitor
DII	Dynamic Invocation Interface
DSI	Dynamic Skeleton Interface
GIOP	General InterORB Protocol
IDL	Interface Definition Language
IIOP	Internet InterORB Protocol
IOR	Interoperable Object Reference
IR	Interface Repository
MICO	MICO Is CORBA
OMA	Object Management Architecture
OMG	Object Management Group
ORB	Object Request Broker
POA	Portable Object Adapter
STL	Standard Template Library

Literaturverzeichnis

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.

[2] Gobosoft. GOBO Eiffel Project. `http://www.gobosoft.de`.

[3] Jon Siegel. *CORBA: Fundamentals and Programming*. John Wiley, 1996.

[4] Kay Römer. MICO - MICO is CORBA vorgelegt an der Johann Wolfgang Goethe-Universität, Frankfurt am Main. `http://www.vsb.cs.uni-frankfurt.de/roemer/papers/diplom.ps.gz`, 1999.

[5] Kay Römer, Arno Puder, Frank Pilhofer, Andreas Schultz, ObjectSecurity Ltd., and others. MICO - MICO IS CORBA. `http://www.mico.org`.

[6] Michael Rumpf, Stephen McConnell, Jesper Pedersen, Shawn Boyce, J. Scott Evans, Viacheslav N. Taranin, Chris Wood, and others. The Community OpenORB Project. `http://openorb.sourceforge.net`.

[7] Michi Henning and Steve Vinoski. *Advanced CORBA Programming with C++*. Addison-Wesley Professional, February 1999.

[8] Object Management Group (OMG). *Bibliographic Query Service*.

[9] Object Management Group (OMG). *Clinical Observations Access Service*.

[10] Object Management Group (OMG). *Control and Management of A/V Streams specification.*

[11] Object Management Group (OMG). *Internationalization and Time Facility.*

[12] Object Management Group (OMG). *Property Service Specification*, April 2000.

[13] Object Management Group (OMG). *Relationship Service Specification*, April 2000.

[14] Object Management Group (OMG). *Common Object Request Broker Architecture (CORBA/IIOP) 3.02*, 2 December 2002.

[15] Prof. Dr. Robert Switzer. Mico/E. `http://www.math.uni-goettingen.de/micoe`.

[16] Prof. Dr. Robert Switzer. Verteilte Systeme und CORBA, 2001. Skript zur Vorlesung Wintersemester 2001/2002.

[17] Robert Sedgewick. *Algorithmen in C++.* Addison-Wesley, 1992.

[18] Sumedh Mungee, Nagarajan Surendran, and Douglas C. Schmidt. The Design and Performance of a CORBA Audio/Video Streaming Service. Technical report, Department of Computer Science, Washington University.

[19] Thomas J. Mowbray and William A. Ruh. *Inside CORBA: Distributed Object Standards and Applications.* Addison-Wesley, 1997.