Technische Universität Darmstadt

Institut für Numerische Methoden und Informatik im Bauwesen

Vertieferarbeit

„Erstellung eines CORBA-basierten Agentensystems für dynamische Informations-Recherchen im Internet“

Vorgelegt von:

Steffen Moldaner
Matr. Nr. 722058

Darmstadt, im Oktober 2000

1 Einleitung 1-1
2 Bestehende Technologien 2-1
2.1 Softwareagenten 2-1
2.1.1 Klassifikation von Softwareagenten 2-1
2.1.1.1 Klassifikation nach Eigenschaften 2-1
2.1.1.2 Klassifikation nach Anwendung 2-3
2.1.2 Vorteile mobiler Softwareagenten 2-3
2.1.3 Anwendungsgebiete 2-4
2.1.4 Nachteile mobiler Softwareagenten 2-5
2.1.5 Agentensysteme 2-6
2.1.6 Standardisierung 2-6
2.2 JAVA 2-7
2.2.1 Entwicklungsgeschichte von JAVA 2-7
2.2.2 Was ist JAVA? 2-7
2.2.3 Java und mobile Softwareagenten 2-8
2.2.4 Java Database Connectivity (JDBC) 2-8
2.2.4.1 Aufbau und Struktur 2-8
2.2.4.2 Verbindungsaufbau 2-9
2.3 XML 2-11
2.3.1 Was bietet XML gegenüber HTML? 2-11
2.3.2 Das XML-Dokument 2-13
2.3.2.1 Zeichenkonventionen 2-13
2.3.2.2 Elemente 2-13
2.3.2.3 Attribute 2-14
2.3.2.4 Kommentare 2-14
2.3.2.5 XML-Deklaration 2-14
2.3.2.6 Dokumententyp 2-15
2.3.2.7 Wurzel-Element / Dokument-Element 2-15
2.3.3 Die logische Struktur (DTD) 2-15
2.3.3.1 Deklaration der Elemente 2-16
2.3.3.2 Die Model group 2-16
2.3.3.3 Deklaration der Attribute 2-19
2.3.4 Parser 2-20
2.3.4.1 SAX 2-22
2.3.4.2 DOM 2-23
2.4 CORBA 2-26
2.4.1 Was ist CORBA? 2-26
2.4.2 Object Request Broker (ORB) 2-26
2.4.3 Interface Definition Language (IDL) 2-27
2.4.4 Weitere Dienste 2-28
2.5 Voyager 2-29
2.5.1 Die Voyager-Laufzeitumgebung 2-29
2.5.2 Erzeugen von Objekten 2-30
2.5.3 Bewegen von Objekten 2-30
2.5.4 Der erweiterte Nachrichtendienst 2-33
2.5.4.1 Dynamischer Nachrichtenaufruf 2-33
2.5.4.2 Vielfachsendungen und Veröffentlichen / Abonnieren 2-35
2.5.5 Der Namensdienst 2-40
3 Konzeption des Agentenprojekts 3-1
3.1 Generelles Konzept 3-2
3.2 Die Kommunikation 3-2

3.2.1 Das Interface INewsEvent 3-4
3.3 Der Datenaustausch 3-4
3.3.1 Übergabe der Suchkriterien 3-4
3.3.2 Übergabe der Ergebnisse 3-6
3.4 Der Agent 3-7
3.4.1 Bereitstellen des Agentendienstes 3-7
3.4.2 Die Reise 3-9
3.4.3 Darstellung der Ergebnisse 3-12
3.5 Der Server für ankommende Agenten 3-14
3.5.1 Aufbau der Datenbank 3-14
3.5.2 Das grafische Interface 3-15
3.5.3 Start des Agentenservers 3-17
3.5.4 Abfrage der Datenbank 3-18
3.6 Klassifikation des Agentensystems 3-19
3.6.1 Agent 3-19
3.6.2 Agentenserver 3-20
4 Anwendungsbeispiel 4-1
4.1 Start der Agentenserver 4-1
4.2 Start des Ressourcenservers 4-2
4.3 Start des AgentCorbaServer 4-3
4.4 Start des Agenten 4-4
4.5 Ankunft beim Server 4-7
4.6 Die Heimreise 4-8
5 Zusammenfassung und Ausblick 5-1
6 Anhang 6-1
6.1 Installation 6-1
6.1.1 Systemvoraussetzungen 6-1
6.2 Softwarekomponenten 6-2
6.2.1 JDK 1.2 6-2
6.2.2 Voyager 6-2
6.2.3 XML for JAVA 6-2
6.2.4 JDBC 6-2
6.2.5 IIB Agent 6-2
6.2.6 IIB Agentenserver 6-2
6.2.7 Der Bauteileditor (Beispielanwendung) 6-2
6.2.8 SQL – Server 6-2
6.2.9 ODBC-Schnittstelle zu Access 6-3
6.3 Kompilieren des Quellcodes 6-4
6.3.1 iibAgent 6-4
6.3.2 iibAgentServer 6-4
6.3.3 Bauteileditor 6-4
6.4 Literaturverzeichnis 6-5
6.5 Abbildungsverzeichnis 6-6

1 Einleitung

Im Bauwesen durchläuft die Erstellung eines Bauobjektes mehrere Planungsphasen: Entwurf durch einen Architekten, Konstruktion durch einen Statiker, Ausführung durch den Bauleiter, usw. Jede Partei benötigt für ihr Aufgabengebiet eigene Informationen, d.h. sie bekommt die Pläne von der vorherigen Partei und erstellt daraus ihre eigenen. Dieser Austausch findet meist in Papierform oder in untereinander nicht kompatiblen Datenformaten statt. Hier wäre es sinnvoll, ein geeignetes, untereinander kompatibles Dateiformat zu finden, in dem alle Ansichten verfügbar und leicht erweiterbar sind. So könnte man mit einem Mausklick von der Architektensicht in die Statiksicht wechseln. Auch würde der Aufwand für jede Partei geringer sein, da Daten aus einer anderen Sicht wiederverwendet werden können.

Die Aktualität von Baustoffdaten, wie bspw. die Stoffeigenschaften oder der Preis, spielt eine wichtige Rolle bei der Planung. Vorhandene Programme verwalten diese Daten in einer eigenen Datenbank die, um aktuell zu sein, regelmäßig aktualisiert werden muss. Dies geschieht bisher durch manuelle Eingabe oder Datenimport. Der Anwender muss sich diese Daten von verschiedenen Firmen besorgen. Auch hier liegen die Daten wieder in unterschiedlichen Formaten vor, die von Hand in eine datenbankgeeignete Form gebracht werden müssen. Es wäre praktisch, einen Automatismus zu finden, der diese Aufgaben übernimmt. So könnte ein Programm bei Bedarf, oder auch in regelmäßigen Abständen, neue Daten beschaffen, sie dann automatisch in ein geeignetes Format bringen und die Datenbank aktualisieren.
Hierfür bietet sich das Konzept der „mobilen Agenten“ an. Einmal die Suchinformationen an den Agenten übergeben, kann er selbstständig die Informationen aus dem Internet beschaffen. Dies kann manuell oder in bestimmten Zeitabständen erfolgen, so dass der Anwender hier entlastet würde.

Es gibt jedoch einige Punkte, die hierbei zu berücksichtigen sind:

- Es muss eine einheitliche Agentenumgebung vorliegen, d.h. es müsste ein Standard geschaffen werden, den jeder Hersteller, der seine Daten im Internet präsentiert, zu berücksichtigten hätte.

- Die Datenanfrage bzw. die Ausgabe der Daten muss in einer Form stattfinden, die sich von jedem System leicht interpretieren lässt.

- Jeder Hersteller hat seine Daten in einer eigenen Datenbankstruktur vorliegen. Wahrscheinlich sollen davon auch nicht alle Informationen zugänglich sein, sondern nur ein bestimmter Teil.

- Der Aufwand für den Hersteller muss minimal sein. Ein Erfolg ist nur dann abzusehen, wenn es dem Hersteller mehr Nutzen als Kosten bringt.

- Der Agent muss für den Anwender einfach zu bedienen sein.

- Die gesammelten Daten müssen richtig interpretiert und in einer geeigneten Form dargestellt bzw. weiterverarbeitet werden.

Im Rahmen dieser Studienarbeit soll ein Agentenserver, der als Host für mobile Agenten dient, erstellt werden. Dieser soll Datenbankanfragen der Agenten, die im XML-Format übergeben werden, auswerten und die Ergebnisse ebenfalls als XML-Datei liefern.
Ebenso soll der dazugehörige Agent erstellt werden. Dieser nimmt die Suchanfragen und die Reiseroute des Anwenders entgegen, erstellt daraus eine XML-Datei und arbeitet dann autonom die Reiseziele ab. Auf einem Zielserver angekommen, stellt er seine Suchanfragen und übernimmt die Ergebnisse. Bei der Rückkehr sollen die Ergebnisse dargestellt und in die vorhandene Baudatenbank geschrieben werden können.
Desweiteren soll der Agent eine CORBA-Schnittstelle bereitstellen, um von anderen Anwendungen, wie z.B. dem vorhandenen Bauteileditor, aus aufgerufen werden zu können.

Die Programmiersprache ist JAVA (JDK 1.2). Als Agentensystem wurde ObjectSpace Voyager benutzt. Die Datenbankabfragen werden über eine JDBC-Schnittstelle geführt. Als XML Parser kommt der „XML for JAVA Parser v2.0“ von IBM zum Einsatz. Als CORBA ORB dient Visigenic Visibroker.

2 Bestehende Technologien

2.1 Softwareagenten

***„Agent** (lat.) der , allg. eine Person, die für andere geschäftl. tätig ist, ...*“ Neues Universal Lexikon

Ein Software Agent ist ein Programm, das auch für eine andere Person, nämlich den Anwender, tätig ist. Das ist nur eine erste grobe Definition, die eine ungefähre Vorstellung von Softwareagenten vermitteln soll. Diese wird dann in den folgenden Kapiteln genauer beschrieben.

2.1.1 Klassifikation von Softwareagenten

Es existieren verschiedene Ansätze der Klassifikation von Softwareagenten. Jeder Ansatz betrachtet den Agent aus einem anderem Blickwinkel. Daraus resultieren unterschiedliche Kriterien für eine Klassifikation. So benötigt z.B. ein Entwickler andere Kriterien zur Klassifikation als ein Anwender.

2.1.1.1 Klassifikation nach Eigenschaften

Die Kriterien zur Klassifizierung nach Eigenschaften basieren auf den charakteristischen Eigenschaften eines Softwareagenten. Die wichtigsten sind:

- **Autonomie**. Agenten sind in der Lage, ihre vorgegebenen Aufgaben ohne regelnde Eingriffe des Benutzers selbst zu lösen. Hier sind verschiedene Grade der Autonomie möglich. Z.B. kann ein Agent den Benutzer erst fragen, ob er gesammelte Daten in eine Datenbank schreiben soll, oder dies selbstständig tun.

- **Mobilität.** Ein Agent kann zu jedem Zeitpunkt seine aktuelle Umgebung (Plattform) wechseln und nach dem Transfer seinen ursprünglichen Zustand wiederherstellen, um seine Arbeit fortzusetzen. Die Mobilität ist eng mit der Autonomie verknüpft, da der Agent auf einem fremden Rechner erst eine Verbindung zum Benutzer herstellen müsste, um bestimmte Entscheidungen anzufordern.

- **Kommunikationsfähigkeit**. Ein Agent benötigt normalerweise Schnittstellen zur Erfüllung seiner Aufgabe. Diese ermöglichen es ihm, mit anderen Programmen, dem

Anwender oder anderen Agenten zu kommunizieren. Die Kommunikation der Agenten untereinander ermöglicht den Einsatz in Multiagentensystemen.

- **Reaktionsfähigkeit.** Die Eigenschaft eines Agenten, auf Änderungen bzw. Informationen aus der realen oder virtuellen Umwelt in geeigneter Weise zu reagieren, wird als Reaktionsfähigkeit oder Reaktivität bezeichnet.

- **Pro-Aktionsfähigkeit.** Agenten handeln nicht nur in Reaktion auf ihre Umwelt, sondern sind auch in der Lage, zielgerichtete Handlungen aus eigener Initiative und Überlegung heraus einzuleiten.

- **Lernfähigkeit** bezeichnet die Eigenschaft eines Agenten, seine Handlungsweise auf Grund von Umwelteinflüssen und Informationen anzupassen.

Um diese Eigenschaften grafisch darzustellen, hat Brenner et al [Böhme 1999] die drei ihm am wichtigsten erscheinenden Kriterien ausgewählt und alle anderen diesen untergeordnet.

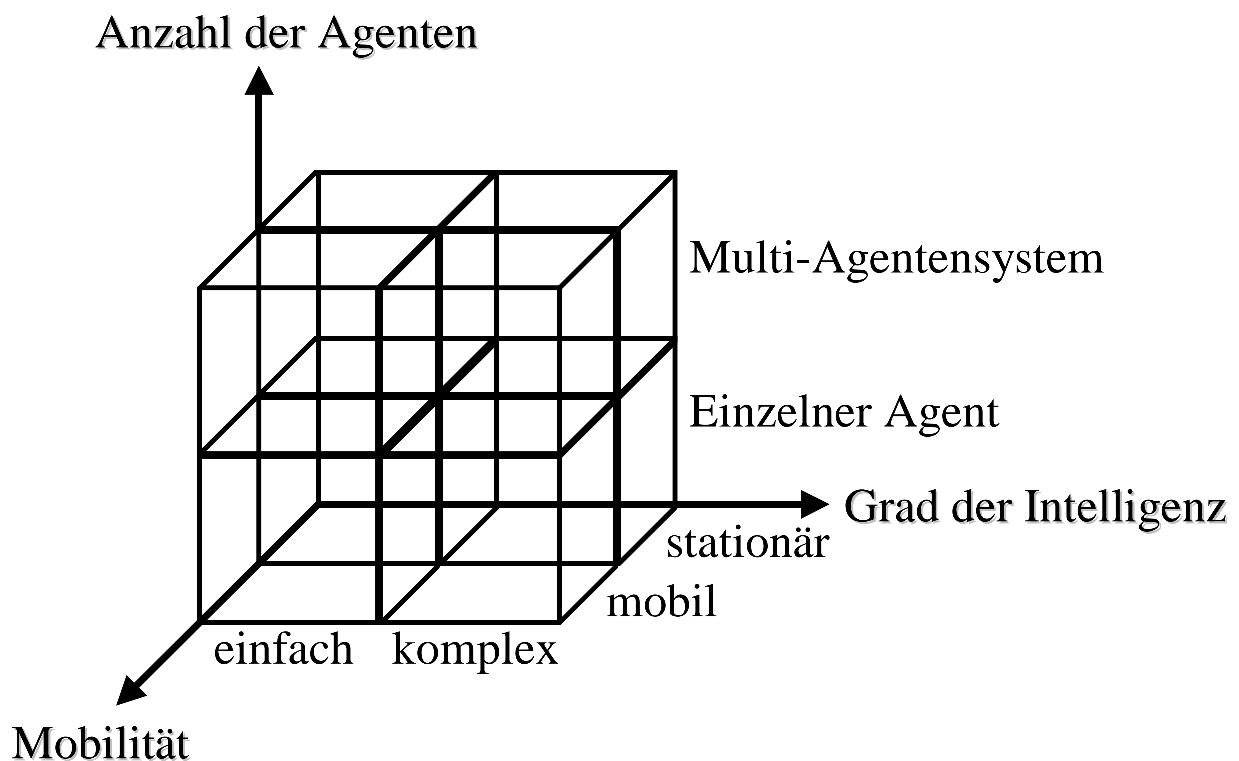


Abbildung 2-1: Klassifikationsmatrix nach Brenner et al [Böhme 1999]

2.1.1.2 Klassifikation nach Anwendung

Es lässt sich auch eine Klassifikation der Agenten nach der Art der Funktionalität vornehmen. Nach [Böhme 1999] kann man Agenten in folgende Kriterien einteilen, wobei es möglich ist, dass ein komplexer Agent mehrere Kriterien erfüllt:

- **Information Management** bezeichnet alle Softwareagenten, die mit der Beschaffung von Informationen beauftragt sind. Z.B. Agenten, die das Internet nach bestimmten Informationen durchsuchen oder Newsgroups überwachen und nur Nachrichten an den Anwender weiterleiten, die diesen wirklich interessieren.

- **Schnittstellenagenten,** treten in direkte Interaktion mit dem Anwender, der den Computer bedient. Sie geben kontextabhängige Hilfe sowie weiterführende Ratschläge. Der bekannteste Vertreter dieser Art dürfte der MS Office Assistent sein.

- **Prozessüberwachung und –steuerung.** Diese Agenten kommen in einer relativ geschlossenen Umgebung zum Einsatz. Sie überwachen z.B. bestimmte Abläufe auf einem Server und leiten entsprechende Maßnahmen ein, falls eine Abweichung auftritt.

- **Unterhaltung.** Hier existieren zwei Hauptrichtungen. Zum einen lassen sich in Computerspielen vorkommende Charaktere als Agenten implementieren. Zum anderen Agenten, die damit beschäftigt sind im Internet Personen mit gleichen Interessen zu suchen, um Kontakt mit diesen herzustellen. Z.B ICQ [http://www.icq.com/] oder der AOL Instant Messenger [http://www.aim.aol.com/].

- **Electronic Commerce.** Hier lassen sich alle Agenten einordnen, die irgendeine Art von Handel betreiben. Unabhängig ob für den Käufer- oder den Anbieter.

2.1.2 Vorteile mobiler Softwareagenten

Mobile Agenten bieten eine Reihe von Vorteilen gegenüber dem heutzutage vorherrschenden Client/Server Paradigma (s. Abbildung 2-2)

- Durch mobile Agenten kann eine Reduzierung der Netzwerklast erreicht werden. Dies trifft dann zu, wenn viele entfernte Anfragen an den Server gestellt werden. Ein Agent kann daraus viele lokale Anfragen machen. Zwar muss der Agent erst zum Server wandern, doch kann die Übertragung von Rohdaten schon größer sein als die Übertragung des Codes und des Zustands des Agenten. Im Schnitt sind lokale Anfragen 1.000 bis 100.000 mal schneller als entfernte Anfragen [Objectspace 1999].

- Agenten können untereinander kommunizieren und Daten austauschen. So kann z.B. ein Agent andere Agenten nach Daten oder Informationen befragen, die ihn der Erfüllung seines Ziels näher bringen

- Ein weiterer Vorteil liegt in der Offlinearbeit. Der Anwender kann, nachdem er dem Agenten einen Auftrag erteilt hat und dieser seine Reiseroute aufgenommen hat, offline gehen. Der Agent verrichtet dann weiter seine Arbeit und wartet, bis der Anwender wieder online ist um zurückzukehren.

- In verteilten Systemen bieten sie den Vorteil, dass der Code nicht auf jeder Maschine installiert werden muss. Man kann den Agenten zentral programmieren bzw. abändern und dann zu seinem Erfüllungsort schicken.

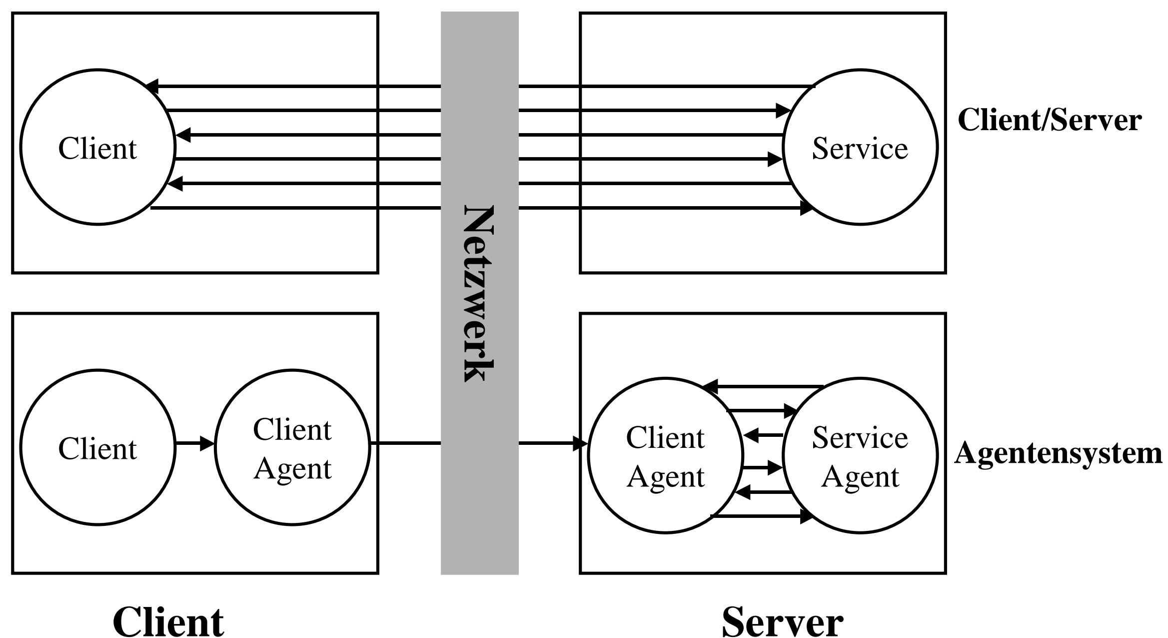


Abbildung 2-2: Client/Server-Paradigma und mobile Agenten

2.1.3 Anwendungsgebiete

Es sind mehrere Anwendungsgebiete mobiler Agenten denkbar:

- **System- und Netzmanagement.** Da das System- und Netzmanagement immer komplexere Formen annimmt, ist es sehr zeitaufwendig, die Funktion der einzelnen Komponenten, wie Server, Router, etc. zu überwachen. Ein Agent kann hier nach einem bestimmten Fahrplan selbstständig alles überwachen und kann eventuell kleinere Fehler selbst beheben bzw. den Administrator benachrichtigen.

- **Electronic Commerce.** Es ist denkbar, dass Agenten sich selbstständig durch das Internet bewegen, um ein bestimmtes Produkt zu suchen, Preisvergleiche anzustellen oder sogar im Auftrag des Benutzers Geschäfte abzuschließen.

- **Verteilte Informationssuche**. Der Agent zieht durch das Netz, sammelt Daten, verarbeitet diese an Ort und Stelle und kehrt nur mit relevanten Daten zurück. Der Anwender muss so keine großen Datenmengen herunterladen und kann sich, während der Agent arbeitet, anderen Aufgaben widmen.

- **Beobachtung von Auktionen oder Börsen**. Hier werden große Mengen von Informationen erzeugt, die aber nur zu einem kleinen Teil für einen Anleger oder Käufer von Interesse sind. Der Agent kann vor Ort die Kurse bzw. Preise beobachten und nur die Ereignisse von Interesse melden. Auch hier ist denkbar, dass er gleich den Vertrag abschließt.

- **Arbeitsgruppen**. Um Terminabsprachen zu treffen, kann z.B. ein Agent die Agenten der anderen Gruppenmitglieder aufsuchen und mit ihnen einen passenden Termin vereinbaren.

2.1.4 Nachteile mobiler Softwareagenten

Doch Agenten bieten nicht nur Vorteile. Sie werfen auch neue Probleme auf:

- **Das Infrastrukturproblem**. Agenten können nur dorthin migrieren, wo eine entsprechende Agentenplattform vorhanden ist. Diese ist zur Zeit kaum oder gar nicht vorhanden. Zur Zeit existieren mehrere Agentensysteme die nicht untereinander kompatibel sind.

- **Sicherheit.** Ein großes Problem für Agenten stellt die Sicherheit dar. Jeder Agent trägt zu einem Teil Informationen mit sich, die für andere interessant sein könnten. Dies kann von einfachen Informationen bis hin zu brisanten Daten alles sein. Im Boom des Electronic Commerce ist es denkbar, dass Agenten sogar elektronische Zahlungsmittel mit sich tragen. Ein Agent bietet ein einfacheres Ziel für Angriffe, da er sich nicht entfernt auf einem Rechner befindet und z.B. durch eine Firewall geschützt ist. Auch muss sich ein Agent darauf verlassen, dass er seinen Code, den er mit sich führt, auch auf dem entfernten Rechner ausführen kann.

- **Größe.** Mit wachsender Intelligenz eines Agenten wächst auch die Größe seines Codes. Auch Agenten, die Daten sammeln, wachsen mit der Anzahl der gewonnenen Informationen, d.h. der Vorteil der Netzwerkentlastung wird für komplexere Agenten immer geringer.

2.1.5 Agentensysteme

Es existieren zur Zeit unterschiedliche Ansätze für Agententechnologien. Da sich die IT-Branche rasend schnell entwickelt, sind hier nur die bekanntesten genannt:

- **ObjectSpace Voyager (http://www.objectspace.com/)**
- **IBM Aglets (http://www.trl.ibm.co.jp/aglets)**
- **General Magic, inc. Odyssey (http://www.generalmagic.com/)**
- **Mitsubishi Concordia (http://www.meitca.com/)**

Die Agentensysteme basieren alle auf JAVA, sind untereinander nicht kompatibel und unterscheiden sich teilweise erheblich in ihrer Syntax. Unter 2.5 wird näher auf ObjectSpace Voyager eingegangen.

2.1.6 Standardisierung

Zur Standardisierung der Agentensysteme existieren eine Reihe verschiedener Ansätze [Böhme 1999]:

- **Knowledge Sharing Effort (KSE).** Entwicklung einer Agenten-Kommunikationssprache für die standardisierte Kommunikation zwischen verschiedenen Agenten (ACL Agent Communication Language).

- **Foundation for Intelligent Physical Agents (FIPA).** Die FIPA wurde eigens mit dem Ziel des Erarbeitens von Richtlinien und Standards für die Entwicklung von Agenten gegründet. Sie definiert Schnittstellen für die mit dem Agenten interagierenden Komponenten. Diese können Software allgemein, andere Agenten oder der Mensch selbst sein.

- **Agent Society (http://www.agent.org/).** Sie hat zum Ziel, die Förderung und Koordination neuer Entwicklungen auf dem Gebiet der Agentensysteme sowie damit verbundener Technologien.

- **Object Managment Group (OMG, http://www.omg.org/).** Spezifikationen für verteilte Objekte. Veröffentlichung der Mobile Agent System Interoperability Facilities Specification (MASIF) mit dem Ziel der Interoperabilität von Agentensystemen mit Unterstützung mobiler Agenten.

- **World Wide Web Consortium (W^{3}C, http://www.w3.org/).** Spezifikation von Übertragungsprotokollen wie XML und Resource Description Framework (RDF).

2.2 JAVA

Java ist eine objektorientierte Programmiersprache. Sie bietet eine portable, leistungsfähige und interpretierende Entwicklungsumgebung. Mit ihr lassen sich plattformunabhängige Anwendungen entwickeln, die auf beliebigen Rechnern laufen.

2.2.1 Entwicklungsgeschichte von JAVA

JAVA wurde 1990 von SUN Microsystems mit dem Ziel entwickelt, die Elektronik von Kleinstgeräten wie Haushaltsgeräten, Set-Top-Boxen oder Routern zu steuern. Den Durchbruch erlangte sie 1995 auf dem Gebiet der PCs mit der Integration in einen populären Browser, den Netscape Navigator 2.0. Durch die Integration der virtuellen Maschine (JAVA Virtual Machine, kurz JVM) können Programme über das Internet geladen und innerhalb einer Webseite ausgeführt werden. 1999 wurde dann die Version 1.2 vorgestellt.

2.2.2 Was ist JAVA?

Ein JAVA Programm wird gleichzeitig kompiliert und interpretiert. Der Compiler übersetzt ein Programm in eine Art „Zwischensprache“ (JAVA Bytecode). Dieser Bytecode kann dann in jeder JVM interpretiert werden.

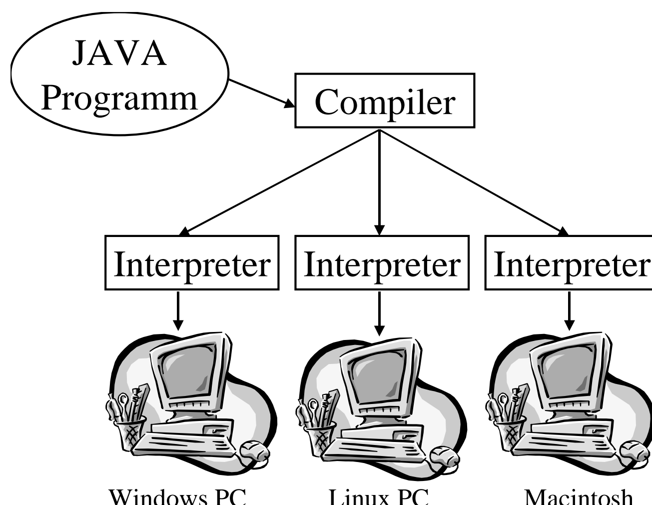


Abbildung 2-3: Die verschiedenen Stufen eines JAVA Programms

Der Vorteil liegt hier in der Kompaktheit (verringerte Ladezeit) des Codes und seiner Plattformunabhängigkeit. Hinzu kommt die Fähigkeit der JVM zu erkennen, welche Operationen ein Programm ausführen will, um so unsichere Operationen, wie z.B. Dateizugriffe, verhindern.

Weitere Merkmale sind:

- **Applets**: Ausführen von Programmen über das Internet in einem Browser.
- **Netzwerkfähigkeit**: URLs, TCP-, UDP-Sockets und IP Adressen.
- **Sicherheit**: Elektronische Signaturen, öffentliche/private Schlüssel, Zugriffskontrolle und Zertifikate.
- **Object serialization**: Erlaubt eine einfache Speicherung von Objekten und Kommunikation über Remote Method Invocation (RMI).
- **Java Database Connectivity (JDBC)**

Eine JAVA Entwicklungsumgebung (JDK) kann kostenlos unter http://www.javasoft.com heruntergeladen werden.

2.2.3 Java und mobile Softwareagenten

Natürlich lässt sich ein Agent in jeder beliebigen Sprache programmieren, doch bietet Java hier zwei bedeutende Vorteile. Zum einen unterstützt Java von sich aus die Codemobilität. So wird bei einem Applet erst der Code von einem Server geladen und dann lokal im Browser ausgeführt. Java besitzt daher auch eine Reihe nützlicher Netzwerkfunktionen. Zum anderen ist Java plattformunabhängig. Daher muss eine Anwendung nur einmal erstellt, und nicht für jede Plattform eigens umgeschrieben werden.

2.2.4 Java Database Connectivity (JDBC)

JDBC ist eine Datenbankschnittstelle, die aus verschiedenen Standards von JavaSoft in Zusammenarbeit mit Intersolv entwickelt wurde. Die Hauptrolle spielte dabei ODBC (Open Database Connectivity). Ziel war es, alle Möglichkeiten, die relationale Datenbanken bieten, zu nutzen und gleichzeitig die Benutzbarkeit einer solchen Schnittstelle zu erleichtern. JDBC ist seit JDK 1.1 fester Bestandteil von JAVA und liegt seit JDK 1.2 in der Version 2.0 vor.

2.2.4.1 Aufbau und Struktur

Es gibt vier Möglichkeiten, JAVA-Anwendungen durch JDBC mit einer Datenbank zu verbinden:

- **ODBC/JDBC Bridge.** Ermöglicht den Zugriff auf jede Datenbank, die eine ODBC-Schnittstelle anbietet. Dabei werden JDBC-Aufrufe in ODBC-Aufrufe übersetzt. Dieser Treiber wird kostenlos von SUN oder Intersolv angeboten.

- **Plattformspezifische JDBC-Treiber.** JDBC-Aufrufe werden in die spezifische API der Datenbank übersetzt. Diese liegt meist nicht in JAVA sondern in C/C++ vor, so dass eine JAVA-Bridge DLL den Zugang zu dieser Software ermöglicht.

- **Universelle JDBC-Treiber.** JDBC-Aufrufe werden über ein datenbankunabhängiges Protokoll in ein Datenbanknetzwerk-Protokoll übersetzt.

- **Direkte JDBC-Treiber.** Übersetzung der JDBC-Aufrufe in das zur Datenbank gehörige Netzwerkprotokoll. Es liegt ein direkter und damit seht schneller Zugang vom Client zur Datenbank vor.Nachteilig ist, dass die Treiber fast ausschließlich von Drittanbietern angeboten werden.

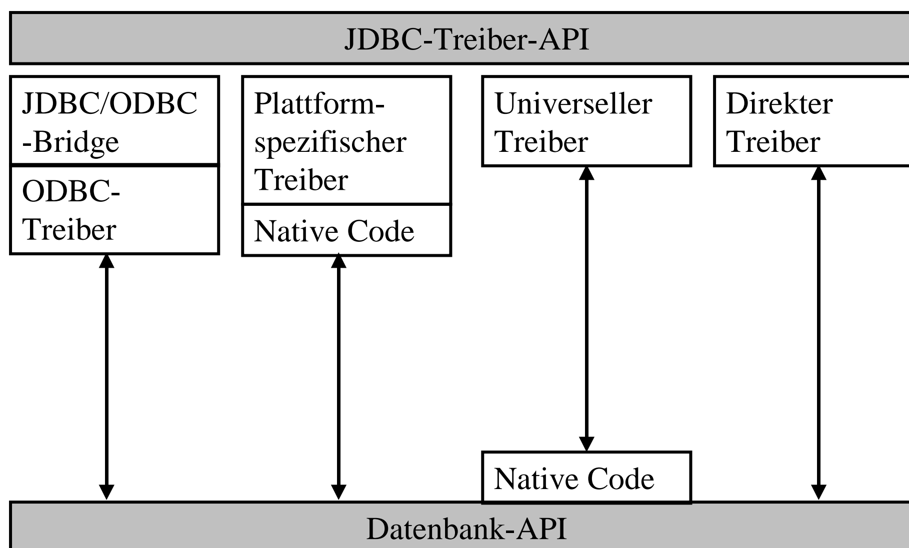


Abbildung 2-4: Die verschieden JDBC-Treiber [Boger 1999]

Die Entscheidung, welcher Treiber zum Einsatz kommt, ist von der Programmierung unabhängig. Es hat lediglich Auswirkungen auf Performanz und Kosten.

2.2.4.2 Verbindungsaufbau

Hierfür ist die zentrale JDBC Klasse `DriverManager` zuständig. Mit dem Aufruf

```
DriverManager.getConnection(String datenbankURL);
```

wird eine Verbindung zu der als Parameter übergebenen URL hergestellt. Die URL hat immer folgende Form: jdbc:*subprotokol:subname*. Das Subprotokoll gibt hierbei an, welcher Treiber verwendet werden soll. Der Subname ist die Bezeichnung einer entfernten oder lokalen Datenbank, die verwendet werden soll.

```
//...
Connection con =
   DriverManager.getConnection("jdbc:odbc://localhost/Baudatenbank");
//...
```

Um Anfragen an die Datenbank zu stellen, muss nur noch ein `Statement` Objekt erzeugt werden, das dann eine Abfrage ausführen kann. Als Ergebnis erhält man ein `ResultSet`.

```
//...
Statement sm = con.create();
ResultSet rst = sm.executeQuery("SELECT * FROM Tbl_Beton");
//...
```

Um die Ergebnisse aus einem `ResultSet` auszulesen, stehen folgende Methoden bereit:

- **`getXXX(String spaltenName).`** Liefert den Wert, der als Spaltennamen übergebenen Spalte des aktuellen Datensatzes. Wobei `XXX` für die verschiedenen JAVA Datentypen steht.

- **`getXXX(int i).`** Liefert den Wert, der als Parameter übergebenen Spaltennummer (erste Spalte=1, Zweite Spalte=2, usw.) des aktuellen Datensatzes. Wobei `XXX` für die verschiedenen JAVA-Datentypen steht.

```
//...
Double preis = rst.getDouble("Preis");
//...
```

2.3 XML

Die Extensible Markup Language (XML) wurde im Februar 1998 vom W³C veröffentlicht. Sie hat eine gewisse Ähnlichkeit mit der Hyper Text Markup Language (HTML), basiert aber auf dem älteren SGML. XML kann zum Darstellen und Austauschen von Daten verwendet werden.

2.3.1 Was bietet XML gegenüber HTML?

XML ist eine Meta-Sprache, d.h. mit ihr lassen sich eigene Sprachen für bestimmte Zwecke definieren. Sie ist wesentlich flexibler als HTML. In HTML existiert ein vorgegebener Satz Tags, mit denen man auskommen muss. In XML hingegen kann man beliebig viele Tags selbst definieren. Darüber hinaus kann man die Beziehung der Tags individuell festlegen. Dies ermöglicht es, beliebig strukturierte, selbstbeschreibende Informationen abzubilden.

Ein Beispiel für Informationen, die sich mit XML modellieren lassen, ist ein Artikel über einen neuen Baustoff:

<artikel>
*<titel>*Neue Baustoffe*</titel>*
<text>
*<produkt>*Die Firma *<hersteller>*Beton Müller*</hersteller>* hat einen neuen
Baustoff entwickelt. *<baustoff>*Elastischer Beton*</baustoff>*besitzt perfekte
Materialkennwerte. Er hat z.B. Eine Dichte von *<dichte>*400*</dichte>* und
...*</produkt><produkt>*Auch die Firma *<hersteller>*Stahl Peter*</hersteller>* hat ein
neues Produkt: *<baustoff>*Flüssiger Stahl*</baustoff>*. Er hat eine Dichte von
*<dichte>*200*</dichte>* und ...*</produkt>*
</text>
</artikel>

XML Tag

Dieses Beispiel lässt sich nun auf verschiedene Arten darstellen. Es kann als einfacher Artikel dargestellt werden, indem die Tags als Textformatierung interpretiert werden, bzw. weggelassen werden:

Neue Baustoffe

Die Firma Beton Müller hat einen neuen Baustoff entwickelt. Elastischer Beton besitzt perfekte Materialkennwerte. Er hat z.B. Eine Dichte von 400 und ... Auch die Firma Stahl Peter hat ein neues Produkt: Flüssiger Stahl. Er hat eine Dichte von 200 und ...

Oder z.B. Tabellarisch:

Tabelle Produkt:

Hersteller	Baustoff	Dichte	...
Beton Müller	Elastischer Beton	400	
Stahl Peter	Flüssiger Stahl	200	

Dies sind nur zwei von vielen denkbaren Darstellungsweisen. Hierin liegt das Potential von XML. Es lassen sich für verschiedene Benutzergruppen eigene Sichten definieren, was zur Folge hat, dass jeder nur den für sich relevanten Teil sieht.

Ein XML-Dokument hat eine logische und eine physikalische Struktur. Die physikalische Struktur ermöglicht es, bestimmte Dokumententeile auszulagern. So können, wie bei HTML, Textpassagen oder Bilder, die in mehreren Dokumenten gleichzeitig vorkommen, separat in eigenen Dateien abgelegt werden. Diese können dann über Verweise in das eigentlich XML-Dokument eingebunden werden.

Mit der logischen Struktur kann man das Dokument in mehrere Bereiche (**Elemente**) unterteilen. Diese Bereiche können frei eingeteilt und benannt werden. Hierfür werden die o.g. Tags benutzt.

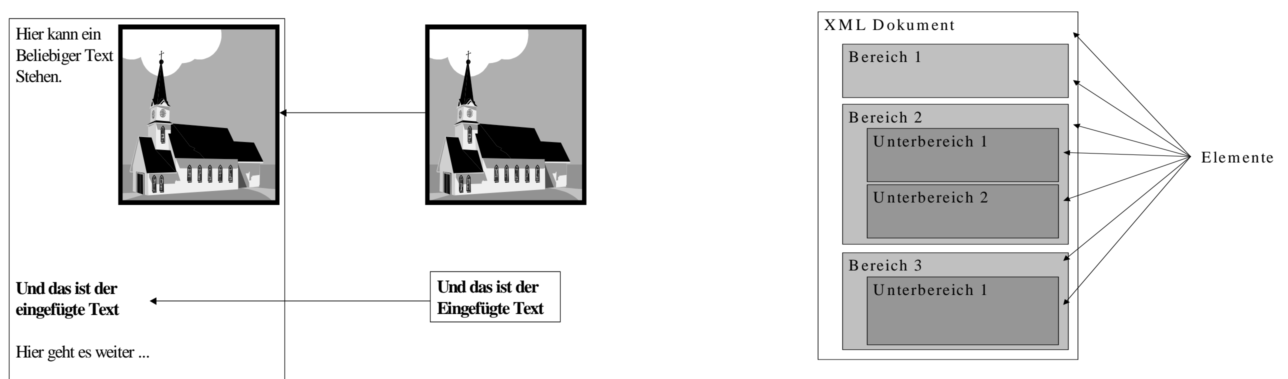


Abbildung 2-5: Aufbau von XML Dokumenten

Datenaustausch

Da XML-Dokumente selbstbeschreibend sind, eignen sie sich hervorragend für den Austausch von Daten zwischen unterschiedlichen Programmen. Das Programm selbst kann sich dann den für sich relevanten Teil leicht auslesen. So lassen sich z.B. Beziehungen in einer Datenbank durch XML besser darstellen als in einer einfachen Textdatei.

2.3.2 Das XML-Dokument

2.3.2.1 Zeichenkonventionen

Es gelten die gleichen Zeichenkonventionen wie in HTML. Da bestimmte Zeichen, die für Tags verwendet werden, nicht benutzt werden dürfen, ersetzt man sie durch Referenzen. So wird in einem Text z.B. ein Größerzeichen durch „>“ ersetzt.
XML bietet darüber hinaus eine „Character Data Section“ (CDATA). In einer CDATA eingeschlossene Zeichen werden nicht als Tags interpretiert.

> *<!CDATA[*Dieser Text enthält Zeichen, die normalerweise so nicht benutzt werden dürfen: <<<!!!&&&>>>*]]>*
>
> **Dieser Text enthält Zeichen, die normalerweise so nicht benutzt werden dürfen: <<<!!!&&&>>>**

2.3.2.2 Elemente

Wie schon erwähnt, wird eine XML-Datei durch Bereiche (Elemente) untergliedert. Dies wird mittels der selbstdefinierten Tags erreicht. Ein Tag besteht aus dem Namen, der von einem Kleiner- und Größerzeichen eingeschlossen ist:

> *<Tagname>*

Der Name ist „case sensitive“, d.h. Klein- bzw. Großschreibung wird beachtet. Ein Bereich, oder auch Element genannt, wird immer mittels eines Tags eingeleitet und endet ebenfalls mit einem Tag. So existiert für jeden Bereich ein Start- und ein Ende-Tag. Der Ende-Tag hat den selben Namen wie der Start-Tag, ihm wird nur ein „Slash“ vorangestellt:

> <tag>Bereich</tag>

Jeder Bereich kann beliebig viel Unterbereiche haben (s. Abb. 3-2) und kann nur komplett in einem anderen Bereich liegen. Als Beispiel kann ein Dokument dienen:

Es enthält mehrere Kapitel und Absätze, jedoch kann ein Absatz nicht über zwei Kapitel gehen.
In einer DTD (s. 2.3.3) lässt sich genau festlegen, welcher Bereich Unterbereiche haben darf. So sind auch rekursive Strukturen möglich.

2.3.2.3 Attribute

Elemente können Informationen (Attribute) über den Bereich, den sie einschließen, enthalten. So kann jedes Element mehrere Attribute besitzen. Attribute sind im Start-Tag eingebettet und setzen sich aus dem Attributnamen und dem Attributwert, getrennt von einem Gleichheitszeichen, zusammen. Der Attributwert muss immer in Anführungszeichen oder Hochkommas gesetzt werden, sonst wird alles hinter dem Gleichheitszeichen bis zum Größerzeichen als Attributwert angenommen.

<tag attributname1="attributwert1" attributname2="attributwert2">

So kann man z.B. die Einheit zu einem bestimmten Wert festlegen:

<dichte einheit="kg_pro_m3">400*</dichte>*

Wird eine DTD benutzt, kann darüber hinaus festgelegt werden, von welchem Datentyp der Attributwert sein soll. Attributname sowie Attributwert werden immer durch Groß-/Kleinschreibung unterschieden

2.3.2.4 Kommentare

Kommentare werden von einem Kleinerzeichen, gefolgt von einem Ausrufezeichen und zwei Minuszeichen eingeleitet und enden mit zwei Minuszeichen und dem Größerzeichen.

<!-- Dies ist ein Kommentar-->

2.3.2.5 XML-Deklaration

In einem XML-Dokument sollte zuerst ein Tag stehen, der Informationen über das Dokument enthält. Dieser wird XML-Deklaration genannt. Er enthält Informationen über die verwendete XML-Version, den Zeichencode, und ob das Dokument alleine interpretiert werden kann oder eine externe DTD hinzugezogen werden muss.

<?XML version="1.0" encoding="UTF-8" standalone="yes" ?>

Wird dies nicht angegeben, wird standardmäßig die *Version 1.0* angenommen. Der *characterset* als *„UTF-8“* und *standalone* als *„no“*.

2.3.2.6 Dokumententyp

Am Anfang eines XML-Dokumentes sollte die „Dokumententyp“ Deklaration stehen.

<!DOCTYPE MeinTyp>

Hierin kann auch die DTD selbst stehen bzw. ein Verweis darauf. Ein Verweis auf eine ausgelagerte DTD-Datei wird mit dem Schlüsselwort „SYSTEM“ und dem Ort der Datei erzeugt.

<!DOCTYPE MeinTyp [
<!--Hier können die Definitionen (DTD) stehen-->
]>

<!DOCTYPE MeinTyp SYSTEM “./dtd/system.dtd“ [
<!--Hier können noch zusätzliche Definitionen (DTD) stehen-->
]>

2.3.2.7 Wurzel-Element / Dokument-Element

In jedem XML-Dokument gibt es ein Element, das den Container für alle anderen Elemente bildet. Der Name dieses Elements erscheint in der Dokumenttyp-Deklaration.

<!DOCTYPE Wurzelelement SYSTEM “./dtd/Wurzelelement.dtd“>
< Wurzelelement>
...
</Wurzelelement>

2.3.3 Die logische Struktur (DTD)

Um zu definieren, welche Tags in einem Dokument vorkommen dürfen und wie diese untereinander in Beziehung stehen, kann eine „Document Type Definition“ (DTD) verwendet werden. In ihr werden die grammatikalischen Regeln für den Einsatz von Tags und deren Attribute definiert. Jedes in der DTD aufgeführte Element kann entweder wieder ein Container für weitere Elemente sein, oder bleibt leer. Ein Container kann Text, weitere Container oder beides enthalten. Es kann genau festgelegt werden, ob ein Container genau ein

Element oder mehrere Elemente enthalten muss bzw. optionale Elemente enthält. Für die Attribute kann der exakte Datentyp festgelegt werden.

Es gibt zwei Möglichkeiten, eine DTD abzulegen. Sie kann entweder im Kopf der XML-Datei stehen (s.o.) oder in einer externen Datei abgelegt werden. Dies ermöglicht, dass die Datei mehreren XML-Dokumenten zur Verfügung steht und nicht immer neu erstellt werden muss. Es ist auch ein Mix aus beidem möglich. So kann z.B. eine allgemeine DTD, die für alle XML-Dokumente gilt, erstellt werden und in jeder XML-Datei noch zusätzliche Definitionen abgelegt werden. Es existieren zahlreiche DTDs für verschiedene Anwendungsgebiete: Z.B. für die Darstellung von mathematischen Formeln oder chemischen Molekülen. Auch lässt sich ein erstelltes Dokument mittels einer spezifischen DTD auf Korrektheit der Tags überprüfen.

2.3.3.1 Deklaration der Elemente

Ein Element wird mit dem Schlüsselwort „ELEMENT“ deklariert, gefolgt vom Elementnamen und einer Deklaration dessen, was das Element beinhalten soll. Der Name muss mit einem Buchstaben beginnen und darf folgende Zeichen beinhalten: Alle Buchstaben, alle Zahlen, und die Zeichen Komma „,“, Punkt „.“, Unterstrich „_“ und Doppelpunkt „:“. Ein Leerzeichen ist nicht zulässig.

<!ELEMENT name inhalt>

Der Inhalt kann entweder „ANY“, „EMPTY“ oder eine „model group“ sein. ANY bedeutet, dass das Element jedes in der DTD deklarierte Element beinhalten kann. EMPTY steht für leer und bedeutet, dass weder Text noch andere Elemente eingeschlossen werden dürfen. Dies macht z.B. Sinn als Platzhalter für ein Bild.
Mit einer „model group“ kann man definieren, ob und welche Unterelemente ein Element beinhaltet. Dies ist etwas komplexer und wird deshalb in einem eigenen Abschnitt behandelt wird.

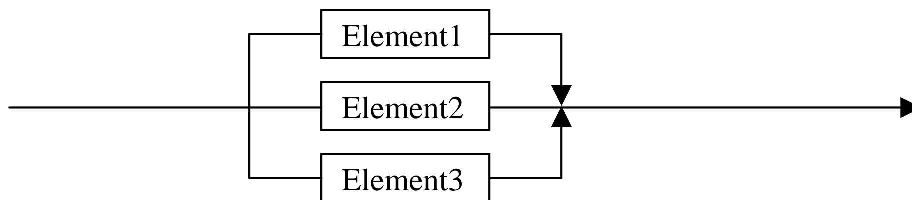
2.3.3.2 Die Model group

Eine „model group“ wird verwendet, um für ein Element zu deklarieren, ob und welche anderen Element es enthalten kann bzw. ob es einen gemischten Inhalt von Text und Elementen haben soll. Eine „model group“ wird immer durch Klammern eingeleitet. Darin befinden sich eine Aufzählung der Elemente, die das Element beinhalten soll. Um die Reihenfolge bzw. das Auftreten eines Elementes genau festzulegen, existieren zwei Verknüpfungsoperatoren.

<u>Die Reihe</u>:
Die Elemente werden durch Kommas getrennt (Element1, Element2, Element3). Dies bedeutet, dass auf Element1 Element2 folgt, auf das wiederum Element3 folgt.
Element1

<!ELEMENT name (Element1, Element2, ...)>

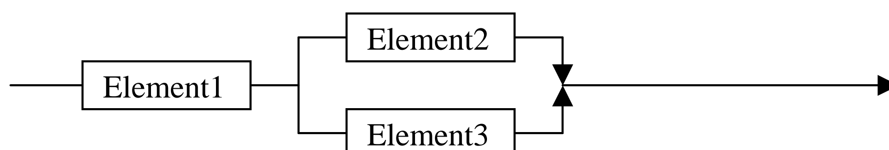
<u>Die Auswahl</u>:
Die Elemente werden durch “|” getrennt (Element1 | Element2 | Element3). Bei der Auswahl kann nur eines der drei Elemente auftauchen. Entweder Element1 oder Element2 oder Element3.



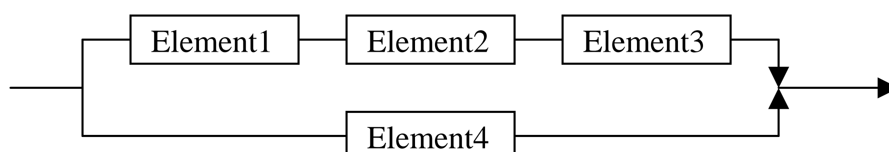
<!ELEMENT name (Element1| Element2| ...)>

Um beide Verknüpfungsoperatoren zu kombinieren, müssen Klammern gesetzt werden.
Beispiele:

<!ELEMENT name (Element1, (Element2 | Element3))>



<!ELEMENT name ((Element1, Element2 , Element3) | Element4)>

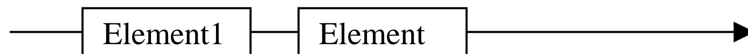


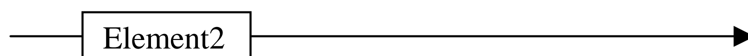
Man hat auch die Möglichkeit zu definieren, wie oft ein einzelnes Element vorkommen darf. Standardmäßig muss jedes Element genau einmal vorkommen. Ist ein Element optional, muss

ein Fragezeichen angehängt werden (Element1?, Element2). Ein Element, dass mindestens einmal vorkommen muss und sich wiederholen darf, wird mit einem Pluszeichen gekennzeichnet. (Elemente1+, Element2). Soll ein Element optional sein, kann sich aber wiederholen, wird ein „*“ angehängt.

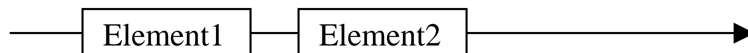
Beispiele:

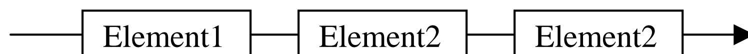
<!ELEMENT name (Element1?, Element2)>



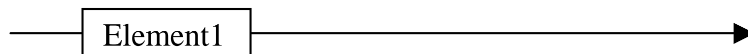


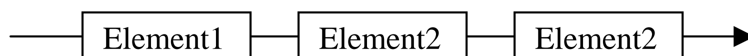
<!ELEMENT name (Element1, Element2+)>





<!ELEMENT name (Element1, Element2*)>





Wenn die Menge für alle Elemente in der Klammer gleich sein soll, kann der Operator auch an die Klammer angehängt werden.

<!ELEMENT name (Element1, Element2, Element3)?>

entspricht

<!ELEMENT name (Element1?, Element2?, Element3?)>

Ein Element das Text beinhaltet, wird mit dem Schlüsselwort „#PCDATA“ definiert.

<!ELEMENT name (#PCDATA)>

Soll ein Element sowohl Text als auch Elemente beinhalten, muss folgende Regel eingehalten werden:

- Das Schlüsselwort „#PCDATA“ muss das erste Element der „model group“ sein.
- Die ganze „model group“ muss eine Auswahlgruppe sein.
- Die „model group“ muss optional und wiederholbar sein.

<!ELEMENT produkt (#PCDATA | hersteller | baustoff)*>

> *<produkt>*Die Firma *<hersteller>*Beton Müller*</hersteller>* hat einen neuen Baustoff entwickelt. *<baustoff>*Elastischer Beton*</baustoff>*besitzt perfekte Materialkennwerte. Er hat z.B. Eine Dichte von *<dichte>*400*</dichte>* und *...</produkt>*

2.3.3.3 Deklaration der Attribute

Alle Attribute eines Elementes werden getrennt in einer „Attributliste“ deklariert. Die Attributliste wird durch das Schlüsselwort „ATTLIST“ eingeleitet, gefolgt vom Namen des Elements zu dem die Attribute gehören sollen. Danach folgen der Name des Attributs und der Datentyp. Es kann dann noch angegeben werden, ob das Attribut erforderlich (#REQUIRED) oder optional (#IMPLIED) ist.

```
<!ATTLIST element attribut1 datentyp erforderlich
                  attribut2 datentyp erforderlich
                  ...       ...      ...>
```

Der Attributname unterliegt denselben Zeichenkonventionen wie der Elementname.

Es existieren folgende Datentypen [Bradley 1998]:

- CDATA   Eine beliebige Zeichenfolge. CDATA ist das Gegenstück zu #PCDATA

- NMTOKEN   Ein Wort oder Zeichen. Es gelten die selben Zeichenkonventionen wie für die Deklarationen.

- NMTOKENS   Mehrere Worte oder Zeichen.

- ENTITY   Verweis auf eine externe Datei. Z.B. wenn ein Element als Platzhalter für ein Bild steht.

- ID, IDREF, IDREFS Wird für Hyperlinks in XML-Dokumenten benötigt.

- NOTATION Gibt eine Liste der Datentypen an, die das Element beinhalten kann. Der Attributwert muss dann einem in der Liste entsprechen.
  <!ATTLIST document format NOTATION (doc | txt)>

- „name group“ Gibt eine Liste von Werten vor. Der Attributwert muss dann einem Wert in der Liste entsprechen.
  <!ATTLIST bild sichtbar (true | false) >

Bei „NOTATION und „name group“ besteht die Möglichkeit, einen Standardwert anzugeben. Dieser wird in Anführungszeichen direkt hinter die Liste geschrieben

<!ATTLIST bild sichtbar (true | false) “true“>

Es können auch mehrere Attributlisten zu einem Element angelegt werden. Diese werden dann kombiniert, wobei die zuerst erscheinende Liste Vorrang hat.

Beispiel für eine Deklaration:

<!ELEMENT hersteller (#PCDATA)>
<!ELEMENT baustoff (#PCDATA)>
<!ATTLIST baustoff preis CDATA #REQUIRED
einheit (kg | m3) “m3”>
<!ELEMENT dichte (#PCDATA)>
<!ATTLIST dichte einheit (kg_pro_m3 | g_pro_cm3) „kg_pro_m3“>
<!ELEMENT produkt (#PCDATA | hersteller | baustoff | dichte)>*

*<produkt>*Die Firma *<hersteller>*Beton Müller*</hersteller>* hat einen neuen Baustoff entwickelt. *<baustoff preis=“500 DM“ einheit=“kg“>*Elastischer Beton*</baustoff>*besitzt perfekte Materialkennwerte. Er hat z.B. Eine Dichte von *<dichte dichte=“g_pro_cm3“>*400*</dichte>* und ...*</produkt>*
*<produkt>*Auch die Firma *<hersteller>*Stahl Peter*</hersteller>* hat ein neues Produkt: *<baustoff preis=“333 DM“>*Flüssiger Stahl*</baustoff>*. Er hat eine Dichte von *<dichte>*200*</dichte>* und ...*</produkt>*

2.3.4 Parser

Im Rahmen von XML wird zwischen gültigen und wohlgeformten Dokumenten unterschieden. Gültig ist ein Dokument, wenn es die Regeln einer vorhandenen DTD einhält.

Ein Dokument, das ohne eine DTD erstellt wurde, kann noch den Status der Wohlgeformtheit erreichen. Dazu müssen folgende Regeln eingehalten werden:

- Alle Attribut-Werte müssen in Anführungszeichen stehen.

- Leere Elemente (EMPTY) müssen entweder mit „/>" enden oder man muss einen regulären End-Tag hinzufügen.
  <*leeresElement*/> oder < *leeresElement* ></ *leeresElement* >

- Alle Elemente müssen ineinander eingebettet sein. (Jeder Bereich muss komplett in einem anderen liegen.)

Wenn es um die Verarbeitung von XML-Dokumenten geht, unterscheidet die XML-Spezifikation zwei Instanzen: Den „Parser" und die „Anwendung". Der Parser verrichtet seine Arbeit im Dienste einer Anwendung.

Ein Parser übernimmt folgende Aufgaben:

- Er überprüft das Dokument auf seine Gültigkeit bzw. Wohlgeformtheit und gibt eventuell Fehlermeldungen aus.

- Wenn sich das Dokument aus Bestandteilen zusammensetzt (Physikalische Struktur), die sich in unterschiedlichen Dateien befinden, führt er die Bestandteile zusammen.

- Er erstellt einen Baum, durch den die Inhalte der Elemente für die Anwendung zugreifbar werden.

Die XML-Spezifikation gibt vor, wie sich der Prozessor beim Umgang mit Dokumenten zu verhalten hat. Es werden jedoch keinerlei Vorgaben für das Verhalten der Anwendung geliefert. Man unterscheidet zwischen validierenden und nicht-validierenden Parsern. Validierende Parser prüfen das Dokument auf Gültigkeit und Wohlgeformtheit, nicht-validierende prüfen lediglich auf Wohlgeformtheit (wobei Verstöße gegen die Wohlgeformtheit die schwerwiegenderen Verstöße sind).

Es existieren zwei Techniken, um ein XML-Dokument zu verarbeiten. Zum einen die „ereignisgesteuerte" und zum anderen die „Baum-Manipulation". Bei der „ereignisgesteuerten" wird das Dokument einfach als Datenstrom eingelesen. Jedes Element löst dann ein bestimmtes Ereignis in der Applikation aus. Die „Baum-Manipulation" liest das Dokument als ganzes und erstellt aus den Elementen eine Baumstruktur. So kann jederzeit auf ein beliebiges Element in beliebiger Reihenfolge zugegriffen werden. Hier gibt es zwei Ansätze für Standards: Die „Simple API for XML" (SAX) als „ereignisgesteuerte" Technik und das „Document Object Model" (DOM) als „Baum-Manipulation".

Zahlreiche Firmen bieten im Internet Parser für die unterschiedlichsten Programmiersprachen an. Z.B. XML4J Parser von IBM (http://www.alphaworks.ibm.com). Dieser wird auch in der Version 2.0.15 für die folgenden Beispiele genutzt. Alle Beispiele sind in JAVA. Man sollte bei der Auswahl des Parsers darauf achten, dass er die Standards SAX und DOM einhält.

2.3.4.1 SAX

Der SAX Standard befindet sich noch immer in der Entwicklung. Unter http://www.microstar.com/XML/SAX/spec.html kann man sich über den aktuellen Stand informieren. Ein SAX Parser liest ein Dokument als Datenstrom. Stößt er dabei auf ein Element oder Text löst er ein Ereignis aus. Dies muss abgefangen und behandelt werden. Der Ablauf ist immer gleich:

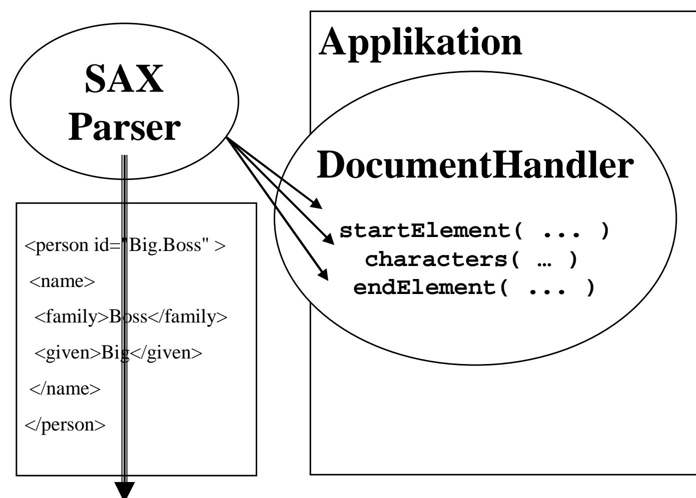


Abbildung 2-6: SAX-Parser

Zuerst muss eine DocumentHandler Klasse erstellt werden, die das DocumentHandler Interface implementiert und zum Abfangen der Ereignisse des Parsers dient. Diese Klasse wird dann dem Parser übergeben:

```
MyDocumentHandler handler = new MyDocumentHandler ();
Parser parser = ParserFactory.makeParser(com.ibm.xml.parsers.SAXParser);
parser.setDocumentHandler(handler);
```

Mit dem Aufruf

```
parser.parse("myXML.xml")
```

wird der Parser veranlasst, das Dokument zu verarbeiten. Er löst dann die verschiedenen Ereignisse im DocumentHandler aus:

```
startDocument()
endDocument()
```

wird aufgerufen, wenn der Parser anfängt bzw. aufhört das Dokument zu lesen. Hier können z.B. Variablen initialisiert werden.

```
startElement(java.lang.String name, AttributeList atts)
endElement(java.lang.String name)
```

wird aufgerufen, wenn der Parser auf ein Start- bzw. Ende-Tag stößt. Die Methode erhält den Namen des Elements und eine Liste der Attribute. Hier kann jetzt z.B. ein Flag gesetzt werden, dass ein bestimmtes Element erreicht wurde. Die Attribute können mit

```
atts.getValue(int i) //die Position innerhalb der Attributliste
atts.getValue(java.lang.String name) //der Name des Attributes
```

abgefragt werden. Die Klasse AttributeList bietet weitere nützliche Methoden, um mit Attributen zu arbeiten. Diese können in der API-Dokumentation nachgelesen werden.
Stößt der Parser auf Text, wird die Methode

```
characters(char[] ch, int start, int length)
```

aufgerufen. Der SAX-Parser kann alle aufeinanderfolgenden Zeichen liefern, oder er kann sie in mehrere Teil aufteilen. Auch müssen die Zeichen nicht am Anfang des Arrays stehen, daher werden die Startposition und die Länge mit übergeben. Die Methode

```
ignorableWhitespace(char[] ch, int start, int length)
```

wird hingegen aufgerufen, wenn der Parser auf ignorierbare undruckbare Zeichen stößt.

2.3.4.2 DOM

Der DOM Standard wurde vom W³C entwickelt. Auch er ist noch nicht abgeschlossen. Der aktuelle Stand wird unter http://www.w3/org/TR/WD-DOM/level-one-xml veröffentlicht. DOM hat einen objektorientierten Ansatz, was die Verwendung einer objektorientierten Sprach wie JAVA oder C++ nahe legt.

DOM liest das Dokument ein und erstellt daraus eine Baumstruktur, die aus einer Anzahl von Knoten besteht. Dabei bilden die verschiedenen Dokumentbestandteile (Dokument, Elemente, Attribute, Text, Kommentare, Verarbeitungsanweisungen) jeweils einen Knoten.

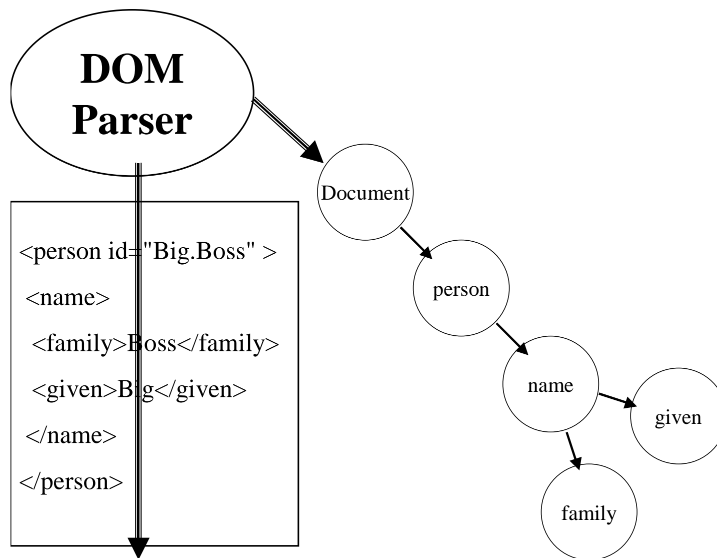


Abbildung 2-7: DOM-Parser

Zuerst wird eine Instanz von einem DOM-Parser erzeugt.

```
DOMParser parser = new com.ibm.xml.parsers.DOMParser();
```

Dieser wird durch den Aufruf

```
parser.parse("myXML.xml");
```

mitgeteilt, welches Dokument zu bearbeiten ist. Daraufhin erstellt der Parser im Speicher die Baumstruktur des Dokuments. Auf diese kann mit

```
Document document = parser.getDocument() ;
```

zugegriffen werden. Alle Dokumentteile basieren auf der Klasse Node. Diese stellt Methoden zur Navigation bereit:

```
getFirstChild(); //liefert den ersten Unterknoten (Kind)
getNextSibling(); //liefert den nächsten Knoten auf gleicher Ebene
                  //(Schwester)
getParentNode(); //liefert den Knoten eine Ebene darüber (Mutter)
getPreviousSibling(); //liefert den vorherigen Knoten auf gleicher
                      //Ebene (Schwester)
getChildren(); //liefert eine NodeList aller Unterknoten
               //(Kinder)
hasChildren() //liefert einen Wert vom Typ boolean, der angibt ob
              //Unterknoten existieren
```

Um den Typ des Knotens zu erfahren, wird die Methode

```
getNodeTyp();
```

verwendet. Sie liefert eine der folgenden Konstanten:

```
0 = Node.NodeTyp.DOCUMENT  //Das Dokument
1 = Node.NodeTyp.ELEMENT  //Element
2 = Node.NodeTyp.ATTRIBUTE  //Attribut
3 = Node.NodeTyp.PI  //Verarbeitungsanweisung
4 = Node.NodeTyp.COMMENT  //Kommentar
5 = Node.NodeTyp.TEXT  //Text
```

2.4 CORBA

Die Common Object Request Broker Architekture (CORBA) wurde von der Object Management Group (OMG), einem Standardisierungsgremium mit mehr als 700 Mitgliedern [CORBA 2000], 1991 in der ersten Version definiert. Ziel war es, eine orts-, plattform- und implementierungsunabhängige Kommunikation zwischen Anwendungen zu erlauben. CORBA liegt mittlerweile in der Version 3.0 vor.

2.4.1 Was ist CORBA?

CORBA ist keine Sprache, sondern eine Middlewareplattform. Mit CORBA kann man nicht programmieren, es stellt nur eine Infrastruktur zur Programmierung von verteilten Systemen dar. Man benötigt noch eine objektorientierte Sprache, die CORBA unterstützt. Das kann z.B. C/C++, JAVA oder Ada sein. Der Vorteil von CORBA ist, dass sich Applikationen in unterschiedlichen Sprachen erstellen lassen, auf verschiedenen Plattformen laufen und trotzdem miteinander kooperieren können. Und dies nicht nur im lokalen Netzwerk, sondern auch über große Strecken.

2.4.2 Object Request Broker (ORB)

Der ORB ist die technische Implementation des Standards CORBA. Objekte können andere Objekte über den ORB ansprechen. Er ist dafür zuständig, das angeforderte Serverobjekt zu finden, die Nachricht weiterzuleiten und gegebenenfalls den Rückgabewert wieder an den Client zu übergeben.

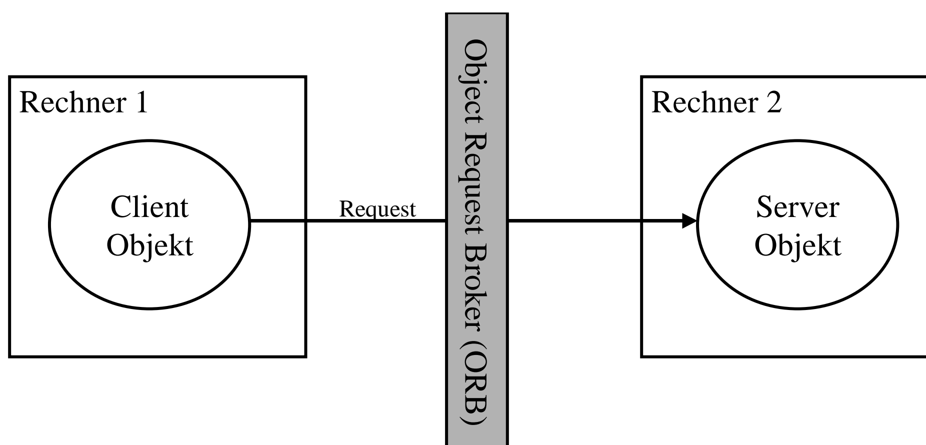


Abbildung 2-8: Der Object Request Broker

Der Client spricht den ORB nicht direkt an. Hierfür existiert ein Stellvertreter des Serverobjektes auf Clientseite. Dieser, auch Client-Stub genannt, kommuniziert mit dem ORB. Er wird durch Kompilieren der IDL erzeugt. Auf Serverseite ist ein Objektadapter, der ebenfalls durch Kompilieren der IDL erzeugt wird, die Schnittstelle zum ORB. Dieser registriert beim ORB die zur Verfügung gestellten Klassen.

2.4.3 Interface Definition Language (IDL)

Die IDL ist eine Schnittstellenbeschreibungssprache. Durch sie werden die Objekte mit ihren Attributen und Methoden beschrieben. Es besteht eine strikte Trennung zwischen der Schnittstellendefinition und deren Implementation. D.h. unabhängig von der Sprache, in der das Objekt implementiert werden soll, muss die Definition in IDL erfolgen. So wird es möglich, dass Client- und Serverobjekte in unterschiedlichen Sprachen erstellt werden können. IDL ist syntaktisch an C++ angelehnt.
Nach der Definition einer Schnittstelle in IDL, muss diese mit einem IDL-Kompiler für die entsprechende Sprache verarbeitet werden. Dieser erzeugt dann je nach Komplexität der IDL mehrere Dateien, die vom Client- bzw. Serverobjekt implementiert werden müssen.

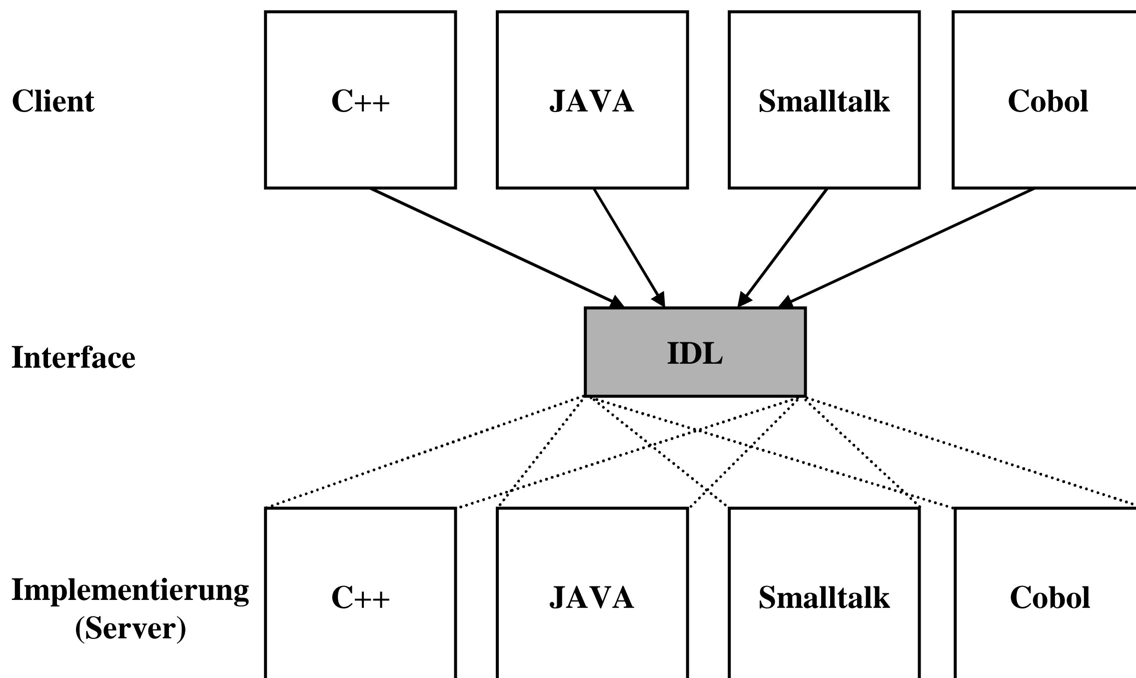


Abbildung 2-9: Die Rolle der IDL [Boger 1999]

Das Erstellen einer CORBA-unterstützen Anwendung setzt sich aus folgenden Schritten zusammen:

- Erstellen einer IDL für alle Objekte, die eine Schnittstelle bereitstellen sollen.
- Kompilieren der IDL.

- Implementierung der generierten Klassen.
- Programmierung des Servers.
- Programmierung des Clients.
- Starten des Servers und des Clients.

2.4.4 Weitere Dienste

Neben der Kernarchitektur existieren eine Reihe weiterer Dienste, die in CORBA integriert sind.

- **Namensdienst.** Identifikation von Objekten über einen Namen (s. auch Voyager Namensdient).

- **Ereignisdienst.** Schnittstellen für Ereignisse.

- **Datenbankdienste.** Schnittstellen für Datenbankoperationen.

2.5 Voyager

ObjectSpace Voyager ist eine zu JAVA vollkommen kompatible Laufzeitumgebung und Bibliothek. Die erste Version von Voyager wurde im Sommer 1997 veröffentlicht. Mittlerweile liegt die Version 3.2 vor, die auch hier verwendet wurde. Voyager bietet viele Techniken für die verteilte Programmierung. Der Voyager ORB unterstützt gleichzeitig CORBA, DCOM, RMI, bietet einen universellen Namensdienst, XML-Parser und eine mobile Objekt-Technologie (mobile Agenten) [Objectspace 1999]. Im folgenden wird nur auf letztere, die mobile Agenten-Technologie, eingegangen.

2.5.1 Die Voyager-Laufzeitumgebung

Um Voyager-Programme auszuführen, muss zuerst die Laufzeitumgebung gestartet werden. Danach stehen alle Funktionen des ORBs zur Verfügung. Dies kann auf zwei Arten erfolgen:

- Aufruf von Voyager über die Befehlszeile. Es muss nur der Port übergeben werden, auf dem Voyager Anfragen entgegennehmen soll.

  ```
  >voyager 8000
  ```

  Und wird beendet durch die Tastenkombination Strg-C.

- Aufruf aus einem Programm:
  Dies erfolgt durch den Aufruf der statischen Methode startup() der Klasse Voyager. Auch hier wird der Port als String übergeben.

  ```
  //...
  Voyager.startup("8000");
  //...
  ```

  Beendet wird Voyager mit dem Aufruf der statischen Methode shutdown()

  ```
  //...
  Voyager.shutdown();
  //...
  ```

2.5.2 Erzeugen von Objekten

Nachdem die Laufzeitumgebung von Voyager gestartet ist, kann Voyager beliebige Objekte auf dem lokalen oder einem entfernten Rechner, auf dem auch die Voyager-Laufzeitumgebung läuft, erzeugen. Diese Objekte werden immer durch einen Stellvertreter repräsentiert, der Proxy genannt wird. Um diesen zu erzeugen, benötigt Voyager ein Interface der Klasse. Dies wird in den folgenden Beispielen immer durch ein vorangestelltes „I" deutlich gemacht. Z.B. Klasse Agent und Interface IAgent. Erzeugt wird das Objekt durch den Aufruf der statischen Methode `create("Klassenname")` in der Klasse `Factory`:

```
//...
IMyAgent iagent = (IMyAgent)Factory.create("MyAgent");
                                              //lokale Erzeugung
IMyAgent iagent = (IMyAgent)Factory.create("MyAgent", "adresse:port");
                                              //entfernte Erzeugung
//...
```

Um einer Klasse beim Erstellen Parameter zu übergeben, muss zuerst ein Array vom Typ Object erzeugt werden, der die Parameter enthält. Dieser wird dann beim Aufruf übergeben:

```
//...
Object[] args = new Object[] {"parameter1", new Integer(42)};
IMyAgent iagent = (IMyAgent)Factory.create("MyAgent", args);
//...
```

Um einen Proxy für ein bestehendes bekanntes Objekt zu bekommen, wird die statische Methode `of(Objekt)` der Klasse `Proxy` verwendet:

```
//...
MyAgent agent = new MyAgent();
IMyAgent iagent = (IMyAgent)Proxy.of(agent);
//...
```

2.5.3 Bewegen von Objekten

Ein mobiler Agent soll sich von Rechner zu Rechner bewegen. Dabei soll er alle gewonnenen Informationen mitnehmen. Der gesamte Zustand des Agenten und alle Referenzen auf andere Objekte müssen dabei erhalten bleiben. Auch ist auf einem entfernten Rechner der Code des Agenten nicht vorhanden. Des Weiteren darf der Agent nicht der Garbage Collection unterliegen, selbst dann nicht, wenn keine lokalen oder entfernet Referenzen auf den Agenten vorhanden sind. Um diesen Anforderungen gerecht zu werden, bietet Voyager die Klasse `Agent`. Mit ihr lässt sich durch den Aufruf

```
//...
Agent.of(Objekt)
//...
```

eine sogenannte Agenten-Facette eines beliebigen Objektes erzeugen. Eine Agenten-Facette ist von sich aus autonom, d.h. sie unterliegt nicht der Garbage Collection und stellt darüber hinaus noch die Methoden des Interfaces IAgent zur Verfügung:

- `moveTo(String url, String methodenName, [,Object[] args])`
  Bewegt den Agenten zu der angegebenen URL, stellt dort den Zustand des Agenten wieder her und ruft die als Parameter übergebene Methode auf (Callback-Methode). Falls die Callback-Methode Parameter benötigt, können diese in einem Object-Array übergeben werden. Dabei müssen primitive Argumente durch ihre Objekte vertreten werden (Bsp.: `Object[] args = new Object[]{new Integer(42)};`).

- `setAutonomous(Boolean Flag)`
  Setzt den Status des Agenten. Wenn der Agent seine Arbeit verrichtet hat und nicht mehr benötigt wird, wird normalerweise `setAutonomous(false)` aufgerufen. Der Agent ist somit nicht mehr autonom und unterliegt der Garbage Collection.

- `isAutonomous()`
  Liefert true, falls der Agent autonom ist.

- `getHome()`
  Liefert die URL, von der der Agent gestartet ist.

Normalerweise wird eine eigene Klasse für einen Agenten erstellt. Diese muss entweder von der Klasse *Serializable* erben oder deren Interface implementieren. Die Klasse bewegt sich dann selbstständig durch den Aufruf:

```
//...
Agent.of(this).moveTo("URL", "Methode");
//...
```

Dieser Aufruf stoppt normalerweise alle Tätigkeiten des Agenten. Daher sollte nach diesem Aufruf nur noch eine Fehlerbehandlung stattfinden. Um etwas über die Bewegung des Agenten zu erfahren, müssen in der Agenten-Klasse folgende Methoden implementiert werden:

- `preDeparture( String source, String destination )`
  Wird aufgerufen, bevor der Agent sich bewegt. Als Parameter erhält man die URL, von der der Agent startet und die URL des Ziels.

- `preArrival()`

Wird bei der Kopie des Agenten auf dem Zielrechner vor dessen Ankunft aufgerufen.

- `postArrival()`
  Wenn diese Methode aufgerufen wird, ist der Agent auf dem Zielrechner ein richtiges Objekt. Der Agent auf dem Ursprungsrechner ist nicht mehr gültig. Der Aufruf erfolgt direkt vor dem Ausführen der Callback-Methode.

- `postDeparture()`
  Wird bei dem nicht mehr gültigen Agenten am Ursprungsrechner aufgerufen. Alle Nachrichten an den alten Agenten werden jetzt über einen Proxy an den neuen Agenten am Zielrechner weitergeleitet.

Damit entfernte Server den Code über das Netz laden können, muss ein Ressourcen-Server gestartet werden. Dies kann entweder die Applikation selbst oder ein im Netz stehender Server sein. Dieser hält alle im CLASSPATH liegenden Klassen zum Abruf bereit.

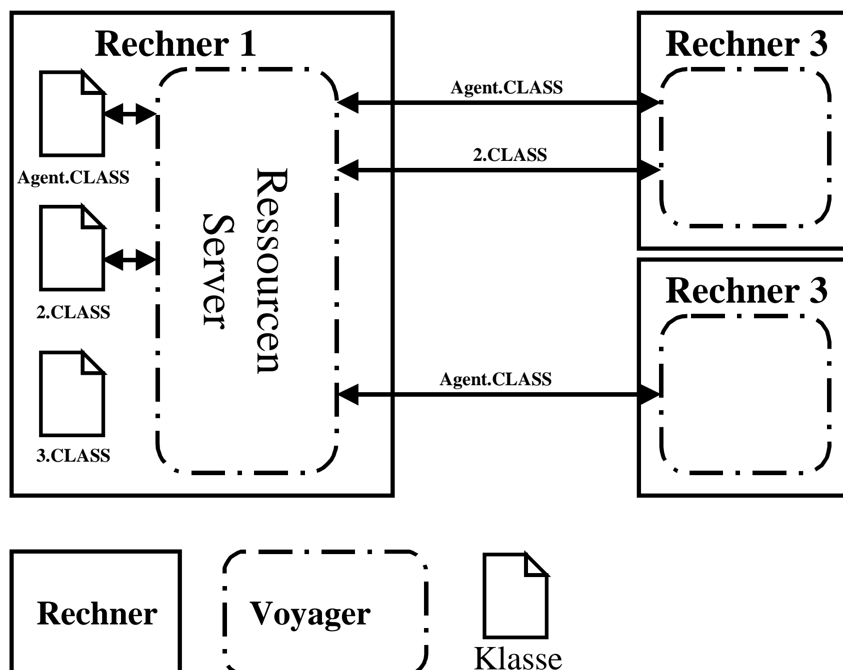


Abbildung 2-10: Der Ressourcen-Server

Ein Ressourcen-Server kann entweder über die Kommandozeile mit

```
Voyager -r
```

oder im Code mit `enableResourceServer()` der statischen Klasse `ClassManager`

```
//...
ClassManager.enableResourceServer();
//...
```

gestartet werden.

Damit ein Programm seinen Code nachladen kann, muss Voyager wissen, wo dieser zu finden ist. Dies geschieht mit der Funktion `setResourceLoader(URLResourceLoader rl)` dynamisch im Code:

```
//...
rl= new URLResourceLoader(new URL("Server:Port"));

Agent.of(this).setResourceLoader(rl);
//...
```

oder statisch über die Kommandozeile mit:

```
voyager 8000 –c http://Server:Port
```

2.5.4 Der erweiterte Nachrichtendienst

2.5.4.1 Dynamischer Nachrichtenaufruf

In Voyager kann die normale JAVA-Syntax zum Versand von Nachrichten verwendete werden. Diese werden standardmäßig synchron versendet. D.h. der Versender blockiert so lange, bis die Nachricht komplett abgearbeitet ist und, falls ein Rückgabewert existiert, dieser zurückgegeben wird. Da in verteilten Systemen die Nachrichtenübermittlung länger dauern kann, macht es Sinn, Nachrichten asynchron zu versenden. D.h. der Aufrufer wird nicht blockiert bis eine Antwort eintrifft. Voyager bietet darüber hinaus noch weitere Möglichkeiten Nachrichten zu versenden:

Synchrone Nachrichten

Synchrone Nachrichten können dynamisch mit dem Aufruf

```
//...
Sync.invoke(Object zielObjekt, String methodenName, Object[] args);
//...
```

versandt werden. Existieren mehrere Methoden mit verschiedenen Parametern, müssen diese mit angegeben werden. Argumente werden mit einem Object-Array übergeben.
Als Rückgabewert erhält man ein Result-Objekt, das den Rückgabewert enthält. Für das Result-Objekt stehen folgende Methoden bereit:

- `isAvailable()`
  Liefert true, falls der Rückgabewert erhalten wurde.

- `readXXX`, `XXX` kann sein `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float`, `Double` oder `Object`
  Liefert den Rückgabewert der Nachricht und blockiert so lange bis entweder der Rückgabewert vorliegt oder ein bestimmtes Zeitintervall (Timeout) abgelaufen ist. Bei einem Timeout wird eine Timeout Exception ausgegeben.

- `isException()`
  Wartet auf den Rückgabewert und liefert true, falls er eine Exception enthält.

- `getException()`
  Wartet auf den Rückgabewert und liefert die Exception oder „Null“ falls keine vorliegt.

Beispiel für einen synchronen Aufruf:

```
//...
Rechner rechner = new Rechner();
Result result = Sync.invoke(rechner, “addiere(int,int)”,
  new Object[]{new   Integer(1), new Integer(2)});
int ergebniss = result.readInt();  //ergebniss=3
//...
```

<u>Einwegnachrichten (One-Way message oder fire-and-forget message)</u>

Ist das Ergebnis einer Methode uninteressant oder liefert sie keinen Rückgabewert, kann eine Einwegnachricht verschickt werden. Eine Einwegnachricht ist sehr schnell. Sie liefert keinen Rückgabewert und der Versender blockiert nicht, solange die Nachricht abgearbeitet wird. Sie wird mit der statischen Methode `invoke` der Klasse `OneWay` verschickt. Die Parameter sind die gleichen wie bei einer synchronen Nachricht (`Sync.invoke()`).

```
//...
Result result = OneWay.invoke(rechner, “quit“, null);
//result kann nur Exception beinhalten, nie einen Rückgabewert!
//...
```

<u>Zukünftige Nachrichten (Future Messages)</u>

Eine zukünftige Nachricht ist asynchron und liefert sofort ein Result-Objekt, das ein Platzhalter für den Rückgabewert ist. Mit den Methoden des Result-Objekts kann zu jeder Zeit der Rückgabewert erfragt werden. Der Aufruf erfolgt durch die statische Methode `invoke()` der Klasse `Future`. Die Parameter sind die gleichen wie bei einer synchronen Nachricht (`Sync.invoke()`).

```
//...
Result result = Future.invoke(rechner, "addiere(int,int)",
  new Object[]{new   Integer(1), new Integer(2)});
tuEtwas(); //Hier können jetzt andere Operationen ausgeführt werden
int ergebniss = result.readInt();  //Ergebniss=3
//...
```

Man kann sich auch durch den JAVA „event/listener Mechanismus“ benachrichtigen lassen, dass der Rückgabewert eingetroffen ist. Dafür muss ein `ResultListener` dem Result-Objekt hinzugefügt werden, der die Nachricht `resultReceived()` abfängt.

2.5.4.2 Vielfachsendungen und Veröffentlichen / Abonnieren

In verteilten Systemen ist es praktisch, wenn Objekte untereinander kommunizieren können. So kann z.B. ein Agentensystem alle darin befindlichen Agenten benachrichtigen, dass das System heruntergefahren wird. Diese können dann ihre Aufgaben abschließen und den Rechner verlassen. Oder ein Agent kann sich bei einem Nachrichtendienst einschreiben, damit er nur die Nachrichten erhält, die ihn auch interessieren.
Voyager bietet hierfür eine Raum-Architektur (Space) an. Ein Raum setzt sich aus einem oder mehreren Unterräumen (Subspace) zusammen, die auch auf verschiedenen virtuellen Maschinen oder Rechnern liegen können. Ein Raum wird durch Verbinden von Unterräumen erstellt. Alle Objekte eines bestimmten Typs in einem Raum können gleichzeitig benachrichtigt werden. Nachrichten werden entlang der Verbindungen von einem Unterraum an alle Nachbarn weitergeleitet, die diese ebenfalls an alle Nachbarn weiterleiten. Voyager sorgt dafür, das jeder Unterraum die Nachricht nur einmal bekommt. Erst wenn jeder Unterraum die Nachricht empfangen hat, wird sie an die Objekte weitergeleitet, für die diese Nachricht bestimmt ist.

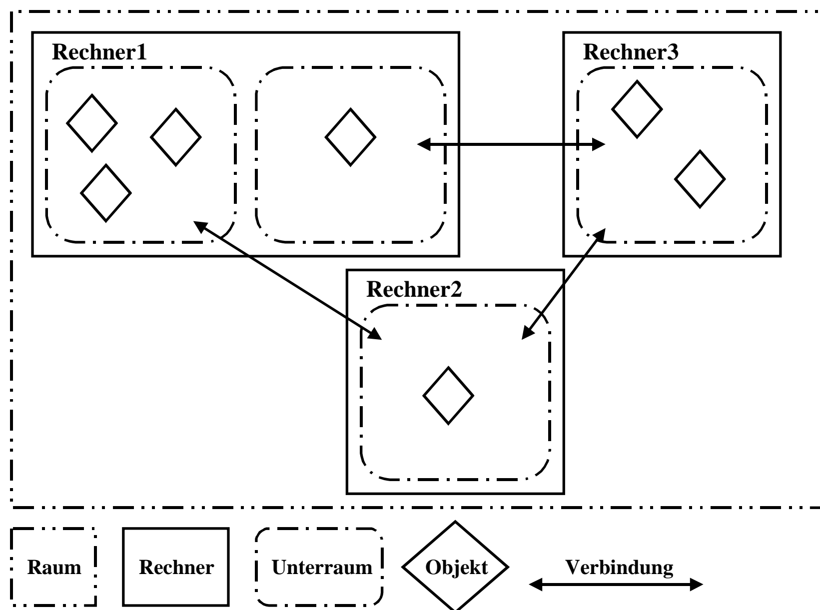


Abbildung 2-11: Raum-Architektur (Space)

Ein Unterraum wird mit `Factory.create()` erzeugt. Der erste Parameter ist die Subspace-Klasse. Der zweite Parameter ist ein String, der sich aus der URL, dem Port und der Unterraumbezeichnung zusammensetzt:

```
//...
ISubspace unterraum1 = (ISubspace) Factory.create(
  "com.objectspace.voyager.space.Subspace",
    "//localhost:8000/Unterraum1" );
//...
```

Verbunden werden zwei Unterräume mit:

```
//...
unterraum1.connect(unterraum2);
//...
```

Dies ist immer eine bidirektionale Verbindung. D.h. wenn `unterraum1` mit `unterraum2` verbunden ist, muss `unterraum2` nicht mit `unterraum1` verbunden werden.

Um ein Objekt einem Unterraum hinzuzufügen, wird die Methode `add(Object object)` der Klasse `Subspace` verwendet.

```
//...
IMyAgent iagent = (IMyAgent)Factory.create("MyAgent");
unterraum1.add(iagent);
//...
```

Vielfachsendungen (Multicast)

Eine Vielfachsendung ist eine Nachricht an alle Objekte gleichen Typs in einem Raum. Dies kann auf zwei Arten erfolgen:

- `Multicast.invoke(ISubspace unterraum, String methodenName, Object[] args, String klassenname)`
  Schickt eine Einwegnachricht an alle Objekte, in dem angegebenen Raum, die eine Instanz der angegebenen Klasse oder des Interfaces sind.

- `getMulticastProxy(String klassenname)`
  Liefert einen Vielfachsendung-Stellvertreter (Multicast-Proxy), der typkompatibel mit der angegebenen Klasse oder des Interfaces ist. Eine Nachricht, die an den Stellvertreter geschickt wird, wird somit an alle Objekte des gleichen Typs verschickt.

Bsp.:

```
//...
Multicast.invoke(unterraum1, ″shutdown″, null,  ‶IMyAgent″);
//...
```

oder

```
//...
IMyAgent iagent = (IMyAgent)unterraum1.getMulticastProxy(‶IMyAgent‶);
iagent.shutdown();
//...
```

Veröffentlichen und Abonnieren (Publish/Subscribe)

Dies funktioniert ähnlich einer Newsgroup. Nachrichten sind in Themenbereiche (Topics) untergliedert. Ein Interessent kann sich nun zu einem oder mehreren Themenbereichen anmelden. Er erhält dann nur die Nachrichten, die ihn interessieren. Dies funktioniert in Voyager genauso. Es können verschiedene Themen erstellt werden. Ein Thema kann wiederum ein Unterthema enthalten. Diese werden durch einen Punkt getrennt (news.computer.datenbanken). Ein Objekt kann sich zu einem Themenbereich anmelden, indem es den genauen Namen (news.computer.datenbanken.access erhält alle Nachrichten zu Access-Datenbanken) angibt oder Platzhalter benutzt (news.computer.datenbanken.* erhält alle Nachrichten zum Thema Datenbanken, d.h. auch alle Nachrichten der Unterthemen).

Um ein Objekt bei einem Themenbereich anzumelden, gibt es drei Möglichkeiten:

- Ein Objekt implementiert einen `PublishedEventListener`. Dieser enthält die Methode `publishedEvent(EventObject ereigniss, Topic thema)`, die alle Ereignisse (Nachrichten) abfängt. Diese müssen dann gefiltert und weiterverarbeitet werden.

- Erzeugen einer Instanz der Klasse `Subscriber`. `Subscriber` implementiert `PublishedEventListener` und bietet Methoden für das Abonnieren / Abbestellen von Nachrichten. Wenn eine Instanz der Klasse `Subscriber` einem Unterraum hinzugefügt wird, leitet er alle Nachrichten, die abonniert wurden an den `PublishedEventListener` weiter.

- Erzeugen einer Subscriber-Facette mittels `Factory.create()`, die dann dem Unterraum hinzugefügt wird.

Eine Nachricht wird mit der statischen Methode `invoke()` der Klasse `Publish` veröffentlicht. Als Parameter wird dem Unterraum ein `EventObject` und das Thema übergeben. Ein EventObject ist eine Klasse, die von `java.util.EventObject` erbt und benutzerdefinierte Methoden enthält. Ein Thema wird mit

```
//...
Topic topic = new Topic("thema.unterthema");
//...
```

erzeugt.

Beispiel für ein EventObject :

```
//newsEvent.java

import java.util.EventObject;
public class NewsEvent extends EventObject
{
  String news;

  public NewsEvent( String news )
  {
    super( news );
    this.news = news;
  }

  public NewsEvent( Object source, String news )
  {
    super( source );
    this.news = news;
  }

  //Liefert nur den Namen, der als Parameter übergeben wurde
  public String toString()
```

```
  {
    return “NewsEvent( “ + news + “ )”;
  }
}
```

Beispiel für einen PublishedEventListener:

```
// MyPublishedEventListener.java

import java.io.Serializable;
import java.util.EventObject;
import com.objectspace.voyager.space.publishing.*;

public class MyPublishedEventListener implements PublishedEventListener
{

  public void publishedEvent( EventObject event, Topic topic )
  {
    System.out.println( "Nachricht " + event + " aus Thema " + topic );
  }
}
```

Beispiel für das Abonnieren und Veröffentlichen:

```
//...

final String subscriberclass =
    “com.objectspace.voyager.space.publishing.Subscriber”;
final String subspaceclass =
    “com.objectspace.voyager.space.Subspace”;

//Unterraum erzeugen
ISubspace unterraum1 = (ISubspace) Factory.create(subspaceclass,
    “//localhost:8000/Unterraum1”);
//Facette von Subscriber erzeugen
ISubscriber subscriber1 = (ISubscriber) Factory.create(subscriberclass,
    “//localhost:8000”);
//Neues Thema erstellen (agenten.bauwesen.baustoffe.*)
subscriber1.subscribe(new Topic(“agenten.bauwesen.baustoffe.*”));
//Den PublishedEventListener (s.o.)implementieren
subscriber1.setListener(new MyPublishedEventListener());
//Subscriber dem Unterraum hinzufügen
unterraum1.add(subscriber1);

//Eine Nachricht veröffentlichen (NewsEvent s.o.)
Publish.invoke(unterraum1, new NewsEvent( “Neuigkeiten!”),
    new Topic(“agenten.bauwesen.baustoffe.beton” ));

//...
```

2.5.5 Der Namensdienst

Der Namensdienst ermöglicht es, beliebige Namen mit einem Objekte zu assoziieren, über die dann auf das Objekt zugegriffen werden kann. Dies ermöglicht es z.B. in einer Client/Server-Architektur Programme zu erstellen, die als Dienstleister für andere Anwendungen dienen. Um mit dem Namensdienst von Voyager zu arbeiten, stehen folgende statische Methoden der Klasse `Namespace` zur Verfügung:

- `lookup( String name )`
  Liefert einen Proxy des, mit dem als Parameter übergebenen Namen, verknüpften Objekts. Oder `null`, falls kein Objekt mit dem Namen in Verbindung steht.

- `bind( String name, Object object )`
  Assoziiert den als ersten Parameter übergebenen Namen mit dem als zweiten Parameter übergebenen Objekt.

- `rebind( String name, Object object )`
  Assoziiert den als ersten Parameter übergebenen Namen mit dem als zweiten Parameter übergebenen Objekt. Überschreibt alle vorherigen Assoziationen mit diesem Namen.

- `unbind( String name )`
  Löscht die Assoziation mit diesem Namen.

Bsp.:

```
//...
IMyAgent iagent1 = (IMyAgent)Factory.create("MyAgent");
Namespace.bind("Agent", iagent)
IMyAgent iagent2 = (IMyAgent)Namespace.lookup("Agent");
//...
```

Wird ein Objekt mittels `Factory.create()`erstellt, kann hier gleich eine Verknüpfung mit einem Namen erfolgen. Dazu muss nur noch die Host Adresse der zu assoziierende Namen mit angegeben werden:

```
/...
IMyAgent iagent = (IMyAgent)Factory.create("MyAgent",
     "localhost:8000/Agent");
/...
```

ist äquivalent zu

```
/...
IMyAgent iagent = (IMyAgent)Factory.create("MyAgent",
     "localhost:8000");
```

```
Namespace.bind("Agent", iagent);
/...
```

3 Konzeption des Agentenprojekts

Die Ausgangssituation

Das vorhandenen VAMOS-System bezieht seine Daten aus einer MS Access-Datenbank. Um die Daten auf dem laufenden Stand der Technik zu halten, müssen die Daten ständig aktualisiert werden. Dies geschieht zur Zeit entweder durch manuelle Eingabe der Daten oder durch einen Import aus einer anderen Datei.

Zielsetzung

Es ist ein Programmmodul zu entwickeln, dass das Aktualisieren der Datenbank erleichtert. Hierfür soll aus dem Bauteileditor heraus das Modul aufgerufen werden können, das dann im Hintergrund die Daten aktualisiert.

Anforderungen

- Das Programmmodul muss sich aus einer anderen Anwendung, die auch in einer anderen Programmiersprache erstellt sein kann, heraus aufrufen lassen.

- Die Beschaffung der Daten soll im Hintergrund erfolgen. D.h. der Anwender soll nach Eingabe der erforderlichen Daten uneingeschränkt mit seiner Arbeit fortfahren können, ohne sich um den weiteren Ablauf kümmern zu müssen.

- Die Aktualisierung der Daten soll über das Internet erfolgen. Der Anwender soll dabei nicht die ganze Zeit mit dem Internet verbunden sein.

- Die Übertragung der Daten in die bestehende Datenbank soll auf einfache Weise erfolgen.

- Es ist ein Konzept für das Bereitstellen der Daten zu erstellen. D.h. es ist eine Grundlage zu schaffen, um aktuelle Daten anzubieten. Hierbei ist die Anbindung an bereits bestehende Datenbanken zu berücksichtigen.

- Beide Programmteile sollen einfach und komfortabel über Fenster bzw. Dialoge verwaltet und bedient werden können.

3.1 Generelles Konzept

Für diese Anwendung bietet sich das Konzept der mobilen Softwareagenten an. Dieses hat den entscheidenden Vorteil, dass der Anwender nach Abreise des Agenten wieder offline weiterarbeiten kann. Die Wahl fiel auf ObjectSpace Voyager, da diese Agentenumgebung ebenso einen CORBA Orb als auch Unterstützung für XML bietet. Daraus resultierte auch die Programmiersprache. Voyager wird nur für JAVA angeboten.
Da der Agent nicht weiß, auf welche Datenbank er trifft, und um die Datenbankabfrage für den Agent möglichst transparent zu halten, werden zwei Programmteile erstellt. Zum einen der Agent, der die Suchkriterien des Anwenders entgegennimmt, und zum anderen ein Agentenserver, der den eintreffenden Agenten mit den entsprechenden Informationen versorgt.

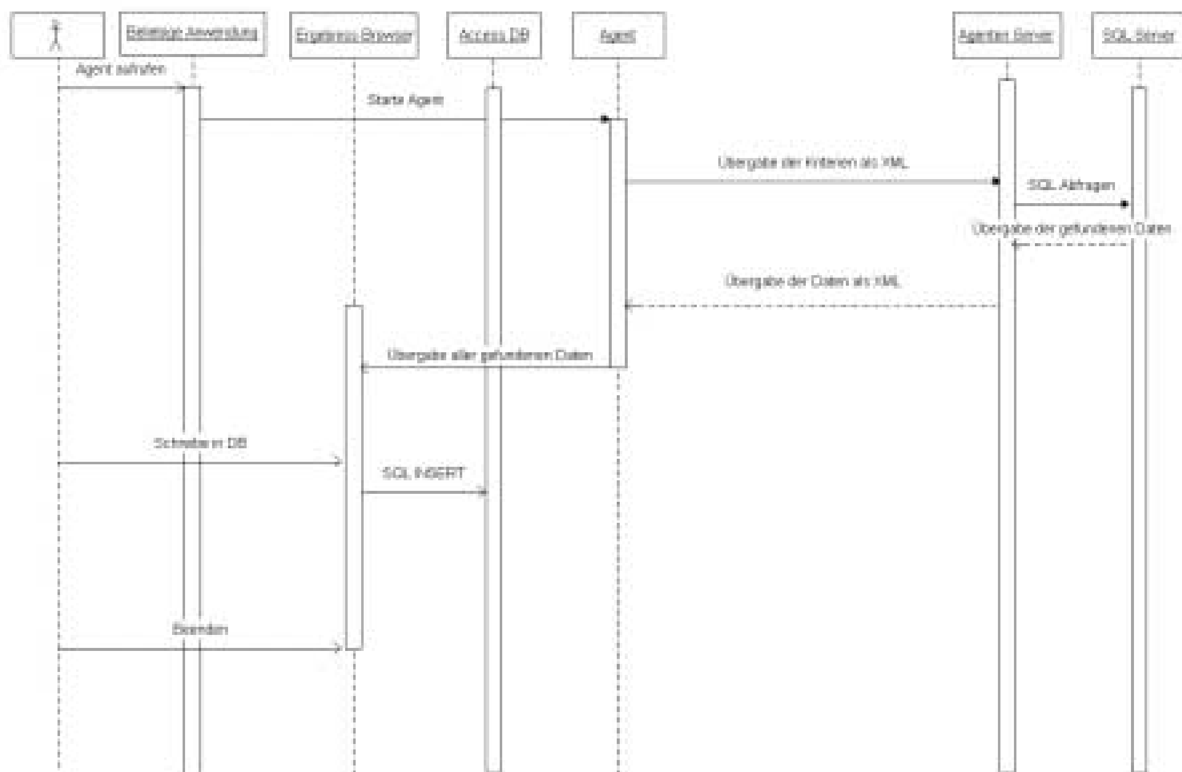


Abbildung 3-1: Übersicht der Kommunikation der Objekte

3.2 Die Kommunikation

Die Kommunikation lässt sich in zwei Bereiche aufteilen.

- **Agent⇔Server.** Am Zielort angekommen, schickt der Agent die Suchkriterien an den Server. Dieser hat dann die Möglichkeit, auf die Anfrage zu antworten.

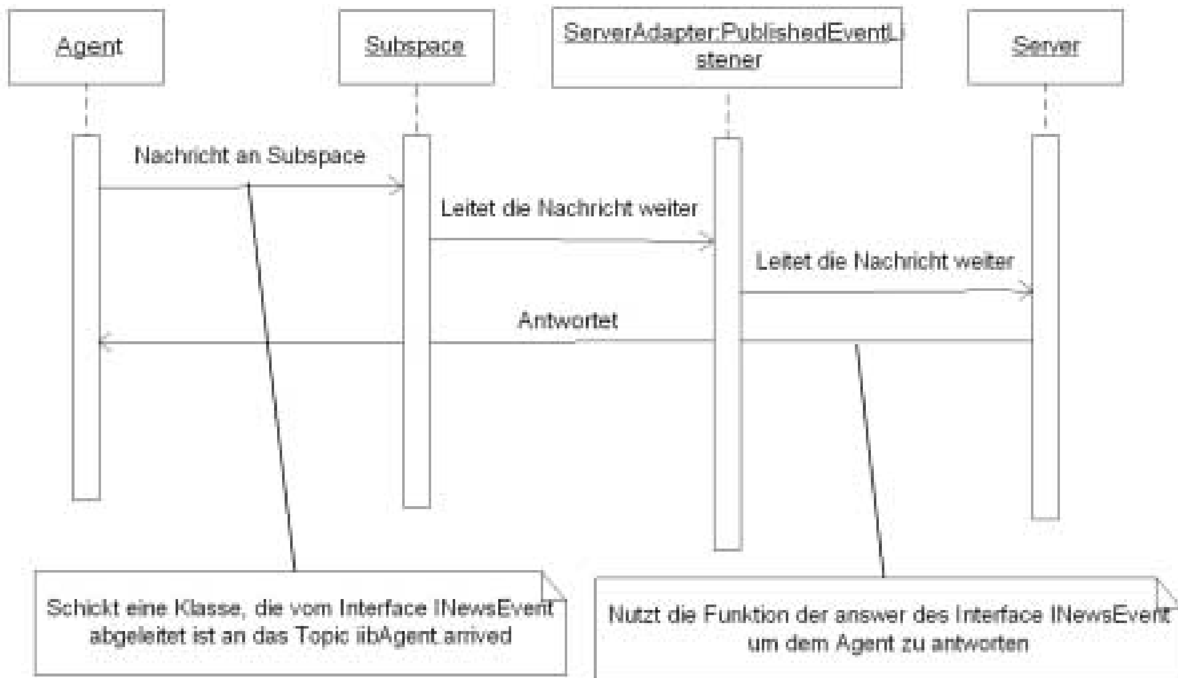
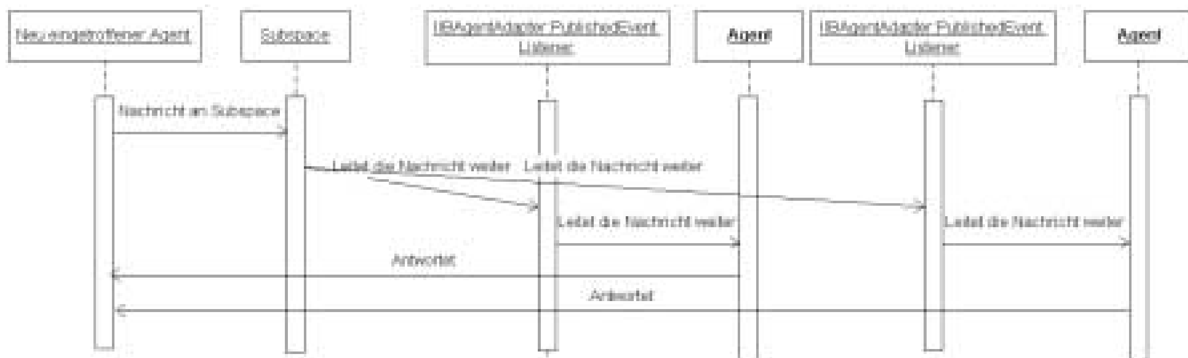


Abbildung 3-2: Kommunikation Agent <-> Server

- **Agent⇔Agent.** Trifft ein Agent auf dem Server ein, kann er einen Nachrichtenkanal abonnieren, auf dem er Anfragen von anderen Agenten annehmen kann. Dies ermöglicht es, anderen Agenten zu antworten und diese gegebenenfalls mit Daten von bereits besuchten Reisezielen zu versorgen. Somit ist eine Interaktion der Agenten untereinander möglich. Daher braucht z.B. ein Reiseziel, das ein anderer Agent vor kurzem besucht und dies mit den selben Suchkriterien befragt hat, nicht noch einmal besucht zu werden. Ebenso kann der Agent in genau diesem Nachrichtenkanal auch seine Suchanfrage veröffentlichen. Diese Schnittstelle ist vorhanden, jedoch nicht implementiert!



Es werden immer nur die Kriterien im XML-Format übergeben. Der Agent hat sich nicht um die Umsetzung der Suche zu kümmern. D.h es ist Aufgabe des Agentenservers, die Daten in entsprechende SQL-Statements umzusetzen und an den Agenten zurückzuliefern.

3.2.1 Das Interface INewsEvent

Entscheidendes Element bei der Kommunikation zwischen Server und Agent bzw. Agent und Agent ist das Interface *INewsEvent.* Ein Agent, der eine Suchanfrage stellen will, muss eine Klasse implementieren, die vom Interface *INewsEvent* abgeleitet ist. Diese beinhaltet die Suchkriterien im XML-Format und bietet eine Funktion zum Antworten.

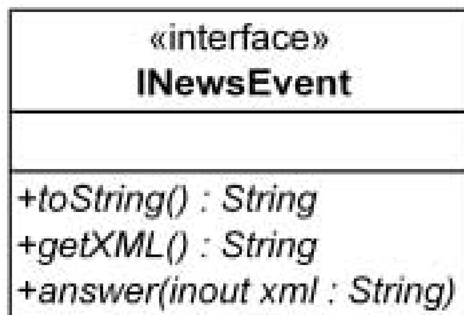


Abbildung 3-3: Das Interface INewsEvent

3.3 Der Datenaustausch

Der Agent und der Agentenserver müssen in irgendeiner Art miteinander kommunizieren. Bei dieser Kommunikation übergibt der Agent die Suchkriterien an den Agentenserver und erhält die Ergebnisse der Suche vom Agentenserver zurück. Beide Seiten müssen die erhaltenen Daten in irgendeiner Weise interpretieren, daher muss hier ein geeignetes Format gefunden werden, das einerseits leicht zu interpretieren ist und andererseits einfach um beliebige Elemente zu erweitern ist. Hierfür bietet sich XML an, das die Daten in einer Textdatei bzw. als String übergibt. Es muss daher kein geeignetes Format gefunden werden, das beide Seiten lesen können. Es muss nur noch dafür gesorgt werden, dass die XML-Datei entsprechend interpretiert wird. Selbst wenn der Agent die XML-Datei nicht interpretieren kann, könnte er sie immer noch als Datei ausgeben und der Anwender sie entsprechend anpassen oder auswerten. Für das einfache Navigieren und Validieren in XML-Datein stehen verschiedene Parser zur Verfügung. Im Anwendungsbeispiel kam der IBM XML4J Parser zum Einsatz, da dieser frei verfügbar ist und die Standards SAX und DOM einhält.

3.3.1 Übergabe der Suchkriterien

Für die Übergabe der Suchkriterien gibt es folgende Voraussetzungen:

- **Boolesche Operatoren**. Ein vorangestelltes Plus („+“) (Boolesches AND) bedeutet, dass dieses Suchkriterium vorkommen muss. Ein vorangestelltes Minus („-„) (Boolesches

AND NOT) signalisiert, dass dieses Kriterium nicht vorkommen darf. Standardmäßig werden die Kriterien mit „oder“ (Boolesches OR) verknüpft.

„Beton –LC“ Sucht nach Informationen über alle Betons außer LC (Leichtbeton).
„Beton +LC“ Sucht nach Informationen über Beton, in denen LC (Leichtbeton) vorkommen muss.

- **Einschränkung der Suche.** Die Suche lässt sich durch weitere Kriterien einschränken. So lassen sich numerische Vergleiche mit Größer- („>“), Kleiner- („<“) und Gleichheitszeichen („= „) anstellen. Auch lässt sich mit dem Gleichheitszeichen nach bestimmten Zeichenfolgen suchen.

  „Beton +dichte>1200“ Sucht alle Informationen über Beton mit einer Dichte, die größer als 1200 ist.
  „Fenster Hersteller=Zenker“ Sucht nach Informationen über Fenster der Firma Zenker.

- Durch Vergabe eines eindeutigen Namens für eine Suchanfrage lassen sich mehrere Suchanfragen zusammenfassen, die später wieder einzeln ausgewertet werden können. So muss sich der Agent nur einmal auf die Reise begeben, um verschiedene Informationen zu sammeln. Diese Funktion ist in der DTD vorgesehen, aber nicht im Agenten implementiert. In der vorhandenen Version lässt sich nur eine Suchanfrage starten, die einen vom System vergebenen Namen erhält.

Der Aufbau der Kriterien-DTD:

```
<?xml version=\"1.0\" ?>
<!DOCTYPE QUERY [
<!ELEMENT QUERY (SEARCH)+>
<!ELEMENT SEARCH (TOKEN)+>
<!ATTLIST SEARCH
      Name NMTOKEN #REQUIRED>
<!ELEMENT TOKEN (OPTION?, RESTRICTION?)>
<!ATTLIST TOKEN
      Name NMTOKEN #REQUIRED
      DBName NMTOKEN #IMPLIED>
<!ELEMENT OPTION (#PCDATA)>
<!ELEMENT RESTRICTION (#PCDATA)>
]>
```

- `QUERY`: Das Hauptelement. Es muss mindestens ein `SEARCH` Element beinhalten.

- `SEARCH`: Die Suchanfrage. Jedes `SEARCH` Element muss einen Namen als Attribut zugewiesen bekommen. Es muss mindesten ein `TOKEN` Element beinhalten.

- `TOKEN:` Ein Suchkriterium. Als Attribute `Name` muss das Suchkriterium angegeben werden. Optional kann noch mit `DBName` eine Spalte in der lokalen Datenbank dem Suchkriterium zugeordnet werden. Das Element kann höchstens ein `OPTION` und/oder ein `RESTRICTION` Element beinhalten.

- `OPTION:` Hier kann der Boolesche Operator für das Kriterium angegeben werden.

- `RESTRICTION:` Hier können die Einschränkungen angegeben werden.

3.3.2 Übergabe der Ergebnisse

Für die Übergabe der Ergebnisse gelten folgende Voraussetzungen:

- Die Ergebnisse müssen einer bestimmten Suche zugeordnet werden können.

- Der Aufbau sollte dem einer Tabelle, mit Zeilen und Spalten, entsprechen. Dies ermöglicht es, die Daten einfach in eine Datenbank zu überführen, oder die Ergebnisse mit Hilfe von Cascading Style Sheets (CSS) in einem Browser darzustellen.

- Für jedes gefundene Element soll der Datentyp mitangegeben werden können.

Der Aufbau der Ergebnis-DTD:

```
<?xml version="1.0" encoding=”ISO-8859-1” ?>
<!DOCTYPE RESULTS [
<!ELEMENT RESULTS (SEARCH)+>
<!ELEMENT SEARCH (RESULTSET)*>
<!ATTLIST SEARCH
      Name NMTOKENS #REQUIRED>
<!ELEMENT RESULTSET (ROW)+>
<!ELEMENT ROW (COLUMN)+>
<!ELEMENT COLUMN (VALUE)>
<!ATTLIST COLUMN
      Name NMTOKENS #IMPLIED
      Type NMTOKEN #IMPLIED>
<!ELEMENT VALUE (#PCDATA)>
]>
```

- `RESULTS:` Das Hauptelement. Es muss mindestens ein `SEARCH` Element beinhalten.

- `SEARCH:` Die Suchanfrage auf die sich das Ergebnis bezieht. Jedes `SEARCH` Element muss einen Namen als Attribut zugewiesen bekommen. Es kann mehrere `RESULTSET` Elemente beinhalten.

- `RESULTSET:` Ein Satz von Ergebnissen. Es muss mindestens ein `ROW` Element einschließen.

- `ROW:` Eine Zeile der Ergebnisse. Muss mindestens ein `COLUMN` Element beinhalten.

- `COLUMN:` Eine Spalte der Ergebnisse. Als Attribute wird der Spaltennamen (`Name`) und der Datentyp (`Type`) angegeben. Enthält genau ein `VALUE` Element.

- `VALUE:` Das Ergebnis

3.4 Der Agent

3.4.1 Bereitstellen des Agentendienstes

Um anderen Anwendungen den Agentendienst zur Verfügung zu stellen, wurde eine CORBA-Schnittstelle implementiert. Dies geschieht über eine gemeinsame IDL-Datei.

```
module iibAgent
{
  module corba
{
  typedef sequence< string > StringArray; // unbounded

  interface AgentCorba
  {
    void start(in string searchstring, in StringArray locations, in string
      voyagerStartUpPort);
    void startUI();
  };
};
};
```

Diese muss dann noch für die entsprechende Programmiersprache kompiliert werden (z.B. mit idl2java oder idl2cpp). Die generierten Dateien werden dann in die Anwendung implementiert und können so den Agent starten.

Der Agent stellt zwei Aufrufe zur Verfügung:

```
void start(in string searchstring, in StringArray locations,
      in string voyagerStartUpPort);
```

Startet den Agenten ohne grafisches Interface. Es müssen die Suchkriterien (`searchstring`) als String, die Reiseziele (`locations`) als String Array, und der Port, (`voyagerStartUpPort`) auf dem Voyager gestartet werden soll, übergeben werden. Der Agent startet sofort und meldet sich erst wieder, wenn er von seinem Auftrag zurückgekehrt ist.

```
void startUI();
```

Startet das grafische Interface (Abbildung 3-4) des Agenten. Hier können dann komfortabel die Suchkriterien, die Reiseziele und die Optionen eingegeben werden.

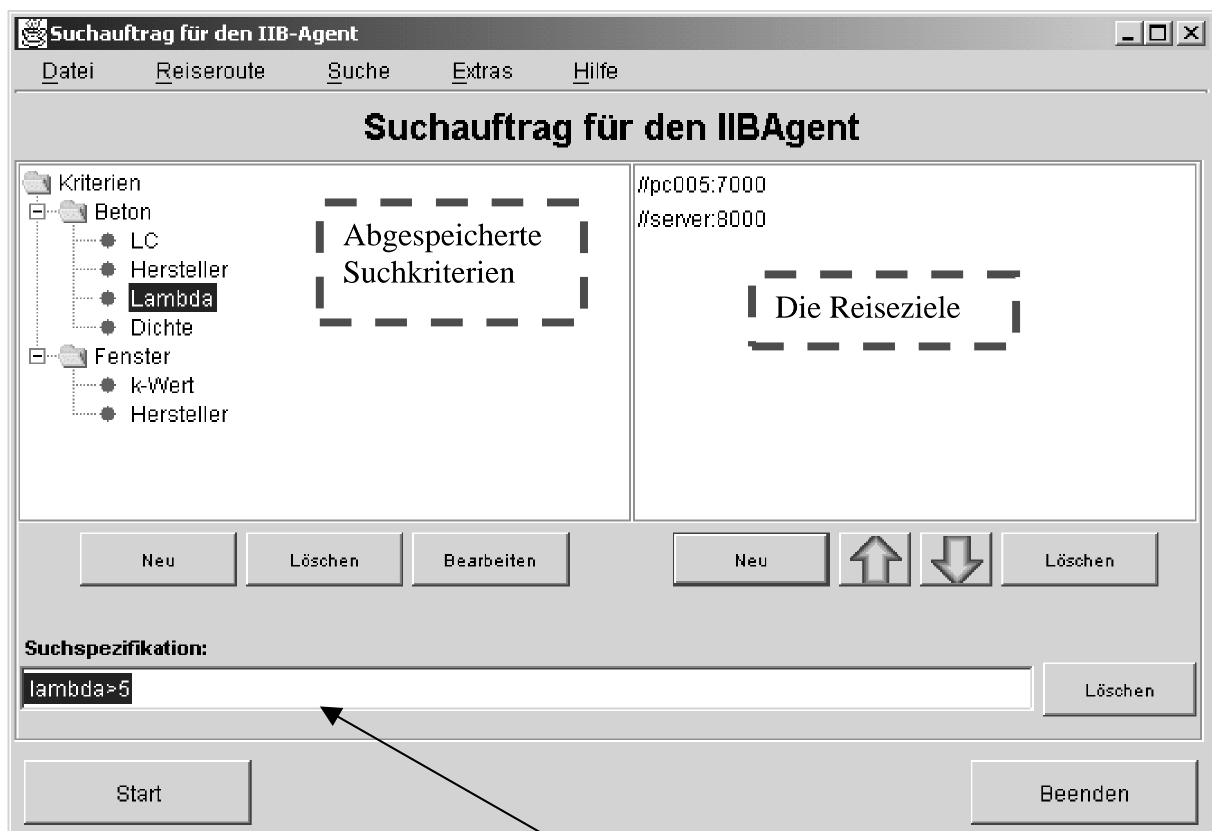


Abbildung 3-4: Das Grafische Interface des Agenten

Das grafische Interface kann auch über die Kommandozeile gestartet werden. Um die Eingabe der Suchkriterien zur erleichtern, ist es möglich, Kriterien zu klassifizieren und abzuspeichern. Zur Darstellung der abgespeicherten Kriterien dient der *Tree View* auf der linken Seite. Hier können beliebige Gruppen angelegt werden, die dann mit Suchkriterien gefüllt werden können. Ein Doppelklick auf ein Element trägt bei der Suchspezifikation die entsprechenden Kriterien ein. Ein Doppelklick auf eine Gruppe trägt alle Kriterien der untergeordneten Elemente ein.

Auf der rechten Seite befindet sich ein *List View,* der die Reiseziele des Agenten anzeigt. Hier lassen sich neue Reiseziele anlegen bzw. die Reihenfolge verändern. Auch die Reiseziele können in einer Datei gespeichert werden.
Standardmäßig sucht das Programm beim Start nach der Datei *default.tre* für die Suchkriterien und *default.trp* für die Reiseziele. Sind diese nicht vorhanden, werden Beispieldaten angelegt.

3.4.2 Die Reise

Nach Betätigen des *Start* Buttons, bzw. beim Aufruf mit Parametern werden zuerst die Suchkriterien in die entsprechende XML-Form gebracht. Hierfür dient die Klasse Parser:

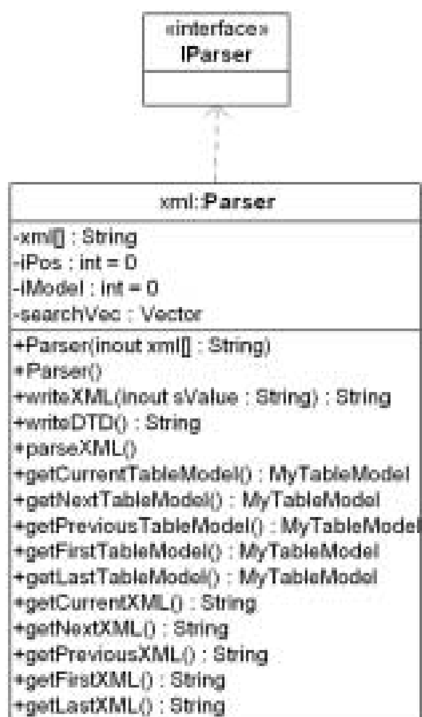


Abbildung 3-5: Die Klasse Parser

Beispiel für einen Suchkriterienstring mit den Kriterien *Beton +dichte<1200 -LC*

```
<?xml version="1.0" ?>
<!DOCTYPE QUERY [
  <!ELEMENT QUERY (SEARCH)+>
  <!ELEMENT SEARCH (TOKEN)+>
  <!ATTLIST SEARCH
    Name NMTOKEN #REQUIRED>
  <!ELEMENT TOKEN (OPTION?, RESTRICTION?)>
  <!ATTLIST TOKEN
        Name NMTOKEN #REQUIRED
        DBName NMTOKEN #IMPLIED>
  <!ELEMENT OPTION (#PCDATA)>
```

```
  <!ELEMENT RESTRICTION (#PCDATA)>
]>

<QUERY>
 <SEARCH Name="MySearch">
  <TOKEN Name="Beton">
  </TOKEN>
  <TOKEN Name="dichte">
    <OPTION>+</OPTION>
    <RESTRICTION>&gt;1200</RESTRICTION>
  </TOKEN>
  <TOKEN Name="LC">
    <OPTION>-</OPTION>
  </TOKEN>
 </SEARCH>
</QUERY>
```

Danach wird Voyager gestartet und der Agent mittels `moveTo` (s. 2.5.3 Bewegen von Objekten) zum ersten Ziel seiner Reise gebracht. Dort angekommen wird die Callback-Methode `atNewLocation()` aufgerufen.

```
public void atNewLocation()
{
  try
  {
    System.out.println( "Arrived at new location, my sweet home = " +
            Agent.of( this ).getHome() );
    timer.startWatch();
    System.out.println("Setting timer to: " + options.getTimeToStay() +
            "ms");

    ISubspace subspace = (ISubspace) Namespace.lookup( trip.currentLocation
            () + "/iibAgentServer" );
    ISubscriber subscriber = (ISubscriber) Factory.create( subscriberclass,
            trip.currentLocation ());
    subscriber.subscribe( new Topic( "iibAgent.Search" ) );
    subscriber.setListener( new IIBAgentAdapter( this ) );
    subspace.add( this );
    subspace.add( subscriber );

    Publish.invoke(subspace, news = new NewsEvent(this, new
            String(this.bSearchXML) ), new Topic("iibAgent.Arrived"));
  }
  catch(Exception ex)
  {
    System.out.println(ex);
  }
}
```

Zuerst wird mit `timer.startWatch()` ein Zeitintervall gesetzt, innerhalb dessen die Ergebnisse eintreffen sollen. Nach Ablauf des Intervalls wird automatisch das nächste Ziel angesteuert.
Über den Voyager Namensdienst (s. 2.5.5 Der Namensdienst) wird mittels `Namespace.lookup` der für den Agenten entsprechenden Subspace (s. 2.5.4 Der erweiterte Nachrichtendienst) ermittelt.
Danach wir mit `subscriber.subscribe()` ein neuer Nachrichtenkanal geöffnet bzw. sich einem bestehenden angeschlossen. Mit `subscriber.setListener` wird festgelegt, welche Klasse eintreffende Nachrichten verarbeitet. Dies ermöglicht die Kommunikation der Agenten untereinander.
Mit `Publish.invoke` wird eine Instanz der Klasse `NewsEvent`, die vom Interface `INewsEvent` abgeleitet ist, an den Agentenserver übermittelt. Diese beinhaltet die Suchkriterien und Funktionen zur Übergabe der Ergebnisse.

Die Klasse `NewsEvent`:

```
package iibAgent;

import java.util.EventObject;
import iibAgentServer.*;
import java.io.*;
import java.util.Vector;

public class NewsEvent extends EventObject implements INewsEvent
{
  private String news;
  private IIBAgent agent;

  public NewsEvent(String news)
  {
    super(news);
    this.news = news;
  }

  public NewsEvent(Object source, String news)
  {
    super(source);
    agent = (IIBAgent)source;
    this.news = news;
  }

  public NewsEvent(IIBAgent agent, String news)
  {
    super(news);
    this.agent = agent;
    this.news = news;
  }

  public String getXML()
```

```
  {
    return news;
  }

  public String toString()
  {
    return "NewsEvent( " + news + " )";
  }

  public void answer (String resultXML)
  {
    agent.addResult(resultXML);
    agent.nextLocation();
  }

}
```

Sind alle Reiseziele abgearbeitet, versucht der Agent, die Heimreise anzutreten. Dies versucht er so lange, bis er erfolgreich zu Hause angekommen ist.

3.4.3 Darstellung der Ergebnisse

Abbildung 3-6: Der Resultbrowser

Zu Hause angekommen wird automatisch der Resultbrowser (Abbildung 3-6) gestartet, der den Import in die Baudatenbank ermöglicht. Im unteren Bereich sind die übermittelten XML-Strings zu sehen und im oberen Bereich die daraus gewonnenen Daten. Die Interpretation der Ergebnisse übernimmt ebenfalls die Klasse Parser (Abbildung 3-5). Um den Import zu erleichtern, werden die Datentypen der Ergebnisse und die der Spalten in der Baudatenbank grafisch angezeigt. Für die grafische Darstellung der Datentypen in den Kombinationsfeldern und Tabellenzellen wurden folgende Klasen erstellt:

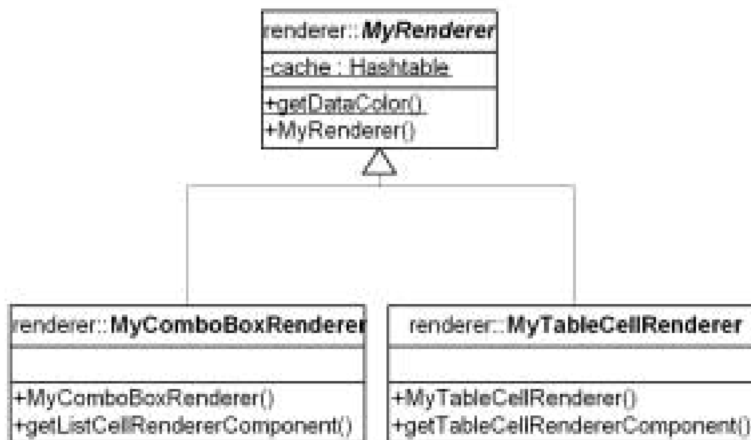


Abbildung 3-7: Klassen für die Darstellung der Datentypen

Die Daten werden mittels einer JDBC/ODBC-Bridge in die Baudatenbank geschrieben. Hierfür muss im System ein ODBC-Treiber eingerichtet werden. Dieser muss dann in der Optionenmaske (Abbildung 3-8) angegeben werden. Desweiteren muss auch ein SQL-Statement angegeben werden, das die Spalten beschreibt, in die die Daten eingetragen werden sollen. Mit *SELECT * FROM Material* werden alle Spalten der Tabelle Material verwendet.

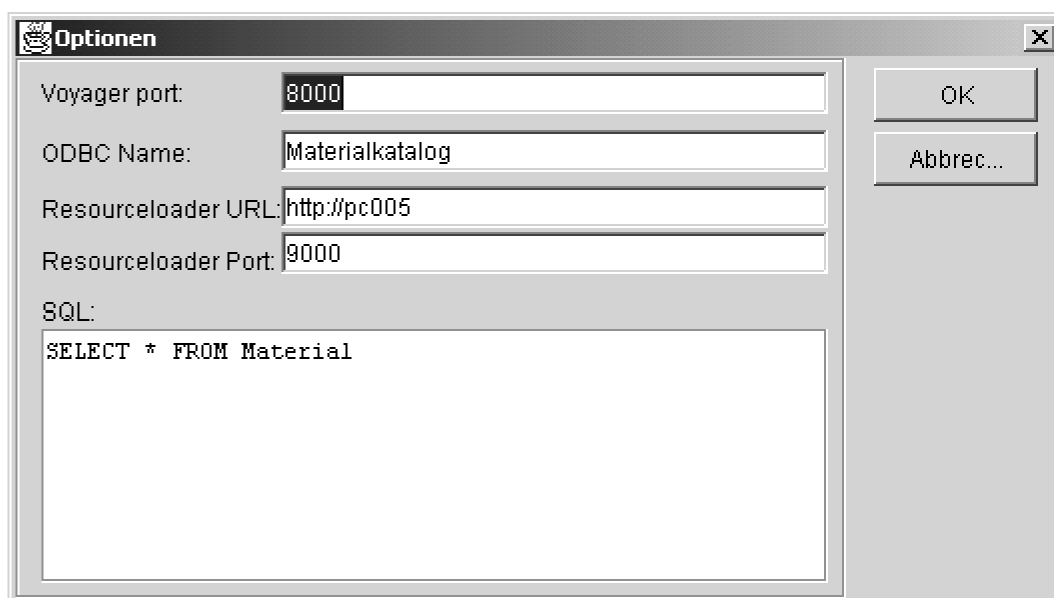


Abbildung 3-8: Die Optionenmaske

3.5 Der Server für ankommende Agenten

Der Agentenserver dient als Host für ankommende Agenten. D.h. er muss eine Ausführungsumgebung für die Agenten bereitstellen und deren Anfragen auswerten sowie die Ergebnisse zurückliefern.

3.5.1 Aufbau der Datenbank

Entscheidendes Kriterium bei der Erstellung des Agentenservers war die Anbindung an eine bestehende Datenbank. Da davon auszugehen ist, dass jeder Hersteller seine Daten in einer eigenen Datenbank pflegt (deren Struktur nicht bekannt ist) wurde versucht, den Aufwand für die Anbindung des Agentenservers möglichst gering zu halten. Dazu müssen folgende drei Tabellen erstellt werden:

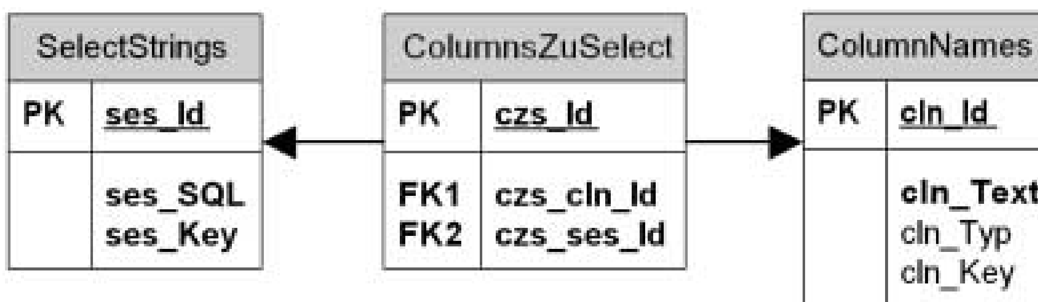


Abbildung 3-9: Aufbau der SQL Server-Tabellen

In diesen Tabellen können nun zerlegte SQL-Strings abgelegt und beliebige Schlüsselwörter zugeordnet werden. Die einfachste Möglichkeit ist es, nur in der Tabelle *SelectStrings* SQL-Statements anzulegen und diesen Schlüsselwörter zuzuordnen. Will man genauer vorgehen, können in der Tabelle *ColumnNames* auch noch den Spalten einer Tabelle Schlüsselwörter zugeordnet werden.

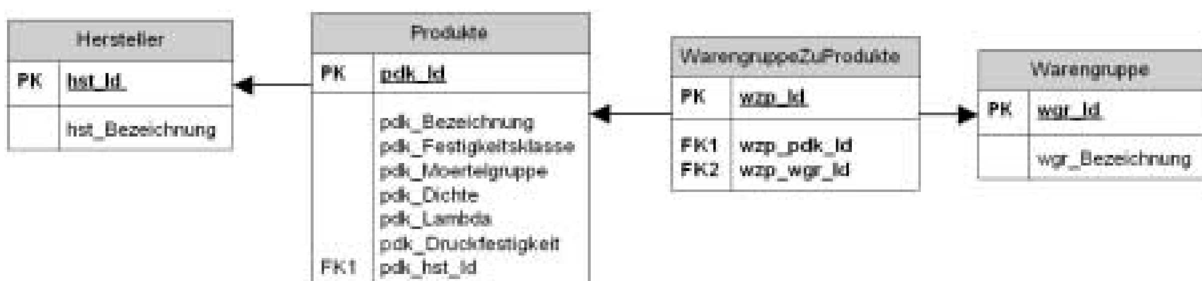


Abbildung 3-10: Beispieldatenbank eines Großhändlers

Beispiel:

Hierfür dient die Beispieldatenbank aus Abbildung 3-10, die auch für das Anwendungsbeispiel verwendet wird. Im einfachsten Fall würde es genügen, in der Tabelle *SelectStrings* folgenden SQL-Teil abzulegen:

```
„SELECT * FROM WarengruppeZuProdukte LEFT OUTER JOIN Warengruppe ON
WarengruppeZuProdukte.wzp_wgr_Id = Warengruppe.wgr_Id RIGHT OUTER JOIN
Produkte ON WarengruppeZuProdukte.wzp_pdk_Id = Produkte.pdk_Id LEFT OUTER
JOIN Hersteller ON Produkte.pdk_hst_Id = Hersteller.hst_Id"
```

Diesem werden dann noch entsprechende Schlüsselwörter wie Ziegel, Beton, etc. zugeordnet.

Den einzelnen Spalten lassen sich dann optional auch noch Schlüsselwörter zuordnen. Z.B. der Spalte *pdk_Lambda* das Schlüsselwort „Lambda“. Dies ermöglicht es, die Abfrage weiter einzuschränken.

Die Anbindung an die Datenbank erfolgt über einen direkten JDBC-Treiber der Firma i-net software. Dieser kann aber gegebenenfalls durch Treiber anderer Firmen bzw. anderer Datenbanken ersetzt werden. Somit lässt sich jede Datenbank für die ein JDBC- bzw. ODBC-Treiber existiert einbinden.

Es lässt sich so mit relativ geringem Aufwand der Agentenserver an jede beliebige Datenbankstruktur anbinden.

3.5.2 Das grafische Interface

Der Server wird ebenfalls über ein grafisches Interface (Abbildung 3-11) bedient. Über die Menüleiste lässt sich der Server starten bzw. beenden. Jede ausgeführte Aktion wird im Logfenster protokolliert. So lässt sich jede Aktion genau mitverfolgen.

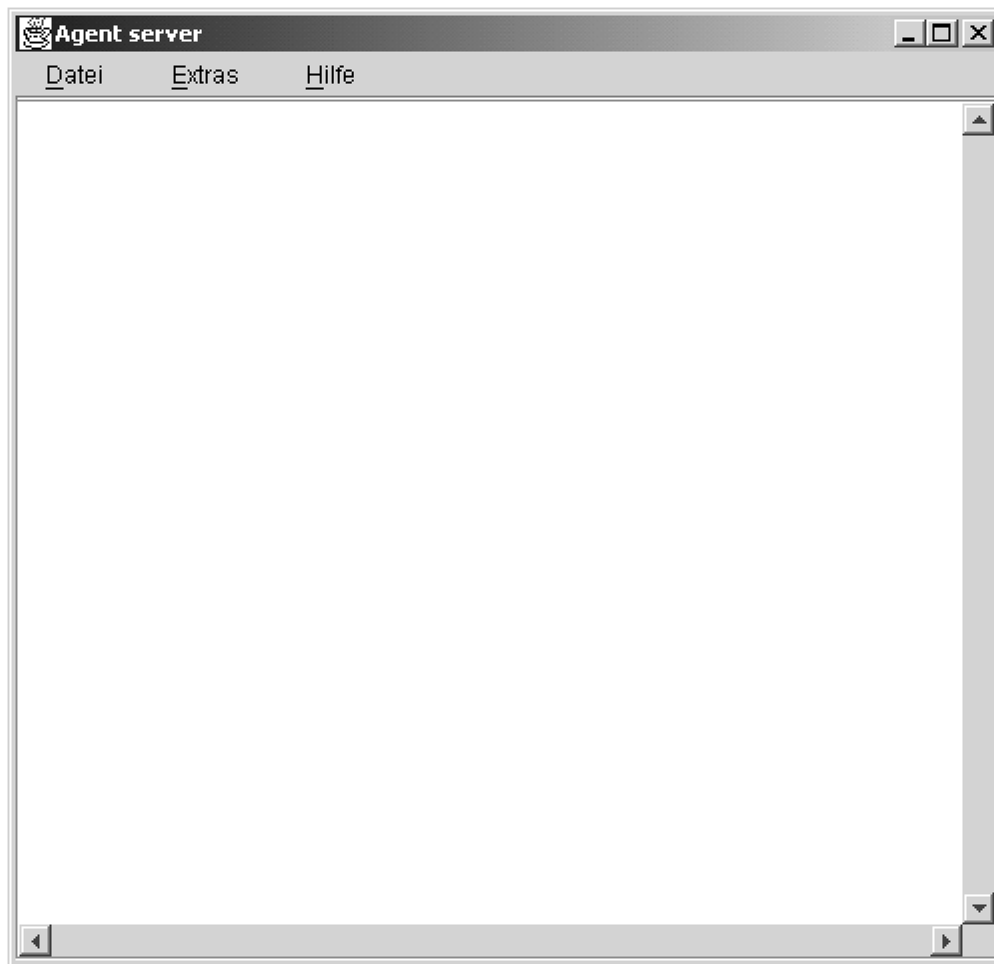
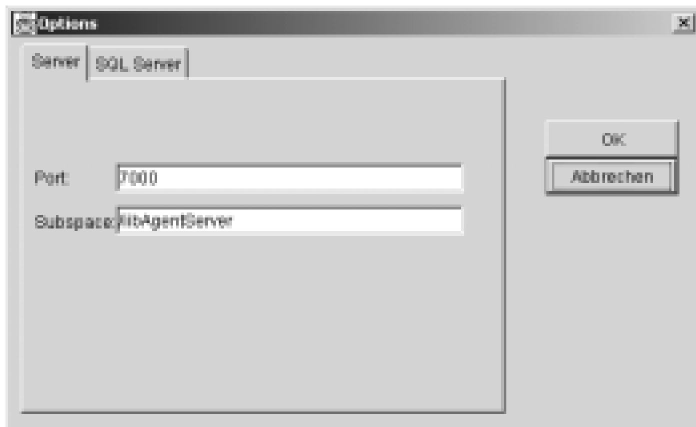


Abbildung 3-11: Der Agentenserver

Die Optionen für den Agentenserver lassen sich ebenfalls über die Optionenmaske einstellen. Dies sind:

- Der Port auf dem der Agentenserver gestartet werden soll.
- Die Bezeichnung für den Subspace.
- Die IP-Adresse des SQL-Servers.
- Der Port des SQL-Servers.
- Die Benutzeranmeldung für den Zugriff auf die Datenbank
- Das dazugehörige Passwort.
- Der Name der Datenbank

Die Optionen lassen sich nur eintragen, solange der Server noch nicht gestartet ist. Sollen die Optionen im laufenden Betrieb geändert werden, muss zuerst der Server angehalten werden. Danach kann er mit den neuen Optionen wieder gestartet werden.



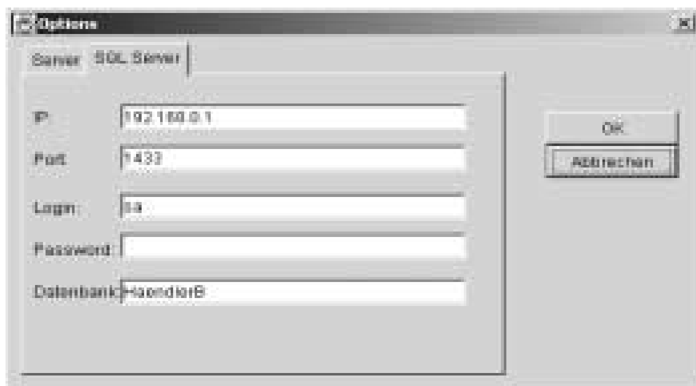


Abbildung 3-12: Die Agentenserver-Optionen

3.5.3 Start des Agentenservers

Der Menüeintrag *Server starten* wird die Funktion *startServer()* aufgerufen, die den Agentendienst startet.

```
void startServer()
  {
    try
    {
      if (!Voyager.isStarted ())
      {
        ClassManager.enableResourceServer();
        Voyager.startup(Options.getPort());
        addEvent("Server on " + Options.getPort() +" started.");

        String serverName = Options.getUrl() + Options.getPort() +
            "/iibAgentServer";

        IServer server = (IServer)Proxy.of(new Server());

        ISubspace subspace = (ISubspace) Factory.create(
             "com.objectspace.voyager.space.Subspace",Options.getUrl() +
               Options.getPort() + Options.getSubspaceName () );
```

```
          addEvent("Subspace " + Options.getUrl() + Options.getPort() +
               Options.getSubspaceName () +" created.");
          ISubscriber subscriber = (ISubscriber) Factory.create(
              subscriberclass, Options.getUrl() + Options.getPort() );
          Namespace.bind( "subscriber", subscriber );
          addEvent("Namespace subscriber added.");
          subscriber.subscribe( new Topic( "iibAgent.Arrived" ) );
          addEvent("Topic iibAgent.Arrived added to subsciber.");
          subscriber.setListener( new ServerAdapter( server ) );
          subspace.add(server);
          subspace.add(subscriber);
          this.mOptions.setEnabled (false);
        }
        else
          addEvent("Server already started.");
      }
      catch(Exception exception)
      {
        addEvent(exception.toString ());
        Voyager.shutdown();
      }
    }
```

Der Ablauf ist folgender:

- Zuerst wird Voyager gestartet.
- Dann wird ein Proxy der Klasse Server erstellt.
- Ein Subspace wird erzeugt.
- Erstellen des Nachrichtenkanals für eintreffende Agenten.
- Zuordnen einer Klasse, die eingegangene Nachrichten verarbeitet (ServerAdapter).
- Der Proxy und der Nachrichtenkanal werden dem Subspace hinzugefügt.

3.5.4 Abfrage der Datenbank

Zuerst werden die Suchkriterien aus dem XML-String extrahiert. Mittels dieser Suchkriterien wird dynamisch ein SQL-Statement generiert:

- Die Tabelle *SelectStrings* wird nach Datensätzen durchsucht, die den Suchkriterien entsprechen. Hierfür muss ein Wort in der Spalte *ses_Keys* mit einem übergebenen Schlüsselwort übereinstimmen.

- Stehen zu diesen Datensätzen Datensätze der Tabelle *ColumnNames* in Beziehung, werden diese als Spaltennamen ausgelesen. Wird ein Suchkriterium weiter eingeschränkt (z.B. lambda<1), kann so ein entsprechender Filter gesetzt werden, indem der zum Schlüsselwort gehörige Spaltenname ausgelesen wird und daraus eine WHERE-Klausel erzeugt wird.

Dieses so generierte Statement wird ausgeführt und eventuell gefundene Datensätze in das XML-Format überführt.

3.6 Klassifikation des Agentensystems

3.6.1 Agent

Der Agent ist mobil. Er besucht während seines Daseins mehrerer Server und kehrt wieder nach Hause zurück.

Primär kommuniziert der Agent mit dem Agentenserver, der nach der Klassifikation ein stationärer Agent ist. Auch bietet der Agent Schnittstellen zur Kommunikation mit anderen Agenten. Daher ist er als Multiagentensystem anzusehen.

Der Agent nimmt die Suchanfragen vom Anwender entgegen und beschafft die gesuchten Daten autonom, durch Anfragen auf mehreren Servern. Er besitzt daher einen gewissen Grad an Intelligenz. Der Grad der Intelligenz lässt sich z.B. dadurch steigern, dass der Agent vor Ort die erhaltenen Daten nach bestimmten Kriterien filtert und unrelevante Daten aussortiert bzw. die Reise abbricht, wenn die gesuchten Daten gefunden wurden. Auch kann er sich Reiseziele merken, an denen er häufig zutreffende Daten erhalten hat und diese beim nächsten Aufruf zuerst ansteuern.

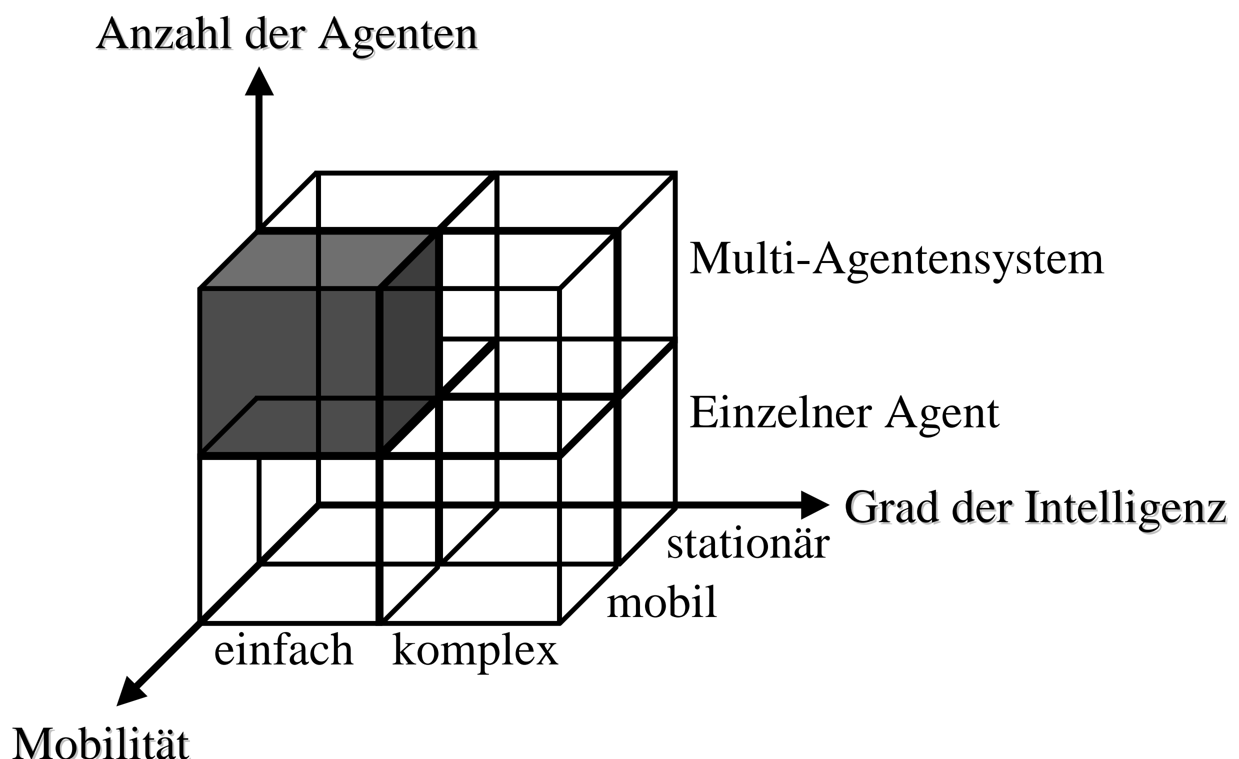


Abbildung 3-13: Klassifikation des Agenten

3.6.2 Agentenserver

Der Agentenserver ist wie der Agent zu klassifizieren, mit dem Unterschied, dass er nur stationär ist, da er nie seinen Host verlässt, sondern auf ankommende Agenten wartet.

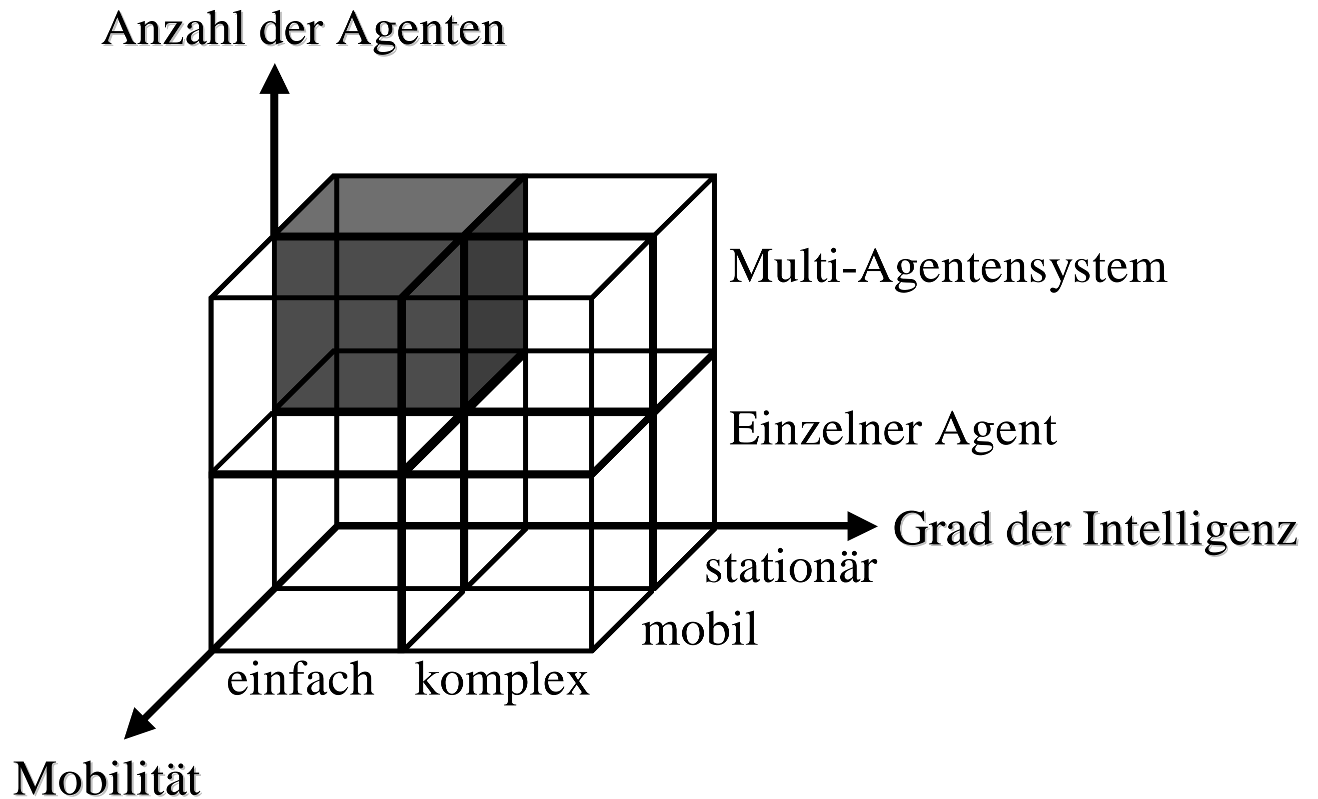


Abbildung 3-14: Klassifikation des Agentenserver

4 Anwendungsbeispiel

Als Beispiel sollen zwei Rechner (Rechner1, Server) dienen. Auf dem Server laufen die Datenbank, der CORBA Orb und der Agentenserver. Rechner1 stellt einen beliebigen Rechner dar, von dem aus der Agent losgeschickt werden soll.

Die Simulation mehrerer Server kann auch auf einen einzigen Rechner erfolgen. Hier müssen dann nur verschiedene Port-Nummern verwendet werden.

4.1 Start der Agentenserver

Gestartet werden zwei Agentenserver. Einer auf Rechner1 und der andere auf dem Server. Beide werden jeweils auf Port 7000 ausgeführt.

Gestartet wird der Agentenserver in der Kommandozeile mit dem Befehl

```
java iibAgentServer.AgentServerUI
```

oder über die gleichnamige Stapelverarbeitungsdatei `iibAgentServer.bat`. Gegebenenfalls können jetzt noch die Optionen angepasst werden. Der Agentenserver-Dienst wird über den Menüeintrag *Server starten* im Menü *Datei* gestartet. Im Logfenster werden die ausgeführten Schritte angezeigt.

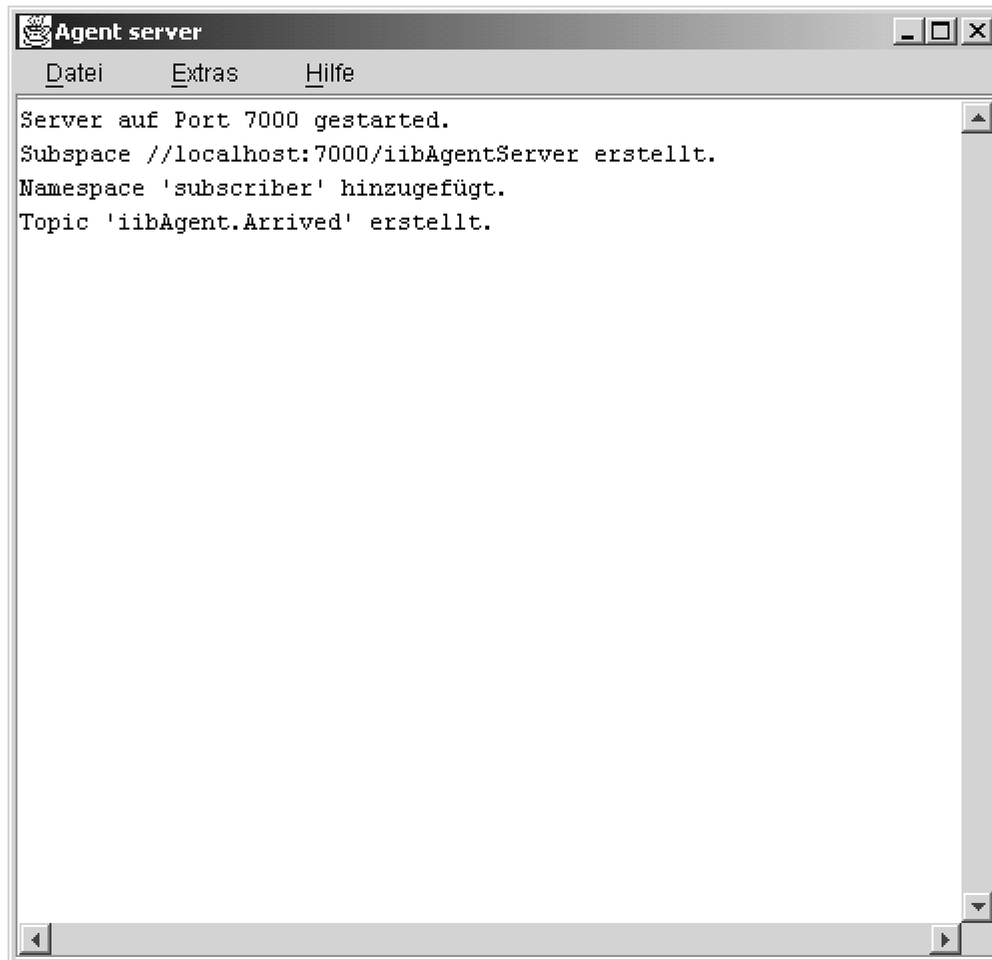


Abbildung 4-1: Start des Agenten-Dienstes

Der Server wartet nun auf eintreffende Agenten.

4.2 Start des Ressourcenservers

Damit Voyager die Klassen über das Netz laden kann, muss ein Ressourcenserver, der die benötigten Klassen bereithält, gestartet werden. Dies kann normalerweise auch die Anwendung selbst, respektive der Agent sein. In der aktuellen Voyager-Version haben jedoch mobile Objekte Probleme, nach Hause zurückzukehren. Um dies zu umgehen, kann ein expliziter Ressourcenserver gestartet werden. Dieser kann sich irgendwo im Netz befinden und muss Zugriff auf die benötigten Klassen haben.

Der Ressourcenserver wird auf Rechner1 mit Port 9000 gestartet:

```
voyager 9000 -r
```

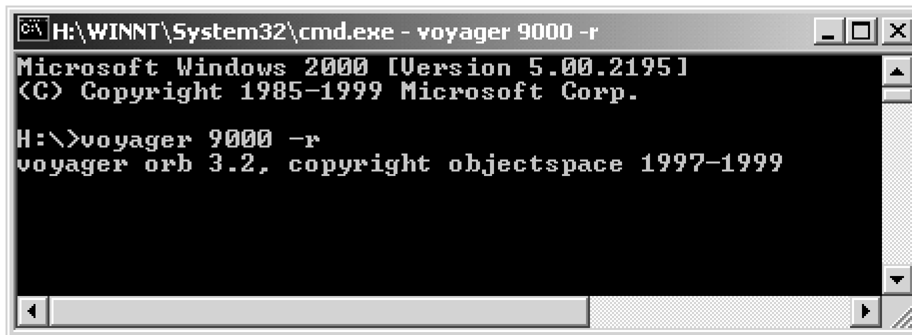


Abbildung 4-2: Der Ressourcenserver

Da sich die *iibAgent.jar* Datei im CLASSPATH befindet, hat der Ressourcenserver somit Zugriff auf die benötigten Klassen.

4.3 Start des AgentCorbaServer

Zuerst muss ein CORBA ORB zur Verfügung stehen. Falls er nicht mit dem NT-Dienst gestartet wurde, kann dies mit

```
osagent
```

in der Kommandozeile nachgeholt werden.

Damit andere Anwendungen auf den Agenten zugreifen können, muss ein "AgentenCORBAServer" Objekt in den Orb geladen werden. Um ein Objekt in den Orb zu laden, ist das Tool „vbj“, das mit Visibroker installiert wird, nötig. Die Handhabung ist ähnlich der JAVA-Runtime. Die `AgentCorbaServer` Klasse befindet sich im Paket `iibAgent.corba` und wird mit

```
vbj iibAgent.corba.AgentCorbaServer
```

in den Orb geladen.

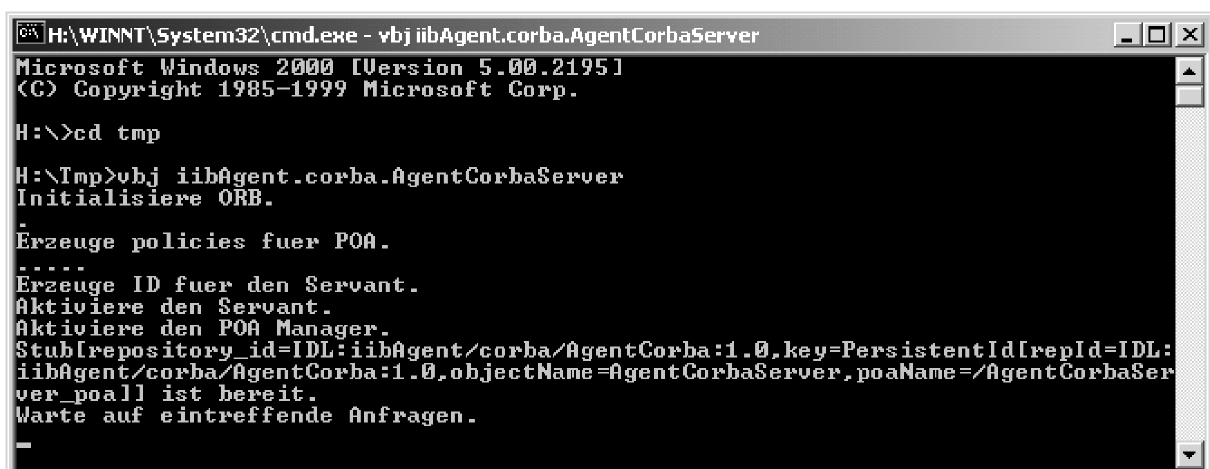


Abbildung 4-3: Start des AgentCorbaServer

Der `AgentCorbaServer` wartet jetzt auf eintreffende Anfragen.

4.4 Start des Agenten

Als ein Beispiel für das VAMOS-System, das den Agenten aufruft, dient das Programm *Bauteileditor.* Es kann einfach per Doppelklick auf die Datei *Bauteileditor.exe* gestartet werden.

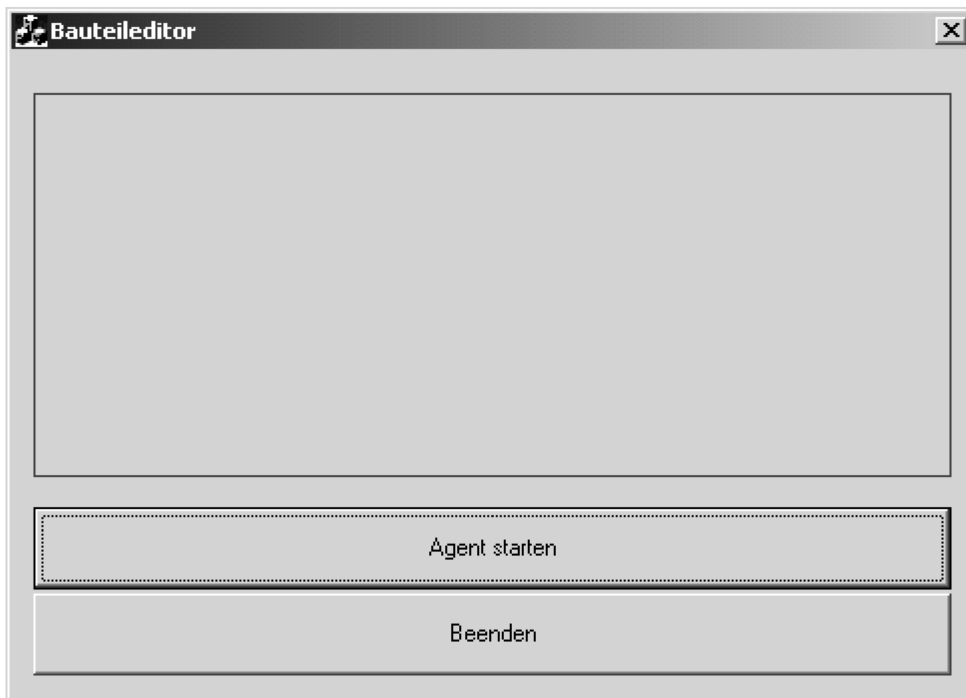


Abbildung 4-4: Der Bauteileditor

Über den Button *Agent starten* wird das grafische Interface des Agenten aufgerufen. Dies kann alternativ in der Kommandozeile erfolgen:

```
java iibAgent.IIBAgentUI
```

Abbildung 4-5: Das grafische Interface des Agenten

Hier lassen sich jetzt beliebige Suchkriterien eingeben. Als Beispiel dient:

Fenster Ziegelmauerwerk +kwert<1

D.h. der Agent soll nach „Ziegelmauerwerk" und „Fenstern" suchen, die einen k-Wert kleiner 1 besitzen. Als Reiseroute wird zuerst der lokale Server angelaufen. Danach wird der Server angesteuert.

In den Optionen müssen nun die URL und der Port des Ressourcenservers eingetragen werden. Als ODBC-Treiber wird der Name des eingerichteten ODBC-Treibers angegeben. Im Feld SQL lässt sich ein SQL-String angeben, der die Felder beinhaltet, in die Daten geschrieben werden können/sollen.

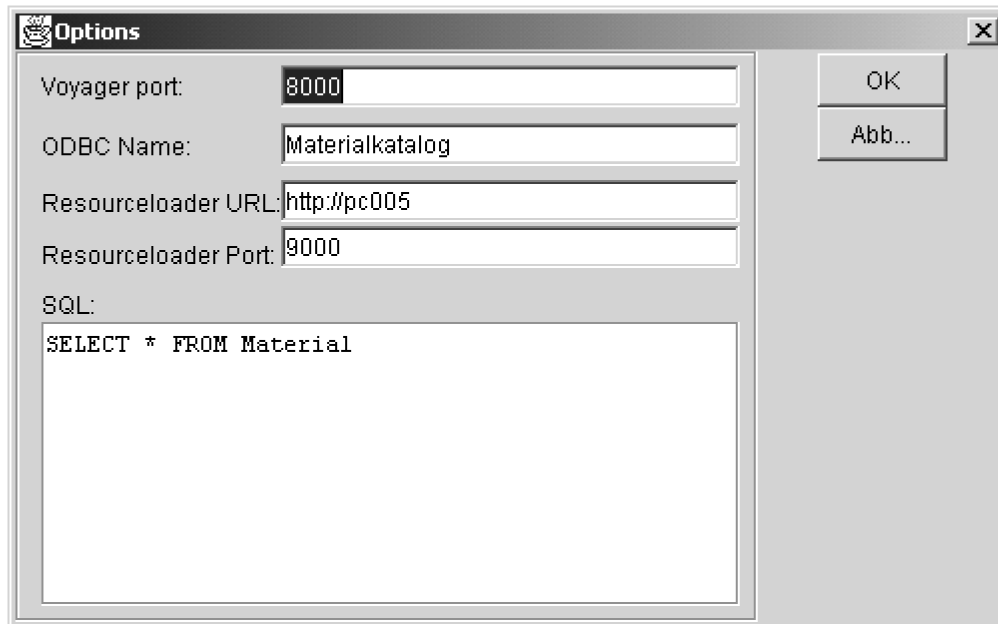


Abbildung 4-6: Die Optionen des Agenten

Der Start des Agenten erfolgt durch Drücken des Buttons *Start*. Das grafische Interface wird geschlossen und der Agent versucht, sein erstes Reiseziel anzulaufen. In der Kommandozeile des *AgentCorbaServer* wird der generierte XML-String angezeigt sowie die Aktionen bis zur Abreise bzw. die Aktionen, die bei der Ankunft ausgeführt werden.

```
H:\WINNT\System32\cmd.exe - vbj iibAgent.corba.AgentCorbaServer
Voyager auf Port 8000 gestarted.
Analysiere: Fenster
Analysiere: Ziegelmauerwerk
Analysiere: +kwert<1
Das XML Dokument wurde generiert:
<?xml version="1.0" ?>
<!DOCTYPE QUERY [
  <!ELEMENT QUERY (SEARCH)+>
  <!ELEMENT SEARCH (TOKEN)+>
  <!ATTLIST SEARCH
    Name NMTOKEN #REQUIRED>
  <!ELEMENT TOKEN (OPTION?, RESTRICTION?)>
  <!ATTLIST TOKEN
        Name NMTOKEN #REQUIRED
        DBName NMTOKEN #IMPLIED>
  <!ELEMENT OPTION (#PCDATA)>
  <!ELEMENT RESTRICTION (#PCDATA)>
]>

<QUERY>
 <SEARCH Name="MySearch">
  <TOKEN Name="Fenster">
  </TOKEN>
  <TOKEN Name="Ziegelmauerwerk">
  </TOKEN>
  <TOKEN Name="kwert">
    <OPTION>+</OPTION>
    <RESTRICTION>&lt;1</RESTRICTION>
  </TOKEN>
 </SEARCH>
</QUERY>

Erstelle Agent.
Erstelle Stoppuhr.
Versuche das naechste Reiseziel anzusteuern.
preDeparture( tcp://pc005:8000, //pc005:7000 )
postDepature()
preArrival()
postArrival()
Zuhause angekommen!!!
```

Abbildung 4-7: Die Abreise/Ankunft des Agenten

4.5 Ankunft beim Server

Sobald der Agent Rechner1 erreicht hat, veröffentlicht er seine Suchanfrage. Danach setzt er einen Timer, der angibt, wie lange er auf die Antwort warten soll.
Der Server seinerseits registriert die Anfrage, wandelt diese in SQL-Abfragen um und generiert aus den Ergebnissen die Rückgabe XML-Datei. Die einzelnen Schritte werden im Logfenster angezeigt. Dies wird in Abbildung 4-8 exemplarisch für Rechner1 dargestellt.

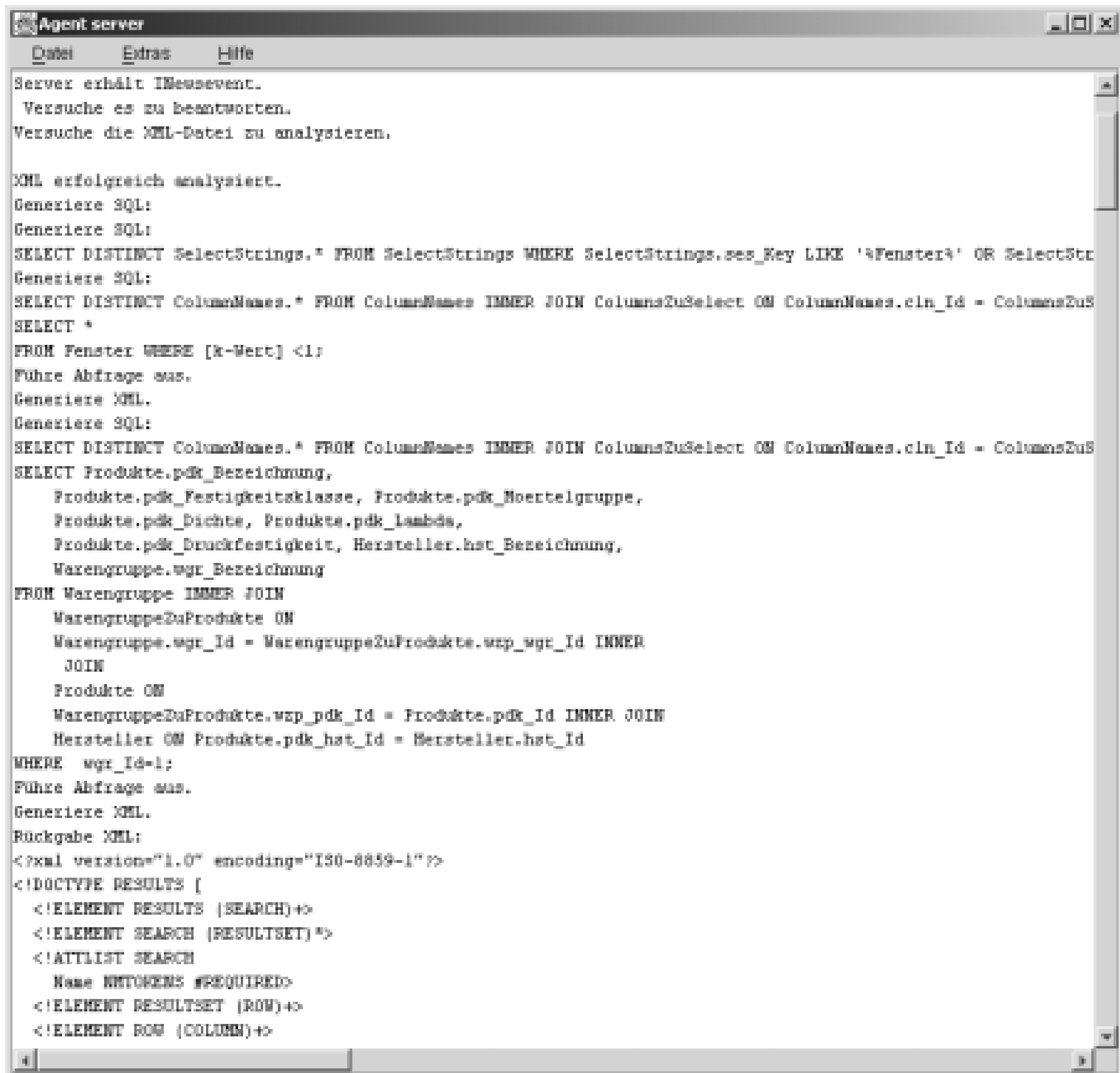


Abbildung 4-8: Logfenster des Agentenservers

Gefunden wurden Daten sowohl zu „Fenstern“ als auch zu „Ziegelmauerwerk“, die an den Agent als XML-String weitergereicht werden. Hat der Agent die Ergebnisse erhalten bzw. ist der Timer abgelaufen, steuert er sein nächstes Reiseziel, den Server, an. Dort veröffentlicht er wieder seine Suchanfrage und setzt den Timer. Hier erhält er nur Ergebnisse zu „Ziegelmauerwerk“. Danach tritt er die Heimreise an.

4.6 Die Heimreise

Zu Hause angekommen, wird automatisch der *ResultBrowser* gestartet.

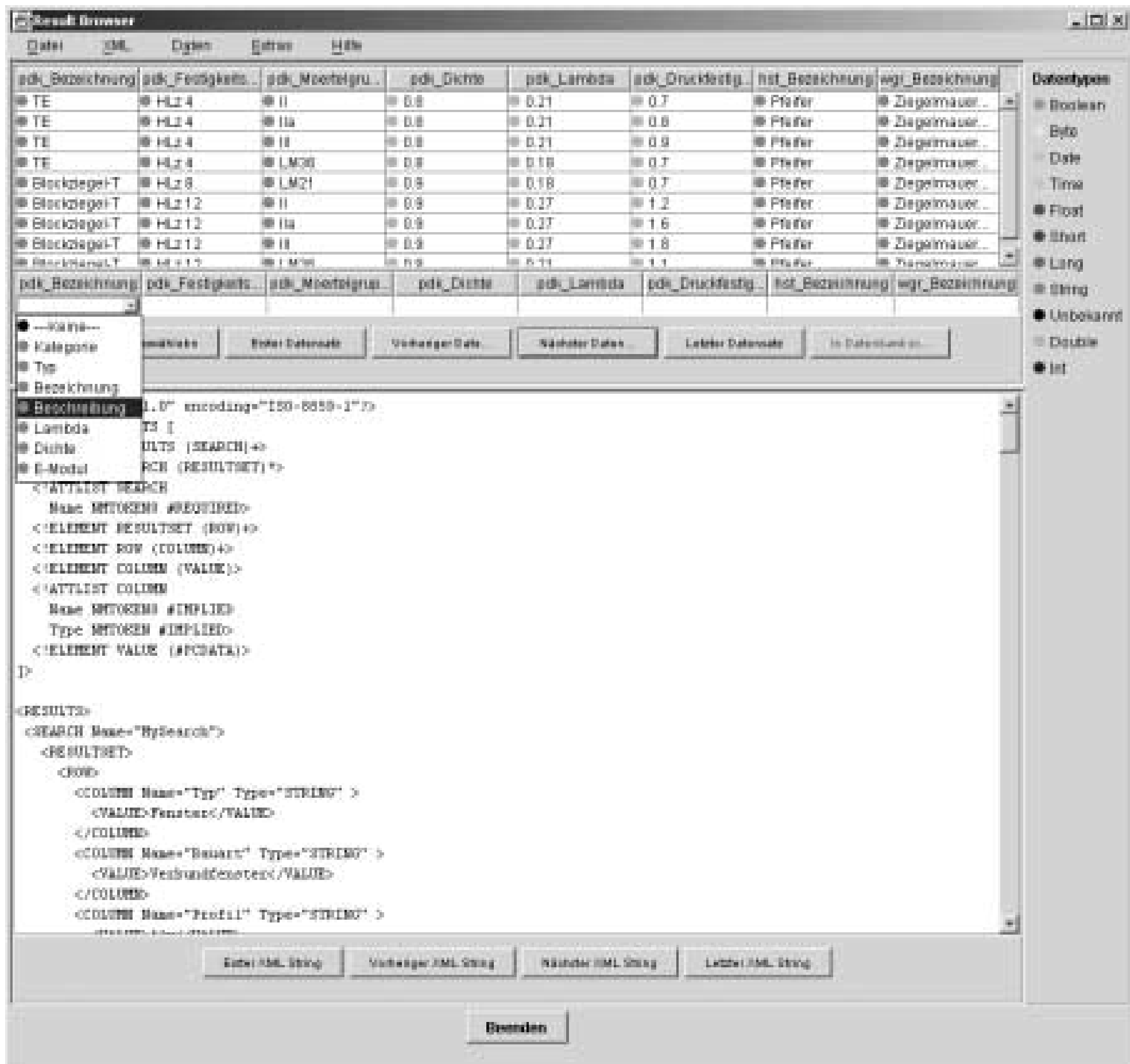


Abbildung 4-9: Der ResultBrowser

Hier werden die verschiedenen Ergebnisse angezeigt. Im unteren Teil des Fensters wird jeweils einer der erhaltenen XML-Strings dargestellt. Durch die verschiedenen Strings kann mit der unteren Buttonleiste navigiert werden. Zu dem angezeigten XML-String werden im oberen Teil des Fensters die extrahierten Daten dargestellt. Enthält ein String mehrerer Ergebnisse, kann durch diese mittels der oberen Buttonleiste navigiert werden.

Jetzt kann entweder der XML-String zur späteren Bearbeitung abgespeichert werden oder es werden gleich die ausgewählten Daten in die Tabelle „Material" der Baudatenbank übertragen. Hierfür muss im mittleren Teil des Fensters eine Spalte der Tabelle „Material" einer Spalte der Ergebnisse zugeordnet werden. Danach müssen die Datensätze selektiert werden, die übernommen werden sollen. Durch Drücken des Buttons *In Datenbank schreiben* werden die selektierten Daten eingefügt.

Abbildung 4-10: Bestätigung des Schreibvorgangs

Der Resultbrowser bestätigt dann die Anzahl der erfolgreich übertragenen Daten.

5 Zusammenfassung und Ausblick

Mobile Agenten können den Anwender bei der Arbeit in vielen Punkten entlasten. Gerade für die Informationssuche im Internet sind sie besonders geeignet, da der Anwender nicht ständig mit dem Netz verbunden sein muss. Durch steigende Intelligenz eines Agenten, lassen sich viele Arbeitsschritte einsparen oder Vorgänge voll automatisieren.
Die Kommunikation der Agenten untereinander ermöglicht ebenfalls eine Steigerung der Effektivität. Gerade in Verbindung mit der Intelligenz des Agenten, kann er so über interessante bzw. uninteressante Reiseziele lernen oder andere Agenten, die ebenfalls Daten sammeln und bereit sind, diese preiszugeben, befragen.
Mit zunehmender Intelligenz des Agenten ist es sogar denkbar, dass er nicht nur Informationen beschafft, sondern auch gleich Bestellungen aufnimmt und diese gegebenenfalls koordiniert. So können bestellte Baustoffe z.B. gleich zur entsprechenden Baustelle geliefert werden. Hierfür müsste er in irgendeiner Form ein Zahlungsmittel mit sich führen bzw. eine sichere Identifikation des Auftraggebers muss gewährleistet sein. Hier spielt die Sicherheit des Agenten eine entscheidende Rolle. Da der Agent nicht auf einem lokalen Rechner, auf dem er von Angriffen relativ gut geschützt ist, sondern in einer unbekannten Umgebung arbeitet, ist er für Angriffe sehr empfindlich. Hier gilt es entsprechende Sicherheitsmechanismen für Agentensystem zu finden. Es existieren z.B. Ansätze zur Verschlüsselung eines Agenten.

Es gibt verschiedenen Meinungen über die Bedeutung von mobilen Agenten. Pessimistische Betrachtungen räumen den Agenten kaum eine Chance ein. So könnte man Agenten auch durch andere Technologien, wie die entfernte Kommunikation ersetzen, die ein wesentlich geringeres Risikopotenzial für Sicherheitsangriffe hat. Optimistische Betrachtungen hingegen gehen davon aus, dass mobile Agenten in immer mehr Bereiche Einzug erhalten und tägliche Arbeiten im Internet erleichtern.
Die Entwicklung auf dem Gebiet der Agenten steht erst am Anfang und so bleibt abzuwarten, ob geeignete Lösungen für die Probleme gefunden werden. Das Konzept der mobilen Agenten bietet auf jeden Fall eine Reihe interessanter Ansätze, die es lohnt weiterzuverfolgen.

Für den Datenaustausch der Agenten untereinander eignet sich XML hervorragend. XML ist selbstbeschreibend, daher ist es einfach, Datenstrukturen wie relationale Datenbanken abzubilden. Diese können dann nach der Interpretation durch einen Parser, entsprechend dargestellt werden. Hier sind für verschiedenen Zwecke unterschiedliche Ansichten möglich. Da XML nichts anderes als eine Textdatei ist, kann sie, falls eine Interpretation der Struktur fehlschlägt, immer noch mit einem einfachem Texteditor betrachtet werden.
Immer mehr Anwendungen unterstützen XML als Dateiformat. So existieren bereits schon jetzt Standard DTDs für den Datenaustausch bestimmter Bereiche (z.B. Chemie oder Mathematik).

6 Anhang

6.1 Installation

6.1.1 Systemvoraussetzungen

Zur Ausführung des Agenten und dem dazugehörigen Agentenserver sind folgende Softwarekomponenten und Systemvariabeln nötig:

Softwarekomponente	Dateien/Pfade die in die Variable CLASSPATH aufgenommen werden müssen	Pfade die in die Variable PATH aufgenommen werden müssen	Bemerkung
XML for JAVA Parser Version 2.0.15	xml4j.jar;		Muss auf Client und Server installiert sein.
Voyager Version 3.2	\lib\voyager.jar ; \lib\jgl3.1.0.jar;	\bin\	Muss auf Client und Server installiert sein.
VisiBroker for Java 4.0	\lib\migration.jar; lib\vbdev.jar; lib\vbjdev.jar; lib\vbjorb.jar;		Erfolgt normalerweise bei der Installation
JDK 1.2	\lib\dt.jar; \lib\jvm.jar; \lib\tools.jar	\bin\	Muss auf Client und Server installiert sein.
MS SQL Server 7.0			Nur Server. Es muss eine Datenbank vorhanden sein, die o.g Bedingungen entspricht.
Der ODBC Treiber der Firma inet Software	Das Oberverzeichnis muss sich im Klassenpfad befinden		Nur Server.
Der iibAgent	iibAgent.jar		Nur Client.
Der Agentenserver	iibAgentServer.jar		Nur Server.
Access ODBC Treiber			Nur Client. ODBC Treiber für die Baudatenbank
Beispielanwendung			Nur Client.

Getestet wurde das System auf einem Celeron 300@450 und 196 MB Arbeitsspeicher unter Windows 2000 und Pentium 233 und 96 MB Arbeitsspeicher unter Windows NT 4.0.

6.2 Softwarekomponenten

6.2.1 JDK 1.2

Das JAVA Development Kit ist einfach über die dazugehörige Setuproutine zu installieren. Danach sind o.g. Systemvariabeln zu setzen.

6.2.2 Voyager

Voyager ist einfach über die dazugehörige Setuproutine zu installieren. Danach sind o.g. Systemvariabeln zu setzen.

6.2.3 XML for JAVA

Die ZIP Datei ist in ein beliebiges Verzeichnis zu entpacken. Danach sind o.g. Systemvariabeln zu setzen.

6.2.4 JDBC

Die ZIP Datei ist in ein beliebiges Verzeichnis zu entpacken. Danach sind o.g. Systemvariabeln zu setzen.

6.2.5 IIB Agent

Die JAR Datei ist in ein beliebiges Verzeichnis zu kopieren. Danach sind o.g. Systemvariabeln zu setzen.

6.2.6 IIB Agentenserver

Die JAR Datei ist in ein beliebiges Verzeichnis zu kopieren. Danach sind o.g. Systemvariabeln zu setzen.

6.2.7 Der Bauteileditor (Beispielanwendung)

Die Dateien Bauteileditor.exe und orb_r.lib sind in ein beliebiges Verzeichnis zu kopieren.

6.2.8 SQL – Server

- Anlegen einer neuen Datenbank.
- Ausführen des SQL-Skripts (HaendlerB.sql) in der neu angelegten Datenbank.
- Eingabe bzw. Import (*Tabellenname*.txt) der Daten in die entsprechenden Tabellen.

6.2.9 ODBC-Schnittstelle zu Access

Die ZIP-Datei ist in ein beliebiges Verzeichnis zu entpacken. Danach sind o.g. Systemvariabeln zu setzen.

6.3 Kompilieren des Quellcodes

6.3.1 iibAgent

Der iibAgent muss zuerst mit *javac* (oder einem anderen JAVA Compiler) kompiliert werden. Danach muss das Verzeichnis *iibAgent\Corba* mit *vbjc* (Visigenic Compiler) erneut kompiliert werden. Dies kann auch mit der sich im selben Verzeichnis befindenden *vbmake.bat* Stapelverarbeitungsdatei erfolgen. Erst dann funktioniert die CORBA-Unterstützung.

6.3.2 iibAgentServer

Der Agentenserver kann mit einem beliebigen JAVA Compiler (z.B. *javac*) kompiliert werden.

6.3.3 Bauteileditor

Dieses Programm wurde mit MS Visual C++ 6.0 erstellt und kompiliert. Die Projektdatei befindet sich beim Quellcode.

6.4 Literaturverzeichnis

[Boger 1999] Marko Boger, „JAVA in verteilten Systemen: Nebenläufigkeit, Verteilung, Persistenz“, dpunkt-Verlag 1999, ISBN 3-932588-32-0

[Sun 1999] Sun Microsystems, Inc. “The JAVA Tutorial”, http://java.sun.com/docs/books/tutorial/

[Bradley 1998] Neil Bradley, „The XML companion“, Addison Wesley Longman Limited, ISBN 0-201-342855

[Objectspace 1999] ObjectSpace, Inc., “Orb 3.1 Developer Guide”, http://www.objectspace.com/

[Böhme 1999] Timo Böhme, „Agentensysteme – aktueller Entwicklungsstand und Konzeption eines universellen News-Watcher-Agenten“, Diplomarbeit Universität Leipzig, März 1999

[Ahmed 1998] Suhail M. Ahmed, “CORBA Programming Unleashed”, Sams Publishing 1998, ISBN 0-672-31026-0

[Lange/Oshima 1998] Danny B. Lange & Mitsuru Oshima, „Programming and deploying Java mobile agents with aglets”, Addison Wesley Longman Limited 1998, ISBN 0-201-32582-9

[Corba 2000] OMG (Übersetzung: corba.ch). “Was ist CORBA?”, http://www.corba.ch/WasIstCorba.html 2000

6.5 Abbildungsverzeichnis

Abbildung 2-1: Klassifikationsmatrix nach Brenner et al [Böhme 1999]..............................2-2
Abbildung 2-2: Client/Server-Paradigma und mobile Agenten 2-4
Abbildung 2-3: Die verschiedenen Stufen eines JAVA Programms......................................2-7
Abbildung 2-4: Die verschieden JDBC-Treiber [Boger 1999]...2-9
Abbildung 2-6: SAX-Parser..2-22
Abbildung 2-7: DOM-Parser 2-24
Abbildung 2-8: Der Object Request Broker 2-26
Abbildung 2-9: Die Rolle der IDL [Boger 1999] 2-27
Abbildung 2-10: Der Ressourcen-Server..2-32
Abbildung 2-11: Raum-Architektur (Space) 2-36
Abbildung 3-1: Übersicht der Kommunikation der Objekte 3-2
Abbildung 3-2: Kommunikation Agent <-> Server...3-3
Abbildung 3-3: Das Interface INewsEvent...3-4
Abbildung 3-4: Das Grafische Interface des Agenten 3-8
Abbildung 3-5: Die Klasse Parser...3-9
Abbildung 3-6: Der Resultbrowser..3-12
Abbildung 3-7: Klassen für die Darstellung der Datentypen 3-13
Abbildung 3-8: Die Optionenmaske 3-13
Abbildung 3-9: Aufbau der SQL Server-Tabellen 3-14
Abbildung 3-10: Beispieldatenbank eines Großhändlers 3-14
Abbildung 3-11: Der Agentenserver..3-16
Abbildung 3-12: Die Agentenserver-Optionen..3-17
Abbildung 3-13: Klassifikation des Agenten...3-19
Abbildung 3-14: Klassifikation des Agentenserver..3-20
Abbildung 4-1: Start des Agenten-Dienstes 4-2
Abbildung 4-2: Der Ressourcenserver...4-3
Abbildung 4-3: Start des AgentCorbaServer 4-3
Abbildung 4-4: Der Bauteileditor 4-4
Abbildung 4-5: Das grafische Interface des Agenten..4-5
Abbildung 4-6: Die Optionen des Agenten 4-6
Abbildung 4-7: Die Abreise/Ankunft des Agenten 4-7
Abbildung 4-8: Logfenster des Agentenservers 4-8
Abbildung 4-9: Der ResultBrowser 4-9
Abbildung 4-10: Bestätigung des Schreibvorgangs...4-10