Diplomarbeit

Entwicklung eines Dialogsystems zur multimodalen Interaktion zwischen Mensch und Agenten unter besonderer Berücksichtigung der natürlichen Sprache

ULRICH PINSDORF

Fachhochschule Bingen
Fachbereich Elektrotechnik
Studiengang Ingenieurinformatik
Berlinstraße 109, 55411 Bingen

Fraunhofer Institut für
Graphische Datenverarbeitung
Abteilung Sicherheitstechnologie
für Graphik- und Kommunikationssysteme
Rundeturmstraße 6, 64283 Darmstadt

Diplomarbeit

Prüfer	Prof. Dr.-Ing. Peter Rausch Fachhochschule Bingen
Betreuer	Dipl.-Ing. Mehrdad Jalali Fraunhofer IGD, Darmstadt
Kandidat	Ulrich Pinsdorf Matr.-Nr. 187327
Tag der Ausgabe	27. Oktober 1999
Tag der Abgabe	26. April 2000

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Die rasante Entwicklung des World Wide Webs und des Electronic Commerce in den letzten Jahren zeigt den Trend, Informationen, Produkte und Dienstleistungen immer näher an den Kunden zu bringen. Die Entwicklung zeigt aber auch, dass Anbieter von Informationsinhalten bereit sind, in Technologien zu investieren, die es dem Kunden erlauben, immer einfacher und bequemer auf die angebotenen Informationen zuzugreifen. Dabei kommt es unter Wettbewerbern zu regelrechten Wettläufen, wenn es um einen möglichst einfachen Zugriff auf ihre Angebote geht.

Eine weitere Vereinfachung des Zugriffs, besonders in den Bereichen Electronic Commerce und Informationsbeschaffung, stellt die Verwendung von Agenten dar. Agenten, und insbesondere Mobile Agenten, sind ein Paradigma, das die Softwareentwicklung dieses Jahrzehnts prägen wird [33]. Aufgrund ihres autonomen Handelns im Auftrag des Benutzers, ihrer Mobilität und der Möglichkeit zur Personalisierung eignen sich Agenten in besonderer Weise für Anwendungsszenarien in den genannten Bereichen Electronic Commerce und Informationsbeschaffung (vgl. [49]).

Möchte ein potentieller Kunde beispielsweise über das Internet Angebote zu einem Produkt einholen, kann er einen personalisierten Mobilen Agenten starten, der die Aufgaben der Informationsbeschaffung, Evaluation nach vorgegeben Kriterien und u.U. sogar den Einkauf des Produktes für ihn erledigt. Dies erfordert natürlich eine

aufgabenspezifische Konfiguration bzw. Instruktion des Agenten durch den Benutzer.

Um die Vorteile einer agentenbasierten Lösung gegenüber den herkömmlichen Methoden der Informationsbeschaffung nicht wieder zu verlieren, muss in besonderem Maße Wert auf eine benutzerfreundliche Mensch-Maschine-Schnittstelle des Agenten gelegt werden [54]. Eine Interaktion des Benutzers mit dem Agenten mittels natürlicher Sprache, oder sogar eine multimodale Interaktion ist also notwendig, um die Akzeptanz Mobiler Agenten für die genannten Aufgaben zu erhöhen. Dafür wird eine intuitive bedienbare Schnittstelle benötigt, über die der Benutzer einem Agenten seine Wünsche aufgabenorientiert mitteilen kann.

Das im Rahmen dieser Arbeit entwickelte Dialogsystem erlaubt eine multimediale Interaktion zwischen Agent und Benutzer. Dabei liegt ein besonderer Schwerpunkt auf der Interaktion mittels natürlicher Sprache.

1.2 Zielsetzung

Ziel dieser Diplomarbeit ist es, eine Lösung zu finden, die es erlaubt, einen (Mobilen) Agenten mit Hilfe der natürlichen Sprache zu konfigurieren. Die natürliche Sprache soll dabei anwendungsspezifisch interpretiert und in ein wohldefiniertes semantisches Protokoll, eine Agentenkommunikationssprache, übersetzt werden. Dieses Protokoll wird vom Agenten verstanden.

Da abzusehen ist, dass die Ergebnisse dieser Arbeit in zwei vom Bundesministerium für Bildung und Forschung (BMBF) geförderten Projekten[1] zum Einsatz kommen, wird eine möglichst offene, und nicht von einer speziellen Anwendung abhängige, Lösung angestrebt. Die zu entwickelnde Lösung soll folgende Eigenschaften haben:

- Die Schnittstellen zum Agenten, wie auch zum Benutzer hin, sollen klar definiert sein.
- Die Lösung soll möglichst allgemein sein, so dass sie sich leicht an neue Dialoge und sprachliche Kontexte anpassen lässt.

[1] Es handelt sich dabei um zwei der Sieger-Projekte des BMBF-Leitprojekt-Wettbewerbs “Mensch-Technik-Interaktion in der Wissensgesellschaft” (siehe Abschnitt 6.2)

- Der Benutzer soll nicht über Gebühr in Wortwahl und Satzbau eingeschränkt werden.
- Die Lösung soll leicht um grammatische und lexikalische Möglichkeiten erweiterbar sein.
- Entwickler eines Agenten bzw. eines Softwaresystems sollen das Produkt möglichst unkompliziert einsetzen können.

Die Lösung des gestellten Problems erfolgt mit Methoden, Konzepten und Werkzeugen des Objektorientierten Software Engineerings.

1.3 Überblick

Im Teil I der Arbeit werden die theoretischen Grundlagen der von dieser Arbeit tangierten Gebiete der Informatik dargestellt. Im Anschluss an die hier dargelegte Einführung in die Aufgabenstellung, stellt *Kapitel 2* die Technologie der Agentensysteme vor und führt in die Thematiken Intelligente Agenten, Multiagentensysteme und Mobilen Agenten ein. In *Kapitel 3* wird die Funktionsweise sprachlicher Benutzerschnittstellen beschreiben. In diesem Kontext wird der Aufbau sprachverstehender Systeme dargelegt und Anforderungen an einen guten Mensch-Maschine-Dialog formuliert. *Kapitel 4* geht auf semantische Protokolle als Möglichkeit des Wissenstransfers ein. Es beschreibt den Aufbau einer allgemeinen Kommunikation und stellt Protokolle vor, die auf den verschiedenen Ebenen einer Kommunikation zum Einsatz kommen. Zum Abschluss dieses ersten Teils beschreibt *Kapitel 5* das SeMoA Projekt der Fraunhofer Gesellschaft, in dessen Umfeld diese Arbeit entstanden ist.

Teil II widmet sich der Lösung des zu entwickelnden Dialogsystems für Mobile Agenten. In *Kapitel 6* erfolgt zunächst die Defintion des Problems. Dann wird in *Kapitel 7* die Lösungsidee dargestellt, die der Systementwicklung zugrunde liegt. In *Kapitel 8* wird zusammenfassend in einer Vorausschau die grundsätzliche Vorgehensweise bei Analyse und Design beschrieben. Dann erfolgt in *Kapitel 9* die Analyse des zu entwickelnden Systems. Problems. Das anschließende Design ist in *Kapitel 10* beschrieben. Der zweite Teil der Arbeit schließt in *Kapitel 11* mit der Besprechung der Implementation.

Abschließend geben die *Kapitel 12* und *Kapitel 13* im Teil III eine Zusammenfassung der in dieser Arbeit gefundenen Lösungen, sowie einen Ausblick auf mögliche Erweiterungen.

Teil I

Theoretische Grundlagen

Kapitel 2

Agententechnologie

In diesem Kapitel werden die Begriffe *Intelligenter Agent*, *Multiagentensystem* und *Mobiler Agent* definiert. Der Mobile Agent steht dabei im Mittelpunkt der Betrachtungen. Deshalb wird besonderer Wert gelegt auf die Erläuterung des Prinzips der Migration, sowie die Abgrenzung des Mobilen Agenten vom allgemeineren Begriff des Mobilen Codes. Am Ende dieses Kapitels werden typische Anwendungsfelder Mobiler Agenten vorgestellt.

2.1 Intelligente Agenten

Das vom lateinischen Wort *agere* (‘handeln’, ‘wirken’) abstammende Wort *Agent* bezeichnet ursprünglich einen ‘Geschäftsträger’ im politischen Sinn. Heute bezeichnet die Brockhaus Enzyklopädie [8] einen Agenten als einen ‘Vertreter’ oder ‘Spion’.

Im Kontext der Informatik ist der Begriff des Agenten etwa seit Ende der 70er Jahre bekannt [7]. Seine Bedeutung ist jedoch sehr weitschweifend und reicht vom einfachen E-Mail Filter[1] bis hin zu einer Kontrollsoftware für die Flugüberwachung [40]. Vor allem durch die Verbreitung des Internets und die damit verbundene Erschließung des Vertriebsweges “World Wide Web” werden agentengestützte Dienste in diesen Tagen mehr und mehr zu einem Schlagwort. Das Marketing hat Mög-

[1] Der Mail-Transportdienst *sendmail* verwendet sog. Mail Delivery Agents (MDA) um E-Mails an den Benutzer auszuliefern. Siehe [11]

lichkeiten von Agenten entdeckt, um dem Internetkunden einerseits mehr Komfort bei der Auswahl der Produkte zukommen zu lassen, und andererseits mehr über seine Interessen und Wünsche zu erfahren. So werden die im World Wide Web angebotenen, personalisierten Informationsdienste, die den Benutzer z.B. automatisch per E-Mail über Bücherneuerscheinungen des Lieblingsautors oder -genres informieren, gerne als *Agent* bezeichnet.

Die exakte Definition eines Agenten ist in der Fachliteratur nicht eindeutig. RUSSELL und NORVIK definieren einen *Agenten* folgendermaßen:

> "An agent is anything that can be viewed as *perceiving* its environment through *sensors* and *acting* upon that environment through *effectors*." [66]

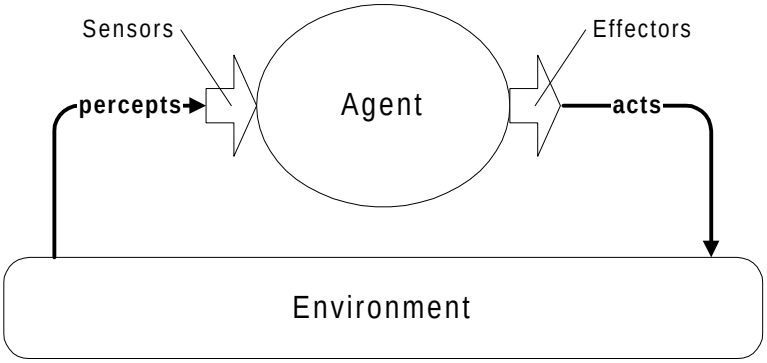


Abbildung 2.1: Definition eines Agenten nach RUSSELL und NORVIK [66]

Abbildung 2.1 veranschaulicht diese Definition. Dieser Umschreibung folgend gibt es z.B. menschliche Agenten, Roboter-Agenten und Software-Agenten. RUSSELL/NORVIK führen weiter aus:

> "A human agent has eyes, ears, and other organs for sensors and hands, legs, mouth, and other body parts for effectors. A robotic agent substitutes cameras and infrared range finders for the sensors and various motors for the effectors. A software agent has encoded bit strings as its percepts and actions." [66]

Diese Definition ist sehr allgemein und sagt noch nichts über die “kognitiven” Fähigkeiten eines Agenten aus. Kritisch betrachtet handelt es sich sogar nur um eine Spezialisierung des allgemeinen IPO[2]-Prinzips dahingehend, dass sich Eingabe und Ausgabe auf die gleiche “Umgebung” beziehen müssen. Dies wiederum kommt der aus der Regelungstechnik bekannten Definition eines geschlossenen Regelkreises sehr nahe. Auf dieser Grundlage kann man auch solche Systeme als Agenten bezeichnen, die man intuitiv vielleicht nicht so benennen würde. Zum Beispiel erfüllt ein Heizungsthermostat die Anforderungen der ersten Definition von RUSSELL und NORVIK.

Aufbauend auf der allgemeinen Definition eines Agenten definieren RUSSELL und NORVIK einen *rationalen Agenten* als einen Agenten, der danach strebt, durch sein Agieren das Maß seines Erfolges zu vergrößern:

> “For each possible percept sequence, an ideal rational agent should do whatever action is expected to maximize its performance measure, on the basis of the evidence provided by the percept sequence and whatever built-in knowledge the agent has.”

Der Agent nimmt also seine Umgebung wahr, ermittelt unter Zuhilfenahme einer eigenen Wissensbasis[3] eine für seine Ziele günstige Handlungsweise, und wirkt damit auf die Umgebung ein. KINNEBROCK fasst dies in seiner Definition eines (autonomen) Agenten zusammen:

> “Ein autonomer Agent ist ein System, welches ein selbständiges Verhalten zeigt. Er kann wahrnehmen (Input) und agieren (Output). Wahrnehmung und Aktion sind so, dass ein adaptives Verhalten entsteht.”[42]

Eine andere, sehr verbreitete Herangehensweise an die Definition eines intelligenten Agenten ist die Aufzählung einer Reihe von Basisfähigkeiten, die einen Agenten von einem Software-Objekt unterscheiden. “Eine solche repräsentative

[2] Input, Processing, Output

[3] Der Begriff *Wissensbasis* ist an dieser Stelle nicht mit dem spezielleren Begriff der KI-Forschung identisch. An dieser Stelle kann das Wissen z.B. auch ein einfacher Algorithmus oder eine Lookup-Table sein.

und weit verbreitete Notation eines Agenten ist die von WOOLRIDGE und JENNINGS [87] aufgestellte Definition, die einen Agenten als ein Computer-System mit den Eigenschaften autonom, sozial, reaktiv und proaktiv beschreibt." [47]

> "Der Begriff des (intelligenten) Agenten beschreibt ein Hardware- oder (häufiger) ein Software-basiertes System mit den Eigenschaften:
>
> **autonom:** Agenten operieren ohne direkten Einfluss eines anderen und verfügen über eine Art von Kontrolle über ihre Aktionen und internen Zustände.
>
> **sozial:** Agenten interagieren mit anderen Agenten über irgendeine Art von Agentenkommunikationssprache.
>
> **reaktiv:** Agenten nehmen ihre Umgebung wahr und antworten auf Umgebungsveränderungen.
>
> **proaktiv:** Agenten verfügen über ein zielorientiertes Verhalten durch Ergreifen der Initiative."[4]

Auf diese Definition von WOOLRIDGE und JENNINGS stützen sich die meisten Autoren [7, 29, 33, 47, 49]. In neueren Publikationen, z.B. [75], findet man häufig eine noch weitergehende Liste von Anforderungen an intelligente Agenten, die aber nur zum Teil erfüllt werden müssen, um der Definition eines Agenten zu genügen. In diesen Definitionen werden zu den Vier genannten zusätzlich häufig folgende Eigenschaften angeführt:

- adaptiv/lernend
- persistent
- zielorientiert
- kommunikativ
- kooperativ
- flexibel

[4] Definition nach WOOLRIDGE und JENNINGS, entnommen aus [47]

Im Rahmen dieser Arbeit wird, wie beim überwiegenden Teil der Fachliteratur auch, die Definition von WOOLRIDGE und JENNINGS für einen intelligenten Agenten zugrunde gelegt. Nach Meinung des Autors bietet sie die beste Möglichkeit, die weiteren Definitionen für die Begriffe *Multiagentensystem* und *Mobiler Agent* abzuleiten, die in den folgenden Abschnitten behandelt werden.

2.2 Multiagentensysteme

Multiagentensysteme sind eine vielversprechende Lösungstrategie in der Informatik. Sie verbinden die Möglichkeiten des verteilten Rechnens mit den Bestrebungen der künstlichen Intelligenz. Die kooperativen Agenten führen soziale Interaktionen mit anderen Agenten, d.h. sie kommunizieren mit anderen Agenten, um für sich ein besseres Ergebnis zu produzieren. Kooperative Agentengruppen können aufgrund von Synergieeffekten hocheffiziente Lösungen ermitteln. Sie sind in der Lage, gemeinsam komplexe Problemstellungen zu lösen, die im voraus nicht zu programmieren sind [68]. Diese Idee wurde bereits sehr früh in der Forschung um verteilte künstliche Intelligenz formuliert.

Ein *Multiagentensystem* besteht aus einer Anzahl von Agenten (oder Agentengruppen), die miteinander kommunizieren und kooperieren. Jeder Agent hat individuelle Ziele, die er erreichen möchte. Andere Agenten, die selbstverständlich ihre eigenen Ziele zum Maßstab ihrer Handlungen setzen, helfen ihm dabei. Die Kommunikation geschieht mit Hilfe von Agentenkommunikationssprachen[5]. Die Agenten müssen nicht notwendigerweise auf verschiedenen Rechnern residieren, sondern sie werden häufig als lokale Anwendungen verwendet. Multiagentensysteme können z.B. in den folgenden Bereichen eingesetzt werden:

- Unterstützung in Arbeitsgruppen
- Produktionsplanung und -überwachung
- intelligente Recherchesysteme
- Luftverkehrskontrolle

[5] siehe Kapitel 4

- Kommunikationsnetzverwaltung

Es gibt eine Vielzahl von Projekten, die mit Multiagenten-Systemen experimentieren. Eine sehr umfangreiche Aufstellung findet sich in [80].

2.3 Mobile Agenten

Ein mögliches Klassifikationsmerkmal von Software-Agenten ist die Fähigkeit zur Fortbewegung; man unterscheidet hier zwischen den Begriffen des *Stationären Agenten* und des *Mobilen Agenten*.

Die Klassifikation *stationär* wird selten auf einen Agenten angewendet. Man verwendet ihn nur im direkten Vergleich mit Mobilen Agenten und meint damit die Klasse der Nicht-mobilen Agenten.

Der Begriff des *Mobilen Agenten* wurde Anfang der 90er Jahre von der Firma General Magic [32], die seit 1997 ein Patent für diese Technologie hat, geprägt. Die Patentschrift [81] beschreibt Mobile Agenten als autonome Programme, die sich in einem heterogenen Netzwerk fortbewegen können und im Auftrag des Benutzers Dienste verrichten. Diese Fähigkeit nennt man Migration. Ein Agent kann selbst entscheiden, ob und wohin er migrieren möchte.

Basierend auf der Definition, die in Kaptitel 2.1 für einen intelligenten Agenten festgelegt wurde, kann man diese für einen *Mobilen Agenten* folgendermaßen erweitern:

> Ein Mobiler Agent ist ein Agent, der in seinem Lebenszyklus auf verschiedenen Wirtssystemen zur Ausführung kommen kann.

Die spezielle Unterschied zwischen einem Mobilen Agenten und einem herkömmlichen Programm, das von Rechner zu Rechner kopiert und dort immer wieder neu gestartet wird, liegt in der Tatsache, dass die Fortbewegung beim Agenten innerhalb *eines Lebenszyklus* geschieht. Er wird für den Transport zu einem anderen Rechner nicht gestoppt und auf dem Zielhost *neu* gestartet, sondern nur “eingefroren”. Das Programm terminiert dabei nicht, sondern der Kontrollfluss wird an beliebiger Stelle unterbrochen und später auf einer anderen Ausführungseinheit

wieder aufgenommen. Im Idealfall wird er an genau der Stelle fortgesetzt, an der er unterbrochen wurde. In der Praxis ist es jedoch häufig so, dass der Agent nach dem "Aufwachen" eine spezielle Routine wie z.B. `onCreation()` ausführt. Aus Sicht des Agenten bewirkt ein Migrationswunsch lediglich den Sprung des Kontrollflusses in diese Routine.[6]

Migration ist also das Verschieben eines laufenden Programms auf eine andere Ausführungseinheit. Der Prozesszustand des Programms bleibt dabei erhalten. Dafür wird der aktuelle Programmzustand, also die Menge aller[7] Variablenwerte, die das Programm konstituieren, gespeichert. Diese Eigenschaft heißt *Persistenz*.

Möchte ein Mobiler Agent auf einen entfernten Host migrieren, so teilt er den Migrationswunsch dem lokalen Server mit. Wird diesem Wunsch seitens des Servers stattgegeben, wird der Agent angehalten, der Programmcode und die persistenten Variablen serialisiert und zum Zielrechner transferiert. Dort wird der empfangene Bitstrom wieder deserialisiert und die persistenten Variablen und Datenstrukturen mit den ursprünglichen Werten initialisiert. Das Ergebnis ist ein Abbild des Mobilen Agenten vom Speicher des Quellhosts in den Speicher des Zielhostes zum Zeitpunkt der Serialisierung.

2.3.1 Mobiler Code

Die dem Konzept der Mobilen Agenten zugrunde liegende Idee des *Mobilen Codes* ist nicht neu. Ältester Vertreter dieser Software-Gattung ist wahrscheinlich die Dokumentbeschreibungssprache PostScript. Beim Drucken von PostScript-Dateien wird der PostScript-Code ebenfalls auf einem Prozessor erzeugt und auf einem anderen Prozessor, nämlich dem Postscript-Interpreter des Druckers, ausgeführt.

[6] Es ist aus psychologischer Sicht sehr interessant, dass der Agent keine Möglichkeit hat, sein "Einschlafen" und "Aufwachen" wahrzunehmen, sondern sich auf externe Informationen, wie z.B. die Systemzeit oder die IP-Adresse des Hostes, verlassen muss. Dies ist vergleichbar mit dem Erwachen eines Menschen aus seiner Bewusstlosigkeit – auch er ist vollständig auf Informationen von Außen angewiesen und kann sich meist nur an die Situation unmittelbar vor dem Verlust des Bewusstseins erinnern.

[7] In der Praxis wird aus Gründen der Effizienz nur der essentielle Teil der Variablenwerte gespeichert. Nichtessentielle Variablen werden nach dem Wiederbeleben des Agenten mit Default-Werten initialisiert.

Ein weiterer prominenter Vertreter des Mobilen Codes sind Java™Applets. Hier wird der Programmcode von einem entfernten Rechner geholt und lokal zur Ausführung gebracht. Damit der Appletcode gleichermaßen auf verschiedenen Betriebssystemen lauffähig ist, wird auf dem lokalen Rechner eine Virtual Machine emuliert. Dadurch findet der Code scheinbar immer die gleiche Systemumgebung vor. Zudem erlaubt die Virtual Machine die Kontrolle der dem Applet zur Verfügung stehenden Betriebsmittel, und schützt so das lokale System (weitgehend) vor potentiell böswilligen Zugriffen des Applets.

Auch Computer-Viren und Netzwerk-Würmer kann man im weiteren Sinne zur Kategorie des Mobilen Codes rechnen. Strenggenommen wandert jedoch nicht *eine* Instanz des Codes, sondern es wird auf jedem neu infizierten System eine *neue* Instanz des Virus angelegt.

Was unterscheidet aber *Mobilen Code* von einem *Mobilen Agenten*? Das wichtigste Unterscheidungskriterium ist die fehlende Persistenz des Mobilen Codes. Kommt beispielsweise ein Java-Applet auf einem Gastsystem zur Ausführung, beginnt der Kontrollfluss stets bei der ersten Programmanweisung. Das Applet wird also nach jeder Übertragung vollkommen neu gestartet. Mobile Agenten hingegen haben einen persistenten Programmstatus. Das bedeutet, dass nach jeder Migration des Agenten einige oder alle Programmvariablen die Werte besitzen, die sie vor der Migration hatten. Ein weiteres Kriterium ist der Umstand, dass sich Mobiler Code üblicherweise reproduziert, von einem Mobilen Agenten hingegen gibt es i.d.R. genau eine Instanz.

2.3.2 Einsatzgebiete für Mobile Agenten

Der Mechanismus der Migration erlaubt es dem Mobilen Agenten, von Server zu Server zu wandern, und Daten — etwa aus einer Datenbank — jeweils lokal abzufragen. Ein Mobiler Agent braucht deshalb keine ständige Netzverbindung zum Heimatserver, sondern kann/muss autonom im Netz agieren. Typische Anwendungfelder Mobiler Agenten finden sich vor allem in Verbindung mit mobilen Endgeräten. Die drahtlose Verbindung zum Festnetz wird nur aufgebaut, um die Agenten zu versenden und um sie wieder in Empfang zu nehmen. Nach ihrer Rückkehr präsentieren sie dann die Ergebnisse ihrer Aufgabe. Diese Fähigkeit unterstreicht

noch einmal die Bedeutung Mobiler Agenten für das kommende Jahrzehnt. Die Bandbreiten mobiler Netzanbindungen sind sehr begrenzt. Dies wird wohl auch in Zukunft so bleiben. Es stehen zwar neue Netzzugangstechniken, die höhere Bandbreiten garantieren, vor der Markteinführung, doch es ist zu erwarten, dass im Zuge dieses Fortschritts auch Anspruch und Umfang mobiler Anwendungen steigen werden. Ein zweiter Zukunftsaspekt ist, dass auch die Arbeitszeit des Menschen eine immer kostbarere Ressource wird. Darum werden routinemäßige, zeitraubende Tätigkeiten delegiert oder von Programmen erledigt. Hier finden Mobile Agenten optimale Einsatzfelder. Konkrete Aufgaben finden sich für sie besonders in den Bereichen:

Informationsbeschaffung: Mobile Agenten fahnden im Auftrage des Benutzers nach Daten in Datenbanken des World Wide Webs (*Data Mining*). Die Adressen der Zielhosts sind entweder vom Benutzer vorgegeben oder werden vom Agenten durch Kommunikation mit anderen Agenten ermittelt. Der Agent präsentiert anschließend eine Zusammenfassung der gesammelten Daten.

Electronic Commerce: Mobile Agenten tätigen für den Benutzer einen Einkauf. Sie besuchen die Server verschiedener Produktanbieter, und evaluieren Preis und Qualität der Angebote nach vorgegebenen Maßstäben. Anschließend tätigen sie den Einkauf bei dem Anbieter, der die Maßgabe des Benutzers am Besten erfüllt.

Personalisierte Dienste: Mobile Agenten ermitteln selbsttätig Neuerscheinungen von Büchern, die den Benutzer interessieren könnten, stellen für eine Reise die benötigten Informationen zusammen oder liefern eine Zusammenschau von Weltnachrichten nach der Interessenlage des Benutzers.

Fernwartung: Mobile Agenten ermitteln den Zustand diverser Systemkomponenten in einem Netzwerk und präsentieren eine auf das Informationsbedürfnis des Benutzers zugeschnittene Zusammenfassung der Ergebnisse.

Workflow Assistence: Mobile Agenten koordinieren Termine zwischen Mitgliedern einer Arbeitsgruppe, reservieren Besprechungsräume und bieten Unterstützung bei der Organisation und Beantwortung von Korrespondenz.

Im Umfeld der Mobilen Agenten gibt es mittlerweile eine Vielzahl von Forschungsprojekten. Zwei sehr umfangreiche Listen finden sich in [38] und [80].

Die starken Forschungsbemühungen begründen sich vor allem darin, dass Mobile Agenten als echte Unterklasse von Multiagentensystemen deren Vorteile in sich vereinen und zudem über die Fähigkeit der Mobilität verfügen. Gegenüber dem Mobilen Code, der eine homogene Infrastruktur in einem typischerweise heterogenen Netzwerk benötigt, haben sie den Vorteil der Plattformunabhängigkeit, denn ein Agentenhost bietet dem Agenten eine eindeutig definierte Schnittstelle zum Wirtssystem. Diese Tatsache ist für sich genommen bereits ein gewaltiger Bonus, den bisherige Systeme[8] nicht bieten können.

[8] Als Beispiele für Systeme, die angetreten sind, um einen homogenen Zugriff auf Daten und Programme in heterogenen Netzwerken anzubieten, seien hier exemplarisch *Remote Procedure Calls* (RPC) und *Remote Method Invocation*(RMI) genannt.

Kapitel 3

Sprachliche Benutzerschnittstellen

Das dritte Kapitel stellt die Grundlagen der Sprachverarbeitung dar. Ausgehend von der momentanen Situation wird in Abschnitt 3.1 die Bedeutung sprachverarbeitender Systeme auf die Zukunft prognostiziert. Abschnitt 3.2 widmet sich dann den verschiedenen Ebenen der Sprachverarbeitung: der Spracherkennung und dem Sprachverstehen. Abschließend werden die gängigsten Formen zur Realisierung sprachverstehender Systeme erläutert. Darauf aufbauend widmet sich Abschnitt 3.3 den Aspekten der sprachlichen Interaktion (Dialogsysteme) und der Multimodalität.

3.1 Bedeutung sprachlicher Schnittstellen

Ein wichtiges Kriterium für die Akzeptanz eines technischen Systems ist seine Benutzerschnittstelle. Schon ein Auto verkauft sich in den meisten Fällen nicht aufgrund seiner technischen Daten, sondern es sind Begriffe wie Fahrverhalten, Komfort und Ausstattung — also im weiteren Sinne die Mensch-Maschine-Schnittstelle — die ein Fahrzeug letztendlich zu einem Verkaufsschlager oder einem Ladenhüter machen. Ebenso verhält es sich bei Softwaresystemen. Ein populäres Beispiel ist der rasche und nachhaltige Erfolg grafischer Betriebssystem-Aufsätze wie Microsoft Windows oder das X Window System für UNIX.

Menschliche Sprache als Eingabemedium gewinnt bei der Entwicklung von Computerprogrammen und computergestützen Systemen zunehmend an Bedeutung. Es ist abzusehen, dass in wenigen Jahren kaum eine Anwendung ohne eine sprachliche Schnittstelle auskommen wird. Die Bedeutung sprachlicher Kommunikation zwischen Mensch und Maschine formuliert SAGERER sehr eindringlich:

> "Sprache ist das bevorzugte und natürliche Kommunikationsmittel zwischen Menschen. Schon alleine deshalb ist Sprache als Kommunikationsform zwischen Mensch und Maschine anzustreben." [67]

Bereits im Abschnitt 1.1 wurde auf die besondere Bedeutung intuitiv bedienbarer Benutzerschnittstellen im Zusammenhang mit Mobilen Agenten hingewiesen. Gerade Mobile Agenten, deren Aufgaben im Umfeld der Informationsbeschaffung und des Electronic Commerce liegen, benötigen ein einfaches, von einem ungeübten Benutzer verwendbares Interface. Dieses Interface kann natürlich eine grafische Oberfläche sein. Intuitiver hingegen ist für den Menschen die Verwendung von Sprache.

Sprache ist für den Menschen eine natürliche Form der Kommunikation, sie ist pervasiv[1], effizient, kann über eine Distanz hinweg genutzt werden und lässt Hände und Augen frei für andere Tätigkeiten.

Eine weitverbreitete Akzeptanz der Sprachinteraktion als Mensch-Maschine-Schnittstelle muss jedoch erst noch erfolgen. Um dieses Ziel zu erreichen, werden viele Anstrengungen unternommen. Die Bank of America möchte beispielsweise, einem aktuellen Bericht [12] zufolge, in den Staaten Florida und Kalifornien mehr als 2500 "sprechende" Geldautomaten aufstellen lassen. Diese werden sich dann auch von Blinden und sehbehinderten Menschen problemlos bedienen lassen. Informationen, die bisher nur auf dem Bildschirm sichtbar waren, werden über Sprachsynthese ausgegeben.

Auch im Bereich der Desktop-Betriebssysteme wird sich in den nächsten Versionen bzgl. der Spracherkennung vieles ändern: Microsoft sich im letzten Jahr in zwei Firmen eingekauft, die sich mit der Entwicklung von Software für Spracherkennung befassen [1]. Eine davon, *Lernout & Hauspie*, ist eine der führenden

[1] *durchdringend*; Sprache kann physische Hindernisse durchdringen oder umgehen

Hersteller in diesem Marktsegment. Dies lässt vermuten, dass in einer der nächsten Versionen der graphischen Benutzeroberfläche *Windows* Spracherkennung als basale Interaktionsform integriert sein wird. Interessant wird sein, wie weit dann Sprache, Tastatur und Maus im Sinne einer echten Multimodalität verbunden sein werden. Im Sinne einer echten Multimodalität könnte der Benutzer dann sagen "Drucke mir dies aus" und gleichzeitig mit der Maus auf das grafische Symbol einer Datei zeigen. Diesen Ansatz hat der Flugzeughersteller Boeing beim AWACS[2] verwirklicht. Dort kann ein Operator Befehle äußern, während er gleichzeitig mit der Maus ein grafisches Objekt auswählt, um den Kontext des Kommandos festzulegen.

Neben den Betriebssystemen betrifft diese Entwicklung auch die Anwendungssoftware. Das W3 Consortium arbeitet z.B. seit letztem Jahr an einer Spezifikation für *Voice Browsers* [84]. In Textverarbeitungsprogramme integrierte Diktiersysteme sind mittlerweile alltägliche Applikationen.

Auch in mobile Endgeräte werden zunehmend sprachliche Benutzerschnittstellen integriert. So gibt es bereits einige Mobiltelefone, die eine Verbindung aufbauen, wenn man den Namen des gewünschten Gesprächspartners in das Telefon spricht. Eine andere Entwicklung in Richtung mobiler Spracherkennung unternimmt IBM. Auf der Konferenz *Mobile Insights 2000* wurde der Prototyp eines Palmtops auf der Basis des Palm III von 3Com gezeigt, der über eine sprachliche Benutzerschnittstelle verfügt [45]. Mittels eines Sprachsynthesemoduls kann man sich beliebig lange E-Mails und Nachrichten vorlesen lassen. Die integrierte Spracherkennung, eine Embedded-Version von IBMs ViaVoice, erkennt etwa 500 Wörter. Damit lassen sich kurze Memos in das Gerät sprechen und Steuerbefehle an den Palmtop per Sprache geben.

Nach Ansicht des Autors ist zu erwarten, dass die Integration von Sprache in den Bereichen der Anwendungsprogramme und der mobilen Endgeräte in den kommenden Jahren stark zunehmen wird. Die sich daraus ergebende Konsequenz ist, dass diese Schnittstellen semantisch flexibler gestaltet werden, um dem Benutzer eine immer freiere Formulierung seiner Befehle zu erlauben.

[2] Airborne Warning and Control System, ein speziell ausgestattetes Flugzeug, das in großer Höhe fliegt und der großflächigen Luftaufklärung dient. Siehe `http://www.boing.com/dsg.awacs.html`

3.2 Sprachverarbeitung

Die Forschung über Systeme zur Verarbeitung der menschlichen Sprache gliedert sich in zwei große Teilbereiche, die *Spracherkennung* und das *Sprachverstehen*. Da das deutsche Wort *Verstehen* umgangssprachlich leider sowohl für *akustisches*, wie auch *semantisches* Verstehen verwendet wird, trifft man die eindeutigeren englischen Bezeichnungen *Recognition* und *Understandig* auch häufig in der deutschsprachigen Literatur an. Ebenso verhält es sich mit dem deutschen Wort *Sprache*. Man verwendet es für die Fähigkeit zu sprechen, wie auch für die Landessprache, also die zugrundeliegende Grammatik und das Vokabular. Im Englischen unterscheidet man hier zwischen *speech* und *language*, und auch das Französische kennt zwei Begriffe – *parole* und *langue*.

Der gesamte Forschungsbereich des *Natural Language Processing* (NLP) teilt sich auf in die beiden Teilgebiete *Natural Language Recognition* (NLR) und *Natural Language Understanding* (NLU). Der Begriff *Natural Language* wird im Deut-

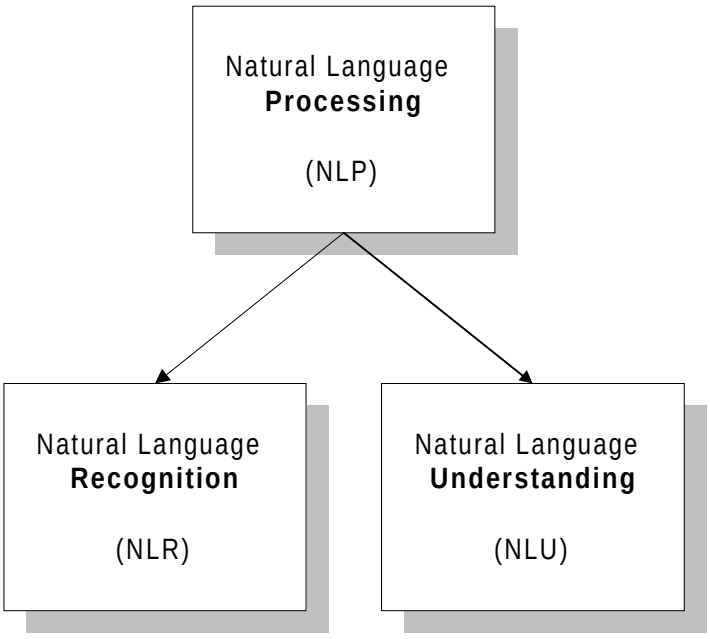


Abbildung 3.1: Forschungszweige der Spracherkennung

schen mit *Natürliche Sprache* übersetzt. Man verwendet diesen Begriff, um die

Sprache des Menschen von den formalen Sprachen zu unterscheiden.

Im Bereich des *Natural Language Recognition* geht es darum, aus gesprochener Sprache eine textuelle Repräsentation des Gesagten zu generieren. Der Zweig des *Natural Language Understanding* versucht, aus einem natürlich-sprachlichen Text den semantischen Inhalt abzuleiten.

In aktuellen sprachverstehenden Systemen wird der Vorgang des Verstehens gesprochener Sprache typischerweise in zwei Schritten durchgeführt. Zunächst werden Hypothesen für potentiell gesprochene Wörter generiert. In der zweiten Verarbeitungsstufe wird dann versucht, aus diesen Hypothesen eine möglichst eindeutige semantische Interpretation der gesprochenen Äußerung zu extrahieren. Zunächst wird die Sprache also *erkannt* und dann *verstanden* (vgl. Abbildung 3.2). Obwohl das zweistufige Verfahren des Verstehens durch die klaren Schnittstellen

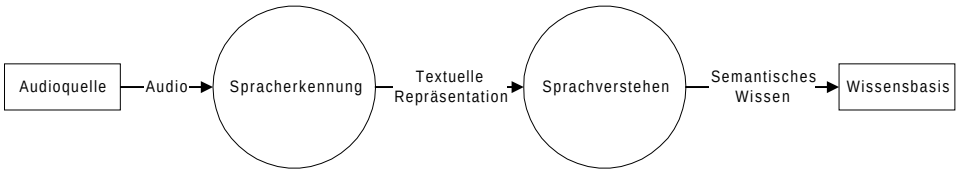


Abbildung 3.2: Gesprochene Sprache wird in einem zweistufigen Verstehensprozess in verschiedene Repräsentationen überführt.

am Ein- und Ausgang der Verarbeitungsstufen leicht zu handhaben ist, birgt sie doch einen großen Nachteil. Da sich der Prozess des Verstehens nur auf die textuelle Repräsentation des Gesagten stützt, geht jede Information über Prosodie, Intonation und Sprechrhytmik verloren. Beispielweise hat der Satz *"Ich werde morgen kommen!"* jeweils eine völlig unterschiedliche Bedeutung, je nachdem, ob er als Drohung oder als Versprechen betont wird.[3]

Trotz dieses Nachteils ist die zweistufige Verarbeitung momentan das am weitesten verbreitete Verfahren [18], da sich so das äußerst komplexe Problem der Sprachverarbeitung gut in Teilprobleme mit geringerer Komplexität teilen lässt. Systeme, die

[3]Man spricht in diesem Zusammenhang von *Sprechakten*. Der Begriff Sprechakt (engl.: *speech act*) ist ein von AUSTIN eingeführter Begriff, um den Handlungsakt sprachlicher Äußerungen zu bezeichnen. AUSTIN untersuchte, inwiefern mit dem Äußern von Sätzen Handlungen vollzogen werden [2].

beide Verarbeitungstufen integrieren, sind momentan noch Forschungsgegenstand. FINK widmet sich diesem Problem in seiner Dissertation [18] über die "Integration von Spracherkennung und Sprachverstehen" und gibt einen Überblick über Forschungsbestrebungen in diesem Bereich.

3.2.1 Aufbau eines sprachverstehenden Systems

Das Vermögen, gesprochene Sprache zu verstehen, lässt sich in vier hierarchisch angeordnete Ebenen unterteilen:

1. Phonetische Ebene
2. Syntaktische Ebene
3. Semantische Ebene
4. Pragmatische Ebene

Auf der phonetischen Ebene werden aus den einzelnen Lauten Wörter gebildet. Die syntaktische Ebene stellt die Ebene der Grammatiken dar; sie legt fest, nach welchen Regeln aus den einzelnen Wörtern Wortketten gebildet werden können. Die semantische Ebene setzt voraus, dass Sender und Empfänger dieselben Wörter für dieselben Begriffe verwenden und bestimmt welche Bedeutung den Wörtern bezüglich ihrer Position in einem Satzes zukommt. Die pragmatische Ebene schließlich dient der Kontrolle der Kommunikation. Hierzu zählen vor allem die Aufmerksamkeitskontrolle des Zuhörers, der Blickkontakt und die Körpersprache.

Grundsätzlich müssen alle sprachverstehenden Systeme — dazu gehört auch der Mensch — mindestens über die ersten drei der vier Verstehensebenen verfügen. Bemerkenswert ist, dass sich dies auf *gesprochene*, aber nicht notwendigerweise *natürliche* Sprache bezieht.[4] Spracherkennende Systeme (NLR) beschränken sich auf die erste Ebene, sprachverstehende Systeme (NLU) beschäftigen sich vor allem mit der zweiten und dritten Ebene.

[4] Ein Beispiel für gesprochene *formale* Kommunikation zwischen Menschen ist der Funksprechverkehr in der Luftfahrt. Hier wird über formalisierte Buchstabencodes miteinander kommuniziert. Diese Form der Kommunikation ist nicht natürlich-sprachlich, sondern man könnte sie als *formalsprachlich* bezeichnen.

3.2.2 Leistungsfähigkeit spracherkennender Systeme

Unter Spracherkennung[5] versteht man das Erkennen gesprochener Sprache. Typischerweise wird Sprache von einer Audioquelle (Mikrophon, Audiodatei) entgegengenommen und in eine textuelle Repräsentation gebracht. Diese eignet sich dann für eine Weiterverarbeitung im Rechner.

Das erste Spracherkennungssystem wurde 1952 entwickelt, und war in der Lage, die Ziffern 0 bis 9 anhand eines (sprecherabhängigen) Mustervergleiches zu erkennen. Die Erkennungsrate dieses Systems wird mit 98% angegeben. Noch in der gleichen Dekade wurde dann ein ähnliches System für Konsonanten und Vokale realisiert. Beide Systeme wurden noch mit analogen Rechnern aufgebaut, erst in den 60er Jahren ging man zu digitalen Rechnern über. Trotz der guten Anfangserfolge in der Forschung um die Spracherkennung, verhinderten Beschränkungen der Rechnerarchitektur hinsichtlich der Performance einen signifikanten kommerziellen Erfolg spracherkennender Systeme. Denn obwohl die Datenübertragungsrate natürlich gesprochener Sprache nur etwa 50 Bits pro Sekunde entspricht, sind die Anforderungen an einen Rechner für die Merkmalsextraktion enorm. Während der letzten Jahre sind einige kommerzielle Systeme entwickelt worden, die sich nun auch in alltäglichen Anwendungen etablieren. Trotzdem sind heutigen Spracherkennungsprogrammen noch Beschränkungen auferlegt. Zu diesen gehören die Sprecherabhängigkeit, die notwendige Vermeidung von Hintergrundgeräuschen, die Landessprache und der Umfang des Vokabulars. (vgl. [35])

Man unterteilt spracherkennende System in drei Gruppen. *Sprecherunabhängige* Systeme sind in der Lage, die Eingaben eines beliebigen Sprechers zu erkennen. Typischerweise ist ihr Vokabular sehr begrenzt und man verwendet sie daher nicht zum Erkennen flüssig gesprochener Texte, sondern beschränkt sich auf kurze Befehle. *Sprecherabhängige* Systeme müssen vor der ersten Verwendung vom Benutzer auf die eigene Stimme trainiert werden. Typischerweise verfügen sie über ein wesentlich umfangreicheres Vokabular und eine höhere Erkennungsgenauigkeit als sprecherunabhängige Systeme. Einen Zwischenweg beschreiten die *adaptiven* Systeme. Sie gehen von sprecherunabhängigen Sprachmustern aus und passen sich mit der Zeit an einen spezifischen Benutzer an. Diese hybriden Systeme kommen

[5] engl.: *speech recognition*

daher ohne vorheriges Training aus (vgl. [35]).

Kommerzielle Diktiersysteme sind mittlerweile in der Lage, flüssig gesprochene Sprache mit einer Genauigkeit von mehr als 90% zu erkennen — bis vor kurzem war es bei Diktiersystemen noch notwendig, nach jedem Wort eine kurze Pause zu machen. Der Umfang aktueller Vokabeldatenbanken von kommerziellen Systemen liegt bei ca. 400.000 Wörtern und die Erkennungsgeschwindigkeit beträgt 160 Wörter pro Minute [44]. Eine Evaluation aktueller Diktiersysteme wird in [51] besprochen.

3.2.3 Spracherkennung

Dieser Abschnitt soll einen Einblick in das gängigste Verfahren der Spracherkennung gegeben werden. Eine detaillierte Beschreibung des Vorgangs der Spracherkennung würde innerhalb dieser Arbeit zu weit führen. Detaillierte Informationen findet man z.B. in [67], [18] und [48].

Die Überführung eines Audiosignals in eine textuelle Repräsentation des Gesagten erfolgt in mehreren Schritten. Die vom Mikrofon aufgenommenen Signale werden in einem ersten Schritt von dem auf der Soundkarte befindlichen A/D-Wandler digitalisiert. Aus der Folge von Amplitudenwerten werden nun vom Spracherkennungssystem kurze Abschnitte extrahiert. Diese Fenster haben im Zeitbereich eine Breite von mehreren Millisekunden und werden meist mit einer Filterfunktion an ihren Rändern gedämpft, um ungewollte Verzerrungen im Frequenzspektrum zu vermeiden. Im nächsten Schritt wird versucht, Hintergrundgeräusche herauszufiltern oder zu dämpfen. Aus diesem Signalanschnitt wird ein Frequenzspektrum erzeugt. Diese wird mit einer frequenzabhängigen Gewichtung versehen, die sich an der spektralen Empfindlichkeit des menschlichen Gehörs orientiert.

Abbildung 3.3 zeigt die spektrale Verteilung einzelner, ähnlich klingender Wörter. Auf der Ordinate ist die Zeit in Millisekunden aufgetragen, die Abszisse stellt linear die Frequenz dar. Nacheinander wurden von einem Sprecher die Worte *heed*, *head*, *had* und *who'd* gesprochen. Die spektrale Verteilung der Audiosignale zeigt deutlich, dass auch ähnlich klingende Wörter ein sehr unterschiedliches Frequenzspektrum haben. Dies ist eine gute Ausgangsbasis für automatische Spracherkennungssysteme.

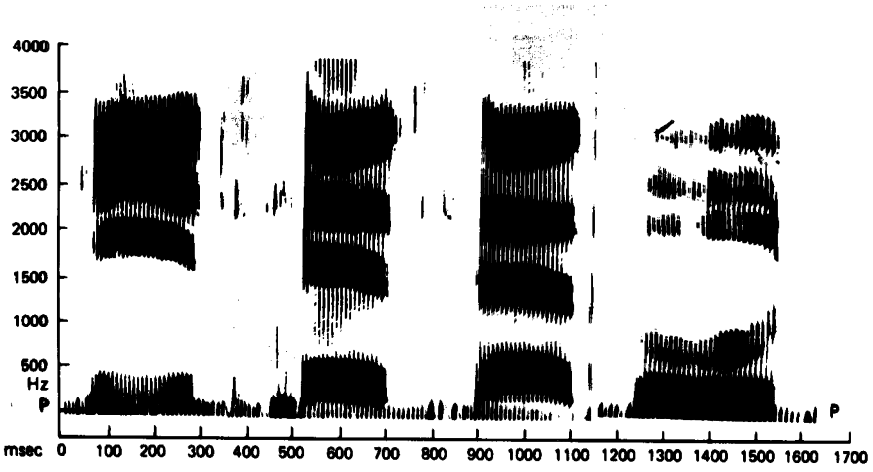


Abbildung 3.3: Spektrogramm der Wörter *heed, head, had* und *who'd*, gesprochen mit britischem Akzent. Entnommen aus [28]

Nach der Signalaufbereitung wird in der folgenden Verarbeitungsphase das Signal in sinnvolle Untereinheiten eines Wortes zerlegt, die Phoneme. Dieser Schritt ist wesentlich für die Erkennung, denn aus den erkannten Phonemen werden dann wiederum die Wörter aufgebaut. Das grundsätzliche Problem der Spracherkennung ist, dass man in einem kontinuierlichen Audiostrom Wortgrenzen fast nie am Signal selbst erkennt. Daher wählt man den Weg über die Phoneme. Es gibt je nach Landessprche gewisse Wahrscheinlichkeiten für das Aufeinanderfolgen zweier Phoneme. Diese Wahrscheinlichkeiten macht man sich zunutze, um zunächst die Phoneme sicher zu erkennen. Aus den Phonemgruppen lassen sich auf die gleiche Weise Wörter und schliesslich Sätze ableiten.

Dabei kommen Hidden-Markov-Modelle (HMM) zum Einsatz. Ein Hidden-Markov-Modell ist ein endlicher stochastischer Automat. Er besteht aus diskreten Zuständen und Zustandsübergängen. Die Zustandsübergänge sind mit Wahrscheinlichkeiten gewichtet. Abbildung 3.4 zeigt ein Hidden-Markov-Modell des englischen Wortes *tomato*. Die Zustände stehen für mögliche Phoneme eines Wortes, die Übergänge zwischen den Phonemen sind mit Wahrscheinlichkeiten gewichtet. Die Spracherkennung versucht, ausgehend von einem Startzustand, entsprechend den ermittelten Phonemen, dem Pfad zu folgen. Dabei besteht das Vokabular aber nicht aus einem Wort, wie in diesem Beispiel, sondern aus einigen Hundertausend.

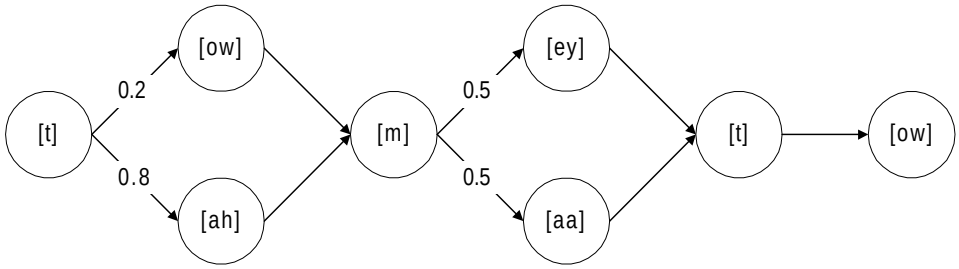


Abbildung 3.4: Beispiel eines Markov-Modells für das Wort *tomato*

Alle Wörter des Vokabulars sind intern in Phoneme und Übergangswahrscheinlichkeiten zerlegt. Gibt es mehrere Möglichkeiten, eine Phonemkette zu interpretieren (z.B. als *head* oder *had*), hilft die Gesamtwahrscheinlichkeit der Phonemübergänge bei der Entscheidung für eine der Worthypothesen.

Aber schon auf der Ebene der Phonemerkennung kommen Hidden-Markov-Modelle zum Einsatz. Jedes Phonem lässt sich mittels eines stochastischen Automaten beschreiben. Hier werden allerdings Folgen von Spektralanteilen auf Phoneme abgebildet. Zur Erkennung von Sätzen kann das Verfahren mittels stochastischer Automaten ebenfalls verwendet werden. Einzelne Wörter sind dabei Zustände von Wortketten oder syntaktischen Einheiten. Wendet man das Prinzip der Abstraktion an, erkennt man, dass ein komplettes Hidden-Markov-*Modell* zu einem *Zustand* des Modells auf der nächsthöher gelegenen Ebene abstrahiert wird.

Hidden-Markov-Modelle werden wegen ihrer Flexibilität und ihrer wohldefinierten Trainingsmöglichkeiten sehr erfolgreich für die Spracherkennung eingesetzt. Beim Trainieren der Spracherkennung lässt man den Benutzer vorgegebene Wörter vorlesen und passt nur die Wahrscheinlichkeiten der Zustandsübergänge entsprechend der Sprechproben an.

3.2.4 Sprachverstehen

Beim Verstehen natürlicher Sprache geht es darum, die Semantik eines Satzes zu erfassen. Diese Aufgabe ist nach wie vor ein großes, bisher nicht umfassend gelöstes Problem. Die Interpretation natürlich gesprochener Sprache ist seit der Ge-

burtsstunde[6] des Forschungsgebietes der “Künstlichen Intelligenz” 1956 eines der wichtigsten Gebiete der KI-Forschung. Das augenblicklich bedeutendste deutsche Projekt in diesem Forschungszweig ist VERBMOBIL des Deutschen Forschungsinstituts für Künstliche Intelligenz (DFKI) in Saarbrücken. Inhalt des vom BMBF geförderten Projektes ist die Übersetzung spontan gesprochener Sprache in eine andere Landesprache[7]. Detaillierte Informationen zum Stand der Forschung findet sich auf der Internetseite des DFKI unter [14].

Bei der Übersetzung in eine andere Sprache muss zunächst die Semantik eines Satzes erfasst werden, bevor er in eine andere Sprache gebracht werden kann. Die Erfassung der Semantik wird in einer Dialog-Situation zusätzlich dadurch erschwert, dass der Mensch nicht immer alle zum Verständnis benötigten Informationen nennt. Entweder stützt sich der Sender auf das i.d.R. vorhandene Weltwissen des Rezipienten, oder er referenziert (semantisch) frühere Äußerungen.

Ein Computerprogramm soll Sprache nun in dem Sinne verstehen, dass es eine sprachliche Eingabe in eine interne Repräsentation übersetzt, die es ihm ermöglicht, angemessen zu reagieren.

> “It is less important [...] whether the machine ‘understands’ its natural-language input in a cognitively plausible way than whether it responds to the input in a way helpful to the user and in accordance with the desires expressed in the input.” [10]

Es ist also nicht wichtig, ob der Computer die Eingabe im kognitiven Sinne “versteht”, sondern ob die Reaktion der Eingabe angemessen ist. Dies erinnert stark an den von TURING eingeführten Test zur Messung der Fähigkeiten eines KI-Systems (Turing-Test). Andererseits ist das Ermessen der Reaktionen des Rezipienten genau die Methode, die auch der Mensch intuitiv anwendet, um das Verständnis seiner eigenen Aussagen zu überprüfen. An diesem Punkt entzünden sich zahlreiche philosophische Diskussionen. Stellvertretend sei hier aus WITTGENSTEINS “Philosophischen Untersuchungen” zitiert:

[6] Als offizielle Geburtsstunde des Forschungsgebietes KI gilt eine Konferenz, die im Sommer 1956 am Dartmouth College stattfand.

[7] VERBMOBIL leistet eine Übersetzung zwischen Deutsch und Englisch und zwischen Deutsch und Japanisch.

> "Wie ein Wort funktioniert, kann man nicht erraten. Man muss seine Anwendung ansehen und daraus lernen."[86]

WITTGENSTEIN meint damit, dass es beispielsweise keinen einheitlichen Begriff *Zeit* gibt, sondern dass das Wort "Zeit" eine unterschiedliche Verwendung hat, je nachdem, ob man eine Verabredung trifft oder Zeiträume misst. Diese Zuordnung von einem Begriff zu einem konkreten oder abstrakten Objekt kann man nicht erlernen, sondern nur anhand der Reaktionen des Zuhörers erfahren.

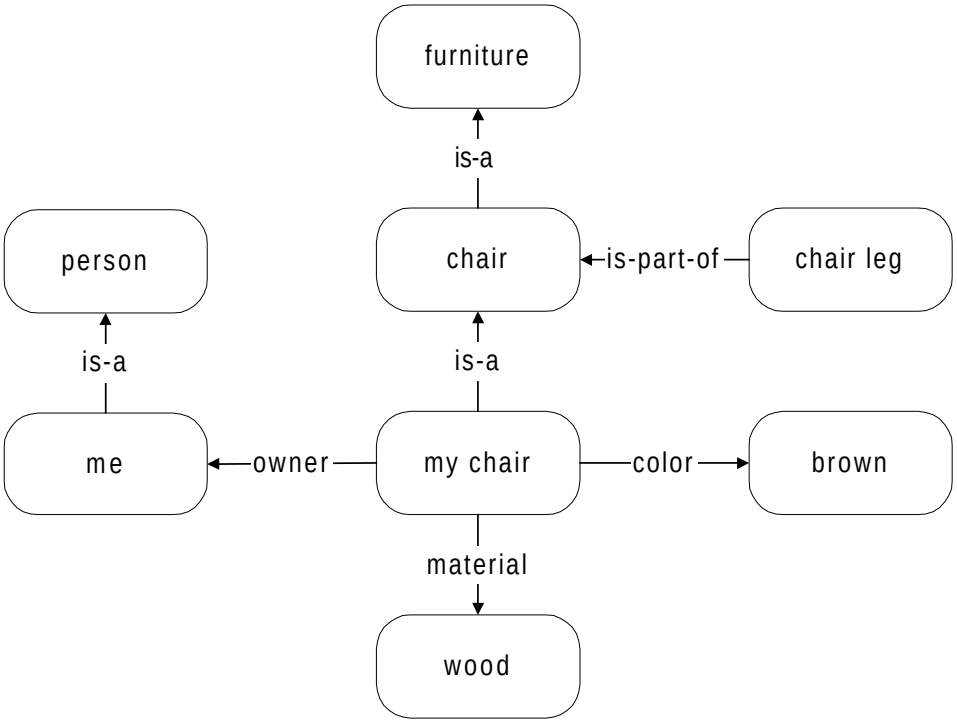


Abbildung 3.5: Beispiel eines semantischen Netzwerkes. Nach [85]

Um überhaupt semantisches Wissen im Rechner repräsentieren zu können, werden häufig semantische Netze verwendet. Sie bestehen aus einer Menge von Knoten, die durch gerichtete, bezeichnete Kanten miteinander verbunden sind. Die Knoten repräsentieren begriffliche Objekte, die Kanten Relationen zwischen diesen Objekten. Es wird zwischen vier verschiedenen Arten von Beziehungen unterschieden:

- A ist eine Teilmenge von B (*is-a*)

- A ist ein Exemplar der Klasse B (*instance-of*)
- A ist ein Teil von B (*part-of*)
- B ist ein Attribut von A

Abbildung 3.5 zeigt ein Beispiel eines semantischen Netzwerkes. Es zeigt, wie der Begriff “my chair” implizit mit anderen Begriffen verbunden ist. Über dieses Wissen muss ein sprachverstehendes System ebenfalls verfügen, um die Bedeutung einer Aussage wie “My chair has a broken leg” einordnen zu können. Mit Hilfe logischer “Wissensoperationen” kann dann im semantischen Netz ermittelt werden, dass Stuhlbeine Teil eines Stuhls sind.

Semantische Netzwerke werden in der Praxis stets in eng umrissenen Kontexten eingesetzt, da sie sonst zu groß würden.

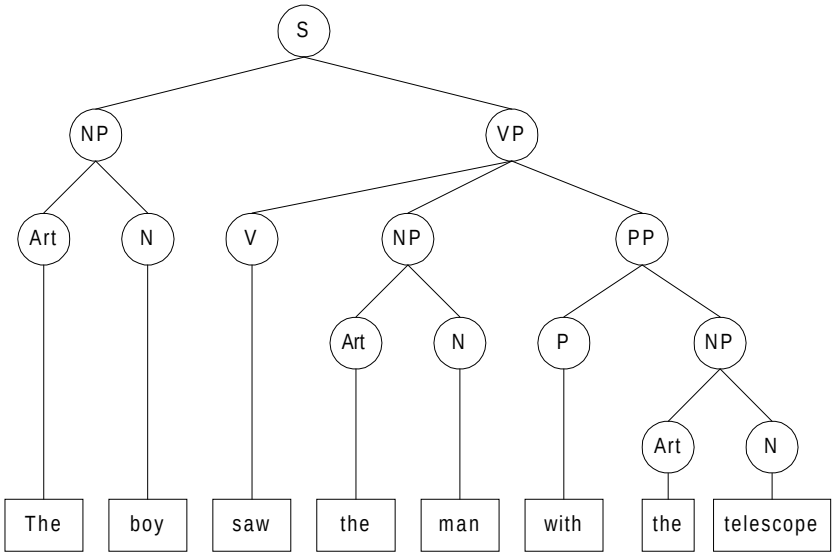
3.2.5 Realisierungsformen für sprachverstehende Systeme

In der Praxis finden sich verschiedene Realisierungsformen für sprachverstehende Systeme:

- Wordspotting
- Syntaxanalyse
- Reguläre Grammatiken

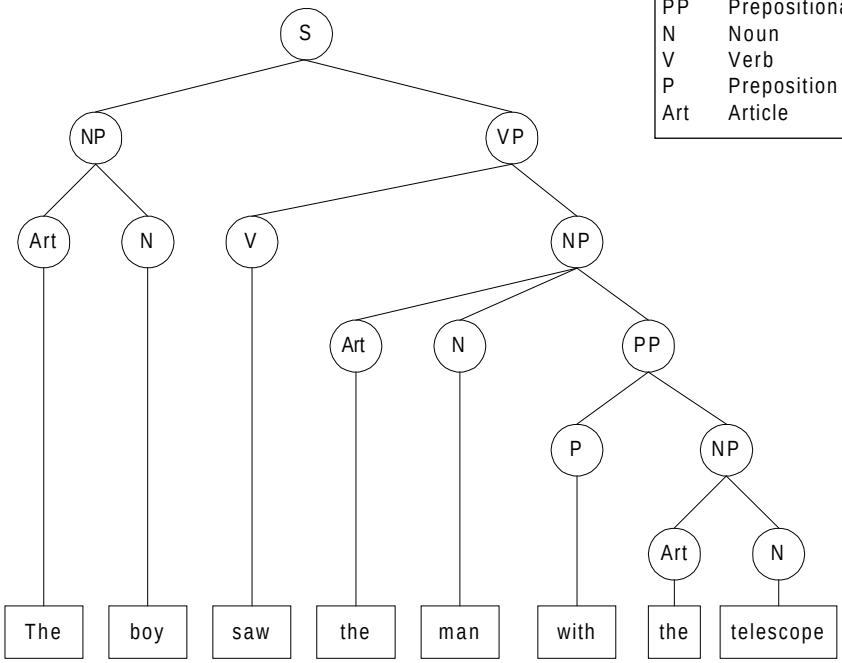
Wordspotting. Das Wordspotting ist ein Verfahren, bei dem in einer Aussage nur nach bestimmten Schlüsselwörtern in einem Satz gesucht wird. Der Rest des Satzes, und sogar dessen Aufbau[8], wird ignoriert. Der Vorteil des Wordspottings ist, dass die Auswertung sehr schnell erfolgen kann, sie lässt sich leicht implementieren und schränkt den Benutzer nicht in der Wahl seiner Formulierung ein. Leider ist das Verfahren, bezüglich der Verständnisrate, sehr ungenau. Auch stellen Negationen im Satz ein Problem dar. Das Wordspotting-Verfahren eignet sich gut für Auskunftssysteme, da die Benutzer das System ohne Vorwissen verwenden können und der Einsatz stets in einem eng umgrenzten sprachlichen Umfeld erfolgt.

[8] Manchmal wird auch die zeitliche Reihenfolge, in der die Schlüsselwörter vorkamen, in Betracht gezogen; von der Auswertung eines *Satzbaus* kann dabei aber wohl kaum gesprochen werden.



Meaning: The boy used a telescope to see the man.

S	Sentence
NP	Noun Phrase
VP	Verb Phrase
PP	Prepositional Phrase
N	Noun
V	Verb
P	Preposition
Art	Article



Meaning: The boy saw a man who was carrying a telescope.

Abbildung 3.6: Syntaxbäume eines mehrdeutigen Satzes. Nach [28]

Syntaxanalyse. Bei der Syntaxanalyse wird die Satzstruktur anhand grammatikalischer Regeln in einen Syntaxbaum zerlegt. Man kann dadurch eine Aussage machen, was z.B. das Subjekt, Verb und Objekt eines Satzes sind, ob er Nebensätze hat und wie diese wiederum aufgebaut sind. Die Syntaxanalyse ist sehr erfolgreich, solange die Sätze eine gewisse Komplexität nicht übersteigen. Sie lässt sich zudem gut mit semantischen Netzen kombinieren. Das Parsen der Sätze ist sehr aufwendig und die Resultate sind bei mehrdeutigen Sätzen nicht eindeutig. Abbildung 3.6 zeigt ein Beispiel eines ambigen Satzes und die zugehörigen Syntaxbäume. In diesem Beispiel ist es nicht klar, ob sich die Präpositionalphrase (PP) auf das Verb "saw" oder das Objekt "the man" bezieht. Hier stößt man an die Grenzen der Syntaxanalyse, denn diese Entscheidung kann nur aufgrund des sprachlichen Kontextes getroffen werden. Trotzdem ist die Syntaxanalyse ein sehr leistungsfähiges, wenn auch aufwendiges Verfahren. Es wird verwendet, wenn es darum geht, Sprache zu interpretieren, die nicht an einen Kontext gebunden ist und frei formuliert wird.

Reguläre Grammatiken. Beim Verstehen natürlich gesprochener Sprache mit Hilfe regulärer Grammatiken wird die Menge der möglichen Eingaben durch eine formale Grammatik definiert. Dies ist natürlich nur in einem eingeschränkten Dialogkontext möglich. Der Anwender des Systems muss sich dann mit seinen Äußerungen an dieser Grammatik orientieren. Die Grammatiken lassen sich für einen definierten Dialogkontext sehr umfangreich gestalten, so dass der Benutzer u.U. gar nicht merkt, dass eine Einschränkung bezüglich seiner Wortwahl und seines Satzbaus besteht. Man spricht daher auch von einer *Pseudo-NL*[9]. Die Verständnisrate eines solchen Systems ist sehr hoch und der Aufwand zur Realisierung verhältnismäßig gering. Der Nachteil ist, dass der Benutzer des Systems angelernt werden muss. Dies ist normalerweise kein expliziter Prozess, sondern wird in die Dialogführung integriert. Dies geschieht mit Hilfe *inkrementaler Prompts* (siehe Abschnitt 3.3.1).

[9] Natural Language

3.3 Dialoge

Ein noch umfassenderes Problem als das des Sprachverstehens ist das maschinelle Führen von Dialogen. Hier muss neben dem Verstehen der individuellen Sätze das Problem der Dialogführung gelöst werden, da jeder der Beteiligten eine gewisse Kontrolle über den Ablauf eines Gespräches hat. Zusätzlich zum Verständnis der einzelnen, semantisch voneinander abhängigen Sätze, können auch alle semantischen Missverständnisse auftreten, die aus der Kommunikation zwischen Menschen bekannt sind [28]. Dialoge, die zwischen einem Menschen und einer Maschine geführt werden, müssen darüber hinaus noch weitergehende Forderungen erfüllen. Diese werden im folgenden Abschnitt 3.3.1 besprochen. Abschnitt 3.3.2 geht dann auf den Aspekt der Multimodalität in Dialogen ein.

3.3.1 Anforderungen an ein Dialogsystem

Wie ein guter computergeführter Dialog aussehen muss, hängt stark von der Art, dem Inhalt und der Zielsetzung des Gespräches ab. Bei dem Dialog kann es sich beispielsweise handeln um:

- den Frage-Antwort-Dialog eines Auskunftssystems,
- einen Dialog zum Unterrichten oder
- einen Dialog zum Überzeugen bzw. Werben.

Natürlich gelten für einen automatisierten Dialog alle Regeln der Rhetorik, die auch für einen zwischenmenschlichen Dialog gelten. Weiterhin gelten natürlich auch die Regeln für eine gute Benutzerschnittstelle eines Softwaresystems. Dies schließt z.B. ein, dass bei der Ausgabe eines Ergebnisses auch die vom Benutzer getätigten Eingaben nocheinmal dargestellt werden. Und einem dritten Anspruch muss ein guter Dialog genügen: der Mensch ist bei sprachlicher Interaktion mit anderen Menschen an eine unmittelbare Rückkopplung in Form von nonverbalen Signalen, Fragen, Anregungen, etc. gewöhnt. Diese Erwartungen des Anwenders an ein Dialogsystem müssen von den Entwicklern des Dialogs berücksichtigt werden. ZINK geht in [88] auf diese Problematik ein:

> "Weil sowohl das System (Systemfehler), als auch der Anwender, Fehler machen können (Anwenderfehler), muss das System beide Fehlerarten erkennen, dem Benutzer rückmelden und entsprechend reagieren. [...] Ein gut konstruiertes Dialogmodul ist genauso wichtig wie ein leistungsfähiger Spracherkenner und ein gutes NLU-Modul."

In der Praxis muss ein Dialogsystem mit verschiedenen Arten von Benutzern umgehen können. Ein Benutzer kann kooperativ, nicht-kooperativ, geschult und nichtgeschult sein. Um sich an das Niveau der Benutzerkenntnisse anzupassen, verwendet man *inkrementale Prompts*. ZINK beschreibt dies am Beispiel eines Telefon-Dialogsystems:

> "Am Anfang hört der Benutzer einen allgemeinen kurzen Prompt (implicit prompt), wie etwa: 'Was kann ich für sie tun?'. Der unerfahrene Anwender wird darauf womöglich nicht antworten, weil er die Möglichkeiten des Systems nicht kennt. Dieses unterstützt ihn dann mit einem etwas spezifischeren Prompt (incremental prompt), etwa: 'Sie können entweder Fluginformationen abrufen oder einen Flug buchen. Was wünschen Sie?' Jetzt weiß der Anwender, was er machen kann, aber noch nicht wie. Kommt erneut keine Antwort von ihm, so wird das System einen sehr ausführlichen Prompt (explicit prompt) abspielen, der alle Befehlsmöglichkeiten auflistet: 'Ich konnte Sie leider nicht hören. Folgende Möglichkeiten stehen zur Verfügung: Fluginformation oder Flug buchen. Bitte sagen Sie die gewünschte Aktion. Sollten Sie ein persönliches Gespräch vorziehen, so sagen Sie bitte persönliches Gespräch.' Der Anwender hat nun die benötigten Informationen, um eine Antwort zu geben."

Die Form des Dialoges ist also entscheidend für die Benutzbarkeit und damit für die Akzeptanz des Systems. Zusammenfassend kann man postulieren, das ein "guter Dialog" folgende Bedingungen erfüllt:

- Einhaltung rhetorischer Randbedingungen
- Wiederholung der vom Benutzer gemachten Eingaben in der Antwort

- Überwachung des Gesprächs und Meldung von Bedien- und Systemfehlern
- Adaption an die Vorkenntnisse des Benutzers

3.3.2 Multimodalität

Bei der Kommunikation benutzt der Mensch meist mehrere Sinne und Kanäle gleichzeitig, um die Variation seiner Ausdrucksmöglichkeiten zu erweitern. Die wohl am meisten verwendete Kombination sind Sprache und Mimik, aber auch Körperhaltung, Verklanglichung und Visualisierung sind Ausdrucksformen menschlicher Kommunikation. Die Kombination mehrerer Ausdrucksformen erhöht die Effektivität der Kommunikation hinsichtlich der Geschwindigkeit als auch der semantischen Ausdrucksmöglichkeit.

Worin besteht nun der Unterschied zwischen den Begriffen *Multimedialität* und *Multimodalität*? LENZMANN führt in [47] bezüglich dieser Fragestellung aus:

> "Üblicherweise bezeichnet *Medium* einerseits sowohl das Objekt, das Informationen eines bestimmten Typs enthält, als auch den zur Übertragung notwendigen Kommunikationskanal. [...]
>
> Für den Begriff der *Modalität* (oder des *Modus*) lassen sich unterschiedliche Definitionen finden. In einer psychologischen Bedeutung werden damit die Kategorien menschlicher Sinne wie Sehen, Hören, Fühlen, etc. referenziert, folglich die Art und Weise, wie Menschen Information kommunizieren und aquirieren. Eine zweite Auslegung betrachtet den Zustand der Interaktion, wie beispielsweise einen Einfügemodus innerhalb eines Texteditors. In einer dritten, am häufigsten verwendeten Sichtweise werden Modalitäten als Interaktionstechnik verstanden, die dem Benutzer erlaubt, ein bestimmtes Ziel zu erreichen. So bezeichnen Tastatur-, Sprach- und Gesteneingabe oder Mausselektion unterschiedliche Eingabemodalitäten."

Dieser Definition folgend, bezieht sich *Multimedialität* auf die physischen Medien der Interaktion, während sich *Multimodalität* auf den abstrakten Interaktionskanal bezieht.

Multimodalität hat für den Benutzer eines Systems den Vorteil, dass er verschiedene Eingabekanäle angeboten bekommt und sich für den (oder die) Effektivsten entscheiden kann. Multimodale Systeme wenden dabei einen Synchronisations-, oder besser Unifikationsprozess an, um die Information mehrerer Kanäle zu einer Information zu verschmelzen. Wenn der Benutzer beispielsweise äußert “Bewege dieses Objekt nach dort” und gleichzeitig mit der Maus auf dem Bildschirm zwei Positionen markiert, müssen die verbale und die grafische Information miteinander verknüpft werden, um die Aktion ausführen zu können. Dieser Vorgang kann je nach Aufgabe und verwendeten Medien sehr schwierig sein [57].

Kapitel 4

Semantische Protokolle

Dieses Kapitel beschäftigt sich mit dem Austausch von Wissen über eine (Agenten-) Kommunikationssprache. Dieses Thema wird vor dem Hintergrund behandelt, dass in dem zu entwickelnden Dialogsystem die gesprochene Eingabe des Benutzers in eine Agentenkommunikationssprache übersetzt wird; dies erleichtert dem Agenten das Verständnis.

Im Abschnitt 4.1 wird erläutert, was unter einem semantischen Protokoll zu verstehen ist, und aus welchen Teilen sich eine Nachricht zusammensetzt. Die darauf folgenden Abschnitte gehen dann auf diese drei Teile, das Kommunikationsprotokoll (Abschnitt 4.2), die Syntax zur Wissensrepräsentation (Abschnitt 4.3) und die Ontologie (Abschnitt 4.4), gezielt ein. Dabei werden die jeweils wichtigsten Protokolle vorgestellt.

4.1 Begriffsdefinition

Unter einem *semantischen Protokoll* versteht man eine formale Sprache, die dem Austausch von Wissen dient.

Kommunikation zwischen Systemen oder Individuen dient den beteiligten Kommunikationspartnern *immer* dazu, ihr Wissen zu teilen oder zu erweitern. Es gibt also keine Kommunikation, bei der keine Information ausgetauscht wird. Selbst eine Begrüßung unter Freunden soll die Empfänger der Nachricht auf die Anwesenheit des Senders und dessen Bereitschaft an einer Kommunikation aufmerksam

machen. WATZLAWICK fasst dies in seinem ersten Axiom der Sprachtheorie sogar noch weiter und formuliert: “Man kann nicht nicht kommunizieren”. Aus kommunikationstheoretischer Sicht erfüllt die Kommunikation zwar meist auch noch den Zweck der Einhaltung von sozialen Regeln und Umgangsformen, dies hat aber aus informatischer Sicht keine Bedeutung.

An einem Kommunikationsakt sind immer mindestens ein Sender und ein Empfänger beteiligt. Grundsätzlich müssen drei Voraussetzungen von Sender und Empfänger *einvernehmlich* erfüllt sein, um miteinander kommunizieren zu können:

- Kommunikationsprotokoll
- Syntax für die Wissensrepräsentation
- Ontologie für die Wissensrepräsentation

Das Kommunikationsprotokoll dient der Adressierung von Nachrichten und ist unabhängig von Art oder Inhalt der zu übermittelnden Information (siehe 4.2). Die semantische Information der Nachricht ist in einer zwischen Sender und Empfänger definierten Syntax abgefasst (siehe 4.3) und verwendet Begriffe, die in einer Ontologie festgelegt werden. Ontologien sind dabei mehr als nur “Wörterbücher”, da sie die enthaltenen Begriffe eindeutig definieren und kategorisieren (siehe Abschnitt 4.4).

Bei den drei Voraussetzungen handelt es sich bei näherer Betrachtung um eine allgemeine Definition einer Kommunikationsschnittstelle. Da die Annahmen vollkommen allgemein formuliert wurden, ist die Gesamtheit der Forderungen die Essenz[1] einer allgemeinen Kommunikationsschnittstelle.

Bemerkenswert ist, dass die *interne* Wissensrepräsentation und -verarbeitung der Kommunikationspartner überhaupt keine Rolle spielt, da Sender und Empfänger über einen Kommunikationskanal mit eindeutiger Schnittstelle verfügen. So können zum Beispiel zwei Programmsysteme miteinander kommunizieren, ohne dass dem Sender bekannt sein muss, wie die Datenstrukturen aussehen, mit denen der Empfänger sein Wissen repräsentiert.

[1] Essenz wird hier verstanden gemäß der Definition *Essenz eines Systems* in [59]

4.2 Kommunikationsprotokolle

Im folgenden soll näher auf Kommunikationsprotokolle eingegangen werden. Wegen der Aufgabenstellung liegt der Schwerpunkt auf den Protokollen, die sich als Agentenkommunikationssprachen eignen. Die beiden bedeutendsten Kommunikationsprotokolle für Agenten sind KQML und FIPA ACL.

Kommunikationsprotokolle unterscheiden sich von den in Abschnitt 4.3 beschriebenen *Wissensrepräsentationen* dadurch, dass es sich nur um sogenannte *Metasprachen* handelt. Dies bedeutet, dass sie den Austausch von Nachrichten ermöglichen. Für die tatsächliche Repräsentation des Nachrichteninhaltes kommen dann Protokolle zur Wissensrepräsentation zum Einsatz. Metasprachen dienen also nur als Nachrichtencontainer. Sie ermöglichen die korrekte Zustellung an den Empfänger, Routing von Nachrichten und die Überwachung von Konversationen. Unter einer Konversation versteht man eine Menge von Nachrichten, die aufeinander Bezug nehmen. Die einfachste Konversation besteht aus einer Nachricht und deren Antwort.

Die drei wichtigsten Angaben in einem Kommunikationsprotokoll sind, außer den Angaben über Sender und Empfänger, das *Performativ*, die *Ontologie* und der *Nachrichteninhalt*. Auf Ontologien wird im Abschnitt 4.4 genauer eingegangen und Abschnitt 4.3 widmet sich der Repräsentation des Nachrichteninhaltes, daher soll an dieser Stelle kurz der Begriff des *Performativ* erläutert werden.

In der Sprechakttheorie von AUSTIN [2] werden diejenigen sprachlichen Akte als performativ[2] bezeichnet, mit deren Hilfe Handlungen vollzogen werden. Performative Äußerungen haben keinen beschreibenden Charakter, daher erfolgt ihre Beurteilung nicht nach den Kriterien *Wahr* oder *Falsch*, sondern nach Kriterien des *Glückens* oder *Nicht-Glückens* der Handlung. Beispiele für performative Verben sind: bitten, auffordern, beauftragen, ernennen, verurteilen, kündigen, danken, gratulieren, loben und begrüßen. Das Gegenstück zu Performativen sind die *Konstative*, die eine beschreibende Aussage machen und sich daher mit *Wahr* oder *Falsch* bewerten lassen.

Diese Theorie hat man aus der Linguistik direkt auf die Problematik der formalen Kommunikation in der Informatik übertragen. Jedes Kommunikationsprotokoll

[2] engl.: *performative*

verfügt daher über die Angabe eines Performativs, das spezifiziert, wie der Inhalt zu verstehen ist. Typische Performative könnten z.B. `Question`, `Reply` oder `Greeting` sein.

4.2.1 KQML

Die Knowledge Query and Manipulation Language (KQML) ist die bedeutendste Kommunikationssprache für Agenten, die Anfang der 90er Jahre unter Federführung von TIM FININ am UMBC, der *University of Maryland Baltimore*, entwickelt wurde. FININ ist ein amerikanischer Wissenschaftler, der sich in sehr vielen Bereichen mit Agententechnologie und Künstlicher Intelligenz beschäftigt. Da KQML in den letzten Jahren die einzig verfügbare Agentenkommunikationssprache war, hat sie sich als de-facto-Standard etabliert.

Ein typische KQML-Nachricht hat den in Abbildung 4.1 dargestellten Aufbau. Der Inhalt der Nachricht ist in LISP (siehe Abschnitt 4.3.1) geschrieben.

```
( :performative TELL
  :sender   pizzaEatingAgent
  :receiver pizzaBakingAgent
  :language LISP
  :ontology pizzaOrder
  :content  (order (pizza tunafish onions))
)
```

Abbildung 4.1: Beispiel einer KQML-Nachricht mit `TELL` Performativ

Der Sender `pizzaEatingAgent` teilt dem Empfänger `pizzaBakingAgent` einen Bestellwunsch für eine Pizza mit. Der Empfänger wird nun entweder die Bestellung quittieren, den Auftrag ablehnen oder noch Rückfragen stellen.

Das Performativ `TELL` bedeutet, dass der Inhalt der Nachricht als Aussage aufzufassen ist. Der Empfänger der Nachricht wird in den meisten Fällen sein Wissen um diese Aussage erweitern. Eine Nachricht mit einem `TELL`-Performativ ist häufig die Antwort auf eine Anfrage.

Abbildung 4.2 stellt ein Beispiel mit einem `ASK-IF` Performativ dar. Der Sender

stellt hier die Frage, ob eine Pizza Tonno teurer ist als eine Pizza Napoli. Der Inhalt der Nachricht ist in KIF (siehe Abschnitt 4.3.3) formuliert. Insgesamt definiert

```
( :performative ASK-IF
  :sender   pizzaEatingAgent
  :receiver pizzaBakingAgent
  :language KIF
  :ontology pizzaOrder
  :content  (> (price (pizza tonno ))
               (price (pizza napoli)) )
)
```

Abbildung 4.2: Beispiel einer KQML-Nachricht mit `ASK-IF` Performativ

KQML etwa zwanzig verschiedene Performative. Die meisten von ihnen dienen allerdings der Kommunikationssteuerung und der Kennzeichnung von Ausnahmesituationen, wie z.B. `STANDBY`, `CANCEL` oder `READY`. Die genaue Spezifikation findet man in [19], nähere Erläuterungen auch in [20].

4.2.2 FIPA ACL

Die *FIPA Agent Communication Language* (FIPA ACL) wurde 1997 von der *Foundation for Intelligent Physical Agents* definiert. Die FIPA selbst wurde im Dezember 1995 von einer Gruppe von Einzelpersonen gegründet, die allesamt namhaften Firmen und Organisationen angehören. Die FIPA entwickelt sich seitdem mehr und mehr zu einem inoffiziellen Standardisierungsgremium für alle Aspekte intelligenter Agentensysteme. Nähere Information zur FIPA, sowie die von ihr herausgegebenen Spezifikationen findet man in [22].

FIPA ACL ist, wie KQML, ebenfalls ein Kommunikationsprotokoll, dient also dem Adressieren, Transportieren und Routen von Nachrichten. Die Definition von FIPA ACL orientiert sich sehr stark am älteren de-facto-Standard KQML. Abbildung 4.3 zeigt ein Beispiel einer FIPA ACL-Nachricht. *FIPA ACL* ist bisher nur der Versuch einer Standardisierung der FIPA für die Agentenkommunikation. Derzeit gibt es nach Wissen des Autors keine Implementation von FIPA ACL. Ob sich die

```
(inform
  :sender   pizzaEatingAgent
  :receiver pizzaBakingAgent
  :content
    (order (pizza tunafish onions))
  :in-reply-to offer007
  :reply-with  order007
  :language LISP
  :ontology pizzaOrder
)
```

Abbildung 4.3: Beispiel einer FIPA ACL-Nachricht

Definition durchsetzen wird, ist ebenfalls noch offen. Da die FIPA kein offizielles Standardisierungsgremium ist, sondern nur Vorschläge erarbeitet, bleibt abzuwarten, ob sich FIPA ACL gegen das etablierte KQML wird durchsetzen können. Die Spezifikation von ACL findet man in [25] bzw. in [26].

4.3 Sprachen zur Wissensrepräsentation

Sprachen zur Wissensrepräsentation werden benötigt, um Wissen über Objekte und ihre Beziehungen untereinander auszutauschen. Da die Repräsentation von Wissen auch eine Grundvoraussetzung für das Lösen von Problemen im Bereich der Künstlichen Intelligenz ist, wurden die Sprachen zur Wissenrepräsentation zumeist dem Kontext der KI entlehnt (LISP und PROLOG) oder eigens für die Verwendung mit ihrer jeweiligen Kommunikationssprache entwickelt (KIF bzw. FIPA-KIF). Die genannten Sprachen werden nachfolgend beschrieben. Weiterhin wird XML vorgestellt, da es ebenfalls zur inhaltlichen Strukturierung von Informationen verwendet wird.

4.3.1 LISP

Die Programmiersprache LISP[3] wurde in den Jahren 1956 bis 1958 am Massachusetts Institute of Technology (MIT) unter der Leitung von JOHN MCCARTHY entwickelt. Die Implementierung von LISP begann 1958 im Rahmen des *M.I.T. Artificial Intelligence Project*, geleitet von MARVIN MINSKY.

LISP ist eine applikative Programmiersprache, deren wichtigste Datenstrukturen lineare Listen und binäre Bäume sind. Sie eignet sich besonders für Anwendungen im Bereich der Künstlichen Intelligenz, da sie speziell für den Umgang mit symbolischen Informationen und mit Strukturen entworfen wurde.

Typischerweise werden die Programme von Interpretern ausgeführt. Auf die sonst in Hochsprachen üblichen Schleifen (oder gar Sprünge) wird vollständig verzichtet; Iterationen werden mit Hilfe von Rekursionen gelöst.[4] Eine Besonderheit des originalen LISP Dialektes pureLISP ist, dass er völlig frei von Seiteneffekten ist. Eine kurze, präzise Einführung in LISP gibt [78]. Weitergehende Informationen findet man z.B. in [70].

Wird mittels LISP-Statements oder sogar ganzer LISP-Programme Wissen ausgetauscht, kann der Inhalt der Nachricht vom Empfänger direkt an einen Interpreter weitergegeben werden. Außerdem sind LISP-Statements auch für den Menschen gut lesbar, so dass sie gerne als Wissensrepräsentationssprache eingesetzt wird. Die besondere Stärke von LISP liegt, im Gegensatz zu PROLOG, in der Beschreibung von *Algorithmen*.

4.3.2 Prolog

PROLOG[5] ist wie LISP eine Hochsprache, deren Anwendung im Bereich der künstlichen Intelligenz angesiedelt ist. Sie wurde 1973 von einer Forschungsgruppe um Alain Colmerauer an der Universität von Marseilles entwickelt. Bis 1981 war PROLOG ein wissenschaftlich orientiertes Projekt, und daher nur wenigen Wissenschaftlern an Universitäten und Forschungsinstituten bekannt. Dies änderte

[3] Abk. für *list p*rocessing language

[4] Heute gibt es allerdings auch LISP-Dialekte, die diese Beschränkung aufheben.

[5] Abk. für *Programming in Logic*

sich jedoch, nachdem das japanische ICOT-Institut PROLOG als die grundlegende Sprache der fünften Computergeneration erklärte. Plötzlich interessierten sich Softwarehersteller für die Sprache und es entstanden viele Dialekte. (vgl. [41])

PROLOG ist eine *prädikative* Programmiersprache. In imperativen Sprachen wird ein gegebenes Problem dadurch gelöst, dass der Programmierer einen Algorithmus formuliert, durch dessen Ausführung man die Lösung erhält. Bei prädikativen Sprachen hingegen formuliert man das Wissen über das Problem, und das PROLOG-System sucht mit Hilfe dieses Wissens selbständig nach der Lösung des Problems. Die Suche erfolgt nach Regeln des *Prädikatenkalküls erster Ordnung*.

Die Stärken von PROLOG liegen in der Beschreibung von Weltwissen und Beziehungen. Diese Sprache bietet sich an, wenn eine Agentenkommunikationssprache vom Agenten dazu benutzt wird, Teile der eigenen internen Wissensrepräsentation anderen Agenten mitzuteilen. Diese können die PROLOG Darstellung direkt verwenden, um ihr eigenes Wissen zu erweitern, oder um ein fremdes Problem mit Hilfe des eigenen Wissens zu lösen.

4.3.3 KIF

Das *Knowledge Interchange Format* (KIF) ist eine an der University of Balitmore Maryland (UMBC) entwickelte Sprache zum Austausch von Information zwischen verschiedenartigen Computersystemen. Im Gegensatz zu den zuvor vorgestellten Sprachen LISP und PROLOG ist KIF *nicht* zur systeminternen Repräsentation von Wissen konzipiert, obwohl die Sprache dafür sehr wohl eingesetzt werden könnte. Üblicherweise wird es so sein, dass ein System eine KIF-Nachricht in eine interne Repräsentation bringen und alle Berechnungen auf dieser internen Wissensbasis durchführen wird. Für eine Kommunikation mit einem externen System wird die Information dann wieder in KIF konvertiert. Es existieren bereits Übersetzungsprogramme, die einen KIF-Ausdruck in einen äquivalenten Ausdruck einer anderen Sprache hin- und rücktransformieren können. Die Entwickler verstehen KIF als eine *Interlingua*, d.h. als eine allgemeine, nicht auf die Agentenkommunikation beschränkte, Sprache zum Austausch von Information. In [21] wird sogar angedeutet, KIF könne SQL ersetzen, um als einheitliche Schnittstelle zum Informationsaustausch zu dienen. KIF wurde als American National Standard vorgeschlagen (vgl.

[21], [31]). In der Spezifikation werden fünf Eigenschaften für KIF definiert, wovon die ersten drei als essentiell bezeichnet werden:

1. Die Sprache hat eine deklarative Semantik. Es ist möglich, die Bedeutung von Ausdrücken zu verstehen, ohne einen Interperter aufzurufen, der die Ausdrücke manipuliert.
2. Die Sprache ist logisch umfassend. Sie sieht die Formulierung beliebiger logischer Sätze vor.
3. Die Sprache sieht die Repräsentation von Wissen über Wissen vor. Es kann fremdes Wissen referenziert werden.
4. Die Sprache kann zur Implementation verwendet werden, muss also Algorithmen und Datenstrukturen darstellen können.
5. Die Sprache ist für den Menschen lesbar.

Syntaktisch ist KIF, abgesehen von einigen Erweiterungen zur Steigerung der Ausdrucksmöglichkeiten, eine Präfixversion des Prädikantenkalküls erster Ordnung. Daher ähneln KIF-Terme auch sehr stark LISP-Ausdrücken. In KIF lassen sich einfache Daten, Beschränkungen, Negationen, Disjunktionen, Regeln, quantifizierende Ausdrücke und Metawissen[6] beschreiben. Die vollständige Definition von KIF findet man in [31].

Bemerkenswert ist, dass auch die FIPA in [26] ein *FIPA-KIF* definiert. Der Text der Spezifikation weist inhaltlich, wie auch im Wortlaut einzelner Textpassagen, frappierende Ähnlichkeit mit der Spezifikation der UMBC auf, so dass man davon ausgehen kann, dass sie in Zusammenarbeit mit dem UMBC entstanden ist. Wenn KIF ein American National Standard wird und zusätzlich von der FIPA unterstützt wird, hat die Spezifikation gute Chancen sich in Zukunft gegen das in diesem Bereich etablierte LISP durchzusetzen.

4.3.4 XML

In jüngster Zeit gewinnt die *Extensible Markup Language* (XML) mehr und mehr an Bedeutung für den universellen Austausch von Informationen. Vielerorts wird

[6] Wissen über das Wissen anderer Individuen

XML schon als “Lingua Franca” gepriesen, darum kommt sie auch als Content Language für die Agentenkommunikation in Betracht. Zudem ist abzusehen, dass XML in den nächsten Jahren HTML als Beschreibungssprache für Hypertextdokumente ablösen wird. Allein dadurch wird sie eine globale Verbreitung erlangen.

XML ist im Wesentlichen eine plattformunabhängige Möglichkeit, Informationen zu strukturieren [61]. Ein XML Dokument kann als ein Baum von Elementen betrachtet werden. Ein Element darf eine Reihe von Attributen, in Form von Schlüssel-Wert-Paaren, besitzen und kann andere Elemente oder Text enthalten.

Die Stärken von XML liegen in einer darstellungsunabhängigen, rein am Inhalt orientierten Repräsentation von Informationen und in der leichten Erweiterbarkeit. XML ist keine fertig definierte Sprache, sondern ist darauf ausgelegt, dass sie jeder nach seinen Bedürfnissen erweitern kann. Dies geschieht durch die Definition und Beschreibung eines eigenen XML-Subsets. Eine solche Subset-Definition beschreibt:

- die zulässigen Tags des Subsets,
- eine erlaubte Strukturierung dieser Tags und
- Parameter-Tags mit ihren Datentypen.

Die Definition eines neuen Subsets werden in einem *Document Type Dictionary* (DTD) abgelegt. Ruft nun ein Client ein XML-Dokument ab, dessen Subset er nicht kennt, gibt die DTD ihm Informationen darüber, wie ein zulässiges Dokument dieses Subsets aussehen muss. Leider macht die DTD nur syntaktische und keinerlei semantische Aussagen; auch Abhängigkeiten zwischen Parameterwerten sind nicht formulierbar:

> “DTDs are fairly weak. They support the definition of simple constraints on structure and content, but provide no facility for expressing data types or complex structural relationships.” [61]

In Abbildung 4.4 ist eine XML Datei dargestellt, die ein Kochrezept darstellt. Die DTD ist in diesem Beispiel in das XML-Dokument eingebettet.[7] Ein Rezept besteht demnach aus einem Namen, einer Beschreibung des Ergebnisses, Zutaten

[7] Es ist natürlich auch möglich — und sogar üblich — die DTD in einer eigenen Datei abzulegen und diese in der XML-Datei nur zu referenzieren.

```
<!-- DTD for recipes -->
<!ELEMENT Recipe (Name, Description?, Ingredients?,
                  Instructions?)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Description (#PCDATA)>
<!ELEMENT Ingredients (Ingredient)*>
<!ELEMENT Ingredient (Quantity, Item)>
<!ELEMENT Quantity (#PCDATA)>
<!ATTLIST Quantity unit CDATA #REQUIRED>
<!ELEMENT Item (#PCDATA)>
<!ELEMENT Instructions (Step)+>
<!ELEMENT Step (#PCDATA)>

<!-- Here starts the XML Content -->
<?xml version="1.0">
<Recipe>
  <Name>
     Cheese Sandwich
  </Name>
  <Description> My grandma’s favorites </Description>
  <Ingredients>
    <Ingredient>
      <Quantity unit="slice">1</Quantity>
      <Item>Dutch Cheese</Item>
    </Ingredient>
    <Ingredient>
      <Quantity unit="slice">2</Quantity>
      <Item>Toast</Item>
    </Ingredient>
  </Ingredients>
  <Instructions>
    <Step>Roast both slices of toast</Step>
    <!-- And so on ... -->
  </Instructions>
</Recipe>
```

Abbildung 4.4: Beispiel einer XML-Datei mit eingebettetem DTD

und Zubereitungsanweisungen. Diese Elemente stellen die erlaubten Kindknoten des Vaterknotens `Recipe` dar. Das '`?`' deutet an, dass die letzten drei Elemente optional sind. Die Zutaten wiederum bestehen aus einer Menge von einzelnen Zutaten ('`*`'-Operator). Jede Zutat wird mit Menge und Typ angegeben.

Der praktische Vorteil von XML liegt in der leichten Umformbarkeit. Ein XML-Dokument wird von einem XML-Parser[8] gelesen und in ein *Document Object Model* (DOM) umgewandelt. Ein DOM ist eine interne Baumstruktur, die das XML-Dokument repräsentiert. Nun kann dieses DOM von einem Programm traversiert, ausgewertet oder umgewandelt werden. Ein geändertes DOM lässt sich dann wieder in ein XML-Dokument überführen. Auf diese Weise kann z.B. ohne großen Aufwand ein Übersetzer zwischen zwei XML-Subsets realisiert werden. Abbildung 4.5 veranschaulicht diesen Vorgang.

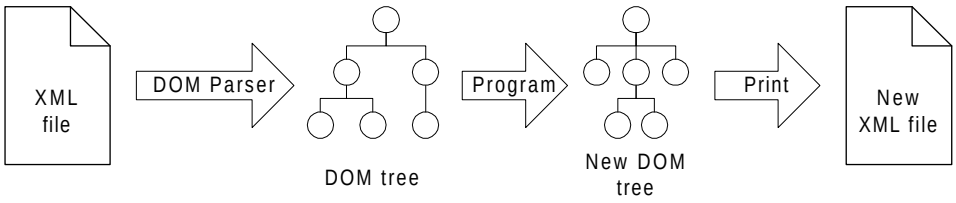


Abbildung 4.5: Transformation eines XML Dokumentes. Nach [6].

Vergleicht man XML mit HTML, so bietet XML vor allem den Vorteil der darstellungsunabhängigen Informationsrepräsentation. D.h., man kann aus einem XML-Dokument beispielsweise sowohl eine Webseite als auch einen Brief generieren. Der wesentliche Nachteil ist allerdings, dass die Semantik der Tags extern definiert werden muss. Bei den vorgenannten Sprachen KIF, LISP und PROLOG ist dies nicht der Fall. Diese sind in der Lage, neue Sematiken und Objektrelationen auch während einer Kommunikation vereinbaren. Trotzdem kommt XML aufgrund seiner Publizität eine gewisse Bedeutung als Inhaltssprache bei der Agentenkommunikation zu. In abgeschlossenen Multiagentensystemen, die u.U. sogar mit Businesssoftware kommunizieren, ist XML eine brauchbare Alternative zu den

[8] Mittlerweile gibt es sehr leistungsfähige (freie) Implementationen für alle gängigen Programmiersprachen. Ein vielbeachtetes Projekt ist *Xerces* von der Apache Group, aber auch Sun Microsystems und andere namhafte Hersteller stellen XML-Parser zur Verfügung. Eine Reihe von Internet-Links findet man z.B. in [6].

vorgenannten Sprachen zur Wissensrepräsentation.

4.4 Ontologien

Der Begriff der Ontologie stammt ursprünglich aus der Philosophie und bezeichnet dort die philosophische Grunddisziplin der Lehre vom Seienden.

> "Sie untersucht (als *formale* Ontologie) die obersten Strukturen und Gesetzlichkeiten [...] und (als *materiale* Ontologie) die inhaltliche Gliederung des Seienden." [15]

Sie untersucht also, was das Gemeinsame alles Seienden ist und die Relationen der Dinge zueinander, unabhängig davon, ob es sich um Steine, Wasser, Pflanzen, Tiere, Menschen oder Planeten handelt.

Der Begriff der Ontologie wurde in der Informatik, und speziell in der Forschung über wissensbasierende Systeme, aufgegriffen und auf den eigenen Kontext übertragen. Hier bezeichnet eine *Ontologie*[9] eine Beschreibung von Begriffen und deren Relationen zueinander. GRUBER gibt eine sehr prägnante Definition einer Ontologie [37]:

> "An ontology is a specification of a conceptualisation."

Ontologien werden benötigt, damit wissensverarbeitende Systeme, zu denen definitionsgemäß auch intelligente Agenten gehören, miteinander kommunizieren können. Die Ontologie legt dabei die Menge der verwendbaren Begriffe, deren Bedeutung und die Relationen untereinander fest. Eine Ontologie gilt dabei stets für einen semantischen Problembereich[10]. Sie stützt sich bei der Beschreibung auf axiomatische Grundbegriffe, wie z.B. *Zahl*, *Maßeinheit* oder *Zeit*.

Forschungsgegenstand in vielen Projekten ist das *Ontology Sharing*, also das Problem, dem potentiell unbekannten Gesprächspartner die eigene Ontologie mitzuteilen, und die Probleme, die aus der Verwendung multipler Ontologien erwachsen (siehe z.B. [79]). Die FIPA hat dazu eine Spezifikation [24] veröffentlicht, die die Anforderungen an ein solches System definiert.

[9] engl.: *ontology*
[10] engl.: *Domain of Discourse*

Kapitel 5

Das SeMoA Projekt der Fraunhofer Gesellschaft

Das in dieser Arbeit entwickelte System ist ein Subsystem des SeMoA Projektes der Fraunhofer Gesellschaft. Um die vorliegende Arbeit besser in das Gesamtkonzept einordnen zu können, soll das umgebende Projekt SeMoA an dieser Stelle vorgestellt werden.

5.1 Intention von SeMoA

Der Name SeMoA steht für “Secure Mobile Agents”. Es handelt es sich bei dem Projekt um ein, am Fraunhofer Institut für Graphische Datenverarbeitung, Abteilung Sicherheitssysteme für Graphik- und Kommunikationssysteme, Darmstadt, durchgeführtes Forschungsvorhaben, mit dem Ziel der Konzeption und Implementierung einer sicheren Softwareplattform für Mobile Agenten. Diese Plattform soll weitgehende Sicherheit bieten vor:

- Angriffen zwischen den Agenten
- Angriffen von Agenten auf den Agentenserver
- Angriffen des Agentenservers auf die Agenten
- Abfangen und Ausspähen von Mobilen Agenten während des Transports

Die Entwickler von SeMoA betrachten Sicherheit als eine “fundamentale Voraussetzung für die Verbreitung von Agentensystemen” [63]. Heutzutage existieren leider kaum Plattformen, die sich der umfangreichen Sicherheitsproblematik annehmen. Häufig sind die Sicherheitsdienste beschränkt auf die Verschlüsselung der Agenten während des Transports und die Authentifizierung der am Transport beteiligten Server. Hierbei kommt meist das verbreitete SSL[1]-Verfahren zur Anwendung. ROTH und JALALI führen in [63] weiter aus:

> “Protection of mobile agents against a malicious host is provided by none of the mobile agent systems at present. The problem of malicious hosts is recognized as a most difficult one, which well explains the lack of mechanisms provided in actual mobile agent implementations. Due to this, even the most basic protection would be welcomed.”

Das angesprochene Problem des *Malicious*[2] *Host* beschreibt die Tatsache, dass ein Agent, der auf einen anderen Server migriert, absolut dem Wohl und Wehe des Servers ausgeliefert ist. Die Möglichkeiten des Servers reichen dabei vom Ausspähen der vom Agenten gesammelten Daten, über deren Manipulation bis hin zur Manipulation und sogar Liquidation des Agenten selbst. Darüberhinaus kann der Server dem Agenten während der Laufzeit Betriebsmittel und Rechenzeit entziehen, um seine Ausführung zu behindern oder zu kontrollieren (Denial of Service Attack). Es gibt also im Bereich der Sicherheit von Mobilen Agentenplattformen ein weites Betätigungsfeld. Um eine angemessene Sicherheit für Agenten und Host zu erreichen, verfügt SeMoA über umfangreiche Vorkehrungen, die die dargestellten Probleme zwar noch nicht vollständig lösen, aber bereits hohes Maß an Sicherheit gewährleisten.

5.2 Systemarchitektur von SeMoA

Dieser Abschnitt stellt die SeMoA zugrunde liegende Software-Architektur vor. Dies soll helfen, das im Rahmen dieser Arbeit erstellte Projekt, besser im Gesamtkonzept einordnen zu können. Abbildung 5.1 gibt einen Überblick über die

[1] *Secure Socket Layer*

[2] engl.: böswillig, arglistig

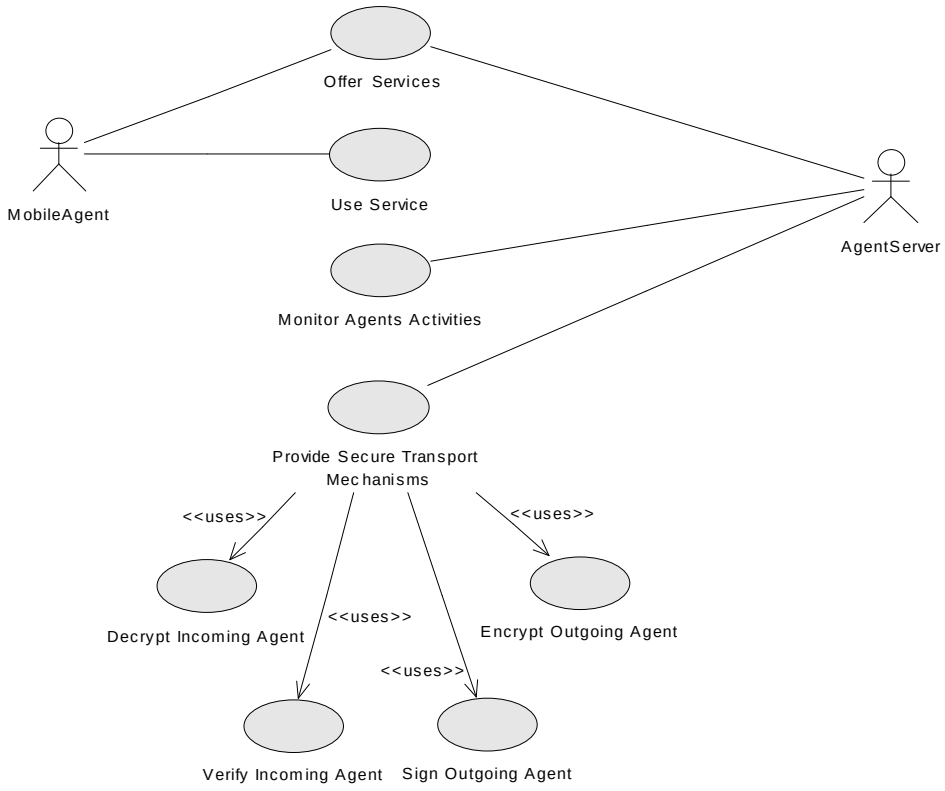


Abbildung 5.1: Use Case Diagramm der SeMoA Architektur

Architektur in Form eines Use Case Diagramms. Es zeigt die Situation nach der Migration und Wiederbelebung des Agenten auf dem Zielhost. Die Aktoren sind der Mobile Agent und der Agentenserver. Ein Agent besitzt auf dem SeMoA Server im Grunde nur die Fähigkeit, auf dem Server angebotene Dienste in Anspruch zu nehmen (*Use Service*), oder eigene, mitgebrachte Dienste anderen Agenten zur Verfügung zu stellen (*Offer Service*). Der Anwendungsfall, den Server wieder zu verlassen, ist hier nicht dargestellt, da dies (auf dem neuen Host) wieder zu dem gleichen Diagramm führen würde. Die vom Server angebotenen Dienste sind für Agenten der Grund, zu einem Server zu migrieren. In der Regel handelt es sich bei den Serverdiensten um Informationsdienste, über die der Agent beispielsweise Einblick in eine lokale Datenbank erhält. Aber auch Dienste zur Kommunikation oder zum Aufgeben von Bestellungen sind hier denkbar.

Darüber hinaus bietet das System noch eine ganze Reihe von Sicherheitsmechanismen an. Dies reicht von der Überwachung der Agentenaktivität (*Monitor Agent*) bis zu Filtermodulen für Transportsicherung und Verifikation[3]. Agenten werden, um sie vor Ausspähung durch Dritte zu schützen, vor dem Transport mit einem Public-Key-Verfahren signiert und anschließend verschlüsselt. Die Mechanismen der Ver- und Entschlüsselung, sowie das Signieren und das Verifizieren der Signatur sind als eigene Use Cases dargestellt, die ihrerseits von dem Use Case *Provide Secure Transport Mechanism* genutzt werden.

Abbildung 5.2 zeigt die Kernarchitektur des Fraunhofer Projektes in einem Klassendiagramm. Die beiden Hauptkomponenten sind, analog zur Darstellung des System als Use Case Diagramm, der Aktor MobileAgent$^{(Class)}$ und der Agenten Server (Server$^{(Class)}$). Eine sehr wichtige Rolle spielen Services (Service$^{(Class)}$) in SeMoA . Fast die gesamte nach außen sichtbare Funktionalität des Servers wird auf Services abgebildet. Exemplarisch sind in der Abbildung dargestellt:

Transportdienste für die Migration des Agenten (InGate$^{(Class)}$ und OutGate$^{(Class)}$),

Verschlüsselungsfilter um einen Agenten für den Transport zu verschlüsseln (DecryptFilter$^{(Class)}$), bzw. um einen empfangenen Agenten zu entschlüsseln (EncryptFilter$^{(Class)}$), und

[3]Der Abschnitt 5.3 geht näher auf die Sicherheitsmechanismen von SeMoA ein.

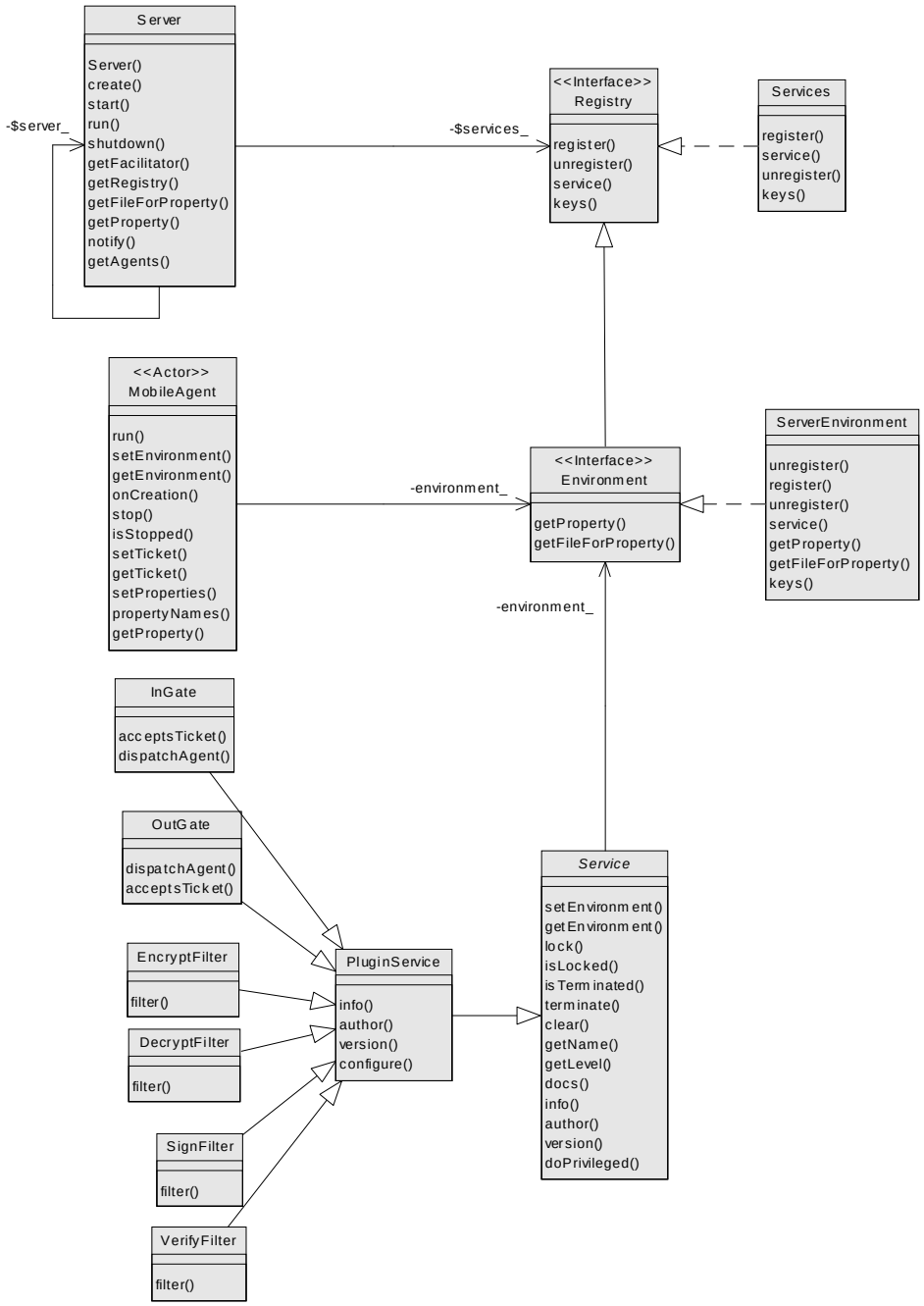


Abbildung 5.2: Klassendiagramm der Kernkomponenten von SeMoA

Authentifizierungsfilter die einen Agenten vor dem Transport signieren (SignFilter$^{(Class)}$), bzw. die Signatur eines empfangenen Agenten prüfen (VerifyFilter$^{(Class)}$).

Wie bereits erwähnt, handelt es sich nur um eine kleine Auswahl der zur Verfügung stehenden Services. Auch viele serverinterne Aufgaben werden durch Services realisiert[4]. Alle serverseitigen Dienste werden von PluginService$^{(Class)}$ abgeleitet. Dies ist von besonderer Bedeutung, da auch das in dieser Arbeit entwickelte Dialogsystem für Mobile Agenten diesen Mechanismus verwenden muss, um sich in SeMoA zu integrieren. Unter einem PluginService$^{(Class)}$ wird in SeMoA ein Dienst verstanden, der automatisch beim Start des Servers geladen wird. Diese Dienste können beim Start eine Konfigurationsdatei vom lokalen Dateisystem lesen, was sie aus Sicht der Sicherheit besonders privilegiert. Der Server muss sich bei seinem eigenen Startvorgang auf ihre Integrität verlassen. In der Regel wird als PluginService nur sog. *Trusted Code*[5] verwendet, der dann vom Betreiber des Agentenhostes selbst stammt. PluginService$^{(Class)}$ ist eine Unterklasse von Service$^{(Class)}$. Ein Agent kann nur einen "normalen" Service$^{(Class)}$ anmelden. Diese Trennung ist, wie angedeutet, durch das Sicherheitkonzept von SeMoA begründet [64].

Eine zentrale Rolle spielt die Schnittstelle Environment$^{(Interface)}$. Ein Agent kann über sein Environment mit dem Server kommunizieren. Das Environment gibt dem Agenten eine spezielle, auf ihn angepasste Sicht auf den Server und schirmt ihn von der Serverimplementation ab. Das Environment ermöglicht dem Agenten bestimmte Umgebungsvariablen des Servers zu lesen, Dienste an- und abzumelden und auf mitgebrachte Dateien zuzugreifen. Die Begründung für die Verwendung eines Environments findet sich wieder in Sicherheitsaspekten. Auch PluginService$^{(Class)}$ und Service$^{(Class)}$ verwenden dieses Environment für die Kommunikation mit dem Server. Environment$^{(Interface)}$ ist abgeleitet von der Klasse Registry$^{(Interface)}$, einer Schnittstelle, die vom Server benutzt wird, um die angemeldeten Dienste zu verwalten.

Ein interessantes Detail ist die statische, reflexive Assoziation (Selbstreferenz) des Servers. Es handelt sich um eine Klassenvariable, die gesetzt wird, sobald der erste

[4] Diese Services sind dann aus Sicherheitsgründen für Mobile Agenten nicht verfügbar.

[5] engl.: *vertrauenswürdiger Code*; Programmcode, der in sicherheitskritischen Bereichen eingesetzt wird, und bei dem man davon ausgeht, dass er kein böswilliges Verhalten zeigt.

Server instantiiert wird. Dadurch kann beim Aufruf bestimmter Klassenmethoden von der *Klasse* geprüft werden, ob bereits ein Server instantiiert wurde und diesen ggf. direkt ansprechen.

5.3 Sicherheitspolitik

Es wurde bereits an mehreren Stellen deutlich, dass die Implementation von SeMoA einem stringenten Sicherheitskonzept folgt. Einige Aspekte dieses Konzeptes sollen hier kurz besprochen werden. Abbildung 5.3 stellt die wichtigsten Ansätze grafisch dar. Die mit *M.A.* bezeichneten Einheiten stellen die Mobilen Agenten dar.

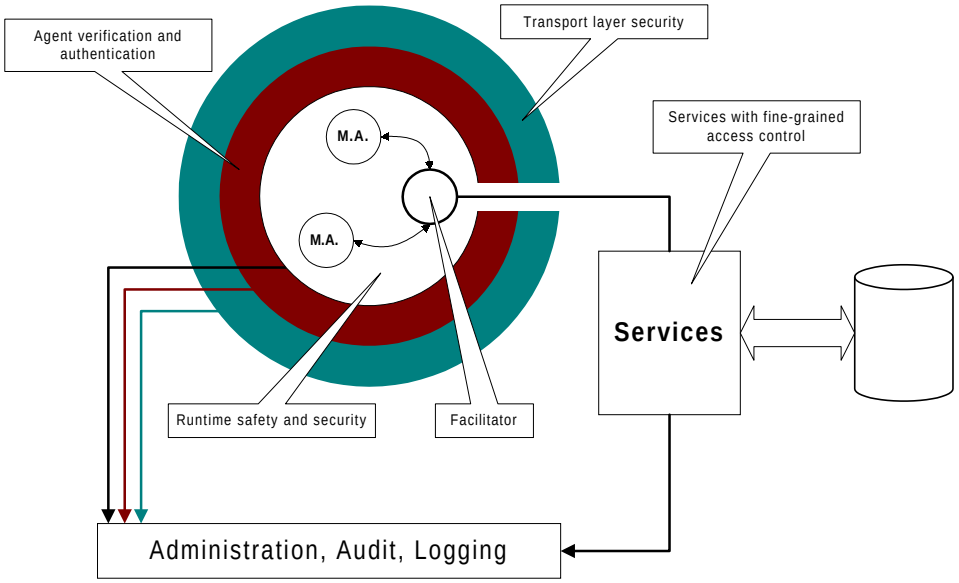


Abbildung 5.3: Sicherheitsarchitektur von SeMoA

Um auf den Server zu gelangen, müssen sie zunächst zwei Sicherheitsfilter passieren, die hier mit *Transport Layer Security* und *Agent Verification and Authentication* bezeichnet sind. Die Sicherheitsfilter der Transportschicht sorgen dafür, dass Agenten für den Transport verschlüsselt werden; hierbei kommt ein Hybridverfahren[6] zum Einsatz. Dieses Verfahren wird so verwendet, dass es die Verschlüsselung

[6]Bei *Hybridverfahren* wird zur Verschlüsselung einer Nachricht ein zufälliger symmetrischer Schlüssel verwendet, der nur für eine Sitzung gilt. Dieser Sitzungsschlüssel wird dann mit dem

nur eines Teils der vom Agenten mitgeführten Dateien ermöglicht [62]. Das bedeutet, auf dem Zielhost angekommen, kann der Server auch nur einen bestimmten Teil wieder entschlüsseln. Der Agent bekommt dadurch die Möglichkeit, gezielt Informationen vor dem Zielhost zu verbergen. Näheres zur Transportsicherung für Mobile Agenten und zu dem damit verbundenen Key Management findet man in [62].

Nach der Übertragung wird er auf dem Zielhost wieder dechiffriert. Danach wird die Signatur des entschlüsselten Agenten geprüft. Mit Hilfe der Signatur lässt sich feststellen, wer der Absender des Agenten ist und ob dieser die Berechtigung hat, einen Agenten auf diesem Host auszuführen. Hat der Agent diese beiden Sicherheitsfilter passiert, wird er gestartet. Dabei kann er über sein Environment auf Services des Servers und damit auf lokale Ressourcen, wie z.B. eine Datenbank, zugreifen. Um eine möglichst flexible Vergabe von Berechtigungen für die Nutzung eines Services zu erreichen, sind in SeMoA verschiedene Service-Level definiert. Jeder Service wird durch seinen Namen und den Service-Level eindeutig identifiziert. Den Agenten werden *Permissions* zugeordnet, die den Zugriff auf die Services eines Level gestatten oder verbieten. Alle genannten Sicherheitsmechanismen sind in weiten Bereichen administrierbar, zudem werden alle Aktivitäten auf dem Server protokolliert.

Aus Gründen der Sicherheit zur Laufzeit (*Runtime Safety and Security*) müssen bei der Implementation die Stärken und Schwächen der Implementationssprache sehr stark berücksichtigt werden. Damit kann man in erster Linie Denial-of-Service Attacken begegnen. Beispielsweise darf ein Agent keine Synchronisationssperre auf die Klasse Server$^{(\text{Class})}$ setzen, da dies den Host komplett blockieren würde. Weiterhin muss jeder Agent in einem eigenen Thread ausgeführt werden, damit er keine Möglichkeit bekommt, fremde Prozesse zu stoppen oder gestoppte Prozesse vorzeitig zu wecken. SeMoA ermöglicht durch geeignete Implementierungsstrategien für Agenten und Server eine weitgehende Sicherheit zur Laufzeit. Es sei in diesem Zusammenhang auf die Dokumentation zum SeMoA Projekts [71] und auf den Artikel von ROTH und JALALI verwiesen [63]. Genauere Beschreibungen des

Public Key des Empfängers verschlüsselt und ausgetauscht. Beide Kommunikationspartner chiffrieren ihre Kommunikation nun mit demselben Sitzungsschlüssel. Dies hat gegenüber symmetrischen Kryptoverfahren und Public-Key-Verfahren Vorteile hinsichtlich Sicherheit und Geschwindigkeit der Kommunikation. Siehe [69]

SeMoA Sicherheitskonzepts finden sich außerdem in [62], [64] und [65].

Teil II

Systementwicklung

Kapitel 6

Problemdefinition

Bereits in Abschnitt 1.2 wurden die Ziele des zu entwickelnden *Dialogsystems zur multimedialen Interaktion zwischen Mensch und Agenten* dargestellt. Diese Zielvorgabe soll in Abschnitt 6.1 noch einmal herangezogen werden, um als Grundlage für die anschliessende Systementwicklung zu dienen. Abschnitt 6.2 ergänzt dies um die Beschreibung zweier Forschungsvorhaben in denen sich der Einsatz dieses Dialogsystems anbietet.

6.1 Anforderungen

Ziel des zu entwickelnden Systems ist es, eine Lösung zu finden, die es dem Menschen erlaubt, einen (Mobilen) Agenten mit Hilfe der natürlichen Sprache aufgabenspezifisch zu konfigurieren. Der Spracheingabe soll dafür der semantische Inhalt entnommen und in Agentenkommunikationssprache überführt werden. Dieses Protokoll wird vom Agenten verstanden. Diese Transskription von der natürlichen Sprache in eine Agentenkommunikationssprache hat den Vorteil, dass der Agent selbst keine Fähigkeiten zur semantischen Analyse natürlicher Sprache braucht. Seine, i. d. R. ohnehin vorhandene Fähigkeit, den Inhalt einer Agentenkommunikations-Nachricht zu verstehen, reicht vollkommen aus. Abbildung 6.1 zeigt den beschriebenen Sachverhalt aus Sicht des Agenten in einer “Black-Box-Sichtweise”. Dargestellt ist die essentielle Kommunikation zwischen dem Agenten und dem Dialogsystem. Der Agent fordert das System auf, einen Dia-

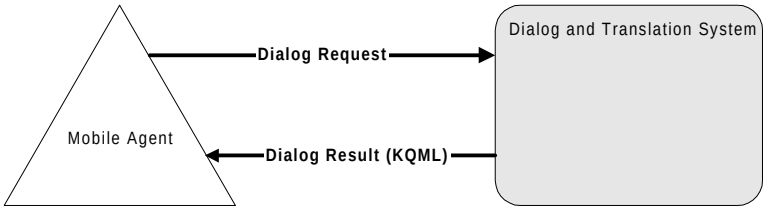


Abbildung 6.1: Sicht des Agenten auf das Dialogsystem

log mit dem Benutzer zu führen und erhält die Ergebnisse des Dialoges in Form einer KQML[1]-Nachricht. Die zu entwickelnde Lösung sollte folgende Eigenschaften haben:

- Die Schnittstellen zum Agenten, wie auch zum Benutzer hin, sollen klar definiert sein.
- Die Lösung soll möglichst allgemein sein, so dass sie sich leicht an neue Dialoge und sprachliche Kontexte anpassen lässt.
- Der Benutzer soll nicht über Gebühr in Wortwahl und Satzbau eingeschränkt werden.
- Die Lösung soll leicht um grammatische und lexikalische Möglichkeiten erweiterbar sein.
- Entwickler eines Agenten bzw. eines Softwaresystems sollen das Produkt möglichst unkompliziert einsetzen können.
- Leicht erweiterbar für weitergehende Anforderungen, z.B. Anbindung an ein Avatarsystem.

[1]KQML steht in diesem Szenario exemplarisch für eine beliebige Agentenkommunikationssprache.

6.2 Einsatzbereich des Dialogsystems

Nach dem derzeitigen Kenntnisstand wird die SeMoA Plattform und das in der vorliegenden Arbeit entworfene Dialogsystem in zwei vom BMBF[2] geförderten, Forschungsprojekten eingesetzt werden. Bei diesen Forschungsvorhaben handelt es sich um zwei der sechs Siegerprojekte des vom BMBF ausgeschriebenen Wettbewerbs *Mensch-Maschine-Interaktion in der Wissensgesellschaft* [3, 4]. Der Wettbewerb sucht nach Lösungen, die dem Menschen einen intuitiven Zugang zu Medien der Information, Kommunikation und Unterhaltung ermöglichen sollen. Es wird also nach der "intuitiven Benutzerschnittstelle von morgen" gesucht. Für alle sechs Projekte werden vom BMBF (5 Projekte) und vom BMWi (1 Projekt) Fördermittel mit einem Volumen von ca. 165 Mio. DM zur Verfügung gestellt.

Bei den beiden Siegerprojekten, in denen SeMoA zum Einsatz kommen soll, ist das Fraunhofer Institut für Graphische Datenverarbeitung federführender Forschungspartner. Konkret handelt es sich um die Projekte *MAP* und *EMBASSI*.

MAP - Multimedia Arbeitsplatz der Zukunft ist ein Projekt, an dem insgesamt 15 Partner beteiligt sind. Federführender Industriepartner ist die Alcatel SEL AG; Forschungspartner sind das Fraunhofer IGD und das Zentrum für Graphische Datenverarbeitung e.V. Im Mittelpunkt des Vorhabens steht die Entwicklung neuartiger Assistenz- und Delegationssysteme. Software-Agenten sollen dem Menschen zeitraubende, routinemäßige Aufgaben abnehmen. Die Ergebnisse werden von den Agenten dann multimedial präsentiert. "Ziel ist es, geeignete Technologien, Basiskomponenten und Methoden zur multimedialen Interaktion zwischen Mensch und Maschine zu entwickeln. Darauf aufbauend sollen neuartige, intelligente Assistenz- und Delegationssysteme entstehen, damit der Nutzer effizienter arbeiten kann." [13]. Ein besonderer Schwerpunkt liegt in der Mobilität des Benutzers. Aufgaben und Informationen sollen vom Benutzer überall delegiert und abgerufen werden können.

EMBASSI - Elektronische Multimedia Bedien- und Serviceassistenz. "Ziel des Projektes ist es, der wachsenden Diversifikation und Komplexität elektronischer Geräte und Medien im privaten Umfeld die Gestaltung einer menschgerechten und intuitiven, weitgehend einheitlichen Bedienung und Wartung mit Hilfe intelligen-

[2] Bundesministerium für Bildung und Forschung

ter Assistenzsysteme entgegenzusetzen, die sich durch Adaptivität, einen ubiquitären[3] Zugriff und integrierte Nutzung vernetzter Geräte auszeichnet." [3] An diesem Projekt beteiligt sich ein Konsortium bestehend aus 20 Partnern aus Industrie und Forschung. Hier ist die Grundig Fernseh-Video Produkte und Systeme GmbH federführender Industripartner, Forschungspartner sind das Fraunhofer IGD, die Humboldt-Universität zu Berlin, die Kunsthochschule für Medien zu Köln, FORWISS u.a. Nähere Informationen zum EMBASSI gibt [17].

In beide Projekte spielen Agenten eine wichtige Rolle. SeMoA stellt dafür die notwendige Infrastruktur zur Verfügung. Es ist zu erwarten, dass das im Rahmen dieser Arbeit erstellte *Dialogsystem zur multimedialen Interaktion zwischen Mensch und Agenten* den Ausgangpunkt zur Entwicklung der zentralen Komponenten für die Benutzerinteraktion in beiden Projekten darstellt. Aus diesen Gründen wurde bei der Systementwicklung besonders darauf geachtet, dass das System später einfach zu erweitern ist.

[3] allgegenwärtigen

Kapitel 7

Lösungsansatz

In diesem Kapitel soll der Lösungsansatz, der bei der Analyse und des Designs verfolgt wurde, erläutert werden. Da die Analysedokumente nur das Endergebnis der Überlegungen präsentieren, ist es zweckmäßig die zugrunde liegende Lösungsidee vorab zu skizzieren. Dies erleichtert dem Leser das Nachvollziehhen des Analyse- und Designentscheidungen in den folgenden Kapiteln. In der Praxis ist die hier dargestellte Lösungsidee nicht vorab entstanden, sondern erst nach vielen Interationen innerhalb der Analysephase und durch das Studium der im Teil I dargestellten Theorie.

7.1 Das Verstehen der Benutzereingaben

Zunächst muss das Problem des Sprachverstehens gelöst werden, da ein sicheres Verstehen eine Übersetzung in eine semantisches Protokoll bedingt. In Abschnitt 3.2.4 wurden drei Möglichkeiten für das Verstehen natürlicher Sprache aufgezeigt: Wordspotting, Syntaxanalyse und reguläre Grammatiken. Bei näherer Betrachtung dieser drei Ansätze zeigt sich, dass die Wordspottting-Methode für dieses Projekt nicht infrage kommt, da sie nur in ganz eng umrissenen Gesprächen mit Erfolg eingesetzt werden kann. Die Syntaxanalyse hingegen kann zwar die Tiefenstruktur eines Satzes erkennen, gibt aber keine Hinweise darauf, wie diese *semantisch* zu interpretieren ist. Das Zuwenig an Information des Wordspottings wird bei der Syntaxanalyse ein Zuviel.

Einen praktikablen Mittelweg stellen die regulären Grammtiken dar. Hier kann, je nach Anwendung, mehr oder weniger aufwendig beschrieben werden, was der Benutzer antworten darf. Äussert der Benutzer einen Satz, der nicht von der Grammatik abgedeckt ist, helfen ihm inkrementale Prompts weiter.

Nun stellt sich die Frage, woher der Agent die Information bekommt, welchen Inhalt der Dialog haben soll. Da Software-Agenten wirklich, d.h. in einem kognitiven Sinne, intelligent sind, muss der Entwickler des Agenten diesem zu irgendeinem Zeitpunkt mitteilen, worüber sie mit dem Kunden sprechen sollen. Im vorliegenden System geschieht dies mit Hilfe einer *Dialogbeschreibung*, die der Agent mit sich führt. Diese, in einer formalen Sprache abgefasste Beschreibung, teilt dem Agenten mit, welche Fragen er dem Kunden stellen soll und welchen Verlauf das Gespräch, abhängig von den erhaltenen Antworten, nehmen soll. Wenn aber — sozusagen problem-immanent — ohnehin eine Beschreibung des Dialogs existieren muss, kann diese auch die möglichen Antworten des Benutzers in Form einer regulären Grammatik enthalten.

7.2 Die Dialogbeschreibung

Als Beschreibungssprache für den Dialog wurde sich für VoiceXML [83] entschieden. Bei VoiceXML handelt es sich um ein Subset von XML zur Automatisierung von Call-Centern und Realisierung interaktiver Telefonansagedienste. Die Spezifikation von VoiceXML stammt von einem Konsortium, das gegründet wurde von Sun Microsystems, IBM, Motorola, Lucent und AT&T. Die genannten Firmen sind schon seit Jahren auf dem Gebiet des Sprachverstehens und der Automatisierung von Dialogen tätig. Dies äußert vor allem darin, dass alle Firmen schon vor Jahren eine eigene Dialogbeschreibungssprachen[1] spezifiziert haben, denen jedoch nie eine Bedeutung zugekommen ist. Nun haben sie durch die Gründung des VoiceXML Consortiums ihre Anstregungen gebündelt, um VoiceXML als de-facto-Standart für Dialogbeschreibungen zu etablieren. Leider gibt es nach derzeitigem Stand der Erkenntis nur einen Interpreter für VoiceXML-Dateien. Dieser stammt von IBM und — es sei gleich an dieser Stelle erwähnt — kann *nicht* als Klassenbibliothek

[1] z.B. SpeechML, VoiceML oser SABLE

im Rahmen dieser Arbeit verwendet werden. Der Interpreter ist Teil einer eigenständigen Applikation, die keine Kontrolle über die verwendeten Ein- und Ausgabegeräte erlaubt. Im Rahmen der vorliegenden Arbeit wird daher ein eigener VoiceXML-Interpreter entworfen und implementiert.

In Abbildung 7.1 ist das Beispiel einer Dialogbeschreibung in VoiceXML dargestellt.[2] Zunächst wird in diesem Beispiel ein Formular `pizza_order` definiert, das seinerseits ein Dialogfeld `topping` enthält. Dieses besteht aus Ausgaben (`<prompt>`) und möglichen Eingaben (`<grammar>`). Letztere werden in einer, der Backus-Naur-Form ähnlichen, Syntax[3] angegeben. Um den Dialog adaptiver und abwechlungsreicher gestalten zu können, besteht die Möglichkeit mehr als einen Prompt zu definieren. Weitere Prompts werden mit einem `count`-Attribut versehen. Dadurch können inkrementale Prompts realisiert werden. Bei jeder Ausgabe wird ein feldspezifischer Zähler inkrementiert. Antwortet der Benutzer falsch, d.h. in einer Form, die nicht von der Grammatik abgedeckt ist, wird jeweils der Prompt mit dem höchsten `count`-Attribut, das kleiner oder gleich dem internen Feldzähler ist, ausgegeben.

Eine wichtiges Detail von VoiceXML ist die Möglichkeit, *Result-Tags* in einer Antwortgrammatik zu definieren. Im VoiceXML-Beispiel sind sie in geschweifen Klammern dargestellt, z.B. `{tonno}`. Passt eine Benutzereingabe auf einen Satz in der Grammatik, stellen die enthaltenen Tags das Ergebnis des Dialoges dar. Dies ist für das zu entwickelnde Dialogsystem von besonderem Vorteil, denn bei geschickter Wahl der Tags wird eine Formalisierung der Benutzereingabe erreicht. D.h., ein beliebig[4] gestalteter Satz wird auf einen *wohldefinierten* String abgebildet.

Diese Funktionalität ist eine der zentralen Lösungsstrategien des in dieser Arbeit entwickelten Dialogsystems. Die Ergebnis-Tags werden verwendet, um die *Semantik* der Benutzereingabe auf *wohldefinierte Begriffe* abzubilden. Diese Begriffe ermöglichen dann die Übersetzung in ein semantisches Protokoll. Bei dem Übersetzungsvorgang handelt es sich dann nur noch um eine Transskription einer formalen

[2] An dieser Stelle wird nur ein Ausschnitt der Dialogbeschreibung präsentiert. Die vollständige Datei findet sich in Anhang C.

[3] Tatsächlich handelt es sich bei der Syntaxdefinition um das von Sun eingebrachte *Java Speech Grammar Format* (JSGF), das im Zusammenhang mit der Java Speech API [74] definiert worden ist.

[4] Hier ist beliebig im Rahmen der Grammatik gemeint.

```
<vxml>
    <form id="pizza_order">
    <field name="size">
        <prompt>
        Hello, I’m your pizza order agent.
        I can order various pizzas for you.
        What kind of pizza would you like?
        </prompt>
        <prompt count=2>
        Please tell me the topping of your pizza.
        You can have tunafish, mushrooms or
        tomatoes.
        </prompt>
        <grammar>
        ( [I’d like to have a] |
          [I want a]           |
          [Give me a]          )
        pizza with
        ( tunafish  {tonno}  |
          mushrooms {funghi} |
          tomatoe   {napoli} )
        </grammar>
    </field>
    ...
    </form>
    ...
</vxml>
```

Abbildung 7.1: Beispiel einer Dialogbeschreibung in VoiceXML

Sprache in eine andere formale Sprache. Dies kann nach vorgegebenen Schablonen erfolgen.

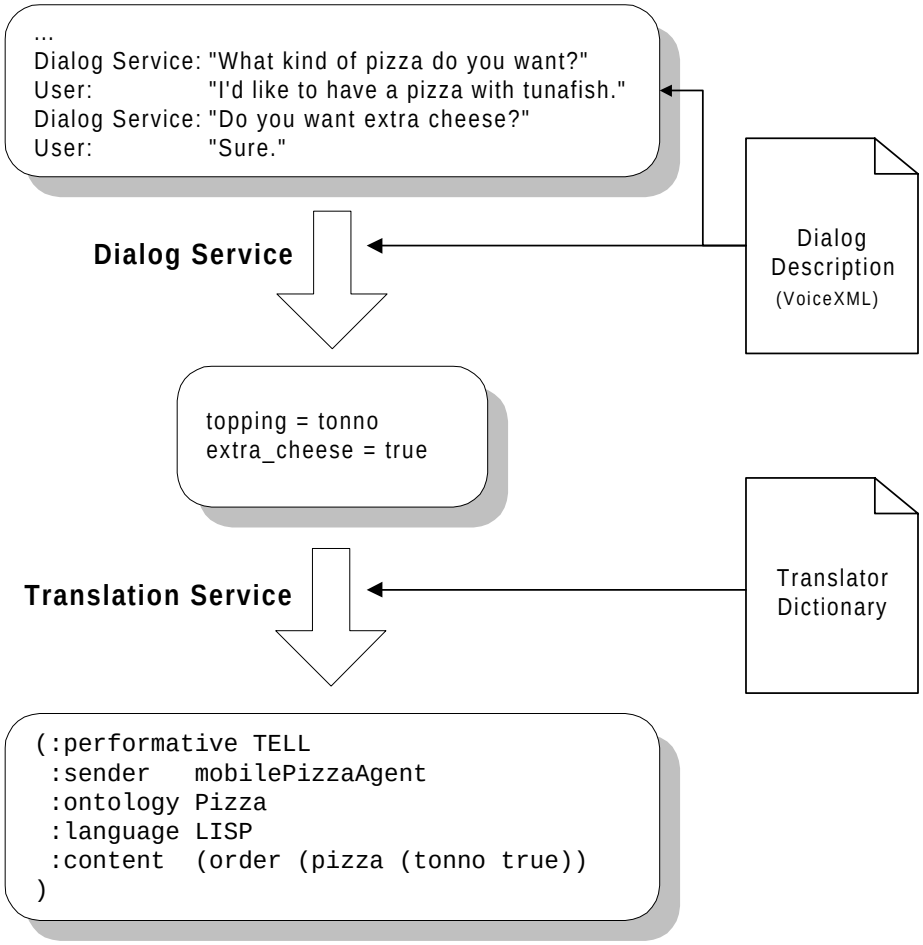


Abbildung 7.2: Übersetzung eines Dialoges nach KQML

Abbildung 7.2 stellt den Übersetzungsvorgang dar. Aus dem Dialog werden nach Maßgabe der Dialogbeschreibung und den Antworten des Benutzers die Ergebnis-Tags extrahiert. Auf Grundlage der Ergebnis-Tags und mit Hilfe von Übersetzungsregeln (*Translator Dictionary*) wird dann die Agentenkommunikationsnachricht erzeugt.

Die Result-Tags werden, gemäß der VoiceXML-Spezifikation, als Schlüssel-Wert-Paare zurückgegeben. Der Schlpüssel ist dabei der Name des Dialogfeldes, in dem das Tag entstanden ist (z.B. `topping`), der Wert ist das Tag selbst (z.B. `tonno`).

Die Menge von Schlüssel-Wert-Paaren eines Dialoges wird als *Dialog-Ergebnis* bezeichnet. VoiceXML sieht für die Weitergabe von Ergebnissen sogar einen Mechanismus vor, mit dem der Autor der Dialogbeschreibung festlegen kann, welche Felder in das Dialog-Ergebnis aufgemnommen werden sollen. Eine VoiceXML Anweisung hierfür sieht wie folgt aus: `<goto submit="topping extra_cheese"/>` In diesem Falle werden nur zwei Schlüssel-Wert-Paare zurückgegeben, z.B. `topping=tonno` und `extra_cheese=true`, obwohl während des Dialoges u.U. wesentlich mehr Felder durchlaufen wurden.

7.3 Der Übersetzungsvorgang

Die Übersetzungsregeln dienen, wie erwähnt, als Schablonen um das Dialog-Ergebnis in die Form einer KQML-Nachricht zu bringen. Der Agent führt diese Übersetzungsregeln in Form einer *Dictionary*-Datei mit sich. Diese wurde ihm, wie die Dialogbeschreibung, von seinem Auftraggeber mitgegeben. Das Dictionary besteht aus einer Sammlung von Übersetzungsregeln. Abbildung 7.3 zeigt ein solches Dictionary.

```
# Global settings
sender   = mobilePizzaAgent
language = LISP
ontology = pizzaOrder

[order]
performative = TELL
content = (order (pizza ($(topping) $(extra_cheese))))

[price request]
performative = ASK
content = (price (amount $(count))
                 (pizza ($(topping) $(extra_cheese)))
          )
```

Abbildung 7.3: Ausschnitt aus einer Übersetzungsbeschreibung für KQML

Das dargestellte Beispiel-Dictionary enthält zwei Regeln, `order` und `price request`. Welche der Regeln zur Übersetzung herangezogen wird, entscheidet der Agent vor der Übersetzung. Im ersten Teil des Dictionaries sind globale Vereinbarungen abgelegt; sie enthalten Definitionen, die für alle Regeln gültig sind und dienen damit der Übersichtlichkeit. Der semantische Inhalt der Nachrichten ist in diesem Beispiel jeweils in LISP abgefasst. Innerhalb der Regeln darf mit Variablen gearbeitet werden. Sie werden mit `$(variablenname)` angegeben. Die Variablen werden beim Übersetzen mit den Werten des Dialog-Ergebnisses gefüllt. Diese Vorgehensweise ist sehr einfach und bietet eine hohe Flexibilität.

7.4 Systemarchitektur

Die Systemarchitektur des skizzierten Lösungsansatzes wird in Abbildung 7.4 gezeigt. Dialog- und Translationssystem sind jeweils als eigener SeMoA Service realisiert. Dadurch braucht der Mobile Agent keinerlei Fähigkeiten zum Sprachverstehen oder zum Übersetzen. Der Entwickler eines Agenten braucht sich nur um die Erstellung der Dialogbeschreibung und das Dictionary zu kümmern. Diese Architektur erlaubt zudem, einen Agenten für viele ähnliche Aufgaben einzusetzen; es müssen nur die beiden Dateien an die jeweilige Aufgabe anpassen werden.[5] Die Tatsache, dass der Agent selbst keine Fähigkeiten zur Spracheerkennung benötigt, sondern nur einen Service benutzt, macht ihn sehr leichtgewichtig. Dies ist ein wichtiges Akzeptanzkriterium bei Agenten im Umfeld mobiler Anwendungen.

Das hier vorgestellte Dialogsystem verwendet die beiden Services *DialogService* und *KQMLTranslationService*. Möchte der Agent einen Dialog mit dem Benutzer führen, übergibt er dem Dialog-Service eine Beschreibung des zu führenden Dialoges. Typischerweise führt der Agent die Dialogbeschreibung als Datei mit sich; es spricht aber prinzipiell nichts dagegen, dass sie erst zur Laufzeit vom Agenten kreiert oder modifiziert wird. Der Dialogservice nimmt die auszuführende Dialogbeschreibung entgegen und sortiert sie in eine FIFO-Warteschlange ein. Sobald der Dialog zur Ausführung kommt, wird die Beschreibung einem VoiceXML-Interpreter übergeben, der die Datei parst und die notwendigen Ein- und Ausgaben

[5] Der Entwickler könnte einem Agenten sogar mehrere Dialogdateien für unterschiedliche Aufgaben mitzugeben.

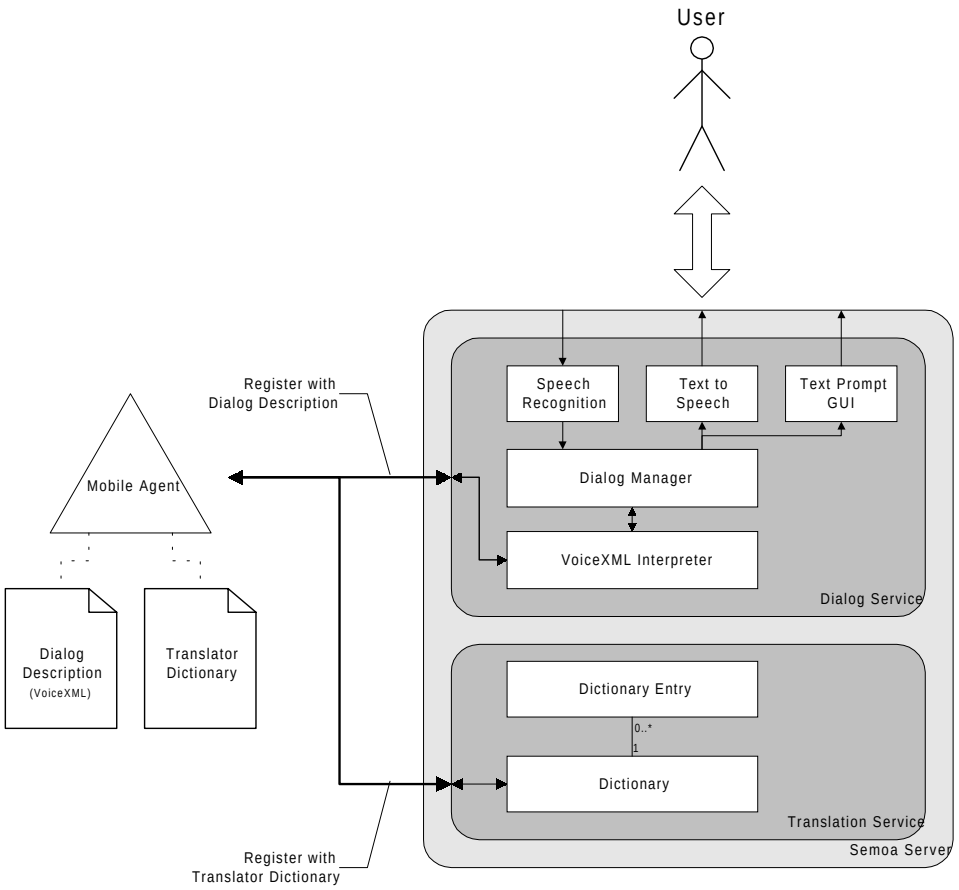


Abbildung 7.4: Systemarchitektur des Dialogsystems

veranlasst. Der VoiceXML-Interpreter steuert den Kontrollfluss im Dialog und hält auch die Result-Tags und den Dialogzustand vor. Die Ausgabe des VoiceXML-Interpreters ist jedoch geräteunabhängig konzipiert.

Muss eine Ausgabe an den Benutzer erfolgen, benachrichtigt er den *DialogDeviceManager*. Dieser sendet dann einen Ausgabeaufforderung an alle ihm bekannten Ausgabegeräte, die daraufhin die Ausgabe auf ein angeschlossenes Gerät vornehmen. Im Rahmen dieser Arbeit wurden Ausgabegeräte entwickelt, die via Sprachsynthese eine Sprachausgabe, sowie eine Textausgabe auf dem Bildschirm erlauben. Das Einbinden weiterer Geräte stellt kein Problem dar; so könnte z.B. ohne großen Implementierungsaufwand eine Braille-Zeile[6] oder ein Avatarsystem als

[6]Eine Braillezeile ist ein Gerät zur Darstellung von Text in Blindenschrift.

Ausgabekanal integriert werden.

Das Entgegennehmen von Eingaben funktioniert analog zur Ausgabe. Die vom DialogDeviceManager verwalteten Eingabegeräte melden diesem eine gültige, d.h. eine der regulären Grammatik entsprechende, Eingabe. Hierbei können mehrere Geräte um die Eingabe konkurrieren. In dem vorliegenden System wird vom Dialog-Manager die erste gültige Eingabe als Antwort gewertet. Das bedeutet, der Benutzer kann sich bei jeder Interaktion je nach Art der Frage für den ihm angenehmsten Modus entscheiden.[7]

Ist die Dialogbeschreibung vom VoiceXML-Interpreter abgearbeitet worden, werden die Dialog-Ergebnisse an den Agenten zurückgegeben. Der TranslationService übersetzt, im Auftrage des Agenten, die Resultate des DialogServices in eine formale Sprache. Im vorgestellten System wird eine Übersetzung in KQML vorgenommen. Möchte der Agent den Translation-Service verwenden, übergibt er diesem einen gültigen Regelnamen des Dictionaries und eine Menge von Schlüssel-Wert-Paaren, das Dialog-Ergebnis. Der Service sucht im ersten Schritt den übergebenen Regelnamen im Dictionary. Aus den dort angegebenen, sowie den globalen Informationen wird die KQML Nachricht zusammengesetzt. Zuvor werden jedoch die, in der gewählten Rubrik verwendeten, Variablen mittels der Schlüssel-Wert-Paare substituiert.

7.5 Vorabbewertung des Lösungsansatzes

Der vorgestellte Lösungsansatz ermöglicht aus Sicht des Agenten bzw. seines Auftraggebers größtmögliche Flexibilität zur Laufzeit. Der Agent kann, trotz fest definierter Dialogbeschreibung und Dictionaries, an folgenden Punkten Einfluss auf die Auswertung des Dialoges nehmen:

- Der Kontrollfluss des Dialoges kann innerhalb der Dialogdatei von Antworten des Benutzers abhängig gemacht werden.
- Je nach Ablauf des Dialoges können andere Dialogvariablen als Ergebnis zurückgegeben werden.

[7] Diese Architektur bereitet eine Multimodalität, bei der sich mehrere Eingabemodi ergänzen, bereits vor. Siehe Abschnitt 12

- Die Übersetzungsregel kann vom Agenten anhand des Dialog-Resultats zur Laufzeit ausgewählt werden.

- Der Agent kann das Dialogresultat, d.h. die Schlüssel-Wert-Paare vor der Übersetzung modifizieren.

- Die Variablenexpansion erlaubt zur Laufzeit parametrisierbare Nachrichten.

Für den Auftraggeber bzw. den Entwickler[8] des Agenten, ist die Integration von Dialogfähigkeiten in einen Agenten sehr einfach. Im besten Falle muss er sich nur noch auf die Planung der Dialoge und die durch diese Dialoge ausgelösten Aktionen konzentrieren. Damit wird das eigentliche Problem der Spracherkennung und des Sprachverständnisses auf die Ebene der Dialoginhalte abstrahiert.

Vergleicht man die skizzierte Lösung mit den Anforderungen, die in Abschnitt 6.1 an das System gestellt wurden, wird deutlich, dass alle Randbedingungen erfüllt sind:

1. Anforderung — “Die Schnittstellen zum Agenten wie auch zum Benutzer hin sollen klar definiert sein”:
   Für den Agenten besteht eine sehr schmale, effektive Schnittstelle zu den Services hin. Der Mensch kann mit dem System multimodal interagieren. Damit ist die Interaktion wesentlich intuitiver, als bei klassischen Anwendungsprogrammen.

2. Anforderung — “Die Lösung soll möglichst allgemein sein, so dass sie sich leicht an neue Dialoge und sprachliche Kontexte anpassen lässt”:
   Durch die Verwendung einer Dialogbeschreibung ist die Anpassung eines Agenten an eine neue Aufgabenstellung allein durch die Änderung der Dialogbeschreibungsdatei möglich. Dies bietet das gleiche Maß an Flexibilität, wie es Inhaltsanbieter von Internetseiten haben.

3. Anforderung — “Der Benutzer soll nicht über Gebühr in Wortwahl und Satzbau eingeschränkt werden”: Die Verwendung von regulären Grammatiken

[8] Der Verfasser der Dialogbeschreibung und des Dictionaries kann, muss aber nicht, der Auftraggeber des Agenten sein.

als Grundlage des Sprachverstehens, legt es in das Ermessen des Dialogentwicklers, wie flexibel die Antworten sein können. Wird hierbei eine ähnliche Akrebie wie bei der Erstellung von Internetseiten aufgewandt, ist das Resultat ein sehr ausgefeilter, vielfältiger und flexibler Dialog. Dies wird noch verstärkt durch den Lerneffekt des Benutzers bei der Verwendung von inkrementalen Prompts.

4. Anforderung — "Die Lösung soll einfach um grammatische und lexikalische Möglichkeiten erweiterbar sein":
   Diese Anforderung ist durch die Verwendung von regulären Grammatiken ebenfalls gewährleistet.

5. Anforderung — "Entwickler eines Agenten bzw. eines Softwaresystems sollen das Produkt möglichst einfach einsetzen können": Da der Agent braucht die angebotenen Services nur aufzurufen. Daher ist der Implementierungsaufwand im Agenten marginal. Der Aufwand für die Erstellung der Dialogdatei, muss, wie oben dargelegt, bei jeder anderen Systemarchitektur ebenfalls erbracht werden und ist vergleichsweise einfach.

Es soll noch einmal wiederholt werden, dass diese Lösung das Ergebnis einer langen Problemanalyse des Problem ist. Sie wird hier in dieser gerafften Form dargestellt, damit man sich in den Kapiteln *Analyse* und *Design* besser auf die Entwicklungsentscheidungen konzentrieren kann.

Zuvor soll noch — ebenfalls aus Gründen der Übersichtlichkeit — die methodische Vorgehensweise bei Analyse und Design vorgestellt werden. Dann braucht man sich bei Analyse und Design nur auf die Entwicklungsentscheidungen zu konzentrieren.

Kapitel 8

Vorgehensweise bei Analyse und Design

In diesem Kapitel wird die Methode vorgestellt, auf die sich der Entwurf des Softwareprojektes gründet.

Für die Analyse und das Design kommen Methoden, Prinzipien und Konzepte des Objektorientierten Software Enigneerings zur Anwendung. Die Vorgehensweise stützt sich im Wesentlichen auf die Methoden nach RAUSCH [60, 59]. Als Beschreibungssprache wird die *Unified Modeling Language* (UML) verwendet. Natürlich kann dieses Kapitel kein Ersatz für eine Abhandlung über die Konzepte des objektorientierten Software-Engineerings sein. Neben den erwähnten Quellen von RAUSCH auch JACOBSON/BOOCH/RAMBAUGH [5] und FOWLER [27] empfohlen.

Grundsätzlich sind alle hier vorgestellten Vorgänge “hochgradig iterativ” [60], d.h. man muss Entwürfe mehrfach überarbeiten und gegen die Ergebnisse des vorhergehenden Schritts validieren, bevor man zum nächten übergeht.

8.1 Bilden des Analysemodells

Beim Bilden des Analysemodells wird die Lösung des gestellten Problems auf einer logischen Ebene entworfen und dargestellt. Sie ist vollkommen unabhängig von der späteren Implementierung. Ausgehend von der Beschreibung des Problems

wird man Schritt für Schritt der Lösung näher gebracht. Das Ergebnis der Analyse ist ein Lösungmodell aus logischer Sicht.

8.1.1 Use Case Diagramm

In der Analysephase wird zunächst ein *Use Case Diagramm* des zu entwickelnden Systems entworfen. Das Use Case Diagramm setzt sich aus Anwendugsfällen[1] und Aktoren[2] zusammen und beschreibt das System "von Außen". Als Aktoren werden Entitäten bezeichnet, die mit dem System interagieren. Der besondere Fokus dieses Diagrammtyps liegt auf der, aus Definition der Aufgabe abgeleiteten, Abgrenzung des zu entwickelnden Systems von seiner Umgebung. Es muss beim Erstellen des Diagramms darauf geachtet werden, dass sich alle Anforderungen der Aktoren an das System in Use Cases wiederfinden. Use Case Diagramme können auch hierarchisiert werden, wenn die Komplexität der Aufgabe dies verlangt.

8.1.2 Szenarien

Zu den einzelnen Use Cases des Use Case Diagramms wird je eine Beschreibung in Form von *Szenarien* erstellt. Diese Szenarien haben einen streng sequentiellen Charakter und beschreiben den typischen Ablauf einer Interaktion des Aktors mit dem System in einem speziellen Anwendungsfall. Dabei bleiben mögliche Störquellen und Ausnahmesituationen, die während der Interaktion auftreten könnten, im Szenario unberücksichtigt. Begonnen wird mit dem "externen Szenario", das nur die Kommunikation zwischen Aktor und System darstellt. Das System selbst wird dabei als "Black-Box" betrachtet. Das externe Szenario wird dann in einem "internen Szenario" dahingehend verfeinert, dass jetzt auch die systeminterne Kommunikation beschrieben wird. Dieser Schritt von der Black-Box- zur White-Box-Sichtweise auf das System ist mit äußerster Sorgfalt vorzunehmen, da dies die erste *interne* Sicht auf das System ist. Werden in dieser Phase Fehler gemacht, pflanzen sich diese mindestens bis ins Analysemodell fort und führen zu zeit- und kostenintensiven Iterationen im Software-Lifecycle [59].

[1] engl.: *use case*
[2] engl.: *actors*

8.1.3 Interaktionsdiagramme

Ist die Erstellung der Szenarien abgeschlossen, werden daraus *Interaktionsdiagramme* abgeleitet. Sie entstehen durch die Identifikation von logisch an der Lösung beteiligten Entitäten[3], und die Formalisierung der, in den Szenarien enthaltenen, Interaktion zwischen diesen Entitäten. Aus den gewonnenen Erkenntnissen lässt sich ein Sequence Diagramm ableiten, das die agierenden Objekte und die gesendeten Nachrichten enthält. Sind diese Diagramme erstellt, sollten sie nochmals auf logische Kohärenz hin untersucht werden. Dabei lautet die Fragestellung, ob das einzelne Objekt und die an dieses gesendeten Nachrichten logisch zusammenpassen. Da sich im nächsten Schritt aus den Interaktionsdiagrammen — fast automatisch — ein erstes Analyse-Klassendiagramm ergibt, werden hier die Weichen für eine gute Objektschnittstellen gestellt.

8.1.4 Analyse-Klassendiagramm

Das *Analyse-Klassendiagramm* präsentiert die Essenz des Systems in einer logischen Sichtweise. Es zeigt die an der Lösung des Problems beteiligten Klassen, ihre Methoden und die Assoziationen zwischen den Klassen. Alle drei genannten Elemente des Diagramms lassen sich aus den Interaktionsdiagrammen ablesen. Der wichtigste Schritt ist die Entscheidung, ob ein, im Interaktionsdiagramm dargestelltes Objekt, zu einer Klasse verallgemeinert wird. Bei durchdachten, essentiellen Szenarien und Interaktionsdiagrammen lassen sich i.d.R. alle Objekte in Klassen umwandeln, da keine redundanten Objekte enthalten sein können. Die Methoden der gefundenen Klassen ergeben sich einfach aus den Nachrichten im Sequence Diagramm. Jede, von einem Objekt empfangene Nachricht, stellt in der korrespondierenden Klasse eine Methode dar. Die Assoziationen ergeben sich aus Nachrichtenaufrufen zwischen zwei Objekten im Seqence Diagramm. Sendet ein Objekt A eine Nachricht an das Objekt B, so manifestiert sich dies im Analyse-Klassendiagramm in einer Assoziation zwischen den korrespondierenden Klassen A und B. Die Assoziationen werden mit geeigneten Bezeichnern versehen. Sind alle Assoziationen erstellt, werden deren Kardinalitäten festgelegt. Diese ergeben sich logisch aus der Art der Relation. Das Klassendiagramm zeigt die Klassen und

[3] Instanzen der Klassenkandidaten

ihre Beziehungen untereinander und gibt damit einen Überblick über die statischen Aspekte der Systemarchitektur.

8.1.5 Objektrelationen

Nun werden die Assoziationen im Einzelnen näher betrachtet. Bisher sind sie in ihrer Art nicht näher spezifiziert worden. Die Benennung und die Festlegung der Kardinalität der Beziehung, die am Ende der Analysephase festgelegt wurden, helfen dabei die Art der Relation festzulegen. Die wichtigsten Beziehungen zwischen Objekten sind:

- Aggregation
- Vererbung
- Unidirektionale Relation
- Bidirektionale Relation
- Instatiierung

Das Wesen der Relationen wird als bekannt vorausgesetzt. Die Identifizierung des richtigen Relationstyps geschieht anhand der bisher erstellten Dokumente und anhand von Wissen aus dem Problembereich. Der Erfahrung nach sollten eindeutige Spezifikationen zuerst festgelegt werden. Häufig ist die Beschaffenheit anderer Beziehungen dann leichter zu erkennen.

8.1.6 Attribute

Die Attribute einer Klasse ergeben sich zum einen aus problemspezifischem Wissen über Art und Aufgabe der Klasse selbst. Weiterhin kann man die Methoden der Klasse untersuchen. Liefern sie Ergebnisse an den Aufrufer zurück, so müssen diese Daten, oder andere, aus denen sich die gefragten Werte berechnen lassen, ebenfalls als Attribute vorhanden sein.

Weitere Attribute der Klasse finden sich oft in unidirektionalen Assoziationen zwischen den Klassen. Sie werden in der referenzierenden Klasse häufig durch ein Attribut ausgedrückt, das eine Referenz auf ein Objekt repräsentiert. Darüber hinaus

kann man die bisher erstellten Analysedokumente nach “verborgenen” Attributen untersuchen. Grundsätzlich sollte eine Klasse so wenig Attribute wie möglich, aber so viele wie nötig enthalten.

8.1.7 Operationen

Die nach Außen sichtbaren Operationen jeder einzelnen Klasse lassen sich wie die Attribute in den Interaktionsdiagrammen, und hier am besten im Sequence Diagram, ablesen. Jede von einem Objekt empfangene Nachricht, resultiert in einer Methode in der korrespondierenden Klasse. Dieses Vorgehen garantiert, dass alle Operationen vorhanden sind, die zur Erfüllung der Szenarien vorhanden sein müssen. Darüber hinaus kann man weitere Operationen identifizieren, indem man die Problemdefinition, die Use Cases oder problemspezifisches Wissen verwendet. Schließlich kann man für jedes einzelne Attribut der Klasse prüfen, ob man es setzen oder lesen können muss.

8.1.8 Zustandsdiagramme

Um das Analysemodell[4] zu vervollständigen, müssen noch die dynamischen Aspekte der Klassen betrachtet werden. Das dafür zu entwickelnde Verhaltensmodell spiegelt die dynamischen Aspekte der einzelnen Klassen wider. Das Verhalten des gesamten Systems nach Außen wird bereits vom Use Case Diagramm dokumentiert. Was noch erfolgen muss, ist die Modellierung der internen dynamischen Vorgänge, also mögliche Zustandsänderungen der einzelnen Objekte. Diese werden in Zustandsdiagrammen dargestellt. Dies erfolgt allerdings nur für Klassen, die eine gewisse Dynamik aufweisen. Typischerweise wird, wenn es überhaupt, nur ein geringer Teil der Klassen ein dynamisches Verhalten zeigen.

Sind die beschriebenen Schritte zur Zufriedenheit des Entwicklers durchgeführt, ist die Analysephase abgeschlossen, es schließt sich die Designphase des Projektes an. Bemerkenswert ist, dass bisher in allen Teilschritten das zu entwickelnde System

[4] Ein Analysemodell besteht aus den Komponenten Analyse-Klassendiagramm (auch “Objektmodell”) und dem Verhaltensmodell. Vgl. [60]

nur unter logischen Gesichtspunkten betrachtet wurde. Das bedeutet, die Lösung ist noch vollkommen unabhängig von der späteren Implementation.[5]

8.2 Bilden des Designmodells

Nun muss das Analysemodell in eine physikalische, d.h. implementationsfähige Repräsentation überführt werden. Durch das in der Analyse erarbeitete, sehr detaillierte Klassendiagramm, ist das Design vergleichsweise einfach. Gegen Ende des Design muss man sich auf eine Sprache zur Implementaion festlegen, da Entscheidungen nur getroffen werden, wenn diese bekannt ist.

Ausgehend vom Analysemodell ist das Klassendiagramm um folgende Aspekte zu erweitern:

- Schnittstellen
- Abstrakte Klassen
- Anbindung an externe Klassenbibliotheken

Diese Erweiterung führt nicht unbedingt zur Einführung neuer Klassen, da auch bekannte Klassen als Schnittstellen deklariert werden können. Im Anschluss daran wird noch die Bildung von Paketen vorgenommen.[6] Aus Platzgründen erfolgt die Beschreibung der Designaspekte unter dem Gesichtspunkt einer späteren Implementierung in Java™. Die meisten der dargestellten Sachverhalte sind jedoch sprachunabhängig bzw. können leicht auf andere Kontexte angewendet werden.

8.2.1 Schnittstellen

Schnittstellen finden sich im Klassendiagramm oft an Stellen, die allgemein zugänglich genutzt werden sollen, oder, wenn verschiedenartige Klassen dieselben

[5] Bemerkenswerterweise geht das dargestellte Verfahren nicht mal von Software als Realisierungsform aus, sondern nur von klassifizierbaren Entitäten, die die Fähigkeit haben, miteinander zu kommunizieren. Das bedeutet, mit der dargestellten Methode ließen sich z.B. auch Geschäftsprozesse modellieren. Die handelnden Entitäten wären dann z.B. Abteilungen oder Einzelpersonen.

[6] engl.: *packaging*

Methoden anbieten sollen, sich eine Vererbungsrealtion aber logisch nicht anbietet. Schnittstellen sind auch von großem bei der Typüberprüfung von Übergabeparametern in Methodenaufrufen. Eine Klasse, die ein bestimmtes Interface implementiert, kann, als Übergabeparameter verwendet, auch von einer fremden Klasse mittels dieses Interfaces definiert angesprochen werden.[7]

Wird eine Klasse als Interface definiert, zieht dies meist auch die Einführung von implementierenden Klassen nach sich. Diese werden dann mit einer *implements*-Beziehung im Klassendiagramm eingetragen. Speziell in Java™helfen Interfaces unübersichtliche Mehrfachvererbungen zu vermeiden.

8.2.2 Abstrakte Klassen

Abstrakte Klassen finden sich im Klassendiagramm an Stellen, die eine Vererbung aufweisen. Möchte man den Nutzer einer Klasse zwingen, ein gewisses Verhalten selbst zu implementieren, ihm den Rest der Implementierung jedoch vorgeben, bieten sich abstrakte Klassen an.

Häufig werden sie auch in Verbindung mit finalen, d.h. nicht reimplementierbaren, Methoden eingesetzt. SeMoA stellt auf diese Weise z.B. in der Klasse Service$^{(Class)}$ sicher, dass der Benutzer der Klassenbibliothek an bestimmten Stellen Methoden überschreiben *muss*, dies aber an anderer Stelle *nicht darf*.

8.2.3 Anbindung an Klassenbibliotheken

Die Anbindung an externe Bibliotheken erfolgt meist in Form einer Vererbung, einer *instatiiates*- oder einer *implements*-Beziehung. Instatiierende Beziehungen ergeben sich, wenn man externe *Klassen* innerhalb von Methoden nutzt. Implementierende Beziehungen entstehen, wenn man ein Interface der Klassenbibliothek verwendet. Vererbungen bieten sich an, wenn man vorhandenes Verhalten einer externen Klasse vollständig kopieren und erweitern möchte. Bei der vorgestellten Form der essentiellen Analyse kommen Vererbungen erfahrungsgemäß eher selten vor, da man sich bei der Analyse nur auf das Problem und nicht auf bereits vorhandene Teillösungen konzentriert.

[7] Ein gutes Beispiel ist das Listener-Konzept im Java Abstract Window Toolkit (AWT). Über Interfaces lassen sich beliebige Klassen als Event-Listener anmelden.

In der Designphase ist es übrigens relativ einfach, Assoziationen zu externen Bibliotheken zu knüpfen, weil die Lösung des Problems in Form des Analysemodells bereits vorliegt. Würde man sich zum Zeitpunkt der Analyse schon mit den Supporterklassen beschäftigen, würde das Analyse-Klassendiagramm und damit die Lösung sehr schnell unübersichtlich.

8.2.4 Design-Klassendiagramm

Das Ergebnis obiger Schritte ist das Design-Klassendiagramm. Wird das Software Engineering mit Hilfe eines Tools durchgeführt, kann in diesem Stadium aus dem Klassendiagramm automatisch Programmcode in der Zielsprache generiert werden. Der erzeugte Programmcode besteht aus den vollständigen Klassendefinitionen, inklusive Vererbung und Deklarationen für Schnittstellenimplementierungen, und der Deklaration von Methoden und Attributen. Jetzt können anhand der Dokumentation zu den einzelnen Nachrichten und den Zustandsdiagrammen die Methoden und das Objektverhalten implementiert werden.

8.2.5 Paketbildung

Ein gutes Packaging ist sehr wichtig für die einfache (Wieder-) Verwendbarkeit der erstellten Systemlösung. Der Entwickler sollte sich von logischen Gesichtspunkten leiten lassen.

Ein effektive Methode ist die Suche nach Schichten im Klassendiagramm. Dabei sollte eine Drei-Ebenen-Architektur angestrebt werden. Wurde das System nach essentiellen Gesichtspunkten entworfen, so sollten sich die eigenen Klassen vor allem in der mittleren Schicht finden da dies die Logik des Systems repräsentiert. Die Anbindung an Infrastrukturen (Netzwerk, Datenbanken) findet sich in der untersten Schicht. Dies wird meist von Supporterklassen erledigt. In die oberste Schicht gehören Klassen, die die Schnittstelle zur Systemumgebung realisieren.

Häufig hilft es auch, im Klassendiagramm solche Klassen auszumachen, die — losgelöst vom konkreten Engineering Projekt — ein autarkes Subsystem bilden könnten. Diese können dann in einem eigenen Package zusammengefasst werden. Dabei sollte der Aspekt der Wiederverwendbarkeit im Vordergrund stehen.

Eine dritte Herangehensweise ist die Sortierung der Klassen nach den Use Cases, in denen sie verwendet werden. Wurden in der Analyse hierarchische Use Case Diagramme verwendet, so werden sich die Pakete analog sinnvoll hierarchisieren lassen, denn sie sind eine unmittelbare Abbildung des Problems auf die Lösung.

Kapitel 9

Bilden des Analysemodells

Dieses Kapitel widmet sich der Analysephase des zu entwicklenden Dialogsystems für Mobile Agenten. Nachdem der LÖsungsansatz und die Vorgehnsweise in den vorhergehenden Kapitel vorab eingehend beschrieben wurden, werden hier nur noch die Entwicklungsentscheidungen dargelegt und begründet. Am Ende von Kapitel 9 steht das Analysemodell des Systems.

9.1 Use Case Diagramme

In Abbildung 9.1 ist die Essenz des Systems aus Anwendersicht, also aus Sicht des Mobilen Agenten und des Benutzers, dargestellt. Der Use Case *Facilitate a Conversation between Agent and User* beschreibt den Anwendungsfall, dass das zu entwickelnde System einen (sprachlichen) Dialog zwischen dem Mobilen[1] Agenten und dem Benutzer ermöglichen soll. Dieser Use Case wird von den Aktoren *Mobile Agenten* und *User* verwendet.

Der Use Case *Translate dialog result into an agent communication language* ermöglicht dem Agenten, das Ergebnis des Dialoges in eine formale Sprache, genauer in eine Agentenkommunikationssprache, zu übersetzen. Das Ergebnis der

[1] Die Eigenschaft der Mobilität ist hier nicht von Bedeutung. Das System ermöglicht natürlich ebenso die Kommunikation mit einem Stationären Agenten. Trotzdem wird der Aktor im Folgenden als *Mobiler Agent* bezeichnet.

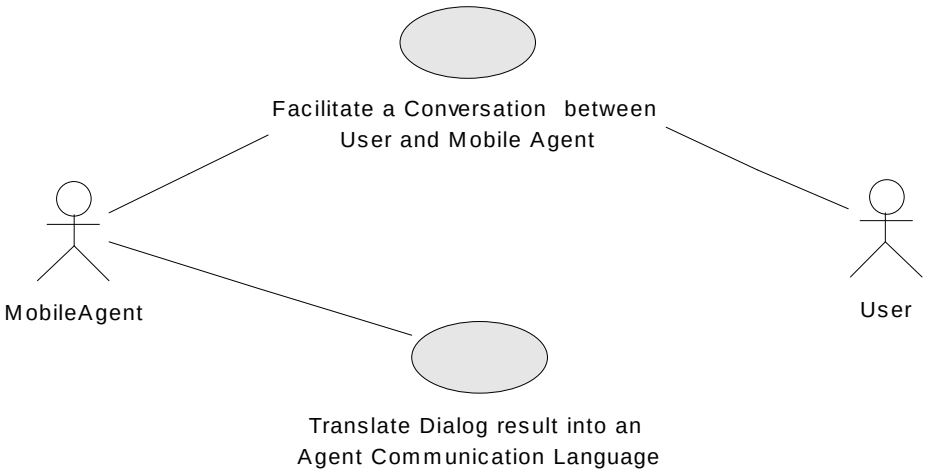


Abbildung 9.1: Use Case Diagramm des zu entwickelnden Systems

Übersetzung[2] ist eine Nachricht, die in einer Agentenkommunikationsprache abgefasst ist, und vom Agenten verstanden werden kann. Dieser Use Case wird nur vom Agenten genutzt.

Bei der Analyse des Systems wurde offenbar, dass die gegebene Aufgabe aus zwei logisch autarken Teilprojekten besteht. Zum einen muss der Agent die Möglichkeit erhalten, mit dem Benutzer zu kommunizieren, und zum anderen muss das Dialogergebnis in eine formale Sprache übersetzt werden. Die beiden Schritte bauen zwar im Sinne einer sequentiellen Nutzung durch den Agenten aufeinander auf, sie hängen aber trotz allem *nicht* voneinander ab. Das Bilden von Nachrichten in einer Agentenkommunikationssprache könnte zudem auch in anderen Zusammenhängen sinnvoll genutzt werden.

Das Gesamtsystem wird daher in zwei unabhängig voneinander agierende Subsysteme modularisiert. Das erweiterte primäre Use Case Diagramm in Abbildung 9.1 veranschaulicht dies durch die gestrichelten Hüllkurven, die in systemtheoretischer Betrachtungsweise die Grenzen der Systeme darstellen. Der Autor vertritt die Auffassung, dass beide Teilsysteme selbständige, essentielle Systeme sind und

[2] Streng genommen handelt es sich um eine *Transkription* und nicht um eine *Translation*. Hier wird trotzdem weiterhin der englische Begriff *translation* verwendet, weil das Wort *transcription* im Englischen meist im Sinne einer Ab- oder Niederschrift verwendet wird.

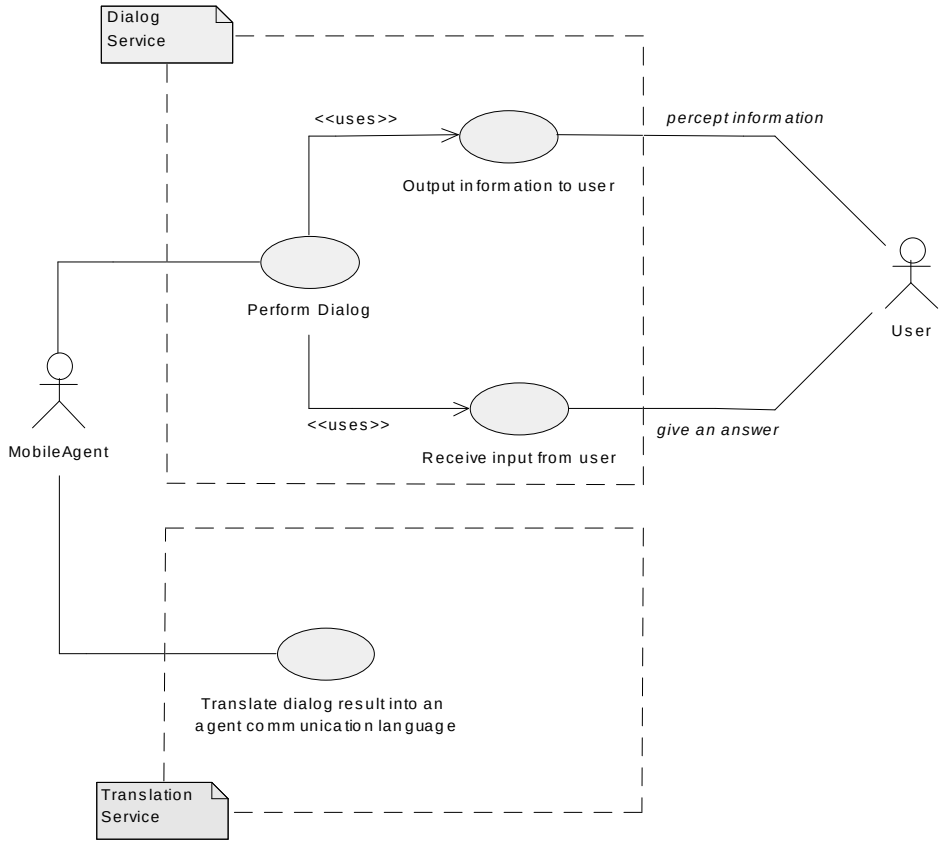


Abbildung 9.2: Erweitertes Use Case Diagramm des zu entwickelnden Systems

sie nicht notwendigerweise miteinander kommunizieren müssen. Die Kommunikation, um aus dem Dialogergebnis eine formale Nachricht zu generieren, geschieht über den Agenten, der als Aktor in systemtheoretischer Betrachtungsweise außerhalb der zu entwickelnden Systeme steht. Im Sinne einer klaren Schnittstelle der einzelnen Teilsysteme und einer besseren Wiederverwendbarkeit bei einer nicht auszuschließenden separaten Nutzung, werden beide Subsysteme getrennt entwickelt.

Das Dialogsystem wurde verfeinert, indem der Use Case *Facilitate a Conversation between User and Mobile Agent* aus Abbildung 9.1 in die Use Cases *Perform Dialog, Output Information to User* und *Receive Input from User* unterteilt wurde.

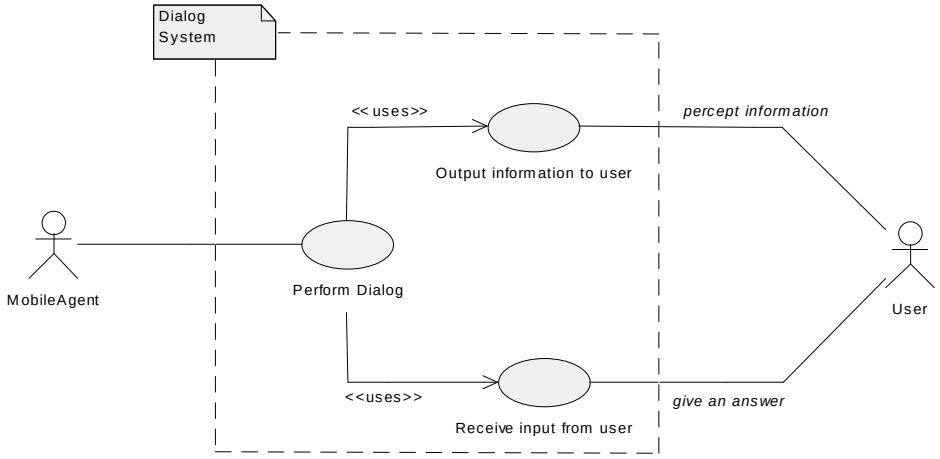


Abbildung 9.3: Internes Use Case Diagramm des Dialogsystems

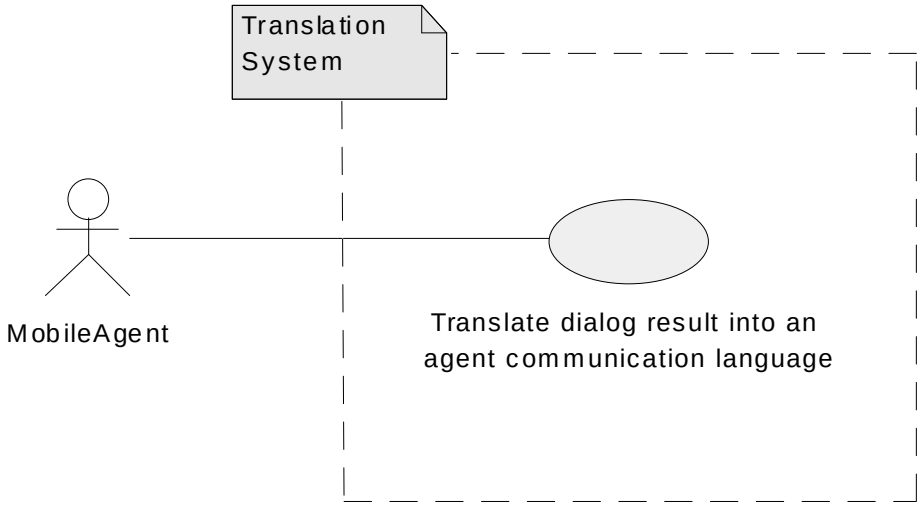


Abbildung 9.4: Internes Use Case Diagramm des Translationssystems

Die beiden letztgenannten Use Cases, mit denen ausschließlich der Benutzer kommuniziert, bereiten bereits eine Entkopplung der Ein- und Ausgabemedien von der Dialogsteuerung vor. Dies ist ein wichtiger Schritt in Richtung des Eingangs formulierten Ziels der Geräteunabhängigkeit.

Das Diagramm aus Abbildung 9.2 wird nun in die beiden, in Abbildung 9.3 und 9.4 dargestellten, internen Use Case Diagramme geteilt. Beide werden von nun an getrennt untersucht und weiterentwickelt.

9.2 Szenarien

Es wird nun für jeden Use Case das Szenario vorgestellt. Dabei wurden die Use Cases *Output information to user* und *Receive input from user* mit in das Szenario zum Use Case *Perform Dialog* aufgenommen. Diese Entscheidung ist vertretbar, da

1. die Use Cases jeweils über eine *uses*-Relation vom Use Case *Perform Dialog* abhängen,
2. diese Use Cases ausschließlich vom Use Case *Perform Dialog* genutzt werden und
3. die Interaktion zwischen den Aktoren die beiden Use Cases unbedingt voraussetzt.

Zunächst wird für jedes Teilsystem das externe Sequence Diagramm vorgestellt. Dieses wird dann erweitert zum internen Sequence Diagramm.

9.2.1 Szenarien zu “Perform Dialog”

Externes Szenario zu “Perform Dialog”

1. Der Agent übergibt dem Dialogsystem eine Beschreibung des zu führenden Dialoges.
2. Das Dialogsystem macht entsprechend der Dialogbeschreibung eine Ausgabe an den Benutzer.

3. Der Benutzer antwortet dem Dialogsystem mit einer Eingabe.

4. Das Dialogsystem meldet dem Agenten das Ergebnis des vollständigen Dialoges.

Internes Szenario zu “Perform Dialog”

1. Der Agent übergibt dem Dialogsystem eine Beschreibung des zu führenden Dialoges.

2. Das Dialogsystem reiht den Dialogwunsch in eine Warteschlange ein. Nach einer gewissen Zeit kommt der Dialog zum Zug.

3. Das Dialogsystem gibt die Dialogbeschreibung an einen Interpreter weiter und weist diesen an, die Dialogbeschreibung zu interpretieren.

4. Der Dialogbeschreibungs-Interpreter gibt die Informationen zu jeweils *einem Wortwechsel* an einen Dialoggeräte-Manager weiter.

5. Der Dialoggeräte-Manager bereitet alle Eingabegeräte auf eine bevorstehende Eingabe vor.

6. Der Dialoggeräte-Manager gibt den Prompttext über alle ihm bekannten Ausgabegeräte an den Benutzer aus.

7. Der Benutzer antwortet dem Dialogsystem mit einer (sprachlichen) Eingabe.

8. Das vom Benutzer verwendete Dialog-Eingabegerät meldet dem Dialoggeräte-Manager die Benutzereingabe.

9. Der Dialoggeräte-Manager meldet die Eingabe dem Dialogbeschreibungs-Interpreter.

10. Das Dialogsystem meldet dem Agenten das Ergebnis des vollständigen Dialoges (sofern der Dialog vollständig abgearbeitet ist).

9.2.2 Szenario zu “Translate dialog result”

Externes Szenario zu “Translate dialog result”

1. Der Agent übergibt dem Translationssystem eine Sammlung von Übersetzungsvorschriften und das Dialogergebnis.
2. Das Translationssystem gibt dem Agenten eine, gemäß der Übersetzungsvorschrift aufgebaute, Nachricht in KQML zurück.

Internes Szenario zu “Translate dialog result”

1. Der Agent übergibt dem Translationssystem eine Sammlung von Übersetzungsvorschriften und das Dialogergebnis.
2. Der KQML-Translator sucht aus dem Dictionary die passende Übersetzungsvorschrift heraus.
3. Mit Hilfe der Übersetzungsvorschrift und dem formalen Dialogergebnis wird die KQML-Nachricht zusammengesetzt.
4. Das Translationssystem gibt dem Agenten die in KQML abgefasste Nachricht.

9.3 Interaktionsdiagramme

Die Interaktionsdiagramme[3] wurden aus den im vorangegangenen Abschnitt dargestellten Szenarien abgeleitet. Üblicherweise werden an dieser Stelle *entweder* Sequence Diagrams *oder* Collaboration Diagrams dargestellt, da sie die gleichen Informationen enthalten (vgl. [60]). Trotzdem sollen hier beide Diagramme gezeigt werden, da an den Collaboration Diagrammen der Datenfluss der, von den Nachrichten erzeugten, Rückgabewerte besonders deutlich wird.

Da die externen Szenarios sehr übersichtlich sind, werden nur die Interaktionsdiagramme der internen Szenarios gezeigt. Ein Verlust an Information für den Leser ergibt sich daraus nicht, da das externen Interaktionsdiagramm vollständig im internen Diagramm enthalten ist.

[3] Sequence Diagramm und Collaboration Diagramm

9.3.1 Interaktionsdiagramme zu “Perfom dialog”

Die Abbildungen 9.5 und 9.6 zeigen die Interaktionsdiagramme des Uses Cases Perform Dialog.

Beide Diagramme zeigen die, an der Lösung des Problems beteiligten Objekte. Die internen Objekte und ihre spezifischen Aufgaben werden nun im Einzelnen erläutert. Der Agent sendet eine Aufforderung an das Objekt *theDialogService*. Diese Aufforderung enthält die formale Beschreibung des zu führenden Dialoges, abgefasst in VoiceXML. Die Aufgabe des Dialog-Services ist es, Dialoganforderungen stellvertretend für den Benutzer entgegenzunehmen und zu schedulen. D.h., die Dialogbeschreibung wird in einer FIFO Warteschlange gepuffert, bis evtl. vorher eingetroffene Dialoge abgearbeitet sind. Am Ende der Bearbeitung gibt der Dialogservice das Dialogergebnis in Form einer Liste von Schlüssel-Wert-Paaren an den Agenten zurück.

Da bestimmte Betriebsmittel, wie etwa die Spracherkennungsoftware, nur einmal vorhanden sind, und der Benutzer nicht mehr als einen Dialog zur gleichen Zeit führen kann, macht es keinen Sinn, mehr als einen Dialog-Service zu instatiieren. Dies würde nur zu unnötiger Synchronisation führen und keinen realen Nutzen bringen.

Die eigentliche Interpretation der Dialogbeschreibung wird von *theDialogService* das Objekt *theDialogDescriptionInterpreter* delegiert. Dieses ist in der Lage, die Datei zu parsen und syntaktische Fehler im Aufbau der Dialogbeschreibung zu erkennen. Beim Einlesen der Datei wird sie von *theDialogDescriptionInterpreter* vollständig gelesen, und entsprechend den VoiceXML-Tags zu einem streng hierarchischen Baum[4] aufgebaut.

Dieser Baum, im Diagramm *theVoiceXMLTree*, wird dann vom Interpreter traversiert. Genaugenommen repräsentiert das Objekt *theVoiceXMLTree* nicht den ganzen Baum, sondern nur die Wurzel des Baumes. Diesem Wurzelelement wird die Nachricht zugestellt, den Dialog auszuführen. Der Vaterknoten delegiert diese Aufgabe dann der Reihe nach an alle seine direkten Söhne. Dies geschieht rekursiv, bis der Baum abgearbeitet ist. Jeder Knoten des Baumes weiß aufgrund des VoiceXML-Tags, das er repräsentiert, was zu tun ist. Typischerweise wird ein

[4] Document Object Model (DOM)

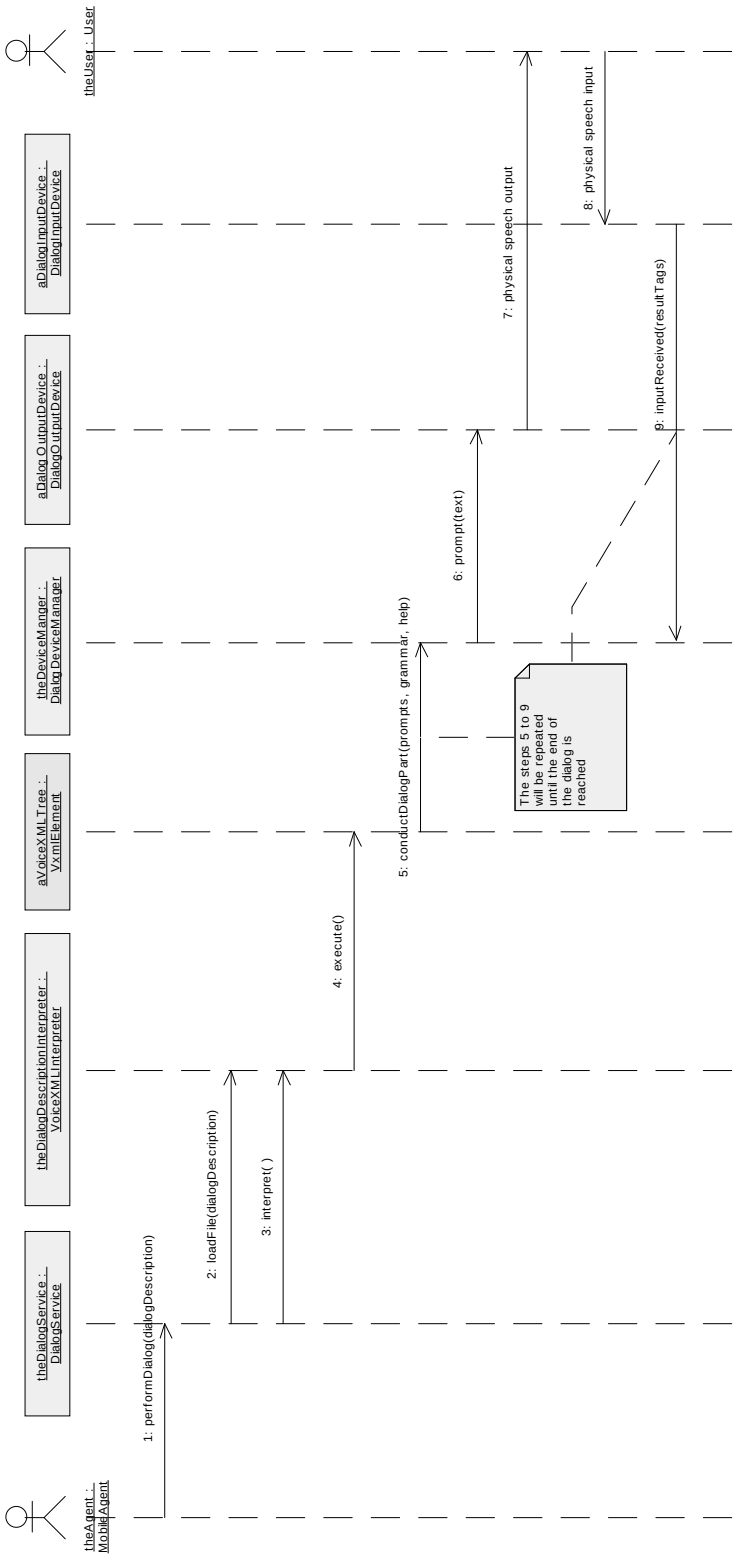


Abbildung 9.5: Sequence Diagramm zu *Perfom Dialog*

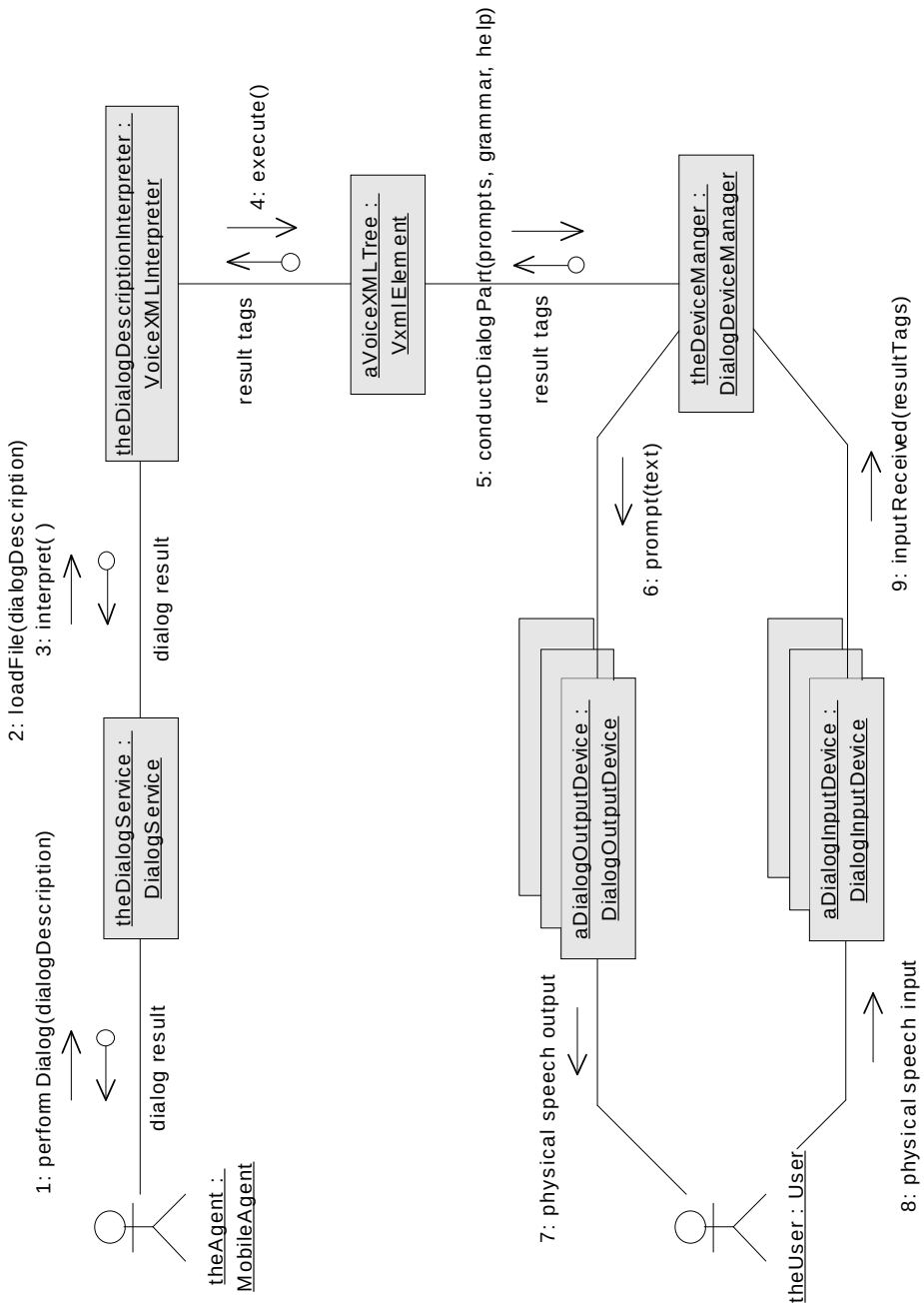


Abbildung 9.6: Colaboration Diagramm zu *Perform Dialog*

Baumknoten, der beispielsweise ein Dialogfeld[5] repräsentiert, zunächst von seinen Söhnen die relevanten Informationen zu dem Dialogteil einholen — das sind der Prompttext, die erlaubte Grammatik für die Antworten und ein optionaler Hilfetext. Dann wird dieser Knoten die Aus- und Eingabe veranlassen.

Die physikalische Ausgabe an den Benutzer nimmt der Baumkonten natürlich nicht selbst vor, da er sonst Informationen über die lokal zu Verfügung stehenden Ein- und Ausgabegeräte vorhalten müsste. Stattdessen gibt er eine Dialogaufforderung an *theDeviceManager* weiter. Dieser verwaltet je eine Liste von Eingabe- und Ausgabegeräten. Erhält er die Nachricht, einen Dialogteil auszuführen, gibt er zunächst allen Eingabegeräten die erlaubten Antworten des Benutzers bekannt. Dann fordert er alle Ausgabegeräte auf, den Prompttext auf ihre spezifische Weise auszugeben.

Ein Objekt *aDialogOutputDevice* ist in der Lage, eine physikalische Ausgabe an den Benutzer zu tätigen. Dafür erhält es den auszugebenden Prompt in Form eines Textes, den es daraufhin ausgibt.

Der Benutzer antwortet auf die Ausgabe mit einer Eingabe. Dafür benutzt er eines der zur Verfügung stehenden *DialogInputDevices*. Wenn nun ein spezifisches Objekt *aDialogInputDevice* die Eingabe des Benutzers empfängt, muss geprüft werden, ob sie der geforderten Grammatik entspricht. Ist dies der Fall, meldet *aDialogInputDevice* die empfangene Antwort an *theDeviceManager*. Die Antwort ist vom Eingabegerät nach Maßgabe der Grammatik bereits formalisiert worden. Es wird also nur das definierte Ergebnis-Tog als Antwort gemeldet.

theDeviceManager seinerseits, gibt die Information als Rückgabe auf die Dialogaufforderung an *aVoiceXMLTree* zurück.[6] Dieser benötigt diese für die weitere Abarbeitung des Baumes. Ist der Dialogbeschreibungsbaum vollständig traversiert, liefert *aVoiceXMLTree* gemäß der Dialogbeschreibung eine Liste von Schlüssel-Wert-Paaren an den Aufrufer *thDialogService* zurück. Dieser reicht die empfangene Liste an den Agenten weiter.

[5] Es handelt sich um das Tag `<field>`.

[6] Den Weg der Rückgabe zeigt ein Sequence Diagramm leider nicht so gut — dies war ein Grund für die Darstellung des Colaboration Diagramms, in dem Rückgabeparameter durch Datenflußsymbole dargestellt werden.

9.3.2 Interaktionsdiagramme zu “Translate dialog result”

Die aus den Szenarien zum Use Case “Translate dialod result” gefolgerten Interaktionsdiagramme sind in den Abbildungen 9.7 und 9.8 abgebildet. Sie werden in diesem Abschnitt besprochen.

Der Aktor *theMobileAgent* fordert *theTranslationService* auf, eine Menge von Schlüssel-Wert-Paaren in eine KQML-Nachricht zu übersetzen. Dafür übergibt ihm der Mobile Agent eine Datei mit Übersetzungsmustern, den Namen eines der Überstzungsmuster und die Schlüssel-Wert-Paare des Dialogesergebnisses.

Der Service initialisiert daraufhin einen Übersetzer, *theTranslator*, mit den Übersetzungsregeln. Das Objekt *theTranslator* ist danach in der Lage, alle Übersetzungen, die dieses Dictionary betreffen, vorzunehmen. Seine Aufgabe ist es, auf die passende Regel anhand des Regelnamens zu ermitteln, die darin enthaltenen Variablen mit Hilfe der Schlüssel-Wert-Paare zu substituieren und das Ergebnis dem Aufrufer wieder zur Verfügung zu stellen. Der Aufrufer ist in diesem Falle das Objekt *theTranslationService*, der das Ergebnis dann seinerseits an den Agenten weiterreicht. Das Parsen der Dictionary-Datei Datei übernimmt er aber nicht selbst, sondern wendet sich dafür an das Objekt *theDictionary*.

Da ein Wörterbuch typischerweise aus einer Anzahl von einzelnen Einträgen besteht, verwaltet *theDictionary* eine Sammlung von Dictionary-Einträgen. Das Objekt *aDictionaryEntry* repräsentiert dabei genau einen dieser Einträge. Er lässt sich durch seinen Namen eindeutig identifizieren. Das Objekt *theDictionary* bietet die Möglichkeit einer Suche an. Das Ergebnis dieser Suche ist eine Referenz auf *aDictionaryEntry*.

Um die Übersetzung vornehmen zu können, sucht *theTranslator* in *theDictionary* nach dem Dictionary-Eintrag mit dem Namen, den ihm der Agent über *theDialogService* genannt hat. *theDictionary* liefert *theTranslator* eine Referenz auf den passenden Eintrag *aDictionaryEntry*. Der Translator kann nun *aDictionaryEntry* direkt ansprechen, d.h. er kann von ihm die konkrete Übersetzungsregel erfragen.

Mit Hilfe der Schlüssel-Wert-Paare werden vom Übersetzer die variablen Teile der Regel substituiert und die (KQML-)Nachricht zusammengesetzt. Das Ergebnis wird über den Service an den Agenten zurückgeliefert.

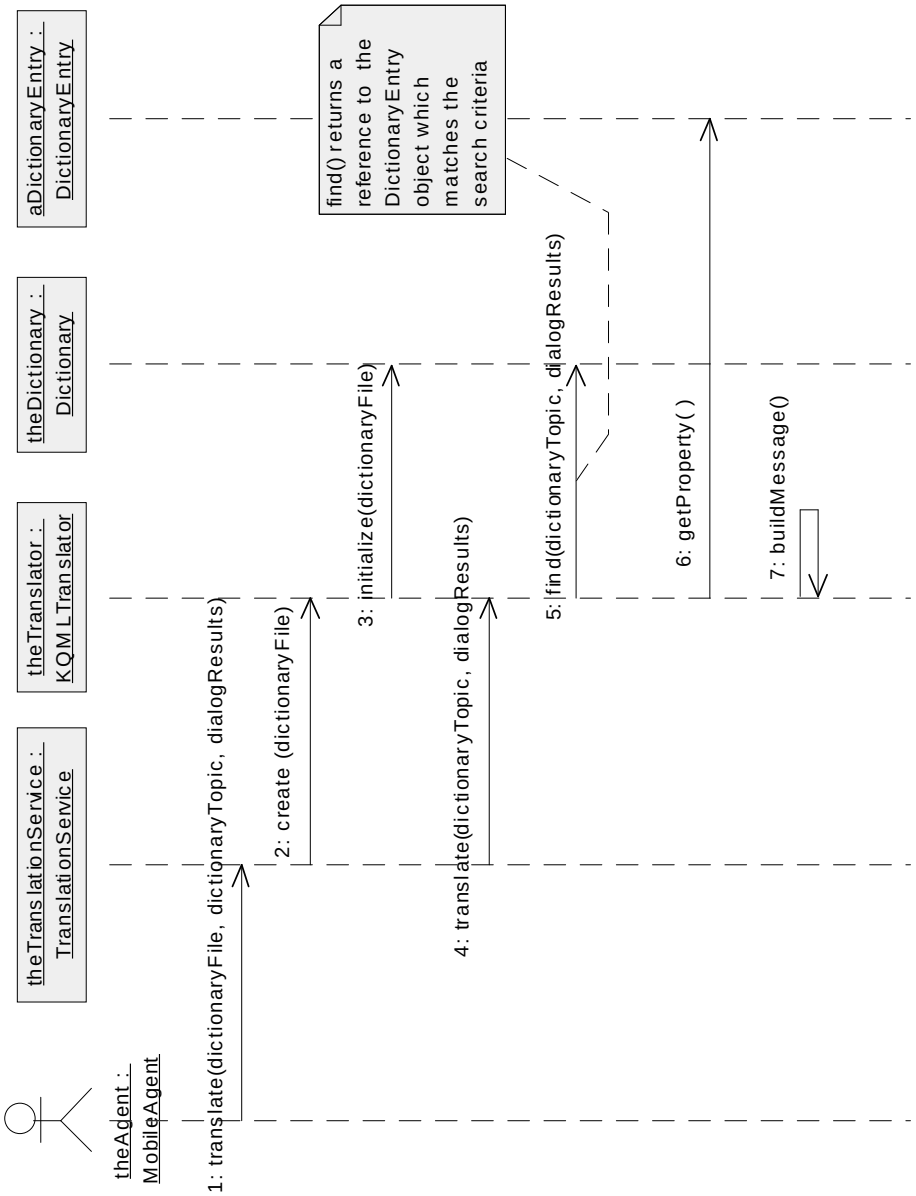


Abbildung 9.7: Sequence Diagramm zu *Translate Dialog Result*

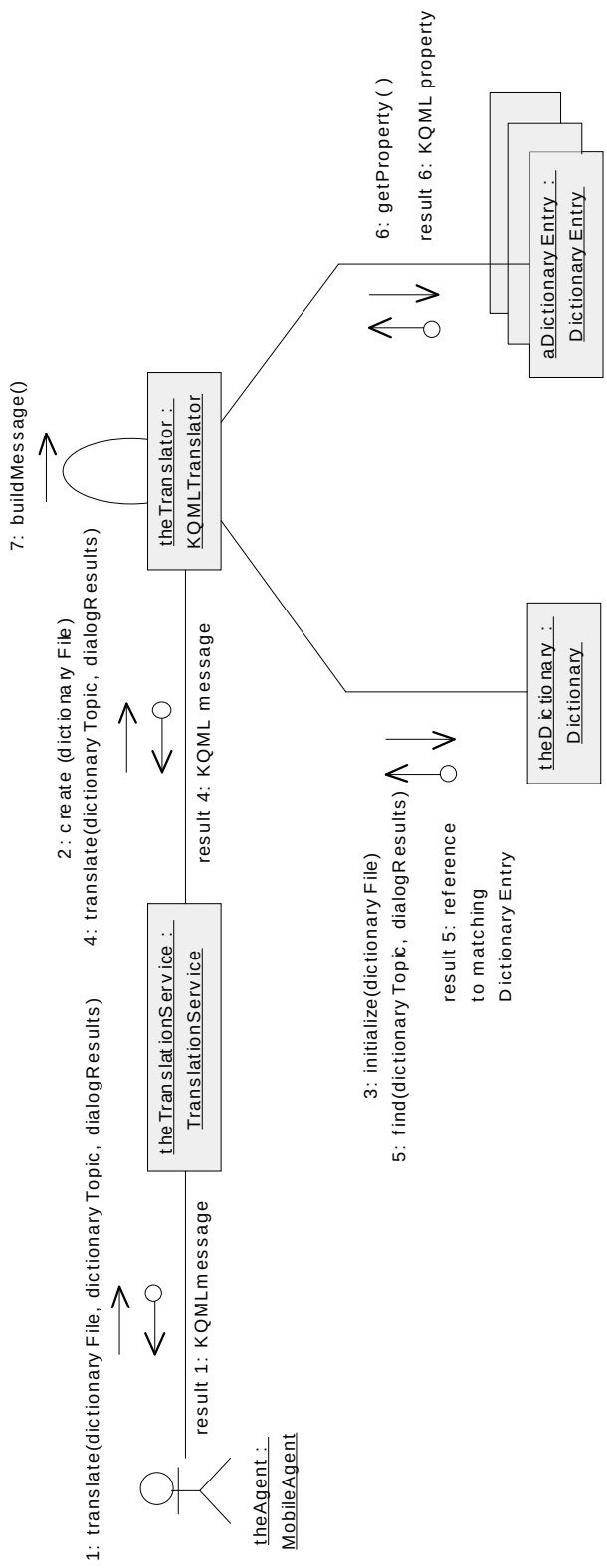


Abbildung 9.8: Colaboration Diagramm zu *Translate Dialog Result*

9.4 Bilden des Analyse-Klassendiagramms

Aus den dargestellten Interaktionsdiagrammen werden nun je ein Klassendiagramm pro Use Case abgeleitet. Da die Use Cases frei von Interaktionen untereinander sind bzw. die Interaktion nur über den Aktor *theMobileAgent* erfolgt, sind auch die entstehenden Klassendiagramme frei von Abhängigkeiten.

Die Klassendiagramme werden aus Gründen der Übersichtlichkeit ausschnittweise präsentiert und besprochen. Das Gesamtdiagramm für jeden Use Case ist Anhang A enthalten. Ebenfalls aus Gründen der Übersichtlichkeit werden die Signaturen der Methoden im Analyse-Klassendiagrammen nicht dargestellt. Es sei auf die Signaturen in den korrespondierenden Nachrichten der Sequence Diagramme und auf das Design-Klassenmodell verwiesen.

9.4.1 Analyse-Klassendiagramm zu “Perfom dialog”

Das Analyse-Klassendiagramm wird in drei Teilen erläutert, aus denen sich das vollständige Diagramm zusammensetzen lässt. Diese drei Teile sind:

- Dialogsteuerung
- Ein- und Ausgabe
- VoiceXML-Interpreter

Dialogsteuerung. Die Kernklassen des Dialogsystems für Mobile Agenten werden in Abbildung 9.9 gezeigt. Den essentiellen Teil bilden die drei Klassen DialogService$^{(Class)}$, VoiceXMLInterpreter$^{(Class)}$ und DialogDeviceManager$^{(Class)}$. Die restlichen dargestellten Klassen sind Hilfsklassen, die zur Kommunikation zwischen VoiceXMLInterpreter$^{(Class)}$ und DialogDeviceManager$^{(Class)}$ verwendet werden.

Der DialogService$^{(Class)}$ hat, betrachtet man ihn als Black-Box, die Fähigkeit, einen Dialog entgegenzunehmen und die Ausführung des Dialoges zu beginnen[7]. Die

[7] Um zur Ausführung zu kommen, muss Dialog natürlich im internen Scheduling zum Zuge kommen.

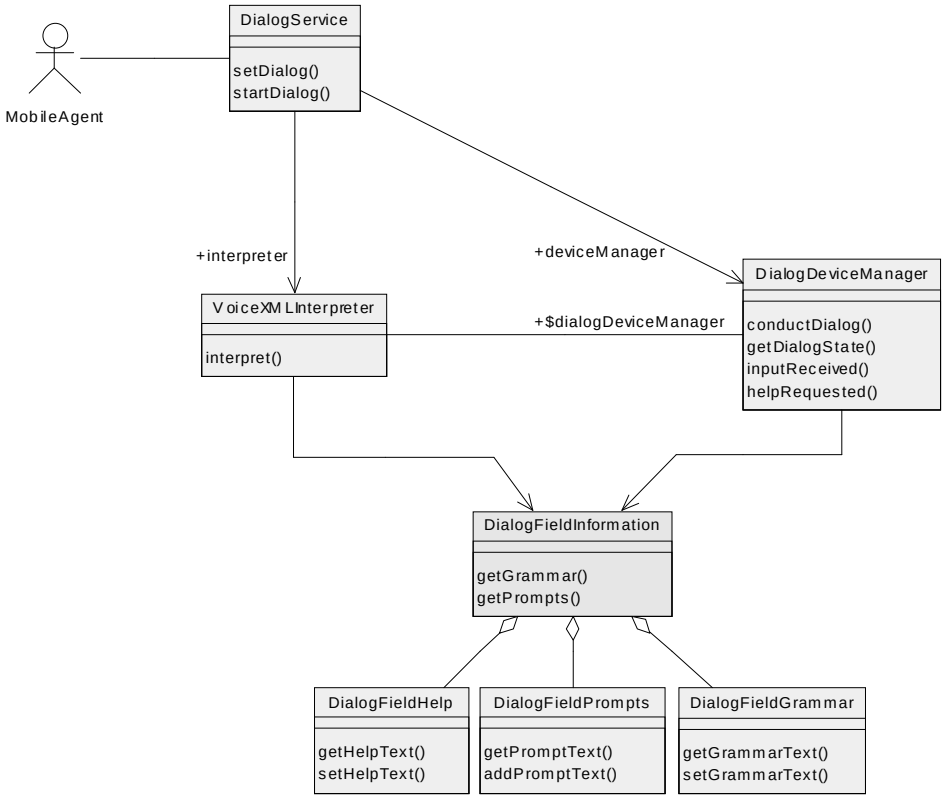


Abbildung 9.9: Kernkomponenten des Dialogsystems

Sichtweise des Agenten entspricht damit der in Abschnitt 6.1 geforderten Sicht des Agenten auf das Dialogsystem (vgl. Abbildung 6.1).

Der Service kommuniziert (unidirektional) mit dem VoiceXMLInterpreter(Class) und dem DialogDeviceManager(Class). Beide werden vom DialogService(Class) verwaltet. Dies stellt sicher, dass der DialogService(Class) die Kontrolle über die beiden Objekte hat. Eine Instantiierung des DialogDeviceMangers(Class) durch den VoiceXMLInterpreter(Class) ist *nicht* möglich, da dieser nur für die Dauer eines Dialoges erhalten bleibt. Der DialogManager(Class) hingegen ist dialogpersistent; es handelt es sich für jeden Dialog um immer die gleiche Instanz von DialogService(Class).

Gelangt der VoiceXMLInterpreter(Class) beim Interpretieren der Dialogbeschreibung an eine Stelle, an der ein Dialogfeld ausgeführt werden soll, so sendet er dem DialogDeviceManager(Class) die Nachricht *conductDialog*. Diese veranlasst den DialogDeviceManager(Class), genau einen Ausgabe- und einen Eingabevorgang durchzuführen.

Die Klassen DialogDeviceManager(Class) und VoiceXMLInterpreter(Class) verwenden bei Übergabe der umfangreichen Daten, die zu einem Kommunikationsakt gehören, die Klasse DialogFieldInfo(Class). Sie stellt eine Komposition aus den den Klassen DialogFieldPrompts(Class), DialogFieldGrammar(Class) und DialogFieldHelp(Class) dar. Alle drei Klassen dienen nur der Datenhaltung.

Ein Objekt vom Typ DialogFieldPrompts(Class) enthält *alle* inkrementalen Prompts des aktuellen Dialogfeldes. DialogFieldGrammar(Class)-Objekte repräsentieren die in diesem Feld gültige Antwortgrammatik, und DialogFieldHelp(Class) den Text der Kontexthilfe zum aktuellen Dialogfeld. Alle diese Klassen verfügen über die, für Datenhaltungsklassen typischen, *get*- und *set*-Methoden.

Ein- und Ausgabe. Von der Klasse DialogDeviceManager(Class) wird genau eine Instanz angelegt. Diese übernimmt die Kontrolle über alle Ein- und Ausgabegeräte. Welche Geräte verwendet werden sollen, erfährt der DialogDeviceManager(Class) bei seiner Instantiierung durch eine Konfigurationsdatei, in die die Namen der vom Benutzer gewünschten Ein- und Ausgabegeräte eingetragen sind. Diese werden dann vom DialogDeviceManager(Class) in-

stantiiert und verwaltet. Daurch hat er die alleinige Kontrolle über die Geräte. Ein Beispiel einer solchen Konfigurationsdatei ist in Abbildung 9.10 dargestellt. Der DialogDeviceManager(Class) bekommt in diesem Beispiel mitgeteilt, dass er über ein Eingabegerät (DialogRecognizer(Class)) und zwei Ausgabegeräte (DialogSynthesizer(Class) und DialogFramePrompter(Class)) verfügt.

```
# Input devices to use
device.input.1  = DialogRecognizer

# Output devices to use
device.output.1 = DialogSynthesizer
device.output.2 = DialogFramePrompter
```

Abbildung 9.10: Konfigurationsdatei für den *DialogDeviceManager*

Abbildung 9.11 zeigt die Beziehungen des DialogDeviceManagers(Class) zu den Geräten. Aus Sicht des DialogDeviceManagers(Class) kommuniziert er stets mit Geräten vom Typ DialogInputDevice(Interface) bzw. DialogOutputDevice(Interface).

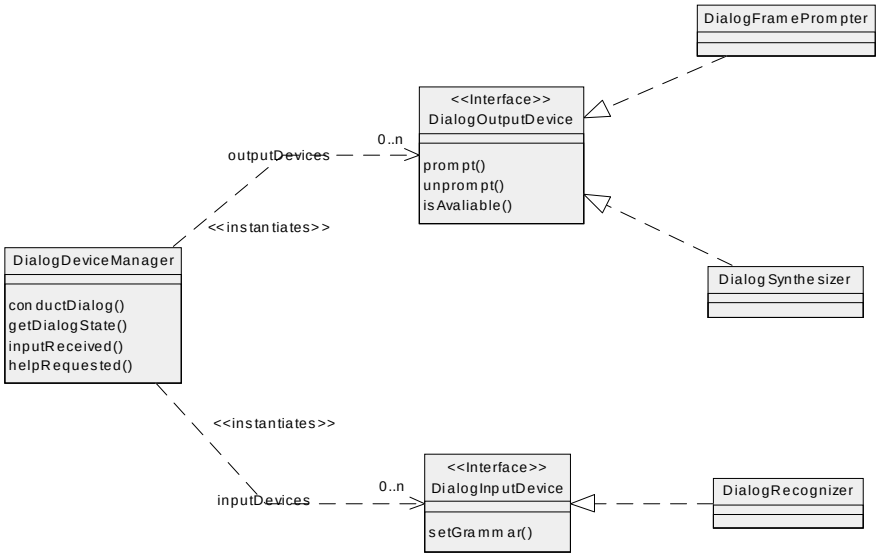


Abbildung 9.11: Ein- und Ausgabekomponenten des Dialogsystems

Diese Schnittstellen[8] werden von den Gerätetreibern implementiert und erlauben dem Gerätemanger einen einheitlichen Zugriff auf die verschiedenen Geräte. Diese Architektur hat den Vorteil, dass die Anzahl der möglichen Geräte nicht beschränkt ist. Mit anderen Worten: die Einführung der Schnittstellen DialogInputDevice$^{(Interface)}$ und DialogOutputDevice$^{(Interface)}$ realisiert die in Abschnitt 6.1 geforderte Geräteunabhängigkeit. Soll das System über neue Geräte mit dem Benutzer kommunizieren, müssen die Treiberklassen der Geräte nur die entsprechende Schnittstelle implementieren.

Die Assoziation zwischen dem DialogDeviceManager$^{(Class)}$ und den Geräten ist eine 1:n-Beziehung, d.h. *der* Gerätemanager hat Referenzen auf viele Ein- und Ausgabegeräte. Die Realisierung wird deshalb später in einem Feld[9] erfolgen.

VoiceXML-Interpreter. Nun soll als dritter und letzter Ausschnitt des Analyse-Klassendiagramms der VoiceXMLInterpreter$^{(Class)}$ betrachtet werden (siehe Abbildung 9.12). Die Aufgabe des VoiceXMLInterpreters$^{(Class)}$ ist das Parsen und Interpretieren des Dialogbeschreibungsdokumentes.

Die VoiceXML-Datei wird zu Beginn vollständig geparst und in einer Datenstruktur abgelegt. Dies stellt *vor* der Ausführung sicher, dass die Dialogbeschreibung syntaktisch korrekt ist und erleichtert später die Implementation.

Die verwendete Datenstruktur ist baumartig organisiert und entspricht einem *Document Object Model* (DOM). Der gesamte DOM-Baum der Dialogbeschreibung besteht aus Instanzen der Klasse VxmlElement$^{(Class)}$; dies drückt sich im Klassendiagramm in der reflexiven Referenz *parent* und das Attribut *children* aus. Jeder Knoten des Baumes steht für ein VoiceXML-Tag in der Dialogbeschreibung.[10] Der VoiceXMLInterpreter$^{(Class)}$ hält nur eine Referenz auf die Wurzel des Baumes. Soll der Dialog ausgeführt werden, wird der Baum einfach traversiert. Dazu ruft der Interpreter die Methode *execute* des Wurzelelementes auf. Die Abarbeitung des Baumes erfolgt rekursiv. Ein Vaterknoten, dessen *execute*-Methode aufgerufen

[8] Eigentlich sollte der Aspekt der Schnittstellen erst in der Designphase betrachtet werden. An dieser Stelle handelt es sich bei dem Schnittstellenkonzept jedoch um eine essentielle Lösungsstrategie. Daher gehörte sie nach Auffassung des Autors zur Analyse.

[9] Bei der Implementierung wurde dafür ein Objekt vom Typ Vector$^{(JDK)}$ verwendet.

[10] Vgl. dazu das Design-Pattern in [76]

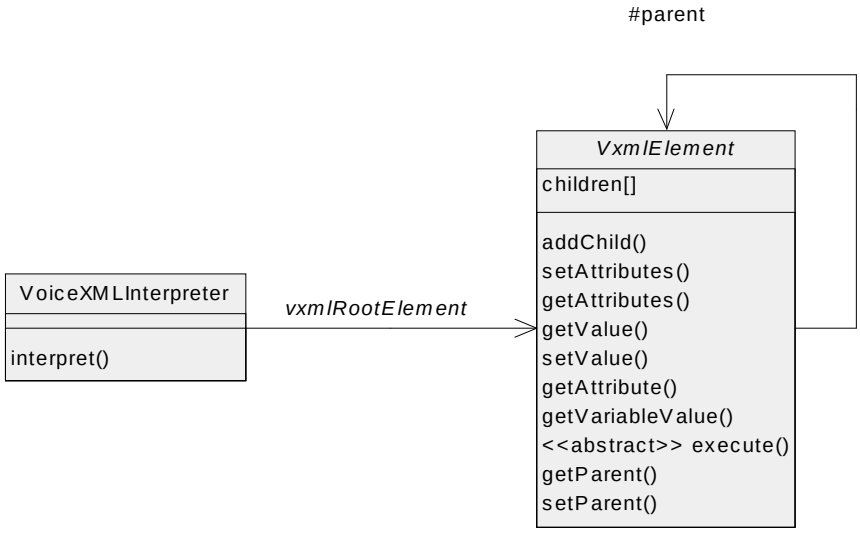


Abbildung 9.12: Dialogbeschreibungs-Interpreter

worden, ruft zunächst die *execute*-Methoden seiner direkten Söhne auf, nimmt deren Ergebnisse entgegen und führt dann seine eigenen Anweisungen aus. Dieses Vorgehen resultiert in einer Postorder-Abarbeitung [56] des Baumes.

Da jeder Knoten “weiß”, für welches VoiceXML-Tag er steht, kann er bei der Ausführung entsprechend dem Tag reagieren. Dies hat jedoch zwei entscheidende Nachteile:

1. Die Klasse VxmlElement$^{(Class)}$ muss das Verhalten *jedes* VoiceXML-Tags in sich implementieren. Dadurch wird die Klasse sehr groß, unübersichtlich und schwer wartbar.

2. Sollen neue VoiceXML-Tags implementiert werden[11], muss der interne Aufbau und die Funktionsweise der Klasse VxmlElement$^{(Class)}$ bekannt sein. Die notwendigen Implementierungsarbeiten finden wieder nur in dieser Klasse statt.

Da diese Nachteile nicht hinnehmbar sind, wird die Architektur dahingehend ver-

[11] Aus Zeitgründen war eine *vollständige* Implementation der VoiceXML-Spezifikation nicht möglich. Daher wird eine möglichst einfach erweiterbare Architektur angestrebt.

ändert, dass VxmlElement(Class) nur die *Basisklasse* eines Sets vom VoiceXML-Klassen ist. Für jedes VoiceXML-Tag `<abc>` wird eine eigene Klasse Abc(Class) vorgesehen, die ihre Basisfähigkeiten von VxmlElement(Class) ererbt und die tag-spezifische *execute*-Methode selbst implementieren muss. Diese Architektur erlaubt eine einfache Erweiterbarkeit und eine logische Aufteilung der Gesamtkomplexität von VoiceXML auf die Einzelklassen. Die beschriebene Architektur wird in Abbildung 9.13 nochmal grafisch veranschaulicht.

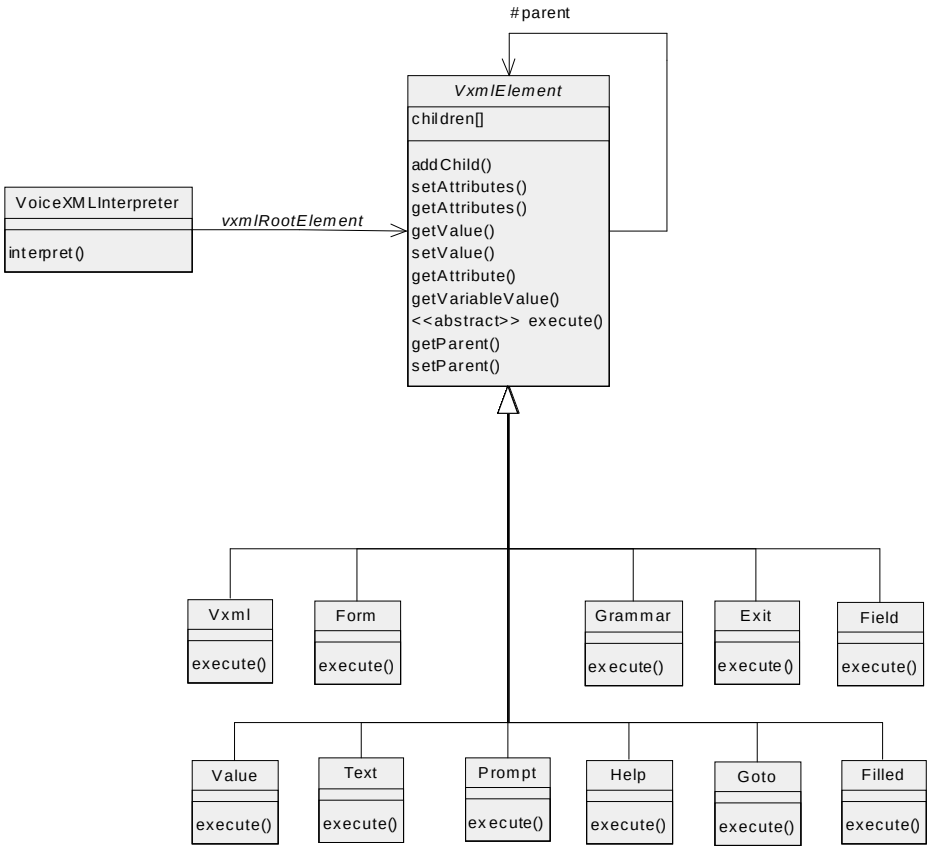


Abbildung 9.13: Dialogbeschreibungs-Interpreter

Die anderen, bisher nicht beschriebenen Methoden von VxmlElement(Class), sind anhand ihrer Bezeichner selbsterklärend. Es handelt sich um Methoden, die:

- Manipulationen erlauben, die typisch für eine baumartige Datenstruktur sind (*addChild, getParent, setParent*),

- einen Zugriff auf Attribute des VoiceXML-Tags bieten (*getAttribute, setAttribute, getAttributes, getValue, setValue*) oder
- Werte von VoiceXML-Variablen[12] lesen und setzen (*getVariable, setVariable*).

Damit ist die Beschreibung der Einzelkomponenten des Analyseklassendiagramms für den Use Case *Perform Dialog* abgeschlossen. Das vollständige Analyse-Klassendiagramm, in dem das Kernsystem, die Ein- und Ausgabe, sowie der Interpreter gemeinsam dargestellt sind, ist in Anhang A zu finden.

9.4.2 Analyse-Klassendiagramm zu "Translate dialog result"

In diesem Abschnitt wird das Analyse-Klassendiagramm des Use Cases *Perform Dialog* besprochen. Es braucht wegen der wesentlich geringeren Komplexität nicht in Abschnitte zerlegt zu werden. In Abbildung 9.14 ist das vollständige Analyse-Klassendiagramm dargestellt.[13]

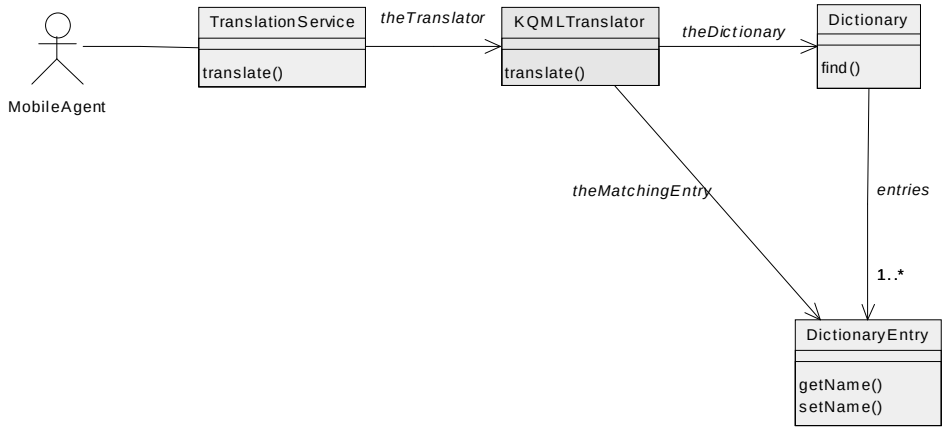


Abbildung 9.14: Analyse-Klassendiagramm zu *Translate Dialog Result*

Die Klassenrelationen gehen in diesem Beispiel besonders deutlich aus dem zugehörigen Interaktionsdiagramm hervor. Dadurch konnte die Entwicklung dieses

[12] VoiceXML kennt Zugriffsbereiche (Scopes) für Variablen.

[13] Dieses Diagramm ist im Anhang A nochmal in vergrößerter Form enthalten.

Klassendiagramms “straight-forward” direkt aus dem Sequence Diagramm erfolgen. Nennenswerte Schwierigkeiten traten nicht auf, da die grundsätzliche Vorgehensweise der Übersetzung schon beim Entwurf des Sequence Diagramms festgelegt worden ist.

Der Translator benutzt des Dictionary(Class) zur Organisation der einzelnen Übersetzungsregeln. Das Dictionary(Class) verwaltet eine Menge von DictionaryEntries(Class) und kann einzelne Regeln anhand ihres Namens ausfindig machen. Ein DictionaryEntry(Class) enthält dann die eigentliche Übersetzungsregel.

Gut zu erkennen ist der, im Szenario und im Sequence Diagramm dargestellte Delegationsmechanismus, durch den der Übersetzungsvorgang realisiert wurde: Der KQMLTranslator(Class) fordert das Dictionary(Class) auf, einen namentlich bekannten Eintrag herauszusuchen. Auf diesen erhält der KQMLTranslator(Class) als Antwort auf die Nachricht dann eine Referenz (*theMatchingEntry*). Der KQMLTranslator(Class) hat jetzt die Möglichkeit, die Übersetungsregel direkt zu verwenden und die Übersetzung durchzuführen. Die Klasse TranslationService(Class) dient dem Agenten als Schnittstelle für den KQMLTranslator(Class). Im Design hat sich bei einer genaueren Betrachtung dieser Stelle herausgestellt, dass sich der TranslationService(Class) als Interface realisieren lässt (dazu mehr in Abschnitt 10.1).

Damit ist die Analyse des Use Cases *Translate dialog result* abgeschlossen. Das Analyse-Klassendiagramm ist im Anhang A in vergrößerter Form abgebildet.

9.5 Zustandsdiagramme

Die eigentliche Dynamik des Systems steckt in der Dialogbeschreibung des Agenten. Man kann den Dialog, im dem laut VoiceXML-Spezifikation [83] Iterationen und Fallunterscheidungen möglich sind, als deterministischen endlichen Automaten auffassen. Dann hat das hier vorgestellte Dialogsystem nur noch die Aufgabe, diesen Dialog *sequentiell* abzuarbeiten.

Dynamisches Verhalten innerhalb des entwickelten Systems zeigt nur die Klasse DialogDeviceManager(Class). Sie ist die Instanz, über die sowohl die Aus- als auch die Eingabe erfolgt; daher reagiert sie unterschiedlich, je nachdem ob z.B. die

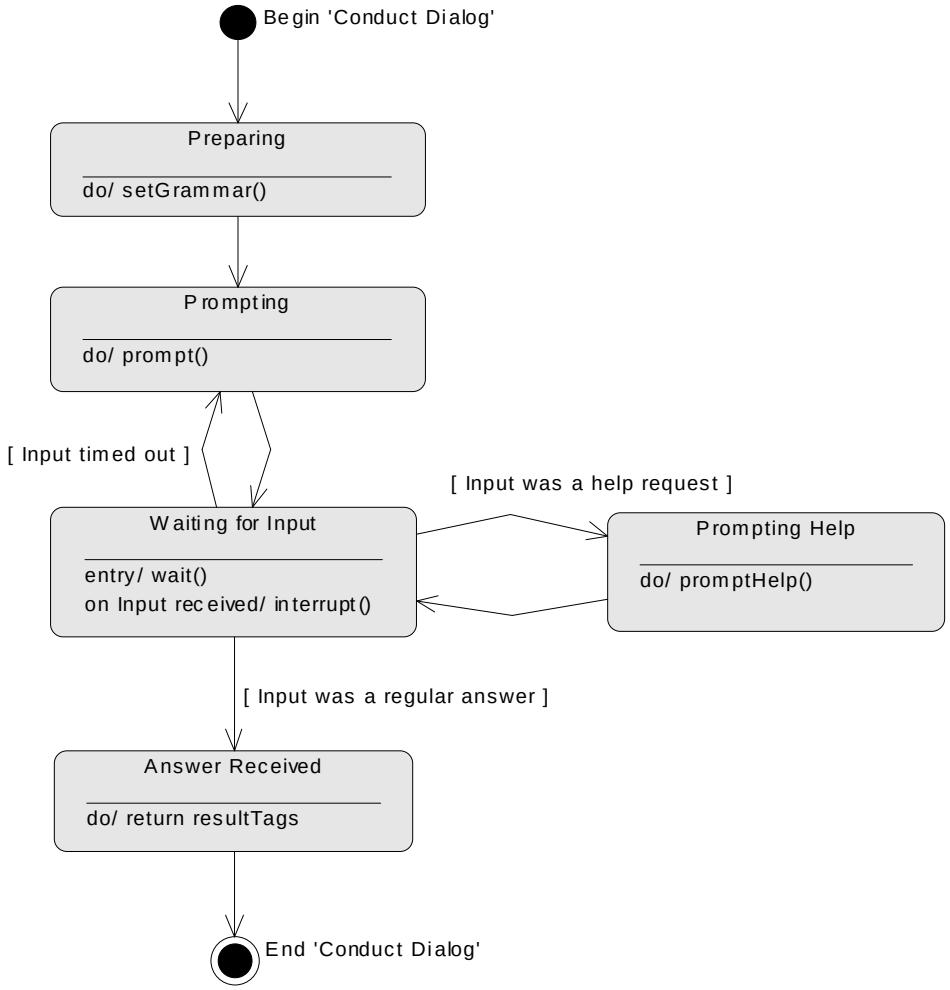


Abbildung 9.15: Zustandsdiagramm der Klasse *DialogDeviceManager*

Ausgabe nach einem Timeout wiederholt werden muss oder ob der Benutzer die kontextsensitive Hilfe angefordert hat.

Dazu kommt noch der Umstand, dass die Antwort des Benutzers die einzige asynchrone Interaktion innerhalb des Systems ist, denn die Antwort des Benutzers ist nicht zeitlich an das System gebunden. Er antwortet u.U. gar nicht oder erst mit langer Verzögerung. Daher wird für die Klasse DialogDeviceManager(Class) ein Zustandsdiagramm für die Aus-/Eingabe-Interaktion mit dem Benutzer modelliert (siehe Abbildung 9.15).

Soll ein Dialog[14] mit dem Benutzer erfolgen, so werden zunächst alle Eingabegeräte auf die neue Grammatik vorbereitet (Zustand *Preparing*). Dies geschieht durch das Versenden der Nachricht *setGrammar* an jedes DialogInputDevice(Interface).

Danach kann die Ausgabe vorgenommen werden (Zustand *Prompting*). Dazu wird allen DialogOutputDevice(Interface) die Nachricht *prompt* zugestellt.

Bis hier ist der Ablauf noch vollkommen sequentiell, doch nun liegt es am Benutzer, ob, wann und wie er reagiert; die Eingabe ist also vollkommen asynchron.[15] Der DialogDeviceManager(Class) wartet auf eine Antwort (Zustand *Waiting for Input*). Dieses Warten geschieht zeitüberwacht. Kommt innerhalb einer gewissen Zeitspanne keine Reaktion vom Benutzer, wird erneut eine Ausgabe gemacht. Dabei wird typischerweise, d.h. je nach Design des Dialoges, der Ausgabetext durch inkrementale Prompts präzisiert.

Antwortet der Benutzer dem System, ist der Dialog beendet und die Ergebnis-Tags, die in der VoiceXML-Datei definiert wurden, werden an die aufrufende Klasse, den VoiceXMLInterpreter(Class), zurückgeliefert (Zustand *Answer Received*).

Bittet der Benutzer statt einer Antwort jedoch um Hilfe, wird Hilfetext dieses Dialogteils (Zustand *Prompting Help*). Dies geschieht wie bei der normalen Ausgabe über das Versenden der Nachricht *prompt* an alle DialogOutputDevices(Interface). Danach geht das System wieder in den Zustand *Waiting for Input*.

[14] Aus Sicht des DialogDeviceMangers besteht ein Dialog aus genau einer Ausgabe und einer Eingabe

[15] Wenn der Benutzer mit Absicht nicht reagiert, könnte er mit diesem “Denial-of-Service-Verhalten” den Agenten blockieren. In Anlehnung an einen *malicious host* könnte man dann von einem *malicious user* spechen.

Im Normalfall führt der Ablauf des Zustandsdiagramms sequentiell vom Startzustand über *Preparing*, *Prompting*, *Waiting for Input*, *Answer Received* in den Stopzustand.

An diser Stelle ist das Bilden des Analysemodells abgeschlossen. Nun muss das logische Lösungsmodell in der anschließenden Designphase in Richtung einer späteren Implementierung verfeinert werden.

Kapitel 10

Bilden des Designmodells

In diesem Kapitel werden die Entwicklungsentscheidungen dargestellt, die während der Designphase getroffen wurden. Da das Analysemodell die spätere Systemstruktur schon sehr detailliert wiedergibt, sind nur noch geringe Änderungen erforderlich. Die wichtigsten Aspekte sind die Anbindung an externe Klassenbibliotheken und die Integration des Dialogsystems in SeMoA .

10.1 Schnittstellenklassen

Schnittstellenklassen zu "Perform Dialog". Die beiden wichtigsten Schnittstellen dieses Teilsystems, DialogInputDevice$^{(Interface)}$ und DialogOutputDevice$^{(Interface)}$, wurden bereits in der Analysephase gefunden und besprochen (siehe 9.4.1). Weitere Schnittstellen bieten sich im Klassendiagramm nicht an.

Schnittstellenklassen zu "Translate Dialog Result". Bei der Untersuchung des Analyse-Klassendiagramms (siehe Abbildung 9.14) fällt auf, dass der TranslationService$^{(Class)}$ als Schnittstelle zwischen Agent und dem KQMLTranslator$^{(Class)}$ fungiert. Gleichzeitig erfüllt der KQMLTranslator$^{(Class)}$ mit der Übersetzung in *eine bestimmte* eine Agentenkommunikationssprache eine sehr spezifische Aufgabe. Es liegt also nahe, den TranslationService$^{(Class)}$ als allgemeinere Klassifikation des KQMLTranslators$^{(Class)}$ anzusehen.

Daher wurde dem *TranslatorService* das Stereotyp eines Interfaces gegeben. Der TranslatorService(Interface) ist dann eine Schnittstelle, über die spezifische Übersetzer[1] angesprochen werden können. Der KQMLTranslator(Class) implementiert dieses Interface und ist somit die konkrete Realisierung eines Übersetzers.

Die Alternative zur Einführung des Interfaces ist eine Vererbungsrelation zwischen beiden Klassen. *TranslationService* wäre dann die Superklasse von *KQML-Translator*. Dies wurde bewusst vermieden, da bei der Anbindung des entwickelten Dialogsystems an das SeMoA -Projekt bereits eine Vererbungsableitung von PluginService(Class) vorgenommen werden muss. Eine zweite Vererbung würde dann zu einer ungewünschten Mehrfachvererbung[2] führen.

10.2 Abstrakte Klassen

Die einzige Klasse in beiden Analysemodellen, die für eine *abstract*-Deklaration in Betracht kommt, ist die Klasse VxmlElement(Class).

VxmlElement(Class) dient, wie in Abschnitt 9.4.1 dargelegt, als Basisklasse für Klassen, die die Funktionalität eines VoiceXML-Tags implementieren. VxmlElement(Class) selbst wird *nie* instatiiert. Diese Klasse stellt nur die Basisfunktionalität zur Verfügung, d.h. sie implementiert die *get*- und *set*-Methoden und die Baumfunktionalitäten (*addChild*, *getParent*, *setParent*) des Document Objekt Models (DOM). Die Methode *execute* kann von VxmlElement(Class) *nicht* implementiert werden, weil jedes abgeleitete VoiceXML-Tag ein anderes Verhalten zeigt. Ein Default-Verhalten gibt es nicht. Dies alles sind Indikatoren, die dafür sprechen, die Klasse VxmlElement(Class) abstrakt zu machen.

Soll der VoiceXML-Interpreter später um ein neues Tag erweitert werden, muss eine Klasse mit dem Namen des Tags von VxmlElement(Class) abgeleitet werden. Um bei der späteren Erweiterung, den Entwickler zu zwingen, die Methode *execute* zu implementieren, wird die Methode *VxmlElement.execute* und damit die Klasse VxmlElement(Class) als abstrakt deklariert.

[1] z.B. für KQML, FIPA ACL, etc.

[2] Zudem wird die Implementation in Java™ erfolgen und dort sind Mehrfachvererbungen nicht möglich.

10.3 Integration externer Klassenbibliotheken

Dieser Abschnitt gibt eine Orientierung darüber, welche externen Klassenbibliotheken bei der Realisierung des Projektes verwendet wurden. Die konkrete Einbindung externer Klassen wird im Zusammenhang mit dem Design-Klassendiagramm in Abschnitt 10.5 besprochen.

Es wurde versucht, Detailprobleme, für die bereits Klassenbibliotheken existieren, mit Hilfe externer Bibliotheken zu lösen. Die externen Klassen befinden sich allesamt an der Peripherie des Systems. Dies ist eine logische Folge eines analysebetonten Software Engineerings.

Die Klassenbibliotheken helfen

- bei der Integration des Systems in seine Umgebung,
- dem Ansprechen von Peripherie für die Ein- und Ausgabe und
- beim Parsen der XML-Dialogbeschreibung.

Dem Einsatz der Klassenbibliotheken ging eine Evaluation infrage kommender Produkte voraus, auf die aber hier nicht näher eingegangen wird. Es sei noch angemerkt, dass ausschliesslich nicht-kommerzielle Bibliotheken verwendet werden. Dies war eine notwendige Randbedingung, da sich das Forschungsprojekt SeMoA nicht von kommerziellen Lizenzen Dritter abhängig machen möchte.

Im einzelnen wurden folgende Bibliotheken eingesetzt:

SeMoA ist die Klassenbibliothek der SeMoA Projektes in desssen Rahmen das hier entwickelte System eingesetzt wird. Sie bietet die Möglichkeit, einen in SeMoA integrierbaren Service zu entwickeln. Auf die Integration des Services wird im nächsten Abschnitt eingegangen.

Java™Speech API ist eine plattformunabhängige API von Sun, die Spracherkennung (*Recognition*) und Spracherzeugung (*Synthesis*) aus Java™-Anwendungen heraus erlaubt. Darüberhinaus leistet die API auch den Vergleich der Spracheingabe mit einer regulären Grammatik. Eine detaillierte

Beschreibung der Java^{TM}Speech API findet man in [74]. Sie wurde in diesem Projekt zur Realisierung von Spracheingabe und Sprachausgabe in den Dialoggeräten verwendet.

Xerces beinhaltet die Implementierung eines XML-Parsers. Er ist eine Entwicklung der Apache XML Group und hält sich an die XML-Standards des W3C-Konsortiums. Der DOM-Parser von Xerces kommt im VoiceXMLInterpreter zum Bilden des DOM-Baumes aus der Dialogbeschreibungs-Datei zum Einsatz. Aus diesem wird dann der Baum aus VxmlElement-Knoten generiert.

10.4 Integration in die Architektur von SeMoA

Das zu entwickelnde System muss sich in die Systemarchitektur von SeMoA einfügen. Um die Funktionalität eines Agentenservers zu erweitern, sieht SeMoA , wie in Abschnitt 5.2 besprochen, einen Mechanismus vor, der auf Diensten basiert. Aus

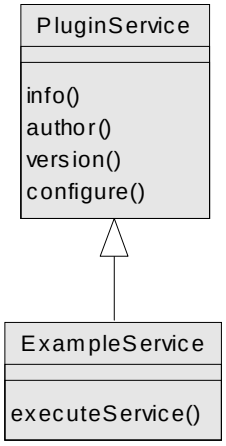


Abbildung 10.1: UML Darstellung eines Plugin-Service für SeMoA

Sicht des Software Engineerings muss ein neuer Service, wie in Abbildung 10.1 gezeigt, von der Klasse *PluginService* abgeleitet sein. Für den Software-Lifecycle des vorliegenden Systems bedeutet dies, dass am Ende des Grobdesigns das entwickelte System an geeigneter Stelle von der Klasse PluginService$^{(Class)}$ abgeleitet werden muss, um in SeMoA integriert werden zu können. Dies beeinträchtigt die

Analysephase und den ersten Teil der Designphase nicht. Das System kann so entwickelt werden, als müsse es nicht integriert werden.

10.5 Design-Klassendiagramm

In den beiden folgenden Abschnitten werden die Design-Klassendiagramme der beiden Teilsysteme *Perform Dialog* und *Translate Dialog Result* dargestellt und die in der Designphase vorgenommenen Entscheidungen erläutert.

Grundsätzlich ist zu den Diagrammen anzumerken, dass Referenzen auf Klassen, im Klassensymbol *nicht* als Attribut aufgeführt sind. Das Vorhandensein einer unidirektionalen Referenz impliziert ein entsprechendes Attribut. Dies erleichtert die Übersichtlichkeit des Diagramms und ist konform mit der UML Notation [27].

Der auffälligste Unterschied zwischen den Klassendiagrammen aus Design und Analyse ist die Darstellung der Zugriffsklassen für Methoden. Es wurde vollständig mit *private*- und *public*-Sichtbarkeit gearbeitet. Sie werden UML-konform mit '+' für *public* und '–' für *private* gekennzeichnet.

In beiden Diagrammen ist die Anbindung an SeMoA in Form einer Ableitung von PluginService$^{(Class)}$ eingetragen.

10.5.1 Design-Klassendiagramm zu "Perfom dialog"

Die Abbildung 10.2 zeigt das Design-Klasssendiagramm von *Perform Dialog*. Es ist in vergrößerter Form nocheinmal in Anhang B enthalten.

Neu hinzugekommen sind Geräteklassen, die die Interfaces DialogInputDevice$^{(Interface)}$ und DialogOutputDevice$^{(Interface)}$ implementieren. Die drei Klassen DialogRecognizer$^{(Class)}$, DialogSynthesizer$^{(Class)}$ und DialogFramePrompter$^{(Class)}$ wurden im Rahmen des Demonstrators entwickelt, um die Interaktion mit dem Benutzer im Rahmen des Demonstrators zu realisieren.

Die Klasse DialogRecognizer$^{(Class)}$ ist ein DialogInputDevice$^{(Interface)}$, das dem Benutzer die Spracheingabe ermöglicht. Die Klasse Recognizer$^{(Class)}$ der Java^{TM}Speech API führt eine Spracherkennung auf der Basis einer regulären Grammatik durch. Das Ergebnis ist eine Anzahl von Result-Tags,

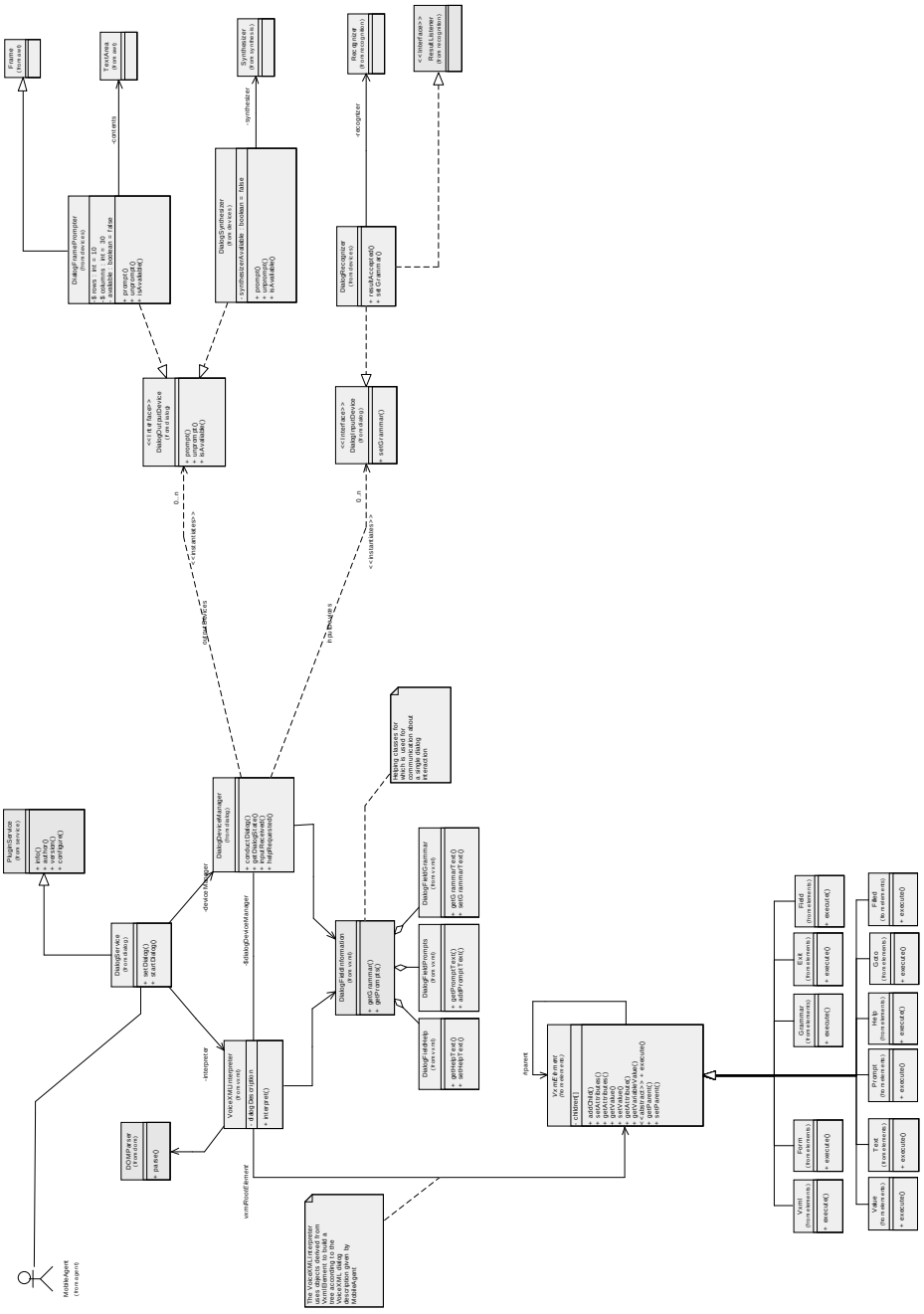


Abbildung 10.2: Design-Klassendiagramm zu *Perform Dialog*

die in der Grammatik definiert sind. Um diese Tags zu erhalten, meldet sich der DialogRecognizer(Class) als Listener bei Recognizer(Class) an. Dafür muss DialogRecognizer(Class) die Schnittstelle ResultListener(Interface) des Java™Speech APIs implementieren. Beide Speech-Klassen sind im Package `javax.speech.recognition` enthalten.

Die Klasse DialogSynthesizer(Class) hat als DialogOutputDevice(Interface) die Fähigkeit, einen Prompt in gesprochener Form an den Benutzer auszugeben. Dafür werden die Text-to-Speech-Fähigkeiten der Klasse Synthesizer(Class) verwendet. Synthesizer(Class) ist ebenfalls ein Bestandteil der von Sun definierten Java™Speech API (Package `javax.speech.synthesizer`).

Der DialogFramePrompter(Class) realisiert seine Ausgabe, indem er auf der Benutzeroberfläche ein Fenster öffnet und den Prompt in Form eines Text in einer TextArea(Class) ausgibt. DialogFramePrompter(Class) ist von der Klasse Frame(Class) abgeleitet und erbt daher vollständig die Standard-Funktionalitäten eines Fensters. TextArea(Class) und Frame(Class) sind Bestandteil des Java™Packages `java.awt`.

Die Klasse VoiceXMLInterpreter(Class) enthält das private Attribut *dialogDescription*. Sie ist eine Referenz auf die Dialogbeschreibung in Form eines DOM-Baumes, der von einem DOMParser(Class) erzeugt wird. Der DOMParser(Class) gehört zum Xerces-Projekt der Apache XML Group (Package `org.apache.xerces.dom`). Das Attribut *dialogDescription* ist das Wurzelelement des Document Object Models der Dialogbeschreibung. Aus ihm wird der Baum aus VxmlElementen(Class) generiert.

10.5.2 Design-Klassendiagramm zu “Translate dialog result”

Die Abbildung 10.3 zeigt das Design-Klassendiagramm von *Perform Dialog*. Es ist in vergrößerter Form nochmal in Anhang B enthalten.

Die wesentliche Änderung im Design-Klassendiagramm des Übersetzungssystems ist die Ableitung der Klasse DictionaryEntry(Class) von Properties(JDK). Diese Klasse erlaubt die komfortable Verwaltung von Schlüssel-Wert-Paaren. Ein DictionaryEntry(Class) erweitert diese Schlüssel-Wert-Paare um einen gemeinsamen Regelnamen. Die Regel selbst lässt sich mit der Funktionalität von Properties(JDK)

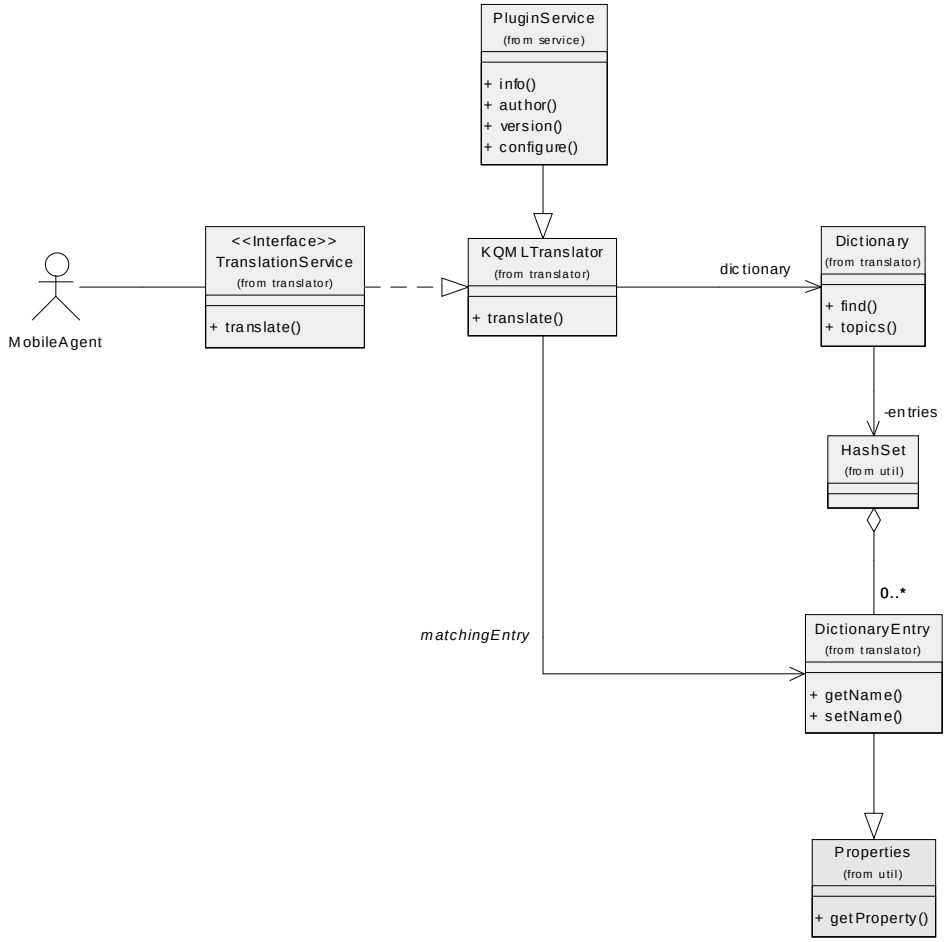


Abbildung 10.3: Design-Klassendiagramm zu *Translate Dialog Result*

ausdrücken. Die Verwaltung der DictionaryEntries(Class) in Dictionary(Class) wiederum geschieht in einem HashSet(Class). Beide Klassen, Properties(Class) und HashSet(Class), sind Standardklassen von Java™.

Der KQMLTranslator(Class) wurde für die Integration in SeMoA von PluginService(Class) abgeleitet. Der TranslationService(Interface) dient, wie in 10.1 erläutert, dem Agenten als allgemeine Schnittstelle zu einem Übersetzer.

10.6 Paketdiagramm

Abbildung 10.4 zeigt das Paketdiagramm dieses Projektes. Die grau hinterlegten Pakete sind im Rahmen dieser Arbeit entstanden. Im Kontext des vollständigen SeMoA -Paketdiagramms handelt es sich um Subpakete des Paketes `DE.FhG.IGD.semoa`.

Die Pakete sind, in *zwei* Ebenen angeordnet. Eine für die Logik des System und eine für die Anbindung an Peripheriegeräte. Dies geschah in Anlehnung an die gängige *Three-Tier-Architecture*, von der im vorliegenden Fall nur zwei der drei Ebenen verwendet werden.

Die Logik des System ist in den Paketen der oberen Ebene enthalten. Die VoiceXML-Elemente werden aufgrund ihrer großen Anzahl und der gleichartigen Funktionalität in einem eigenen Paket `dialog.vxml.elements` zusammengefasst. Das gleiche gilt für die Dialoggeräte, die im Paket `dialog.devices` zu finden sind. Externe Pakete wurden ebenfalls dieser Ebene zugeordnet, wenn sie in ihrer Funktion die Logik des Systems unterstützen.

Die untere Ebene besteht aus den Paketen, die die Anbindung an die Ein- und Ausgabe-Peripherie ermöglichen. Sie wurden bei der Implementierung der DialogInputDevices$^{(Interface)}$ und der DialogOutputDevices$^{(Interface)}$ herangezogen. In dem vorliegenden Demonstrator sind das die Pakete `java.awt` und `javax.speech`.

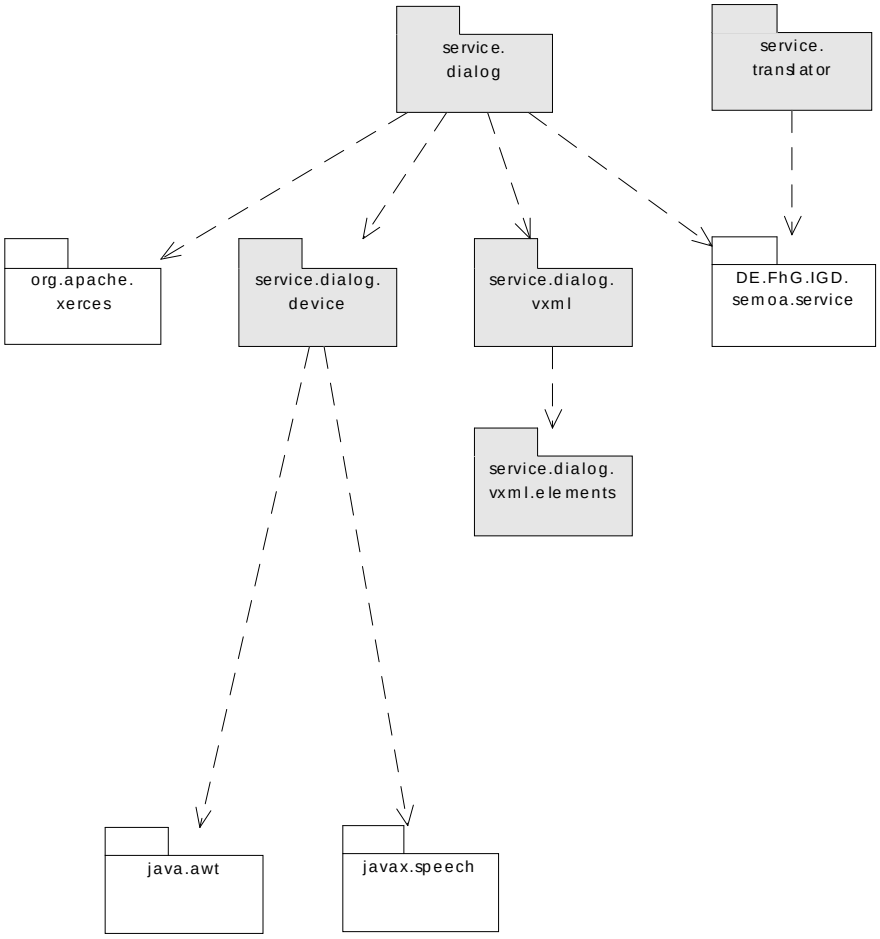


Abbildung 10.4: Paketdiagramm der entwickelten Services

Kapitel 11

Implementation

SeMoA ist aus Gründen der Portabilität und der Sicherheit vollständig in Java™ realisiert. Die Portabilität des Codes spielt im Bereich der Mobilen Agenten eine wichtige Rolle, da der Programmcode der Agenten, in einem heterogenen Netzwerk auf verschiedenen Plattformen zur Ausführung kommen soll. Hier sind die Stärken von Java™ in Bezug auf Plattformunabhängigkeit von großem Vorteil.

Auch die Sicherheitsphilosophie von Java ist, wenn auch nicht perfekt, so doch ausgereifter als bei anderen Programmiersprachen. Die *Virtual Machine* kontrolliert den Zugriff auf Ressourcen des Betriebssystems, es gibt einen weitreichenden Synchronisationsmechanismus und eine zuverlässige Typ-Identifikation zur Laufzeit. Zudem bietet das neue Sicherheitskonzept ab JDK 1.2 die Möglichkeit, diverse Ausführungs- und Zugriffsrechte bis hinunter auf Paket- und Klassenebene zu vergeben [34]. Aus diesen Gründen ist SeMoA vollständig in Java™ implementiert, und daher ist auch die Implementation dieser Arbeit in Java™ vorzunehmen. Getreu den Prinzipien des Objektorientierten Software Engineerings spielte der Aspekt der Implementierungssprache während der Analysephase (siehe Kapitel 9) keine Rolle. Es wurde eine allgemeine Lösung des gestellten Problems erarbeitet. Auch die Designphase (siehe Kapitel 10) konnte zu Beginn noch unabhängig von der Systemumgebung und damit der Implementierungssprache erfolgen. Gegen Ende der Designphase mussten aber Aspekte der Anbindung an Supporterklassen berücksichtigt werden. Hier ist vor allem die Anbindung an die Spracherkennung und die Sprachsynthese zu nennen.

Die Implementation verlief problemlos. Es traten keine Probleme auf, die nicht in Analyse oder Design beachtet wurden. Eine große Hilfe waren eine Reihe von Testrahmen, die während der Evaluation der verwendeten Klassenbibliotheken erstellt wurden. Durch die Evaluation der Bibliotheken war deren Programmierschnittstelle bereits bekannt und so konnte die Anbindung an die Spracherkennung, etc. zügig vorgenommen werden.

Nach der Implementation einer Klasse wurde in einem Testrahmen die funktionale Integrität der Methoden nachgewiesen. Durch geschickte Wahl dieser Testrahmen entstanden daraus kleine Testapplikationen, mit denen sich dann in einzelnen Abschnitten des Klassendiagramms eine referenzielle Integrität nachweisen ließ. Dieses Vorgehen erleichterte die Integrationsphase und den anschließenden Nachweis der funktionalen Korrektheit des Gesamtsystems erheblich. Bei der Integration wurde separat für jeden der beiden entwickelten Services nach der Top-Down-Methode vorgegangen.

Nachdem das System korrekt und stabil lief, wurden Tests mit verschiedenen Agenten und Dialogdateien durchgeführt. Alle Komponenten versehen ihre Funktion gemäß der Spezifikation.

Eine Installation auf dem Zielsystem ist nicht erforderlich. Da es sich bei dem Projekt um ein Subsystem des SeMoA -Systems handelt, sind die entwickelten Services sind bei der Installation eines SeMoA -Servers automatisch enthalten. Die Installation belief sich daher auf das Einchecken der Projektklassen in die Pakethierarchie von SeMoA (`DE.FhG.IGD.semoa`).

Teil III

Resümee

Kapitel 12

Zusammenfassung und Ausblick

12.1 Zusammenfassung

Im Rahmen dieser Diplomarbeit wurde ein Dialogsystem entwickelt, dass einem (Mobilen) Agenten eine multimodale Interaktion mit dem Menschen ermöglicht. Dabei wurde besonderer Wert auf die natürlich-sprachliche Kommunikation gelegt, da sie in wenigen Jahren die vorherrschende Mensch-Maschine-Schnittstelle sein wird. Sprachliche Interaktion wird in Verbindung mit den bisher gängigen Eingabemodalitäten eine hocheffiziente und intuitive Mensch-Maschine-Kommunikation ermöglichen.

Im ersten Teil der Arbeit wurde ein Überblick über alle von dieser Arbeit berührten Bereiche der Informatik gegeben: Agententechnologie, Sprachverarbeitung und Kommunikation über formale Sprachen. Es wurde versucht, die Themen so weit zu besprechen, dass eine fundierte theoretische Grundlage für die Beschreibung der Entwicklungsentscheidungen vorliegt.

Der zweite Teil beschreibt die Entwicklung einer Software, die es einem Mobilen Agenten erlaubt, mit dem Menschen in einen interaktiven Dialog zu treten. Der Inhalt des Dialog wird vom Auftraggeber des Agenten in einer Dialogbeschreibung spezifiziert. Die Flexibilität der Dialogbeschreibungssprache VoiceXML ermöglicht die Kontrolle des Dialogstrangs in Abhängigkeit von bereits vom Benutzer eingeholten Antworten. Aus Gründen der Sicherheit, der Synchronisation und der einfachen Schnittstelle, wird der Dialog nicht vom Agenten selbst, sondern von

einem Dialogservice im Auftrag des Agenten geführt. Dabei verwendet der Dialogservice die ihm bekannten, lokal zu Verfügung stehenden Medien, um mit dem Benutzer zu kommunizieren. Die Antworten des Benutzers werden in einer durch die Dialogbeschreibung festgelegten Weise als formales Resultat an den Agenten zurückgegeben. Der Translations-Service übersetzt dieses formale Resultat dann mit Hilfe eines vom Agenten mitgeführten Dictionaries in eine Agentenkommunikationssprache, hier KQML. Dafür wird ein Eintrag im Dictionary, in Abhängigkeit vom Ergebnis des zuvor geführten Dialoges, vom Agenten ausgewählt. Die Verwendung von Variablen im Dictionary-Eintrag erlaubt es, Nachrichten erst zur Laufzeit zu parametrisieren.

Die Lösung des gestellten Problems erfolgte mit Methoden, Konzepten und Werkzeugen des Objektorientierten Software Engineerings. Das Problem wurde auf seine Essenz hin analysiert, modelliert und schließlich implementiert. Der Demonstrator beherrscht als Modi die verbale Eingabe (Sprachverstehen) und Ausgabe (Sprachsynthese), sowie eine textuelle Ausgabe.

12.2 Ausblick

Die erarbeitete Lösung hat aufgrund ihrer, auf Erweiterbarkeit ausgerichteten Architektur die Chance, in zwei richtungsweisenden Forschungsprojekten eingesetzt zu werden. Dafür werden Erweiterungen des Systems notwendig, die im Rahmen einer Diplomarbeit nicht geleistet werden können.

Zunächst ist hier die Erweiterung des VoiceXML-Interpreters um bisher nicht unterstützte VoiceXML-Tags zu nennen. Die zeitliche Begrenzung der Arbeit war der Grund dafür, dass nur die Hauptfunktionalität von VoiceXML implementiert werden konnte. Das offene Design des VoiceXML-Interpreters ermöglicht diese Erweiterung, ohne bestehenden Code ändern zu müssen.

Ein weiterer Schritt in Richtung menschgerechterer Benutzerschnittstelle würde die Verwendung multimodaler Avatare darstellen. Auch hierfür ist durch die Schnittstellen für Dialogein- und -ausgabegeräte Sorge getragen worden. Das Avatarsystem muss nur die entsprechende Schnittstelle implementieren und kann ansonsten unverändert übernommen werden.

Ähnlich verhält es sich bei der Integration einer Multimodalität, bei der die Eingaben mehrerer Kanäle zu einer Information verknüpft werden. Diese Funktionalität ist im DialogManager nicht vorgesehen. Sie lässt sich jedoch leicht implementieren, wenn man einen *MultimodalityManager* kreiert und diesen als Eingabegerät betrachtet. Er befände sich dann im Klassendiagramm zwischen dem DialogDeviceManager und den tatsächlichen Eingabegeräten. Der *MultimodalityManager* kondensiert mehrere Eingabeinformationen zu einem Dialog-Ergebnis und reicht dieses an den *DialogDeviceManager* weiter.

Es gibt also noch an einigen Stellen des Systems Raum und Bedarf für Erweiterungen. Dank einer gut strukturierten und auf die Zukunft ausgerichteten Architektur können diese Herausforderungen aber sämtlich ohne eine Veränderung der bisherigen Systemstruktur vorgenommen werden. Die zeitliche Investition in eine lange und gründliche Analyse dürfte sich allein dadurch amortisiert haben.

12.3 Schlussbemerkung

Abschließend möchte ich sagen, dass mir die Arbeit an diesem innovativen Projekt sehr viel Freude gemacht hat. Dies verdanke ich nicht zuletzt meinen Kollegen am Fraunhofer Institut, insbesondere Dipl.-Ing. Mehrdad Jalali und Dipl.-Inf. Volker Roth, die für alle Ideen stets ein offenes Ohr hatten mir mit ihrem Rat zur Seite standen.

Prof. Peter Rausch danke ich für seine Lehren über das strukturierte und systematische Software Engineering, mit deren Hilfe die Bewältigung dieser Aufgabe möglich wurde.

Mein letzter und größter Dank gehört meinen Eltern, die mir das Studium ermöglicht haben.

Schließen möchte ich diese Arbeit mit einem Zitat von Marvin Minski über die Beziehung zwischen Mensch und Maschine in der Zukunft [50]:

> “If machines are more similar to us, but are factors of millions of times faster, then it’s hard to see how they could enjoy our company for a few minutes. Perhaps, to maintain a comfortable relationship, we’ll have to handicap them, in significant ways.”

Teil IV

Anhang

Anhang A

Analyse-Klassendiagramme

Anhang B

Design-Klassendiagramme

Anhang C

Beispiel einer Dialogbeschreibung in VoiceXML

```
<?xml version="1.0"?>
<!DOCTYPE vxml SYSTEM "vxml.dtd">

<vxml>
<form id="pizza_order">

    <field name="size">
        <prompt count="1">
            Hello I’m Franco, your piz-
za agent. I can order
            various pizzas for you.

            We have small, medium and large pizzas.
            The small pizza has a diameter of 20 cen-
timeters,
            the medium 30, and the small 40 centimeters.
            Do you want a small, medium or lar-
ge pizza?
        </prompt>
```

```
        <prompt count="2">
            Do you want a small, medium or lar-
ge pizza?
        </prompt>

        <prompt count="3">
            Sorry. I couldn’t understand you.
            Please just say small, medium or large.
        </prompt>

        <grammar>
            [I want [to have] a]
            ( small  {small}
            | medium {medium}
            | large  {large} )
            [pizza]
        </grammar>

        <help>
            I asked you for the size of your pizza.
            Please say small, medium or large.
        </help>
    </field>

    <field name="topping">
        <prompt count="1">
            What kind of pizza do you want?
        </prompt>

        <prompt count="3">
            Please say tunafish, mushrooms or tomatos.
        </prompt>
```

```
        <grammar>
            [[I want [to have] a] [pizza] with]
            ( tunafish  {tonno}
            | mushrooms {funghi}
            | tomatos   {napoli}
            )
        </grammar>

        <help>
           Please say the topping of your pizza.
           You can have mushrooms, tuna-
fish or tomatos.
        </help>
    </field>

    <field name="extra_cheese" type="boolean">
        <prompt>
            Do you want extra cheese?
        </prompt>
    </field>

    <field name="pizza_acknowledge" type="boolean">
        <prompt>
            I have a <value name="size"/> piz-
za <value name="topping"/>.
            Is this correct?
        </prompt>

        <filled>
            <if cond="pizza_acknowledge == true">
            <goto submit="meal size topping ex-
tra_cheese"/>
            </if>
            <clear name="type num date"/>
```

```
        </filled>
    </field>

</form>
</vxml>
```

Anhang D

Bemerkung zur Dokumentationssprache

Wegen der starken internationalen Orientierung der Fraunhofer Gesellschaft werden i.d.R. alle projektbegleitenden Dokumentationen in englischer Sprache abgefasst. Die meisten, in dieser Arbeit dargestellten Abbildungen entstanden im Laufe des Software-Lifecycles, und sind, dieser Konvention folgend, in Englisch gehalten. Dies bezieht sich nicht nur auf die Beschriftungen in Abbildungen, sondern wurde konsequent für jede Art von Dokumentation zur vorliegenden Arbeit durchgehalten. Im Einzelnen betrifft dies:

- sämtliche Dokumente der Analysephase,
- sämtliche Dokumente der Designphase,
- die Kommentierung des Quellcodes,
- Bezeichnerwahl und Kommentare in Konfigurationsdateien,
- Beispieldateien für Dialogbeschreibungen und Übersetzungsregeln

Die einzige Ausnahme stellt der vorliegende Text dar. Wegen der besseren Zuordnenbarkeit wurde sich dafür entschieden, in Erläuterungen zu Abbildungen die englischen Bezeichner auch im deutschen Text zu verwenden.

Literaturverzeichnis

[1] The Associated Press News Service (AP): *Microsoft setzt auf Spracherkennung*. Meldung von Dienstag, 2.November 1999, 02:17 Uhr.

[2] Austin, John: *Zur Theorie der Sprechakte (How to do things with words)*. Reclam, Stuttgart, 1972.

[3] Bundesministerium für Bildung und Forschung: *Pressedokumentation zur Auswahl von Leitprojekten zum Themenfeld "Mensch-Maschine-Interaktion in der Wissensgesellschaft"*. Pressemitteilung vom 02.03.1999, Bonn,`http://www.bmbf.de/deutsch/veroeff/pressedok/pressedok99/pd011999.htm` [Stand: März 2000].

[4] Bundesministrium für Bildung und Forschung: *Bundesministerin Edelgard Bulmahn präsentiert Sieger-Leitprojekte des Wettbewerbs "Mensch-Maschine-Interaktion in der Wissensgesellschaft"*. Presseinformation vom 02.03.1999, Bonn, urlwww.bmbf.de/deutsch/veroeff/presse/pm99/pm030299.htm [Stand: März 2000].

[5] Jacobson, Ivar; Booch, Grady; Rambaugh, James: *The Objectory Software Developement Process*. Addison Wesley, 1998.

[6] Johnson, Mark: *XML for the absolute beginner. A guided tour from HTML to processing XML with Java*. in: JavaWorld, 4/1999, `http://www.javaworld.com/jw-04-1999/jw-04-xml-p.html` [Stand: Dezember 1999].

[7] Braun, Peter: *Über die Migration bei Mobilen Agenten*. Jenaer Schriften zur Mathematik und Informatik, Nr. 99/13, `http://www.minet.uni-jena.de/fakultaet/ips/braunpet/agents/state/ma.html` [Stand: März 1999].

[8] *Brockhaus-Enzyklopädie in 24 Bänden*. 19. Auflage, F. A. Brockhaus Verlag, Mannheim 1986.

[9] Collin, S. M. H.: *Pons Fachwörterbuch Datenverarbeitung. Englisch-Deutsch/Deutsch-Englisch*. 2. neubearbeitete Auflage 1997, Ernst Klett Verlag GmbH, Stuttgart.

[10] Carbonell, Jaime; Hayes, Philip: *Natural-Language Understanding*. aus: *Ecyclodedia of artificial intelligence*, S. 660-677, John Wiley & Sonc Inc., 1990.

[11] Craig, Hunt: *TCP/IP Netzwerk Administration*. O'Reilly 1995.

[12] Dechtjarew, Bianca: *Geldautomaten mit Stimme statt Display*. Heise Online Newsticker, Heise Verlag, `http://www.heise.de/newsticker/data/bid-23.03.00-000` [Stand: März 2000].

[13] Denninghoff, Sabine: *Das Virtuelle Sekretariat*, in: Fraunhofer Magazin, 3/1999, S. 20ff.

[14] Deutsche Forschungsanstalt für Künstliche Intelligenz (DFKI): *Verbmobil*. `http://verbmobil.dfki.de/` [Stand: Januar 2000].

[15] *dtv-Lexikon in 20 Bänden*. Deutscher Taschenbuch Verlag, München 1995.

[16] Drosdowski, Günther [Hrsg.]: *Duden »Etymologie«. Herkunftswörterbuch der deutschen Sprache*. (Der Duden; Bd. 7), Dudenverlag, 1989.

[17] EMBASSI — Elektronische Multimediale Bedien- und Service-Assistenz, `http://www.embassi.de/` [Stand: März 2000].

[18] Fink, Gernot A.: *Integration von Spracherkennung und Sprachverstehen*. (Dissertationen zur künstlichen Intelligenz; Bd. 103) Infix, Sankt Augustin, 1995

[19] Finin, Tim; Wiederhold, Gio; Genesereth, Michael; et al: *DRAFT Specification of the KQML Agent-Communication Language*. The DARPA Knowlegde Sharing Initiative, External Interfaces Working Group, 1993.

[20] Finin, T.; Labrou, Y.; Mayfield, J.: *KQML as an Agent Communication Language*. in: Bradshaw, J. (Hrsg.): Software Agents, MIT Press, 1997.

[21] Finin, T.; Labrou, Y.; Mayfield, J.: *KIF101. A brief introduction to the knowledge interchange format*. UMBC AgentWeb, `http://www.cs.umbc.edu/kse/kif/kif101.shtml` [Stand: November 1999].

[22] Foundation for Intelligent Physical Agents, `http://www.fipa.org/` [Stand: März 2000].

[23] Foundation for Intelligent Physical Agents: *Human-Agent Interaction*. FIPA 98 Specification Part 8, Version 1.0, `http://www.fipa.org` [Stand: März 2000].

[24] Foundation for Intelligent Physical Agents: *Ontology Service*. FIPA 98 Specification Part 12, Version 1.0, `http://www.fipa.org` [Stand: März 2000].

[25] Foundation for Intelligent Physical Agents: *Agent Communication Language*. FIPA 99 Specification Part 2, Version 0.2, `http://www.fipa.org` [Stand: März 2000].

[26] Foundation for Intelligent Physical Agents: *FIPA Content Language Library*. FIPA 99 Specification Part 18, Version 0.2, `http://www.fipa.org/` [Stand: Januar 2000].

[27] Fowler, Martin: UML Distilled. *Applying the Standard Object Modeling Language*. Addison Wesley Longman, 1997.

[28] Fromkin, Victoria; Rodman, Robert: *An Introduction to Language*. Harcourt Brace College Publishers, Orlando 1993.

[29] Gehmeyr, Andreas: *Multiagentensysteme*. `http://www.sigs.de/publications/docs/jspk/gehmeyr.html` [Stand: Februar 2000].

[30] Genesereth, Michael; Ketchpel, Steven: *Software Agents*. in: Communications of the ACM, 7/1994, S. 48ff.

[31] Genesereth, Michael: *Knowledge Interchange Format. draft proposed American National Standard (dpANS)*. NCITS.T2/98-004, `http://logic.stanford.edu/kif/dpans.html` [Stand: Dezember 1999].

[32] General Magic, `www.generalmagic.com/technology/mobile_agent.html` [Stand: März 2000].

[33] Gilbert, Don: *The Role of Intelligent Agents in the Information Infrastructure*. IBM Intelligent Agent Center of Competency, 1995.

[34] Gong, Li: *Java™Security Architecture (JDK 1.2)*. Lokal im Installationsverzeichnis des JDK1.2 unter `/docs/guide/security/spec/security-spec.doc.html`.

[35] Grasso, Michael; Finin, Tim: *Task Integration in Multimodal Speech Recognition Environments*. ACM Crossroads, `http://www.acm.org/crossroads/xrds3-3/taskint.html` [Stand: März 2000].

[36] Greif, Irene: *Desktop Agents in Group-Enabled Products*. in: Communications of the ACM, 7/1994, S. 100ff.

[37] Gruber, Tom: *What is an Ontology?*, Stanford University, `http://www-ksl.stanford.edu/kst/what-is-an-ontology.html` [Stand: Oktober 1999].

[38] Hohl, F.: *The mobile agent list*. `http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole/mal/mal.html` [Stand: März 2000].

[39] IBM: *Sprachforschung bei IBM*. `http://www.research.ibm.com/hlt/` [Stand: Januar 2000].

[40] Ilmberger, Hermann; Schmitz, Jürgen; Thürmel, Sabine: *Mobile Agents — Mobile Components*. in: Component User Conference 1996, Cambridge University Press, 1998.

[41] Kinnebrock, Werner: *Handbuch Turbo Prolog*. Oldenbourg Verlag, München, 1990.

[42] Kinnebrock, Werner: *Künstliches Leben. Anspruch und Wirklichkeit*. Oldenbourg Verlag, München 1996.

[43] Kuhlmann, Ulrike: *Achtung Aufnahme. Der sprachgesteuerte Videorecorder*. in: c't 23/1999, S.250.

[44] Kuhlmann, Ulrike: *Lehrmeister. Diktiersoftware IBM ViaVoice Millennium*. in: c't 23/1999, S.84.

[45] Kuhlmann, Ulrike: *Spracherkennung für Palm & Co*. Heise Online Newsticker, Heise Verlag, `http://www.heise.de/newsticker/data/uk-08.03.00-001` [Stand: März 2000].

[46] Lea, Doug: *Concurrent Programming in Java™. Entwurfsprinzipien und Muster*. Addison Wesley, Bonn, 1997.

[47] Lenzmann, Britta: *Benutzeradaptive und multimodale Interface-Agenten*. (Dissertationen zur künstlichen Intelligenz; Bd. 184) Infix, Sankt Augustin 1998.

[48] Mangold, Helmut (Hrsg.): *Sprachliche Mensch-Maschine-Kommunikation*. Oldenbourg Verlag, München, 1992.

[49] Mattern, Friedemann: *Mobile Agenten*. in: it+ti – Informationstechnik und Technische Informatik, 4/98, S. 12ff, Fachbereich Informatik, Technische Universität Darmstadt, `http://www.informatik.tu-darmstadt.de/VS/Publikationen/papers/mobags.html` [Stand: März 2000].

[50] Minsky, Marvin; Riecken, Doug: *A Conversation with Marvin Minsky about Agents*. in: Communications of the ACM, 7/1994, S.23ff.

[51] Mintert, Stefan: *Persönliche Ansprache. Marktübersicht Diktiersoftware*. in: iX 2/2000, S. 85ff.

[52] Machine Understanding Group at the MIT Media Laboratory: *An Unprincipled Parser*. Massachusetts Institute of Technology, `http://mu.www.media.mit.edu/phraser.html` [Stand: Dezember 1999].

[53] Müller, Frank: *Was der Fall ist. Entwicklungswerkzeuge im Vergleich: OEW und OTW*. in: iX 2/2000, S.68ff.

[54] Norman, Donald: *How Might People Interact with Agents*. in: Communications of the ACM, 7/1994, S.68ff.

[55] Object Mangement Group: *OMG Unified Modeling Language Specification*. Version 1.3, 1999, `http://www.omg.org/` [Stand: Dezember 1999].

[56] Ottmann, Thomas; Widmayer, Peter: *Algorithmen und Datenstrukturen*. (Reihe Informatik, Bd. 70), BI-Wissenschafts-Verlag, Mannheim, 1993.

[57] Oviatt, Sharon: *Ten Myths of Multimodal Interaction*. in: Communications of the ACM, 11/1999, S.74ff.

[58] Radev, Dragomir: *Natural Language Processing FAQ*. Monatliches FAQ Posting in der Newsgroup `comp.ai.nat-lang`, Aktuelle Version unter `ftp://rtfm.mit.edu/pub/usenet-by-hierarchy/comp/ai/nat-lang`.

[59] Rausch, Peter: *Software Engineering*. Skript zur Vorlesung “Software Engineering”, Fachhochschule Bingen, 1995.

[60] Rausch. Peter: *Objektorientierte Systementwicklung mit der Unified Modeling Language (UML)*. Skript zur Vorlesung “Objektorientierte Systementwicklung”, Fachhochschule Bingen, 1998.

[61] Reinhold, Mark: *An XML Data-Binding Facility for the Java™ Platform*. Core Java Platform Group, Sun Microsystems, 1999.

[62] Roth, Volker; Jalali, Mehrdad: *Access Control and Key Management for Mobile Agents*. in: Computer & Graphics, Vol. 22, 4/1998, S. 457-461.

[63] Roth, Volker; Jalali, Mehrdad: *SeMoA — Secure Mobile Agents*. Unveröffentlichtes Manuskript, Fraunhofer Institut für Graphische Datenverarbeitung, Darmstadt 1999.

[64] Roth, Volker; Jalali, Mehrdad: *Concepts and Architecture of a Security-centric Mobile Agent Server*. Fraunhofer Institut für Graphische Datenverarbeitung, Rundeturmstraße 6, 64283 Darmstadt, Germany.

[65] Roth, Volker; Jalali, Mehrdad; Hartman, Roger; Roland, Christophe: *An Application of Mobile Agents as Personal Assistents in Electronic Commerce*. Fraunhofer Institut für Graphische Datenverarbeitung, Rundeturmstraße 6, 64283 Darmstadt, Germany.

[66] Russell, Stuart; Norvig, Peter: *Artificial Intelligence. A Modern Approach*. Prentice Hall International, 1995.

[67] Sagerer, Gerhard: *Automatisches Verstehen Gesprochener Sprache*. (Reihe Informatik Bd. 74) BI-Wissenschaftsverlag, Mannheim 1990.

[68] Schefe, Peter: *Künstliche Intelligenz — Überblick und Grundlagen: Grundlegende Konzepte und Methoden zur Realisierung*. (Reihe Informatik, Bd. 53), BI-Wissenschafts-Verlag, Mannheim, 1991.

[69] Schneier, Bruce: *Angewandte Kryptographie. Protokolle, Algorithmen und Sourcecode in C*. Addison Wesley, 1996.

[70] Schoffa, Georg: *Die Programmiersprache Lisp*. Franzis-Verlag GmbH, München, 1987.

[71] SeMoA : Dokumentation zur SeMoA Plattform, im Installationsverzeichnis von SeMoA unter `docs`.

[72] Sommers, Bret: *Agents: Not just for Bond anymore*. in: JavaWorld 4/97, `http://www.javaworld.com/jw-04-1997/jw-04-agents.html` [Stand: Juni 1999].

[73] Spierling, Ulrike; Behr, Johannes: *Conversational Integration of Multimedia and Multimodal Interaction*. in: CG topics 4/1999, S.8ff.

[74] Sun Microsystems: *Java™Speech API Homepage*, `http://java.sun.com/products/java-media/speech/` [Stand: Dezember 1999].

[75] Sundsted, Todd: *Agents on the move*. in: JavaWorld 7/98, `http://www.javaword.com/jw-07-1998/jw-07-howto_p.html` [Stand: Juni 1999].

[76] Sundsted, Todd: *Java make most of XML. How to build applications taht handel XML's extensibility*. in: JavaWorld 7/99, `http://www.javaworld.com/jw-07-1999/jw-07-howto.html` [Stand: Dezember 1999].

[77] Sycara, Katia; Dajun, Zeng: *Coordination of Multiple Intelligent Software Agents*. The Robitics Institute, Carnegie Mllon University, Pittsburgh, PA 15213, United States of America. in: International Journal of Cooperative Information Systems, World Scientific Publishing Company.

[78] Tanimoto, Steven L.: *KI : Die Grundlagen*. Oldenbourg Verlag, München 1990.

[79] Tekada, Hideaki; Iino, Kenji; Toyoaki, Nishida : *Agent Communication with Multiple Ontologies*, Graduate School of Information Science, Nara Institute of Science and Technology (NAIST), Nada, Japan.

[80] UMBC AgentWeb: *News and Information about Agent Technology*, `http://www.cs.umbc.edu/agents` [Stand: April 2000].

[81] U.S. Patent and Trademark Office: *System and Method for Distributed Computation Based upon the Movement, Execution and Interaction of Processes in a Network*. US patent no. 5603031, `http://www.uspto.gov/` [Stand: April 2000].

[82] Venners, Bill: *Under the Hood: The architecture of aglets*. in: JavaWorld 4/97, `http://www.javaworld.com/jw-04-1997/jw-04-hood.html` [Stand: Juni 1999].

[83] VoiceXML Consortium: `http://www.voicexml.org/` [Stand: November 1999].

[84] World Wide Web Consortium: *Voice Browsers. An introduction an glossary for the requirement drafts*. W3C Working Draft 23 Decembre 1999, `http://www.w3.org/TR/voice-intro/` [Stand: März 2000].

[85] Warth, Dora: *Künstliche Intelligenz: Spracherkennug und Sprachverstehen*. Johannes Gutenberg-Universität Mainz, Fachbereich Angewandte Sprach- und Kulturwissenschaft, Referat zum Hauptseminar “Psycholinguistik: Mentale Prozesse in der Sprachverarbeitung”, SS 1997, letzte Bearbeitung 19.8.1999, URL: `http://www.fask.uni-mainz.de/user/warth/Ki.html` [Stand: November 1999].

[86] Weischedel, Wilhelm: *Die Philosophische Hintertreppe: 34 große Philosophen in Alltag und Denken*. 26.Auflage, Deutscher Taschenbuch Verlag, München, 1996.

[87] Woolridge, Michael; Jennungs, Nicholas: *Intelligent Agents: Theory and Pratice*. Knowledge Engineering Review, 10(2), 1995.

[88] Zink, Michael: *Zusammenspiel, Mensch-Maschine-Schnittstellen in Sprachsystemen*. in: c´t 3/1999, S.133 .

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ort, Datum (Ulrich Pinsdorf)