Diplomarbeit

Entwurf und Implementierung eines verteilten objektorientierten Fehlerbehandlungssystems in einem heterogenen Systemumfeld

Praxisarbeit bei der Hamburg-Mannheimer Versicherungs-AG

August 1998

Andreas Felten
Lohbekstieg 10
22529 Hamburg
Matr.-Nr: 4555777

Erstbetreuer:
Prof. Heinz Züllighoven
Universität Hamburg
Fachbereich Informatik
Arbeitsbereich Softwaretechnik

Zweitbetreuer:
Prof. Winfried Lamersdorf
Universität Hamburg
Fachbereich Informatik
Arbeitsbereich Verteilte Systeme

Ich bestätige hiermit, daß ich die hier vorgelegte Diplomarbeit ausschließlich mit den aufgeführten Mitteln erstellt habe.

Danksagung
Die Danksagung gilt allen Personen, die mich bei der Erstellung der vorliegenden Diplomarbeit unterstützt haben. Vor allem möchte ich Herrn Tobias Grahl, meinem Betreuer in der Hamburg-Mannheimer Versicherungs-AG, für viele anregende Diskussionen und das Korrekturlesen der Arbeit danken. Außerdem möchte ich meinem Kommilitonen Martin Lippert danken, mit dem ich interessante Diskussionen über Architekturkonzepte zur Kapselung von Middleware geführt habe. Ebenfalls herzlichen Dank an meine Projektgruppe vom Projektseminar CORBA, WWW, Java: Verteilte Anwendungsentwicklung am Beispiel von KIS[1] des Wintersemesters 1997/98 namentlich Holger Breitling, Hilger Müller und Timmy Blank für die Unterstützung der Integration von CORBA in das JWAM-Framework.

[1] Krankenhausinformationssystem

Inhaltsverzeichnis

Legende ... vi

1 Einleitung ... 1

2 Fehlerbehandlung ... 4

2.1 Terminologie ... 5

2.2 Ausnahmebehandlung ... 7

2.3 Theoretischer Hintergrund ... 9

3 Das Vertragsmodell von Bertrand Meyer ... 12

3.1 Vor- und Nachbedingungen ... 12

3.2 Invarianten ... 13

3.3 Weitere Konstrukte mit Zusicherungen ... 14

3.4 Vererbung von Bedingungen und Invarianten ... 14

3.5 Umsetzung des Vertragsmodells in anderen Programmiersprachen ... 15

4 Kontext und Anforderungen des Fehlerbehandlungssystems ... 18

4.1 Das Java-Forschungsprojekt der HM ... 18

4.2 Anforderungen an das Fehlerbehandlungssystem ... 20

4.3 Anforderungen an die Middleware ... 21

5 Message-orientierte Middleware am Beispiel von MQSeries ... 23

5.1 Einführung in das Messaging und Queuing ... 23

5.2 Die Komponenten von MQSeries ... 25

5.2.1 Message ... 25

5.2.2 Queue-Manager ... 26

5.2.3 Message-Queues ... 28

5.2.3.1 Lokale Queues ... 29

5.2.3.2 Entfernte Queues ... 29

5.2.3.3 Transmission-Queues ... 29

5.2.4 Channels ... 30

5.2.5 Message Queuing Interface (MQI) ... 32

5.3 Message Routing ... 33

5.4 MQSeries Client für Java ... 34

5.4.1 Architektur ... 35

5.4.2 Die Java-Klassenbibliothek ... 35

5.5 Die Architektur des Fehlerbehandlungssystems ... 36

6 CORBA ... **38**

6.1 IDL ... **39**

6.2 Die Struktur von CORBA ... **41**

6.2.1 Object Request Broker ... 41

6.2.2 Komponenten des statischen CORBAs ... 42

6.2.2.1 Client ... 42

6.2.2.2 Client IDL Stubs ... 43

6.2.2.3 Object Adaptor ... 43

6.2.2.4 Implementation Repository ... 44

6.2.2.5 Server IDL Stub ... 44

6.2.2.6 Serverobjekt ... 45

6.2.3 Komponenten des dynamischen CORBAs ... 45

6.2.3.1 Interface Repository ... 46

6.2.3.2 Dynamic Invocation Interface ... 46

6.2.3.3 Dynamic Skeleton Interface ... 46

6.3 Objekt Management Architektur ... **47**

6.3.1 Object Services ... 47

6.3.2 Common Facilities ... 48

6.3.3 Domain Interfaces ... 48

6.3.4 Application Objects ... 49

6.4 Visibroker for Java - eine CORBA-Implementierung ... **49**

6.5 Integration von CORBA ins JWAM-Framework ... **50**

7 Entwurf und Implementierung des Fehlerbehandlungssystems ... **56**

7.1 Die Ausnahmen ... **56**

7.2 Das PersistenzInterface ... **57**

7.3 Das Vertragsmodell ... **58**

7.4 Die mehrsprachige Fehlertexterfassung ... **60**

7.4.1 Architektur ... 60

7.4.2 Benutzung ... 62

7.5 Besonderheiten bei der Implementation in Java ... **64**

7.5.1 Java und der Acht-Punkte-Katalog von Dony ... 64

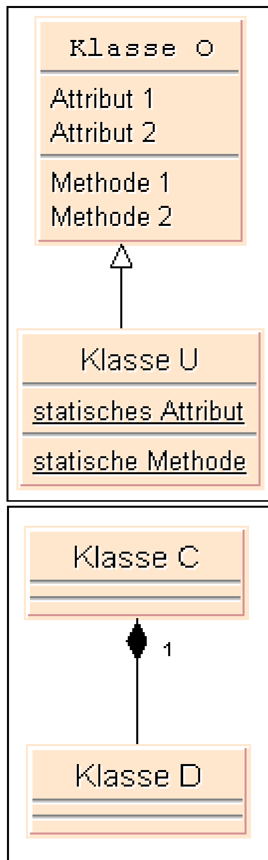
7.5.2 Nachteile ... 65

7.5.3 Vorteile ... 65

8 Abschlußdiskussion des Fehlerbehandlungssystems und der Middleware-Systeme ... 66
8.1 Überprüfung der Anforderungen an das Fehlerbehandlungssystem ... 66
8.2 Vergleich der beiden Middleware-Systeme ... 67
8.2.1 Erfüllung des Anforderungskatalogs ... 67
8.2.1.1 Middlewarekriterien ... 67
8.2.1.2 Unterstützung der Objektorientierung ... 67
8.2.1.3 Kompatibilität der Produkte zu anderen Middleware-Produkten ... 68
8.2.1.4 Persistenz ... 68
8.2.1.5 Möglichkeit der asynchronen Kommunikation ... 69
8.2.1.6 Erlernbarkeit ... 69
8.2.1.7 Performance ... 69
8.2.1.8 Sicherheit ... 70
8.2.1.9 Praxiseinsatz ... 70
8.2.1.10 Verfügbarkeit in der HM ... 70
8.2.2 Entscheidung zugunsten eines der Middleware-Systeme ... 70
9 Zusammenfassung und Ausblick ... 71
10 Anhang ... 73
10.1 MQSeries-Code für das Fehlerbehandlungssystem ... 73
10.2 MQSeries-Mapping-Tabelle ... 74
10.3 Stack-Spezifikation ... 74
Abkürzungsverzeichnis ... 75
Abbildungsverzeichnis ... 77
Literaturverzeichnis ... 79

Legende

In der vorliegenden Arbeit wurde die UML (Unified Modelling Language) benutzt. Die benutzten Komponenten werden hier kurz erläutert:

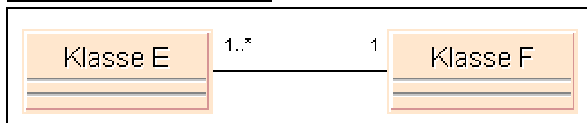


Die Klasse O besitzt die Attribute *Attribut 1* und *Attribut 2*, sowie die Methoden *Methode 1* und *Methode 2*.

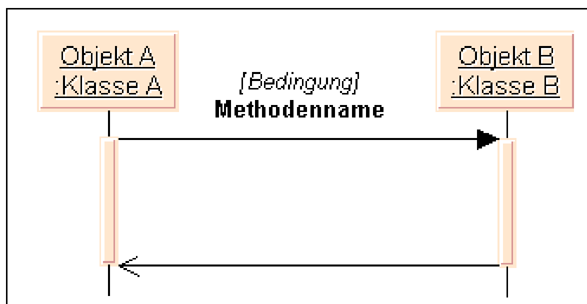
Die Klasse U erbt von Klasse O.

Die Klasse U besitzt ein *statisches Attribut* und eine *statische Methode*.

Die Klasse C aggregiert die Klasse D.



Ein Exemplar der Klasse E ist mit mindestens einem Exemplar der Klasse F assoziiert. Ein Exemplar der Klasse F ist mit genau einem Exemplar der Klasse E assoziiert.



Das Objekt A, welches ein Exemplar der Klasse A ist, ruft die Methode *Methodenname* beim Objekt B, welches ein Exemplar der Klasse B ist, auf, wenn die angegebene Bedingung erfüllt ist.

1 Einleitung

Bei dieser Diplomarbeit handelt es sich um eine Praxisarbeit in der Hamburg-Mannheimer Versicherungs-AG (HM), einer großen Versicherung mit Hauptsitz in Hamburg. Seit dem Sommer 1997 ist sie Teil der ERGO-Versicherungsgruppe, zu dem auch die Victoria, die D.A.S. (Deutsche Automobil-Schutz) und die DKV (Deutsche Krankenversicherung) gehören.

In den Sommersemesterferien 1995 habe ich ein Praktikum bei der HM absolviert. Dabei lernte ich den Großrechner mit MVS (Multiple Virtual Storage, Betriebssystem von IBM) und JCL (Job Control Language), sowie die Programmiersprache PL/1 kennen. Interessant ist das Konzept der Fehlerbehandlung, welches in der HM eingesetzt wird, da die Programmiersprache PL/1 selbst keine ausreichende Fehlerbehandlung zur Verfügung stellt. Bei der Fehlerbehandlung in der HM wird zwischen Batch- und TP-Programmen unterschieden: Um den Grund von Programmabstürzen bei Batch-Programmen herauszufinden, wurde ein Programm namens Abend-Aid gekauft, das der Analyse von Programmabbrüchen dient. Abend-Aid wandelt den Fehlercode, mit dem der Großrechner die Programmausführung abbricht, in eine Fehlermeldung um und versucht, die Codezeile herauszufinden, in der der Fehler entstanden ist. Dabei wird der Prozedurname und entweder der Offset der nächsten ausführbaren Instruktion oder die Zeilennummer angegeben. Mit diesen Informationen und der vom Compiler erzeugten Umwandlungsliste, die die Zeilennummern bzw. Offsets der einzelnen Instruktionen enthält, kann der Programmierer die Absturzstelle herausfinden und beseitigen.

Die Fehleranalyse im TP-Bereich ist deutlich schwieriger. In der Hamburg-Mannheimer Versicherungs-AG existiert ein selbst implementierter Task-Manager. PL/1 kommuniziert mit diesem über Makro-Aufrufe, über die der Task-Manager die Kontrolle erhält. Jedem PL/1-Programm werden verschiedene Bereiche (Areas) zugeteilt, die die Variablen enthalten. Es gibt lokale und verteilte Areas. Letztere werden für die Interprozeßkommunikation verwendet. Bei Programmabbrüchen steht nur der Fehlercode und die verursachende PKN (persönliche Kennummer) zur Verfügung; selten hingegen der nächste ausführbare Befehl. Von mehreren Programmierern, mit denen ich ein Interview führte, erfuhr ich, daß es zwar einen Debugger gibt, dieser aber nur äußerst selten benutzt wird. Die Programmierer behelfen sich zur Laufzeit mit dem Speichern von Variableninhalten in einer Datei, wenn sie auf Fehlersuche gehen.

Um den Programmierern ihre Programmfehler, die während der Produktion auftreten, mitzuteilen, werden die zur Verfügung stehenden Informationen ausgedruckt und per Hauspost an den verantwortlichen Programmierer geschickt. Bei fachlichen Fehlern wird der Programmierer direkt von der jeweiligen Fachabteilung angerufen.

Weil die HM an neuen Technologien stark interessiert ist, wurde 1996 ein Großprojekt namens TOPAS (Top-Anwendungslandschaft) gestartet. Dieses beschäftigt sich mit der Implementation von Kernanwendungen für die Fachabteilungen (z.B. im Schadensbereich und bei der Lebensversicherung) in Smalltalk auf PCs. Innerhalb von TOPAS werden Themen wie Objektorientierung, Design Patterns, Analyse Patterns, Architekturkonzepte (z.B. MVC), Frameworks (z.B. VisualBanker von IBM) und vieles mehr diskutiert. Im Sommer 1997 wurde ein Forschungsprojekt mit dem Ziel, die Programmiersprache Java zu evaluieren, gestartet. Ziel meiner Diplomarbeit ist es, für dieses Java-Forschungsprojekt ein verteiltes objektorientiertes Fehlerbehandlungssystem zu entwickeln.

Dieses Fehlerbehandlungssystem soll unabhängig von dem Ausführungsort eines Programms auftretende Fehler erkennen, möglichst viele Informationen über den Fehler sammeln und diese entweder zentral speichern oder als Bildschirmdialog dem Benutzer anzeigen. Da es verteilt in einem heterogenen Systemumfeld eingesetzt werden soll, wird eine Middleware benötigt, die von den heterogenen Hardware- und Softwareplattformen und Protokollen abstrahiert und für das Fehlerbehandlungssystem eine einheitliche Basis darstellt. In dieser Arbeit untersuche ich dazu MQSeries, ein message-orientiertes Middleware-Produkt von IBM, und die CORBA-Implementation Visibroker von Inprise (ehemals Visigenic). Eine wichtige Fragestellung ist in diesem Zusammenhang, welche der beiden Middleware-Systeme für die Implementation des Fehlerbehandlungssystems geeigneter ist.

Ein weiteres Ziel ist die frühzeitige Erkennung von Fehlern. Um dieses Ziel zu erreichen, habe ich das Vertragsmodell von Bertrand Meyer in Java implementiert. Um der Problematik der zentralen Speicherung der Fehler und der Ausgabe der Fehler in einem Bildschirmdialog gerecht zu werden, habe ich jeweils eine Ausnahmeklasse implementiert, die diese Funktionalität anbietet, wenn die Ausnahme zur Laufzeit einem von mir implementierten Ausnahmebearbeiter übergeben wird.

Fehler, die zentral gespeichert werden sollen, werden Programmfehler genannt. Sie haben gravierende Konsequenzen auf den Programmablauf und können zu unvorhersehbaren Effekten führen. Fehler, die in einem Bildschirmdialog dem Anwender mitgeteilt werden, heißen Benutzerfehler. Solche Fehler haben keine gravierenden Auswirkungen auf den Programmablauf. Die Verarbeitung der beiden Fehlerarten stellt ein Standardverhalten dar, welches durch den Anwendungsprogrammierer bei Bedarf leicht geändert werden kann. Auf diese Art wird einerseits ein einheitliches Vorgehen bei der Reaktion auf Fehler angeboten, andererseits bleibt die Flexibilität der Anwendungsentwickler gewahrt, falls ein konkretes Problem eine individuelle Fehlerbehandlung erfordert.

Zusätzlich zur Lösung der von der HM gestellten Anforderung habe ich mich wissenschaftlich damit auseinandergesetzt, was in der gängigen Literatur unter Fehlerbehandlung verstanden wird. Von den Anfängen der Fehlerbehandlung in den siebziger Jahren mit J.B. Goodenough bis heute ist eine Vielzahl von wissenschaftlichen Artikeln geschrieben worden. Dabei gibt es teilweise sehr unterschiedliche Definitionen von Begriffen rund um die Fehlerbehandlung. Die Entwickler von Programmiersprachen haben unterschiedliche Modelle der Fehlerbehandlung in ihre Sprachen integriert, so daß die Mächtigkeit in diesem Punkt von Sprache zu Sprache sehr unterschiedlich ist.

Die Verbindung vom praktischen und theoretischen Teil meiner Arbeit ist fließend. Während die Anforderungen an das Fehlerbehandlungssystem und die Implementation zum praktischen Teil gehören, stellt die Literaturarbeit über Fehlerbehandlung und Vertragsmodell den theoretischen Teil der Arbeit dar. Dazwischen liegt die Thematik der Middleware, die sowohl einen theoretischen als auch einen praktischen Aspekt beinhaltet. Insgesamt läßt sich sagen, daß der theoretische Teil meiner Diplomarbeit deutlich mehr Aufwand gekostet hat als der praktische Teil und damit auch den größeren Teil in der vorliegenden Arbeit ausmacht.

Meine Diplomarbeit gliedert sich wie folgt:

Kapitel 2 besteht aus einer Einführung in die Fehlerbehandlung. Es werden wichtige Definitionen von Begriffen wie Störung, Fehler, Fehlverhalten, Detektor und Ausnahme aus der gängigen Literatur diskutiert. Zusätzlich wird das Konzept der Ausnahmebehandlung

vorgestellt. Außerdem wird dargestellt, welche Zustände eine Softwarekomponente besitzt und wie sie sich im Fehlerfall verhalten soll.

Um zur Laufzeit eines Softwareprodukts Fehler entdecken zu können, werden Detektoren benötigt. Das dieser Diplomarbeit zugrunde liegende Modell zur Fehlerentdeckung ist das Vertragsmodell von Bertrand Meyer. Dieses Konzept, das Meyer in die von ihm entwickelte Programmiersprache Eiffel integriert hat, besagt, daß zwischen einer Klasse und ihrem Aufrufer ein Vertrag geschlossen wird, der die Rechte und Pflichten beider Seiten festlegt. In *Kapitel 3* wird das Vertragsmodell erläutert und erste Ansätze zur Implementation in anderen Programmiersprachen dargelegt.

Da es sich bei dieser Diplomarbeit um eine Praxisarbeit handelt, wird in *Kapitel 4* kurz auf den Kontext der Arbeit bei der Hamburg-Mannheimer Versicherungs-AG eingegangen. Zusätzlich werden Anforderungen aufgestellt, denen das Fehlerbehandlungssystem genügen muß. Weil es sich beim Fehlerbehandlungssystem um ein verteiltes Umfeld handelt, wird eine Middleware benötigt, die von den unterschiedlichen Plattformen abstrahiert. Dieses Kapitel beschreibt Anforderungen, die eine für das Fehlerbehandlungssystem geeignete Middleware erfüllen sollte.

In *Kapitel 5* wird die message-orientierte Middleware (MOM) am Beispiel von MQSeries, einem IBM-Produkt, vorgestellt. Die rechnerübergreifende Kommunikation findet bei MOM-Produkten über Messages statt. Die kommunizierenden Anwendungen benutzen dazu Queues, in die sie Messages schreiben und aus denen sie Messages lesen können.

Um eine alternative Middleware zu untersuchen, wird in *Kapitel 6* CORBA vorgestellt. CORBA ist eine rein objektorientierte Middleware, die in den letzten Jahren deutlich an Bedeutung gewonnen hat. Als CORBA-Implementation wurde Visibroker von Inprise (ehemals Visigenic) eingesetzt.

In *Kapitel 7* wird die der Diplomarbeit zugrunde liegende Java-Implementation erläutert. Es werden Ausnahmen vorgestellt, die ein Defaultverhalten für Anwendungsprogrammierer zur Verfügung stellen, wie z.B. Speicherung in einer zentralen Queue oder Anzeige von Fehlermeldungen in einem Fenster. Außerdem wird anhand eines Klassendiagramms erläutert, wie das Vertragsmodell implementiert wurde. Weiterhin wird in diesem Kapitel überprüft, inwieweit die aufgestellten Anforderungen an das Fehlerbehandlungssystem erfüllt wurden.

Es stellt sich nun die Frage, welche Middleware für die Implementation des Fehlerbehandlungssystems geeigneter ist. In *Kapitel 8* wird geprüft, welche der aufgestellten Anforderungen von MQSeries und Visibroker erfüllt werden. Aufgrund dieses Vergleichs wird entschieden, welches der beiden Produkte für die Fehlerbehandlung besser geeignet ist.

Kapitel 9 beinhaltet eine Zusammenfassung der Arbeit und gibt Anregungen für weitere Untersuchungen.

Der *Anhang* enthält den MQSeries-Code, der für das Fehlerbehandlungssystem erforderlich war, eine MQSeries-Mapping-Tabelle, sowie die Spezifikation des Stacks, denn der Stack stellt das Anwendungsbeispiel dieser Arbeit dar.

Die Arbeit schließt mit Abkürzungsverzeichnis, Abbildungsverzeichnis und Literaturverzeichnis.

2 Fehlerbehandlung

Ein wesentlicher Teil des Softwareentwicklungsprozesses ist die Auseinandersetzung mit der Fehlerbehandlung. Nur wenn bei einem Softwareprodukt großer Wert auf die Fehlerbehandlung gelegt wird, kann Software als zuverlässig bezeichnet werden. In jeder Situation und in jedem Zustand muß die Software ein definiertes Verhalten zeigen. Um dies zu erreichen, ist es wichtig zu verstehen, daß ein Fehlerbehandlungsmechanismus kein Aufsatz auf eine vorhandene Software ist, sondern von Beginn des Softwareentwicklungsprozesses an in die entstehende Software integriert werden muß. Nur auf diese Weise kann eine hohe Softwarequalität gewährleistet werden.

Ein wichtiges Ziel der objektorientierten Softwareentwicklung ist die Wiederverwendbarkeit von Softwarekomponenten. Eine wesentliche Voraussetzung dafür ist die Zuverlässigkeit der Komponenten. Bertrand Meyer definiert Zuverlässigkeit als Korrektheit und Robustheit. Korrektheit ist die Fähigkeit einer Softwarekomponente, ihre Spezifikation korrekt zu erfüllen. Robustheit bedeutet, daß eine Softwarekomponente auch unter außergewöhnlichen Bedingungen funktioniert. Um die Zuverlässigkeit zu gewährleisten, muß eine vollständige Spezifikation der Softwarekomponente entwickelt und diese fehlerfrei implementiert werden. Dieses Ziel ist jedoch in der Realität kaum erreichbar: Jedes Softwareprodukt enthält nach Jürgen Jentsch noch mindestens einen Fehler. Aus diesem Grund werden Mechanismen benötigt, die laufende Systeme vor nicht gewolltem Verhalten schützen. Entscheidend ist dabei, daß die Software mit unerwarteten Situationen zurechtkommt:

> „Expect the unexpected! What we anticipate seldom occurs, what we least expected generally happens.“ (Benjamin Disraeli)

Als Beispiel für „expect the unexpected“ sei die Schleife von Rückversicherungen genannt: Eine Kette von drei Rückversicherungen wurde zum Zyklus, als die dritte Versicherung sich unwissenderweise bei der ersten rückversicherte. Daraus resultierte, daß alle drei Versicherungen sich selbst rückversichert haben, dafür Kommission zahlen und ihr eigenes Risiko decken. Das Programm hat nur kleinere Zyklen geprüft....[2]

Verhalten	vorhersehbar	nicht vorhersehbar
wünschenswert	1) normales Verhalten	2) kaum möglicher Zustand, inkorrekte Spezifikation und korrekte Implementierung
nicht wünschenswert	3) nicht akzeptables Verhalten, fehlerhafter Zustand	4) unkontrolliertes Verhalten, katastrophaler Zustand

Abb. 1: Klassifikation vom Systemverhalten[3]

In [DENE91] wird das Verhalten von Software in die Kategorien vorhersehbar, nicht vorhersehbar, wünschenswert und nicht wünschenswert aufgeteilt (siehe Abb. 1).

Das Ziel der Fehlerbehandlung ist, die Software auf die Situation, daß das Programm in einen fehlerhaften Zustand übertritt, vorzubereiten. Zwar kann die Software nur selten vom fehlerhaften Zustand in den fehlerfreien Zustand zurückkehren, aber zumindest kann das Vorkommen von Fall 4 minimiert werden, indem solche Fehler vorhergesehen werden und somit zu Fall 3 mutieren.

[2] aus [NEUM95]
[3] frei nach [ARCU97], S. 5

Der erste Teil dieses Kapitels führt in die Terminologie der Fehlerbehandlung ein (Kapitel 2.1). Es folgt ein Abschnitt, der die Ausnahmebehandlung von objektorientierten Programmiersprachen erläutert (Kapitel 2.2). Zum Schluß wird der theoretische Hintergrund der Fehlerbehandlung betrachtet (Kapitel 2.3).

2.1 Terminologie

Eine **Störung** (fault nach ANSI/IEEE) „ist die Unfähigkeit der Betrachtungseinheit, ihre geforderte Funktion auszuführen". Auf Softwarekomponenten bezogen wird in [MEYE97] eine Störung als das Ereignis in einem Softwaresystem bezeichnet, das von seinem beabsichtigten Verhalten während eines seiner Ausführungen abweicht. In [ARCU97] wird ein anderer Akzent gesetzt: Eine Störung wird hier als Herkunft eines Fehlverhaltens verstanden. Dabei wird zwischen Softwarestörung, Hardwarestörung, Lower Level Service Störung und Spezifikationsstörung unterschieden.

Ein **Fehler** (error nach ANSI/IEEE) „ist die Abweichung zwischen dem berechneten, beobachteten oder gemessenen Wert oder einem Zustand der Betrachtungseinheit und dem entsprechenden spezifizierten oder theoretisch richtigen Wert". In [MEYE90] wird als Fehler „die Anwesenheit eines Elements in der Software, das seiner Spezifikation nicht genügt" bezeichnet[4]. Dagegen definiert Meyer in seiner neuer Auflage[5] einen Fehler als eine falsche Entscheidung, die während der Entwicklung eines Softwaresystems getroffen wurde. Dies entspricht bei [ARCU97] jedoch der obigen Definition einer Störung, während ein Fehler hier als Exemplar einer Störung bezeichnet wird. Er ist Teil eines Systemzustands der anfällig dafür ist, zu einem Fehlverhalten zu führen. Die Gruppe der Benutzerfehler gehört nicht zu den Fehlern in diesem Sinne, weil Software auf fehlerhafte Eingaben reagieren kann. Nachdem ein Benutzerfehler bemerkt wird, sollte der Anwender als Reaktion einen entsprechenden Hinweis erhalten. Aus Softwaresicht liegt somit kein Fehlverhalten vor. Als Beispiel hierfür sei ein Taschenrechner genannt, mit dem versucht wird, eine Division durch Null durchzuführen. Der Taschenrechner wird aufgrund der falschen Eingabe nicht abstürzen, sondern eine Fehlermeldung liefern[6].

Ein **Ausfall** (failure) ist nach ANSI/IEEE „die Beendigung der Fähigkeit der Betrachtungseinheit, die geforderte Funktion auszuführen". In [MEYE90] wird failure mit Fehlverhalten übersetzt und als „Unfähigkeit eines Softwaresystems, seinen Zweck zu erfüllen" definiert. Parallel dazu bezeichnet [ARCU97] mit Fehlverhalten die Abweichung eines Softwareprodukts von seiner Spezifikation. Ein Fehlverhalten kann daher nur über eine Relation zu einer Spezifikation definiert werden.

In dieser Arbeit werden die Definitionen von [ARCU97] zugrunde gelegt. Eine Softwarestörung ist somit zur Compilezeit vorhanden, während ein Softwarefehler erst zur Laufzeit auftreten kann. Daraus folgt, daß eine Störung der Verursacher von einem Fehler oder mehreren Fehlern sein kann. Dabei kann die Latenzzeit, das ist das Intervall zwischen der Entstehung der Störung und dem Auftreten eines Fehlers, sehr groß sein. Ein Fehler kann durch eine oder mehrere Störungen verursacht werden und zu mehreren Fehlverhalten führen. Ein Fehlverhalten wiederum kann von mehr als einem Fehler verursacht werden.

[4] [MEYE90], S. 160

[5] [MEYE97]

[6] Viele Taschenrechner zeigen in diesem Fall ein „E" auf dem Display an. Das „E" steht zwar umgangssprachlich für Error, entspricht aber nicht obiger Definition.

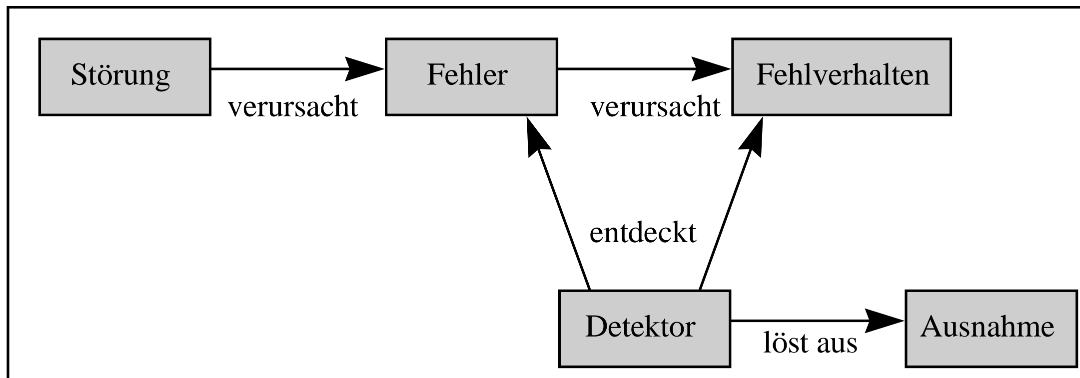


Abb. 2: Zusammenhang der Definitionen[7]

Bevor auf Fehler reagiert werden kann, müssen sie entdeckt werden. Würde die Fehlerbehandlung als unabhängige Softwarekomponente betrachtet werden, könnte man sagen, daß Fehler von der Fehlerbehandlung beobachtet werden. Ein **Fehlerdetektor** (oder einfach nur Detektor) ist dafür zuständig, die Fehler aufzuspüren (siehe Abb. 2) und Gegenmaßnahmen einzuleiten, die diesen Fehler beseitigen, bevor ein Fehlverhalten eintreten kann. Die zugrunde liegende Störung wird dabei nicht beseitigt. Ein Detektor stellt somit eine Verbindung zwischen dem Auftauchen der Fehler und der Behandlung der Fehler dar. Die Qualität eines Fehlerbehandlungssystems hängt stark von der Anzahl, dem Ort und der Art dieser Detektoren ab.

Meyer definiert eine **Ausnahme** (Exception) als „das Auftreten einer abnormen Bedingung während der Ausführung eines Softwareelements“. Konform hierzu ist die Definition von [ARCU97]: Eine Ausnahme ist das Auftreten einer anormalen Situation, dessen Ursache eine Unterbrechung des normalen Kontrollflusses ist. Eine Ausnahme ist ein Ereignis, das zur Laufzeit eines Programms eintreten kann. Eine Ausnahme wird ausgelöst, wenn eine anormale Situation in einer Softwareeinheit auftritt, d.h. ein Fehler entdeckt wurde.

Ein **Ausnahmebearbeiter** (Exception-Handler) ist ein Abschnitt eines Programms, der nach Eintritt einer Ausnahme ausgeführt wird ([ARCU97], [ISSA93]). Der Kontrollfluß geht nach dem Auslösen einer Ausnahme direkt an den Ausnahmebearbeiter über. Welche Möglichkeiten der Ausnahmebearbeiter hat, auf die Ausnahme zu reagieren, wird in Kapitel 2.2 erläutert.

Unter einem **fehlertoleranten** System versteht man ein System, daß fähig ist, sich von Fehlern zu erholen und die normale Verarbeitung wieder aufzunehmen. In [ANDE81] und [ANDE81b] wird Fehlertoleranz in vier Phasen aufgeteilt: Die erste Phase ist dafür zuständig, den fehlerhaften Zustand zu entdecken (error detection). Daran schließt sich die Phase der Schadensbeschränkung und Einschätzung des Fehlers an (damage assessment). Aufgrund der Latenzzeit kann es sein, daß bereits weitere Fehler aufgetreten sind, die bisher noch nicht entdeckt worden sind. Die dritte Phase ist die Fehlerbehebungsphase (error recovery). Der fehlerhafte Zustand wird in den fehlerfreien Zustand überführt, so daß die normale Verarbeitung fortgesetzt werden kann. Dieses ist die arbeitsintensivste Phase. Die letzte Phase ist die Störungsbehandlungsphase (fault treatment and continue service), in der die dem Fehler zugrunde liegende Störung beseitigt wird. Wie bereits erwähnt ist die Lösung dieser Proble-

[7] aus [ARCU97], S. 7

matik sehr komplex, da eine Störung zu unterschiedlichen Fehlern führen und die gleichen Fehler von unterschiedlichen Störungen hervorgerufen werden können.

Fehlertolerante Systeme sind vor allem für sicherheitskritische Anwendungen (z.B. Kontrollsysteme von Flugzeugen) relevant. In diesem Bereich muß sehr sensibel auf Fehler reagiert werden. Bei betrieblichen Informationssystemen ist es meistens ausreichend, daß das Programm im Fehlerfall terminiert. Im Vordergrund steht die Garantie der Korrektheit im Hinblick auf Datenintegrität und die Verhinderung von Datenverlust. Die Sicherheit wird durch andere Maßnahmen wie Backups, technische Infrastruktur und redundante Hard- und Software erreicht.

Als letztes bleibt zu klären, was unter einem Fehlerbehandlungssystem zu verstehen ist. [KOEN90] fordert von einer **Fehlerbehandlung**, daß sie einfach und ohne viel Overhead funktionieren soll. Außerdem soll ein einheitlicher Mechanismus dafür sorgen, daß nicht jeder Fehler eine eigene Behandlung erhält, sondern daß alle anormalen Situationen gleichermaßen behandelt werden. Ein Fehlerbehandlungssystem sollte nach [DONY90] Materialien und Protokolle anbieten, um eine Kommunikation zwischen Aufrufer und Aufgerufenem aufzubauen. Es sollte das Auslösen von Ausnahmen und Behandeln von Ausnahmen möglich sein. Dony[8] hat einen Acht-Punkte-Katalog aufgestellt der beschreibt, was ein Fehlerbehandlungssystem anbieten muß:

1. Programmierer sollten ihre eigenen Ausnahmen definieren können, auch wenn schon grundlegende Ausnahmen existieren.
2. Es sollte möglich sein, Bearbeiter an Ausnahmen anzuhängen. Diese Bearbeiter sollen eine allgemeine Fehlerbehandlung zur Verfügung stellen (z.B. Fehlermeldung ausgeben).
3. Es sollte möglich sein, Bearbeiter an Klassen anzuhängen.
4. Es sollte möglich sein, Bearbeiter an Ausdrücke anzuhängen (z.B. try-catch-Konstrukt).
5. Ausnahmen sollten entlang der Aufrufkette propagiert werden.
6. Ausnahmen sollten hierarchisch organisierte Klassen sein.
7. Alle Bearbeiter sollten die Ausnahme-Hierarchie berücksichtigen (z.B. können spezielle Ausnahmen unter einem allgemeineren Typ zusammengefaßt behandelt werden).
8. Alle Anweisungen eines Bearbeiters sollten zur Ausnahme passen. Damit ist gemeint, daß sehr schwerwiegende Ausnahmen nicht fortgesetzt, sondern terminiert werden sollten.

2.2 Ausnahmebehandlung

Zur Zeit der Programmierung ohne Ausnahmebehandlung (exception handling) lieferten Funktionen Fehlercodes zurück, wenn in ihnen ein Fehler aufgetaucht war. Der Aufrufer der Funktion war im Anschluß an den Aufruf dafür verantwortlich, den Returncode zu prüfen und festzustellen, ob während der Abarbeitung der Funktion ein Fehler aufgetreten ist. Die Funktion, die den Fehler verursacht hat, besaß keine Möglichkeit, den Aufrufer dazu zu zwingen.

Um die unsichere Methode der Fehlercode-Programmierung zu eliminieren, bieten die objektorientierten Programmiersprachen (Java, Smalltalk, C++, etc.) den Fehlerbehandlungs-Mechanismus an. Die Grundlage hierfür hat Goodenough im Jahre 1975 gelegt. In [GOOD75] betrachtet er Ausnahmen als eine Möglichkeit, mit dem Aufrufer einer Operation zu kommunizieren. Dabei unterscheidet er verschiedene Fälle, wie Fehler während der Operationsausführung, Klassifikation des Ergebnisses der Operation und das Auftauchen von

[8] [DONY88], S. 148 - 151

verschiedenen signifikanten Ereignissen (nicht notwendigerweise Abweichungen) während der Ausführung der Operation.

Der Sinn der Ausnahmebehandlung in den gängigen Programmiersprachen ist es, den Code, der für die Fehlerbehandlung zuständig ist, von dem Code, der die normale Verarbeitung durchführt, zu trennen. Außerdem wird der Aufrufer einer Methode dazu gezwungen, die von der aufgerufenen Methode ausgelösten Ausnahmen zu behandeln oder an seinen Aufrufer weiterzuleiten. Die Ausnahmebehandlung besteht aus zwei Teilen: Der erste Teil enthält den normalen Code, der Fehler verursachen kann (try-Block), der zweite Teil wird als Reaktion auf einen Fehler ausgeführt (catch-Block). Zu einem try-Block kann es mehrere catch-Blöcke geben, die unterschiedliche Fehler behandeln.

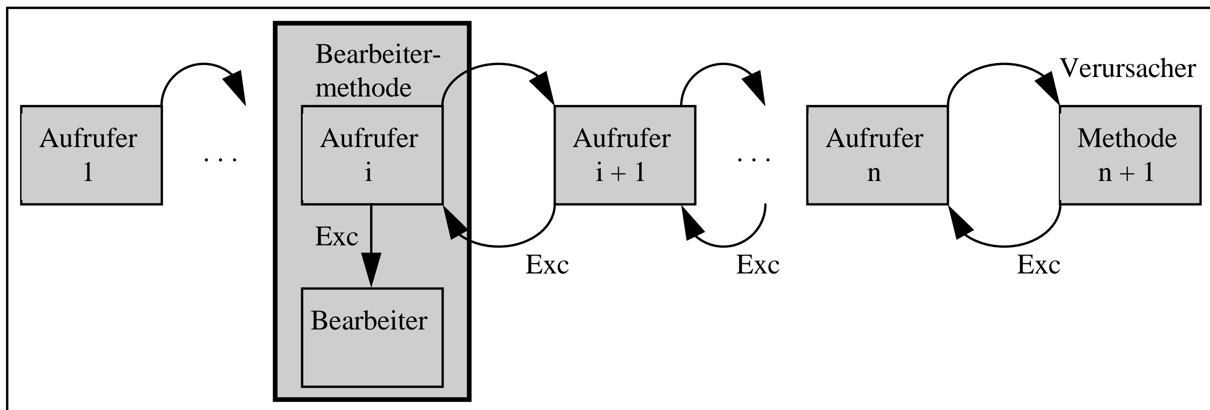


Abb. 3: Kontrollfluß von Methoden, die Ausnahmen (Exc) auslösen[9]

In Abb. 3 ist eine Aufrufreihenfolge (Aufrufer 1 bis n und die von n aufgerufene Methode n + 1) dargestellt. Die Methode n + 1 verursacht eine Ausnahme und gibt diese an ihren Aufrufer n zurück. Der Aufrufer n besitzt keinen Bearbeiter für diese Ausnahme und reicht diese den Aufruf-Stack weiter nach oben, bis zum Aufrufer i, der einen Bearbeiter für diese Ausnahme anbietet. In der Literatur werden verschiedene Möglichkeiten genannt, die ein Bearbeiter hat, um auf einen Fehler zu reagieren.

[LACO91] beschreibt dazu drei Möglichkeiten: Erstens die Fortsetzung der Ausführung am Punkt des Entstehens der Ausnahme (resumption), zweitens die Fortsetzung des Kontrollflusses auf der Ebene der Entität mit dem Ausnahmebearbeiter (termination) und drittens die Wiederholung des fehlgeschlagenen Aufrufs (retry), nachdem einige Parameter verändert wurden. In [MILL97] wird zusätzlich noch die Möglichkeit vorgesehen, daß der Bearbeiter einen Wert anstelle des Ergebnisses, das vom Aufrufer erwartet wurde, zurückgeben kann (replacement). Weiterhin besteht die Möglichkeit, die Ausnahme nach außen zu propagieren. Dabei wird zwischen impliziter (automatischer) und expliziter Weiterreichung unterschieden. Knudsen ([KNUD87]) unterscheidet das Terminationmodell in abruptes Terminieren, d.h. daß alle Blöcke, die weiter innen als der Bearbeiter liegen, beendet werden und fließendes Terminieren, bei dem alle inneren Blöcke vor ihrer Terminierung Aufräumarbeiten leisten können. Meyer spricht hierbei von organisierter Panik[10].

In den meisten Programmiersprachen wird das Termination-Modell benutzt. Der Grund hierfür ist die einfachere Implementierung. In Smalltalk stehen allerdings alle in [LACO91] genannten Möglichkeiten zur Verfügung.

[9] aus [FEDE90], S. 82
[10] aus [MEYE92]

Kritiker des Ausnahmebehandlungs-Mechanismusses behaupten, die Ausnahmebehandlung sei nichts anderes als eine Goto-Anweisung und erhöhe die Komplexität der Programmierung. Dieses Argument kann am besten entkräftet werden, indem man das Benutzen von Ausnahmen außerhalb der Fehlerbehandlung konsequent verbietet.

2.3 Theoretischer Hintergrund

Dieser Abschnitt beschreibt den theoretischen Hintergrund der Fehlerbehandlung. Abb. 4 zeigt die disjunkten Verhaltensmöglichkeiten einer Methode aus der Laufzeitsicht: Das gewünschte Verhalten einer Methode ist das normale Verhalten. Darüber hinaus gibt es das Ausnahmeverhalten, das genau wie das normale Verhalten Teil der Spezifikation der Methode ist. Das Ausnahmeverhalten ist somit kein Fehler oder Fehlverhalten. Im dritten Quadranten der Grafik ist das Fehlverhalten (Failure) dargestellt. Die Methode wurde mit gültigen Parametern aufgerufen, zeigt aber ein Verhalten, das dem der Spezifikation widerspricht. Als letzte Verhaltensmöglichkeit gibt es das unspezifizierte Verhalten. Die Methode wurde mit falschen Parametern aufgerufen. Das Ziel der Spezifikation ist es zu erreichen, daß das Verhalten der unteren beiden Quadranten zur Laufzeit niemals eintreten kann.

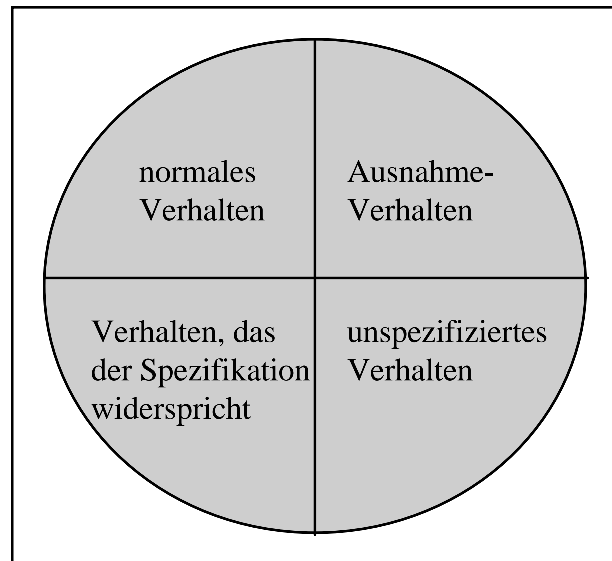


Abb. 4: Verhalten einer Methode aus Laufzeitsicht[11]

Abb. 5 zeigt die Verhaltensmöglichkeiten einer Methode aus Sicht der Spezifikation. Die Anzahl der Verhaltensmöglichkeiten zum Verhalten aus Laufzeitsicht hat sich um eine verringert. Identisch in beiden Sichtweisen ist das normale Verhalten (Standardverhalten), das den zentralen Teil der Spezifikation bildet. Das weiter oben als Ausnahmeverhalten definierte Verhalten wird aus Spezifikationssicht als anormales Verhalten bezeichnet. In der Spezifikation werden Fehlermeldungen und Warnungen definiert. Der Aufrufer muß diese Ausnahmen erwarten. Zusammengenommen bilden beide Verhaltensweisen das korrekte Verhalten (korrekt im Sinne von spezifiziert). Jedes andere Verhalten ist unspezifiziert und damit fehlerhaft und unerwünscht. Die Behandlung solcher Fehler

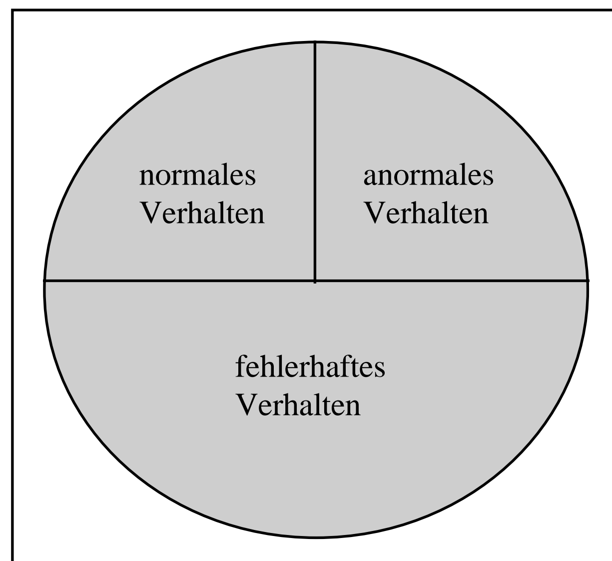


Abb. 5: Verhalten einer Methode aus Spezifikationssicht[12]

[11] frei nach [ARCU97], S. 115
[12] frei nach [ARCU97], S. 116

sollte robust und fehlertolerant sein. Sie erzeugt Fehlermessages, die möglichst ein Systemfehlerprotokoll (Backtracing des Aufruf-Stacks) enthalten sollten.

In [MILL97] wird dargelegt, daß durch die Ausnahmebehandlung ein Objekt nicht nur gültig oder ungültig sein kann, sondern daß ein dritter Zustand eingeführt wird, der partieller Zustand genannt wird. Der Wechsel von einem Zustand in einen anderen wird in der objektorientierten Welt als atomar angenommen, ist somit nicht von außen beobachtbar. Durch die Ausnahmebehandlung wird diese Atomarität aufgehoben, da ein Zustandswechsel beim Auslösen einer Ausnahme noch nicht vollständig abgeschlossen sein muß. Es stellt sich die Frage, was mit Objekten, die sich im partiellen Zustand befinden, geschehen soll. Am sinnvollsten erscheint es, den ursprünglichen gültigen Zustand wiederherzustellen. Falls dies fehlschlägt, ist die Zerstörung des Objekts notwendig.

Da die Reaktion auf einen partiellen Zustand ähnlich der Reaktion auf einen ungültigen Zustand ist, faßt [ISSA93] diese Zustände zusammen. Die Operationsdomäne besteht daher nur aus zwei Bereichen: Der Standard- und der Ausnahmedomäne.

In Abb. 6 ist das Modell einer idealen fehlertoleranten Komponente dargestellt. Sie ist ein Bestandteil eines Systems, von dem Anfragen an sie gestellt werden (oben links). Im Regelfall wird die Komponente das gewünschte normale Verhalten zeigen und ein entsprechendes Ergebnis zurückliefern. Dazu kann sie die Hilfe von Subkomponenten in Anspruch nehmen (unten links). Wenn die Komponente einen ungültigen Aufruf erhält, so löst sie eine entsprechende Ausnahme (Interface exception) aus, die an die aufrufende Komponente signalisiert wird. Tritt während der normalen Ausführung einer Methode eine Ausnahme auf (entweder in der Komponente selber oder in einer ihrer Subkomponenten), so wechselt die Komponente in den fehlerhaften Zustand, der für die Ausnahmebehandlung zuständig ist. In diesem Zustand versucht die Komponente, den Fehler selbst zu beheben. Falls dies Erfolg hat, wechselt sie wieder in den normalen Zustand zurück; wenn der Erfolgsfall ausbleibt, wird eine Ausnahme (Failure Exception) an die aufrufende Komponente signalisiert. Es ist zu beachten, daß eine Komponente niemals die Ausnahmen, die in Abb. 6 unterhalb der Komponente dargestellt sind, direkt an die aufrufende Komponente propagieren darf. Diese Ausnahmen müssen von ihr selber behandelt werden, da die aufrufende Komponente nichts davon wissen sollte, wie die aufgerufene Komponente implementiert ist. Falls die Komponente den Fehler nicht selber beheben kann, sollte sie eine neue Failure Exception auslösen. Bevor die Kontrolle durch die Ausnahme an den Aufrufer übergeht, muß eine ideale fehlertolerante Komponente den internen Status konsistent machen, damit bei einem erneuten Aufruf nicht unnötige Ausnahmen signalisiert werden. Am sichersten ist es, daß eine Komponente, die eine Anfrage nicht fehlerfrei

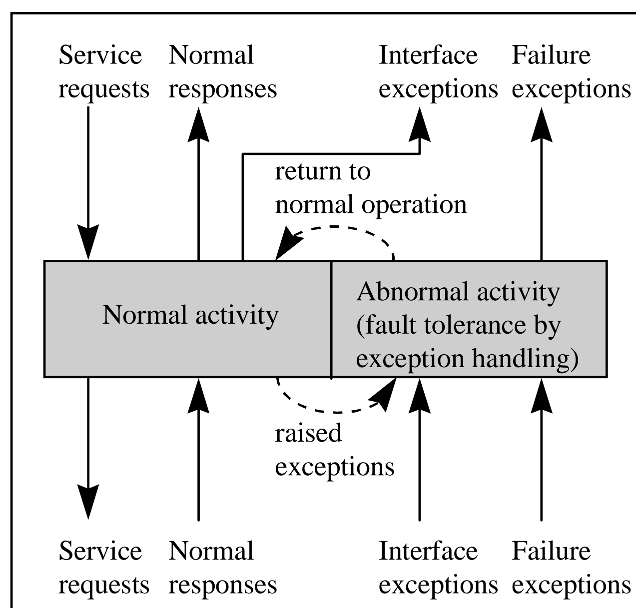


Abb. 6: Ideale fehlertolerante Komponente[13]

[13] aus [ANDE81], S. 298

abarbeiten kann und deshalb eine Ausnahme auslösen muß, den Zustand wieder herstellt, den sie vor dem Aufruf inne hatte.

Der erste Teil dieses Kapitels hat in die Terminologie von Fehlern eingeführt und bildet damit die begriffliche Grundlage dieser Arbeit. Die Einführung in das Konzept der Ausnahmebehandlung und der Überblick über den theoretischen Hintergrund wurde dargelegt, um ein allgemeines Verständnis der Thematik der vorliegenden Diplomarbeit zu gewinnen. Im folgenden Kapitel wird das Vertragsmodell von Bertrand Meyer erläutert, das ein Konzept zur Entdeckung von Fehlern ist.

3 Das Vertragsmodell von Bertrand Meyer

Nachdem im vorigen Kapitel der allgemeine Aspekt der Fehlerbehandlung im Vordergrund stand, wird in diesem Kapitel ein Konzept zur Fehlerentdeckung vorgestellt, mit dessen Hilfe die dem Fehler zugrunde liegende Störung vom Programmierer behoben werden kann. Es handelt sich hierbei um das Vertragsmodell von Bertrand Meyer[14]. Dieses Vertragsmodell hilft dem Programmierer, korrekte Software zu erstellen. Der Grundgedanke des Vertragsmodells ist, daß die Beziehung zwischen zwei Klassen, von denen die eine die andere benutzt, eine formale Abmachung (einen Vertrag) enthält. Dieser Vertrag spezifiziert die Verantwortlichkeiten der beiden Klassen. Jede Klasse hat dabei Rechte und Pflichten: So liegt es in der Verantwortlichkeit des Aufrufers, gültige Parameter zu übergeben (Pflicht des Aufrufers und damit Recht der aufgerufenen Klasse, die Contractor genannt wird) und in der Verantwortung des Contractors, das in der Spezifikation definierte Verhalten zu zeigen (Pflicht des Contractors und Recht des Aufrufers). Dies wird Programmieren durch Vertrag genannt, weil beide Seiten sich an die Spezifikation (den Vertrag) halten müssen. Das Vertragsmodell ist Bestandteil der von Bertrand Meyer entwickelten Programmiersprache Eiffel.

Das Vertragsmodell legt die Semantik einer Klasse fest: Eine Klasse enthält nicht nur Attribute und Methoden, sondern auch Bedingungen, die für die Methoden gelten und Axiome, die für die ganze Klasse erfüllt sein müssen. Es gibt zwei Arten von Bedingungen: Vor- und Nachbedingungen. Diese sind Thema von Kapitel 3.1. Die Axiome werden von Meyer als Invarianten bezeichnet. Sie werden in Kapitel 3.2 diskutiert. In Kapitel 3.3 wird kurz auf weitere Konstrukte mit Zusicherungen der vollständigkeitshalber eingegangen, welche im weiteren Verlauf der Arbeit keine große Rolle spielen. Es folgt Kapitel 3.4, das die Problematik der Vererbung von Vor-/ Nachbedingungen und Invarianten behandelt. Im letzten Kapitel dieses Abschnitts werden die Möglichkeiten der Umsetzung des Vertragsmodells in anderen Programmiersprachen als Eiffel untersucht.

3.1 Vor- und Nachbedingungen

Eine Methode sollte als Bestandteil eines ADTs eine sinnvolle Aufgabe ausführen, die in einer Spezifikation festgelegt ist. Aus dieser Spezifikation sollte hervorgehen, welche Eigenschaften erfüllt sein müssen, damit die Methode korrekt arbeitet. Diese Eigenschaften werden Vorbedingung (precondition) genannt. Die Vorbedingung wird jedesmal geprüft, wenn die Methode aufgerufen wird. Die Nachbedingung (postcondition) einer Methode spezifiziert die Eigenschaften, die garantiert sind, wenn die Methode beendet wird. Die Nachbedingung wird jedesmal vor dem Verlassen der Methode aufgerufen.

Ein Aufrufer einer Klasse hat die Pflicht, die Vorbedingung zu erfüllen. Wenn er diese Pflicht erfüllt, hat er ein Recht darauf, daß die Nachbedingung vom Contractor erfüllt wird. Der Contractor wiederum hat das Recht, daß die Vorbedingung vom Aufrufer erfüllt wird, und die Pflicht, die Nachbedingung einzuhalten. Da beide Seiten einen Vorteil vom Vertrag haben, sollte es ihnen nicht schwer fallen, die Verpflichtungen, die sie mit dem Vertrag eingegangen sind, zu erfüllen.

Wenn die Vorbedingung einer Methode nicht erfüllt ist, hat der Aufrufer den Vertrag gebrochen. Der Contractor ist dann nicht mehr verpflichtet, sich an den Vertrag zu halten und die Nachbedingung zu erfüllen. Wenn die Vorbedingung erfüllt ist und die Nachbedingung nicht erfüllt ist, so hat der Contractor den Vertrag gebrochen. Ist eine der beiden Bedingungen

---

[14] siehe [MEYE88], [MEYE90], [MEYE92], [MEYE97]

verletzt, wird eine Ausnahme ausgelöst, die den Bruch des Vertrags signalisiert. Nach Meyer sollte diese Ausnahme normalerweise zum Programmabbruch führen, da ein gravierender Programmfehler vorliegt.

Die Extremfälle der Vor- und Nachbedingung sind die boolschen Werte false und true: False ist die härteste Vorbedingung für einen Aufrufer und die einfachste für den Contractor. Es spielt keine Rolle, welchen Inhalt der Methodenrumpf besitzt. Kein Aufrufer kann diese Vorbedingung je erfüllen. Andererseits ist False die härteste Nachbedingung für den Contractor. Er wird sie niemals erfüllen können. Für die Aufrufer ist diese Nachbedingung aber die beste, da ihnen alles zugesichert wird.

Der Vorteil des Vertragsmodells ist, daß genau festgelegt wird, welcher Vertragspartner welche Pflichten besitzt und daher keine Prüfungen durchgeführt werden müssen, die von dem jeweiligen Vertragspartner abgedeckt werden. Der Aufgerufene kann davon ausgehen, daß die Vorbedingung eingehalten wird. Der Aufrufer ist dafür verantwortlich. Dadurch wird garantiert, daß die Daten genau einmal auf der Seite des Aufrufers geprüft werden und somit keine redundanten Prüfungen durchgeführt (defensives Programmieren) oder die Prüfungen vollständig vergessen werden. Dies erhöht die Korrektheit, Effizienz und Verständlichkeit des Programmcodes.

Die Vor- und die Nachbedingung bestehen aus boolschen Ausdrücken, die Zusicherungen (Assertions) genannt werden. Die Klasse Stack soll hierfür ein Beispiel sein[15]. Es wurde die Notation von Java gewählt.

Methode *pop()*:
  Vorbedingung: *!isEmpty()*
  Nachbedingung: *!isFull()*

Methode *push(String)*:
  Vorbedingung: *!isFull()*
  Nachbedingung: *!isEmpty()*

Methode *top()*:
  Vorbedingung: *!isEmpty()*
  Nachbedingung: *!isEmpty()*

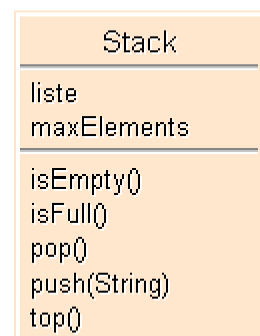


Abb. 7: Die Klasse Stack

Zusicherungen können in Eiffel zur Laufzeit mittels eines leistungsstarken Debugging-Tools überwacht werden.

3.2 Invarianten

Vor- und Nachbedingungen spezifizieren ausschließlich die Eigenschaften ihrer Methode. Es gibt aber zusätzlich Eigenschaften, die für alle Exemplare einer Klasse unabhängig von der Ausführung einer Methode gleich sind. Diese Eigenschaften werden Invariante genannt. Beim Stack wäre eine Invariante *0 <= liste.size() <= maxElements.*

Eine Invariante wird immer direkt nach der Vorbedingung und direkt vor der Nachbedingung einer Methode geprüft. Daraus folgt, daß die Invariante Bestandteil des Vertrags zwischen dem Aufrufer und dem Contractor ist. Während der Ausführung einer Methode darf die

[15] aus [MEYE97]

Invariante zwar verletzt werden, insbesondere auch bei privaten Methodenaufrufen, am Ende der Methode muß sie aber wieder erfüllt werden. Auch nach der Beendigung des Konstruktors muß die Invariante gültig sein.

Eine Invariante verschärft sowohl die Vor- als auch die Nachbedingung, da sie mit beiden Bedingungen konjunktiv verknüpft wird. Prinzipiell könnte die Invariante Bestandteil der Vor- und Nachbedingungen sein. Dies hätte aber eine Verwischung der Trennung zwischen Methodenbedingung und Exemplarbedingung zur Folge. Außerdem besteht die Gefahr, daß bei einer Erweiterung der Klasse die Invariante vergessen werden könnte.

Als Ergebnis bleibt festzuhalten, daß die Invariante das Programmieren einer Klasse sowohl vereinfacht, als auch verkompliziert: Durch die konjunktive Verknüpfung mit der Vorbedingung wird die Pflicht des Aufrufers erhöht und damit auch das Recht des Contractors; dies vereinfacht das Programmieren. Durch die konjunktive Verknüpfung der Invariante mit der Nachbedingung wird das Programmieren erschwert, da die Anzahl der Zusicherungen am Methodenende durch die Invariante erhöht wird (Pflicht des Contractors und Recht des Aufrufers).

3.3 Weitere Konstrukte mit Zusicherungen

Vorbedingungen, Nachbedingungen und Invarianten sind die wichtigsten Arten von Zusicherungen. Der vollständigkeitshalber werden die weiteren Konzepte von Bertrand Meyer kurz dargestellt.

In Eiffel existiert eine Check-Instruktion, mit der eine übergebene Zusicherung geprüft werden kann. Check-Instruktionen können an beliebiger Stelle im Sourcecode stehen. Wenn beispielsweise auf einem Stack n push Operationen und m pop Operationen durchgeführt wurden und n > m ist, so braucht vor der nächsten pop Operation keine isEmpty-Prüfung durchgeführt werden, da der Stack garantiert nicht leer ist. In diesem Fall kann die Check-Instruktion (*check (!Stack.isEmpty()*) benutzt werden. Check-Instruktionen dienen auch zu Kommentarzwecken.

Außer der Check-Instruktion gibt es noch zwei Zusicherungen für Schleifen: Schleifen-Invarianten und Varianten. Schleifen-Invarianten sind Invarianten, die bei jedem Schleifendurchlauf erfüllt sein müssen. Eine Variante ist eine nicht negative ganze Zahl, die bei jedem Schleifendurchlauf um mindestens eins verringert wird und die maximale Anzahl der Schleifendurchläufe angibt. Sie garantiert die Terminierung der Schleife.

3.4 Vererbung von Bedingungen und Invarianten

Die Vererbung ist ein mächtiges Konstrukt des objektorientierten Paradigmas. Um das Vertragsmodell sinnvoll bei Klassen, die Erben von anderen Klassen sind, anwenden zu können, hat Meyer das Subvertragsmodell eingeführt. Dieses wird im folgenden erläutert.

Zunächst folgen zwei Definitionen: Eine Zusicherung ist **stärker** als eine zweite Zusicherung, wenn die erste die zweite impliziert. Eine Zusicherung ist **schwächer** als eine zweite Zusicherung, wenn die zweite die erste impliziert. Die Zusicherung x >= 0 ist stärker als die Zusicherung x > 4; die Zusicherung x > 7 ist schwächer als die Zusicherung x > 3.

Für Vor- und Nachbedingungen gilt folgende Redefinitionsregel: Sei r eine Routine der Klasse A und s eine Redefinition von r in einer von A erbenden Klasse B oder eine tatsächliche Implementierung von r, wenn r abstrakt war. In diesem Fall darf die Vorbedingung von s nicht stärker sein als die Vorbedingung von r, und die Nachbedingung von s darf nicht

schwächer sein als die Nachbedingung von r. Diese Regel begründet sich durch das dynamische Binden: Die Methode s muß alle Aufrufe akzeptieren, die r akzeptiert. Deshalb darf die Vorbedingung nicht stärker sein. Sie darf allerdings schwächer sein. Die Methode s muß die Nachbedingung von r garantieren. Die Nachbedingung darf nicht schwächer sein, kann aber stärker sein.

Für die Invarianten einer Klasse gilt folgende Regelung: Die Invarianten aller Oberklassen einer Klasse sind auch immer Invarianten der Klasse selbst. Daraus resultiert eine konjunktive Verknüpfung der Oberklasseninvarianten mit den eigenen Invarianten (falls keine vorhanden wird der boolsche Wert true angenommen). In Eiffel werden automatisch die Invarianten aller Oberklassen mitgetestet. Der Grund für diese Regel ist, daß ein Exemplar der Unterklasse auch immer ein Exemplar der Oberklasse ist und daher auch dessen Invariante erfüllen muß.

Durch das Subvertragsmodell besteht die Möglichkeit, Verträge billiger und besser zu implementieren als die Oberklasse: Die Vorbedingung wird schwächer und damit billiger; die Nachbedingung wird stärker und damit besser. Auf diese Weise wird sichergestellt, daß der Subcontractor das einhält, was der Contractor im Originalvertrag versprochen hat. Außerdem wird durch das Subvertragsmodell deutlich, daß eine Methode der Subklasse nicht etwas völlig anderes implementiert als in der Oberklasse spezifiziert wurde, sondern daß die neue Version einer Methode der Subklasse kompatibel zur Spezifikation der überschriebenen Methode der Oberklasse ist.

3.5 Umsetzung des Vertragsmodells in anderen Programmiersprachen

Bertrand Meyer hat das Konzept des Vertragsmodells in die von ihm entwickelte Programmiersprache Eiffel integriert. Er hat Schlüsselwörter eingeführt, mit deren Hilfe die jeweiligen Zusicherungen codiert werden können. Andere Programmiersprachen besitzen diese Schlüsselwörter jedoch nicht. Es stellt sich also das Problem, wie das Vertragsmodell in diesen Programmiersprachen implementiert werden kann.

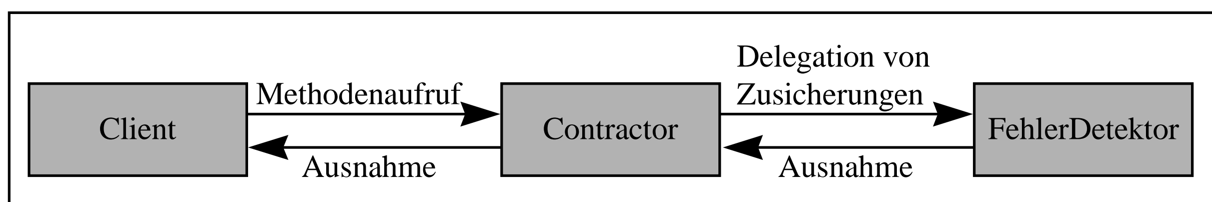


Abb. 8: Propagierung von Zusicherungen und Ausnahmen

[ARCU97] schlägt für die Implementation des Vertragsmodells einen Delegationsmechanismus vor: In jeder Methode wird die Vor- und Nachbedingung mit Hilfe eines boolschen Ausdrucks beschrieben, ausgewertet und das Ergebnis zur Prüfung an eine Klasse namens *FehlerDetektor* delegiert, die als Singleton[16] implementiert ist (siehe Abb. 8). Sie besitzt für die Überprüfung jeder Zusicherungsart (Vorbedingung, Nachbedingung, Invariante, usw.) eine Methode. Der *FehlerDetektor* prüft diese Bedingung und löst im Fall der Verletzung der Zusicherung eine Ausnahme aus, die von der Klasse, die das Vertragsmodell benutzt (Contractor), an seinen Client weiterpropagiert wird. Der Client muß diese Ausnahme behandeln. In Abhängigkeit von der Programmiersprache kann diese Ausnahme auch zur Terminierung der Anwendung führen.

[16] [GAMM95]

Ebenso wird die Prüfung der Invariante vom *FehlerDetektor* übernommen. Im Unterschied zu der Vor- und Nachbedingung wird hier kein boolscher Ausdruck übergeben. Weil die Invariante klassenweit eindeutig ist, wird sie in einer Methode formuliert, die einen boolschen Wert als Rückgabe liefert. Der *FehlerDetektor* ruft diese Methode des Contractors auf und übernimmt im Fall der Verletzung der Invariante das Auslösen einer entsprechenden Ausnahme.

Die Mächtigkeit der Formulierung von Zusicherungen hängt vom Umfang der durch die jeweilige Programmiersprache zur Verfügung gestellten boolschen Operatoren ab. Um eine mächtige Zusicherungssprache zu erhalten, sollten der All- und der Existenzquantor vorhanden sein. In Eiffel besteht die Möglichkeit mittels des Schlüsselworts *old* auf die Attributwerte zuzugreifen, die das Objekt besaß, bevor die aktuell auszuführende Methode aufgerufen wurde. Die Umsetzung dieses Schlüsselworts in eine andere Programmiersprache ist sehr aufwendig, da eine Kopie aller Attributwerte zu Beginn einer jeden Methode angelegt werden muß. Dabei ist zu beachten, ob die jeweilige Programmiersprache Kopier- oder Referenzsemantik benutzt[17].

Wenn die für die Implementierung benutzte Programmiersprache Makros zur Verfügung stellt, können diese die Funktionalität des Detektors auf einfache Weise ersetzen. Für jede Zusicherung kann ein Makro definiert werden, dem die Zusicherung in Form eines boolschen Ausdrucks übergeben werden kann. Das Makro prüft die Zusicherung und kann eine entsprechende Ausnahme auslösen.

In Eiffel wird der Kontrollfluß nach dem Auslösen einer Ausnahme im rescue-Block derselben Klasse fortgesetzt. Sollte beispielsweise eine Zusicherung verletzt werden, kann im rescue-Block versucht werden, den Aufruf erneut auszuführen (retry) oder, falls dies nicht möglich ist, den ursprünglichen Zustand des Objekts wiederherzustellen. Dadurch wird gewährleistet, daß die Invariante wieder erfüllt ist. Die Ausnahme wird anschließend zum Aufrufer propagiert. Ein rescue-Block ist in vielen Programmiersprachen nicht vorhanden. Deshalb sollte die Ausnahme gleich zum Aufrufer propagiert werden. Dieser muß anschließend die Ausnahme behandeln.

Sollte die Programmiersprache einen Preprozessor enthalten, so können die Zusicherungen vor dem Compilevorgang aus dem Sourcecode entfernt werden für den Fall, daß sie zur Laufzeit nicht geprüft werden sollen. Andernfalls könnte ein Skript geschrieben werden, das die Zusicherungen ein- bzw. auskommentiert.

Einen interessanten Aspekt bringt [MCKI96]. Das Ziel des Vertragsmodells sei eine vollständige Spezifikation, die von technisch orientierten Personen lesbar sein soll. Eine kompilierbare Spezifikation sei daher nicht erforderlich.

In dieser Arbeit soll das Vertragsmodell implementiert werden, da es nach Meinung des Autors nicht ausreicht, die Zusicherungen nur zu spezifizieren. Eine Überprüfung zur Laufzeit erscheint für das Ziel, zuverlässige Software zu schreiben, zumindest während der Testzeit sinnvoll. Da die für das Fehlerbehandlungssystem zugrunde liegende Programmiersprache Java ist und Java weder Preprozessoren noch Makros noch All-/Existenzquantoren besitzt, können einige der in diesem Kapitel angesprochenen Aspekte nicht berücksichtigt werden. Die Implementation beschränkt sich daher auf den angesprochenen Delegationsmechanismus

[17] siehe Beispiel in [PAYN98]

mit der Klasse *FehlerDetektor*, die Methoden zur Überprüfung von Vorbedingung, Nachbedingung und Invariante besitzt. Die konkrete Implementation ist Kapitel 7.3 zu entnehmen.

4 Kontext und Anforderungen des Fehlerbehandlungssystems

In diesem Kapitel geht es um den Kontext und die Anforderungen des Fehlerbehandlungssystems. Da die vorliegende Arbeit eine Praxisarbeit ist, existiert ein zugrunde liegendes Projekt, in dem das Fehlerbehandlungssystem eingesetzt wurde. Eine kurze Beschreibung des Projekts wird im ersten Abschnitt (Kapitel 4.1) gegeben. Es folgt eine detaillierte Beschreibung der Anforderungen, die an des Fehlerbehandlungssystem gestellt werden (Kapitel 4.2). Hierbei wird zunächst nicht beachtet, daß das Fehlerbehandlungssystem in einem heterogenen Systemumfeld laufen soll. Um dem Verteilungsaspekt zu genügen, wird eine Middleware eingesetzt, die von konkreten Hardware- und Betriebssystemplattformen abstrahiert. In Kapitel 4.3 wird zunächst definiert, was eine Middleware ist und anschließend ein Katalog von Anforderungen aufgestellt, die für das Fehlerbehandlungssystem relevant sind.

4.1 Das Java-Forschungsprojekt der HM

Das Ziel des Java-Forschungsprojekts war es, Erfahrungen mit der Programmiersprache Java aufzubauen. Eine der Fragestellungen, die das Projekt klären sollte, war, ob Java eine Programmiersprache ist, mit der neue Anwendungssysteme in der Hamburg-Mannheimer Versicherungs-AG (HM) entwickelt werden können oder ob Java nur ein aktueller vorübergehender Trend ist, der in der HM keine weitere Berücksichtigung finden soll. Im Vordergrund der Betrachtung stand dabei die Entwicklung von verteilten Anwendungen und die Anbindung von Host-Programmen. Außerdem wurde überprüft, inwieweit auf Smalltalk basierende Konzepte der Anwendungsarchitektur mittels Java realisierbar sind.

```
P120  EDVV / TP-Auskunft                                                  23.10.97
 Kontoauskunft                                                     Kontoübersicht
 ---------------------------------------------------------------------------------
 Beitragszahler: Mustermann,Heinrich           BNR: 00.000.001-00        1. Bild
 Anschrift     : Nebenstraße 0815, 12345 Hintertupfing

 ------------ZAHLUNGSSTAND------------- ! ------------BANKVERBINDUNG------------
 Buchungsdatum               06.08.1997 ! Zahlungsweg   02        Abruf
 Belastungsstatus               10.1997 ! Bankleitzahl            000 000 00
 Laufende Fälligkeit             123,45 ! Kreditinstitut          Bank
 -          Guthaben             567,89 ! Konto                   1234567890
                             ---------- !
 G e s a m t b e t r a g           0,00 !

 ---------------------------LETZTE ZAHLUNGSBUCHUNGEN----------------------------
 VERARB.DAT. BUCH.DAT.  B U C H U N G S T E X T          LASTSCHRIFT  GUTSCHRIFT
 01.07.97    30.06.97   Banküberweisung                                    52,00
 01.07.97    30.06.97   Banküberweisung                                    26,00
 01.06.97    29.05.97   Banküberweisung                                    52,00
 01.06.97    29.05.97   Banküberweisung                                    26,00
 --------------------------------------------------------------------------------
 Auswahl:      (B=Buchungsring, K=Kette)
 BNR/VSNR:                                     F3 zurück   F11 Druck   F12 Hilfe
```

Abb. 9: Kontoauskunft (TP 120)[18]

Bei dem im Java-Forschungsprojekt entwickelten Prototypen handelt es sich um eine Intranetanwendung aus dem Bereich der Vertriebsunterstützung. Außendienstmitarbeiter sollen die Möglichkeit bekommen, über ein HM-Intranet auf Daten zuzugreifen, die sich in der HM-Datenbank auf dem Host befinden. Abb. 9 zeigt den Aufbau des Hostprogramms TP 120 Kontoauskunft. Der Anwender gibt wahlweise eine Buchungsnummer (BNR) oder Versicherungsscheinnummer (VSNR) ein und erhält Konto-Informationen eines Ver-

[18] aus [GRAH97b], S.70

sicherungsnehmers. Diese Host-Maske wurde durch eine neue Java-Oberfläche (siehe Abb. 10) ersetzt.

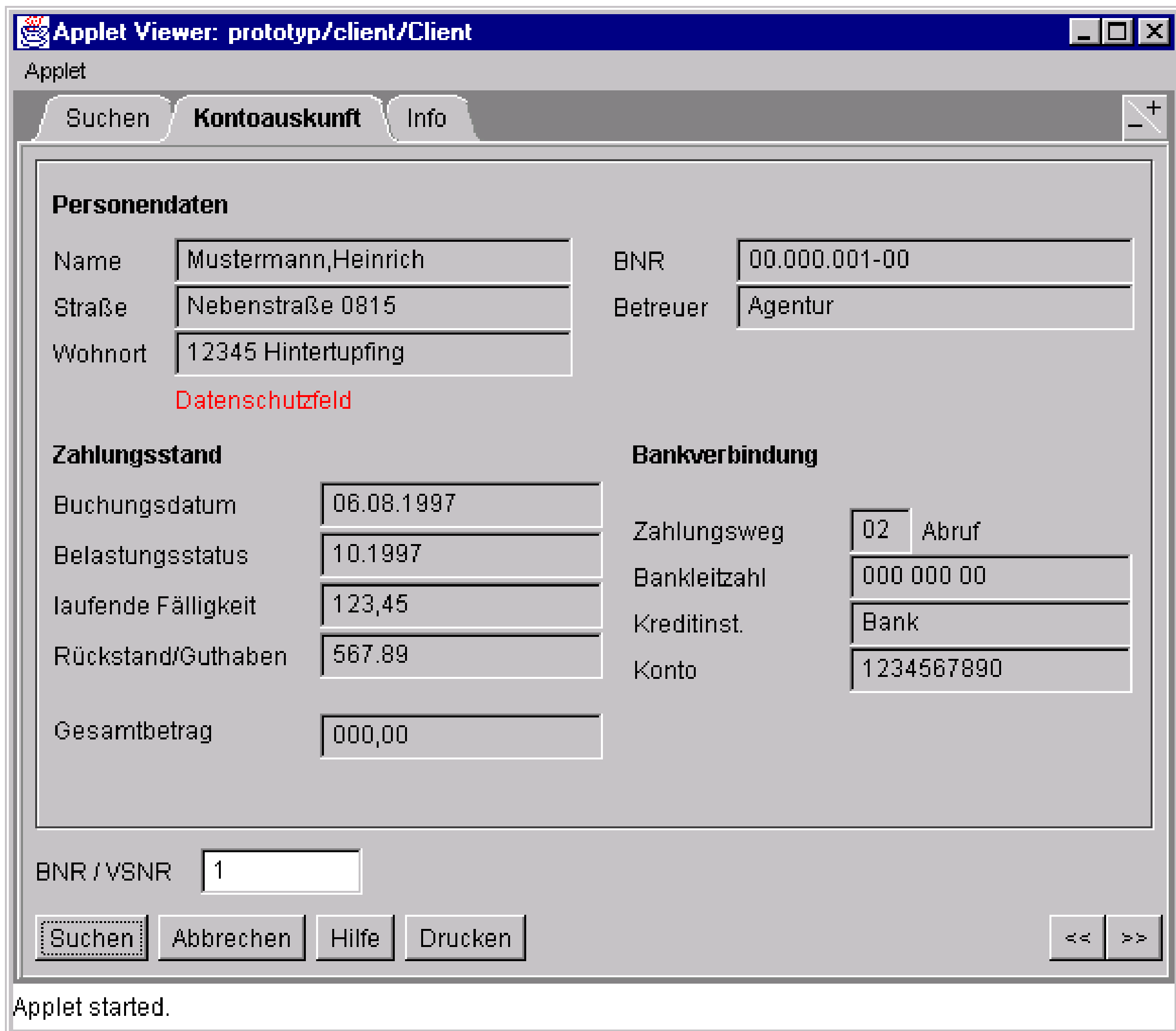


Abb. 10: Prototyp[19]

Der Java-Client kommuniziert per RMI (Remote Method Invocation) mit dem RMI-Server (siehe Abb. 11). Für die Datenkommunikation zwischen RMI-Server und dem Host-Programm TP 120 wurde MQSeries eingesetzt. MQSeries ist eine message-orientierte Middleware (MOM) von IBM. Für diesen speziellen Prototyp wurde nur auf ein Host-Programm zugegriffen. Generell besteht aber die Möglichkeit, durch Kombination von mehreren Host-Programmen neue Funktionalität bereitzustellen.

Im Rahmen dieses Forschungsprojekts wurde ein Fehlerbehandlungssystem benötigt, um für die Anwendungsprogrammierer eine einfache und einheitliche Möglichkeit der Fehlerbehandlung zu schaffen. Im folgenden Kapitel werden die Anforderungen an dieses System beschrieben.

---

[19] aus [GRAH97b], S. 77

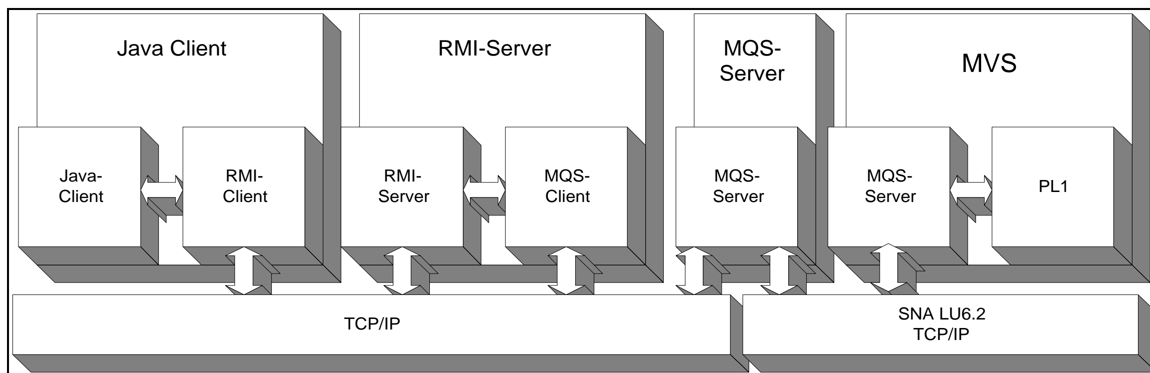


Abb. 11: Systemarchitektur des Prototyps[20]

4.2 Anforderungen an das Fehlerbehandlungssystem

In der Großrechner-Welt der HM (gemeint ist hiermit das Hostsystem mit MVS und PL/1-Programmierung) werden Softwarefehler, die Speicherfehler oder Division-by-Zero Fehler sind, von der Hardware zur Laufzeit entdeckt. Ein Transaktionssystem sorgt dafür, daß diese Fehler mit Programm- und ProgrammiererID zusammen ausgedruckt werden. Dieses Papier wird per Hauspost dem verantwortlichen Programmierer zugeschickt. Das zu entwickelnde Fehlerbehandlungssystem soll eine ähnliche Funktionalität zur Verfügung stellen.

Die erste Anforderung ist, daß das Fehlerbehandlungssystem in der Programmiersprache Java implementiert werden soll, da Java die für das Forschungsprojekt (siehe Kapitel 4.1) gewählte Programmiersprache ist. Dabei soll das Fehlerbehandlungssystem in das Ausnahme-Konzept von Java eingebaut werden.

Die zweite Anforderung stellt die Implementation des im vorigen Kapitel beschriebenen Vertragsmodells in Java dar, weil die Benutzung des Vertragsmodells die beste Möglichkeit ist, Fehler schnell zu lokalisieren.

Die dritte Anforderung besteht darin, daß der Anwendungsprogrammierer die Möglichkeit erhalten soll, auf unterschiedliche Weise auf einen Fehler zu reagieren. Dies kann ein kontrollierter Programmabbruch sein, der Versuch, das Programm fortzusetzen oder ein für ein bestimmtes Anwendungssystem spezifisches Verhalten. Das Fehlerbehandlungssystem soll ausschließlich ein Defaultverhalten vorschlagen.

Außerdem soll das System für eventuelle Neuerungen offen und leicht erweiterbar sein. Es sollten möglichst alle Kriterien des Acht-Punkte-Kataloges ([DONY88], siehe Kapitel 2.1) erfüllt werden.

Das Fehlerbehandlungssystem soll Informationen (wie z.B. Datum, Uhrzeit, ProgrammiererID, Methodenname, Klassenname, Ausnahmetyp) sammeln, um dem zuständigen Programmierer möglichst konkrete Hinweise über den aufgetretenen Fehler zu geben, damit dieser die zugrunde liegende Störung schnell beseitigen kann. Die Fehlermeldungen, die den Mitarbeitern der Fachabteilung angezeigt werden, sollen von der Fachabteilung selbst festgelegt werden, da sie für diesen Personenkreis verständlich sein müssen. Hierbei ist eine direkte Zusammenarbeit zwischen Softwareentwicklern und Fachabteilung notwendig. Die

[20] aus [GRAH97b], S. 76

Fehlertexte sollen auf Wunsch in einer anderen als der deutschen Sprache angezeigt werden können.

Eine weitere Anforderung an das Fehlerbehandlungssystem ist, daß dieses nicht nur auf das fehlerhafte Verhalten eines Programmteils (Programmfehler) reagieren soll, sondern zusätzlich noch auf das anormale Verhalten, das zwar spezifiziert ist, aber nicht zum normalen Verhalten zählt (siehe Kapitel 2.3). Hierzu gehören fehlerhafte Benutzereingaben. Als Default-Reaktion soll ein Dialog mit dem Benutzer erfolgen. Das fehlerhafte Verhalten einer Methode fällt dagegen in die Verantwortung des Programmierers und muß von ihm behoben werden. Um den Programmierer davon in Kenntnis zu setzen, soll als Default-Reaktion die Ausnahme zentral gespeichert werden. Diese Speicherung soll über eine Queue oder eine Datei möglich sein. Zusätzlich soll die Möglichkeit bestehen, die Meldung auf eine Bildschirmkonsole und auf einen Drucker auszugeben. Der Programmierer soll von dem Fehler in Kenntnis gesetzt werden, damit er die dem Fehler zugrunde liegende Störung beseitigen kann. Auf diese Weise können Störungen, die beim Testen der Software nicht gefunden wurden und in der Produktion als Fehler in Erscheinung treten, beseitigt werden.

Die in diesem Kapitel genannten Anforderungen bilden den Leistungsumfang des zu entwickelnden Fehlerbehandlungssystems.

4.3 Anforderungen an die Middleware

Im vorigen Abschnitt wurde davon abstrahiert, daß das Fehlerbehandlungssystem in einem heterogenen Umfeld funktionieren soll. Um dies zu ermöglichen wird eine Middleware benötigt. Bevor die Anforderungen an die Middleware diskutiert werden, soll als erstes der Middleware-Begriff definiert werden. Die kürzeste Definition von Middleware ist der Slash zwischen Client/Server[21]. Eine etwas detaillierte Middleware-Definition ist in [FISC97] zu finden:

> „Middleware is an enabling software technology that resides between business applications and the underlying layer of heterogenous platforms and network protocols. It enables elements of applications to interoperate network links in spite of the variety of underlying communication protocols, operating systems, hardware, system architectures, databases and other application services.“

Damit das Fehlerbehandlungssystem in einer verteilten Umgebung laufen kann, wird eine Middleware benötigt, die die Anwendung von den Kommunikationsprotokollen, dem Betriebssystem und der Hardware isoliert. An die Middleware werden genau wie an das Fehlerbehandlungssystem Anforderungen gestellt. Diese werden im folgenden beschrieben.

- Die Middleware sollte in jedem Fall hardware- und plattformunabhängig sein, damit das Fehlerbehandlungssystem in einem heterogenen Systemumfeld funktionieren kann. Zusätzlich sollten die wichtigsten Protokolle wie z.B. TCP/IP und SNA unterstützt werden. Wichtig in diesem Zusammenhang ist, daß die Middleware notwendige Datenformat-Konvertierungen erledigt, wenn die miteinander kommunizierenden Plattformen unterschiedliche Datenformate besitzen.
- Die Middleware sollte ein objektorientiertes High-Level API besitzen, das für die Programmiersprache Java angeboten wird. Das API sollte von der Netzwerkebene abstrahieren, so daß die Verteilungsproblematik vollständig vor dem Programmierer verborgen bleibt.

[21] aus [ORFA97c]

- Die Middleware sollte ein hohes Maß an Kompatibilität zu anderen Middleware-Produkten besitzen, damit ein Datenaustausch zwischen verschiedenen Middleware-Produkten möglich wird.
- Die Middleware sollte eine Möglichkeit der Persistenz beinhalten, da die Fehler zentral gespeichert werden sollen.
- Die Middleware sollte die Möglichkeit der asynchronen Kommunikation anbieten. Dies ist erforderlich, da während der Speicherung des Fehlers auf einem anderen Rechner, der Kontrollfluß im Programm fortgesetzt werden kann.
- Die Middleware sollte leicht erlernbar sein. Da für das Fehlerbehandlungssystem kein großer Funktionsumfang der Middleware benötigt wird, wäre es negativ, wenn erst ein vollständiges Verständnis der Middleware erforderlich wäre, um die eigentliche Aufgabe, nämlich das Versenden von Fehlern, eine lange Einarbeitungszeit kosten würde.
- Die Performance der Middleware ist unkritisch, da es nicht wichtig ist, ob die Fehler mit einiger Verzögerung oder sofort nach Entstehung gespeichert werden.
- Die Sicherheit der Middleware muß differenziert betrachten werden: Wichtig ist, daß alle Fehlermessages auch tatsächlich ihren Zielrechner erreichen. Es darf keine im Netzwerk verloren gehen. Der Schutz vor unberechtigtem Zugriff auf die Fehlermessages ist nur bedingt wichtig. Es gibt unbestritten sicherheitskritischere Anwendungen.
- Die Middleware, die dem Fehlerbehandlungssystem zugrunde liegen wird, sollte bereits einige Zeit in der Praxis erprobt und verbreitet sein, um sicherzustellen, daß die Middleware eine gewisse Akzeptanz bei den Anwendungprogrammierern besitzt und genügend ausgereift ist.
- Die Middleware sollte nach Möglichkeit in der HM verfügbar sein, da sonst erst eine Middleware gekauft werden muß und zusätzliche Anschaffungskosten anfallen.

Nachdem der Katalog für die Anforderungen feststeht, kann die Suche nach der geeigneten Middleware für das Fehlerbehandlungssystem beginnen. Da in der HM das MOM-Produkt MQSeries von IBM zur Verfügung steht, bietet es sich an, dieses Produkt für die Implementation zu benutzen. Um eine Vergleichs-Implementation zu erhalten, wurde das CORBA-Produkt Visibroker von Inprise, das vom Arbeitsbereich Softwaretechnik des Fachbereichs Informatik an der Universität Hamburg für diese Diplomarbeit zur Verfügung gestellt wurde, benutzt. In den beiden folgenden Kapiteln wird von den konkreten Produkten so weit wie möglich abstrahiert und die übergeordnete Middleware-Kategorie (MOM und CORBA) vorgestellt.

5 Message-orientierte Middleware am Beispiel von MQSeries

> „Jeder DAD (Distributed Application Development) braucht eine MOM (Message-orientierte Middleware).“[22]

Um dem Fehlerbehandlungssystem einen Verteilungsmechanismus zur Verfügung zu stellen, wird eine Middleware benötigt. Dieses Kapitel führt in die Message-orientierte Middleware (MOM) ein. Dabei wird speziell auf das Produkt MQSeries von IBM Bezug genommen.

MQSeries ist ein Message-Queuing-System, das von IBM entwickelt wurde. Es bietet eine flexible, schnelle und einfache Lösung für Program-to-Program-Kommunikation an und ist hardware- und betriebssystemunabhängig. Bei der Implementation wurde darauf geachtet, daß die exactly-once-Semantik erfüllt wird, d.h. daß dem Programmierer zugesichert wird, daß die gesendete Message tatsächlich genau einmal ankommt (mehr zu den Semantiken siehe [LAME94]). MQSeries erhielt den well-connected award 1997 (Network Computing)[23].

Um einen allgemeinen Einblick in die Funktionsweise message-orientierter Middleware zu geben, wird in Kapitel 5.1 eine Einführung in das Messaging und Queuing gegeben. Darauf aufbauend werden die Komponenten von MQSeries eingeführt (Kapitel 5.2). Im Anschluß daran wird deutlich gemacht, wie die Komponenten zusammenarbeiten, so daß Messages von einem Rechner zu einem anderen übertragen werden können (Message Routing, Kapitel 5.3). Danach wird die Anbindung an Java erläutert (MQSeries Client für Java, Kapitel 5.4). Zum Schluß folgt die Beschreibung der MQSeries-Architektur des Fehlerbehandlungssystems (Kapitel 5.5).

5.1 Einführung in das Messaging und Queuing

Messaging ist ein Konzept mit dessen Hilfe Softwarekomponenten Daten (Messages) austauschen können. Dieser Messageaustausch ist orts- und zeitunabhängig: Der Messagesender muß weder den Aufenthaltsort des Messageempfängers kennen noch müssen die Messagepartner aufeinander warten. Die Messages können zu beliebigen Zeitpunkten gesendet werden (**No-Wait-Kommunikation**). Ermöglicht wird dies durch die asynchrone Kommunikation der Messagepartner. Durch das No-Wait-Grundprinzip werden Parallelität und konkurrierende Zugriffe ermöglicht, die eine Anwendung beschleunigen können, wenn mehrere voneinander unabhängige Aufgaben zu bearbeiten sind. Der Ablauf des Messagings besteht aus zwei Teilen: Zuerst plaziert der Messagesender eine Message an einem definierten Ort. Zu einem späteren Zeitpunkt holt der Messageempfänger sie von diesem Ort ab. Es findet also kein direkter Kontakt zwischen Sender und Empfänger statt. Ein Nachteil des Messagings ist, daß die Empfangsreihenfolge der Messages von der Senderreihenfolge abweichen kann. Wird Messaging ohne Queuing benutzt, so findet der Messageaustausch über einen gemeinsamen Speicherbereich statt. Die Messages sollten in diesem Fall schnell empfangen werden, um Messageverlusten durch vorzeitiges Überschreiben des Speichers durch andere Programme vorzubeugen.

Der wesentliche Bestandteil des **Queuingkonzepts** sind Queues, die als Zwischenspeicher für Messages fungieren. Die Kommunikationspartner kommunizieren über diese Queues und besitzen daher keine feste Kommunikationsverbindung (**No-Connection-Kommunikation**). Die Kommunikation der Entitäten erfolgt indirekt und mit unterschiedlicher Geschwindigkeit.

---

[22] von MOMA (Message-oriented Middleware Assoziation)

[23] aus [TECH98]

Queuing ist eine Zeitanpassungstechnik zum Sichern von Messages bis der Empfänger bereit ist, diese zu empfangen. Der größte Vorteil des Queuings ist, daß sich die Verarbeitungszeit von Sender und Empfänger überlappen und dadurch summa summarum verkürzen kann. Weitere Vorteile bestehen in der einfachen Netzwerkadministration mit geringerer Netzwerklast, da keine Verbindungen aufgebaut werden müssen, und der Möglichkeit der Lastbalance und Lastverteilung durch die Teilung von Datenströmen, wenn mehrere Queues eingesetzt werden. Ein gängiges Beispiel für Queuing aus dem Alltag stellt ein Anrufbeantworter dar.

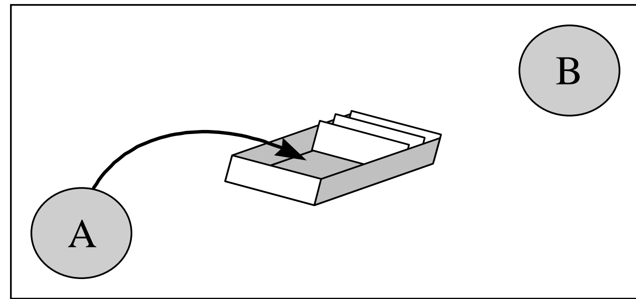


Abb. 12: Messageaustausch mit inaktivem Empfänger[24]

Normalerweise benutzen MOM-Produkte beide Konzepte miteinander vereint. Man spricht dann von **Message Queuing**: Der Sender stellt Messages in eine Queue (put) und der Empfänger holt sich die Message zu einem späteren Zeitpunkt aus der Queue heraus (get). Die Queue gewährleistet eine totale Entkopplung von Sender und Empfänger und bietet auf Wunsch eine persistente Speicherung der Messages an. Wenn das Empfängerprogramm bereit ist, kann es die Messages nach und nach einlesen und verarbeiten. Abb. 12 ist zu entnehmen, wie Programm A Messages in die Queue stellt, während Programm B inaktiv ist. Daran kann man sehen, daß Queues unabhängig von den Anwendungsprogrammen existieren.

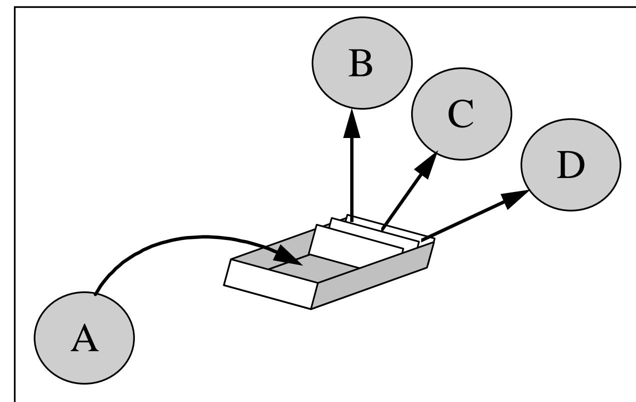


Abb. 13: 1:n-Beziehung von Anwendungen[25]

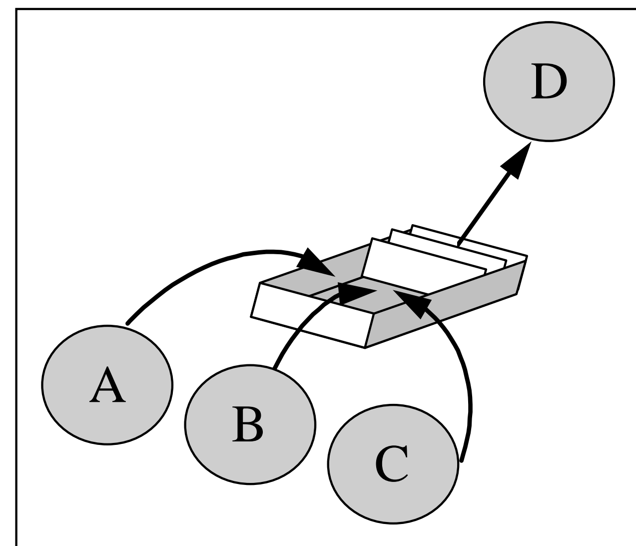


Abb. 14: n:1-Beziehung von Anwendungen[26]

Bisher wurde nur die Kommunikation zwischen zwei Programmen beschrieben. MOMs unterstützen aber auch komplexere Anwendungsstrukturen. In Abb. 13 ist eine 1:n-Beziehung von Programmen dargestellt. Programm A verteilt die Messages auf B, C und D. Dabei können B, C und D Kopien derselben Anwendung zwecks Lastverteilung sein oder drei verschiedene Programme, die voneinander unabhängige Aktivitäten parallel ausführen, um Zeit zu gewinnen.

[24] aus [IBM94] S. 5
[25] aus [IBM94] S. 6
[26] aus [IBM94] S. 7

Entsprechend gibt es die Möglichkeit der n:1-Beziehung, bei der mehrere Programme Messages in dieselbe Queue stellen, die von nur einem Programm abgearbeitet wird (Abb. 14). Dies entspricht mehreren Clients, die Anfragen an einen Server stellen. Der Server kann die Messages nach dem FIFO-Verfahren (first in, first out) oder nach vergebenen Prioritäten lesen. Die drei vorgestellten Basistechnologien können miteinander kombiniert werden und so komplexe über das Netz verteilte Architekturen bilden.

Erfolgt die Kommunikation bidirektional, sollten mehrere Queues eingesetzt werden, damit die Programme nicht die Messages, die sie selber in die Queue geschrieben haben, wieder aus der Queue lesen und das eigentliche Empfängerprogramm die Messages deswegen nicht erhält. Auf diese Weise läßt sich zwischen Eingangs-Queues und Ausgangs-Queues unterscheiden.

5.2 Die Komponenten von MQSeries

Die zentrale Komponente von jedem MOM-Produkt und somit auch von MQSeries ist die Message. Sie enthält die Daten, die die Anwendungen austauschen wollen. In Kapitel 5.2.1 wird die Message und ihre Typen näher beleuchtet. Um die Anwendungen von der Netzwerkschicht zu befreien, gibt es den Queue-Manager, der auf allen Servern installiert sein muß und die gesamte Kommunikation im Netzwerk abwickelt (Kapitel 5.2.2). Damit Messages auf ihrem Transport nicht verloren gehen, wird ein Persistenz-Mechanismus benötigt, der die Messages auf dem Weg zur Zielanwendung speichert. Dieses sind die Message-Queues, die in Kapitel 5.2.3 erläutert werden. Die Kommunikationsverbindung zwischen den Queue-Managern einerseits und zwischen Client und Queue-Manager andererseits ist der Channel, der das Thema von Kapitel 5.2.4 bildet. Für den Zugriff auf die Queues wird das Message Queuing Interface (MQI, Kapitel 5.2.5) benötigt. Dadurch wird den Anwendungsprogrammen die Unabhängigkeit von Plattformen und Programmiersprachen gewährleistet. Die Programme können leicht im Netzwerk verschoben werden, da sie alle das gleiche Interface benutzen.

5.2.1 Message

Eine Message besteht aus einem String, der die Daten enthält, die zwischen zwei Anwendungen ausgetauscht werden. Dieser String besteht aus zwei Teilen, den **Messagedaten** und dem **Messageheader** (siehe Abb. 15). Die Messagedaten sind die Anwendungsdaten, die im Falle des Fehlerbehandlungssystems die gesammelten Fehlerinformationen enthalten. Der Messageheader besteht aus MQSeries internen Daten. Diese enthalten Eigenschaften einer Message, die der Queue-Manager und der Empfänger braucht, um zu entscheiden, wie die Message verarbeitet werden soll.

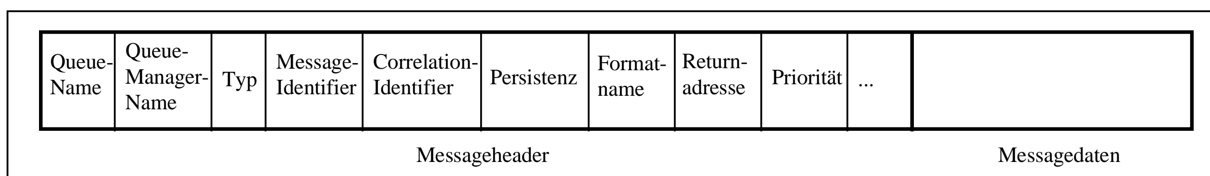


Abb. 15: Aufbau einer Message

Die wichtigste Information, die im Header einer Message steht, ist der Queue-Name und der Queue-Manager-Name, für den die Message bestimmt ist. Der Queue-Name ist die Identifikation der Ziel-Queue der Message. Der Queue-Manager-Name identifiziert den Queue-Manager, der die Queue besitzt. Der vollständige Empfänger lautet Queue-Name@Queue-Manager-Name.

Es gibt vier verschiedene Arten von Messagetypen: Ein **Request** ist eine Message, die von einem Programm benutzt wird, wenn es sich um eine Datenanfrage an ein anderes Programm handelt. Die Antwort auf ein Request ist ein **Reply**. Eine Message, auf die keine Antwort folgt, wird **One-Way-Message** oder Datagram genannt. Ein **Report** wird benutzt, wenn etwas Unvorhergesehenes passiert, beispielsweise wenn die Daten einer Reply-Message nicht brauchbar sind, wenn eine Message nicht zu seiner Ziel-Queue geliefert werden konnte oder wenn eine Bestätigung über die Ankunft der Message in der Ziel-Queue (confirmation on arrival) oder das Empfangen der Message vom Anwendungsprogramm bestätigt werden soll (confirmation on delivery). Es besteht die Möglichkeit, zusätzliche Messagetypen zu definieren. Der Messageheader enthält die Information, um welchen Messagetyp es sich handelt, damit das Empfängerprogramm erfährt, was von ihm verlangt wird. Für den Queue-Manager ist der Messagetyp uninteressant. Für das Fehlerbehandlungssystem wird der One-Way-Messagetyp benötigt, da die Fehler von den Anwendungsprogrammen zu einer zentralen Queue zur Speicherung gesendet werden.

Damit die einzelnen Messages eindeutig identifiziert werden können, enthält jede Message einen **Messageidentifier**. Dieser Identifier wird vom Queue-Manager vergeben. Zusätzlich enthält jede Message einen **Correlationidentifier**. Im Gegensatz zum Messageidentifier kann dieser frei benutzt und vom Programm vergeben werden. Während der Messageidentifier für jede einzelne Message eindeutig ist, klassifiziert der Correlationidentifier die Messages in einzelne Gruppen, die jeweils verwandte Messages enthalten. Zum Beispiel sollte ein Request einen Correlationidentifier erhalten, den das Serverprogramm in den Reply kopiert, damit das anfragende Programm die Reply-Message zuordnen kann. Mit Hilfe der beiden Identifier kann ein Anwendungsprogramm eine Queue nach Messages mit bestimmten Identifiern fragen.

Es gibt **persistente** und **nicht persistente** Messages. Wird eine Message als persistent definiert, so kann sie nach einem Restart aus einer MQSeries Log-Datei wiederhergestellt werden, andernfalls geht sie verloren.

Durch den **Formatnamen** teilt der Sender dem Empfänger mit, wie die Message zu interpretieren ist, indem der Name einer Datenstruktur übermittelt wird. Diese Datenstruktur muß von beiden Seiten verstanden werden.

Handelt es sich bei einer Message um eine Request-Message, so enthält die Message die **Returnadresse**, die den Namen der Queue enthält, zu der der Reply gesendet werden soll. Da Programme nicht direkt miteinander kommunizieren, wird der Name des Empfänger-Programms nicht mit angegeben.

Messages können verschiedene **Prioritäten** haben. Der Defaultwert ist der Prioritätswert der Queue, in die die Messages gestellt wird.

5.2.2 Queue-Manager

Das Herz von MQSeries ist der Queue-Manager. Es handelt sich dabei um einen Systemservice, der das Message-Queuing für die Anwendungsprogramme bereitstellt. Über das Message Queuing Interface (MQI) können Anwendungsprogramme direkt mit dem Queue-Manager kommunizieren (siehe Abb. 16). Queue-Manager und das MQI sind für diverse Plattformen von UNIX und MVS bis hin zu OS/2, DOS und Windows und viele andere erhältlich. Damit ist die für das Fehlerbehandlungssystem geforderte Plattformunabhängigkeit gewährleistet.

Das Message-Queuing und damit das MQI kann den Anwendungsprogrammen auf zwei Arten verfügbar gemacht werden: Entweder muß auf den Rechnern, auf denen die Anwendungsprogramme ablaufen sollen, ein Queue-Manager installiert sein oder es muß eine Verbindung zu einem Queue-Manager bestehen. Es besteht aber auch die Möglichkeit, mehrere Queue-Manager auf einem Rechner laufen zu lassen, um beispielsweise Produktion und Entwicklung aus Sicherheitsgründen zu trennen. In diesem Fall spricht man dann von einem Queue-Manager-Exemplar. Jeder Queue-Manager hat einen netzwerkweit eindeutigen Namen, damit eine eindeutige Zuordnung möglich ist.

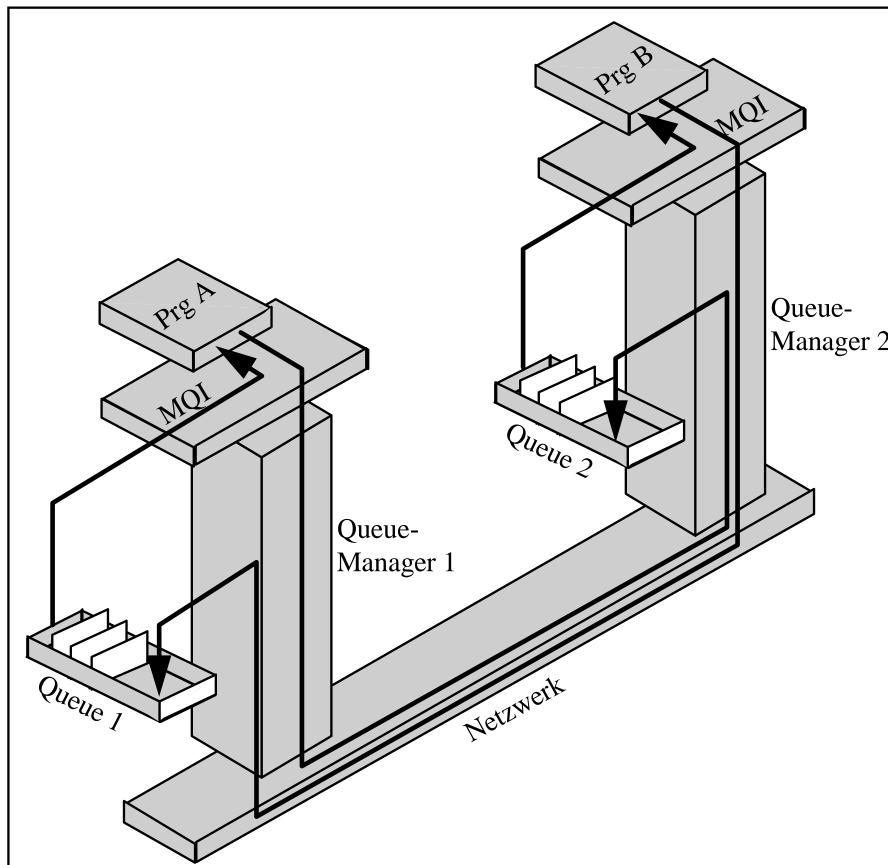


Abb. 16: Programme kommunizieren über das MQI mit dem Queue-Manager[27]

Wenn ein Anwendungsprogramm über das MQI eine Message versenden möchte, so prüft der lokale Queue-Manager, ob die Ziel-Queue eine seiner eigenen Queues ist oder ob es sich um eine Queue eines anderen (entfernten) Queue-Managers handelt. Im ersten Fall kann der Source-Queue-Manager die Message direkt in die Ziel-Queue stellen, im zweiten Fall muß er die Message zu dem entfernten Queue-Manager (Target-Queue-Manager) weiterleiten. Abb. 16 zeigt, daß das Programm A eine Message in die Queue 2 stellt, die zum Queue-Manager 2 gehört. Programm B kann diese Message später lesen. Der umgekehrte Fall ist der Graphik ebenfalls zu entnehmen.

Der Queue-Manager kann drei Arten von Objekten besitzen: Queues, Channels und Process-Definitions. Queues und Channels werden in den folgenden Kapiteln ausführlich beschrieben. Process-Definitions sind Prozesse, die beispielsweise als Reaktion auf eine ankommende Message gestartet werden. Es könnte sich um ein Programm handeln das immer gestartet

[27] frei nach [IBM94] S. 11

wird, wenn Messages in einer Queue ankommen und das nach Abarbeitung dieser Messages terminiert.

Aufgrund der Existenz des Queue-Managers brauchen Anwendungsprogrammierer nicht zu wissen, auf welchem Rechner das Programm läuft, mit dem sie kommunizieren wollen. Die Message wird nur in eine Queue gestellt; den Transport erledigt der Queue-Manager. Dazu benutzt er TCP/IP.

Der Queue-Manager besitzt Attribute, die nach Erzeugung nicht mehr änderbar sind: Der Queue-Manager-Name kann nicht geändert werden, um die Eindeutigkeit des Namens im Netz nicht zu gefährden. Ebenfalls kann die Plattform, auf der der Queue-Manager läuft, die maximale Priorität, die die Messages erhalten können, die von diesem Queue-Manager verarbeitet werden, der Queue-Name, zu dem Anwendungsprogramme MQSeries-Kommandos senden können, die maximale Messagelänge und die Möglichkeit, Synchronisationspunkte setzen zu können, nicht mehr geändert werden.

Zur Laufzeit änderbare Attribute sind die textuelle Beschreibung des Queue-Managers, der Name der Dead-Letter-Queue (Queue für nicht zustellbare Messages), Name der Default-Transmission-Queue (siehe 5.2.3.3) und die maximale Anzahl von gleichzeitigen Verbindungen zu Anwendungsprogrammen.

5.2.3 Message-Queues

Eine Message-Queue ist ein Speicherbereich, der vom Queue-Manager eingerichtet wird, um Messages, die von einem Programm zu einem anderen transportiert werden, zwischenzuspeichern. Durch diese Zwischenspeicherung kann das Anwendungsprogramm, nachdem es eine Message in die Queue eingefügt hat, mit seiner Verarbeitung fortfahren und zu einem späteren Zeitpunkt eine mögliche Antwort aus der Queue lesen (asynchrone Verarbeitung). Der Zugriff erfolgt in der Regel nach dem FIFO-Verfahren, d.h. Messages werden von hinten eingefügt (put) und von vorne entnommen und gelöscht (get). Queues können sich im Hauptspeicher oder in einem persistenten Sekundärspeicher befinden. Für das Fehlerbehandlungssystem wird eine Queue benutzt, die sich im Sekundärspeicher befindet, damit die Fehlermessages gespeichert werden können.

Jede Queue gehört zu genau einem Queue-Manager, der sie verwaltet. Innerhalb eines Queue-Managers besitzt jede Queue einen eindeutigen Namen. Zusätzlich zu diesem Queue-Namen kann eine Queue andere Namen haben, die die Anwendungsprogramme verwenden, die die Queue benutzen. Durch diese Namenstrennung wird erreicht, daß die Queue-Namen der Programme bei Änderungen am Queue-Namen des Queue-Managers oder dem Aufenthaltsort der Queue eindeutig der richtigen Queue zugeordnet werden können.

Über das MQI können Anwendungsprogramme Messages aus den Queues lesen und Messages in die Queues schreiben. Es besteht auch die Möglichkeit, Kopien einer Message anzufordern. Die Message wird in diesem Fall nicht gelöscht, so daß erneut auf sie zugegriffen werden kann. Es sollte jedoch darauf geachtet werden, daß Messages, die nicht mehr benötigt werden, gelöscht werden, so daß Ressourcen gespart werden können. Da Anwendungsprogramme ausschließlich über das MQI auf die Queues zugreifen können, wird die physikalische Verwaltung vor den Anwendungsprogrammen verborgen.

Queues haben eine Vielzahl von Attributen. Das wichtigste Attribut ist der eindeutige Queue-Name. Zusätzlich wird der Typ der Queue gespeichert (mehr über Queue-Typen siehe

folgende Unterkapitel). Eine Queue-Definition enthält Angaben über die maximale Anzahl von Messages, die gleichzeitig in der Queue gespeichert werden können und die maximale Länge einer Message. Anwendungsprogramme können per MQI eine Queue nach der aktuellen Anzahl der Messages, die in der Queue gespeichert sind, fragen. Eine Queue enthält die Anzahl der Programme, die gleichzeitig Messages von ihr lesen können. Weiterhin enthält eine Queue diverse Defaultwerte für Messages, wie z.B. Persistenz und Priorität.

Queues können in lokale und entfernte Queues klassifiziert werden. Ob eine Queue lokal oder entfernt ist, hängt dabei von der Perspektive des Betrachters ab (siehe Kapitel 5.2.3.1 und 5.2.3.2). Lokale Queues können in normale Queues und Transmission-Queues (siehe Kapitel 5.2.3.3) differenziert werden. Es gibt noch eine Reihe weiterer Queue-Arten, die aber für das Fehlerbehandlungssystem nicht relevant sind und daher hier nicht näher beleuchtet werden.

5.2.3.1 Lokale Queues

Lokale Queues befinden sich bei dem Queue-Manager, der mit der gerade betrachteten Anwendung verbunden ist. Alle Messages, die für lokale Programme bestimmt sind, befinden sich in der lokalen Queue. Dabei handelt es sich um Messages, die entweder von Programmen desselben Rechners stammen oder um eingehende Messages, die von Programmen entfernter Rechner gesendet wurden.

Anwendungsprogramme können Messages in lokale Queues schreiben bzw. aus lokalen Queues lesen. Die Queue 1 aus Abb. 16 ist eine lokale Queue des Queue-Managers 1. Deshalb kann Programm A auf die Messages von Queue 1 zugreifen.

5.2.3.2 Entfernte Queues

Entfernte Queues sind Queues eines entfernten Queue-Managers. Jede entfernte Queue, zu der vom lokalen Queue-Manager aus eine Message gesendet werden soll, braucht eine lokale Definition der entfernten Queue.

Anwendungsprogramme können Messages in entfernte Queues schreiben, aber nicht von entfernten Queues lesen. Die Queue 2 aus Abb. 16 ist eine entfernte Queue des Queue-Managers 1. Deshalb kann Programm A die Messages von Queue 2 nicht lesen, sondern nur Messages in die Queue 2 schreiben.

5.2.3.3 Transmission-Queues

Messages, die für Queues eines anderen Queue-Managers bestimmt sind, werden zunächst in eine lokale Transmission-Queue gestellt. Zu einem geeigneten Zeitpunkt werden die Messages der Transmission-Queue zu dem entfernten System gesendet (siehe Abb. 17). Dort werden sie in einer lokalen Queue gespeichert. Transmission-Queues enthalten also ausgehende Messages aus Sicht des lokalen Systems. Sie werden als Zwischenschritt benutzt, um Messages zu einer mit der Transmission-Queue assoziierten entfernten Queue zu senden. Eine Kopie der Message bleibt solange in der Transmission-Queue gespeichert, bis sie korrekt zur entfernten Queue übertragen wurde. Auf diese Art wird die exactly-once-Semantik realisiert.

Normalerweise besitzt ein Queue-Manager für jeden entfernten Queue-Manager, mit dem kommuniziert werden soll, eine Transmission-Queue. Der Queue-Manager stellt die Messages jeweils in die Transmission-Queue, die für den Queue-Manager zuständig ist, dem die Ziel-Queue gehört. Transmission-Queues sind für die Anwendung transparent. Sie werden ausschließlich intern vom Queue-Manager benutzt.

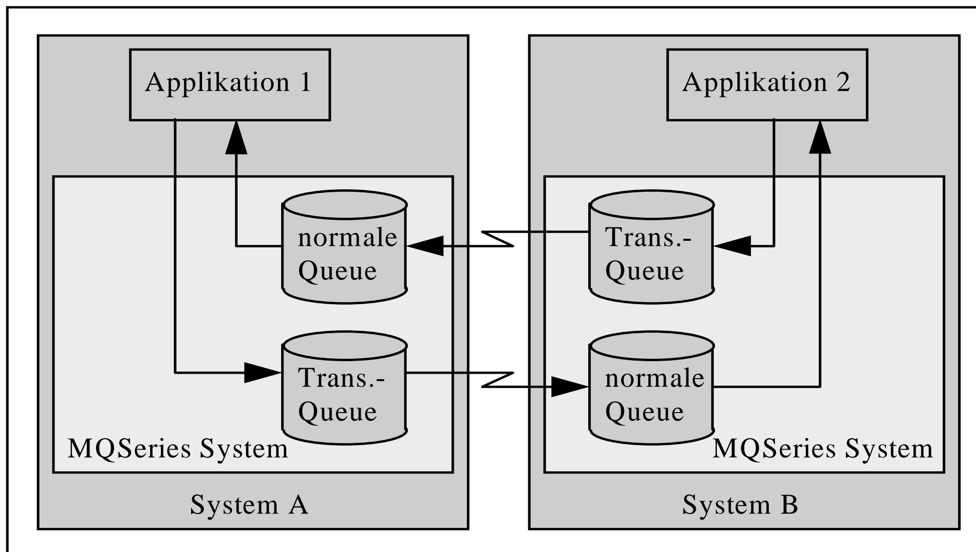


Abb. 17: Queue-Typen[28]

5.2.4 Channels

Channels sind logische Kommunikationsverbindungen. Generell werden zwei Arten von Channels unterschieden: **MQI-Channels** und **Message-Channels** (siehe Abb. 18). Ein MQI-Channel verbindet einen MQI-Client mit einem Queue-Manager. MQI-Clients haben keinen eigenen Queue-Manager, so daß alle MQI-Calls zuerst zum Server übertragen werden und anschließend dort ausgeführt werden. Das Ergebnis eines MQI-Calls wird zum Client zurückgegeben. Aus diesem Grund sind MQI-Channels immer bidirektionale Channels.

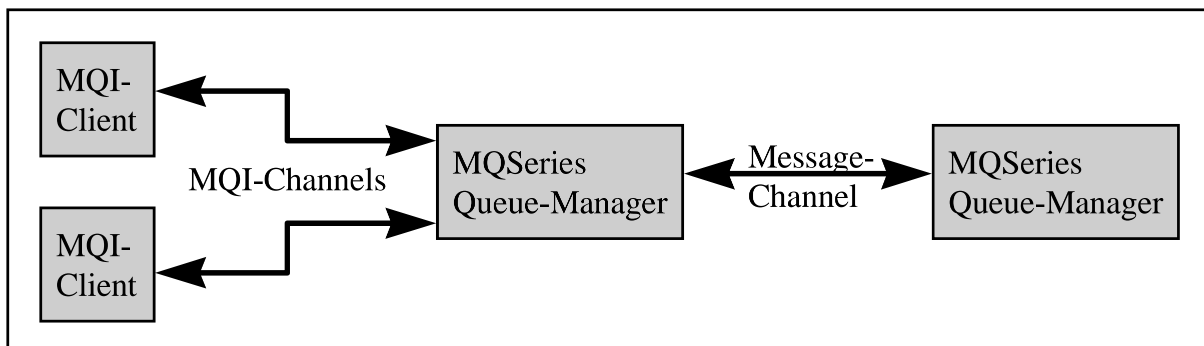


Abb. 18: MQI- und Message-Channels[29]

Message-Channels (im folgenden einfach als Channels bezeichnet) hingegen verbinden zwei Queue-Manager. Messages können über einen Channel nur in eine Richtung versendet werden. Daher werden bei einem Message-Austausch zwei Channels benötigt. Message-Channels sind langsamer als MQI-Channels.

Jede Seite eines Channels wird durch einen MCA (Message Channel Agent) gesteuert. MCAs sind Programme, die Messages von einer Transmission-Queue zu einem Kommunikations-Link transportieren und umgekehrt.

Ein MCA kann als **Caller** oder **Responder** agieren. Ein Caller MCA startet einen Channel durch Senden von einem Connection-Request zu einem Responder MCA. MQSeries bietet ein

[28] aus [IBM93] S. 8
[29] aus [BRAS97] S. 62

Programm namens Channel-Listener an, das auf Connection-Requests wartet. Der Connection-Request enthält die Adresse von dem Ziel-MCA, mit Hilfe derer der Channel-Listener den Ziel-MCA starten kann. Wenn der Responder-MCA einen Connection-Request empfängt, antwortet er dem Caller. Der Channel ist anschließend bereit, MQSeries Messages zu transportieren.

Ein Channel ist durch zwei kompatible MCAs definiert, an jeder Seite des Channels befindet sich einer. Sie müssen denselben Namen haben, damit sie als Paar identifizierbar sind. Die Definition bestimmt, welche Seite Messages sendet und welche Seite Messages empfängt. Es gibt vier Gruppen von Channel-Definitionen: **Sender** (sendet Messages), **Receiver** (empfängt Messages), **Server** (sendet Messages) und **Requester** (empfängt Messages). Sender, Server und Requestor können caller oder responder sein, Receiver ist immer responder und kann somit keinen Channel starten. Eine Channelseite muß ein Sender oder Server sein, um Messages von der Transmission-Queue zum Channel zu übertragen, die andere Seite muß ein Receiver oder Requester sein, um die Messages vom Channel in die Destination-Queue zu stellen. Es gibt mehrere denkbare Kombinationsmöglichkeiten:

1. **Sender-Receiver**: Der Sender (Caller) ruft den Receiver (Responder), um den Channel zu starten und sendet nach erfolgreichem Verbindungsaufbau die Messages von seiner Transmission-Queue zum Receiver, der die Messages in die Destination-Queue stellt (z.B. E-Mail).
2. **Requester-Server**: Der Requester ist Caller, der Server Receiver. Der Server sendet nach erfolgreichem Verbindungsaufbau die Messages (z.B. Mail-Programm, daß zum Sammeln der Mail einen Call absetzt).
3. **Requester-Sender**: Requester ist Caller, Sender ist Responder. Der Sender terminiert die Verbindung und wird danach zum Caller. Er ruft den Requester, der zum Responder wird. Der Sender sendet die Messages als call-back (z.B. Mail-Applikation, die den Benutzern erlaubt, Mails mit Initial-Calls anzustoßen. Anschließend macht die Applikation ein call-back, um die Mails zu liefern. Die Benutzer brauchen für die Übertragung nicht zahlen).

Sender-Requester, **Server-Receiver** und **Server-Requester** verhalten sich genau wie Sender-Receiver. Alle weiteren Kombinationsmöglichkeiten sind ungültig, da der Receiver kein Caller sein kann. Das Fehlerbehandlungssystem benutzt die MCA-Kombination Sender-Receiver.

Die beiden Enden des Channels sind auf verschiedenen Queue-Managern definiert, deshalb können sie verschiedene Attribute besitzen. Nicht alle sind kompatibel. Deshalb gibt es beim Verbindungsaufbau eine Verhandlung zwischen den beiden MCAs. Können sich die beiden MCAs nicht einigen, kommt keine Verbindung zustande.

Channels werden in Channel-Gruppen (maximal 32 Channels pro Gruppe) zusammengefaßt, die zu einem Queue-Manager gehören. Die Channel-Gruppe der Caller-Seite besitzt einen Caller-MCA, der gestartet wird, wenn die Channel-Gruppe gestartet wird. Die Channel-Gruppe der Responder-Seite enthält nur den

Channels	Gruppe auf Caller-Seite	Gruppe auf Responder-Seite
Sender-Server	Sender	Listener
Sender-Receiver, Requester-Server	Sender, Requester	Listener
Sender-Receiver, Requester-Server mit Call-Back	Sender, Requester, Listener	Listener

Abb. 19: Beispiel für Channel-Gruppen

Listener. Dieser startet die Responder-MCAs. Abb. 19 zeigt dazu ein Paar Beispiele.

5.2.5 Message Queuing Interface (MQI)

Das Message Queuing Interface ist das API (Applikation Programming Interface), über das das Fehlerbehandlungssystem mit MQSeries kommuniziert. Es stellt eine Menge von Befehlen zur Verfügung, die benötigt werden, um die Services des Queue-Managers zu nutzen. Das MQI befreit das Fehlerbehandlungssystem von Wissen über die interne MQSeries Struktur. Das MQI gibt es für verschiedene Programmiersprachen, unter anderen COBOL, PL/1, C, C++ und Java.

Das MQI enthält elf Befehle. Der erste Befehl, der ausgeführt werden muß, ist der MQCONN-Call (connect). Durch diesen Call bindet sich die Applikation an den lokalen Queue-Manager. Ist der Call erfolgreich, returniert der Queue-Manager ein sogenanntes Connection-handle, ein Identifier, der bei jedem weiteren Call übergeben werden muß. Eine Applikation kann nur mit einem Queue-Manager zur Zeit verbunden sein und daher auch nur ein Connection-handle zur Zeit besitzen.

Als nächster Befehl bietet sich der MQOPEN-Call an, mit dem die gewünschte Queue geöffnet wird. Die Open-Optionen geben an, ob Messages geschrieben, gelesen oder gelöscht werden oder ob Attribute der Queue abgefragt oder gesetzt werden sollen. Ist dieser Call erfolgreich, so übergibt der Queue-Manager der Applikation ein Object-handle. Da eine Applikation gleichzeitig mehrere Queues geöffnet haben kann, kann sie mehrere Object-handles besitzen. Es besteht sogar die Möglichkeit, dieselbe Queue mehrfach zu öffnen. In diesem Fall erhält man entsprechend viele Object-handles für dieselbe Queue. Dies ist z.B. sinnvoll, wenn ein Object-handle zum Lesen und ein anderes zum Löschen von Messages benutzt wird.

Im Anschluß daran kann eine Applikation Messages in die Queue schreiben (MQPUT) oder Messages aus der Queue lesen (MQGET). Dabei wird sowohl der Connection-handle als auch der Object-handle übergeben, der die entsprechende Queue identifiziert. Beim MQPUT-Call müssen noch der Messagetyp, einige Optionen (z.B. MessageID und CorrelationID) und die Messagedaten übergegeben werden. Beim MQGET-Call kann entweder die erste Message der entsprechenden Queue gelesen werden (FIFO oder FIFO mit Prioritäten, abhängig von den Einstellungen der Queue) oder die erste Message, die die übergebene MessageID und/oder CorrelationID besitzt. Es kann eine Wartezeit angegeben werden, die gewartet werden soll, wenn keine passende Message in der Queue gefunden wird. Zusätzlich besteht die Möglichkeit von periodischen Calls, mit denen die Ankunft einer Message festgestellt werden kann (polling). Der Queue-Manager fügt beim Ausführen eines Calls einen Message-Deskriptor hinzu, der die für die Übertragung notwendigen Daten enthält.

Wenn das sendende Programm auf einer anderen Plattform als das empfangende Programm läuft, so wird möglicherweise eine Datenkonvertierung notwendig. Die Konvertierung kann entweder vom sendenden Queue-Manager oder vom empfangenden Queue-Manager übernommen werden. Dies wird in einem Attribut des Channels festgelegt.

Falls in eine Queue nur eine Message geschrieben werden soll, so kann der MQPUT1-Call benutzt werden. Dieser Call faßt das Öffnen, Schreiben und Schließen einer Queue zusammen.

Zum Abfragen von Informationen über eines der Queue-Manager-Objekte dient der MQINQ-Call (inquire).. Das Ändern von Attributen ist nur bei Queues möglich. Dazu gibt es den MQSET-Call.

Um Synchronisationspunkte zu setzen, steht der MQCMIT-Call (commit) zur Verfügung. Dieser Call beendet die aktuelle Arbeitseinheit (unit of work). Alle Messages, die Teil der Arbeitseinheit waren, werden anderen Applikationen verfügbar gemacht. Die empfangenen Messages dieser Arbeitseinheit werden gelöscht. Ein Rollback kann mit dem MQBACK-Call gestartet werden. Der Queue-Manager löscht daraufhin alle Messages, die seit dem letzten Synchronisationspunkt gesendet wurden und setzt alle Messages zurück, die seit dem letzten Synchronisationspunkt empfangen wurden. Mittels dieser zwei Calls können sichere Transaktionen benutzt werden. Die PUT-, PUT1- und GET-Calls die zur Arbeitseinheit dazugehören, müssen die Synchpoint-Option setzen.

Der MQCLOSE-Call schließt die Queue. Das Object-handle verliert seine Gültigkeit. Der MQDISC-Call (disconnect) löst die Verbindung zum Queue-Manager (disconnect). Dadurch wird der Connection-handle ungültig.

Im Falle eines Fehlers bei Ausführung eines MQI-Calls returniert der Queue-Manager einen CompletionCode (success, partial completion oder failure) und einen ReasonCode, der genauere Hinweise auf den Fehler enthält.

Bei jedem der genannten Calls prüft der Queue-Manager, ob die Anwendung für den jeweiligen Call mit den übergebenen Optionen autorisiert ist. Falls dies nicht der Fall ist, liefert er einen entsprechenden Returncode zurück.

5.3 Message Routing

Im einfachen lokalen Fall wird der Queue-Name bei einem OPEN-Call in die lokale Queue und der Queue-Manager-Name in den lokalen Queue-Manager gemappt. Soll die Message zu einer entfernten Queue gesendet werden, so wird der Queue-Name in den entfernten Queue-Namen und der Queue-Manager-Name in eine Transmission-Queue, die die Message zum entfernten System sendet, gemappt (siehe Anhang 10.2).

Ein Beispiel-Ablauf für das Message-Routing könnte wie folgt aussehen (siehe Abb. 20):

1. Das MQSeries-System erhält eine Message mittels MQPUT von Applikation 1. Das übergebene object-handle gehört zur remote Queue, in die die Message gestellt wird.
2. Die remote Queue ist eine entfernte Definition der lokalen Queue von Queue-Manager B. Sie enthält in ihrer Definition den entfernten Queue-Manager Namen, den Namen der lokalen Queue auf dem System B und den Namen der Transmission-Queue, die über einen Channel mit dem Queue-Manager B verbunden ist. Die Message erhält einen Transmission-Header, der den Queue-Manager Namen B und den Namen der lokalen Queue des Systems B enthält, und wird in die Transmission-Queue gestellt.
3. Mit Hilfe des MCPs (Message Channel Protocol) wird die Message über einen Channel zum entfernten Queue-Manager gesendet.
4. Der entfernte Queue-Manager stellt die Message in die lokale Queue mit dem Queue-Namen aus dem Transmission-Header und entfernt den Header.
5. Die Applikation 2 kann die Message erhalten, wenn sie einen MQGET-Call absetzt. Wenn diese Applikation eine Antwort senden möchte, so geschieht dies entsprechend über die remote Queue und die Transmission-Queue von Queue-Manager B.

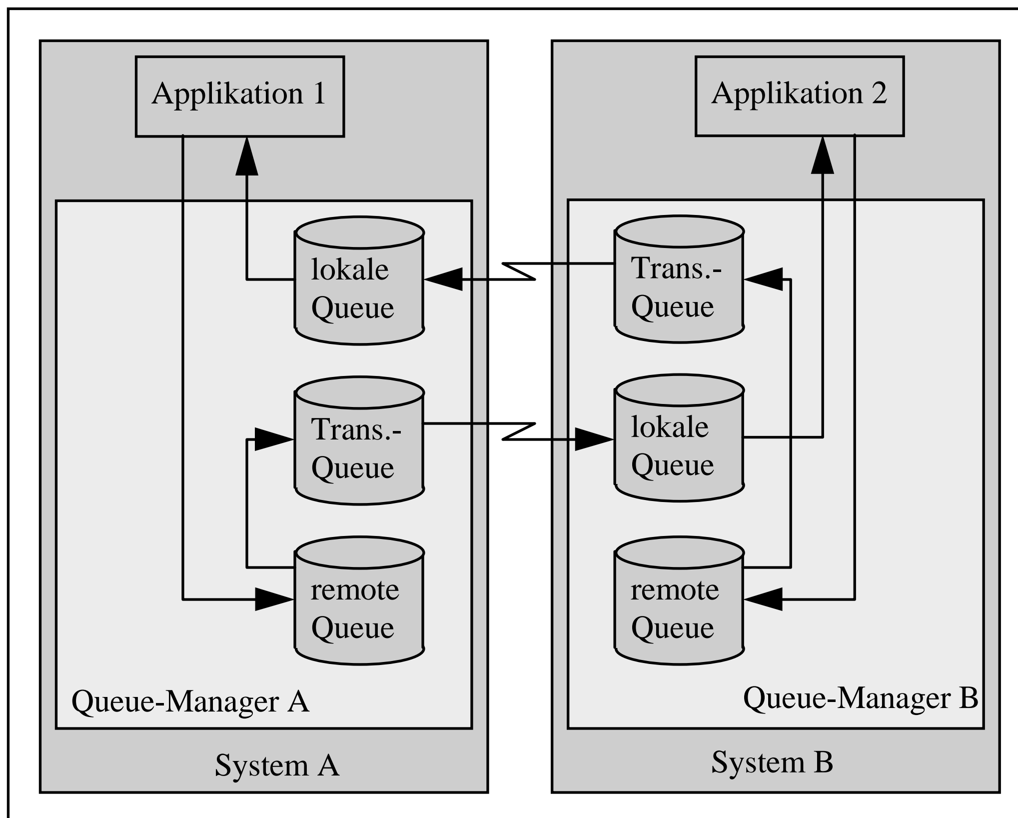


Abb. 20: Message Routing

Um zu verhindern, daß die Queue-Namen zuviel Netzwerkinformation beinhalten, benutzt MQSeries Routingtabellen, in denen die Namen und Aliasse der Queues den Netzwerknamen zugeordnet werden. Durch diese Technik können die Queues zur Laufzeit geändert werden, ohne daß der Programmcode geändert werden muß.

Damit die Queue-Manager miteinander unabhängig von ihrer Plattform kommunizieren können, benutzen sie das MCP. Es liefert die Messages an die Transportschicht-Protokolle (LU6.2, TCP/IP, NetBIOS). Durch das Zusammenspiel von MCP und dem Store-and-Forward-Prinzip wird gewährleistet, daß Messages auch bei Unverfügbarkeit des Netzwerks nicht verloren gehen, sondern daß die Messages nach Wiederherstellung der Verbindung erneut gesendet werden. Eine Kopie der Message bleibt solange in der Transmission-Queue, bis sie korrekt über einen Channel zum entfernten Queue-Manager übertragen wurde.

Wenn keine direkte Verbindung zwischen dem Quell-Queue-Manager und dem Ziel-Queue-Manager besteht, so kann die Message über weitere Queue-Manager geleitet werden. Dies wird als **Multihopping** bezeichnet. Dazu müssen Channels zwischen den Queue-Managern bestehen, und auf jedem Queue-Manager muß eine Transmission-Queue eingerichtet sein.

Die Definitionen für die Queues und Channels des Fehlerbehandlungssystems sind Kapitel 5.5 zu entnehmen.

5.4 MQSeries Client für Java

MQSeries Client für Java ist eine Klassenbibliothek, mit deren Hilfe Java-Applets MQSeries-Anwendungen benutzen können. MQSeries Client für Java ist ein MQSeries-Client, der in Java geschrieben wurde und über TCP/IP kommunizieren kann. Dadurch wird es möglich, daß Java-Applets über das Internet Messages versenden können. Damit erhalten die Applets die

Möglichkeit, Anwendungen zu benutzen, die auf Großrechnern und einer Vielzahl von Servern laufen.

5.4.1 Architektur

MQSeries Client for Java ist - wie der Name schon sagt - ein MQSeries-Client. Im Regelfall handelt es sich dabei um einen **Slim-Client**, das bedeutet, daß kein Queue-Manager und keine anderen MQ-Objekte (insbesondere auch keine Queues) auf dem Client-Rechner installiert sind. Daraus resultiert, daß jeder MQI-Call des Slim-Clients über einen MQI-Channel zum Queue-Manager übertragen werden muß. Kann keine Verbindung zum Queue-Manager aufgebaut werden, schlagen alle MQI-Calls fehl. Mehrere Slim-Clients können sich zu demselben Queue-Manager verbinden. Die Queues können damit von allen Clients aus benutzt werden.

Es besteht aber auch die Möglichkeit, MQSeries for Java als **Fat-Client** zu installieren. In diesem Fall befindet sich ein Queue-Manager auf dem Client-Rechner. Damit kann sich der Client eigene Queues einrichten. Dies ist vor allem dann sinnvoll, wenn es sich um einen mobilen Rechner handelt. Die MQ-Applikationen können somit trotz einer temporären Verbindungsunterbrechung ausgeführt werden. Für das Fehlerbehandlungssystem wurden sowohl die Fat-Client als auch die Slim-Client Variante gebraucht, da nicht vorausgesetzt werden kann, daß auf jedem Rechner ein Queue-Manager installiert ist.

Es gibt drei Möglichkeiten, wie MQSeries-Programme, die in Java geschrieben sind, ablaufen können:

Die erste Möglichkeit ist das Ausführen von MQ-Applets in einem **Java-fähigen Web-Browser**. Dabei muß beachtet werden, daß der Ort des Queue-Managers, auf den zugegriffen wird, den Sicherheitsbeschränkungen des Browsers unterliegt. Um dieses Problem zu vermeiden, sollte der Queue-Manager auf dem Server installiert sein, auf dem der Webserver läuft. Ist das der Fall, kann das Java-Applet auf die Queues des Queue-Managers zugreifen. Auf den Clients braucht kein MQSeries Code installiert zu werden, da dieser, wenn er benötigt wird, automatisch geladen wird, wenn das Applet ausgeführt wird.

Die zweite Möglichkeit zum Ausführen von MQ-Applets ist die Benutzung eines **AppletViewers**. Dazu ist es erforderlich, daß das Java Developer´s Kit (JDK) auf dem Client-Rechner installiert ist. Um eine URL anzusehen, muß der MQSeries Client für Java auf der Server-Maschine installiert sein. Es gelten die gleichen Restriktionen wie beim Einsatz eines Java-fähigen Browsers. Um lokale Dateien zu sehen, muß der MQSeries Client für Java auf dem Client-Rechner zusammen mit einem Java oder AppletViewer Programm installiert sein. In diesem Fall kann der Queue-Manager, zu dem sich die Clients verbinden, auf einem beliebigen über TCP/IP adressierbaren Rechner installiert sein.

Die dritte Möglichkeit MQSeries-Programme in Java ablaufen zu lassen ist, ein **Stand-Alone Java Programm** auszuführen. In diesem Fall muß das JDK auf der Client-Maschine installiert sein.

5.4.2 Die Java-Klassenbibliothek

Die Java-Klassen befinden sich im Package *com.ibm.mq*. Wird dieses Package importiert, stehen dem Programmierer die Klassen zur Verfügung.

Exemplare der Klasse *MQMessage* repräsentieren die Messages. *MQMessage* enthält als Attribute sowohl die einzelnen Teile des Messageheader (siehe 5.2.1) als auch den Message-Buffer. Diese Klasse implementiert die Java-Interfaces *DataInput* und *DataOutput*.

MQManagedObject ist die Oberklasse von *MQQueue*, *MQQueueManager* und *MQProcess*. Ihre wesentlichen Methoden sind set und inquire, um diverse Attribute zu lesen und zu ändern. *MQQueue* repräsentiert die Queues mit ihren Attributen. Da die Java-Klassenbibliothek objektorientiert ist und die Queues, wie auch alle anderen Komponenten, als Objekte vorhanden sind, ist das Object-handle für den Programmierer transparent. Die Methoden dieser Klasse können damit ohne Object-handle aufgerufen werden. Außerdem besteht die Möglichkeit, direkt über set- und get-Methoden die Attribute der Queue (siehe 5.2.3) zu ändern. Die Methoden set und inquire von *MQManagedObjekt* brauchen damit nicht benutzt zu werden. Sie dienen nur der Einheitlichkeit der Implementierung des MQIs.

Die Klasse *MQQueueManager* bildet die Schnittstelle zum Queue-Manager. Ebenso wie das Object-handle ist auch das Connection-handle für den Programmierer transparent. Über Methoden dieser Klasse kann auf Queues und Prozesse zugegriffen werden.

Exemplare der Klassen *MQGetMessageOptions* und *MQPutMessageOptions* werden als Parameter beim Aufruf von get bzw. put einer Message-Queue übergeben. Sie steuern die Aktivitäten der jeweiligen Operation. Für get-Calls kann festgelegt werden, ob die Message gelesen oder gelöscht werden soll, ob und wenn wie lange auf die Message gewartet werden soll, ob ein Synchronisationspunkt gesetzt werden soll, ob die Message gelockt werden soll und ähnliches. Für put-Calls können ebenfalls Synchronisationspunkte gesetzt werden.

Die Klasse *MQChannelDefinition* enthält Information über den Queue-Manager, die zu sogenannten Exits gesendet werden. Dieses sind Code-Fragmente, die an speziellen Stellen ausgeführt werden. Wird beispielsweise eine Message in die Transmission-Queue gestellt, wird der SecurityExit aufgerufen. Der Programmierer kann an dieser Stelle Sicherheitskontrollen durchführen. Dazu müssen die Interfaces *MQReceiveExit*, *MQSecurityExit* und/oder *MQSendExit* implementiert und der Umgebung (*MQEnvironment*) bekannt machen. Die Klasse *MQChannelExit* enthält Kontextinformationen für die Exits.

Weitere wichtige Klassen sind *MQEnvironment* (enthält Channel, Hostname und Portnummer des Queue-Managers, Paßwort, Exits, etc.) und *MQException*, die die Basis für alle MQSeries-Ausnahmen ist. Im MQC-Interface sind die Konstanten für die MQ-Klassen definiert.

5.5 Die Architektur des Fehlerbehandlungssystems

In diesem Kapitel werden die MQSeries-Komponenten vorgestellt, die für das Fehlerbehandlungssystem benötigt werden, und das Zusammenspiel dieser Komponenten erläutert.

Die zentrale Komponente der Architektur des Fehlerbehandlungssystems ist ein Server, auf dem sich der Queue-Manager SERVER mit der Queue FEHLER.LOCAL (im folgenden Fehler-Queue genannt) befindet. Dieses ist die zentrale Queue, in die die Fehler gespeichert werden. Sie kann sowohl von Fat-Clients als auch von Slim-Clients benutzt werden. Abb. 21 zeigt die Fat-Client Variante. Zur Laufzeit können sich mehrere Clients an die Queue-Manager CLIENT binden; aus Übersichtlichkeitsgründen ist in Abb. 21 jedoch nur ein Client dargestellt.

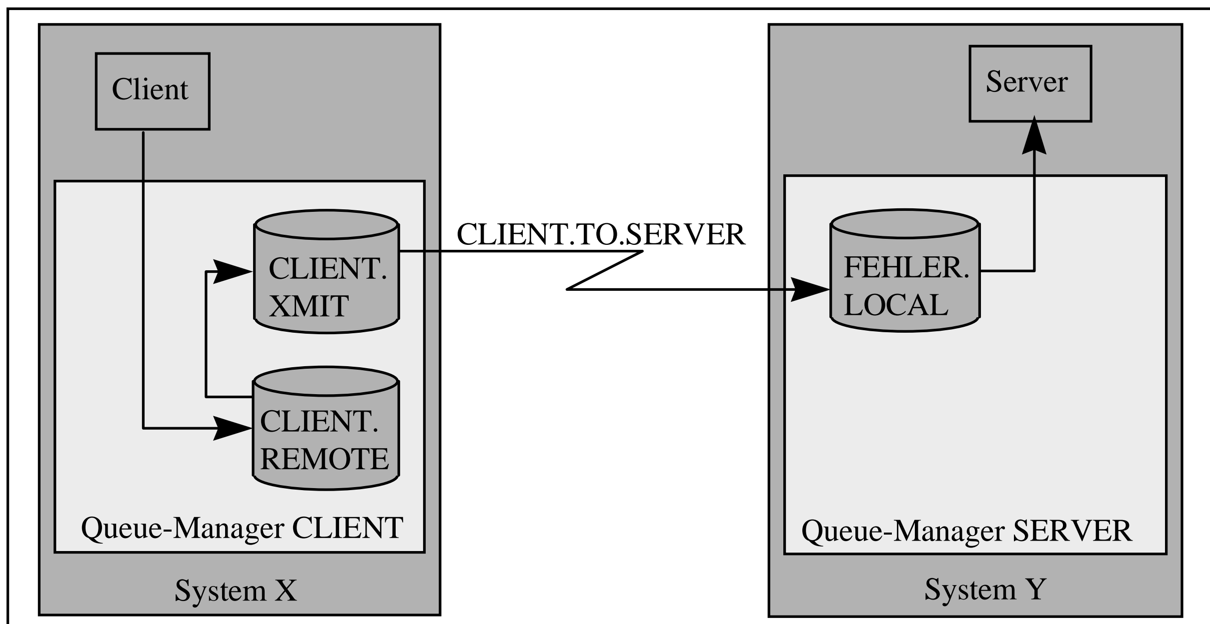


Abb. 21: MQ-Komponenten der Fat-Client Variante des Fehlerbehandlungssystems

Für den Client ist die Fehler-Queue entfernt, da sie dem Queue-Manager SERVER gehört. Aus diesem Grund braucht CLIENT eine lokale Definition dieser Queue (CLIENT.REMOTE), eine Transmission-Queue (CLIENT.XMIT), um die Message zum SERVER zu senden, sowie einen Message-Channel zum SERVER (CLIENT.TO.SERVER). Wenn beim Client ein Fehler aufgetreten ist, der in der Fehler-Queue gespeichert werden soll, so muß sich der Client zunächst an den Queue-Manager CLIENT binden. Nachdem das erfolgt ist, kann er den Fehler in eine Fehler-Message verpacken. Wenn der Client eine Message zu Fehler.LOCAL sendet, nimmt CLIENT die Message und stellt sie in die Transmission-Queue. Der MCA, der auf CLIENT läuft, transferiert die Message zum MCA, der auf dem SERVER läuft. Dieser transportiert die Message vom Channel in die Queue FEHLER.LOCAL. Der MQSeries-Code für diese Komponenten ist dem Anhang zu entnehmen.

Bei der Slim-Client Variante kann sich ein Client direkt über einen MQI-Channel an den Queue-Manager SERVER binden und die Fehlermessages in die Fehler-Queue speichern. Diese Variante kann aber nur angewendet werden, wenn der Rechner, auf dem der Client läuft, eine Netzwerkverbindung zum Server besitzt und scheidet daher für die Laptops der Versicherungsvertreter aus.

Dieses Kapitel hat die message-orientierte Middleware am Beispiel von MQSeries vorgestellt und bildet die Basis für den Verteilungsaspekt des Fehlerbehandlungssystems, auf dessen Implementierung in Kapitel 7.2 eingegangen wird. Zunächst wird aber im nächsten Kapitel eine alternative Middleware (CORBA) diskutiert.

6 CORBA

In der Fachliteratur erscheinen seit einigen Jahren immer mehr Artikel über CORBA. Die Begeisterung der Experten für diese neue Middleware für Objektsysteme scheint unaufhaltsam, vor allem seitdem die Kombination CORBA-Java für Furore sorgt. Java bietet portable Objekte an, die auf jedem größeren Betriebssystem funktionsfähig sind; CORBA als Middleware für verteilte Objekte sorgt für die Netzwerk- und Programmiersprachen-Transparenz. Verbindet man CORBA und Java, so können portable Client-Server-Applikationen entwickelt werden, die weltweit über das Internet verteilt sein können. Diese Verbindung von verteilten Objekten und Internet wird als Object Web bezeichnet. Dieses Kapitel erläutert die Funktionsweise von CORBA, um die Grundlage dafür zu schaffen, das verteilte Fehlerbehandlungssystem mit CORBA, als einer alternativen Middleware zu MOM, zu implementieren.

CORBA (Common Object Request Broker Architecture) ist das größte Middleware-Projekt, das jemals von der Industrie getätigt wurde. CORBA wurde 1989 von der OMG (Object Management Group), dem weltweit größten Softwarekonsortium mit über 700 Unternehmen als Mitglieder (u.a. IBM, Sun, Oracle, Microsoft) spezifiziert.

Es handelt sich bei CORBA um eine offene Architektur für verteilte Objekte, die das Ende der systemnahen Netzwerkprogrammierung einschließlich des damit verbundenen Wartungstraumas erreicht hat. Das C von CORBA steht dabei für zwei APIs, die statische und dynamische Aufrufe zur Verfügung stellen.

Die Kommunikation von Objekten über CORBA kann durch vier Adjektive charakterisiert werden: CORBA-Kommunikation ist **verteilungstransparent**, d.h. das Client-Objekt kann vollständig von dem Aufenthaltsort des Serverobjekts abstrahieren. Die Kommunikation mittels CORBA ist **programmiersprachenunabhängig**. Ein Objekt, das beispielsweise in Java implementiert ist, kann mit einem Objekt in C++, Smalltalk oder einer anderen Programmiersprache, die das CORBA Sprach-Binding unterstützt, kommunizieren. Dies funktioniert über die Spezifikationssprache IDL (Interface Definition Language, Kapitel 6.1). Alle Schnittstellen der Serverobjekte werden in IDL spezifiziert und somit von der Implementationssprache getrennt. Weiterhin ist CORBA-Kommunikation, da CORBA eine Middleware ist, **betriebsystem-** und **hardwareunabhängig**.

Die von der OMG entwickelte Spezifikation kann von beliebigen Herstellern implementiert werden. Die derzeit bekanntesten Implementationen sind Visibroker von Inprise, Orbix von IONA, Joe von SUN, NEO von SunSoft, DSOM von IBM, HP-ORB von Hewlett-Packard, ObjectBroker von Digital (Weiterentwicklung bei BEA) und Distributed-Smalltalk (DST) von ParcPlace. Damit durch die Vielfalt der CORBA-Produkte keine proprietären Insellösungen entstehen, hat die OMG in der CORBA Version 2.0 vom Dezember 1994 der Spezifikation ein Protokoll hinzugefügt, das die Kommunikation zwischen verschiedenen CORBA-Produkten standardisiert[30]. Dieses Protokoll ist das GIOP (General Inter-ORB Protokoll). Die Umsetzung, wie GIOP-Messages über ein TCP/IP-Netz transportiert werden, wird IIOP (Internet Inter-ORB Protokoll) genannt. Mit diesem Protokoll können Objekte, die mit unterschiedlichen CORBA-Produkten implementiert sind, miteinander kommunizieren. Die großen Unternehmen Sun, IBM, Netscape, Apple, Oracle, BEA und HP haben beschlossen,

[30] Die erste CORBA Version enthielt nur die Spezifikation für IDL, das Sprach-Mapping und die API des Objekt Request Brokers.

CORBA/IIOP als gemeinsamen Weg zu benutzen, um verteilte Objekte über das Internet und Intranet zu verbinden.

Die aktuelle Version ist 2.2 vom Februar 1998. In Kapitel 6.2 wird die vollständige Architektur von CORBA dargestellt.

CORBA ist jedoch mehr als nur ein Objekt Request Broker (ORB). Zusätzlich zu dem Verteilungsmechanismus hat die OMG auch noch Komponenten definiert, die den ORB schichtenartig um wichtige Funktionalität erweitern. Die Diskussion dieser Komponenten ist Bestandteil von Kapitel 6.3. Die CORBA-Implementation, die dieser Diplomarbeit zu Grunde liegt, ist der bereits erwähnte Visibroker von Inprise. Er wird in Kapitel 6.4 vorgestellt. Zum Schluß dieses Kapitels wird die Integration von CORBA ins JWAM-Rahmenwerk erläutert, weil die CORBA-Implementation des Fehlerbehandlungssystem dieses Rahmenwerk benutzt.

6.1 IDL

Der für Anwendungsprogrammierer wichtigste Teil von CORBA ist die Interface Definition Language (IDL). Es handelt sich hierbei um eine rein deklarative, stark getypte, plattformunabhängige Definitionssprache. IDL besitzt ausschließlich Elemente zur Beschreibung von Daten, also keinerlei Anweisungskonstrukte oder Implementierungsdetails. Um die Erlernbarkeit dieser Spezifikationssprache für Programmierer möglichst einfach zu gestalten, hat die OMG IDL an C++ angelehnt. Nur für den Verteilungsaspekt sind zusätzliche Schlüsselwörter eingeführt worden.

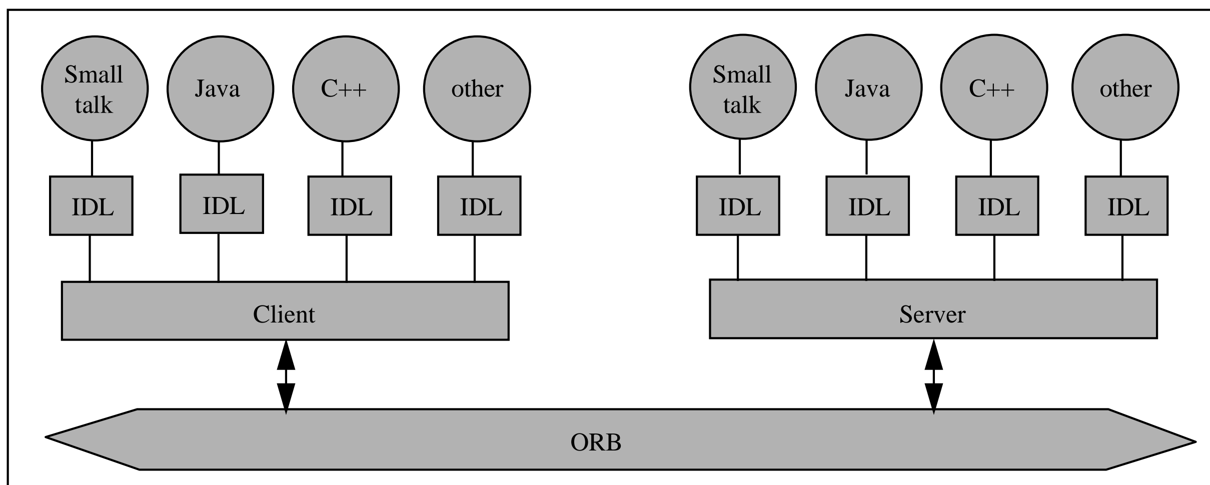


Abb. 22: IDL Sprach-Binding[31]

Ziel der OMG ist es, die gesamte Client/Server-Middleware einschließlich aller Komponenten, die den ORB benutzen, mit IDL zu versehen, um die Programmiersprachen-Unabhängigkeit zu gewährleisten. Alle Programmiersprachen, die das CORBA-Binding unterstützen (dies sind zur Zeit Smalltalk, C++, C, Java, COBOL und Ada), können die in IDL spezifizierten Interfaces benutzen. Dabei bleibt die Implementationssprache des aufgerufenen Dienstes für den Aufrufer transparent (siehe Abb. 22). Auf diese Weise kann auf Legacy-Systeme, die auf einem Großrechner in COBOL implementiert sind, von objektorientierten Programmiersprachen aus zugegriffen werden. Die Host-Programme brauchen nur IDL-Schnittstellen zu erhalten und anschließend können diese Programme vom PC oder dem Internet aus benutzt werden. Dadurch hat der Host als ausfallsicherer Server in Großunternehmen weiterhin eine gute Zukunftsperspektive.

---

[31] aus [ORFA97a], S.5

IDL erfüllt aber noch einen ganz anderen Zweck. Da alle Komponenten, die CORBA-kompatibel sein wollen, ein IDL-Interface besitzen müssen, ist CORBA ein selbstbeschreibendes System mit IDL als Metadatensprache. Die IDL-Interfaces sind die Metadaten, die eine Voraussetzung für die Entwicklung von flexiblen, dynamischen und hoch-konfigurierbaren Client/Server-Systemen darstellen. Durch diese Metadaten sind CORBA-Komponenten in der Lage, sich gegenseitig zur Laufzeit zu finden und zu benutzen. Ohne Metadaten wäre CORBA nur eine weitere festkodierte, starre und unflexible Middleware.

```
module <identifier>
{
  <type declarations>;
  <constant declarations>;
  <exception declaration>;

  interface <identifier> [:<inheritance>]
  {
      <type declarations>;
      <constant declarations>;
      <attribute declarations>;
      <exception declaration>;

      [<op_type>] <identifier> (<parameters>)
           [raises exception] [context];
                 :
      [<op_type>] <identifier> (<parameters>)
           [raises exception] [context];
                 :
  }

  interface <identifier> [:<inheritance>]
                 :
}
```

Abb. 23: Syntax eines IDL-Moduls[32]

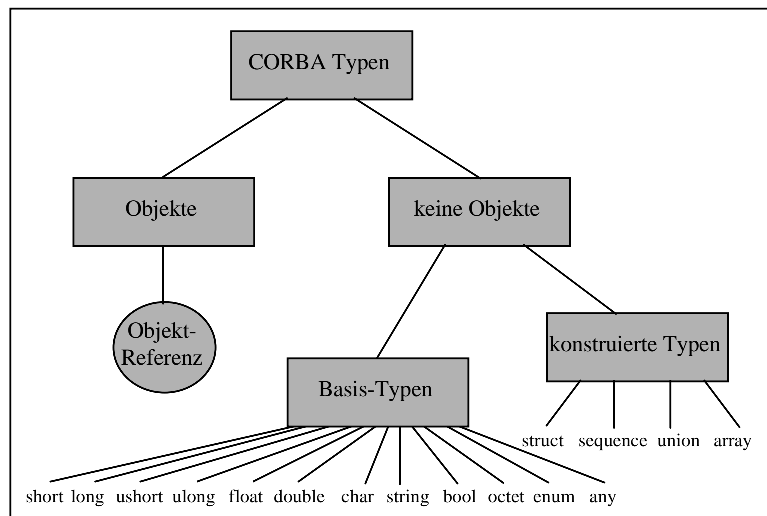


Abb. 24: Datentypen von IDL[33]

In IDL werden die Interfaces von den Objekten spezifiziert, auf die entfernt zugegriffen werden soll. Ein Interface enthält Typ-Deklarationen, Konstanten, Attribute, Ausnahme-Deklarationen und Signaturen der Operationen (siehe Abb. 23). Die Signatur besteht aus Rückgabetyp, Operationsnamen, einer Parameterliste, einer Ausnahmeliste mit Ausnahmen, die von dieser Operation ausgelöst werden können, und einen Kontext, der eine Menge von Client-Attributen enthält, um bei Bedarf dem Server die Client-Umgebung mitteilen zu können. Ein einzelner Parameter besteht aus dem Tripel Modus, Typ und Name. Der Modus zeigt an, ob der Parameter einer Operation vom Client zum Server (in), vom Server zum Client (out) oder in beide Richtungen (inout) gereicht wird. Zusätzlich können für jede Operation Pragma-Direktiven codiert werden. Diese dienen beispielsweise der Generierung von global eindeutigen Identifiern für die Interfaces oder bei Programmiersprachen wie Smalltalk dem Umdefinieren von Methodennamen, wenn die spezielle Syntax einer Programmiersprache dies erfordert. Hinter dem Interfacenamen werden die Parent-Interfaces angegeben. Da IDL Mehrfachvererbung unterstützt, können dies mehrere sein.

[32] aus [ORFA97a], S. 88
[33] aus [ORFA97a], S. 89

Die benutzbaren IDL-Typen können entweder IDL-Interfaces oder einfache Datentypen sein. Die einfachen Datentypen teilen sich in Basistypen und konstruierte Typen auf. Die Basistypen sind short, long, ushort, ulong, float, double, char, string, bool, octet, enum und any (repräsentiert einen beliebigen IDL-Typ oder eine Objektreferenz), die konstruierten Typen struct, sequence, union und arrays. Diese Datentypen sind aus C und C++ ebenso bekannt wie der Rückgabetyp void. Alle IDL-Typen können sprach-, betriebsystem- und ORB-übergreifend benutzt werden (siehe Abb. 24).

Damit ein IDL-Interface in einer Implementationssprache implementiert werden kann, muß das IDL-File zuerst von einem IDL-Compiler übersetzt werden. Der Compiler generiert aus dem IDL-File den Client Stub (siehe 6.2.2.2) und das Static IDL Skeleton (siehe 6.2.2.5). Abhängig von der Zielsprache werden noch Headerfiles (C++) oder Interfaces (Java) generiert, die in den Code eingebunden werden müssen. Für die Attribute der Interfaces werden get- und set-Methoden definiert (Ausnahme: Bei read-only Attributen wird nur die get-Methode generiert).

Weil auf die Fehler des Fehlerbehandlungssystems verteilt zugegriffen wird, benötigen sie ein IDL-Interface. Da die Klasse Fehler (siehe Kapitel 7.1) nur eine Methode besitzt, die verteilt benutzt wird, braucht das IDL-Interface entsprechend nur die eine Methode *toString()* zu enthalten (siehe Abb. 25).

```
module Fehlerbehandlungssystem
{
    interface Fehler
    {
        string toString();
    }
}
```

Abb. 25: IDL-Modul Fehlerbehandlungssystem

6.2 Die Struktur von CORBA

Der zentrale Teil der CORBA-Struktur ist der Object Request Broker (Kapitel 0). Über den ORB können die Clients auf zweierlei Arten auf die Serverobjekte zugreifen. In Kapitel 6.2.2 wird der statische Aufruf behandelt; in Kapitel 6.2.3 folgt der dynamische Aufruf.

6.2.1 Object Request Broker

Der Object Request Broker (ORB) ist ein Objektbus, der als Nachrichtenvermittler (Broker) fungiert. Er transportiert die Requests der Clients zu den jeweiligen Serverobjekten und die Rückgabewerte der Requests (dies können auch Ausnahmen sein) von den Serverobjekten zu den Clients. Der Zugriff über den ORB auf das Serverobjekt ist für den Client transparent. Der Client braucht sich dabei weder um den Aufenthaltsort des Serverobjekts noch um die Programmiersprache, in der es implementiert ist, noch um das Betriebssystem, auf dem es sich befindet, zu kümmern. Der ORB bildet somit eine Abstraktionsschicht zwischen Plattform und Anwendung (siehe Abb. 26).

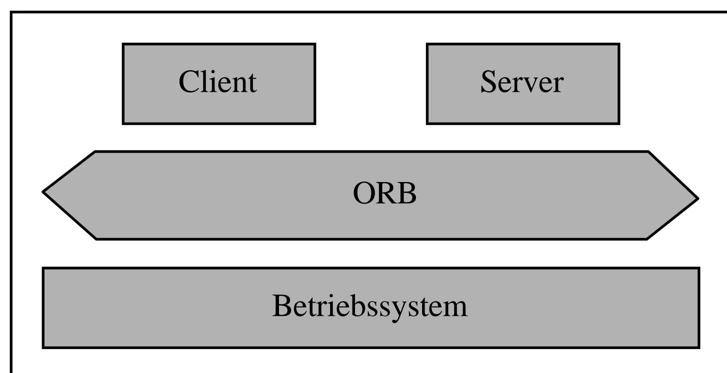


Abb. 26: Der ORB als Middleware

Der ORB bietet eine Menge von verteilten Middleware Services (Object Services, siehe Kapitel 6.3.1) an. Durch diese können sich Objekte zur Laufzeit entdecken und ihre Services nutzen.

Gemäß der CORBA-Spezifikation der OMG muß der ORB nicht als eine Komponente implementiert werden. Entscheidend ist nur, daß das vorgeschriebene ORB-Interface unterstützt wird. Dieses Interface ist unabhängig von den Objektinterfaces und dem Objektadaptor. Es enthält nur wenige Operationen, da die meisten Operationen in den Stubs, Skeletons und dem Objektadaptor spezifiziert sind. Die wichtigsten Methoden des ORB-Interfaces sind *string_to_object* und *object_to_string*. Diese Methoden wandeln eine Objektreferenz in einen String, der persistent gemacht werden kann. Wird dieser String einige Zeit später in eine Objektreferenz zurückgewandelt, so wird das ursprüngliche Objekt referenziert. Das ORB-Interface kann sowohl von den Clients als auch von den Serverobjekten benutzt werden.

Die verschiedenen Hersteller von ORBs haben die Möglichkeit, unterschiedliche Implementierungen für ihren ORB zu wählen. In diesem Punkt unterscheidet sich die Qualität und die Eigenschaften der Implementationen erheblich. Auch die Repräsentationen der Objektreferenzen sind herstellerabhängig (siehe Kapitel 6.2.2.6).

Aus Skalierbarkeitsgründen können sich mehrere Broker in einem LAN befinden. Diese können über das Internet Inter-ORB Protokoll (IIOP) miteinander kommunizieren.

6.2.2 Komponenten des statischen CORBAs

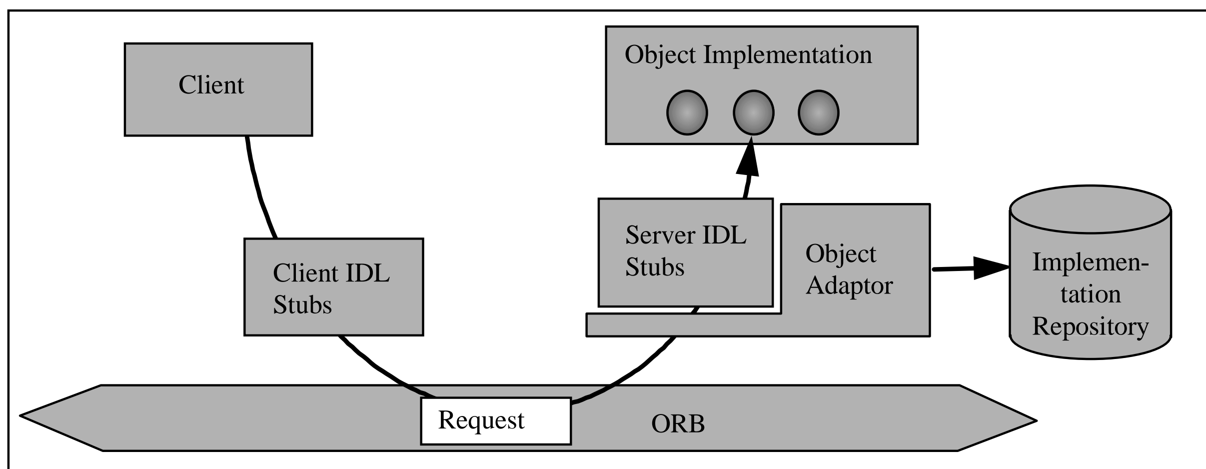


Abb. 27: Statische CORBA-Komponenten

Abb. 27 zeigt die Komponenten, die beim statischen Aufruf benötigt werden. Der Client schickt einen Anfrage über den Client IDL Stub und den Object Adaptor zum Server IDL Stub. Dieser leitet den Request zum Server weiter.

6.2.2.1 Client

Ein Client ist ein Objekt, das Operationen bei einem Serverobjekt aufruft. Daraus folgt, daß ein Objekt nur relativ zu einem anderen Objekt ein Client sein kann. Weil Objekte gleichzeitig in verschiedenen Relationen stehen können, kann ein und dasselbe Objekt zur gleichen Zeit Client und Server sein.

Ein Client kann sich beispielsweise über einen Namen eine Objektreferenz besorgen (Naming Service) und auf ihr die in IDL spezifizierten Operationen aufrufen. Alle Methoden, die nicht in IDL spezifiziert sind, können von einem entfernten Client nicht aufgerufen werden, da dieser nur das IDL-Interface des Serverobjekts kennt und nicht die Objektimplementation des Servers. Dies und die Tatsache, daß dem Client der Aufenthaltsort des Servers nicht bekannt

ist, machen den Client sehr portabel. Er ist ohne Code-Anpassung auf allen ORBs lauffähig, die das Sprach-Mapping von IDL in der Programmiersprache des Clients anbieten.

CORBA bietet überwiegend die synchrone Request/Response-Semantik an. Der Client ist nach einem Aufruf solange blockiert, bis das Serverobjekt die Operation vollständig verarbeitet hat. Zusätzlich gibt es die Möglichkeit, One-Way-Requests zu benutzen. Der Client erhält dabei sofort nach Methodenaufruf die Kontrolle zurück. Dies ist allerdings nur bei Methodenaufrufen ohne Rückgabeparameter möglich. Ein Nachteil der One-Way-Requests ist, daß es keine Garantie für eine erfolgreiche Ausführung gibt. Als weitere Möglichkeit bietet CORBA zurückgestellte synchrone Aufrufe mittels DII (Dynamic Invocation Interface, siehe Kapitel 6.2.3.2) an. Dabei handelt es sich um eine asynchrone Ausführung, deren Beendigung mit periodischem Polling abgeprüft werden kann.

6.2.2.2 Client IDL Stubs

Client IDL Stubs sind der CORBA-Kontext für Proxies[34]. Sie agieren als Stellvertreter für das Serverobjekt (Server-Proxy) und implementieren daher dieselbe Schnittstelle wie der Server. Der Client ruft lokal beim Client IDL Stub die Methoden des Servers auf. Der Stub übernimmt das Marshalling, d.h. er übersetzt den Aufruf in einen Bytecode, der über das Netz übertragen wird und auf der anderen Seite vom Server IDL Stub zurück in den Aufruf gewandelt wird. Wenn die Methoden einen Wert als Ergebnis zurückgeben, so empfängt der Client IDL Stub diesen vom Server IDL Stub und führt mit ihm das Unmarshalling durch. Anschließend wird der Wert dem Client übergeben.

Bei statischen Methoden-Aufrufen braucht ein Client für jedes Serverobjekt, das er entfernt aufrufen möchte, einen Stub. Der Stub wird vom IDL-Compiler generiert, wenn das IDL-Interface des Servers compiliert wird. Es muß darauf geachtet werden, daß der Stub in der Sprache generiert wird, in der der Client implementiert werden soll. Andernfalls könnte der Client den Stub nicht aufrufen.

Java Client Stubs sind unabhängig von einem speziellen CORBA-Produkt, da die OMG im CORBA-Java-Binding ein portables Stub-Format spezifiziert hat. In Java ist es damit möglich, einen von Orbix generierten Stub für statische Aufrufe des Clients mittels Visibroker zu nutzen. Da der Stub-Code nichts anderes als ein Byte-Code ist, kann dieser zusammen mit dem Client-Code geladen werden.

6.2.2.3 Object Adaptor

Bevor Clients Methoden von Serverobjekten aufrufen können, müssen sich die Serverobjekte, die ihre Dienste zur Verfügung stellen wollen, beim Object Adaptor anmelden. Für jedes angemeldete Objekt wird ein Eintrag im Implementation Repository erzeugt, der unter anderem die physikalische Adresse des Servers beinhaltet.

Wenn ein Request eines Clients beim Object Adaptor eintrifft, prüft dieser, ob das angefragte Objekt bereits aktiv ist oder ob es noch aktiviert werden muß. Im letzteren Fall wird das Objekt mittels einer Aktivierungsstrategie aktiviert: Es besteht die Möglichkeit, das Serverobjekt in einem neuen Prozeß oder einem neuen Thread im alten Prozeß zu aktivieren. Es kann aber auch ein alter Thread oder Prozeß für die Aktivierung wiederverwendet werden. Nachdem das Objekt aktiviert worden ist, wird der Request samt Parametern an das Objekt weitergeleitet.

---

[34] [GAMM95]

Weitere Services des Object Adaptors sind Generierung und Interpretation von Objektreferenzen, Sicherheit von Interaktionen (Authentisierung bei Methodenaufrufen) und Deaktivierung von Objekten.

Weil es eine große Vielzahl von Lebenszyklen, Implementierungs-Stilen und anderen Eigenschaften von Objekten gibt, ist es für den ORB Core schwer, ein einfaches Interface anzubieten, welches allen Objekten gerecht wird. Über verschiedene Object Adaptoren ist es möglich, Zielgruppen von Objekten mit ähnlichen Anforderungen passende Interfaces anzubieten.

Der einfachste und zu allen CORBA-Implementationen dazugehörende Object Adaptor ist der BOA (Basic Object Adaptor). Er gibt Clients den Schein, als wenn alle Objekte, die bei ihm angemeldet sind, aktiviert seien. Tatsächlich können nicht alle Objekte zur Startzeit des ORBs aktiviert werden, da bei einer großen Anzahl von registrierten Objekten die Ressourcen größter Rechner nicht ausreichen würden. Also wird eine automatische Start-Funktionalität oder ein Starten on-demand benötigt. Der BOA unterstützt imperative und objektorientierte Sprachen.

6.2.2.4 Implementation Repository

CORBA benötigt eine Datenbank auf die der Objekt Adaptor zugreifen kann, wenn ein Client ein Serverobjekt lokalisieren oder aktivieren möchte. In der CORBA-Terminologie wird diese Datenbank als Implementation Repository bezeichnet. Sie enthält Informationen darüber, welche Klassen von welchem Server angeboten werden, welche Objekte exemplarisiert sind und welche IDs diese enthalten.

Ein neuer Server meldet sich über den BOA beim CORBA-System an und erhält einen Eintrag im Implementation Repository. Registriert werden alle vom Server administrierten Serverobjekte mit Namen und physikalischer Position (Verzeichnispfad des ausführbaren Server-Programms). Empfängt der BOA einen Aufruf, wird das Implementation Repository nach dem Zielobjekt durchsucht und ggf. aktiviert.

Zusätzlich können im Implementation Repository Informationen, die mit ORB-Objekten zusammenhängen, gespeichert werden (z.B. Debug-Informationen, Sicherheitskontrollen).

6.2.2.5 Server IDL Stub

Alle Objekte, die entfernt aufgerufen werden sollen, benötigen ein vom IDL-Compiler generiertes Server IDL Stub (oder auch Skeleton genannt). Das Skeleton hat dieselbe Aufgabe wie der Client IDL Stub, nämlich die Durchführung des Marshallings und Unmarshallings, nur daß das Skeleton für die Serverseite tätig ist.

Der Server IDL Stub tritt dem Serverobjekt als Client entgegen und leitet die Methodenaufrufe entsprechend weiter. Das Skeleton befindet sich im Serverprozeß, so daß es sich dabei um lokale Aufrufe handelt.

Die Existenz des Skeletons ist unabhängig von der Existenz der Client Stubs.

6.2.2.6 Serverobjekt

Ein Serverobjekt ist ein beliebiges Objekt, welches ein IDL-Interface implementiert und somit über CORBA verteilt aufgerufen werden kann. Dieses Objekt ist in einer Programmiersprache implementiert, die das Sprach-Binding unterstützt. Es ist vollkommen unabhängig vom ORB, den Clients, dem Object Adaptor und dem Betriebsystem. Das Serverobjekt weiß nicht, ob seine Methoden statisch oder dynamisch aufgerufen werden.

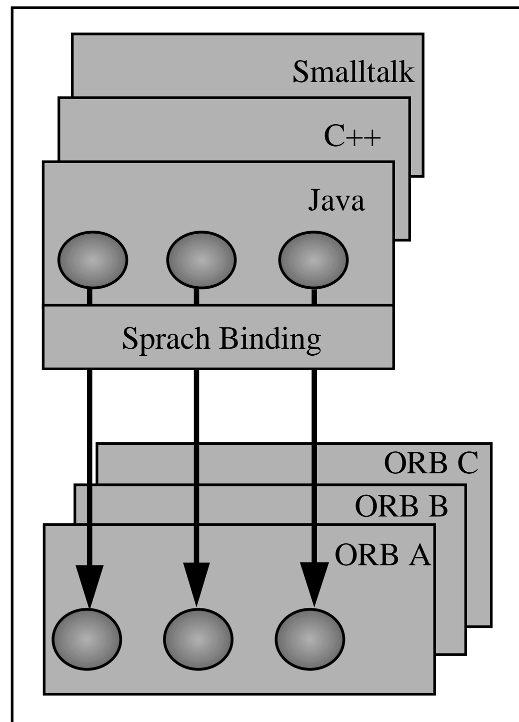


Abb. 28: Sprachunabhängige Objektreferenzen[35]

Jedes Serverobjekt besitzt eine Objektreferenz. Eine Objektreferenz ist ein eindeutiger Identifier, der benötigt wird, um ein Objekt innerhalb eines ORBs zu referenzieren. Die Implementation von Objektreferenzen ist nicht von CORBA spezifiziert und daher herstellerabhängig. Um es dennoch zu ermöglichen, daß Objektreferenzen zwischen ORBs verschiedener Hersteller ausgetauscht werden können, hat die OMG die interoperablen Objektreferenzen (IOR) spezifiziert. Die Repräsentationen der einzelnen Objektreferenzen der Programmiersprachen werden über das Sprach-Binding isoliert (siehe Abb. 28). Alle ORBs müssen dasselbe Sprach-Binding für Objektreferenzen einer Programmiersprache benutzen. IORs können über das ORB-Interface in Strings und von Strings wieder in IORs gemappt werden.

6.2.3 Komponenten des dynamischen CORBAs

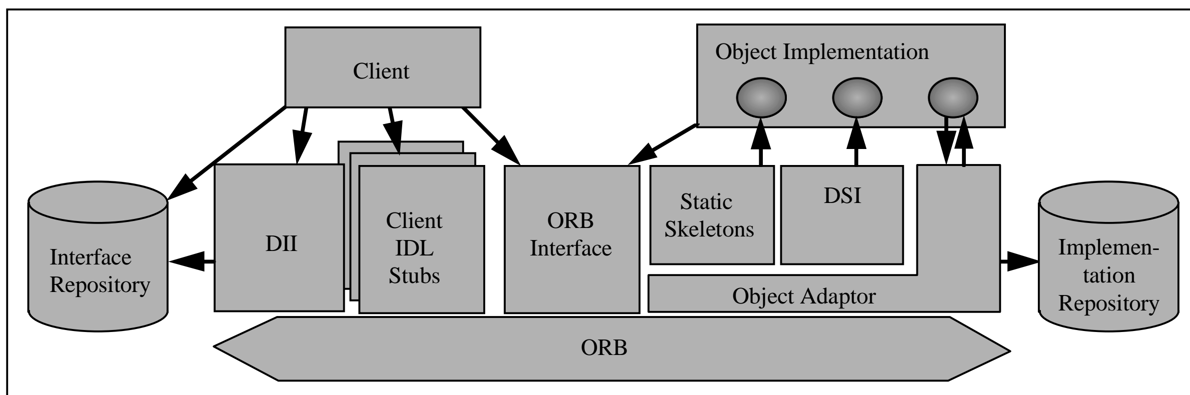


Abb. 29: CORBA-Komponenten[36]

Außer der Möglichkeit statisch auf Objekte zuzugreifen, bietet CORBA den dynamischen Methodenaufruf an. Durch diese Aufrufmöglichkeit wird CORBA zu einem sehr flexiblen System, da Objekte nicht zur Compilezeit bekannt sein müssen, sondern dynamisch zur Laufzeit entdeckt und deren Dienste benutzt werden können. Abb. 29 stellt die statischen und dynamischen CORBA-Komponenten dar.

---

[35] aus [ORFA97a], S.56

[36] aus [ORFA97a], S.11

6.2.3.1 Interface Repository

Das Interface Repository ist ein Repository, in dem die IDL-Interfaces in einer maschinenlesbaren Form gespeichert werden. Es handelt sich hierbei um eine dynamische Metadatenbank. Objekte können zur Laufzeit auf diese Datenbank zugreifen, um sich Informationen über die Interfaces geben zu lassen.

Das API des Interface Repositories stellt Methoden zur Abfrage und Manipulation von Metadaten zur Verfügung. Clients benötigen die Informationen der IDL-Interfaces, um den Dienst von Objekten in Anspruch nehmen zu können, deren Schnittstellen zur Compilezeit des Clients nicht bekannt waren. Diese Informationen sind außerdem für Tools wie Klassenbrowser und Compiler notwendig, um Vererbungsstrukturen und Klassendefinitionen zu erhalten. Zusätzlich bietet das Interface Repository ein Type-Checking der Methoden-Signaturen an.

Interface Repositories können lokal, als Abteilungs- oder als Unternehmensressource eingesetzt werden. Einzelne ORBs haben die Möglichkeit, auf mehrere Interface Repositories zuzugreifen.

Jedes Interface, das im Interface Repository gespeichert werden soll, kann eine eindeutige Repository ID erhalten, um eindeutig identifizierbar zu sein. Diese Identität muß auch bei der Benutzung von mehreren ORBs unterschiedlicher Hersteller gewährleistet sein, damit keine Namenskonflikte auftreten können. Die IDs ermöglichen eine Replikation von IDL-Interfaces. Die IDs werden vom ORB vergeben und können über ein Pragma (siehe 6.1) einem IDL-Interface zugewiesen werden.

6.2.3.2 Dynamic Invocation Interface

Mit Hilfe des Dynamic Invocation Interfaces (DII) können Clients zur Laufzeit Requests erzeugen, ohne dabei auf IDL-Stubs zugreifen zu müssen. Nachdem ein Client eine Referenz auf ein Objekt über einen String, den Naming Service oder den Trader Service (siehe 6.3.1) erhalten hat, kann er dieses Objekt nach seinem Interface fragen. Dieses Interface befindet sich im Interface Repository. Es enthält alle Methodensignaturen, die in IDL definiert sind. Unter Zuhilfenahme dieser Typinformationen ist es dem Client möglich, einen Request mit der aufzurufenden Operation, den benötigten Parametern und dem entsprechenden Rückgabetyp zu generieren. Die Voraussetzungen für diese flexiblen Aufrufmöglichkeiten sind, daß CORBA ein selbstbeschreibendes System ist und daß alle entfernt aufrufbaren Objekte ein IDL-Interface besitzen.

Der Vorteil des dynamischen Methodenaufrufs ist, daß Clients auf Interfaces zugreifen können, die zur Compilezeit noch nicht vorhanden sind. Der Nachteil ist die schlechtere Performance gegenüber statischen Aufrufen. Dynamische Aufrufe sollten demnach nur benutzt werden, wenn der Client das Serverobjekt zur Laufzeit entdeckt oder wenn der Client nur selten Operationen des Serverobjekts aufruft.

6.2.3.3 Dynamic Skeleton Interface

Das Dynamic Skeleton Interface (DSI) ist ein Interface für die dynamische Behandlung von Objektaufrufen (Server-Äquivalent zum DII). Über dieses Interface können Server CORBA-Objekte zur Verfügung stellen, ohne bereits zur Compilezeit Wissen über die Schnittstellen dieser Objekte besitzen zu müssen. Das bedeutet, daß keine statischen Skeletons benötigt werden. Statt dessen bestimmt das dynamische Skeleton anhand der Parameter des einge-

henden Requests das Objekt und die aufzurufende Methode. Der Einsatz vom DSI ist vor allem für Gateways zu anderen Objektsystemen oder Debugger sinnvoll.

6.3 Objekt Management Architektur

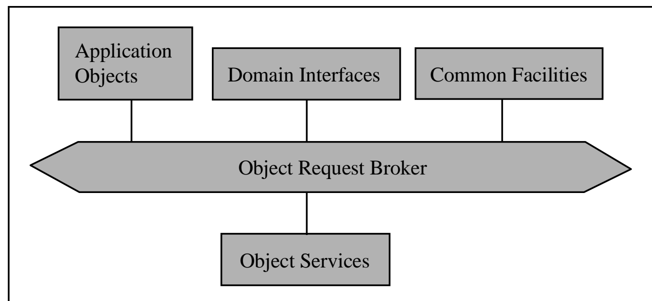


Abb. 30: Objekt Management Architektur[37]

Nachdem bisher die Funktionsweise von CORBA als Middleware beleuchtet wurde, beschreibt dieses Kapitel die Architektur von CORBA. Diese wird als Objekt Management Architektur (OMA) bezeichnet und bildet die Grundlage aller Spezifikationen der OMG (siehe Abb. 30). Diese Basisarchitektur teilt sich in eine systemorientierte und eine anwendungsorientierte Komponente auf. Zur systemorientierten Komponente gehören der ORB, der bereits in Kapitel 0 vorgestellt wurde, und die Object Services, die in Kapitel 6.3.1 behandelt werden. Die anwendungsorientierte Komponente besteht aus den Common Facilities (Kapitel 6.3.2), den Domain Interfaces (Kapitel 6.3.3) und den Application Objects (Kapitel 6.3.4).

6.3.1 Object Services

Die Object Services bestehen aus 15 standardisierten fundamentalen systemnahen Diensten. Diese Dienste, die in vielen CORBA-Anwendungen benötigt werden, besitzen jeweils ein IDL-Interface, um sie den Anwendungsentwicklern zugänglich zu machen. Weil die Object Services einen wesentlichen Bestandteil der Funktionalität von CORBA ausmachen, folgt eine Aufzählung mit Kurzbeschreibung jedes einzelnen Dienstes, auch wenn viele der Dienste bei den verschiedenen CORBA-Produkten (siehe Seite 38) noch nicht implementiert sind.

- Der **Life Cycle Service** bietet Operationen zum Erzeugen, Kopieren, Bewegen und Löschen von Objekten auf dem Objektbus an. Dies geschieht mit Hilfe des Fabrik-Musters[38].
- Mit Hilfe des **Persistent Object Service** können die normalerweise transienten CORBA-Objekte persistent gemacht werden. Angeboten wird ein Interface zum Speichern von Objekten in unterschiedliche Speicher wie z.B. ODBMS, RDBMS und einfache Dateien.
- Über den **Naming Service** können Objekte andere Objekte mittels eines externen Namens finden. Diese Namen sind hierarchisch organisiert. Dieser Service löst die Problematik, ein Objekt zu finden, ohne dabei den Ort zu kennen, an dem es sich aufhält (Ortstransparenz).
- Der **Event Service** erlaubt den Objekten, sich dynamisch für spezielle Ereignisse zu registrieren und zu deregistrieren. Wenn das Ereignis eintritt, werden alle registrierten Objekte benachrichtigt. Über die benutzten EventChannels wird eine Entkopplung von Sender und Empfänger des Ereignisses erreicht. Der Event Service unterstützt sowohl das Push- als auch das Pull-Modell.
- Der **Concurrency Control Service** bietet einen Lock-Manager an, der Transaktionen oder Threads sperren kann, um nebenläufige Handlungen zu synchronisieren.
- Der **Transaction Service** bietet das Zwei-Phasen-Commit-Protokoll für flache und eingebettete Transaktionen an. Er sichert die ACID-Eigenschaften (atomar, konsistent, isoliert und dauerhaft) von Transaktionen. Um die Nebenläufigkeit zu kontrollieren, wird der Concurrency Control Service verwendet.

---

[37] frei nach [STAL97] S. 29
[38] [GAMM95]

- Der **Relationship Service** bietet die Möglichkeit an, dynamische Assoziationen zwischen Objekten anzulegen, ohne daß die Objekte sich gegenseitig kennen.
- Durch den **Externalization Service** können zusammengesetzte Objekte in einen Stream transformiert werden. Dieser Stream kann von Objekten außerhalb des ORBs eingelesen oder in einer Datei gespeichert werden.
- Der **Query Service** dient dazu, Anfragen an Kollektionen von Objekten zu stellen. Die Implementation des Query Service unterstützt dabei entweder SQL oder OQL (structured / object query language). Dieser Service ist z.B. im Zusammenhang mit dem Relationship Service sinnvoll, da dieser immer Kollektionen von Objekten liefert.
- Durch den **Licensing Service** wird gewährleistet, daß die nicht lizensierte Benutzung von CORBA-Applikationen verhindert wird.
- Mit Hilfe des **Properties Services** können Eigenschaften mit einen Objekt assoziiert werden.
- Der **Time Service** bietet ein Interface zur Synchronisation der Uhrzeit und zur Behandlung von zeitgesteuerten Events an.
- Der **Security Service** stellt ein vollständiges Framework für verteilte Objektsicherheit zur Verfügung. Er bietet Datenschutz und Zugriffsrechte an.
- Der **Trader Service** bietet ein Branchenbuch für Objekte an. Jedes Objekt kann seinen Dienst anbieten oder einen anderen Dienst aufgrund definierter Kriterien suchen. Der Trader Service ist eine Alternative zum Naming Service in Bezug auf das Finden von Objektreferenzen zur Laufzeit.
- **Collection Service**: Dieser Service bietet ein CORBA-Interface an, mit dem die meisten Typen von Kollektionen (Mengen, Queues, Stacks, Listen, binäre Bäume, usw.) erzeugt und manipuliert werden können.

Der Implementationsaufwand der Objekt Services ist nicht gering. Darum haben viele ORB-Hersteller es bisher nicht geschafft, alle Services zu implementieren. Deshalb hat der Trend eingesetzt, daß zunehmend Drittanbieter einige der Services implementieren. Beispielsweise hat die Firma ObjectDesign den Persistence Service für NEO und O2Technology in Kooperation mit IONA den Persistence, Concurrency Control und Transaction Service implementiert.

6.3.2 Common Facilities

Common Facilities stellen eine Sammlung von Frameworks dar, die ein IDL-Interface besitzen und Services anbieten, die von Objekten direkt genutzt werden können. Diese infrastrukturellen Services sind von der Anwendungsdomäne unabhängig (z.B. E-Mail, Dokumentenverwaltung) und werden daher als horizontale Dienste bezeichnet. Die Spezifikation von Common Facilities ist ein niemals endendes Projekt, da alle verteilten Services ein IDL-Interface erhalten sollen. Die Common Facilities bilden die nächste Schicht über den Objekt Services (siehe Abb. 31).

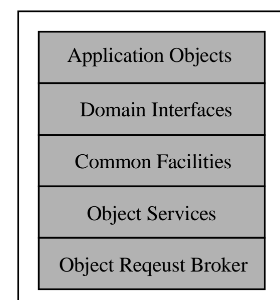


Abb. 31: OMA-Schichtenmodell

6.3.3 Domain Interfaces

Die Domain Interfaces sind Rahmenwerke für spezielle Domänen, z.B. für Gesundheitsdienste, Transportwesen, Telekommunikation und Finanzdienstleistungen. Sie bilden die Schicht über den Common Facilities. In früheren CORBA-Versionen wurden die Domain Interfaces als vertikale Common Facilities bezeichnet.

6.3.4 Application Objects

Application Objects (Geschäftsobjekte) sind vom Anwender entwickelte CORBA-Objekte. Diese Objekte werden von der OMG nicht standardisiert. Geschäftsobjekte können beispielsweise Kunde, Geld, Rechnung oder Auto sein. Sie sind die Konsumenten der CORBA-Infrastruktur. Geschäftsobjekte gehören zur obersten Schicht (Anwendungsebene) der Softwarehierarchie. Sie können in unvorhersehbaren Kombinationen eingesetzt werden und sind vollkommen unabhängig von einer einzelnen Anwendung. Deshalb können sie einfach wiederverwendet werden.

6.4 Visibroker for Java - eine CORBA-Implementierung

In diesem Kapitel wird die für das Fehlerbehandlungssystem eingesetzte CORBA-Implementation vorgestellt. Es handelt sich hierbei um den Visibroker Version 3.0 von Inprise (ehemals Borland, davor Visigenic). Ursprünglich wurde diese CORBA-Implementation von PostModern unter dem Namen Black Widow hergestellt. Doch 1996 kaufte Visigenic PostModern auf und benannte das Produkt in Visibroker um. Black Widow war der erste ORB, der Client- und Serverobjekte in Java unterstützte. Er ist vollständig in Java implementiert. Zusätzlich bietet Inprise noch den Visibroker for C++ (ehemals PostModerns Orbeline) an.

Der Arbeitsbereich Softwaretechnik des Fachbereichs Informatik der Universität Hamburg hat den Visibroker for Java gekauft. Zur Zeit scheint diese CORBA-Implementation sich am Markt durchzusetzen: Netscape hat Ende Juli 1996 bekannt gegeben hat, daß Visibroker for Java in Netscape ONE (Open Network Environment, Standardplattform für die Erzeugung von verteilten Client/Server Anwendungen fürs Internet) als ORB integriert werden soll. Oracle teilte im Februar 1997 mit, daß sie sich für VisiBroker entschieden haben.

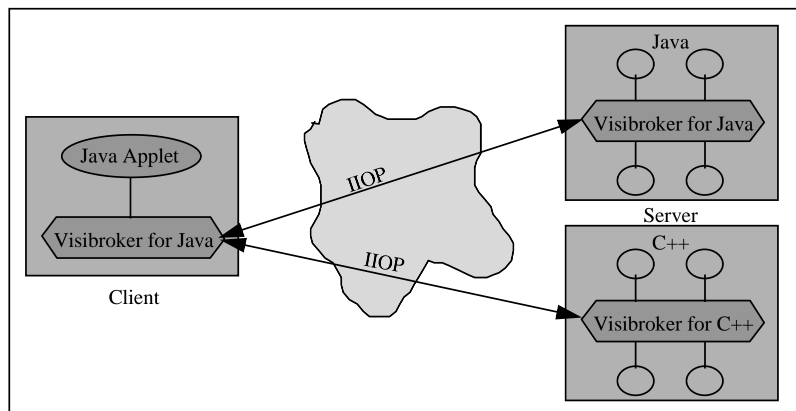


Abb. 32: Internetkommunikation mit Visibroker[39]

Visibroker ist eine weniger als 100 KB Java-Bytecode umfassende CORBA 2.0 Implementation plus vollständiger IIOP-Implementierung, Naming und Event Service. Mit Visibroker werden eine Reihe von Tools mitgeliefert:

- IDL2Java-Compiler für Compilierung von IDL nach Java
- Object Request Debugger
- OSAgent, ein fehlertoleranter Naming-Service. Wenn mehrere OSAgents im selben Netzwerk laufen, können sie sich gegenseitig lokalisieren und den Namensraum für Replizierung und Lastverteilung aufteilen. Wenn ein Server ausfallen sollte, wird der Client automatisch auf ein Replikat umgeleitet.
- Caffeine, ein spezielles Visibroker Feature, das automatisch aus Java-Bytecode Stubs, Skeletons und IDL generieren kann (Java2IIOP, Java2IDL, IDL2IR). Auf diese Weise wird CORBA für Java-Entwickler transparent gemacht, da kein IDL geschrieben werden muß.

---

[39] frei nach [ORFA97a] S. 48

6.5 Integration von CORBA ins JWAM-Framework

Am Arbeitsbereich Softwaretechnik des Fachbereichs Informatik der Universität Hamburg wurde das JWAM-Framework (Java-Werkzeug-Aspekt-Material[40]) entwickelt. Dort wurde die Verteilungsproblematik bisher mit RMI (Remote Method Invocation), einer zum JDK 1.1 dazugehörenden Middleware, gelöst. Im Rahmen dieser Diplomarbeit wurde die Verteilungskomponente des Frameworks um CORBA erweitert. Aus diesem Grund wurde besonderer Wert auf die Wiederverwendbarkeit der angestrebten Lösung gelegt, so daß sich der Verteilungsaspekt nicht nur auf das Fehlerbehandlungssystem bezieht, sondern von beliebigen Anwendungen, die das Problem der Verteilung besitzen, benutzt werden kann.

Prinzipiell kristallisieren sich drei Alternativen heraus, wie CORBA am sinnvollsten als Middleware eingesetzt werden kann. Die erste Möglichkeit ist die Generierung von IDL-Interfaces für Klassen, auf die entfernt zugegriffen werden soll. Dies führt zu dem Problem, daß das Framework die Middleware nicht wegkapseln kann, da die Klassen mit den IDL-Interfaces von speziellen CORBA-Klassen erben müssen und sich diese CORBA-Abhängigkeit durch die gesamte Klassenbibliothek durchzieht. Damit geht die gewünschte Unabhängigkeit des Frameworks von einer speziellen Middleware verloren. Außerdem müssen bei Erweiterungen des Frameworks erneut IDL-Interfaces generiert werden. Es scheint so, daß die Generierung von IDL-Interfaces nur dann sinnvoll ist, wenn man sich bei der Implementationsphase ganz auf CORBA festlegen will und alle Interfaces in IDL statt in der Implementationssprache spezifiziert.

Die zweite Möglichkeit des Einsatzes von CORBA ist ebenfalls recht naheliegend: Die Benutzung des CORBA Event-Service. Dieser ist vom damaligen Unternehmen Visigenic für Visibroker implementiert worden. Der CORBA Event-Service ermöglicht Objekten, sich dynamisch zur Laufzeit für bestimmte Ereignisse an- und abzumelden. Wenn dieses Ereignis eintritt, werden alle Interessenten asynchron benachrichtigt. Um eine lose Kopplung zwischen Ereignissendern (Suppliern) und Ereignisempfängern (Consumern) zu erhalten, wird zwischen Suppliern und Consumern ein Event-Channel geschaltet. Ein Event-Channel stellt ein Ereignis dar. Er übernimmt sowohl die Rolle des

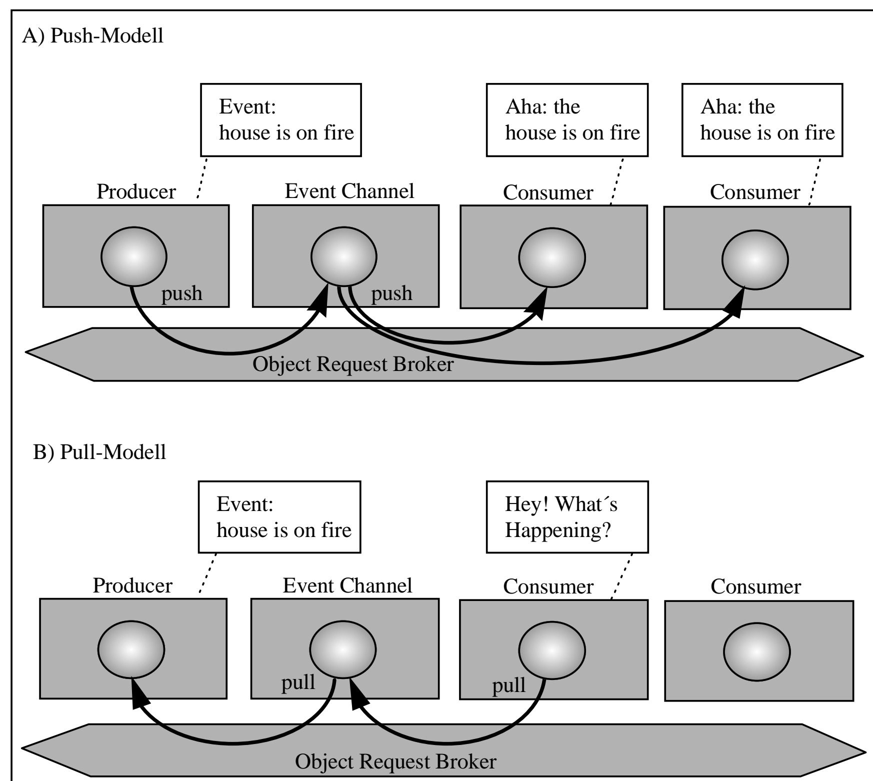


Abb. 33: Push- und Pull-Modell des CORBA Event-Service[41]

---

[40] siehe [ROOC98]
[41] aus [ORFA97], S. 117

Consumers, wenn ein Supplier ein Ereignis auslöst, als auch die Rolle des Suppliers, wenn das Ereignis vom Event-Channel zu den Consumern weitergeleitet wird. Es gibt zwei Kommunikationsmodelle: Das Push- und das Pull-Modell. Beim Push-Modell liegt die Initiative beim Supplier. Dieser pushed ein Ereignis zum Event-Channel, der es asynchron an alle angemeldeten Consumer weiterleitet (siehe Abb. 33A). Beim Pull-Modell liegt die Initiative beim Consumer. Er pullt den Event-Channel an um festzustellen, ob ein Ereignis für ihn anliegt. Der Event-Channel leitet die Anfrage an den Supplier weiter. Andere Consumer, die in dem Moment nicht pullen, erhalten das Ereignis nicht (der rechte Consumer in Abb. 33B). Supplier und Consumer, die an demselben Event-Channel angemeldet sind, können mit unterschiedlichen Kommunikations-Modellen kommunizieren. Die häufigere Kommunikationsart ist allerdings das Push-Modell, da es im Prinzip dem Observer-Pattern [GAMM95] entspricht.

Um Messages zwischen verteilten Anwendungen auszutauschen, müßte pro Messagetyp ein Event-Channel eingerichtet werden (siehe [FELT97], S. 35ff.). Problematisch wird es jedoch, wenn man die Vererbungsbeziehungen der Messagetypen berücksichtigen möchte. Angenommen Message B erbt von Message A und beide Messagetypen besitzen einen entsprechenden Event-Channel, bei dem sich die Consumer für den jeweiligen Messagetyp registrieren können (siehe Abb. 34). In diesem Fall muß bei dem Eintreten der Message B der Event-Channel A benachrichtigt werden. Wird eine neue Messageklasse C entwickelt, die von B erbt, so muß bei Erzeugung des Event-Channels C zur Laufzeit geprüft werden, ob es einen Event-Channel B gibt. Falls das der Fall ist, muß Event-Channel B sich bei Event-Channel C anmelden; falls das nicht der Fall ist, muß Event-Channel C in der Vererbungshierarchie weiter nach oben laufen, um einen bereits erzeugten Event-Channel zu finden. Falls es keinen gibt, kann sich kein Event-Channel bei Event-Channel C anmelden. Wird zu einem späteren Zeitpunkt ein Event-Channel erzeugt, der in der Hierarchie über dem Event-Channel C steht (z.B. A), so muß sichergestellt werden, daß sich dieser bei C registriert. Dazu müßte eine Klasse seine Erben kennen, was aber nicht der Fall ist.

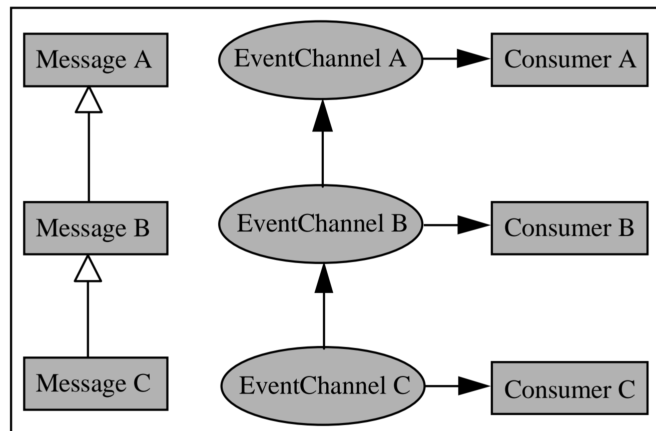


Abb. 34: Lösung mit Event-Channeln

Diese Art der Registrierung ist daher sehr aufwendig. Alternativ könnten bei der Erzeugung eines Event-Channels alle Channels der Supertypen mit erzeugt werden. Dies würde aber zu einer hohen Anzahl von Prozessen führen, die möglicherweise gar nicht benötigt werden (Ressourcenverschwendung). Außerdem verursacht eine große Anzahl von Event-Channeln eine hohe Netzbelastung (jedes Eintreten einer Message zieht eine Benachrichtigungskette bis zur Wurzelklasse der Messages nach sich). Der Effekt verstärkt sich noch, wenn nicht alle Event-Channel auf demselben Rechner laufen. Aus diesen Gründen wurde auf eine Implementation mit Event-Channeln verzichtet.

Ein weiteres Problem bei der Implementierung mit Event-Channeln besteht darin, daß das Konzept der Klauseln nicht funktioniert. Dieses in JWAM integrierte Konzept ermöglicht es, zusätzlich zu einer Message noch eine Klausel (Boolscher Wert) anzugeben. Wenn die Message eintritt, wird zuerst überprüft, ob die Klausel erfüllt ist. Nur in diesem Fall wird die

Message an die angemeldete Komponente gesendet. Ein Event-Channel sendet die Message an alle angemeldeten Consumer. Der Consumer müßte anschließend selber den Klausel-Check durchführen. Dies ist jedoch inakzeptabel, da der Klausel-Check die Aufgabe des Frameworks ist.

Die dritte Möglichkeit CORBA als Middleware zu benutzen besteht darin, CORBA in das Message-broker-Konzept des JWAM-Frameworks zu integrieren[42]. Dieses Konzept ähnelt dem Event-Channel-Konzept, nur daß sich ein Client (Consumer) immer bei dem lokalen *ToolAutomaton-MessageCoordinator* (TAMCO) für alle Messages registriert bzw. deregistriert (siehe Abb. 35). Dieser registriert sich für jeden Messagetypen einmal beim *MessageBroker*. Dieser läuft im selben Prozeß wie der Client und ist für das Versenden von Messages zum *MessageBrokerServer* und für das Empfangen von Messages vom *MessageBrokerServer* zuständig. Für jeden Messagetypen registriert sich der *MessageBroker* beim *MessageBrokerServer*.

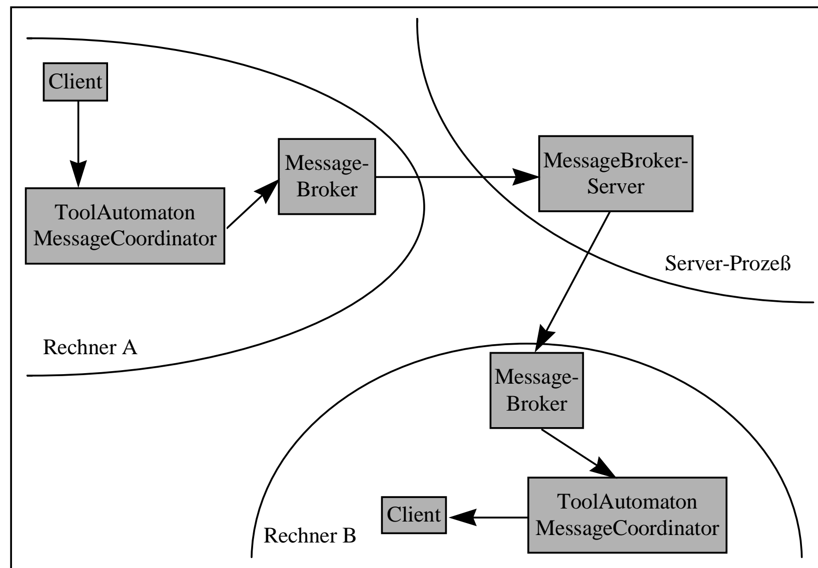


Abb. 35: Architektur des Messagebrokers

Eine Message ist dabei immer ein Exemplar der Klasse *msgObject*. Von ihr kann ein Anwendungsentwickler spezielle Messages ableiten, die beispielsweise Zustandsinformationen von speziellen Objekten beinhalten können.

Wird eine Message von einem Client an den TAMCO gesendet, so leitet der TAMCO die Message an den *MessageBroker* weiter. Dieser konfiguriert die Message mit einer eindeutigen ProzeßID und sendet die Message zunächst an die lokal angemeldeten Clients und anschließend an den *MessageBrokerServer* weiter. Alle *MessageBroker*, die sich für diesen Messagetypen angemeldet haben, werden in jeweils eigenen Threads benachrichtigt. Dabei wird zuerst überprüft, ob die ProzeßID der Message mit der des eigenen Prozesses übereinstimmt. Falls dies der Fall ist, kommt die Message aus dem eigenen Prozeß und braucht nicht weiter beachtet zu werden, da sie bereits lokal versendet wurde. Im anderen Fall werden die lokal angemeldeten Clients benachrichtigt. Eine detailliertere Beschreibung des Benachrichtigungsverfahrens von JWAM einschließlich der Synchronisation von Messages mit GUI-Events, die der Benutzer gleichzeitig zu ankommenden Messages auslösen kann, ist in [ROOC98], S. 141ff. zu finden[43].

Das vorgestellte Konzept ist mittels RMI (Remote Method Invocation) implementiert worden. Ziel ist es gewesen, CORBA parallel zu RMI derart in das Java-Framework zu integrieren, daß die Schnittstellen für die Clients (also dem TAMCO) unverändert bleiben. Um dies zu

[42] siehe [ROOC98], S.138ff. und [JWAM98]
[43] siehe auch [JWAM98]

erreichen, wurde die gesamte RMI-Funktionalität aus dem *MessageBroker* entfernt und in die Klasse *RMIMessageBroker* (RMIMB) verschoben (siehe Abb. 36, in der punktierte Pfeile eine Implementationsbeziehung und durchgezogene Pfeile eine Benutztbeziehung darstellen).

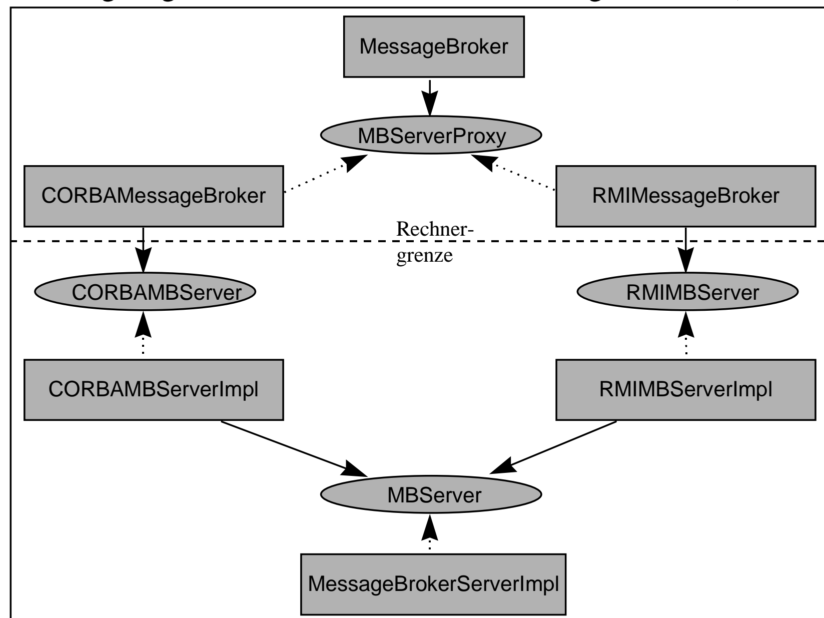


Abb. 36: Architektur - vom Client zum Server

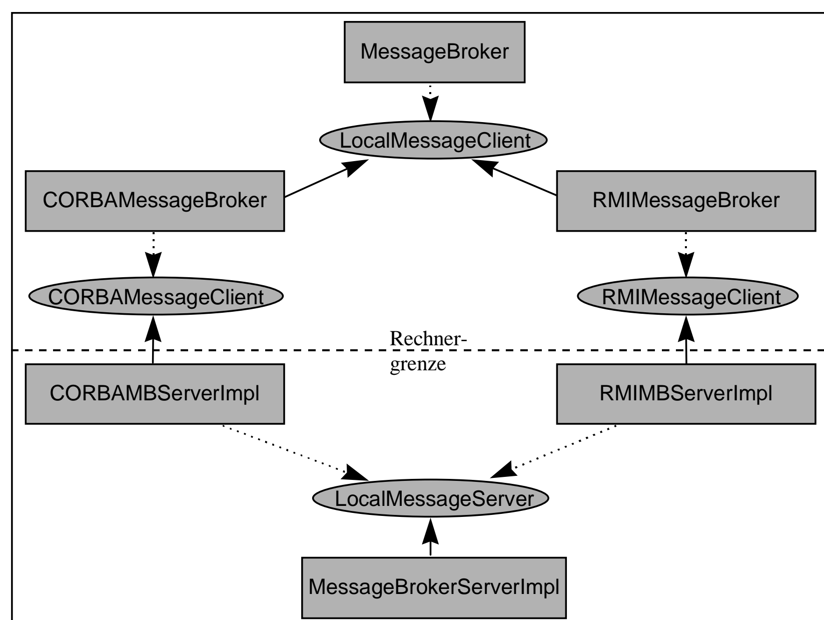


Abb. 37: Architektur - vom Server zum Client

Parallel zu dieser Klasse gibt es den *CORBAMessage-Broker* (CORBAMB), der die entsprechende CORBA-Funktionalität enthält. Abhängig von der Umgebungsvariable *MBSProxyClass* beschafft sich der *Message-Broker* zur Laufzeit das jeweils einzige Exemplar vom RMIMB oder CORBAMB, das unter der Schnittstelle *MBServerProxy* als Attribut in der Klasse *MessageBroker* gehalten wird. Der CORBAMB bindet sich an den *CORBA-MessageBrokerServerImpl* (CORBAMBSI), der RMIMB an den *RMIMessage-BrokerServerImpl* (RMIMBSI). Diese beiden Paare transportieren in Abhängigkeit von ihrer Middleware die Messages über das Netzwerk. Das Konzept kann durch beliebige weitere Middleware-Systeme ausgebaut werden. Bevor sich die Middleware-abhängigen *MessageBroker* jedoch an die Server binden können, müssen diese zunächst gestartet werden. Dies geschieht durch Starten des *MessageBrokerServer-Impl*s (MBSI). Als Argumente werden dabei die Klassennamen der Middleware-abhängigen Server übergeben (CORBAMBSI, RMIMBSI, etc.). Diese werden erzeugt und den Clients unter einem definierten Namen bekannt gemacht. Der MBSI macht sich selber noch bei den von ihm erzeugten Servern unter der Schnittstelle *MessageBrokerServer* bekannt. Die RMI-Funktionalität des MBSI wurde konsequenterweise in die Klasse RMIMBSI verschoben. Als Ergebnisarchitektur erhält man zwei Middleware-unabhängige Klassen *MessageBroker* und *MessageBrokerServerImpl*, die jeweils über ein Paar von Middleware-abhängigen Klassen miteinander kommunizieren können.

Wenn nun ein Client eine Message vom Typ *msgObject* versenden möchte, sendet er diese über den TAMCO an den *MessageBroker*. Dieser sendet sie, wenn CORBA als Middleware benutzt wird, an den CORBAMB, der das *msgObject* in ein ByteArray verpackt und zum CORBAMBSI weiterreicht (siehe Abb. 36). Dieser packt das *msgObject* aus dem ByteArray wieder aus und sendet es zum MBSI. Weil der MBSI nicht wissen kann, welcher Client über welche Middleware angemeldet ist, sendet er die Message samt seiner Empfänger an alle ihm bekannten middlewareabhängigen Server (siehe Abb. 37). (Dies ist etwas vereinfacht, da noch einige Hilfsklassen dazwischen liegen, die jeweils eigene Threads zur Verarbeitung erzeugen, um eine asynchrone Kommunikation zu gewährleisten.) Bei detaillierterem Interesse siehe [ROOC98][44].) Das CORBAMBSI prüft, ob der Typ des Empfängers ein CORBAMessageClient ist. Falls das der Fall ist, verpackt er die Message in ein ByteArray und sendet es zum CORBAMB, der die Message aus dem ByteArray auspackt und an den *MessageBroker* weiterleitet, der die Clients benachrichtigt.

Der Grund dafür, daß *msgObject* kein IDL-Interface erhalten hat, was zunächst recht naheliegend erscheint, hat mit dem JWAM-Framework als solches zu tun. Ein Framework soll möglichst unabhängig von speziellen Middleware-Systemen sein. Wenn Klassen des Frameworks ein IDL-Interface erhalten ist dies leider nicht mehr der Fall. Aus diesem Grund besitzen nur die beiden CORBA-Klassen ein IDL-Interface (siehe Abb. 38). Das Interface *CORBA-MessageBrokerServer* besitzt Methoden zum Registrieren und Deregistrieren von Clients. Die Methode *createSenderProcessID* wird von den CORBAMB benötigt, um eindeutige Prozeß-IDs zu erzeugen. Die Methode *sendMessage* dient dem Senden einer Message zum CORBAMBSI. Das Interface *CORBAMessageClient* besitzt nur eine Methode zum Empfangen von Messages in Form von ByteArrays vom CORBAMBSI. Um die Messages ohne IDL-Interface versenden zu können, wurde für die Java-Klasse *ByteArrayOutputStream* ein IDL-Interface spezifiziert. Über dieses Interface können beliebige Java-Klassen in ein ByteArray geschrieben werden.

```
module wam {
 module distribution {
  module messaging {
   module CORBA {
    module MessageModul {
     typedef sequence<octet> meinArray;

     interface ByteArrayOutputStream{
      meinArray toByteArray();
     };

     interface CORBAMessageClient {
      void receiveMessage(in ByteArrayOutputStream byteStream);
     };

     interface CORBAMessageBrokerServer{
      unsigned long createSenderProcessID();
      void sendMessage (in ByteArrayOutputStream byteStream);
      void register (in CORBAMessageClient client, in string messageType);
      void register2 (in CORBAMessageClient client, in string messageType,
                      in ByteArrayOutputStream clause);
      void unregister (in CORBAMessageClient client, in string messageType);
      void unregister2 (in CORBAMessageClient client, in string messageType,
                      in ByteArrayOutputStream clause);
     };
    };
   };
  };
 };
};
```

Abb. 38: IDL-Modul

[44] siehe auch [JWAM98]

Die Vorteile der vorgestellten Lösung sind, daß

- Clients sich erst zur Laufzeit für eine Middleware (CORBA oder RMI) entscheiden müssen,
- Clients unabhängig von der Middleware dieselbe Message erhalten,
- es nur eine Messagehierarchie mit *msgObject* als Wurzel gibt und daß
- keine IDL-Verschmutzung des Java-Frameworks droht.

Der entscheidende Nachteil der vorgestellten Lösung besteht darin, daß andere Programmiersprachen, die über CORBA Messages senden oder empfangen wollen, den Java-Serialisationsmechanismus nachimplementieren müssen. Dies ist zwar möglich, da die Java-Sprachspezifikation allen offen steht (virtuelle Maschine), aber dennoch umständlich. Die aktuelle Lösung ist demnach sprachabhängig und kann gehässig als „CORBA ohne CORBA" bezeichnet werden, da der Hauptgedanke von CORBA schließlich die Sprachunabhängigkeit ist. Um dies Problem zu lösen, müßte *msgObject* und alle seine Nachkommen ein IDL-Interface erhalten, um die Sprachunabhängigkeit zu gewährleisten. Daraus resultiert ein CORBA-Overhead, der ebenfalls bei RMI-Benutzung anfällt. Um diesen Overhead vor RMI zu verbergen, müßten parallele Messagehierarchien für jede eingesetzte Middleware aufgebaut werden. Dies führt aber zu erhöhtem Programmier- und Wartungsaufwand und dem Verlust der Möglichkeit, middlewareübergreifend zu kommunizieren. Im übrigen müßten entsprechend die fachlichen Klassenbibliotheken mit IDL versehen werden, da die fachlichen Klassen ebenfalls entfernt mit anderen Programmiersprachen benutzt werden. Das führt dazu, daß zum Zeitpunkt der Entscheidung, mit welcher Programmiersprache entwickelt werden soll, sich für oder gegen CORBA entschieden werden muß. Damit würde die Transparenz der Middleware im Framework verloren gehen.

Nachdem nun die Basis für die Verteilung des Fehlerbehandlungssystem gelegt wurde, wird im folgenden Kapitel der Entwurf und die Implementation des Fehlerbehandlungssystems vorgestellt.

7 Entwurf und Implementierung des Fehlerbehandlungssystems

Dieses Kapitel zeigt auf, wie die in Kapitel 4.2 aufgestellten Anforderungen an das Fehlerbehandlungssystem umgesetzt wurden. Kapitel 7.1 stellt die Ausnahmen vor, die dem Programmierer zur Verfügung gestellt werden. Um die Ausnahmen unabhängig von einer speziellen Middleware persistent machen zu können, wird ein besonderer Mechanismus verwendet, der Bestandteil von Kapitel 7.2 ist. Im Anschluß daran wird die Umsetzung des Vertragsmodells erläutert (Kapitel 7.3). Während das Fehlerbehandlungssystem ohne die Benutzung der definierten Ausnahmen nicht eingesetzt werden kann, ist die Anwendung des Vertragsmodells zwar empfohlen, aber nicht unbedingt erforderlich. Der nächste Teil dieses Kapitel (7.4) beschäftigt sich mit der Problematik der mehrsprachigen Erfassung der Fehlermeldungen, die von den jeweiligen Fachabteilungen vorgenommen werden kann. Während die Klassen der ersten drei Kapitel in dem Package *Fehlertool* zusammengefaßt sind, befinden sich die Klassen der mehrsprachigen Fehlertexterfassung im Package *Fehlertool.Fehlertext*. Im letzten Teil dieses Kapitels wird auf die Besonderheiten bei der Implementation mit Java eingegangen.

7.1 Die Ausnahmen

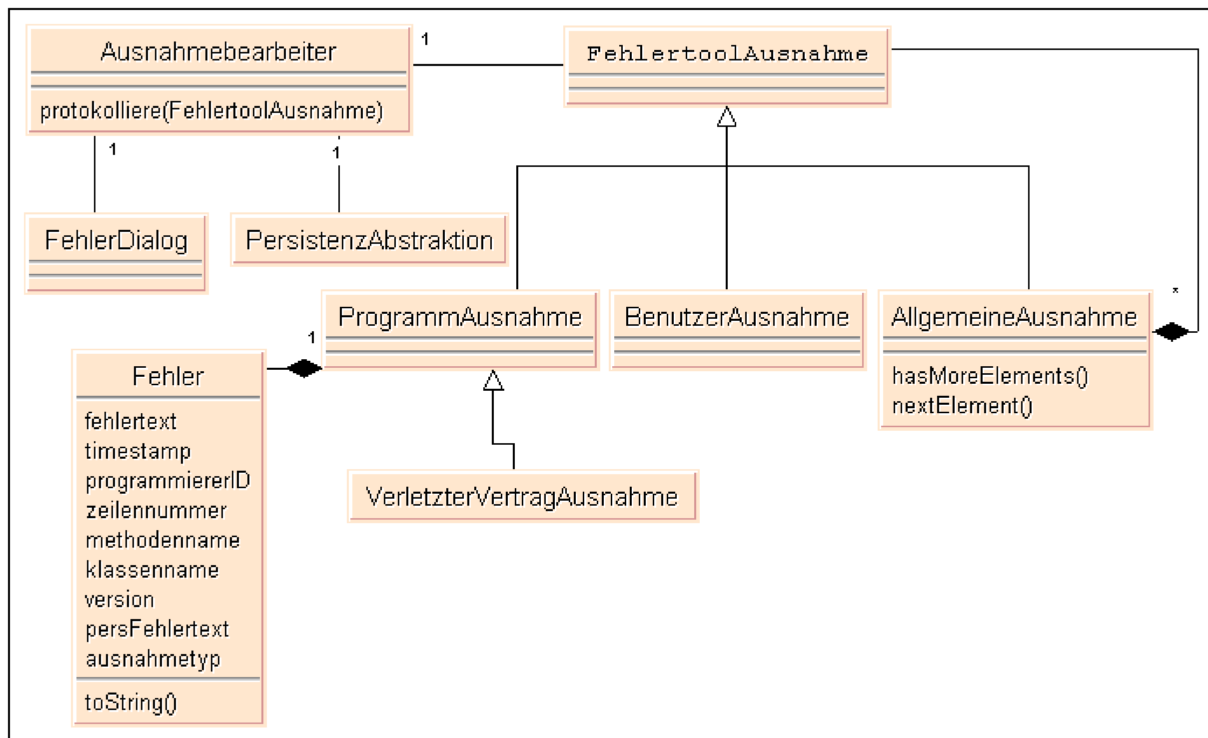


Abb. 39: Klassendiagramm Fehlertool-Ausnahmen

Die Oberklasse aller Ausnahmen des Fehlerbehandlungssystems ist die abstrakte Klasse *FehlertoolAusnahme* (siehe Abb. 39). Sie erbt direkt von *java.lang.Exception* (der Oberklasse aller Ausnahmen, die von Anwendungsprogrammen aufgefangen werden sollten) und ist damit in das Ausnahme-Konzept von Java, wie in einer Anforderungen verlangt wurde, integriert worden. Die beiden geforderten Ausnahmegruppen (anormales Verhalten und Programmfehler) werden durch die Klassen *BenutzerAusnahme* und *ProgrammAusnahme* repräsentiert: Die *BenutzerAusnahme* dient dazu, Fehlermeldungen, Warnungen und Informationen dem Benutzer in einem Dialogfenster anzuzeigen, um ihn auf fehlerhafte Eingaben aufmerksam zu machen. Die Fehlermeldung wird im Konstruktor angegeben. Die *ProgrammAusnahme* zeigt Programmfehler an. Sie aggregiert die Klasse *Fehler*, die die für

die Fehlersuche dringend benötigten Informationen zusammenfaßt. Der Methodenname und die Zeilennummer des Fehlers müssen vom Programmierer explizit übergeben werden. Das Attribut *persFehlertext* enthält den Grund für ein mögliches Problem beim Speichern des Fehlers (z.B. MQSeries-Fehler). Die *AllgemeineAusnahme* aggregiert standardmäßig die *BenutzerAusnahme* und die *ProgrammAusnahme*. Dabei wurde das Kompositummuster[45] benutzt. Die *AllgemeineAusnahme* wird benötigt, wenn ein Programmierfehler aufgetreten ist und der Anwender eine Fehlermeldung erhalten soll. Die Ausnahme-Klassen und die Klasse Fehler stellen somit das Material gemäß der WAM-Metapher ([ZÜLL98]) dar.

Wenn das Defaultverhalten der Ausnahmen benutzt werden soll, so muß die *FehlertoolAusnahme* dem *Ausnahmebearbeiter* übergeben werden. Dieser sorgt dafür, daß abhängig vom konkreten Typ der Ausnahme ein Fehlerdialog geöffnet wird und/oder der Fehler persistent gemacht wird. Das Speichern wird von der Klasse *PersistenzAbstraktion* übernommen (siehe Kapitel 7.2). Gemäß dem WAM-Metapher handelt es sich beim *Ausnahmebearbeiter* um einen Automaten.

7.2 Das PersistenzInterface

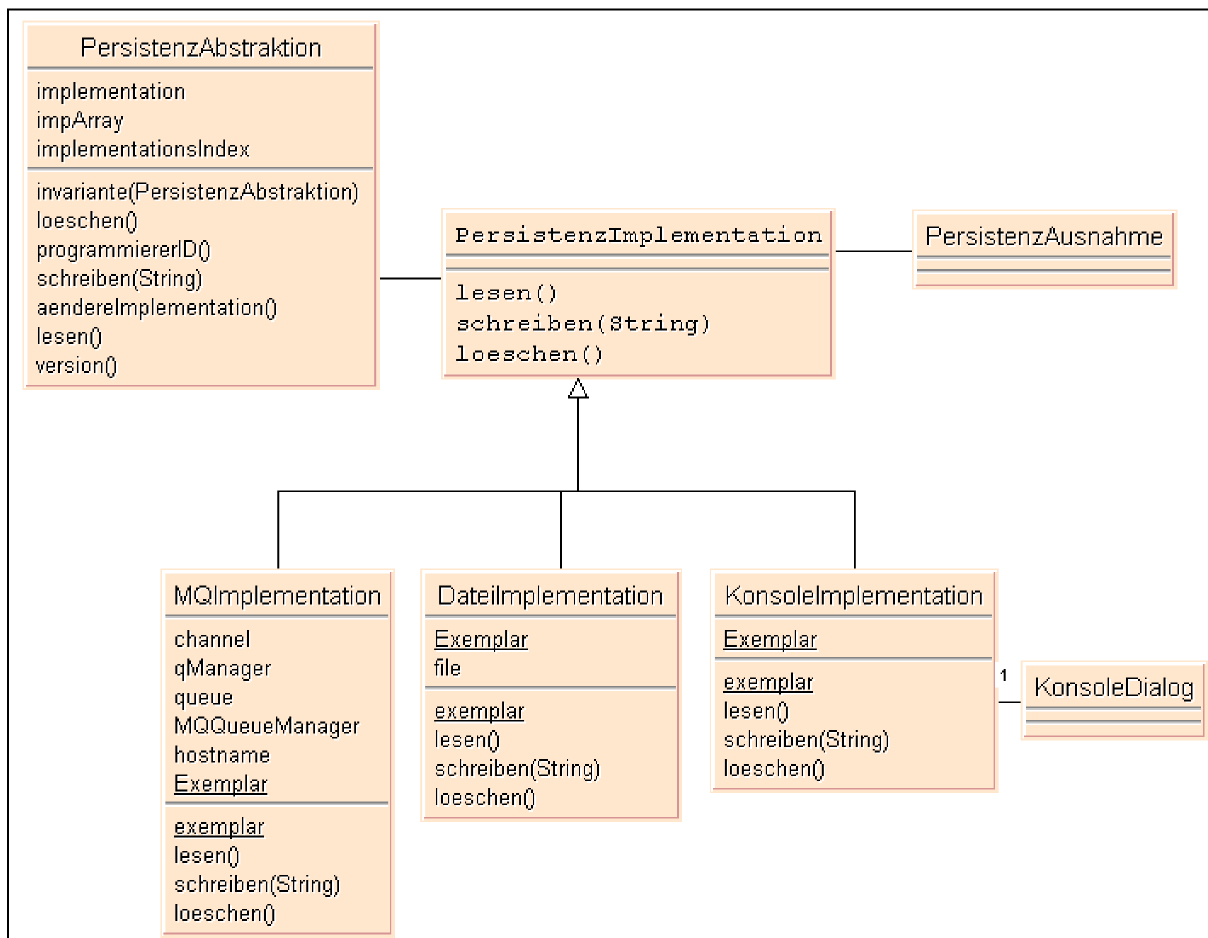


Abb. 40: Klassendiagramm Persistenz-Mechanismus

Um den Fehler persistent zu machen, benutzt der *Ausnahmebearbeiter* die *PersistenzAbstraktion* (siehe Abb. 40), die eine Brücke zum Persistenz-Mechanismus darstellt (Brückenmuster[46]). Der Persistenz-Mechanismus stellt Schnittstellen für unterschiedliche

[45] aus [GAMM96], S.213 ff.
[46] aus [GAMM96], S.165 ff.

Persistenz-Mechanismen zur Verfügung. In der Hamburg-Mannheimer Versicherungs-AG existiert eine Implementation für MQSeries (*MQImplementation*) mit einem festgelegten Queue-Manager und einer Fehler-Queue (siehe Kapitel 5.5), eine Implementation für eine Datei (*DateiImplementation*) und eine Implementation für eine Konsole (K*onsoleImplementation*), von der der Fehler ausgedruckt werden kann. Als erstes wird versucht, den Fehler mit MQSeries persistent zu machen. Gelingt dies nicht, so wird als zweiter Schritt versucht, den Fehler in eine Datei zu schreiben. Sollte dies ebenfalls fehlschlagen (beispielsweise wegen fehlendem Schreibrecht), so wird der Fehler auf einer Konsole ausgegeben.

Um mit CORBA eine weitere Implementation zu erhalten, wird eine Klasse *Fehlermessage* eingeführt, die von *msgObject* (siehe Kapitel 6.5) erbt und ein Attribut *Fehler* mit den beiden üblichen get- und set-Methoden besitzt. Diese *Fehlermessage* wird von der *CORBAImplementation*, die anstelle der *MQImplementation* eingesetzt werden kann, zu einem *CORBAFehlerServer* gesendet, der die Fehler anzeigen und in einer Datei speichern kann.

7.3 Das Vertragsmodell

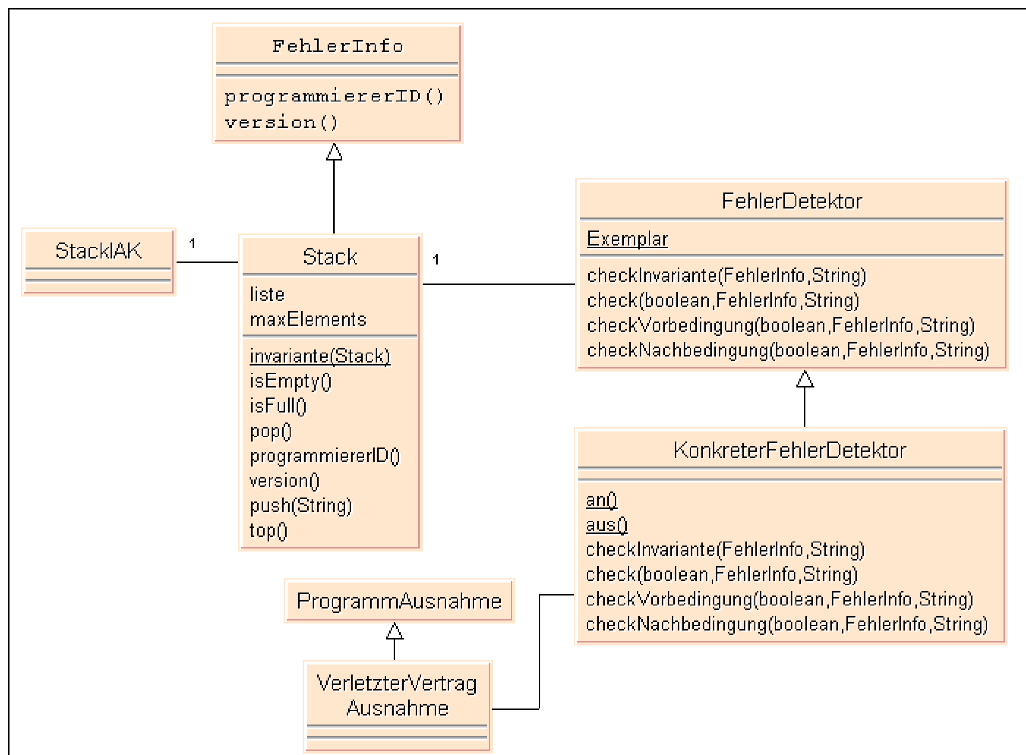


Abb. 41: Klassendiagramm FehlerDetektor

Eine Anforderung an das Fehlerbehandlungssystem ist die Implementation des Vertragsmodells von Meyer. Da Java die von Meyer in Eiffel eingeführten Schlüsselwörter nicht enthält, bietet es sich an, die Reaktion auf die Prüfung von Vor-/Nachbedingungen und Invarianten durch Delegation zu lösen (siehe Kapitel 3.5). Dazu muß sich die Anwendungsklasse (im Beispiel in Abb. 41 ist das die Klasse Stack) über die statische Methode *Exemplar* des *FehlerDetektors* das einzige Exemplar (Singleton-Pattern[47]) von *KonkreterFehlerDetektor* besorgen. Über dieses Exemplar kann mittels der check-Routinen die Vor- und Nachbedingung sowie die Invariante überprüft werden. Bei der Vor- und Nachbedingung (und bei der

[47] aus [GAMM95], S.139ff.

einfachen Check-Routine) werden als Parameter die Bedingung, die eigene Klasse (unter der Schnittstelle *FehlerInfo*) und optional der eigene Methodenname (als String) übergeben. Ein Beispiel ist in Abb. 42 zu sehen.

```
public String pop () throws VerletzterVertragAusnahme {
   FehlerDetektor.Exemplar().checkVorbedingung(!isEmpty(),this);
   FehlerDetektor.Exemplar().checkInvariante(this);
   // eigentlicher Code.
   String result = ...
   ...
   FehlerDetektor.Exemplar().checkInvariante(this);
   FehlerDetektor.Exemplar().checkNachbedingung(!isFull(),this);
   return result;
 }
```

Abb. 42: Pop-Methode des Stacks mit Vertragsmodell

Alle Klassen, die das Vertragsmodell benutzen wollen, müssen das Interface FehlerInfo implementieren. Zusätzlich muß die statische Methode *invariante* implementiert werden, die als Parameter ein Objekt der eigenen Klasse erhält. Diese Methode wird bei der Invariantenüberprüfung vom *FehlerDetektor* aufgerufen. Sollte die Oberklasse diese Methode implementiert haben, so wird die Methode der Oberklasse (konform mit dem Vetragsmodell) zusätzlich aufgerufen und konjunktiv verknüpft. Wenn eine der Check-Routinen einen Fehler entdeckt hat, wird eine *VerletzterVertragAusnahme* ausgelöst. Im Fehlertext steht, um welche verletzte Bedingung es sich dabei handelt.

Die von Meyer geforderte disjunktive Verknüpfung der Vorbedingung einer Routine mit den Vorbedingungen der überschriebenen Routinen aller Oberklassen und die konjunktive Verknüpfung der Nachbedingung einer Routine mit den Nachbedingungen der überschriebenen Routinen aller Oberklassen wurde bei der Implementierung nicht berücksichtigt. An dieser Stelle muß der Anwendungsentwickler selbst dafür sorgen, daß seine Bedingungen diese Problematik berücksichtigen.

Um Performanceverbesserungen zu erzielen, gibt es die Möglichkeit, die gesamte Fehlerentdeckung auszuschalten (Methode *Aus* von *KonkreterFehlerDetektor*). In diesem Fall wird ein Objekt der Klasse *FehlerDetektor* anstelle von *KonkreterFehlerDetektor* von der Methode *Exemplar* zurückgegeben. Die Check-Methoden von *FehlerDetektor* sind im Gegensatz zu denen von *KonkreterFehlerDetektor* leer implementiert (Umsetzung des Null-Objektmusters[48]).

Bei der Benutzung des Fehlerbehandlungssystems sollten folgende Richtlinien eingehalten werden (siehe auch Abb. 42):

- Der Vorbedingungs-Check ist die erste Zeile einer Routine, z.B.: *FehlerDetektor.Exemplar().checkVorbedingung(!isEmpty(),this);*
- Der Invarianten-Check findet immer in der zweiten Zeile einer Routine statt. Außerdem wird er bei Methoden, die void liefern, in der vorletzten, bei allen anderen Methoden in der drittletzten Zeile durchgeführt, z.B.: *FehlerDetektor.Exemplar().checkInvariante(this);*
- Der Nachbedingungs-Check ist bei Methoden, die void liefern, die letzte Zeile, bei allen anderen die vorletzte, z.B.:

[48] aus [ANDE96]

FehlerDetektor.Exemplar().checkNachbedingung(!isFull(),this);

- Methoden, die kein void liefern, besitzen eine Variable result vom Typ des Rückgabewertes, die vor dem zweiten Invarianten-Check zugewiesen wird. Die letzte Zeile solcher Routinen lautet immer *return result;*.
- VerletzterVertragAusnahmen sollten immer protokolliert werden, da sie gravierende Fehler anzeigen.
- Die Implementation der Methode *version* vom FehlerInfo-Interface sollte einen Timestamp beinhalten (z.B.: *return "1.0 vom 9.9.1997";*). Bei jeder Änderung einer Klasse sollte dieser Code aktualisiert werden. Ist dies der Fall, kann ein Timestamp aus kürzester Vergangenheit darauf hindeuten, daß der geänderte Code noch nicht ausreichend getestet wurde. Da Änderungen immer fehleranfällig sind, ist es in einem laufenden System wahrscheinlich, daß die Ursache für den aufgetretenen Fehler eine Folge der durchgeführten Änderung ist.

7.4 Die mehrsprachige Fehlertexterfassung

Die Möglichkeit, Fehlermeldungen mehrsprachig erfassen zu können, stellte eine weitere Anforderung dar. Die Umsetzung dieser Anforderung ist Inhalt dieses Kapitels. Im folgenden wird zunächst auf die Architektur der Fehlertexterfassung eingegangen; im Anschluß daran wird beschrieben, wie diese von den Mitarbeitern der Fachabteilung benutzt werden sollte. Zum Schluß wird erläutert, wie die von der Fachabteilung generierten Klassen in die Anwendung eingebaut werden müssen.

7.4.1 Architektur

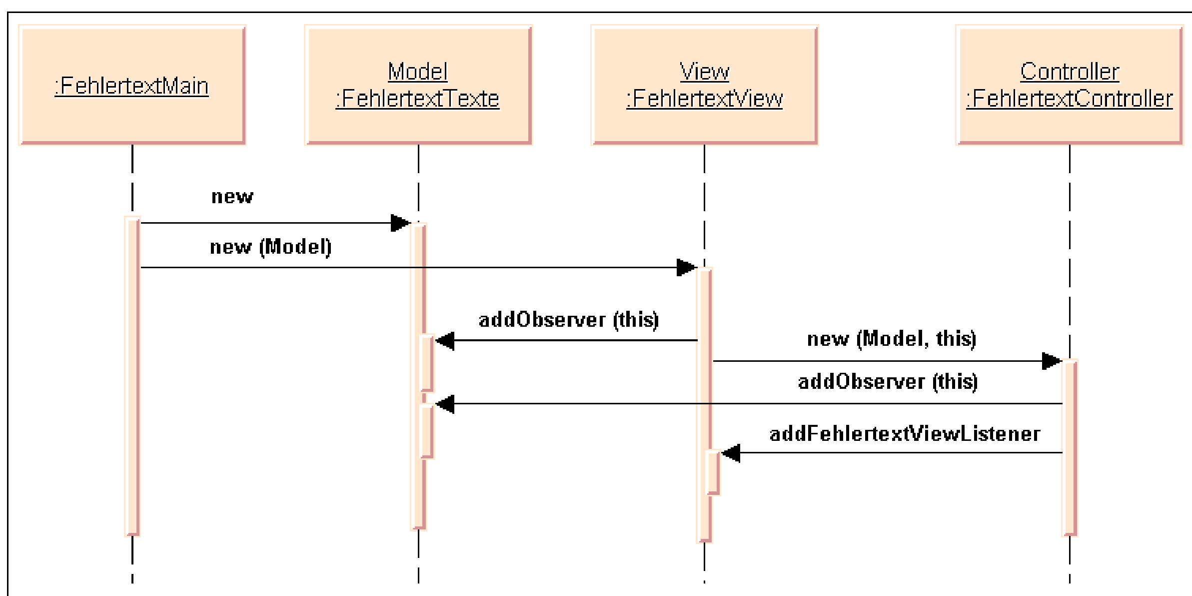


Abb. 43: Aufbau der Fehlertext-Anwendung gemäß MVC

Grundlage für die Architektur der Erfassung der Fehlertexte des Fehlerbehandlungssystems ist das MVC-Paradigma (Model-View-Controller, [RUMB94]). Der Grund dafür ist, daß in der HM standardmäßig mit dem MVC-Konzept gearbeitet wird, da Smalltalk als OO-Programmiersprache eingesetzt wird (mit Ausnahme des Java-Forschungsprojekts) und MVC die Grundlage für Smalltalk darstellt. Dabei enthält das Model die fachlichen Daten und die fachliche Logik. Die View dient der grafischen Darstellung des Models am Bildschirm. Der Controller steuert die Benutzereingaben, wie z.B. Buttonklicks oder Textfield-Eingaben. Sowohl die View als auch der Controller stehen in einer Observer-Observable-Beziehung ([GAMM95], S. 257ff.) zum Model: Sobald sich am Modell etwas ändert, werden View und

Controller benachrichtigt. Die View und der Controller hingegen stehen in einer beidseitigen Benutzt-Beziehung. Der Controller erhält die Events der Widgets (Buttons, Textfields, etc.) und reagiert darauf, indem er entsprechende Methoden des Modells aufruft. Die Eingabedaten beschafft sich der Controller bei der View, die entsprechend öffentliche Methoden zur Verfügung stellen muß.

Abb. 43 zeigt den Aufbau der Fehlertext-Anwendung, wie es das MVC-Konzept vorschreibt. Als erstes wird von der Klasse *FehlertextMain* die Methode main() aufgerufen. Diese erzeugt das Model *FehlertextTexte* und übergibt es der View, die als zweites erzeugt wird. Im Konstruktor der *FehlertextView* meldet sich die View als Observer beim Model an, um von Veränderungen an diesem zu erfahren. Außerdem wird der *FehlertextController* erzeugt, der im Konstruktor ebenfalls das Model und zusätzlich die View übergeben bekommt. Der *FehlertextController* meldet sich im Anschluß daran beim Model als Observer und bei der View als Listener an. Das ist notwendig, damit die Ereignisse der Widgets von der View zum

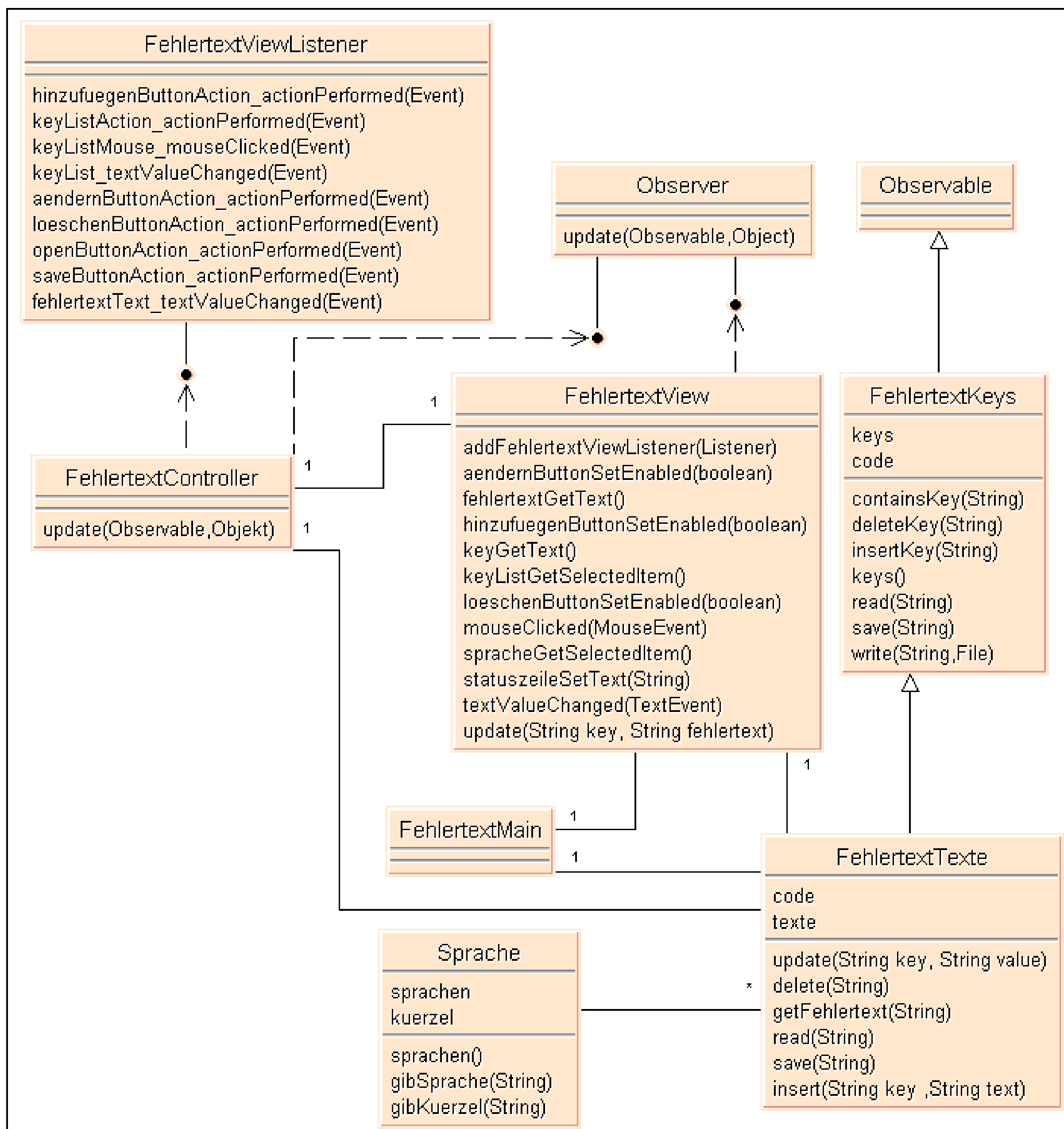


Abb. 44: Klassendiagramm des Fehlertext-Packages

Controller gelangen. Damit ist der Aufbau der Fehlertext-Anwendung abgeschlossen. Der *FehlertextController* wartet auf Ereignisse des Benutzers.

Aus Abb. 44 ist ersichtlich, daß das Model *FehlertextTexte* von *FehlertextKeys* erbt. *FehlertextKeys* hat als Attribut eine Liste von Keys. Jede Fehlermeldung, die erfaßt wird, benötigt so einen Key. Die erfaßten Paare aus Key und Fehlermeldung werden in einer Liste in der Klasse *FehlertextTexte* gehalten. Diese werden jeweils in einer Sprache erfaßt, die der Benutzer auswählen kann. Für jede erfaßte Sprache existiert ein Exemplar von *FehlertextTexte*. Sowohl *FehlertextKey* als auch *FehlertextTexte* dienen dazu, Javacode zu generieren, der in die Anwendung eingebettet werden muß. Der generierte Code der Oberklasse enthält dabei eine Liste mit vom Benutzer eingegebenen Fehlerkeys, die Unterklasse enthält eine Liste mit Fehlerkey-Fehlertext-Paaren in der gewünschten Sprache.

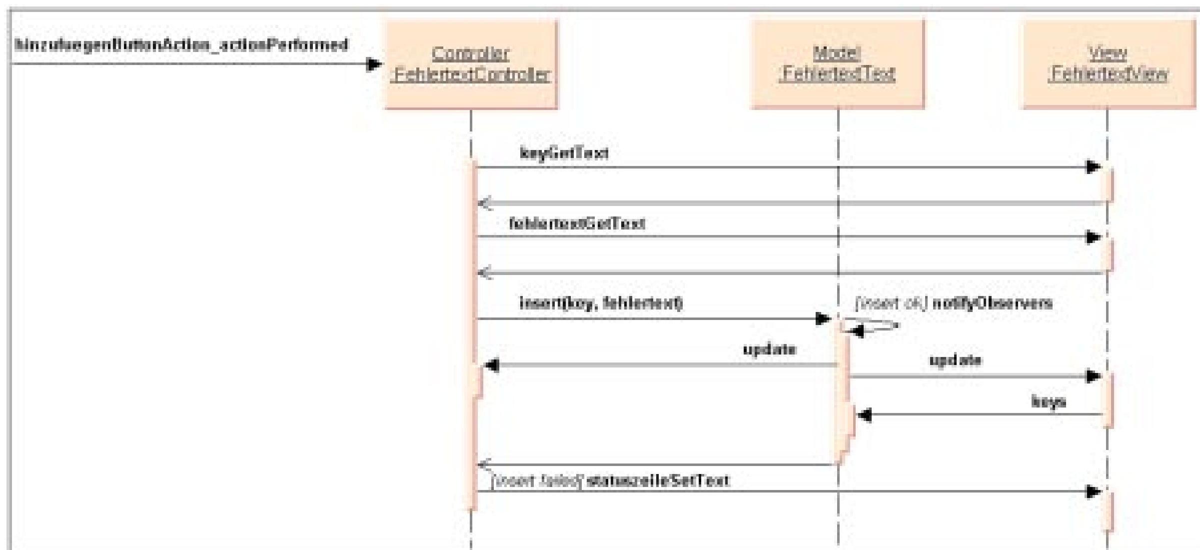


Abb. 45: Beispiel eines Eventing-Ablaufs

Wie die Erfassung eines Fehlertextes dynamisch zur Laufzeit funktioniert, zeigt das Beispiel in Abb. 45. Nachdem der Benutzer den Hinzufügen-Button gedrückt hat, wird ein Event ausgelöst, das beim *FehlertextController* die Methode *HinzufuegenButtonAction_action-Performed()* aufruft. Bevor beim Model die insert-Methode aufgerufen wird, muß sich der Controller die Parameter dieser Methode von der View besorgen (Inhalt der Textfields Key und Fehlertext). Das Model prüft daraufhin, ob das Einfügen möglich ist (der Key darf noch nicht vorhanden sein). Wenn die Prüfung erfolgreich ist, wird die von Observable geerbte Methode *notifyObservers* aufgerufen, die die im Interface *Observer* spezifizierte Methode *update* bei allen angemeldeten Observern (in diesem Fall View und Controller) aufruft. Die View aktualisiert daraufhin das Fenster und der Einfügevorgang ist damit abgeschlossen.

7.4.2 Benutzung

Die Fehlertext-Erfassung dient dazu, Files zu generieren, die die Anwenung braucht, um die Fehlertexte auszugeben. Generiert wird dabei pro Sprache ein File, das die jeweiligen Fehlertexte enthält, und zusätzlich ein weiteres File, in dem die Keys der Fehlertexte gespeichert sind (siehe Abb. 46). Die in diesem File gespeicherte Klasse ist die Oberklasse der Klassen, die in den sprachabhängigen Files enthalten sind. Die sprachabhängigen Files müssen dabei der Sprachkonvention von Javas RessourceBundle-Konzept[49] genügen. Sie haben denselben

[49] das RessourceBundle-Konzept von Java kann in allen gängigen Java 1.1 Büchern nachgelesen werden

Namen wie die Oberklasse plus einen Unterstrich mit dem international gängigen Sprachkürzel (de = deutsch, en = englisch, fr = französisch, etc.).

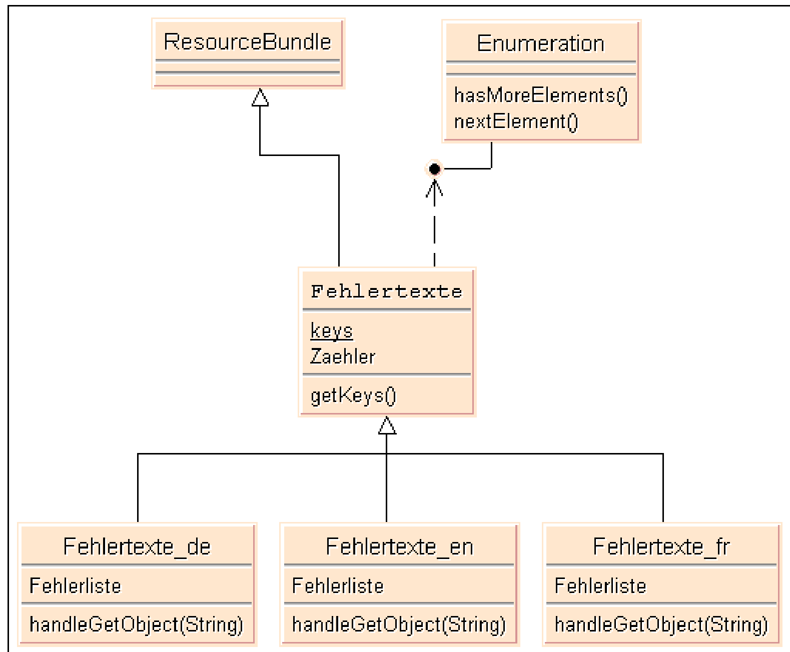


Abb. 46: Generierte Files

Gestartet wird diese Fehlertext-Erfassung durch den Aufruf der Main-Routine der Klasse *FehlertextMain*. Nachdem sich das Fenster der Fehlertext-Erfassung geöffnet hat (siehe Abb. 47), besteht die Möglichkeit, eine früher erzeugte Klasse zu laden, um Änderungen an den Fehlermeldungen vorzunehmen. Dazu muß als erstes die gewünschte Sprache aus der Combobox ausgewählt werden und anschließend der Open-Button gedrückt werden, der den Standard-Open-Dialog öffnet, in dem das entsprechende Java-File ausgewählt werden kann. Dabei ist zu beachten, daß immer die Oberklasse des generierten Codes ausgewählt wird, da die sprachabhängige Unterklasse in Abhängigkeit von der vorher getätigten Sprachauswahl automatisch dazugeladen wird. Sollte es das entsprechende Sprachenfile nicht geben, so werden die Keys ohne Fehlertexte angezeigt. So kann der Benutzer die Texte für eine neue Sprache eingeben, ohne daß er die vorhandenen Keys erneut zu erfassen braucht.

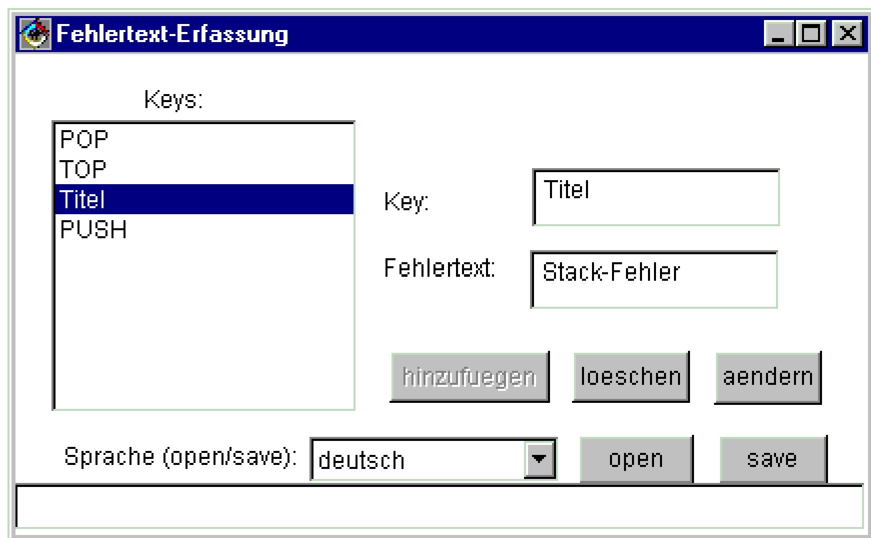


Abb. 47: Fehlertext-Erfassung

Beim Speichern sollte ebenso wie beim Öffnen darauf geachtet werden, daß zuerst die Sprache ausgewählt wird, die aktuell erfaßt wurde. Erst danach sollte der Save-Button gedrückt werden; es erscheint der Standard-Speichern-Dialog. Der Dateiname benötigt die Endung *.java*. Diese wird automatisch hinzugefügt, wenn sie nicht explizit mit eingegeben wird. In diesen Dateinamen wird der Code für die Oberklasse generiert. Der Code der sprachabhängigen Unterklasse wird in einer Datei gespeichert, die eine sprachabhängige Endung hat. Angenommen die Datei heißt *Fehlertexte.java*, so heißt die deutsche Unterklasse *Fehlertexte_de.java* und die englische *Fehlertexte_en.java*.

Damit diese Dateien wirksam in eine Applikation eingebunden werden können, müssen sie in dem passenden Verzeichnis stehen. Passend heißt dabei, daß die Hierarchie des Verzeichnisbaums mit dem Package, zu dem die Dateien hinzugefügt werden sollen, übereinstimmen muß. Wenn dies der Fall ist, können die Dateien einfach kompiliert werden. Um sie in der Applikation benutzen zu können, muß während der Initialisierung das Bundle zugeordnet werden, indem der vollqualifizierte Klassenname der generierten Fehlerklasse angegeben wird:

ResourceBundle fehlertextBundle =
ResourceBundle.getBundle("Fehlertool.Fehlertext.Fehlertexte",Locale.getDefault());

7.5 Besonderheiten bei der Implementation in Java

Im Verlauf dieses Kapitels wurde bisher der in Kapitel 2.1 aufgestellte Acht-Punkte-Katalog von Dony nicht berücksichtigt. Diese Thematik wird in Kapitel 7.5.1 betrachtet. Im Anschluß daran werden die Vor- und Nachteile der Java-Implementation des Fehlerbehandlungssystems beschrieben.

7.5.1 Java und der Acht-Punkte-Katalog von Dony

Der von Dony aufgestellte Acht-Punkte-Katalog beschreibt Anforderungen, die an ein Fehlerbehandlungssystem gestellt werden (siehe Kapitel 2.1). Die meisten dieser Anforderungen beziehen sich aber auf das Ausnahme-Behandlungs-Konzept, welches bereits von vielen Programmiersprachen angeboten wird. Da für diese Arbeit von vornherein feststand, daß das Fehlerbehandlungssystem in Java implementiert werden soll, wird im folgenden dargestellt, inwieweit Java den Acht-Punkt-Katalog erfüllt.

Der erste Punkt ist die Forderung, daß Programmierer ihre eigenen Ausnahmen definieren können sollen. Dies ist in Java möglich. Beachtet werden muß nur, daß alle Ausnahmen Unterklassen von *java.lang.Throwable* sein müssen.

Die Forderungen zwei, drei und vier besagen, daß es möglich sein soll, Bearbeiter an Ausnahmen, Klassen und Ausdrücke anzuhängen. Diese Forderungen erfüllt Java nur partiell. Generell können Bearbeiter nur in einem catch-Konstrukt nach einem try-Block stehen (Forderung vier). Es besteht aber die Möglichkeit, daß der Bearbeiter in Form einer Methode der Ausnahme oder Klasse implementiert wird. Tritt die Ausnahme ein, so kann im catch-Block diese Methode aufgerufen werden. Dieses Konzept wird vom Fehlerbehandlungssystem aber nicht benutzt. Um die Default-Implementation zu benutzen, muß die Ausnahme im catch-Block einem Exemplar der Klasse *Ausnahmebearbeiter* übergeben werden. Dadurch weiß die Ausnahme (Material) nichts über ihre Behandlung (Automat). Dies ist konform zum WAM-Konzept. Sollten Programmierer aber zu eigenen Ausnahmen spezielle Bearbeiter benötigen, so besteht immer noch die Möglichkeit, diese in einer Methode der Ausnahme zu implementieren.

Die fünfte Forderung von Dony ist, daß Ausnahmen entlang der Aufrufkette propagiert werden sollen. Dies ist exakt das Konzept, auf dem die Ausnahmebehandlung von Java basiert. Die Aufrufkette wird dabei so weit zurückverfolgt, bis ein passender Bearbeiter, ein catch-Block, der den Typ der aktuellen Ausnahme erwartet, gefunden wurde.

Die Forderung, daß die Ausnahmen hierarchisch organisierte Klassen sein sollen, ist von Java ebenfalls erfüllt. An der Spitze der Ausnahme-Hierarchie steht - wie bereits weiter oben erwähnt - *java.lang.Throwable*.

Die siebte Forderung, die an ein Fehlerbehandlungssystem gestellt wird, besagt, daß die Bearbeiter die Ausnahmehierarchie berücksichtigen sollen. Dies erfüllt Java ebenfalls, da Java eine objektorientierte Sprache ist.

Die letzte der acht Forderungen besagt, daß die Anweisungen eines Bearbeiters zur Ausnahme passen sollten. Dies liegt in der Verantwortung des jeweiligen Designers und wird von Java nicht überprüft.

Zusammenfassend läßt sich sagen, daß Java die meisten Forderungen von Dony erfüllt. Die Forderungen zwei und drei können leicht - wie oben beschrieben - simuliert werden.

7.5.2 Nachteile

Ein Nachteil der vorliegenden Implementation besteht in der Tatsache, daß die Invarianten aller Oberklassen mittels reflektiver Programmierung aufgerufen werden und daher mit Performance-Einbußen zu rechnen ist. Der Grund dafür ist, daß die Anwender des Fehlerbehandlungssystems das Vertragsmodell von Meyer nicht kennen und daher so viele automatische Prüfungen wie möglich eingebaut worden sind, damit die Programmierer, die mit dem Fehlerbehandlungssystem entwickeln, nicht die Invarianten aller Oberklassen mit der aktuellen Klasse in eine konjunktive Verknüpfungsbeziehung setzen müssen. Dieser Automatismus konnte jedoch nicht bei Vor- und Nachbedingung benutzt werden, da es in Java nicht möglich ist, explizit eine ausgewählte Zeile einer Methoden auszuführen (die der Vor- und Nachbedingung). Die einzige Alternative dazu wäre, die Methode der Oberklasse aufzurufen, um die Vor- und Nachbedingung zu testen. Dies kann von der Logik des Programmcodes her zu fatalen Fehlern und zu starken Performanceverlusten führen, da bei jedem Methodenaufruf die überschriebenen Methoden der Oberklasse ausgeführt würden. Bei tiefen Klassenhierarchien können dies sehr viele sein.

Ein Nachteil von Java ist, daß keine Makros unterstützt werden, die beispielsweise die Zeilennummer der aktuellen Zeile angeben (wie in C++). Ebensowenig gibt es einen Preprozessor, der z.B. die Zusicherungen vor dem Compilevorgang auskommentieren kann, um beim Endprodukt die Performance zu erhöhen.

7.5.3 Vorteile

Den eben aufgeführten Nachteilen von Java steht ein wesentlicher Vorteil gegenüber: Java besitzt ein einfaches Ausnahmebehandlungs-Konzept, das für die Zwecke des Fehlerbehandlungssystems gute Vorarbeit leistet. Die Oberklasse aller Ausnahmen *java.lang.Throwable* besitzt eine Methode, die den vollständigen Stacktrace ausgeben kann (*PrintStackTrace*). Aus diesem Grund ist kein spezielles Design Pattern für das Backtracing nötig. [RENZ97] erläutert einige nützliche Design Patterns für den Fall, daß eine andere Implementationssprache als Java gewählt wird.

8 Abschlußdiskussion des Fehlerbehandlungssystems und der Middleware-Systeme

Die Abschlußdiskussion, die in diesem Kapitel geführt wird, teilt sich in zwei Teile auf: Im ersten Teil wird überprüft, inwieweit das in Kapitel 7 vorgestellte Fehlerbehandlungssystem den in Kapitel 4.2 gestellten Anforderungen gerecht wird. Der zweite Teil geht darauf ein, ob die in Kapitel 4.3 aufgestellten Anforderungen an die Middleware von MOM (dargestellt in Kapitel 5) und CORBA (Thema von Kapitel 6) erfüllt werden.

8.1 Überprüfung der Anforderungen an das Fehlerbehandlungssystem

Die erste der in Kapitel 4.2 aufgestellten Anforderungen ist, daß das Fehlerbehandlungssystem in Java implementiert und in das Ausnahme-Konzept von Java integriert werden soll. Diese Anforderung ist erfüllt, da die Klasse *FehlertoolAusnahme* direkt von *java.lang.Exception* erbt (siehe 7.1).

Die zweite Anforderung ist, daß das in Kapitel 3 vorgestellte Vertragsmodell von Bertrand Meyer implementiert werden soll. Kapitel 7.3 ist zu entnehmen, daß dies mittels Delegation an die Klasse *FehlerDetektor* geschehen ist.

Die dritte Anforderung an das Fehlerbehandlungssystem ist, daß dieses dem Anwendungsprogrammierer ein Defaultverhalten vorschlagen soll, das jedoch bei Bedarf geändert werden kann. Diese Anforderung ist ebenfalls erfüllt, da der Anwendungsprogrammierer, wie in Kapitel 7.1 dargestellt, die Exemplare der Klasse *FehlertoolAusnahme* einem Exemplar der Klasse *Ausnahmebearbeiter* übergeben kann, um das Defaultverhalten der Fehlerbehandlung zu benutzen. Um bei Bedarf eine spezielle Ausnahmebearbeitung anzubieten, besteht die Möglichkeit, von der Klasse Ausnahmebearbeiter abzuleiten,. Ebenso können weitere Ausnahmenklassen hinzugefügt werden. Aus diesem Grund ist das Fehlerbehandlungssystem offen und leicht erweiterbar.

Eine weitere Anforderung besteht in der Erfüllung des Acht-Punkte-Kataloges von Dony (siehe Kapitel 2.1). Dieser Katalog bezieht sich allerdings nicht direkt auf das Fehlerbehandlungssystem, sondern vor allem auf die Möglichkeiten einer Programmiersprache, mit Ausnahmen umzugehen, und wurde deshalb bereits in Kapitel 7.5.1 geprüft. Als Ergebnis bleibt festzuhalten, daß Java die ersten sieben Forderungen des Acht-Punkte-Katalog entweder erfüllt oder zumindest auf einfache Weise simulieren kann (Forderungen zwei und drei). Die achte Forderung, die besagt, daß ein Bearbeiter zu einer Ausnahme passen sollte, fällt nicht in die Verantwortung einer Programmiersprache, sondern in die des Programmierers. Dies kann damit auch nicht von dem Fehlerbehandlungssystem gefordert werden.

Die nächste Anforderung an das Fehlerbehandlungssystem besagt, daß im Fehlerfall so viele Informationen wie möglich gesammelt werden, um dem Programmierer das Beheben der dem Fehler zugrunde liegenden Störung zu vereinfachen. Diese Anforderung wird ebenfalls vom Fehlerbehandlungssystem erfüllt. Die Klasse *Fehler* enthält die benötigten Informationen (siehe Kapitel 7.1); dies sind Fehlertext, Timestamp, ProgrammiererID, Klassenname, Versionsnummer und Ausnahmetyp. Dazu kommen noch Zeilennummer und Methodenname, wenn diese bei der Fehlerentdeckung dem Konstruktor von *Fehler* übergeben werden.

Eine weitere Anforderung besteht darin, daß die Fehlermeldungen von der Fachabteilung selber festgelegt werden sollen und die Sprache der Fehlermeldung einfach gewechselt

werden kann. Um diese Anforderung zu erfüllen, wurde eine mehrsprachige Fehlertexterfassung implementiert (siehe Kapitel 7.4).

Die letzte Anforderung an das Fehlerbehandlungssystem ist, sowohl auf das fehlerhafte Verhalten als auch auf das anormale Verhalten[50] reagieren zu können. Diese Anforderung wird jeweils durch eigene Ausnahmen (*ProgrammAusnahme* und *BenutzerAusnahme*, siehe Kapitel 7.1) umgesetzt.

Das implementierte Fehlerbehandlungssystem genügt damit den in Kapitel 4.2 aufgestellten Anforderungen.

8.2 Vergleich der beiden Middleware-Systeme

In diesem Abschnitt wird geprüft, inwieweit die beiden Middleware-Systeme MOM und CORBA den aufgestellten Anforderungskatalog erfüllen. Danach folgt die Entscheidung, welches Middleware-System für das implementierte Fehlerbehandlungssystem geeigneter ist.

8.2.1 Erfüllung des Anforderungskatalogs

In Kapitel 4.3 wurde ein Anforderungskatalog an die Middleware gestellt. In den folgenden Unterkapiteln wird Punkt für Punkt untersucht, inwieweit CORBA (respektive Visibroker) und MOM (respektive MQSeries) die Kriterien dieses Anforderungskatalogs erfüllen.

8.2.1.1 Middlewarekriterien

MOM-Produkte sind für nahezu alle Hard- und Softwareplattformen erhältlich und damit von beiden unabhängig, solange für jede benötigte Plattform ein MOM-Produkt desselben Herstellers gekauft wird. MOM-Produkte unterstützen eine ganze Reihe von Protokollen, wie z.B. TCP/IP, SNA, LU6.2, IPX und DECnet. Die APIs sind für unterschiedliche Programmiersprachen erhältlich. Problematisch ist bei einigen Produkten, daß keine Datenformat-Konvertierungen durchgeführt werden: Dies muß dann der Programmierer übernehmen. Bei MQSeries ist das jedoch nicht der Fall.

CORBA-Produkte sind ebenfalls für die meisten Hard- und Software-Plattformen erhältlich. Da die CORBA-Produkte unabhängig vom Hersteller miteinander kommunizieren können, muß nicht für jede Plattform eine CORBA-Implementation desselben Herstellers installiert werden. Das Sprachmapping von und nach IDL ist derzeit für C, C++, Smalltalk, Java, Ada und COBOL spezifiziert. Die Datenformat-Konvertierungen übernimmt der ORB.

Das Middlewarekriterium wird damit sowohl von MQSeries als auch von Visibroker erfüllt.

8.2.1.2 Unterstützung der Objektorientierung

Die Unterstützung der Objektorientierung hängt stark von den einzelnen MOM-Produkten ab. MQSeries besitzt APIs für C++ und Java, die vollständig objektorientiert sind. Andere Produkte besitzen keine entsprechenden APIs.

CORBA baut auf der objektorientierten Technologie auf und ist deswegen vollständig objektorientiert.

Dieses Kriterium wird ebenfalls von MQSeries und Visibroker erfüllt.

[50] Verhalten, das nicht normal aber spezifiziert ist (siehe Kapitel 2.3)

8.2.1.3 Kompatibilität der Produkte zu anderen Middleware-Produkten

Die APIs der MOM-Produkte sind nicht standardisiert und nicht kompatibel zueinander. Vor der Programmierung muß somit ein proprietäres MOM-Produkt ausgewählt werden. Nur in wenigen Fällen gibt es Gateways zwischen MOM-Produkten, über das produktübergreifend kommuniziert werden kann (z.B. MQSeries mit DECmessageQ). Auch die Services der einzelnen Produkte, wie Naming, Directory und Security, sind proprietär. Der Wechsel von einem MOM-Produkt zu einem anderen ist folglich mit hohem Programmieraufwand verbunden.

Dieses Problem sollte durch die Standardisierung von MOM-Produkten gelöst werden. Deshalb wurde die Message Oriented Middleware Association (MOMA), eine internationale, nicht auf Gewinn ausgelegte Vereinigung von 40 Unternehmen (u.a. IBM, BEA Systems, Software AG, Sun Microsystems), Benutzern und Consultants im Jahr 1994 gegründet. Die Zielsetzung der MOMA lautete:

> „MOMA serves as an advocate for Message Oriented Middleware (MOM) users and vendors by providing an independent information source, a forum for idea exchange, and a platform for partnering with complementary industry technology groups to further the acceptance of MOM technology.“[51]

Seit 1997 ist die MOMA an der Zusammenarbeit mit anderen Industrieorganisationen (vor allem der OMG und deren Mitgliedern) interessiert, um eine Annäherung von MOM an CORBA bzw. eine Integration von MOM in CORBA zu erreichen. Die Spezifikation des Message Service ist ein Ergebnis und wird in der CORBA Version 3.0 enthalten sein. Der Erfolg des Standardisierungsprozesses der MOM-Produkte ist bisher jedoch nicht eingetreten. Kritiker der MOMA behaupten sogar, daß die MOMA daran gar nicht mehr interessiert sei, sondern nur den Bekanntheitsgrad von MOM steigern wolle.

Die Inkompatibilität von CORBA-Produkten ist dagegen zumindest theoretisch von vornherein ausgeschlossen, weil CORBA ein Standard ist, der von all seinen Implementationen eingehalten werden muß. Wenn ein Interface in IDL spezifiziert ist, kann es sowohl vom IDL-Generator von Visibroker als auch von jedem anderen IDL-Generator übersetzt werden. Im Falle des Wechsels einer CORBA-Implementation muß nur der IDL-Code neu kompiliert und die genierten Stubs und Skeletons dazugelinkt werden. Die Kommunikation von CORBA-Objekten, die auf ORBs unterschiedlicher Hersteller laufen, ist über das IIOP-Protokoll standardisiert. Bei dieser herstellerübergreifenden Kommunikation gibt es in der Praxis allerdings noch einige Probleme, da nicht immer gewährleistet ist, daß sich die Implementationen der CORBA-Produkte vollständig an die CORBA-Spezifikation halten. Weil die OMG CORBA nur spezifiziert und nicht implementiert, gibt es einen Timelag zwischen dem Zeitpunkt der Spezifikation und der Implementation eines Herstellers. Deshalb sind beispielsweise bei Visibroker nur wenige CORBA Services implementiert worden.

Das Kompatibilitätskriterium wird von Visibroker deutlich besser erfüllt als von MQSeries.

8.2.1.4 Persistenz

Persistenz ist eine typische Eigenschaft für MOM-Produkte. Für jede Queue kann definiert werden, ob sie persistent oder transient sein soll.

---

[51] aus [MOMA98]

Die OMG hat für CORBA den Persistence Service spezifiziert, der ein standardisiertes Objektinterface zur Verfügung stellt, mit dessen Hilfe die Objekte in unterschiedliche Arten von Datenbanken (objektorientierte, relationale, etc.) oder Dateien gespeichert werden können. Visibroker hat dieses Interface nicht spezifiziert und bietet somit keinen Persistenzmechanismus an. Das Persistenzkriterium wird damit verletzt. Um dennoch eine Speicherung der Fehler zu ermöglichen, wurde in der Implementation eine Datei benutzt.

Im Gegensatz zu Visibroker erfüllt MQSeries das Persistenzkriterium.

8.2.1.5 Möglichkeit der asynchronen Kommunikation

Die Kommunikation von MOM-Produkten ist generell asynchron, d.h. der Sender einer Message muß nicht warten, bis der Empfänger die Verarbeitung beendet hat, sondern kann mit der eigenen Verarbeitung fortfahren. Es besteht zusätzlich die Möglichkeit, synchrone Aufrufe zu simulieren. Außerdem ist die Kommunikation zeitunabhängig und verbindungslos, d.h. der Empfänger der Message muß zum Zeitpunkt des Sendens der Message nicht aktiv sein. Er kann zu einem späteren Zeitpunkt die Messages empfangen und verarbeiten. Dies ist vor allem wichtig für Außendienstmitarbeiter-Software, die auf Laptops installiert ist. Tritt bei dieser Software ein Fehler auf, so wird er zunächst in einer lokalen Queue gespeichert. Wenn der Außendienstmitarbeiter eine Verbindung zur Zentrale aufbaut, werden alle Fehlermessages von der lokalen Queue zur zentralen Fehlerqueue gesendet.

CORBA-Aufrufe sind im allgemeinen synchron, d.h. der Aufrufer ist solange geblockt, bis der Aufgerufene seine Verarbeitung beendet hat. Dies kann für die Performance kritisch werden, weil die Aufrufe Prozeßgrenzen durchschreiten und somit von dem Netzwerk abhängig sind. Dies merkt man beispielsweise bei DistributedSmalltalk. Das DII bietet aber zusätzlich zwei ungeblockte Aufrufe an: Der eine Aufruf funktioniert nur bei Methoden, die keinen Rückgabewert zurückliefern (*send_oneway*), der andere gibt die Kontrolle sofort an das aufrufende Objekt zurück (*send_deferred*). In diesem Fall muß das Ergebnis gepollt werden (*poll_response*). CORBA-Aufrufe können nur getätigt werden, wenn eine Verbindung zu dem entfernten Server-Objekt besteht. Der Rechner, auf dem sich das Server-Objekt befindet, muß zum Aufrufzeitpunkt adressierbar sein. Dies ist bei einem Laptop nicht der Fall.

Visibroker erfüllt dies Kriterium im Gegensatz zu MQSeries nicht.

8.2.1.6 Erlernbarkeit

MOM-Produkte zu erlernen ist relativ einfach, weil die API klein und einfach zu benutzen ist. Das Konzept von Messaging und Queuing ist aus dem Alltag bekannt (Anrufbeantworter, Fax, E-Mail etc.). Der Administrator hat dafür höheren Aufwand, da er die Queues und andere Komponenten definieren muß. MOM-Produkte werden überwiegend in der traditionellen Programmierung eingesetzt, so daß kein neues Programmierkonzept erlernt werden muß.

Das Erlernen von CORBA ist deutlich schwieriger, da CORBA erheblich komplexer ist als MOM: Es muß eine neue Spezifikationssprache (IDL) erlernt werden und die CORBA-Klassenbibliothek muß zumindest teilweise verstanden werden. CORBA beschränkt sich auf das objektorientierte Umfeld.

8.2.1.7 Performance

Die Performance hängt jeweils stark von einem konkreten Produkt ab. Es ist schwer, hier einen allgemeinen Vergleich durchzuführen. Generell kann festgestellt werden, daß die Performance im LAN ähnlich gut ist, aber im WAN die MOM-Produkte besser abschneiden.

Im Rahmen dieser Diplomarbeit konnte kein Performance-Vergleich durchgeführt werden, weil MQSeries nur in der HM und Visibroker nur in der Universität zur Verfügung stand.

8.2.1.8 Sicherheit

Das Sicherheitskonzept ist bei den meisten MOM-Produkten gut ausgereift. CORBA hat einen Security Service spezifiziert, der auf SSL (Secure Socket Layer) aufbaut. In IDL kann für jede Methode ein Kontext definiert werden, der Informationen für die Laufzeitumgebung enthält. Diese Information kann für Sicherheitsabfragen benutzt werden.

8.2.1.9 Praxiseinsatz

Messaging und Queuing sind alt bewährte und weit verbreitete Techniken für Inter-Programm-Kommunikation in der traditionellen Softwareentwicklung. MOM-Produkte werden oft im Bankenbereich (Barclays Bank, Swedish Nordbanken, Finnish Banking Group, OKOBank-Group), aber auch im Telekommunikationsbereich, sowie in Unternehmen wie Software AG und Sybase Inc. eingesetzt.

Der Einsatz von CORBA explodiert seit einiger Zeit förmlich. In den unterschiedlichsten Branchen und Ländern wird CORBA eingesetzt: Department of Defense (DOD), US Navy, Boing, Dresdner Bank, dpa, Financial Times, Franc Telecom und Hongkong Telecom.

8.2.1.10 Verfügbarkeit in der HM

In der HM wurde für die Implementation des Fehlerbehandlungssystems MQSeries benutzt, da keine CORBA-Implementation zur Verfügung stand.

8.2.2 Entscheidung zugunsten eines der Middleware-Systeme

Für das Fehlerbehandlungssystem ist MQSeries als Middleware besser geeignet als Visibroker. Auch wenn es sich bei CORBA um eine offene Architektur handelt, die sich als Standard durchzusetzen scheint und Visibroker deshalb mit anderen CORBA-Produkten kompatibel ist, fehlt sowohl die Persistenz als auch die Möglichkeit, asynchron verbindungslos zu kommunizieren. Außer diesen beiden gravierenden Nachteilen kommt noch die schwerere Erlernbarkeit hinzu.

9 Zusammenfassung und Ausblick

Die vorliegende Diplomarbeit hat in die Terminologie der Fehlerbehandlung eingeführt. Als wesentliche Begriffe haben sich Fehler, Störung, Fehlverhalten, Detektor und Ausnahme herauskristallisiert. Es wurde verdeutlicht, daß sich jede Komponente entweder im normalen oder im fehlerhaften Zustand befindet. Der fehlerhafte Zustand sollte durch einen Ausnahmebearbeiter in den normalen Zustand überführt werden. Scheitert dieser Versuch, so hat die Komponente dafür zu sorgen, daß der fehlerhafte Zustand in einen konsistenten Zustand überführt wird, um spätere Anfragen ordnungsgemäß abarbeiten zu können. Dem Aufrufer ist mittels einer Ausnahme anzuzeigen, daß seine Anfrage nicht erfolgreich bearbeitet werden konnte.

Um einen fehlerhaften Zustand erkennen zu können, werden Detektoren benötigt. Von der Qualität und der Anzahl der Detektoren hängt es ab, wie frühzeitig Fehler erkannt werden. Das Vertragsmodell von Bertrand Meyer ist eine Möglichkeit, den Programmierer bei der Erstellung von korrekter Software zu unterstützen. Dabei wird zwischen der Klasse und ihrem Aufrufer ein Vertrag geschlossen, in dem Rechte und Pflichten der Beteiligten spezifiziert werden. Diese werden in Form von Vor- und Nachbedingung sowie Invarianten festgelegt.

Bevor das Fehlerbehandlungssystem implementiert werden kann, muß festgelegt werden, welchen Anforderungen es genügen soll. Da es sich um ein verteiltes System handelt, wird eine adäquate Middleware benötigt, die ebenfalls bestimmte Anforderungen erfüllen muß. Als Middleware wurden MQSeries, ein Produkt der MOM-Familie, und Visibroker, eine CORBA-Implementation, untersucht.

Mit Hilfe von MOM-Produkten können Anwendungsprogramme asynchron Messages verschicken. Die Messages können persistent oder transient sein. Messages werden in Queues gespeichert. Anwendungsprogramme können über das MQI auf die Queues zugreifen und Messages lesen oder speichern. Der Queue-Manager ist der zugrunde liegende Systemservice, der für den korrekten rechnerübergreifenden Transport sorgt.

Im Gegensatz zu MOM bietet CORBA einen transparenten Raum, in dem synchron auf alle CORBA-Objekte zugegriffen werden kann, unabhängig davon, auf welchem Rechner sie aktiv sind. Um Asynchronität und Persistenz zu erhalten, müssen der Event- und der Persistence-Service benutzt werden. Diese sind aber nicht zwangsläufig bei am Markt erhältlichen CORBA-Implementationen implementiert. Dadurch entsteht das Problem, daß zwar eine große Funktionalität spezifiziert, aber nicht implementiert ist. Dies trifft auch auf Visibroker zu, bei dem der Persistence-Service nicht implementiert wurde.

Mit Hilfe einiger Klassendiagramme wurde die der Diplomarbeit zugrunde liegende Implementation des Fehlerbehandlungssystem erläutert.

Den Abschluß der Arbeit stellt die Überprüfung der Anforderungen an das Fehlerbehandlungssystem dar. Zusätzlich wurde geprüft, inwieweit die beiden betrachteten Middleware-Systeme den aufgestellten Anforderungen gerecht werden. Darauf aufbauend wurde die Entscheidung getroffen, daß für das Fehlerbehandlungssystem MQSeries als Vertreter der MOM-Fraktion geeigneter ist als die CORBA-Implementation Visibroker.

Ein Grund für diese Entscheidung ist, daß der Persistence-Service bei Visibroker nicht implementiert wurde. Um diesen Nachteil zu beseitigen, könnte dieser ergänzend zu der

vorliegenden Arbeit implementiert werden. Es bleibt zu hoffen, daß zukünftige Versionen von Visibroker die bisher fehlenden CORBA-Services anbieten werden.

Im Rahmen dieser Diplomarbeit war es nicht möglich, weitere CORBA-Implementationen und andere MOM-Produkte zu betrachten. Auf diesem Gebiet bieten sich weitere Untersuchungen an.

Über den allgemeinen Vergleich von MOM und CORBA sagt die FROST (Front Range Object STudy Group):

> „Common observation about **CORBA**: This is the right architectural and programming abstraction, but current implementations don't provide the robust, scalable behavior I need for high-performance critical systems. Common observation about **MOM**: Messaging middleware gives me the performance and reliability I need, but the APIs are too low-level, it's not object-oriented, and there are no standards."[52]

Die Entwicklung geht derzeit in die Richtung, daß einerseits CORBA robuster und skalierbarer wird und andererseits die MOM-Hersteller objektorientierte Schnittstellen für ihre Produkte entwickeln.

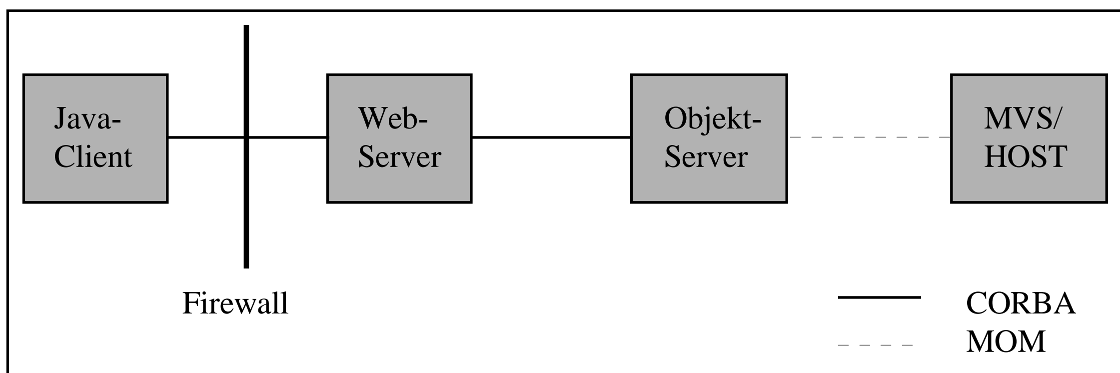


Abb. 48: Einsatz von CORBA und MOM

Der entscheidende Vorteil von MOM ist, daß es sich um eine herkömmliche Middleware handelt, die auf Hostsystemen weit verbreitet ist. Da auf dem Host keine Objekte existieren, scheint es zu aufwendig, die vorhandenen Programme mit Objektwrappern zu umgeben, um CORBA einsetzen zu können. Die Kommunikation sollte deshalb mit MOM-Produkten erfolgen. CORBA dagegen sollte eingesetzt werden, wenn objektorientiert entwickelt wird. Abb. 48 zeigt, daß derzeit eine Kombination aus beiden denkbar scheint: Ein Java-Client kann über einen Webserver mit einem Objektserver per CORBA kommunizieren. Dieser greift über MOM auf einen Host zu.

Der Markt der Middleware-Produkte wird sich in Zukunft weiter ändern, und es bleibt abzuwarten, ob der CORBA-Trend anhält oder durch neuere Technologien abgelöst wird.

Leider war es im Rahmen dieser Diplomarbeit nicht möglich, vorhandene Fehlerbehandlungssysteme zu untersuchen. Ein Beispiel für ein solches System ist NuMega DevPartner for Java (siehe [NUME98]). Hierbei handelt es sich um ein Tool zur automatischen Entdeckung von Softwarefehlern. Ein weiteres Beispiel ist das Produkt Purify von Rational (siehe [RATI98]).

[52] aus [FROS98]

10 Anhang

Um einen kleinen Einblick in die MQSeries Definitionssprache zu erhalten, ist in diesem Anhang der Sourcecode für das Fehlerbehandlungssystem sowie eine MQSeries-Mapping-Tabelle enthalten. Im Anschluß daran folgt die Spezifikation des Stacks.

10.1 MQSeries-Code für das Fehlerbehandlungssystem

Der MQSeries-Code für die Clientseite der in Kapitel 5.5 beschriebenen Architektur ist Abb. 49, der Code für die Serverseite Abb. 50 zu entnehmen.

```
* Definition der lokalen Transmission-Queue
DEFINE QLOCAL(CLIENT.XMIT’) +
       DESCR(’Lokale Transmission-Queue’) +
       USAGE(XMITQ)

* Definition der Remote-Queue
DEFINE QREMOTE(’CLIENT.REMOTE’) +
       DESCR(’Remote-Queue definiert beim Client’) +
* Die Queue wird als persistente Queue deklariert
       DEFPSIST(YES) +
* Dies ist der Name der lokalen Queue vom Server
       RNAME(’FEHLER.LOCAL’) +
* Dies ist der Name des Queue-Managers der entfernten Maschine
       RQMNAME(’SERVER’) +
* Dies ist der Name der lokalen Transmission-Queue
       XMITQ(’CLIENT.XMIT’)

* Definition des Channels, der die Messages von der
* Transmission-Queue CLIENT.XMIT zum Server sendet
DEFINE CHANNEL (’CLIENT.TO.SERVER’) +
* Channeltyp ist Sender
       CHLTYPE(SDR) +
* Transportprotokoll ist TCP/IP
       TRPTYPE(TCP) +
* Transmission-Queue, von der die Messages versendet werden
       XMITQ(’CLIENT.XMIT’) +
* TCP/IP-Adresse des Empfängers
       CONNAME('SERVER TCP/IP Maschinenname') +
        DESCR('Sender channel fuer Messages zum Queue-Manager
               Server')
```

Abb. 49: MQ-Komponenten für die Clientseite

```
* Definition des Channels, der die Messages vom Client Queue-
* Manager annimmt.
DEFINE CHANNEL (’CLIENT.TO.SERVER’) +
* Channeltyp Receiver
       CHLTYPE(RCVR) +
       TRPTYPE(TCP) +
       DESCR(’Receiver-Channel fuer Messages vom Queue-Manager
              CLIENT’)

* Definition der lokalen Queue, in die Messages gespeichert
* werden sollen
DEFINE QLOCAL(’FEHLER.LOCAL’) +
       DESCR(’Local-Queue’) +
       DEFPSIST(YES) +
       SHARE
```

Abb. 50: MQ-Komponenten für die Serverseite

10.2 MQSeries-Mapping-Tabelle

Beim Open-Call einer Queue wird der Queue-Manager-Name und der Queue-Name wie folgt aufgelöst:

Eingabe in MQOD		**Aufgelöste Namen**		
ObjectQMgrName	**ObjectName**	**ObjectQMgrName**	**ObjectName**	**Transmission queue**
blank oder lokaler Queue-Manager	Lokale Queue	lokaler Queue-Manager	ObjectName	blank
blank oder lokaler Queue-Manager	Lokale Definition einer remote Queue	RemoteQMgrName des remote Queue Definition Objects	RemoteQName des Remote Definition Objects	XmitQName Attribut
Name der lokalen TransmissionQueue	wird nicht aufgelöst	ObjectQMgrName	ObjectName	ObjectQMgrName
Queue-Manager ist kein Name eines lokalen Objekts	wird nicht aufgelöst	RemoteQMgrName	ObjectName	DefXmitQName Attribut des Queue Managers

10.3 Stack-Spezifikation

Methode *pop()*:
    Vorbedingung: *!isEmpty()*
    Nachbedingung: *!isFull()*

Methode *push(String)*:
    Vorbedingung: *!isFull()*
    Nachbedingung: *!isEmpty()*

Methode *top()*:
    Vorbedingung: *!isEmpty()*
    Nachbedingung: *!isEmtpyl()*

Invariante:
    (0 <= liste.size()) && (liste.size() <= maxElements)

Abkürzungsverzeichnis

ANSI	American National Standards Instutitute
API	Application Programming Interface
BOA	Basic Object Adaptor
CORBA	Common Object Request Broker Architecture
CORBAMB	CORBAMessageBroker
CORBAMBSI	CORBAMessageBrokerServerImplementation
DAD	Distributed Application Development
D.A.S.	Deutsche Automobil-Schutz (Versicherung)
DBMS	Database Management System
DCE	Distributed Computing Environment
DII	Dynamic Invocation Interface
DKV	Deutsche Krankenversicherung
DOD	Department of Defense
DSI	Dynamic Skeleton Interface
DST	Distributed-Smalltalk
FIFO	First In First Out
FROST	Front Range Object Study Group
GIOP	General Inter-ORB-Protokoll
GUI	Graphical User Interface
HM	Hamburg-Mannheimer Versicherungs-AG
IDL	Interface Definition Language
IEEE	Institute of Electrical and Electronic Engineers
IIOP	Internet Inter-ORB-Protokoll
IOR	Interoperable Objektreferenz
JCL	Job Control Language
JDK	Java Developers Kit
JWAM	Java-WAM-Framework
KIS	Krankenhausinformationssystem
LAN	Local Area Network
MBSI	MessageBrokerServerImplementation
MCA	Message Channel Agent
MCP	Message Channel Protocol
MOM	Message-oriented Middleware
MOMA	Message-oriented Middleware Assoziation
MQI	Message Queuing Interface
MQOD	MQSeries Object Description
MVC	Model-View-Controller
MVS	Multiple Virtual Storage
ODBMS	Object-oriented Database Management System
OMA	Object Management Architecture
OMG	Object Management Group
OO	Objektorientierung
OODB	Object-oriented Database
OQL	Object Query Language
ORB	Object Request Broker
PKN	Persönliche Kennummer
PL/1	Programming Language 1
POA	Portable Object Adaptor

RDBMS	Relational Database Management System
RMI	Remote Method Invocation
RMIMB	RMIMessageBroker
RMIMBSI	RMIMessageBrokerServerImplementation
RPC	Remote Procedure Call
SNA	System Network Architecture
SQL	Structured Query Language
SSL	Secure Socket Layer
TAMCO	ToolAutomatonMessageCoordinator
TCP/IP	Transmission Control Protocol / Internet Protocol
TOPAS	Top Anwendungslandschaft
TP	Transaction Processing
TPM	Transaction Processing Monitor
UML	Unified Modelling Language
URL	Uniform Resource Locator
UUID	Universal Unique Identifier
WAM	Werkzeug-Aspekt/Automat-Material-Metapher
WAN	Wide Area Network

Abbildungsverzeichnis

Abb. 1: Klassifikation vom Systemverhalten....................4
Abb. 2: Zusammenhang der Definitionen....................6
Abb. 3: Kontrollfluß von Methoden, die Ausnahmen (Exc) auslösen....................8
Abb. 4: Verhalten einer Methode aus Laufzeitsicht....................9
Abb. 5: Verhalten einer Methode aus Spezifikationssicht....................9
Abb. 6: Ideale fehlertolerante Komponente....................10
Abb. 7: Die Klasse Stack....................13
Abb. 8: Propagierung von Zusicherungen und Ausnahmen....................15
Abb. 9: Kontoauskunft (TP 120)....................18
Abb. 10: Prototyp....................19
Abb. 11: Systemarchitektur des Prototyps....................20
Abb. 12: Messageaustausch mit inaktivem Empfänger....................24
Abb. 13: 1:n-Beziehung von Anwendungen....................24
Abb. 14: n:1-Beziehung von Anwendungen....................24
Abb. 15: Aufbau einer Message....................25
Abb. 16: Programme kommunizieren über das MQI mit dem Queue-Manager....................27
Abb. 17: Queue-Typen....................30
Abb. 18: MQI- und Message-Channels....................30
Abb. 19: Beispiel für Channel-Gruppen....................31
Abb. 20: Message Routing....................34
Abb. 21: MQ-Komponenten der Fat-Client Variante des Fehlerbehandlungssystems....................37
Abb. 22: IDL Sprach-Binding....................39
Abb. 23: Syntax eines IDL-Moduls....................40
Abb. 24: Datentypen von IDL....................40
Abb. 25: IDL-Modul Fehlerbehandlungssystem....................41
Abb. 26: Der ORB als Middleware....................41
Abb. 27: Statische CORBA-Komponenten....................42
Abb. 28: Sprachunabhängige Objektreferenzen....................45
Abb. 29: CORBA-Komponenten....................45
Abb. 30: Objekt Management Architektur....................47
Abb. 31: OMA-Schichtenmodell....................48
Abb. 32: Internetkommunikation mit Visibroker....................49
Abb. 33: Push- und Pull-Modell des CORBA Event-Service....................50
Abb. 34: Lösung mit Event-Channeln....................51
Abb. 35: Architektur des Messagebrokers....................52
Abb. 36: Architektur - vom Client zum Server....................53
Abb. 37: Architektur - vom Server zum Client....................53
Abb. 38: IDL-Modul....................54
Abb. 39: Klassendiagramm Fehlertool-Ausnahmen....................56
Abb. 40: Klassendiagramm Persistenz-Mechanismus....................57
Abb. 41: Klassendiagramm FehlerDetektor....................58
Abb. 42: Pop-Methode des Stacks mit Vertragsmodell....................59
Abb. 43: Aufbau der Fehlertext-Anwendung gemäß MVC....................60
Abb. 44: Klassendiagramm des Fehlertext-Packages....................61
Abb. 45: Beispiel eines Eventing-Ablaufs....................62
Abb. 46: Generierte Files....................63
Abb. 47: Fehlertext-Erfassung....................63
Abb. 48: Einsatz von CORBA und MOM....................72

Abb. 49: MQ-Komponenten für die Clientseite....73
Abb. 50: MQ-Komponenten für die Serverseite....73

Literaturverzeichnis

[ANDE81] Anderson, T.; Lee, P. A.: Fault Tolerance Principles and Practice, London et. al., 1981

[ANDE81b] Anderson, T. ; Knight, J. C.: Practical Software Fault Tolerance for Real-Time Systems, University of Newcastle, 1981

[ANDE96] Anderson, Bruce: Null Object, EuroPLoP Proceedings aus JavaSpektrum Ausgabe 5, S. 22, Sept/Okt 1996

[AVRE92] Avresky, Dimitri R.: Hardware and Software Fault Tolerance in parallel computing systems, Chichester, 1992

[BLAK95] Blakely, Burnie; Harris, Harry; Lewis, Rhys: Messaging & Queuing Using the MQI, USA, 1995

[BRAS97] Brasch, Frank; Carey, George T.; Colyer, Adrian; JaeWook, Jin; Wackerow, Dieter: Internet Application Development with MQSeries and Java (IBM), USA, 1997

[DENE91] Denert, Ernst: Software-Engineering: Methodische Projektabwicklung, Berlin, 1991

[DONY88] Dony, Christophe: An Object-oriented Exception Handling System for an Object-oriented Language, ECOOP Proceedings, S. 146 - 161, 1988

[DONY90] Dony, Christophe: Exception Handling and Object-Oriented Programming: towards a syntheses, ECOOP/OOPSLA Proceedings, S. 322 - 330, 1990

[FEDE90] Feder, Christiane: Ausnahmebehandlung in objektorientierten Programmiersprachen, Berlin, Heidelberg, 1990

[FELT97] Felten, Andreas; Langmann, Christian: Entwicklung einer CORBA-basierten verteilten Anwendung mit Distributed Smalltalk am Beispiel eines Pausenplaners, Studienarbeit am Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, 1997

[FISC97] Fischer, Peter: Meeting Business Challenges with Middleware, in OBJEKT Magazine, Oktober 1997

[FLAN96] Flanagan, David: Java in a nutshell, Bonn, 1996

[FRIC96] Fricke, Nils: Das Projekttagebuch - Ein Fallbeispiel für eine verteilte Anwendung nach WAM, Studienarbeit am Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, 1996

[FROS98] FROST: URL: http://www.vsys.com/FROST/9-30/mom/tsld002.htm, 1998

[GAMM95] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John: Design Patterns, 5. Auflage, New York, et al., 1995

[GOOD75] Goodenough, J.B.: Exception Handling: Issues and a Proposed Notation, Communications of the ACM, vol. 18, no. 12, S.636-696, Dez. 1975

[GRAH97] Grahl, T.; Knoll, P.: OO Vorgehensmodell der Hamburg-Mannheimer Versicherungs AG, Version 1.1, Hamburg, 1997

[GRAH97b] Grahl, T., et.al.: Java-Forschungsprojekt - Entwicklung eines Prototypens mit der Programmiersprache Java und Darstellung der Erkenntnisse

[IBM93] IBM: MQSeries – Concepts and Architecture, 1993

[IBM94] IBM: MQSeries – An Introduction to Messaging and Queuing, Second Edition, 1994

[IBM95] IBM: MQSeries Application Programming Reference, Version 2 Release 2, Second Edition, 1995

[IBM96] IBM: MQSeries Application Programming Guide, 1996

[IBM97] IBM: Internet Application Development with MQSeries and Java, 1997

[IBM98] IBM, URL: http://www.software.ibm.com/ts/mqseries/library/independent/mq-mom, 1998

[ISSA93] Valérie Issarny: An exception-handling mechanism for parallel object-oriented programming: Towards reusable, robust distributed software; aus Journal of Object-Oriented Programming, Vol. 6 No. 6, S. 29 - 40, Oktober 1993

[JACO92] Jacobson, Ivar; Christerson, Magnus; Jonsson, Patrik; Övergaard, Gunnar: Object-Oriented Software Engineering, USA, 1992

[JWAM98] Dokumentation des JWAM-Frameworks, URL: http://swt-www.informatik.uni-hamburg.de/Software/JWAMV1.0, 1998

[KILB94] Kilberth, Klaus; Gryczan, Guido; Züllighoven, Heinz: Objektorientierte Anwendungsentwicklung, 2. Verbesserte Auflage, Braunschweig/Wiesbaden 1994

[KNUD87] Knudsen, J.L.: Better Exception-Handling in Block-Structured Systems, IEEE Software, vol. 17, no.2, S. 40-49, Mai 1987

[KOEN90] Koenig, Andrew: An Exceptional Ideal, JOOP, Vol. 3, No.2, S. 52 - 59, Jul/Aug 1990

[LAME94] Lamersdorf, Winfried: Datenbanken in verteilten Systemen, Braunschweig/Wiesbaden, 1994

[LACO91] Lacourte, Serge: Exceptions in Guide, an Object-Oriented Language for Distributed Applications, ECOOP´91 Proceedings, S. 268 - 287

[MCKI96] McKim Jr., James C.; Programming by contract: Designing for correctness, aus JOOP Mai 96 Vol. 9, No. 2, S. 70 - 74

[MEYE88] Meyer, Bertrand: Object-oriented Software Construction, Hemel Hempstead,1988

[MEYE90] Meyer, Bertrand: Objektorientierte Softwareentwicklung, Übersetzung von [MEYE88], London, 1990

[MEYE92] Meyer, Bertrand: Applying „Design by Contract“ aus Computer (10), S. 40 - 51, 1992

[MEYE92a] Meyer, Bertrand: Design by Contract, aus Mandrioli, Dino; Meyer, Bertrand: Advances in Object-Oriented Software Engineering, Cambridge, 1992

[MEYE94] Meyer Bertrand: Reusable Software the base object-oriented component libraries, Cambridge, 1994

[MEYE97] Meyer, Bertrand: Object-oriented Software Construction, Second Edition, Santa Barbara, 1997

[MIDD98] URL: http://www.middlewarespectra.com, 1998

[MILL97] Miller, Robert; Tripathi, Anand: Issues with Exception Handling in Object-Oriented Systems, ECOOP´97 Proceedings, 1997

[MOMA98] Message Oriented Middleware Association Homepage, URL: http://www.moma-inc.org, 1998

[NEUM95] Neumann, Peter G.: Computer-Related Risks, 1995

[NUME98] URL: http://www.numega.com, 1998

[OMAG92] Object Management Architecture Guide: OMG Document 92.11.1 Revision 2.0, 1992

[ORFA96] Orfali, Robert; Harkey, Dan; Edwards, Jeri: The Essential Distributed Objects Survival Guide, New York, et.al., 1996

[ORFA97a] Orfali, Robert; Harkey, Dan; Edwards, Jeri: Instant CORBA, New York, et. al., 1997

[ORFA97b] Orfali, Robert; Harkey, Dan; Edwards, Jeri: Client/Server Programming with Java and CORBA, New York, et. al., 1997

[ORFA97c] Orfali, Robert; Harkey, Dan; Edwards, Jeri: The Essential Client/Server Survival Guide, New York, et.al., 1997

[PAYN98] Payne, Jeffery E.; Schatz, Michael A.; Schmid, Matthew N.: Implementing Assertions in Java, aus Dr. Dobb´s Journal, Jan. 1998, URL: http://www.ddj.com/ddj/1998/1998_01/payn/payn.htm

[RATI98] URL: http://www.rational.com/products/purify, 1998

[RENZ97] Renzel, Klaus: Error Handling for Business Information Systems, Version 1.0 (ARCUS sd&m), München, 1997

[ROOC98] Roock, Stefan; Wolf, Henning: Die Raummetapher zur Entwicklung kooperationsunterstützender Softwaresysteme für Organisationen, Diplomarbeit am Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, 1998

[RUMB93] Rumbaugh, J.; Blaha, M.; Premerlani, W.; Eddy, F.; Lorensen, W.: Objektorientiertes Modellieren und Entwerfen, London, 1993

[RUMB94] Rumbaugh, James: Eine Betrachtung der Architektur Model-View-Contoller (MVC) aus OBJEKTSpektrum Nr. 3, S. 49-54, Jul/Aug 1994

[STAL97] Stal, Michael: World Wide CORBA - verteilte Objekte im Netz, aus OBJEKTSpektrum Nr.6, S. 28-34, Nov/Dez 1997

[TECH98] URL: http://techweb.cmp.com/nc/808/808wellcon18.html, 1998

[VANH97] Vanhelsuwé, Laurence: JavaBeans, Alameda, 1997

[VINE97] Visigenic: Visibroker Naming and Event Services Programmer´s Guide Version 3.0, 1997

[VISI97] Visigenic: Visibroker for Java Programmer´s Guide Version 3.0, 1997

[ZÜLL98] Züllighoven, Heinz: Das objektorientierte Konstruktionshandbuch nach dem Werkzeug- & Material-Ansatz, 1998