Objektorientierter Entwurf und Realisierung eines Agentensystems für kooperative Internet-Informationssysteme

Diplomarbeit

Holm Wegner
Fachbereich Informatik
UNIVERSITÄT HAMBURG

Betreuung:

Prof. Dr. Florian Matthes,
Arbeitsbereich Softwaresysteme
TECHNISCHE UNIVERSITÄT HAMBURG-HARBURG

Prof. Dr. Leonie Dreschler-Fischer,
Fachbereich Informatik
UNIVERSITÄT HAMBURG

26. Mai 1998

Zusammenfassung

In dieser Arbeit wird eine Umgebung für mobile Agenten zur anwendungsnahen Modellierung und Implementierung von verteilten Systemen entworfen und realisiert. Die Formalisierung der zur Kooperation der Partner nötigen Kommunikation geschieht durch das Modell der *Business Conversations*, das die zielgerichtete, langandauernde Interaktion zweier Akteure zur Erfüllung einer gemeinsamen Aufgabe beschreibt. Das System wird mit objektorientierten Techniken entworfen und beschrieben sowie in der objektorientierten Programmiersprache TL-2 des persistenten TYCOON-2-Systems implementiert. Zur Erprobung des Modells wird eine prototypische Implementierung eines Internet-Informationssystems am Beispiel eines Hotel-Reservierungssystems verwendet.

Deutschsprachige Arbeiten in Informatik zu schreiben ist wegen der überwiegend englischen Fachterminologie nicht einfach. Natürlich ist es notwendig und sinnvoll, eine international einheitliche Fachsprache zu entwickeln und zu benutzen, aber der Lesbarkeit deutscher Arbeiten ist die übermäßige Verwendung englischer Fachausdrücke abträglich. Leider ist vielfach eine unkritische und unnötige Verwendung englischen *computer-slangs* (sic!) zu beobachten. Die vielen englisch-deutschen Komposita sind dabei die traurige Krönung.

In dieser Arbeit wird folgendermaßen verfahren: Für die meisten englischen Begriffe existieren deutsche Synonyme, die stattdessen verwendet werden; im Zweifel wird das Fremdwort zusätzlich in Klammern angegeben. Läßt es sich – etwa in Ermangelung eines (guten) deutschen Korrelats – nicht vermeiden, so werden englische Fremdwörter auch im laufenden Text verwendet. In allen Fällen jedoch werden sie durch *kursive* Schriftart abgehoben. Außerdem werden neu eingeführte oder besonders wichtige deutsche Begriffe ebenfalls *kursiv* ausgezeichnet; Firmen- und Produktnamen sowie Personennamen werden in KAPITÄLCHEN gesetzt. Um ein ausgeglichenes Schriftbild zu erhalten und damit die Lesbarkeit des Textes zu erhöhen, wird nur noch eine weitere Schriftarten verwendet: Inhärent englische Bezeichnungen wie Klassennamen und Programmfragmente sowie Schlüsselwörter von Programmiersprachen werden durchgängig in `Schreibmaschinenschrift` gesetzt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Ich hätte gerne ein gutes Buch hervorgebracht.
Es ist nicht so ausgefallen; aber die Zeit ist vorbei,
in der es von mir verbessert werden könnte.

– L. Wittgenstein

Kapitel 1

Einleitung

Kooperative Informationssysteme sind in den letzten Jahren als die Informationssysteme der nächsten Generation (*next generation information systems* [DDJ+97]) nicht nur ein Schwerpunkt des Forschungsinteresses geworden, sondern haben sich insbesondere durch das explosive Wachstum des Internet auch als kommerziell überaus wichtige Entwicklung herausgestellt. Dabei versucht dieses Gebiet, die bisher eher getrennt verlaufenen Entwicklungen der Systemtechnik, der computerunterstützten Zusammenarbeit (*computer supported cooperative work*, CSCW) und der Organisationsmodellierung und -planung zusammenzufassen.

Beobachtet man die allgemeine Entwicklung von Informationssystemen, so wird deutlich, daß sich sowohl die Menge der bereitstehenden und verarbeiteten Informationen als auch die Dynamik der Veränderung der zugrundeliegenden organisatorischen Gegebenheiten und Geschäftsprozesse drastisch erhöht hat. Als Folge (oder Ursache?) davon läßt sich ein tiefgreifender Wandel in den organisatorischen und technischen Strukturen der verwendeten Informationssysteme ausmachen.

Diese Wandlung von abgeschlossenen, monolithischen und zentralisierten Informationssystemen hin zu offenen, integrativen und verteilten Lösungen erfordert neue Konzepte, Methoden und Modelle für den Entwurf solcher sich rapide entwickelnden Systeme, um die Auswirkungen der Änderungen in der realen Welt auf die Systeme beschreiben und ihnen folgen zu können.

Kooperative Informationssysteme wollen wir hier als verteilte Systeme verstehen, in denen mehrere autonome Agenten oder Akteure bestrebt sind, durch Koordinierung langandauernder Aktivitäten ihre gemeinsamen Ziele zu erreichen [Mat97b].

Die Aufgabe der Programmiersprachen und der sogenannten *middleware* (Datenbanken, Programmgeneratoren, Entwicklungssysteme etc.) ist in diesem Zusammenhang vor allen Dingen, dem Anwender, also dem Entwickler, Abstraktionsmechanismen an die Hand zu geben, die es ihm erlauben, von den Details der Kooperation zu abstrahieren [Mat97a]. Dabei sind drei Arten der Abstraktion zu nennen:

1. Abstraktion von der Kooperation über die *Zeit*: Die Programme und die von ihnen bearbeiteten Daten müssen unabhängig von der Lebensdauer der sie ausführenden Betriebssystem-Prozesse, Datenbanktransaktionen und der sie beschreibenden Sche-

mata zumindest über die Dauer der tatsächlichen, in der realen Welt vorhandenen Geschäftsprozesse hinweg bestehen bleiben können.

2. Abstraktion von der Kooperation im *Raum*: Die softwaretechnischen Akteure und die Daten müssen außerdem in der Lage sein, zur effizienten Erfüllung ihrer Aufgaben frei in der physisch verteilten Umgebung zu migrieren. Dabei dürfen die verschiedenen Rechnerplattformen, Betriebssysteme etc. keine Hindernisse darstellen.

3. Abstraktion von der Kooperation in verschiedenen *Modalitäten*: Alle Daten müssen in gleicher Weise uniform von den Akteuren verwendet werden können, unabhängig von der Modalität der Kooperation. Es muß völlig davon abstrahiert werden können, ob klassische Stapelverarbeitung, *online transaction processing*, direkte Manipulation der Daten mit Hilfe graphischer Benutzungsoberflächen durch Menschen, Workflow-managementsysteme, mobile Softwareagenten etc. stattfinden und auf welche Art und Weise, mit welchem Protokoll und über welches Medium die Daten dann transportiert werden (*e-mail*, HTML, HTTP, CORBA-Objekte, RPC-Aufrufe, UNIX-*Sockets*, schriftliche Formulare, Post, FAX etc.).

Bei der Systementwicklung soll so eine Konzentration auf das wesentliche, das „was“ und das „wie“ stattfinden; abstrahiert werden soll so weit wie möglich vom „wann“, „wo“ und „wer“. Konkret bedeutet dies beispielsweise, daß man bei der Entwicklung von unnötigen, zeitraubenden und fehleranfälligen Details der Datenkonversationen und des Datentransfers sowie von Synchronisierungsaufgaben entlastet wird. Im Extremfall geschehen diese völlig transparent.

Dem ersten der oben genannten Punkte wurde bereits früh durch die Entwicklung der klassischen Datenbanksysteme begegnet, die den Anwendern nahezu vollständige *Persistenzabstraktion* für die Daten bieten. Ein Überblick über die damit verbundenen Konzepte und Lösungen bietet [LS87]. Die Ausdehnung der Persistenzabstraktion nicht nur auf Daten, sondern auch auf Programmcode und zum Schluß sogar auf Prozesse durch die *orthogonale Persistenz* in Objektsystemen ermöglicht nicht nur langandauernde, sichere und fehlererholende Prozesse, sondern auch die *orthogonale Mobilität* der Daten, des Programmcodes und der Prozesse. Durch die damit gegebene Möglichkeit, einen Prozeß mitsamt der von ihm bearbeiteten Daten von System zu System migrieren zu lassen, wurde in eindrucksvoller Weise dem zweiten Punkt Rechnung getragen [MMS96, Mat96].

Der dritte Punkt stellt das gegenwärtig noch am weitesten offene Gebiet dar. Der dort geforderten *Medien- und Aktorenunabhängigkeit* der Kooperation wird häufig durch den Einsatz eines Agentenmodells entsprochen, wie es etwa das TELESCRIPT-System [Whi94] darstellt. Ein dabei häufig ungenügend berücksichtigter Punkt ist aber die Medienunabhängigkeit selbst, die ohne eine *Formalisierung* der zur Kooperation notwendigen Interaktion noch nicht wirklich gegeben ist.

Das in [Mat97a, Mat97b] vorgestellte Modell der *Business Conversations*, das sich bei der Modellierung der Interaktion zweier Akteure an die linguistische Theorie der Sprechakte anlehnt, schafft dafür eine geeignete Grundlage, so daß seine Verwendung in einem Agentensystem vorteilhaft erscheint.

1.1 Ziele der Arbeit

Eine prototypische Implementierung des Modells der *Business Conversations* zusammen mit einem Agentensystem wurde in [Joh97] vorgenommen und dort als *Tycoon Business Conversations* bezeichnet. Diese Implementierung fand in TL statt, der funktionalen Programmiersprache der am Arbeitsbereich DBIS der Universität Hamburg entwickelten polymorphen, persistenten Programmierumgebung TYCOON [Mat93, MMM93, MMS94].

Ziel dieser Arbeit ist, für das in [Mat97a, Mat97b] beschriebene Modell der *Business Conversations* eine objektorientierte Modellierung zu erarbeiten und darauf eine Implementierung in der objektorientierten Programmiersprache TL-2 des neuen TYCOON-2-Systems vorzunehmen. Das resultierende System wird im folgenden ebenfalls *Tycoon Business Conversations* (TBC) genannt.

Um der Anforderung, den Bau kooperativer Informationssysteme zu unterstützen, gerecht zu werden und um die Leistungsfähigkeit des Modells und der Implementierung zu untersuchen, soll ferner eine prototypische Anwendung realisiert werden. Gewählt wird dafür ein Szenario für ein Internet-Informationssystem, das die Bereitstellung eines interaktiven Hotel-Reservierungssystems zur Aufgabe hat. Der Vorteil ist, daß nicht eine artifizielle „Spielzeuganwendung“ mit geringer Aussagekraft verwendet werden, sondern ein realen Anforderungen genügendes System geschaffen werden muß. Die den Rahmen dieser Arbeit natürlich sprengende Aufgabe der Analyse der zugrunde liegenden Geschäftsprozesse und der Modellierung, des Entwurfs und der Implementierung des eigentlichen Reservierungssystems wird in einer anderen Arbeit vorgenommen, wobei dort der Schwerpunkt auf der Entwicklung einer Methode zur bruchlosen Gestaltung des Analyse- und Entwurfsprozesses liegt [Rip98].

Als weitere Anforderungen und in dieser Arbeit zu untersuchende bzw. zu berücksichtigende Aspekte ergeben sich die folgenden Punkte:

- Die objektorientierte Modellierung und Notation soll mit Hilfe der *Unified Modelling Language* (UML) erfolgen [BJR96, FS97].
- Beim Entwurf soll nach Möglichkeit von bewährten Entwurfsmustern (*design pattern*) Gebrauch gemacht werden, wie sie etwa in [GHJV95] beschrieben werden.
- Die neusten Ansätze des sich rapide entwickelnden Gebiets der *mobilen Agenten* sollen untersucht und bewertet werden.
- Es soll überprüft werden, welche Teil-Funktionalitäten der in [Joh97] vorgenommenen Implementierung sich als unnötig und welche sich als erhaltenswert erweisen. Diesem Punkt kommt insofern Bedeutung zu, als die vorliegende, alte Modellierung und die zugehörige Implementierung sich sehr stark an dem TELESCRIPT-Agentensystem [Whi94] und dessen Konzepten orientieren.
- Neue Ansätze, die die Evolution in kooperativen Informationssystemen unterstützen, sollen berücksichtigt werden. Hier sind insbesondere Erweiterungen des bisherigen Modells der *Business Conversations* möglich.

- Die Eignung der Systemumgebung TYCOON-2 für die Entwicklung und für den Betrieb des prototypischen Informationssystems soll überprüft werden.
- Vorhandene Systemtechnik, die das TYCOON-2-System bereits für die Realisierung von Internet-Anwendungen enthält, soll bewertet und ggf. wiederverwendet werden. Auch mögliche und anderswo verwendete Alternativen sollen dargestellt, verglichen und bewertet werden.
- Das zu entwerfende System soll möglichst klar gegliedert, einfach verständlich und offen für zukünftige Erweiterungen des Modells sein. Insbesondere soll diese Arbeit einen Leitfaden dafür darstellen.

1.2 Aufbau der Arbeit

Im anschließenden Kapitel 2 werden zunächst die Bedeutung von kooperativen Informationssystemen, die dabei auftauchenden Probleme sowie als Lösung dafür das Modell der *Business Conversations* dargestellt. Darauf folgen eine Klassifikation von Agentensystemen sowie eine Vorstellung von drei weiteren, konkreten Systemen. Abgeschlossen wird das Kapitel mit einer Bewertung der vorgestellten Agentensysteme und einer Einordnung des zu entwerfenden Systems.

Im Kapitel 3 werden dann eine Einführung in die zur Realisierung verwendete Programmiersprache TL-2 und eine Übersicht über das TYCOON-2-System gegeben, soweit dies für das Verständnis der Arbeit notwendig erscheint.

Kapitel 4 dokumentiert den Systementwurf und das dynamische Verhalten des realisierten Systems ausführlich. Ferner werden einige weiterführende Konzepte vorgestellt, die die Evolution von kooperativen Informationssystemen betreffen.

Die Realisierung des prototypischen Informationssystems des Hotel-Reservierungssystems mit Hilfe der *Tycoon Business Conversations* wird danach in Kapitel 5 beschrieben. Dabei werden auch alternative Ansätze dargestellt, verglichen und bewertet.

Schließlich gibt Kapitel 6 eine Zusammenfassung und Bewertung des Erreichten und stellt einen Ausblick auf weitergehende Möglichkeiten und offengebliebene Fragen dar.

Kapitel 2

Kooperative Informationssysteme

In diesem Kapitel wird in Abschnitt 2.1 die Bedeutung von verteilten Anwendungen und von kooperativen Informationssystemen dargestellt. Insbesondere werden die Probleme der Evolution solcher Systeme beleuchtet. In Abschnitt 2.2 wird dann als Antwort auf die zuvor aufgeworfenen Fragen und Probleme das Modell der *Business Conversations* genau dargestellt. Dabei werden umfassend das dahinterliegende Leitbild, die formale Spezifikation der Interaktion und der dynamische Ablauf der Kommunikation behandelt. Der Begriff des *Agenten* wird in Abschnitt 2.3 näher bestimmt. Dabei werden drei neuere Entwicklungen auf dem Gebiet der Agenten näher untersucht und verglichen. Abschließend wird in Abschnitt 2.4 eine Einordnung und Definition des in dieser Arbeit entwickelten Systems im Lichte der vorangegangenen Erkenntnisse vorgenommen.

2.1 Verteilte Anwendungen

Der bereits in der Einleitung angesprochene Wandel von Informationssystemen läßt sich grob durch drei Entwicklungsstufen, die die Struktur von Geschäftsanwendungen durchlaufen hat, charakterisieren [Mat97a]. Die praktisch relevanten und im Einsatz befindlichen Systeme beruhen allerdings noch hauptsächlich auf der ersten Stufe.

Verteilte Daten: Die gemeinsame Benutzung der unternehmensweit vorhandenen Daten, typischerweise inzwischen durch relationale Datenbanksysteme, ist die älteste Form der Verteilung. In der Regel geschieht dabei die Datenhaltung auf zentralen Datenbankservern, und die Abwicklung der eigentlichen Geschäftsprozesse erfolgt durch integrierte Anwendungen wie SAP R/3, BAAN oder andere branchenspezifische oder individuelle Softwarelösungen, die darauf aufsetzen.

Verteilte Objekte: Mit Aufkommen der objektorientierten Sprachen und besonders dem verstärkten Einsatz für die Erstellung von Geschäftsanwendungen begann auch die Suche nach äquivalenten Architekturen für die Modellierung der zugrundeliegenden Daten und Prozesse. Die Idee dabei ist, durch den Einsatz von verteilten Objekten, die nicht nur die Daten irgendwelcher Geschäftsobjekte der realen Welt, sondern auch die

Methoden zum Zugriff und zu deren Manipulation enthalten, flexiblere, skalierbarere und besser wartbare Systeme bauen zu können [OHE96].

Durch Verwendung von objektorientierter Eigenschaften ließen sich z.B. die Hausratversicherungen auf die generellere Klasse der Schadensversicherungen zurückführen. Der Wunsch ist ferner, auf diese Art und Weise standardisierte Schnittstellen für alle relevanten Geschäftsdaten in Form von objektorientierten Klassen zu erhalten. Hinzu kommt die Forderung nach transparenter Verteilung der Objekte durch geeignete *middleware*, wie sie etwa das Modell der CORBA (*common object request broker architekture*) vorsieht.

Fraglich bleibt dabei allerdings, ob dieses Modell tatsächlich den Anforderungen an kooperativer Informationssysteme genügt. Dagegen spricht vor allen Dingen die sich bereits jetzt abzeichnende Schwierigkeit, generelle, für alle Beteiligten ausreichende Schnittstellen der zu modellierenden Geschäftsobjekte zu entwerfen und sich auf eine gemeinsame Semantik der Methoden zu einigen. Des weiteren ist durch diese Modellierung eine Kapselung des gemeinsamen Verhaltens mehrerer Objekte nicht oder nur sehr schwer zu erreichen, da das Verhalten meist sehr komplex und deshalb über viele Objekte verteilt sein wird.

Verteilte Agenten: Der Gedanke von „Agenten“, die durch die Vorgabe von Zielen und die Benutzung einer Wissensbasis autonom eine Aufgabe erledigen, fasziniert seit einiger Zeit nicht nur die Forschungsgemeinde. Der Aspekt der Autonomie, die nur zum Zwecke der Kommunikation zur Erledigung der Aufgabe unterbrochen wird, stellt dabei den Schlüsselgedanken für die agentenzentrierte Verteilung dar. Die Idee ist dabei, Systeme so zu zerlegen, daß die gewonnenen kleinsten Teile, die Agenten, wohldefinierte, kurzzeitige oder langfristige Aufgaben bewältigen, die sowohl durch einen Menschen als auch durch ein Programm ausgeführt werden können. Dazu sollen die Softwareagenten auch in der Lage sein, von System zu System (Rechner zu Rechner) zu migrieren.

Es wäre ob der oben geschilderten Probleme mit verteilten Objekten ein Fehler, Agenten einfach dadurch zu definieren, daß man Geschäftsobjekten um die Fähigkeit zu migrieren und zu kommunizieren erweitert. Stattdessen wollen wir uns konzeptuell auf die Beschreibung der zwischen Agenten möglichen Interaktion konzentrieren. Diese Interaktion soll durch Kommunikation zur Koordinierung der langandauernden Aktivitäten der Agenten führen, wobei die Agenten ein Maximum an Autonomie bewahren müssen. Ein weitergehender Vergleich von objekt- und agentenbasierter Kommunikation wird in [Ric97] angestellt.

Drei weitere Beobachtungen, die die Notwendigkeit und die Nützlichkeit des in Verbindung mit einer Formalisierung der Interaktion definierten agentenzentrierten Modells unterstützen, sollen an dieser Stelle aufgeführt werden.

Das explosive und zur Zeit anscheinend immer noch exponentielle Wachstum des Internet belegt die Akzeptanz und den Markt für solche Dienste, wenn man das zugrundeliegende Kommunikationsprotokoll betrachtet. Die interessanteste Beobachtung dabei ist, daß viele Anbieter von Standardsoftware dazu übergehen, Anbindungen an ihre Produkte (z.B. Datenbanken) zu erstellen, die den direkten Zugriff aus dem Internet darauf erlauben. Eine weitere Entwicklung, die sich erst am Anfang befindet, ist das Angebot von sogenannten

Mehrwertdiensten im Internet, die, als *Broker* oder Agentur fungierend, durch das Heranziehen mehrerer anderer Dienste kombinierte oder verbesserte neue Dienste bieten oder an andere Dienste vermitteln. In beiden Fällen zeigt sich häufig die Notwendigkeit für neue Konzepte, die die Interaktion zwischen den beteiligten Partnern spezifizieren. Die bestehenden Techniken und Protokolle führen ob ihrer für diese Zwecke eingeschränkten Tauglichkeit schnell zu komplizierten, undurchschaubaren Lösungen (siehe dazu auch Kapitel 5).

Die zweite wichtige Beobachtung ist, daß bei der Entwicklung sehr großer, monolithischer Anwendungssysteme mit der Zeit die Probleme bei ihrer Wartbarkeit (*maintainance*) immer größer werden und zum Schluß sogar den größten Teil der Weiterentwicklung ausmachen: MEYER schätzt einen Kostenanteil von 70% für die Softwarewartung [Mey88]. Besonders kraß ist dies beim System R/3 der SAP zu beobachten [Rip98]. Die auch von SAP verfolgte Strategie könnte sein, die großen, monolithischen Systeme in kleinere, unabhängig wartbare Module (Satellitensysteme) mit wesentlich geringerer Komplexität aufzuteilen. Dazu allerdings sind wiederum standardisierte Kommunikationsprotokolle mit hohem Abstraktionsgrad notwendig, die insbesondere robust gegenüber kleineren Änderungen bei nur einem der Kommunikationspartner sind.

Als Drittes muß gesehen werden, daß die Investitionen der Firmen in ihre zum Teil sehr umfangreichen Systeme beträchtlich sind und eine Runderneuerung aller Anwendungen – so verlockend sie technologisch auch sein mag – z.T. gewaltige Kosten verursachen würde. Deshalb ist es vielfach vorteilhafter, Schritt für Schritt nur Teile des Gesamtsystems auszutauschen und die verbliebenen alten Systeme (*legacy systems*) „irgendwie“ an die neuen anzubinden. Hierfür bietet sich wiederum eine Betrachtung sowohl der alten als auch der neuen Anwendungen als Agenten an, die spezielle Aufgaben erfüllen und zur Interaktion Protokolle auf hohem Abstraktionsniveau benötigen.

Es ist also zu konstatieren, daß die Formalisierung der Interaktion bei Agentensystemen das entscheidende Problem ist, nicht so sehr die Modellbildung der Agentenfunktionalität selber. Dies ist angesichts der geforderten Autonomie von Agenten eigentlich auch offensichtlich.

Zuletzt soll ein in den letzten Jahren immer wichtiger gewordener Aspekt angesprochen werden, dem hier ebenfalls Rechnung getragen werden muß. Das unter dem Schlagwort *business process reengineering* bekannte Bestreben, Geschäftsprozesse ständig zu verbessern, führt zu sich wesentlich schneller ändernden organisatorischen und betriebswirtschaftlichen Abläufen als früher, denen natürlich auch auf der Seite der Systemtechnik zu folgen ist. Wurde früher die Systemtechnik als der die Entwicklung (zeitlich) bestimmende Faktor gesehen, so scheint es heute eher umgekehrt zu sein. Schichtenmodelle, die eine Entkoppelung der Anwendungssichten von der Realisierung der Systeme leisten (wie etwa die ANSI/SPARC-Architektur für Datenbanksysteme [BS82, LS87]), sind also nach wie vor erforderlich, müssen aber auch völlig neuen Anforderungen genügen. In [DDJ+97] wird dies als wesentliche Herausforderung an heutige Informationssysteme betrachtet; deshalb wird dem sogenannten *change management* als Teildisziplin kooperativen Informationssysteme ein zentraler Platz eingeräumt.

Eine zentrale Anforderung an die oben beschriebene Agenteninteraktion muß also auch die Unterstützung der *Evolution* der beteiligten Systeme sein.

2.2 Das Modell der *Business Conversations*

In den nun folgenden Abschnitten soll das Modell der *Business Conversations* ausführlich erklärt werden. In Abschnitt 2.2.1 werden die Entstehung des Modells und seine Wurzeln dargestellt. Die Grundlagen des Modells werden in Abschnitt 2.2.2 dargelegt; in den Abschnitten 2.2.3 und 2.2.4 werden die wesentlichen Aspekte der verschiedenen möglichen Kooperationsformen vorgestellt. In Abschnitt 2.2.5 wird dann eine formale Definition der zu spezifizierenden Kommunikation gegeben und deren Semantik weitgehend informell beschrieben; ferner werden die dynamischen Aspekte der Kommunikation geschildert. Schließlich wird in Abschnitt 2.2.6 eine Modellierung durch Petrinetze vorgestellt.

Die Darstellung lehnt sich dabei weitgehend an [Mat97a, Mat97b] sowie [Joh97, Ric97] an, der Abschnitt 2.2.1 orientiert sich auch an [GHS91].

2.2.1 Leitbild

Die Sprache dient nicht nur dazu, die Welt zu beschreiben. Diese Aussage enthält eine jener Wahrheiten, von denen WITTGENSTEIN sagt, daß sie „dem Bemerktwerden nur entgehen, weil sie ständig vor unseren Augen sind“ [Wit60]. Dennoch hat erst der englische Philosoph J.L. AUSTIN diese Erkenntnis in eine Theorie umgesetzt [Aus62].

Diesem „deskriptiven Fehlschluß“ setzt AUSTIN die Erkenntnis entgegen, daß es nicht nur Äußerungen gibt, die Tatsachen *feststellen*, sondern auch solche, die Tatsache *schaffen*. Dieser Aspekt bildet die Grundlage der von ihm entwickelten und von J.R. SEARLE fortgeführten [Sea69] *Sprechakttheorie*.

AUSTIN unterscheidet dazu im ersten Teil seiner Theorie zwischen *performativen* und *konstativen* Äußerungen. Zwar revidiert und verfeinert er im folgenden diese Distinktion, da sich die Unterscheidungen zwischen wahren und falschen konstativen Aussagen einerseits und dem Glücken bzw. Nicht-Glücken performativer Aussagen nicht aufrecht erhalten ließen (zudem ließen sich keine grammatischen oder lexikographischen Kriterien für diese Klassifizierung finden), als stark vereinfachte Grundlage für die Modellierung von Äußerungen innerhalb von Geschäftsprozessen ist sie aber brauchbar.

Die später von AUSTIN vorgenommene Differenzierung in *lokutionäre*, *illokutionäre* und *perlokutionäre* Sprechakte kritisiert SEARLE und schlägt eine Klassifikation in die folgenden fünf Arten von Sprechakten vor: *Assertive* Äußerungen (Behauptungen, Feststellungen, Beschreibungen), *direktive* Äußerungen (Befehle, Aufforderungen, Erlauben, Raten), *kommissive* Äußerungen (Versprechen, Ankündigungen, Drohungen), *expressive* Äußerungen (Dank, Gratulation, Entschuldigungen) sowie *deklarative* Äußerungen (Kriegserklärungen, Heiraten, Kündigungen). Nach SEARLE ist jede Äußerung dabei unmittelbar an ihre Bedeutung gekoppelt. Sprechakte werden daher als Mitteilung einer legalen Zustandsänderung ihres Sprechers innerhalb einer Konversation betrachtet.

Auf diesen Theorien aufbauend versuchten Arbeiten im Umfeld der computerunterstützten Gruppenarbeit (*computer supported cooperative work*, CSCW), Sprechakte, die im Rahmen von Geschäftsprozessen auftreten, zu identifizieren und zu klassifizieren. In [Win87] und [FGHW88] werden in dem sogenannten *language/action*-Ansatz die zwei dafür wesentlichen

Arten von Sprechakten genannt: Zum einen *Conversations for Possibilities* (CfP) und zum anderen *Conversations for Actions* (CfA). Der zentraler Gedanke dabei ist, daß zwei Partner zum Zwecke der Erledigung ihres gemeinsamen Anliegens eine Reihe von Sprechakten vollführen; mit anderen Worten: eine *Konversation führen*. Eine CfA ist demnach eine Folge von Aufforderungen und Versprechen, die direkt auf die gemeinsamen Aktivitäten und Ziele der Partner gerichtet ist. Jeder Sprechakt kann im Sinne einer *performativen* Äußerung eine Aktion der Partner repräsentieren. Dabei ist innerhalb einer CfA die Reihenfolge der einzelnen Akte, also die Menge der erlaubten Gesprächstransitionen, unter den Partnern genau festgelegt. Werden hingegen in einer Konversation keine Verpflichtungen eingegangen, sondern wird z.B. die Funktion eines Managers wahrgenommen, so wird von einer CfP gesprochen. In diesem Fall werden in einer Konversation lediglich Feststellungen, Meinungen oder Erklärungen ausgetauscht und bei keinem der Partner wird dadurch eine Aktion impliziert. Eine CfP oder CfA strukturiert und koordiniert einzelne Sprechakte. Durch die Ausführung dieser Sprechakte bilden die beteiligten Kommunikationspartner eine gemeinsame Historie, vor deren Hintergrund neue Sprechakte erzeugt und interpretiert werden.

Dem Kooperationsmodell der *Business Conversations* liegt die CfA zugrunde. Dabei vereinbaren die Partner wechselseitig künftige Aktionen, die direkt an die geäußerten Sprechakte der jeweiligen Partner gekoppelt sind.

2.2.2 Grundlagen des Modells

Das eben geschilderte Leitbild bildet die Basis für folgende Annahmen:

- Das Interaktionsmuster der Sprechakte ist *universell* in dem Sinne, daß es unabhängig von der Art der Akteure (Software oder Menschen) und den von ihnen verwendeten Kommunikationsmedien und -protokollen ist.
- Die Interaktion zwischen zwei menschlichen Akteuren, zwischen einem Menschen und einem maschinellen Akteur und auch zwischen zwei maschinellen Akteuren ist innerhalb des Modells transparent.

Das bedeutet, daß die schon zuvor geforderte *Medien- und Aktorenunabhängigkeit* gegeben ist.

Das Modell der *Business Conversations* geht von der Interaktion zwischen zwei Partnern aus, von denen der eine in der Rolle des *Kunden* und der andere in der des *Dienstleisters* handelt. Die Interaktion zwischen beiden Akteuren kann in einen Zyklus mit vier Phasen unterteilt werden (siehe Abbildung 2.1) und folgt stets diesem Verlauf:

1. Im ersten Schritt, der *Anfragephase*, fragt der Kunde unverbindlich beim Dienstleister nach einer Dienstleistung an. Die Phase endet mit der Annahme der Anfrage. Die Initiative geht dabei immer vom Kunden aus.
2. Mit Annahme der Anfrage beginnt die zweite, die *Übereinkunftsphase*. In ihr erfolgt die Verhandlung zwischen Kunde und Dienstleister über die genauen Modalitäten des zu erbringenden Dienstes. Am Ende steht die Übereinkunft.

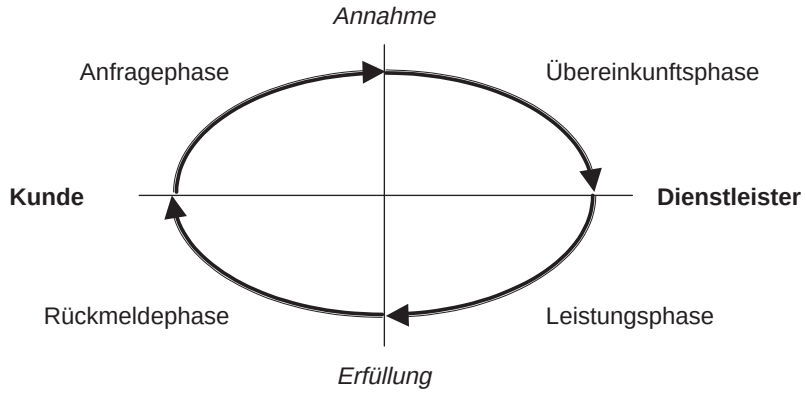


Abbildung 2.1: Unterteilung der Interaktionsphasen

3. Nach der Übereinkunft beginnt die *Leistungsphase*, in der der Dienstleister die vereinbarte Leistung erbringen muß. Sie endet aus Sicht des Dienstleisters mit der Erfüllung der Leistung.

4. Die vierte Phase, die *Rückmeldephase*, dient dem Kunden dazu, dem Dienstleister mitzuteilen, ob die zuvor erbrachte Leistung seinen Anforderungen und Erwartungen entspricht.

Nur wenn alle vier Phasen erfolgreich abgeschlossen wurden, haben Kunde und Dienstleister das gemeinsame Ziel erreicht. Die Ausführung von Sprechakten wird als *Konversation* bezeichnet und bezieht sich immer auf einen konkreten Gegenstand, nämlich die Dienstleistung oder Aktion.

Der Zyklus kann natürlich durch den Verzicht auf einzelne Phasen vereinfacht werden, wenn die zugrundeliegende Beziehung zwischen Kunde und Dienstleister dies erlaubt. Es können so z.B. die Phasen der Anfrage und der Übereinkunft zusammengefaßt werden, oder die Leistungsphase kann (implizit) durch das Zustandekommen der Übereinkunft bereits erbracht worden sein, oder es kann aus Gründen der Einfachheit auf die Rückmeldephase verzichtet werden. In allen Fällen bleibt jedoch sowohl die Reihenfolge der Phasen als auch der Zyklus als solcher erhalten.

Weiterhin steht jederzeit beiden Partnern die Möglichkeit offen, die begonnene Konversation abzubrechen, sei es durch Nichtannahme der Anfrage, Scheitern der Übereinkunft, Nichterbringen der Leistung oder weitere Gründe.

2.2.3 Modalitäten

In diesem Modell kann jedes Unternehmen, jeder Geschäftsbereich oder jeder geeignete Teil daraus als ein Akteur betrachtet werden, der gleichzeitig eine ganze Reihe von simultanen Konversationen mit anderen Akteuren wie Kunden, Lieferanten, Behörden oder anderen Teilen des Unternehmens führt. Innerhalb jeder einzelnen Konversation hat der Akteur die fest zugewiesene, nicht veränderliche Rolle entweder des Kunden oder des Dienstleisters.

Sofern eine formale Spezifikation der zu führenden Konversationen vorliegt, gestatten es die Medien- und inbesondere die Aktorenunabhängigkeit des Modells, Kooperationen in verschiedenen Modalitäten zu unterstützen.

Anwendungskoppelung: Wenn sowohl der Kunde als auch der Dienstleister als voneinander unabhängige Anwendungen (d.h. maschinelle Akteure) realisiert sind, ist die Kooperation z.B. über den Austausch von synchronen oder asynchronen Nachrichten möglich (*application linking*).

Benutzungsschnittstellen: Ein menschlicher Benutzer kommuniziert mit einem Softwaresystem, beispielsweise mit einem Informationssystem. Wenn die Interaktion formal als Konversation spezifiziert ist, kann ein generischer Dienst diese Beschreibung interpretieren und zum Aufbau einer evtl. graphischen Benutzungsschnittstelle nutzen. Durch Verwendung dieses *generischen Kunden* (*generic customer*) sind nur legale Interaktionen des Benutzers als Kunden mit dem System als Dienstleister möglich. Für den Dienstleister ist aber weder unmittelbar erkennbar noch nötig zu wissen, ob ein menschlicher oder ein maschineller Akteur sein Partner ist.

Workflow-Management: Der komplementäre Fall zum eben genannten ist, daß ein Softwaresystem in der Rolle eines Kunden Anfragen etc. an einen menschlichen Akteur sendet, der dann durch die Verwendung eines softwarebasierten *generischen Dienstleisters* die Anfragen interpretiert und ausführt. Ein Beispiel dafür wäre der Einsatz von Workflowsystemen.

Arbeitsstrukturierung: Die Zusammenarbeit von Menschen kann mit Hilfe von Softwarewerkzeugen, die die formalen Spezifikationen der Zusammenarbeit oder Kooperation kennen, durch Überprüfung der Einhaltung eben dieser Regeln unterstützt werden.

Basierend auf diesen Teilszenarien lassen sich komplexe Szenarien der Integration verschiedener vorhandener oder neu zu erstellender Dienste organisatorische Struktur von Unternehmensprozessen etc. entwerfen. Wesentliche Vorteile der relativ losen, autonomieerhaltenden Koppelung der Komponenten über *Business Conversations* sind, daß

- Teilsysteme zunächst unabhängig von der Umgebung nur auf Basis der Interaktionsmuster entworfen und realisiert werden können; wenn sie später in verteilten, kooperierenden Umgebungen eingesetzt werden sollen, ist es nicht nötig, die Anwendungslogik zu modifizieren,
- die Integration vorhandener Altsysteme durch Kapselung in sogenannte *wrapper*, die die Konversationen im obigen Sinne auf die Schnittstellen des Altsystems abbilden, leicht zu bewerkstelligen ist; dazu muß der Quellcode der Anwendungen oft nicht einmal vorhanden sein und
- die Modalität einzelner Interaktionsbeziehungen ohne Verlust entweder statisch durch Umorganisation der Systeme oder Prozesse oder sogar dynamisch während der Konversation geändert werden kann.

Die in Abschnitt 2.1 genannte wichtige Forderung nach Unterstützung der Evolution wird also durch dieses Modell in großem Umfang erfüllt.

2.2.4 Abstraktionen und Verfeinerungen

Gemäß den schon angesprochenen Strukturierungsprinzipien kann ein durch einen Akteur angebotener einzelner Dienst von außen als einheitlicher Dienst erscheinen, während der Akteur tatsächlich nur interne Konversationen zwischen weiteren Akteuren anstößt, die als Gemeinschaft den Dienst erbringen. Beispiele wären das Zusammenwirken von teilweise unabhängigen Geschäftsbereichen eines Unternehmens, oder die Gliederung des firmeninternen Teils eines Geschäftsprozesses nach den beteiligten Personen oder Rollen.

Die erste Konversation wird dabei als die *primäre,* alle weiteren, von dem ursprünglich angesprochenen Akteur initiierten, als die *sekundären* Konversationen bezeichnet. Sekundäre Konversationen folgen dabei genau demselben Schema wie primäre. Dadurch ist es möglich, daß ein Akteur zur selben Zeit in der Rolle des Dienstleisters und des Kunden agiert, allerdings in unterschiedlichen Konversationen.

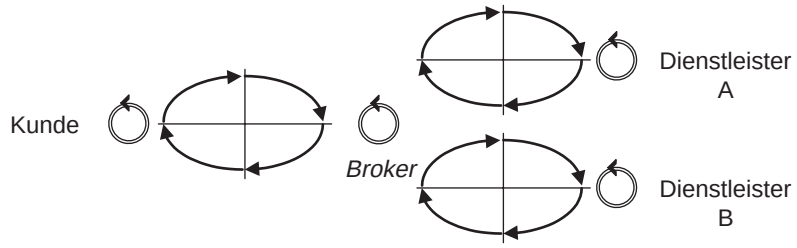


Abbildung 2.2: Delegation von Konversationen durch einen *Broker*

Wie in Abbildung 2.2 dargestellt, kann ein Akteur als *Broker* eine (primäre) Konversation in der Rolle des Dienstleisters mit einem Kunden führen und dazu gleichzeitig zur Erledigung seines Auftrages in der Rolle des Kunden mehrere sekundäre Konversationen mit weiteren Akteuren führen. In diesem Fall kann man auch von *Delegation* sprechen, denn der *Broker* muß zur Erfüllung seiner Aufgabe die primäre Konversation zunächst aufschieben, um untergeordneten Stellen Anfragen stellen zu können, deren erfolgreiche Ausführung erst die Ausführung der primären ermöglicht.

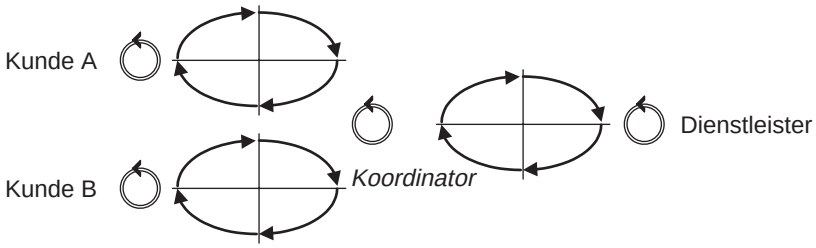


Abbildung 2.3: Koordinierung von Konversationen durch einen Koordinator

Andererseits kann ebenso ein Akteur als Dienstleister für mehrere Kunden fungieren (also mehrere primäre Konversationen führen) und gleichzeitig Kunde in einer sekundären Konversation sein (siehe Abbildung 2.3). Dies ist der Fall der *Koordinierung* der Arbeit mehrerer Kunden in Bezug auf einen Dienstleister.

Zu unterscheiden sind synchrone von asynchronen sekundären Konversationen. Synchrone sekundäre Konversationen unterbrechen die initiierende primäre Konversation, während asynchrone parallel zur primären ablaufen.

Zusätzlich sind natürlich noch komplexere Verfeinerungen denkbar, in denen ein Akteur gleichzeitig mehrere primäre und mehrere sekundäre Konversationen führt und synchrone und asynchrone gemischt auftreten. Dabei kann die erfolgreiche Beendigung einer neuen asynchronen sekundären Konversation die Vorbedingung für einen Schritt der parallel ablaufenden primären sein, so daß sich insgesamt Abhängigkeiten auch zwischen asynchronen Konversationen bilden. Eine Modellierung mit Hilfe von Petrinetzen bietet sich dafür an, ist aber nicht Teil des Modells der *Business Conversations*.

In den Darstellungen von [Joh97, Ric97] wird außerdem die *Subkonversation* als Mittel zur Strukturierung von Konversationen eingeführt: Eine Konversationsspezifikation darf anstelle eines einzelnen Gesprächsschrittes eine weitere, ganze Konversationsspezifikation als *Subkonversation* referenzieren, die an dieser Stelle den weiteren Ablauf festlegt. Dadurch soll die Wiederverwendung von Spezifikationen erleichtert werden. Die operationale Semantik dieses Konzepts wird allerdings nicht klar dargestellt, zumal es auch in der begleitenden Implementierung nicht realisiert wurde. Die Technik der Subkonversation ähnelt der der synchronen sekundären Konversationen stark. Der Unterschied liegt in der expliziten Modellierung der Subkonversation gegenüber der transparent für den Partner ablaufenden sekundären Konversation.

Die Nutzung von Subkonversationen verspricht die inkrementelle und auf der Wiederverwendung von bereits bestehenden Diensten basierende Entwicklung von neuen und die Weiterentwicklung von alten Diensten zu verbessern. Damit entspricht sie der Forderung nach Unterstützung der Evolution von kooperativen Informationssystemen. In dieser Arbeit wird in Abschnitt 4.5.1 eine Definition der genauen Semantik der Subkonversationen vorgenommen.

Die genannten technischen Prinzipien lassen sich auch zur semantischen Strukturierung und Modellierung eines nach außen hin geschlossen erscheinenden Systems nutzen: Tatsächlich können mehrere Akteure in gleichberechtigten Positionen ihre Arbeit aufteilen, oder sie können eine hierarchische Struktur in der Art von Vorgesetzten und Untergebenen unter Verwendung von Befehlsketten bilden.

In diesem Sinne kann man die Fähigkeit, an einer solchen Konversation teilzunehmen, als die charakteristische Fähigkeit eines *Agenten* sehen [Mat97a]. Wenn man dies tut, kann man weiterhin sowohl ein ganzes Unternehmen, einzelne Geschäftsbereiche daraus, Softwaresysteme oder einzelne Personen bzw. Rollen, die sie innehaben, als Spezialisierung des generellen Konzepts des Akteurs betrachten. So gesehen, liefert die wiederholte Dekomposition der Kunde-Dienstleister-Beziehungen eines großen Systems oder eines Unternehmens zuletzt eine Ansammlung von autonomen Agenten, die menschliche oder maschinelle Akteure repräsentieren und nur noch durch ein Geflecht von Konversationsbeziehungen verbunden sind. Die sich damit eröffnenden Möglichkeiten der Systemanalyse und des Entwurfs insbesondere von Geschäftsprozessen werden näher in [Rip98] untersucht.

2.2.5 Konversationsspezifikationen

Um eine Brücke zwischen diesen abstrakten Modellen einerseits und den softwaretechnischen Realisierungsmöglichkeiten andererseits schlagen zu können, ist eine Formalisierung des eigentlichen Inhalts der Konversationen und deren genauen Ablaufs nötig.

Wir verwenden im folgenden zwei getrennte Beschreibungsebenen für Konversationen:

- Die Ebene der *Konversationsspezifikationen*: Auf dieser Ebene wird die Struktur von möglichen Konversationen beschrieben. Sie entspricht der Typebene einer Programmiersprache. Diese Spezifikationen müssen eine Laufzeitrepräsentation besitzen.
- Die Ebene der *Konversationsinstanzen*: Dies ist die der konkreten Konversationen, die zwischen zwei konkreten Akteuren in den Rollen von Kunden und Dienstleistern stattfinden. Analog zu Programmiersprachen ist jede Konversationsinstanz Ausprägung ihres Typs, der Konversationsspezifikation.

Das Modell der *Business Conversations* sieht die Gliederung jeder Konversation in einzelne *Dialogschritte* vor, wobei nach der Eröffnung der Konversation durch eine initiale Anfrage des Kunden in jedem Schritt ein formal spezifizierter *Dialog*, den man als Abstraktion eines Formulars sehen kann, vom Dienstleister an den Kunden geschickt wird, der Kunde diesen inspiziert und den Inhalt ggf. verändert und ihn schließlich unter Angabe einer aus einer Menge von für diesen Dialog angegebenen *Anfragen* zurückschickt. Dies geschieht solange, bis beide Partner übereinkommen, daß die Konversation beendet ist. Das Prinzip wird in Abbildung 2.4 deutlich; der grau hinterlegte Bereich wird dabei evtl. vielfach wiederholt.

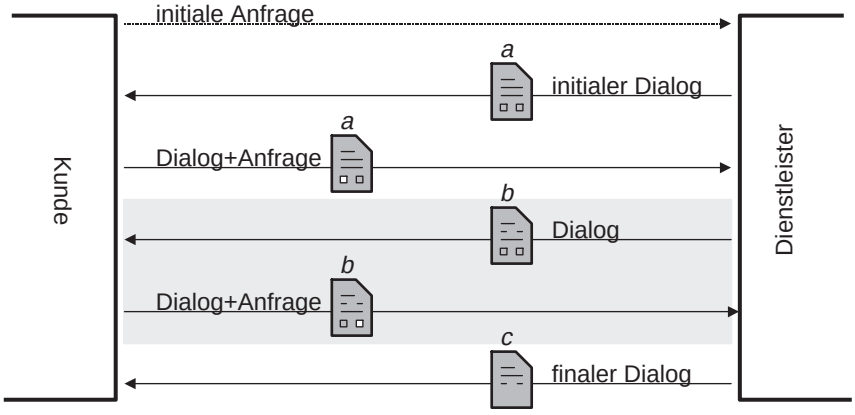


Abbildung 2.4: Ablauf einer Konversation

Formal kann man diese Transitionen folgendermaßen als Funktionen notieren:

$$\begin{aligned} \text{Dienstleister:} \quad (d_n, r_n) &\mapsto d_{n+1} \in r_n \\ \text{Kunde:} \quad d_n &\mapsto (d'_n, r_n) \end{aligned}$$

Die d_i stellen dabei die Dialoge und die r_i die Anfragen dar, wobei i der Index des Dialogschritts ist. Eine Anfrage r besteht formal aus der Menge der möglichen Folgedialoge.

Damit eine aktorenunabhängige Kommunikation möglich ist, bedarf es zu jeder konkreten Konversation einer abstrakten *Konversationsspezifikation*. Sie fungiert – in Analogie zu Programmiersprachen – als Typ einer konkreten Instanz einer Konversation. Diese genaue Spezifikation erst ermöglicht es den Partnern, den Kontext der Konversation zu erfassen und geeignet auf Nachrichten zu reagieren. Insbesondere sind erst dadurch maschinelle Akteure möglich.

Eine Konversationsspezifikation definiert im einzelnen folgendes:

- ihren Namen;
- alle in einer Konversation möglichen Dialoge, dazu erhält jeder Dialog einen innerhalb der Konversationsspezifikation eindeutigen Namen sowie eine formale, strukturierte Beschreibung des Inhalts durch ein simples Typsystem;
- eine Menge von Anfragen für jeden Dialog, aus der eine als Antwort des Kunden auf den Dialog (neben seinem Inhalt selbst) gefordert wird; jede Anfrage hat dazu einen innerhalb des Dialoges eindeutigen Namen;
- alle direkt auf einen Dialog als Folgedialog möglichen Dialoge, dazu wird zu jeder Anfrage, die ein Dialog enthält, die Menge der erlaubten Folgedialoge (durch Angabe der Namen) angegeben;
- den initialen Dialog, den der Dienstleister als ersten Dialog direkt auf die initiale Anfrage eines Kunden auf Eröffnung einer Konversation schickt.

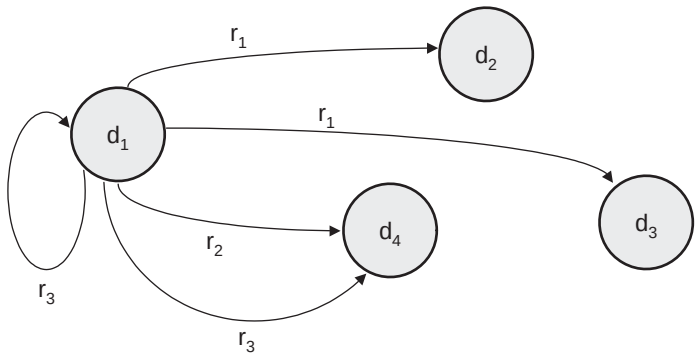


Abbildung 2.5: Konversationsspezifikation als nichtdeterministischer endlicher Automat

Abgesehen von dem Typsystem für die Dialoginhalte läßt sich eine Konversationsspezifikation abstrakt als Spezifikation eines nichtdeterministischen endlichen Automaten (NFA) und der Ablauf einer Konversation als Interpretation oder Simulation desselben betrachten. Dabei bilden die Dialoge die Zustände und die Anfragen des Kunden die Eingaben des Automaten, wobei der Nichtdeterminismus des Automaten durch die Möglichkeit mehrerer Folgedialoge eines Dialoges auf eine Anfrage zum Ausdruck kommt (siehe Abbildung 2.5). Die Zustandsübergänge werden also im Rahmen der Spezifikation durch die Auswahl der Anfrage durch den Kunden und die Auswahl des Folgedialoges durch den Dienstleister festgelegt.

Überdies existieren in jeder Konversationsspezifikation zwei weitere Anfragen, die nicht Teil der expliziten Definition sind:

1. Die *initiale Anfrage*, die von einem Kunden zwecks Eröffnung einer Konversation an einen Dienstleister gerichtet wird. Sie ist nicht Bestandteil der Spezifikation, da die Konversation, zu der die Anfrage führen soll, zum Zeitpunkt der Anfrage noch nicht existiert. Die initiale Anfrage muß eine geeignete Repräsentation der Spezifikation der zu eröffnenden Konversation enthalten, damit der Dienstleister überprüfen kann, ob er in der Lage ist, die Konversation zu führen. Die Art der Repräsentation bleibt der Implementierung überlassen.

2. Die spezielle Anfrage nach dem *Abbruch der Konversation*. Sie ist implizit Bestandteil jedes Dialogs und kann deshalb jederzeit vom Kunden gestellt werden. Mit ihrer Hilfe lassen sich nicht behebbare Fehler sowohl in der Anwendung als auch solche des Systems signalisieren. Das System könnte beispielsweise auch nach Zeitüberschreitungen (*timeout*) bei der Antwort eines Partner die Konversation abbrechen und dies damit signalisieren. Auch Kunde oder Dienstleister können damit den Abbruch der Konversation erzwingen. Der Implementierung des Systems bleibt die Spezifikation eines speziellen *Fehlerdialoges* vorbehalten, der die Gründe für das Scheitern der Konversation beschreibt und im Falle des Absendens der Anfrage nach Abbruch statt des eigentlich erwarteten Dialoges vom Kunden mitgesendet bzw. vom Dienstleister als Zeichen des Abbruchs der Konversation verschickt wird.

Eine besondere Rolle spielen weiterhin die *finalen Dialoge*. Definitionsgemäß sind dies die Dialoge, die eine leere Menge von Anfragen (also keine Anfragen) enthalten. Das Fehlen von Anfragen bedeutet, daß es keine Folgedialoge gibt, also die Konversation beendet ist, wenn ein solcher Dialog versendet wird. Jede Konversationsspezifikation kann keinen, einen oder mehrere finale Dialoge enthalten. Wenn es keinen gibt, wird dadurch eine unendliche Konversation modelliert. Mehrere finale Dialoge stellen verschiedene Endzustände der Konversation dar. Der oben genannte Fehlerdialog stellt einen speziellen finalen Dialog dar.

Der finale Ablauf ist folgender: Sendet der Dienstleister einen finalen Dialog an den Kunden, so ist für den Dienstleister in genau diesem Moment die Konversation beendet. Nach Erhalt des finalen Dialoges durch den Kunden und die entsprechende Auswertung seinerseits ist die Konversation auch für ihn beendet.

Der *initiale Dialog* als ein weiterer ausgezeichneter Dialog ist von der Anforderung an die Struktur her nicht anders als andere normale Dialoge, kann also auch final sein (womit die Konversation sofort nach Beginn wieder beendet wäre).

Die Spezifikation des Inhalts eines Dialoges wird durch ein einfaches Typsystem geleistet. Es sieht neben Basisdatentypen wie `boolean`, `integer`, `real`, `string`, `date` und `currency` zusammengesetzte Typen wie `records`, `variants`, `single-` und `multiple-choice` sowie `sequences` vor. Dabei werden alle Komponenten des Inhalts sowie alle `record-` und `variant`-Komponenten durch Namen gekennzeichnet, um den Zugriff zu ermöglichen. Die `record-` und `variant`-Typen dienen zum Aufbau von komplexeren, strukturierten Typen. Dabei sind alle Typen orthogonal kombinierbar. Die `choice`-Typen, deren Instanzen sich wertmäßig nicht von den entsprechenden Typen unterscheiden, dienen dazu, die

mögliche Wertemenge einzuschränken. Die Wertemengen sind nicht Teil der Spezifikation, sondern werden erst zur Laufzeit den Instanzen zugewiesen. Der Gedanke ist dabei, mit der Bereitstellung solcher Typen auch die Visualisierung von Datenstrukturen zu unterstützen, etwa (je nach Kardinalität) durch *checkboxes, radiobuttons* oder geeignete Auswahllisten. Der `sequence`-Typ dient zur Definition und Benutzung von Massendaten.

Der Aufbau einer vollständigen Konversationsspezifikation läßt sich durch Angabe einer Grammatik formalisieren. Wir geben dazu hier eine solche in EBNF an:

Conversation	:=	**CONVERSATION** Name **WITH** Dialog {Dialog};
Dialog	:=	**DIALOG** DialogName **WITH** {Request} [Content] **END**;
Request	:=	**REQUEST** RequestName **TO** DialogName {DialogName} **END**;
Content	:=	**CONTENT** NamedType {NamedType} **END**;
NamedType	:=	Name **OF** Type;
Type	:=	AtomicType | CompoundType | SpecType;
SpecType	:=	**CONVSPEC** | **DIALOGSPEC** | **CONTENTSPEC**;
AtomicType	:=	**BOOL** | **INT** | **REAL** | **STRING** | **DATE** | **CURRENCY**;
CompoundType	:=	**RECORD** {NamedType} **END** |
		VARIANT {NamedType} **END** |
		MULTIPLE_CHOICE OF Type |
		SINGLE_CHOICE OF Type |
		SEQUENCE OF Type;

Konkrete Konversationsspezifikationen müssen selbstverständlich nicht in einer der obigen EBNF entsprechenden textuellen Form angegeben werden; diese EBNF dient nur dazu, die Struktur als solche zweifelsfrei festzulegen. Die Art der Konstruktion von Spezifikationen und deren Instanziierung für konkrete Konversationen bleiben den konkreten Mechanismen der Implementierung überlassen.

Die zuvor bereits genannte Forderung nach Vorhandensein einer Laufzeitrepräsentation der Konversationsspezifikation ist nicht nur für die initiale Anfrage nötig, damit die Partner die Spezifikation inspizieren können, sondern kann auch zur Entwicklung von Meta-Diensten verwendet werden, die auf Spezifikationen arbeiten: Denkbar sind graphische Editoren, generische Visualisierungswerkzeuge, Modellverifikatoren, Spezifikationsmakler etc. Ein wichtiger Nutzen ist auch die Überprüfbarkeit von Konversationen zur Ausführungszeit, um eine gesicherte maschinelle Abwicklung der Konversation zu gewährleisten.

Durch die Einführung der Typen `CONVSPEC`, `DIALOGSPEC` und `CONTENTSPEC` sind alle Spezifikationen selbst Objekte erster Klasse in ihrem eigenen Typsystem, nicht nur im Typsystem der Implementierungssprache. Dadurch können Spezifikationen wiederum Inhalte sein, die innerhalb von Konversationen verwendet werden können. So wird es besonders einfach, die oben angesprochenen Meta-Dienste zu realisieren, wenn diese selbst *Business Conversations* zur Interaktion verwenden.

2.2.6 Modellierung durch Petrinetze

Abschließend soll hier kurz eine weitere interessante Möglichkeit für die Darstellung des Ablaufes der Interaktion nach der *language/action*-Perspektive gegeben werden, die [Rip98] entnommen ist.

Dabei werden, wie Abbildung 2.6 zu entnehmen ist, der Kunde, der Dienstleister und der Konversationszyklus in einem gemeinsamen Petrinetz beschrieben, das den Ablauf durch geeignete Vorbedingungen einschränkt. Der stark durchgezogene, graue Kreis symbolisiert dabei lediglich den Konversationszyklus; er hat keine Bedeutung für das Netz.

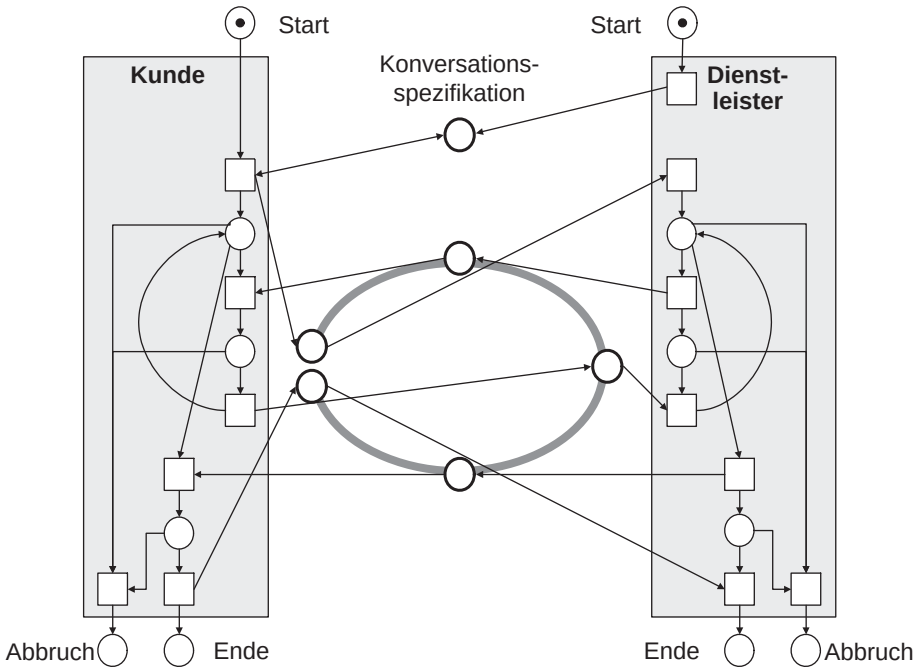


Abbildung 2.6: Petrinetz zur Darstellung der *language/action*-Perspektive

Die markierten Stellen stellen den Ausgangszustand dar. Die rechte modelliert die Bereitstellung der Konversationsspezifikation durch den Dienstleister. Wenn dies geschehen ist, kann ein Kunde den Dienst in Anspruch nehmen. Dabei können auch mehrere Kunden gleichzeitig agieren, dies wäre durch mehrere Instanzen der linken Seite zu modellieren. Die Steuerung der Zyklus geschieht durch die in der Mitte angeordneten Stellen. Sie sind Ein- und Ausgangsstellen der entsprechenden Transitionen, die die Schritte von Kunde und Dienstleister darstellen. Gut zu erkennen ist auch die Möglichkeit, eine begonnene Konversation jederzeit abzubrechen. Dazu dienen die Transitionen, die zu den mit „Abbruch“ bezeichneten Stellen führen.

Dieses Netz kann als Grundlage für die Modellierung der *Business Conversation* in Workflow-Systemen dienen, die Gebrauch von Petrinetzen machen.

2.3 Agenten

Wie bereits anfangs gesagt, fasziniert der Gedanke von *Agenten*, die durch die Vorgabe von Zielen und die Benutzung einer Wissensbasis autonom eine Aufgabe erledigen, die Forschungsgemeinde seit geraumer Zeit.

Dabei ist allerdings zu beobachten, daß der zugrundeliegende Begriff des *Agenten* alles andere als eindeutig ist: Viele (sich auch widersprechende) Definitionen sind in Gebrauch, und es wird fast alles, was ein Mindestmaß an Autonomie aufweist, als Agent bezeichnet.

In diesem Abschnitt soll versucht werden, den dieser Arbeit zugrundeliegenden Agentenbegriff gegenüber anderen abzugrenzen und einen Vergleich anzustellen. In [Joh97] werden bereits die Systeme TELESCRIPT, FACILE, MOLE und COSY sowie TCL / MIME-Agenten untersucht und verglichen. Diese sollen hier daher nicht weiter betrachtet werden. Mittlerweile sind eine Vielzahl von neuen Agentensystemen vorgestellt worden. Genannt seien hier nur – ohne Anspruch auf Vollständigkeit – die IBM AGLETS, CONCORDIA, ODYSSEY (der Nachfolger von TELESCRIPT) und VOYAGER, die alle auf JAVA basieren. Neben diesen kommerziellen Entwicklungen gibt es eine noch größere Zahl von akademischen Projekten. Einen guten Überblick und einen Vergleich liefert etwa [Cha97]. Weiterhin liegen Entwürfe zur Standardisierung gewisser Aspekte von Agentensystemen und ihrer Interoperabilität seitens der OMG (*Objekt Management Group*) vor [CGG+97].

Als für uns interessante neuere Entwicklung im Bereich *objektorientierter* mobiler Agenten stellen sich die AGLETS der IBM CORPORATION [LC96, OK97] und das JAFMAS-System, das an der UNIVERSITY OF CINCINNATI entwickelt wurde [Cha97], dar, die beide eine Umgebung für mobile Agenten auf Basis von JAVA bieten. Interessant sind sie deshalb, weil beide wegen der Realisierung mit JAVA ähnliche Eigenschaften und Möglichkeiten erwarten lassen wie die hier vorgenommene Implementierung, sowie eines dem mehr kommerziellen und das andere mehr dem akademische Lager zuzuordnen ist.

Nach einer einführenden Klassifikation und der Aufstellung genereller Merkmale mobiler Agenten in Abschnitt 2.3.1 werden in Abschnitt 2.3.2 die AGLETS, in Abschnitt 2.3.3 das JAFMAS-System und in Abschnitt 2.3.4 die OMG-Spezifikation vorgestellt. Abschließend wird in Abschnitt 2.3.5 eine Bewertung vorgenommen.

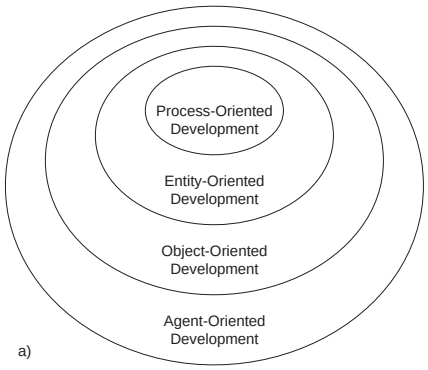
2.3.1 Klassifikation

In [Joh97] wird eine Klassifikation der verschiedenen Konzepte für Agenten vorgenommen. Dabei wird unterschieden zwischen *User Agents*, Leitagenten, autonomen Agenten, symbiotischen, kooperativen Agenten, anthropomorphen Agenten und mobilen Agenten. Die dort angegebene und zugrundegelegte Definition eines Agenten ist [BW94] entnommen:

> Ein Agent kann als eine relativ einfache, heterogene, autonome und kommunizierende Software-Komponente definiert werden; viele dieser Agenten existieren gleichzeitig und arbeiten zusammen, um eine Aufgabe für einen Benutzer zu erfüllen.

Darüber hinaus gehören die dort betrachteten Agenten zur o.g. Klasse der *mobilen* Agenten. Häufig trifft man auch auf den Begriff *aktives Objekt* für Agenten. Eine andere Aufstellung weiterer Arten von Agenten und ihrer Eigenschaften findet sich in [Cha97].

Historisch gesehen wird die Entwicklung des Agenten-Paradigmas als logisch folgerichtiger, jüngster Schritt in der Evolution der Konzepte und Programmiersprachen gesehen, der auf die Objektorientierung folgt. Die interessanten, [Cha97] entnommenen und leicht veränderten Abbildungen 2.7a und 2.7b illustrieren diese Aspekte. Als Sprachen der sechsten Generation stellen sich dort die die Entwicklung von Agenten unterstützenden heraus. Als erster Vertreter ist hier natürlich JAVA zu nennen.



b)

1. GL	binary data types	octal, hexadecimal, ...
2. GL	basic data types	real, integer, character, ...
3. GL	structured data types	strings, arrays, records, ...
4. GL	abstract data types	queues, sets, lists, trees, ...
5. GL	objects	windows, buttons, toolbars, ...
6. GL	autonomous agents	radar station, internet shop, ...

Abbildung 2.7: Die Entwicklung von Konzepten und Sprachen

Trotz der vielen unterschiedlichen Konzepte für Agenten findet sich „im Mittel“ eine gemeinsame Schnittmenge von Eigenschaften. Als wesentliche Eigenschaften, die ein Agent besitzen muß, sind also im Rahmen der hier angestellten Betrachtungen die folgenden zu fordern:

Mobilität: Migriert ein mobiler Agent von einem Ort zu einem anderen, so nimmt er dabei seinen gesamten Zustand, evtl. sogar seinen Ausführungszustand (Stapel, Programmzähler etc.) sowie alle von ihm benötigten oder referenzierten Objekte mit.

Autonomie: Dieser Aspekt hat zwei Facetten: Einerseits bedeutet die Autonomie, daß ein Agent genügend Wissen oder Informationen besitzt, um selbständig seine Mobilität nutzen zu können und zu entscheiden, wann und wohin er migrieren will. Um auch *faktisch* mobil zu sein, muß ein Agent andererseits möglichst große technische Autonomie bewahren können. Er darf nur die unbedingt nötigen Bindungen an das ihn beherbergende Agentensystem aufbauen, und außerdem darf das System keine direkten Bindungen an den Agenten weitergeben, da beides die Mobilität einschränkt.

Kommunikation: Die lokale Interaktion eines Agenten an seinem jeweiligen Aufenthaltsort mit anderen Agenten und dem Agentensystem wird direkt durch seine Autonomie bestimmt: Um die Autonomie weitgehend zu erhalten, sind spezielle Formen der Kommunikation erforderlich. Bei genauerer Betrachtung stellt sich die Kommunikation sogar als entscheidender Punkt für die Verwirklichung von Interoperabilität zwischen Agenten und Agentensystemen heraus: Nur wenn die Agenten eine „gemeinsame Sprache“ sprechen, ist sie möglich.

Koordination: Neben dem mehr technisch geprägten Aspekt der Kommunikation bedarf es zur anwendungsgerechten Modellierung der Aufgaben verteilter Agenten der *Koordination* der Interaktion. Dies bedeutet, daß die zur Erreichung des gesetzen Zieles ausgetauschten Nachrichten einem Schema entsprechen müssen, das die Partner nicht in unklaren über die verfolgten Ziele und den Stand der Bemühungen läßt. Die *Sprechakttheorie* hat für das Verständnis davon wertvolle Beiträge geleistet.

Asynchronizität: Die Asynchronizität ist eine Folge der Forderung nach Autonomie: Jeder Agent wird von einem eigenen Prozeß (evtl. *thread*) ausgeführt, so daß er zum einen (relativ) unabhängig von der Umgebung seinen Aufgaben nachgehen kann und zum anderen auch die Arbeit mehrerer Agenten gleichzeitig möglich ist. Durch Erfüllung dieser Forderung sind auf einfache Weise Kommunikationsformen realisierbar, die ein Maximum an Autonomie gewährleisten. Weiterhin kann man auch die Möglichkeit, mehrere gleichartige Agenten an verschiedenen Orten arbeiten zu lassen sowie die Fähigkeit eines Agenten, weiterzuarbeiten, wenn er physisch von seinem Ursprungsort getrennt ist (getrennte Netzwerkverbindung, mobiler Computer etc.), als Erscheinungsformen der Asynchronizität betrachten.

Benennbarkeit: Ohne Meta-Dienste, die die Agenten, Agentensysteme und die von Agenten angebotenen Dienste aufzufinden helfen, sind verteilte Agentenszenarien nicht sinnvoll realisierbar und erst recht nicht kommerziell erfolgreich. Die Unterstützung und Teilnahme an weit verfügbaren Namensdiensten (*naming services*) sind daher sehr wichtige Aufgaben jedes Agentensystems.

Sicherheit: Dem Modell mobiler Agenten inhärent ist die Möglichkeit des Mißbrauchs eines Agenten als Virus, trojanisches Pferd etc. Systeme, die wirklich den Empfang und die Ausführung beliebiger mobiler Agenten gestatten, müssen deshalb Sicherheitskomponenten vorsehen, die verschiedene, abgestufte Rechte der Agenten vorsehen, um die Ausführung verschiedener Funktionen kontrollieren zu können und um Mißbrauch aufzudecken, ferner sollten sie adäquate Mechanismen zur digitalen Autorisierung und Authentisierung bereithalten. Weiterhin müssen die Belange der Verschlüsselung und der Vertraulichkeit gesichert sein.

Gerade die letzten beiden Punkte erfahren in jüngster Zeit bedingt durch die zunehmende Kommerzialisierung des Internet und die Annäherung der Agententechnologie an das Internet erhöhtes Interesse, nachdem ihnen in den eher akademisch geprägten Forschungsansätzen zuvor weniger Beachtung geschenkt wurde.

Neben diesen oben aufgeführten Punkten muß das Agentensystem selbst, also der Teil, der stationär und immobil ist und zur Beherbergung der Agenten dient, eine Reihe von Leistungen erbringen können:

- Die Ablaufumgebung für die mobilen Agenten muß den oben aufgestellten Forderungen genügen.
- Eine geeignete Programmierschnittstelle zur Erzeugung und Verwaltung der Agenten ist notwendig.

- Häufig stellt sich eine aus Sicht der Agenten logische Gliederung des Agentensystems in *Orte* (*places*), die strukturierte Aufenthaltsräume für Agenten bieten, als vorteilhaft heraus. Dies ist sowohl mit der Migration als auch der Benennbarkeit eng verknüpft.

Alle Punkte zusammen stellen hohe Anforderungen an die zur Realisierung verwendeten Sprachen und Systeme. Daher überrascht es nicht, daß mittlerweile viele Agentensysteme JAVA verwenden, da dies vielen der Forderungen bereits weit entgegenkommt. Zu nennen sind hier besonders die Punkte Plattformunabhängigkeit, Kommunikation, Sicherheit sowie Nebenläufigkeit.

2.3.2 Die IBM AGLETS WORKBENCH

Die IBM AGLETS WORKBENCH, oder im folgenden kurz AGLETS, stellt eine visuelle Umgebung für den Bau, die Suche, den Zugriff und die Verwaltung von Geschäftsdaten und anderen Informationen durch mobile Agenten dar [LC96]. Dabei werden die oben aufgestellten Forderungen an ein System für mobile Agenten weitgehend erfüllt.

Die Realisierung geschieht durch ein *framework* von JAVA-Klassen; als wichtigste ist dabei die Klasse `Aglet` zu nennen. Die Instanzen der von ihr abgeleiteten Klassen stellen die Einheiten der Mobilität dar, d.h. die Agenten. Die Mobilität wird dabei unter Verwendung der JAVA-Objekt-Serialisierung durch ein ATP (*Agent Transfer Protocol*) genanntes Protokoll erreicht. Es ist an HTTP angelehnt und bietet die Möglichkeiten, Agenten zu verschicken (*dispatch*), wieder zurückzuholen (*retract*), allgemeine Klassen zu laden (*fetch*) sowie weitere Nachrichten auszutauschen (*message*). Erwähnenswert ist das Konzept der *codebase*, das die Definition eines Ortes (bzw. *servers*) umfaßt, der sämtliche von einem Agenten benötigten Klassen bereithält. Diese *codebase* wird beim erstmaligen Erzeugen eines Agenten angegeben und kann nicht mehr geändert werden. Dieses Konzept ist wichtig, weil durch die Verwendung der Objektserialisierung nur eine Pseudo-Mobilität erreicht wird, da ein migrierender Agent am Ende einer Reise vom Zielsystem tatsächlich ganz neu instantiiert werden muß. Die Schnittstelle des `Aglet` umfaßt Methoden, die zu genau definierten Zeitpunkten aufgerufen werden: So etwa `onCreation()` bei der Erzeugung, `run()` bei Erzeugung, Duplizierung, Versand, Ankunft etc. und `onDisposing()` beim Zerstören des Agenten. Zur besseren Handhabbarkeit der Migration ist die Möglichkeit der Anmeldung selbstdefinierter `MobilityListener` bei einem Agenten vorhanden; die in Subklassen zu überschreibenden Methoden `onDispatching()`, `onReverting()` und `onArrival()` lassen gezielte Maßnahmen bei der Migration zu. Unterstützt wird die Mobilität ferner durch ein *travel itinerary* genanntes Konzept, das es gestattet, komplexe Muster von Reisen an viele verschiedene Orte nacheinander auszuführen. Dazu wird eine automatische Fehlerbehandlung angeboten.

Die Autonomie eines Agenten wird durch ein Stellvertreter-Konzept (*proxy*) verbessert. Das Agentensystem gibt dabei keine direkten Referenzen auf die Agenten-Instanzen heraus, sondern nur an ein sogenanntes `AgletProxy`, das die Kommunikation mit dem eigentlichen Agenten regelt. Bei Migration des Agenten ist es auch dafür zuständig, Anfragen an den migrierten Agenten weiterzuleiten (Migrationstransparenz). Nebenbei schirmt es den Agenten durch enge Koppelung an die Sicherheitsmechanismen des Systems vor Angriffen ab.

Die Kommunikation der Agenten wird durch eine Reihe von Schnittstellen wirkungsvoll unterstützt: Neben einem *whiteboard*-Mechanismus, der die asynchrone Kooperation und den Informationsaustausch zwischen mehreren Agenten gestattet, existiert ein Nachrichtensystem, das sowohl die lose Koppelung durch asynchronen Nachrichtenaustausch als auch die synchrone *peer-to-peer*-Kommunikation zwischen Agenten ermöglicht. Die Methode `handleMessage()` der Klasse `Aglet` dient dabei dazu, Nachrichten jeglicher Art zu empfangen. Die Verwaltung der Nachrichten, der zugehörigen Warteschlangen und die nötigen Synchronisierungsmaßnahmen obliegen dabei dem Agentensystem. Zur Verfügung stehen weiterhin eine Reihe von Möglichkeiten, Nachrichten an andere Agenten zu versenden (*Now-Type*, *Future-Type*, *Oneway-Type* etc.), die unterschiedliche Arten der Synchronisierung erlauben, sowie vorgefertigte Klassen, die *Muster* bieten, die den Entwurf kommunizierender Systeme vereinfachen und standardisieren. Durch das Vorhandensein der `AgletProxys` kann so auch einfach mit entfernten Agenten kommuniziert werden. Als weitere Form der Kommunikation steht einem Agenten natürlich die gesamte weitere JAVA-Infrastruktur zur Verfügung; genannt seien hier nur JDBC (*java database connectivity*) und RMI (*remote method invocation*).

Die AGLETS stellen keinerlei explizite Mechanismen und keine Modelle zur Koordination der Aktivitäten der Agenten bereit.

Die Asynchronizität der AGLET-Agenten ergibt sich aus der Tatsache, daß die oben erwähnten Methoden (`run()` etc.) sowie die Methoden der `MobilityListener` vom Agentensystem zu klar definierten Zeitpunkten aufgrund klar definierter Ereignisse aufgerufen werden. In diesem Sinne besitzen `Aglets` also keinen separaten Kontrollfluß in Form eines eigenen *threads*. Durch die Kapselung eigener Daten in der Klasse des `Aglets` ist aber dennoch das Vorhandensein eines eigenen Kontextes gesichert. Zudem soll durch eine nicht näher spezifizierte Anzahl von System-*threads* die effiziente, parallele Arbeit der `Aglets` gewährleistet sein.

Der erforderlichen Benennbarkeit von Agenten und Agentensystemen wird durch das vom ATP verwendete Namensschema, das an die im Internet verwendeten URL (*uniform resource locator*) angelehnt ist, Rechnung getragen. Eine Adresse eines Agentensystems könnte dann z.B. `atp://aglets.trl.ibm.com:434/test` lauten.

Dem Sicherheitsaspekt schließlich wird durch das Vorhandensein eines recht umfangreichen Sicherheitssystems Rechnung getragen. Es bietet drei Schichten an: Die erste wird durch das in JAVA ohnehin vorhandene Sicherheitssystem gebildet (z.B. den *java bytecode verifier*), die zweite durch den *security manager*, der es dem Betreiber des Agentensystems gestattet, verschiedenen Agenten Rechte einzuräumen, Betriebsmittel zu nutzen (z.B. für *filesystem*, *network*, Teile des Agentensystems etc.), je nachdem, ob und wieweit ihm – oder genauer, dem Erzeuger – vertraut wird. Die dritte Schicht stellt eine Schnittstelle für Dienste zur Nutzung von Kryptographie, digitalen Unterschriften und Verschlüsselung durch Agenten bereit.

Das Agentensystem selbst besitzt ein visuelles Werkzeug zur Erzeugung und Verwaltung der Agenten namens TAHITI. Damit ist insbesondere die Einstellung der umfangreichen Sicherheitsparameter leicht möglich. Nach „innen“ hin, also für die `Aglets`, existiert eine uniforme Umgebung, die eine Programmierschnittstelle in Form mehrerer Klassen, z.B. des `AgletContext`, anbietet. Sie bietet alle Funktionen an, die ein Agent benötigt.

Als Besonderheit ist noch zu nennen, daß diese Umgebung zusätzlich noch die Persistenz von Agenten ermöglicht. Mit der Methode `Aglet.deactivate()` läßt sich jeder Agent *externalisieren*, d.h. auf ein sekundäres Speichermedium auslagern und zu gegebener Zeit wieder *internalisieren*. Diese „Lagerung“ hat natürlicherweise die Persistenz zur Folge. Analog zur Migration existiert die Möglichkeit zur Anmeldung von Instanzen von Subklassen von `PersistencyListener`, die eine gute Kontrolle über die Persistenz bietet.

Die Autoren streben eine weite Verbreitung der AGLETS WORKBENCH an; daher stellen sie z.Z. die gesamte Umgebung zusammen mit ihrer Dokumentation kostenlos zur Verfügung. Des weiteren existiert ein *Applet* namens FIJI, das es gestattet, ohne daß das Agentensystem installiert sein muß, nur durch Nutzung eines *WWW-Browers* Agenten zu starten und zu administrieren. Schließlich wurden das ATP und die Schnittstelle des Agentensystems der OMG (*Object Management Group*) als Vorschlag zur Standardisierung vorgelegt (siehe auch Abschnitt 2.3.4).

2.3.3 Das JAFMAS-System

Das an der UNIVERSITY OF CINCINNATI entwickelte JAFMAS-System (*Java-based Agent Framework for MultiAgent Systems*) bietet einen auf JAVA basierenden Rahmen für die Entwicklung und die Implementierung von Multi-Agentensystemen [Cha97]. Konzeptuell stammen Multi-Agentensysteme aus der Domäne der *verteilten* künstlichen Intelligenz (*distributed AI*). Die in [Cha97] getroffene Definition eines Multi-Agentensystems ist folgende:

> *[A multiagent system is] a loosely coupled network of problem solvers that work together to solve problems that are beyond their individual capabilities.*

Die Zusammenarbeit der Agenten in JAFMAS, die auch als soziales Verhalten bezeichnet wird, gehorcht dabei einem Modell, das auf der Sprechakttheorie basiert. Dabei wurden manche Prinzipien aus dem COOL-System verwendet, das eine erweiterte Fassung von KQML verwendet und auf LISP basiert [BF93, BF94]. Begrifflich wird zwischen folgendem unterschieden:

Kommunikation: Die Kommunikation ermöglicht den es den Agenten, Informationen auf der Grundlage des Koordinationsmodells auszutauschen. Es stehen gerichtete und ungerichtete (*broadcast*) Mechanismen bereit. Zur Verfügung steht zusätzlich ein themenorientiertes Verfahren, bei dem bei Interesse an einem Thema Nachrichten dafür abonniert werden können.

Interaktion: Damit sind Handlungen und Entscheidungen von Agenten gemeint, die durch die Anwesenheit oder das Wissen anderer Agenten beeinflußt wurden. Der Sprechakttheorie entsprechend geschieht die Interaktion durch *Äußerungen*, die Absichten oder Wissen bekunden können. Diese werden auf der über der Kommunikation liegenden *linguistischen Ebene* ausgetauscht und interpretiert und verfügen über ein spezielles Format mit agentenunabhängiger Semantik. Insbesondere entkoppelt dies die Interaktion von der internen Struktur des Agenten.

Koordination: Die Koordination von Agenten durch ihr gemeinsames Verhalten ist eine Eigenschaft der Interaktion. Zur Lösung des Koordinierungsproblems werden Modelle der Organisationstheorie angewendet. Agenten in JAFMAS haben einen *Plan*, der die Möglichkeiten der Problemlösung vorgibt. Die Beschreibung dieser Pläne führt zur Modellierung durch endliche Automaten. Diese Pläne werden in Form von *Konversationen* spezifiziert, die Gruppen von *Äußerungen* bilden. Konversationen werden Mengen von *Regeln* assoziiert, die die Ausführung der abstrakten Zustandsübergänge ermöglichen, indem sie alternative Reaktionen auf Äußerungen anderer Agenten implementieren.

Kohärenz: Dies betrifft das Multiagentensystem als ganzes und beschreibt, wie gut es bei der Lösung der gesetzten Probleme arbeitet und wie es sich dabei verhält. Durch die Zusammenfassung der Automatenmodelle der einzelnen Konversationen in *Petrinetze* ergibt sich die Möglichkeit, das Systemverhalten insgesamt zu analysieren.

Die technische Umsetzung geschieht dabei durch lediglich 16 JAVA-Klassen. Zum Erstellen einer Anwendung sind sogar nur 4 davon zu erweitern. Die Realisierung des Agentensystems bedient sich dabei in starken Maße der *threads* sowie des RMI (*remote invocation interface*). Jeder Agent besitzt einen assoziierten *thread*, sowie jede laufende Konversation. Zusätzlich existiert für jedes initiiertes Thema ein weiterer, der die eingehenden Nachrichten an die interessierten Agenten weiterleitet. Das RMI wird zur Kommunikation der Agenten intensiv genutzt; die Objektserialisierung macht sogar den Versand von Agenten möglich. Schließlich ist die Grundlage für ein visuelles Administrationswerkzeug vorhanden, das für eigene Agenten erweitert werden kann.

Zur Entwicklung von Multiagentensystem-Anwendungen wird ferner eine aus fünf Schritten bestehende Methode präsentiert.

1. *Identifizierung der Agenten*. Dies entspricht der klassischen Analysephase. Hier müssen die verschiedenen, miteinander interagierenden Einheiten festgestellt werden, sowie die benötigten Dienste festgelegt werden. Dies führt zur Definition von anwendungsspezifischen Agentenklassen.

2. *Identifizieren der Konversationen*. Mit dem Wissen der möglichen Interaktionen lassen sich jetzt alle möglichen Konversationen zwischen den Agenten ausarbeiten. Sie werden als Automatenmodelle spezifiziert. Aus ihnen entstehen später die anwendungsspezifischen Konversationsklassen.

3. *Identifizieren der Konversationsregeln*. Hier wird das Verhalten eines Agenten in jeder Situation einer Konversation festgelegt.

4. *Analyse des Konversationsmodells*. In dieser Phase muß die logische Kohärenz des bisher entworfenen Systems überprüft werden. Die entwickelten Automaten lassen sich z.B. durch Zusammenfassung zu Petrinetzen auf Lebendigkeit, Verklemmungen, nicht erreichbare Zustände etc. analysieren.

5. *Implementierung*. Als letzter Schritt müssen die so entworfenen Teile in einer geeigneten Umgebung implementiert werden.

Zusammenfassend ist zu sagen, daß das vorgestellte System sowohl konzeptuell (durch explizite Modellierung der Interaktion mittels Sprechakten und starke Übereinstimmungen in der Terminologie) als auch in der technischen Realisierung (durch eine ähnlich nebenläufig entworfene Architektur und Objektorientierung) dem in dieser Arbeit vorgestelltem TBC-System sehr ähnlich ist. Unterschiede sind vor allem in der Modellierung des genauen Aufbaus der Nachrichten (beeinflußt von KQML vs. strukturierter Dokumentenbeschreibung) sowie der in JAFMAS möglichen ungerichteten und themenzentrierten Kommunikation, die das Modell der *Business Conversations* nicht kennt, zu sehen. Hervorzuheben sind die klare Begriffsbildung und das Vorhandensein einer Methode für die Analyse und den Entwurf von Agentensystemen.

2.3.4 Die *Mobile Agent System Interoperability Facilities Specification*

Der der OMG (*Object Management Group*) vorliegende Vorschlag zur Standardisierung gewisser Aspekte mobiler Agenten stammt von verschiedenen wichtigen Unternehmen und Institutionen, die in diesem Bereich tätig sind und ist *Mobile Agent System Interoperability Facilities Specification* betitelt, abgekürzt MAF [CGG+97].

Ausgangspunkt der Argumentation ist, daß die sehr junge Technologie mobiler Agenten zu einer Vielzahl sehr verschiedener Systeme verschiedener Hersteller geführt hat, die nicht interoperabel sind. Um diesem Mißstand abzuhelfen, wird die Notwendigkeit gesehen, gewisse Aspekte zu standardisieren. Das Ziel ist, Interoperabilität zwischen Agentensystemen verschiedener Hersteller zu erreichen, die in derselben Sprache geschrieben sind. Das Ziel ist aber *nicht*, zwischen verschiedenen Sprachen Interoperabilität zu schaffen. Nicht standardisiert werden sollen (und können) die Ausführung von Agenten, die Einzelheiten der Migration (Serialisierung und Deserialisierung) etc. Der Standardisierungsvorschlag umfaßt die folgenden Punkte:

Agentenverwaltung: Es soll jedem Agentensystem ermöglicht werden, Agenten anderer Systeme zu kontrollieren. Dazu soll eine einheitliche Schnittstelle am Agenten vorhanden sein, die Funktionen zum Steuern der Agentenaktivitäten besitzt.

Agentenverfolgung: Diese zu standardisierenden Dienste sollen das Auffinden beliebiger Agenten anhand ihrer Namen leisten.

Agentenkommunikation: Die Kommunikation soll ausdrücklich *nicht* standardisiert werden. Vielmehr wird auf CORBA als Grundlage der Kommunikation verwiesen.

Agententransport: Dies ist der als am schwierigsten eingeschätzte Punkt. Er umfaßt die Methoden und Protokolle zum Transport von Agenten und den von ihnen benötigten Klassen.

Des weiteren werden in dem Vorschlag weite Teile der im Umfeld der mobilen Agenten gebräuchlichen Terminologie definiert. Die wichtigsten Begriffe, soweit sie bisher noch nicht erläutert wurden bzw. ohnehin gängig sind, sollen hier kurz wiedergegeben werden.

- Der *Ort* eines Agenten (*agent location, place*) ist die Adresse, an der er sich zu einem gegebenen Zeitpunkt aufhält. Wie in Abbildung 2.8 dargestellt ist, kann ein Agentensystem mehrere Orte enthalten. An jedem Ort können sich beliebig viele Agenten zugleich aufhalten. Ein Ort ist definiert als eine Umgebung, in der ein Agent ausgeführt (beherbergt) werden kann. Orte stellen die Anfangs- und Endpunkte einer Migration dar.

- Die in Abbildung 2.8 gleichfalls dargestellte *Kommunikationsinfrastruktur* ist der unterste Teil der vorgesehenen Schichtenarchitektur. Sie ist die für die Migration, die Abwicklung der Kommunikation – etwa via RPC (*remote procedure call*) – und die Sicherheitsbelange zuständige Komponente. Sie vermittelt transparent zwischen den einzelnen Orten und Agentensystemen.

- Authentisierung und Autorisierung sind die Konzepte, die die Sicherheit in Agentensystemen gewährleisten sollen.

- Mehrere Agentensysteme können zu *Regionen* zusammengefaßt werden, die derselben Authentisierungs- und Autorisierungsinstanz zugeordnet sind.

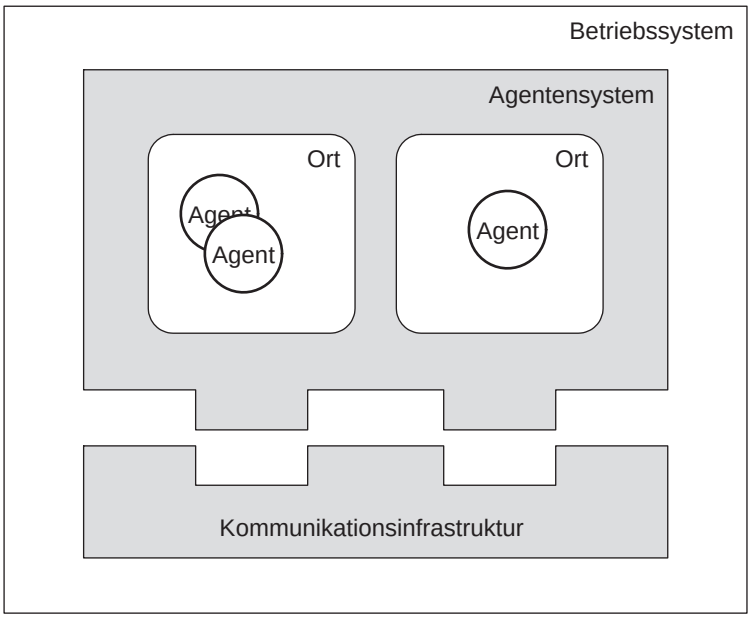


Abbildung 2.8: MAF-Architektur eines Agentensystems

Ein wesentlicher Bestandteil der Spezifikation ist eine Sammlung von Definitionen (s.o.) und Schnittstellen, die Interoperabilität gewährleisten sollen. Letztere unterteilen sich in das `MAFAgentSystem` und den `MAFFinder`.

Das erste definiert Agentenoperationen (Erzeugen, Empfangen, Anhalten, Beenden), der zweite dient zum Registrieren und Auffinden von Agentensystemen, Orten und Agenten. Beide sind in der bei der OMG gängigen IDL (*Interface Definition Language*) spezifiziert.

Zur Illustration ist hier das vorgeschlagene `MAFAgentSystem` zum Teil (unter Auslassung der meisten Parameter und der Ausnahmen) wiedergegeben:

```
interface MAFAgentSystem {
  Name create_agent(...);
  OctetStrings fetch_class(...);
  Location find_nearby_agent_system_of_profile(...);
  AgentStatus get_agent_status(...);
  ...
  MAFFinder get_MAFFinder();
  NameList list_all_agents();
  NameList list_all_agents_of_authority(...);
  Locations list_all_places();
  ...
  void receive_agent(in Name a_name, ...);
  void resume_agent(in Name a_name);
  void suspend_agent(in Name a_name);
  void terminate_agent(in Name a_name);
  ...
  void terminate_agent_system();
};
```

Die durch den `MAFFinder` vermittelten Namensdienste bedienen sich zur Adressierung von Orten etc. (`Locations`) Namen, die entweder die Form einer URI mit einem CORBA-Namen oder die einer URL mit einer Internet-Adresse haben dürfen.

Die Schnittstelle des `MAFFinder` sei hier ebenfalls (stark gekürzt) wiedergegeben, um einen Eindruck seines Leistungsumfanges zu vermitteln:

```
interface MAFFinder {
  void register_agent(in Name a_name, in Location a_location, ...);
  void register_agent_system(in Name as_name, in Location as_l, ...);
  void register_place(in Name p_name, in Location p_location);

  Location lookup_agent(in Name a_name, in AgentProfile a_profile);
  Location lookup_agent_system(in Name as_name, ...);
  Location lookup_place(in string p_name);

  void unregister_agent(in Name a_name);
  void unregister_agent_system(in Name as_name);
  void unregister_place(in string p_name);
};
```

Damit wird lediglich die Schnittstelle spezifiziert; wie im einzelnen die Suche und die Verwaltung der Agenten etc. zu geschehen hat, bleibt jeder Implementierung überlassen.

2.3.5 Bewertung

Grundsätzlich gilt es festzuhalten, daß sich nach der allerdings noch immer anhaltenden Phase der rapiden Fortentwicklung der Agententechnik eine gewisse Konvergenz sowohl

in der Terminologie als auch in den Bestrebungen, zusammenzuarbeiten, abzeichnet. Nicht zu übersehen ist auch hier der starke Trend, Agentensysteme mit Hilfe objektorientierter Sprachen und Systeme zu bauen. Dies liegt nicht zuletzt am Siegeszug, den JAVA begonnen hat. Fast alle neueren Systeme sind mittlerweile darin entwickelt worden.

Interessant ist, daß sich eine klare Trennung zwischen solchen Systemen abzeichnet, die die explizite Koordination der Agenten auf Grundlage von aus der Sprechakttheorie abgeleiteten Modellen bieten, sowie solchen, die lediglich Kommunikationsmechanismen ohne Formalisierung oder Modell der ablaufenden Interaktion anbieten. Tendenziell sind dabei die Systeme der ersten Art im akademischen Umfeld entstanden, während die der zweiten Art eher kommerzielle Produkte sind. Der in [Cha97] angestellte Vergleich belegt dies.

Bei genauerer Betrachtung ähnelt die vorgestellte Basisfunktionalität vieler Systeme sich stark. Die Unterschiede liegen im wesentlichen in den Schnittstellen, der Realisierung der Mobilität und den Kommunikationsformen der Agenten untereinander.

Bemerkenswert ist, daß sich bereits jetzt Standardisierungsbemühungen abzeichnen. Nicht von ungefähr stehen dahinter die meisten der mit der Materie befaßten Firmen und Institutionen. Als Grund dafür ist die Unmöglichkeit zu sehen, ohne einen breiten Standard (insbesondere ohne Interoperabilität) diese Systeme kommerziell erfolgreich einzusetzen.

Das Fehlen anerkannter und erprobter Sicherheitsmechanismen wird allerdings vermutlich noch einige Zeit einen größeren Durchbruch, der über rein akademische Versuche hinausgeht, verhindern. Der breite Raum, der diesem Thema mittlerweile in allen Publikationen eingeräumt wird, belegt die Dringlichkeit dieses Problems. Insofern spiegelt sich die allgemeine Diskussion über elektronische Zahlungsmöglichkeiten im Internet hier ebenfalls wider.

Auch wenn die Hersteller von Agentensystemen sich auf einen gemeinsamen Standard einigen können und konforme Produkte entwickeln, bleibt es – ganz im Sinne der MAF – dabei, daß Agentensysteme nur Agenten, die in der gleichen Sprache oder vom gleichen Hersteller entwickelt wurden, „verstehen“, d.h. ausführen können. Dies bedingt entweder abgeschlossene, proprietäre Lösungen, da man für die Nutzung eines Dienstes von einem bestimmten Agentensystem abhängig ist, oder aber die Installation sämtlicher relevanter Agentensysteme auf jedem beteiligten Rechner. Durch Umsetzung der Interoperabilitätsspezifikationen laut MAF ist letzteres zwar prinzipiell machbar, aber dafür erscheint das Problem der Kommunikation in voller Deutlichkeit.

Unserer Ansicht nach ist die Koordination der Kommunikation zwischen verschiedenen Agenten das hauptsächliche Problem. Die Nutzung beliebiger Dienste wird ohne eine einheitliche Dienstbeschreibung und ohne einheitliches Protokoll stark erschwert. Der große Erfolg des *WWW* basiert gerade auf der Tatsache, daß beides existiert. Das Fehlen wird den Erfolg von kommerziellen mobilen Agenten wahrscheinlich stark verzögern.

Deshalb liegt der Schwerpunkte der im Rahmen dieser Arbeit vorgenommenen Entwicklung eines Agentensystems in der Realisierung der *Business Conversation*, da diese dazu geeignet sind, die geschilderten Probleme zu lösen. Die vielen ebenfalls auf der Sprechakttheorie basierenden Ansätze unterstützen diese Ansicht.

2.4 Einordnung des TBC-Agenten-Modells

Abschließend sollen die bisherigen Ergebnisse betrachtet, in Beziehung zueinander gesetzt und dazu genutzt werden, die Eigenschaften des in dieser Arbeit zu entwerfenden Systems einzuordnen und auf abstrakter Ebene genauer zu definieren.

Der Wandel von Informationssystemen hat von der Verteilung der Daten über Objekte bis hin zur agentenorientierten Verteilung geführt. Dem Modell der Objektverteilung mangelt es dabei hauptsächlich an geeigneten Interaktionsmodellen.

Die Entwicklung von Agentensysteme ist bisher konzeptuell gesehen zweigleisig geschehen. Während die überwiegend technischen Anforderungen an die Mobilität und an die Verteilung weitgehend problemlos (wenn auch nicht immer elegant) gelöst worden sind, trifft dies auf die Anforderungen der Interaktion nur bedingt zu. Wie in Abschnitt 2.3.5 gezeigt, wird das Problem entweder ganz ausgeblendet und dem Anwender überlassen, oder es werden Interaktion und Koordination explizit meist unter Nutzung der Sprechakttheorie modelliert. Der Hauptbeitrag der Agentensysteme liegt bisher in der Betonung der Aspekte der Autonomie und der daraus folgenden Asynchronizität. Die in Abschnitt 2.3.1 aufgestellten Forderungen fassen die Beiträge zusammen.

Das Modell der *Business Conversations* abstrahiert von allen technischen, organisatorischen und architekturellen Gegebenheiten und konzentriert sich allein auf die Interaktion zweier Akteure. Es ist *per se* weder auf die Verwendung in einem Agentensystem ausgelegt noch trifft es irgendwelche Aussagen über Implementierung oder operationale Aspekte wie Nebenläufigkeit, Koppelung an Anwendungen oder Protokolle. Charakterisiert wird dies durch den Begriff der *Medien- und Aktorenunabhängigkeit*. Begrifflich kommt dies auch durch die Verwendung des Terminus *Akteur* zum Ausdruck.

Die Brücke wird nun durch die Integration der *Business Conversations* als Interaktionsmodell in ein den sonstigen, oben genannten Forderungen an ein Agentensystem Rechnung tragendes System geschlagen. Die Synergie beider Konzepte soll dazu verwendet werden, die Analyse, den Entwurf und die Implementierung von kommunizierenden, verteilten Anwendungen zu verbessern. Besondere Unterstützung soll die Modellierung der Evolution von Anwendungssystemen dadurch erhalten.

In Hinblick auf die in Abschnitt 2.1 genannten kooperativen Informationssysteme und die Aspekte der *legacy systems*, Satellitensysteme und Internet-Informationssysteme rücken so die von den klassischen Agentensystemen meist stark betonten Merkmale der Mobilität, der Plattformunabhängigkeit und der Sicherheit etwas in den Hintergrund. Im Vordergrund stehen die Interaktion, Koordination und Kommunikation der Partner. Begrifflich sind nun Agenten und Akteure gleichzusetzen.

Kapitel 3

Tycoon

In diesem Kapitel werden die Sprache TL-2 und das TYCOON-2-System beschrieben, die für die in den Kapiteln 4 und 5 beschriebene Realisierung der *Tycoon Business Conversations* (TBC) und des prototypischen Informationssystems verwendet werden.

Nach einem Abriß der Entwicklungsgeschichte im folgenden Abschnitt 3.1 werden in Abschnitt 3.2 die Kernkonzepte der Sprache TL-2 vorgestellt und schließlich in Abschnitt 3.3 die weiteren, nicht zum eigentlichen Sprachumfang gehörenden Systemeigenschaften von TYCOON-2 dargestellt.

3.1 Historie

Das TYCOON-System (***Ty**ped **C**ommunicating **O**bjects in **O**pen E**n**vironments*) und seine Programmiersprache TL (***T**ycoon **L**anguage*) wurden seit etwa 1993 am Arbeitsbereich Datenbanken und Informationssysteme (DBIS) des Fachbereichs Informatik der Universität Hamburg im Rahmen des europäischen Projekts FIDE entwickelt. Wesentliche Merkmale sind der funktional-imperative Stil von TL, parametrischer und Subtyppolymorphismus, Funktionen höherer Ordnung als Sprachobjekte erster Klasse sowie die orthogonale Persistenz und Mobilität von Daten, Code und Prozessen [Mat93, MMM93, MMS94, MSS97, MMS96, Mat96].

Zwischen 1995 und 1996 wurde dann ein objektorientierter Dialekt namens TOOL (***T**ycoon **o**bject **o**riented **l**anguage*) geschaffen [GM95, GM96], der zusammen mit dem alten TYCOON-System die Grundlage für das ab 1996 entwickelte TYCOON-2-System bildete. Letzteres entstand im wesentlichen im Rahmen der von Absolventen und Mitarbeitern von DBIS neu gegründeten Firma HIGHER-ORDER [GMSS97, HOX98].

Im Verlaufe der Entwicklung und des gleichzeitigen erfolgreichen kommerziellen Einsatzes wurde das TYCOON-2-System grundlegend revidiert, wobei eine eigene, vom ursprünglichen TYCOON-System unabhängige, neue virtuelle Maschine, ein Typprüfer und ein Übersetzer entstanden [Wei98, Ern98, Wie97]. Eine breite Darstellung der Sprache TL-2 und der ihr zugrundeliegenden Entwurfsentscheidungen wird [Wah98] bieten.

Zur Zeit wird das TYCOON-2-System von der Firma HIGHER-ORDER zur Realisierung von anspruchsvollen Projekten insbesondere im Medienbereich und vom Arbeitsbereich Softwaresysteme (STS) der TU-Hamburg-Harburg zu Forschungszwecken im Bereich der persistenten Objektsysteme verwendet.

3.2 Die Sprache TL-2

Syntaktisch und lexikalisch ist TL-2 relativ stark an C und JAVA angelehnt. Semantisch und pragmatisch sind die größten Anleihen bei SMALLTALK, z.T. bei CLOS und natürlich historisch bedingt bei TYCOON gemacht worden.

3.2.1 Klassen, Objekte und Nachrichten

TL-2 ist eine *statisch typisierte* Programmiersprache. Das bedeutet, daß jedem Wertausdruck statisch, d.h. zur Zeit der Übersetzung, durch Analyse des Programmtextes ein Typ zugeordnet werden kann. Dies garantiert, daß der Ausdruck zur Laufzeit auf jeden Fall zu einem Wert dieses Typs evaluiert. Typfehler, in objektorientierten Sprachen also insbesondere der Fehler, daß ein Objekt eine Nachricht nicht versteht, sind somit ausgeschlossen.

Die Programmiersprache TL-2 gehorcht dem objektorientierten Paradigma, d.h. *Objekte* und *Nachrichten* zwischen ihnen bilden die alleinige Grundlagen der Programmierung. Ausnahmslos jedes Objekt in TL-2 ist ein Exemplar einer *Klasse*. Es gibt keine primitiven Datentypen; auch Zeichenketten und Zahlen werden durch Objekte und Klassen dargestellt. Sie sind dadurch nahtlos in das Typsystem integriert. Man spricht in diesem Zusammenhang dann auch von „reiner“ Objektorientierung.

Die Objektorientierung geht einher mit einer strikten *Referenzsemantik*. Grundlage dafür ist das Vorhandensein der von Werten unabhängigen und unabänderbaren *Objektidentität* eines jeden Objekts. Alle Objekte sind auf einer Halde (hier einem persistenten Objektspeicher) angelegt, so daß alle Variablen tatsächlich nur Verweise auf die Objekte darstellen; die Dereferenzierung geschieht stets implizit. Dadurch entfallen fehleranfällige Zeigerkonstrukte und Probleme mit der Speicherverwaltung. Die automatische Freispeicherverwaltung beruht auf transitiver Erreichbarkeit von einem Wurzelobjekt aus [Wei98]. Die Referenzsemantik bedingt die Begriffe der Objektidentität, der flachen sowie tiefen Gleichheit von Objekten. Einher damit gehen analoge Begriffe der Kopiersemantik. Einer Einschränkung unterliegt die Objektidentität gewisser elementarer Klassen wie die der Zahlen, Wahrheitswerte etc.: Deren Instanzen sind bereits bei Wertgleichheit objektidentisch und somit ununterscheidbar. Dies ist eine Folge der Implementierung, die allerdings auch sinnvoll ist: 14 ist immer gleich 14 ist immer gleich 14.

In einer Klasse sind *Methoden* und *Variablen* (*slots*) definiert. Sie bestimmen das Verhalten und den Zustand eines Objektes dieser Klasse. Klassen sind also Abstraktionen gleichartiger Objekte.

Das Senden einer Nachricht an ein Objekt im rein objektorientierten Sinne bewirkt die Ausführung des Codes einer Methode der Klasse des Objekts. Dabei können dieser Nachricht

neben dem Methodenselektor (dem Namen der Methode) beliebig viele Parameter (weitere Objekte) mitgegeben werden, und es kann weiterhin ein Ergebnis zurückgegeben werden. Dies entspricht ziemlich genau dem Aufruf imperativer oder funktionaler Prozeduren oder Funktionen, nur daß jetzt ein Empfängerobjekt als impliziter Parameter mit im Spiel ist. Das Objekt, das die Nachricht empfängt, wird innerhalb der Methodendefinitionen seiner Klasse mit dem Schlüsselwort `self` referenziert.

Sowohl Methoden als auch Variablen können als privat deklariert werden; dann haben nur die Methoden der Klasse Zugriff auf diese, und auch nur auf das aktuelle Objekt `self`. Das folgende Beispiel verdeutlicht die zu verwendende Syntax:

```
class Counter

public methods

  increment() :Void
  {
    _count := _count + 1
  }

private

  _count :Int
```

Die Klasse `Counter` hat eine öffentliche Methode `increment` sowie eine private Variable `_count` (privaten Variablen wird per Konvention ein Unterstrich vorangestellt). Tatsächlich ist die zunächst sehr imperativ anmutende Zuweisung eine Folge von Methodenaufrufen, die auch folgendermaßen geschrieben werden kann:

```
...
_count.":="(_count."+"(1))
...
```

Die Methodenselektoren, die nicht nur aus alphanumerischen Zeichen bestehen, müssen dabei in Anführungsstrichen geschrieben werden. So ist gut zu erkennen, daß zunächst dem Objekt `_count` die Nachricht `"+"` mit dem Parameter 1 gesendet wird; das daraus resultierende Ergebnis, ein Objekt, wird als Parameter der Nachricht `":="` wiederum an `_count` verwendet. Die gewohnte Schreibweise ist allerdings der Übersichtlichkeit halber vorzuziehen. Hingegen besser (und üblich) ist die o.g. *normalisierte* Notation in folgendem Code-Fragment, das die Verwendung des Zählers illustriert:

```
...
let c :Counter = Counter.new,
c.increment,
c.increment,
...
```

Dies erst macht deutlich, daß die intuitive Verwendung von `_count` überhaupt nicht so klar ist. Genaugenommen müßte geschrieben werden:

```
...
self._count.":="(self._count."+"(1))
...
```

Da in TL-2 alle Ausdrücke Nachrichten an Objekte sind, ist natürlich auch `_count` eine Nachricht, und zwar an das aktuelle Objekt `self`. Sie gibt die Objektreferenz auf die Variable zurück. Erst an das solchermaßen ermittelte Objekt wird die Nachricht `"+"` geschickt. Der Lesbarkeit halber kann der Empfänger `self` meistens weggelassen werden.

Generell ist die Notation also `Empfänger.Methodenselektor`, gefolgt von den eventuellen Parametern in runden Klammern. Ist der Empfänger nicht angegeben, so wird `self` angenommen. Tatsächlich sind alle Operationen, also auch numerische und logische, durch Nachrichtenaustausch realisiert. Sogar die üblichen Kontrollstrukturen sind als Methoden in der generellen Superklasse `Object` definiert.

Letztlich liefern alle Ausdrücke in TL-2 Werte; es sei denn, sie sind vom Typ `Void`. Der Wert, den ein Methodenaufruf liefert, ergibt sich aus dem letzten Wert, den er berechnet. Die Methode im folgenden Beispiel liefert den Wert 14 zurück:

```
calculate() :Int
{
  let c :Int = 7,
  c := c - 1,
  c + 8
}
```

Die Implementierung von einigen elementaren Methoden geschieht allerdings durch in dem Laufzeitsystem eingebaute Routinen; der Methodenrumpf ist dann durch das Schlüsselwort `builtin` ersetzt.

3.2.2 Vererbung und Metaklassen

Klassen können voneinander erben. Die erbende Klasse, die Subklasse, übernimmt dabei alle Methoden- und Variablendefinitionen der Superklassen. Dabei können Methodendefinitionen durch Spezialisierungen überschrieben werden. Das Senden einer Nachricht an ein Objekt bewirkt die Suche nach einer Methodendefinition in der Klasse des aktuellen Empfängerobjektes. Wird keine gefunden, so wird in der oder den Superklassen weitergesucht. Wird auch dort keine gefunden, so wird ein Laufzeitfehler ausgelöst. Diese Art der Bindung von Nachrichten an Methodendefinitionen wird als *späte* oder *dynamische Bindung* bezeichnet. Weiterhin ist es möglich, mit Hilfe des Schlüsselwortes `super` als Empfängerobjekt alte, überschriebene Methodendefinitionen zu verwenden. Zusätzlich unterstützt TL-2 Mehrfachvererbung. Konflikte können dabei nicht auftreten, da die Superklassen einer Klasse durch ein der Breitensuche ähnliches Verfahren in eine lineare Ordnung gebracht werden (*class precedence list*, CPL). Dadurch ist stets genau definiert, auf welche Superklasse sich `super` bezieht, und außerdem ist gewährleistet, daß kaskadierende `super`-Aufrufe keine Klasse auslassen.

Vererbung in TL-2 bedeutet immer auch Vererbung von Implementierungen, nicht nur von Schnittstellen (anders als z.B. in JAVA, das nur einfache Vererbung (*subclassing*) und zusätzlich Schnittstellenvererbung durch eine `implements`-Klausel anbietet). Dennoch sind in TL-2 – neben beliebigen Mischformen – sowohl abstrakte Klassen, die niemals instantiiert werden dürfen, als auch reine Schnittstellendefinitionen als Klassen möglich. Erstere können bereits teilweise Methodendefinitionen besitzen, letztere ergeben sich einfach durch das vollständiges Weglassen der Methodenrümpfe und die Verwendung des Schlüsselwortes `deferred` an ihrer Stelle. Die Mehrfachvererbung unterstützt so einen Programmierstil, der von sehr fein granularen Abstraktionen und dem sogenannten *mixin-style* Gebrauch macht.

Die Klasse `Object` ist die allen anderen Klassen gemeinsame Superklasse. Sie definiert das für alle Objekte des Systems gemeinsame Protokoll. Tatsächlich sind alle auf den ersten Blick wie Konstanten etc. aussehende Konstrukte, z.B. die Wahrheitswerte `true` und `false`, lediglich Methoden oder Variablen von `Object`, die automatisch durch die dynamische Bindung verwendet werden, wenn sie in Ausdrücken auftauchen.

Ähnlich wie in SMALLTALK sind auch in TL-2 Klassen Objekte erster Klasse. Das bedeutet, daß sie Objekte sind, die einer Klasse angehören, die Methoden und evtl. sogar Variablen hat. Diese Klasse wird als *Metaklasse* einer Klasse bezeichnet. Die Methoden der Metaklasse sind für die Erzeugung von Instanzen der Klasse zuständig. Dies wird normalerweise von einer Methode namens `new` geleistet. Diese Konstruktormethoden sind also anders als in vielen objektorientierten Sprachen (wie z.B. C++) keine besonderen Sprachkonstrukte. Die Möglichkeit, Metaklassen selbst zu definieren, läßt eine genaue Kontrolle und ggf. Verwaltung der Objekterzeugung zu. Ganz leicht läßt sich so zum Beispiel eine Klasse bauen, die nur genau eine Instanz besitzt (*singleton*).

Bei jeder Klassendefinition müssen die Metaklasse sowie die Superklasse bzw. die Superklassen angegeben werden. Das obige Beispiel des `Counter` lautet also vollständig:

```
class Counter
super Object
metaclass SimpleConcreteClass(Counter)
...
```

`SimpleConcreteClass` ist eine generische Metaklasse, die ein einfaches Protokoll zur Erzeugung und Initialisierung von Objekten anbietet:

```
class SimpleConcreteClass(Instance <: Object)
super ConcreteClass(Instance)
metaclass MetaClass

public methods

  new() :Instance
  {
    _new._init
  }
```

Sie verwendet die von ihrer Superklasse `ConcreteClass` geerbte Methode `_new` (eine in der Maschine eingebaute Methode), die ein neues Objekt der Klasse `Instance` zurückgibt, im vorigen Beispiel also ein `Counter`-Objekt. Das Initialisierungsprotokoll sieht dann den Aufruf der parameterlosen Methode `_init` der Klasse `Instance` vor, die das Objekt bei Bedarf initialisieren kann. Sie muß das Objekt selbst zurückgeben; der letzte Ausdruck einer jeden `_init`-Methode wird also normalerweise `self` sein. Enthält die Klasse keine `_init`-Methode, so schadet dies nicht; durch die dynamische Bindung wird die Methode der Superklasse verwendet. Die Klasse `Object`, von der alle Klassen erben, enthält eine leere `_init`-Methode, die nichts tut, so daß immer gewährleistet ist, daß es überhaupt eine gibt. Definiert man selbst eine, so ist es ratsam, darin zu Anfang mit `super._init` auch die Initialisierung der Superklassen aufzurufen. Durch das Modell der Klassenpräzedenzliste (CPL) ist gewährleistet, daß auch bei Mehrfachvererbung alle Initialisierungsmethoden zum Zuge kommen. Das obige Beispiel des Zählers wird nun durch die Angabe einer privaten `_init`-Methode vervollständigt:

```
...
private methods

  _init() :Self
  {
    _count := 0,
    self
  }
```

Es ist gängiger Stil, neben dem parameterlosen Konstruktor `new` in der Metaklasse weitere Methoden mit gewissen Initialisierungsparametern zu definieren. Da die Methoden einer Metaklasse ausnahmsweise auch Zugriff auf die privaten Attribute und Operationen ihrer Klasse haben, läßt sich ein neues Objekt so auch einfach initialisieren. Hier ist anzumerken, daß TL-2 kein Überladen von Methodennamen (*overloading*) kennt, so daß alle Methoden, also auch alle Konstruktoren verschiedene Namen erhalten müssen.

Zu erläutern bleibt noch das Konzept des Klassenobjekts. Jede Klasse ist ein Objekt seiner Metaklasse. Das Objekt wird vom Compiler zum Zeitpunkt der Übersetzung der Klasse angelegt. Damit es aber auch verwendbar ist, existiert im TYCOON-2-System der sogenannte *Pool*, der alle Klassenobjekte enthält. Er ist stets letztes Glied in jeder Klassenpräzedenzliste (CPL), so daß jetzt auch klar ist, warum `Counter.new` verstanden wird: Die Nachricht `new` wird an das Klassenobjekt der Klasse `Counter` gesendet, das der oben geschilderten Suchstrategie zufolge im *Pool* gefunden wird. In diesem Zusammenhang ist noch die Methode `"class"` (oder synonym, aber einfacher zu schreiben, `clazz`) von `Object` zu nennen, die das Klassenobjekt eines Objektes liefert. Dadurch sind reflektive Mechanismen wie einfache Typtests zur Laufzeit leicht zu realisieren.

Es bleibt zu bemerken, daß gewisse elementare Klassen wie die der Zahlen oder Wahrheitswerte eine spezielle Metaklasse haben (`OddballClass`), die keinen expliziten Konstruktor besitzt. Die Konstruktion der Werte wird implizit durch Notation der Literale im Programmtext bzw. durch eingebaute elementare Methoden geleistet.

3.2.3 Funktionen höherer Ordnung

Dem alten TYCOON-System und seinem funktional orientierten Paradigma entlehnt ist das Konzept der Funktionen höherer Ordnung als Objekte erster Klasse. Auch in TL-2 ist es möglich, ein evtl. sogar parametrisiertes Stück Programmcode als Wert erster Ordnung an Variablen zu binden, weiterzureichen, als Parameter zu übergeben oder als Wert zurückzugeben, um es erst später auszuführen. Die Bindungen an statisch im Programmtext sichtbare Variablen werden dabei in einem Funktionsabschluß (*closure*) für den Benutzer transparent zusammen mit der Referenz gespeichert. Damit lassen sich elegante Muster wie z.B. *Iterationsabstraktionen* und andere Kontrollstrukturen realisieren. Die oben bereits erwähnte Realisierung der Kontrollstrukturen als Methoden der Klasse `Object` gelingt erst durch die Übergabe von parameterlosen Funktionen. Dies ist insbesondere für das `if`-Konstrukt notwendig, da TL-2 strikte Auswertung betreibt und keine Sonderfälle mit *lazy evaluation* kennt. Funktionen werden als Objekte verschiedener Klassen mit der Superklasse `Fun` repräsentiert. Insbesondere haben sie eine Methode `[]`, die die Funktion auswertet.

Besonders eindrucksvoll demonstrieren läßt sich der so mögliche Stil der reinen Objektorientierung anhand der Realisierung des `if`-Konstrukts. Es ist tatsächlich als Methode der abstrakten Klasse `Bool` und seiner beiden konkreten Subklassen `True` und `False` implementiert:

```
class Bool
...
  "?:"(T<:Void, iftrue :Fun():T, iffalse :Fun():T) :T  deferred

class True
super Bool
...
  "?:"(T<:Void, iftrue :Fun():T, iffalse :Fun():T) :T
  {
    iftrue[]
  }

class False
super Bool
...
  "?:"(T<:Void, iftrue :Fun():T, iffalse :Fun():T) :T
  {
    iffalse[]
  }
```

Ein Objekt, daß als `Bool` typisiert ist, führt je nachdem, ob zur Laufzeit aktuell eine Instanz von `True` oder `False` die Nachricht empfängt, den einen oder den anderen Parameter in Form eines Funktionsobjekts aus. Die Syntax des `if`-Konstrukts ist an die Kurzschreibweise in C angelehnt (daher auch der Methodenname `"?:"`) und wird wie folgt verwendet:

```
...
a.isNil ? {"Leer!"} : {"Voll!"}
...
```

Die Methode `isNil`, die in der Klasse `Object` definiert ist, liefert `true` oder `false` in Abhängigkeit davon zurück, ob das Empfängerobjekt (hier `a`) ein Verweis auf das `nil`-Objekt ist oder nicht. An dieses Ergebnis wird dann die Nachricht `"?:"` mit zwei Funktionsobjekten (Blöcken) als Parameter geschickt; das Ergebnis ist eine der beiden Zeichenketten. Objekte von parameterlosen Funktionen lassen sich syntaktisch gesehen durch die Verwendung der geschweiften Klammern { und } erzeugen. Die Kontrollstrukturen für Schleifen sind in ähnlicher Form als Methoden der Klasse `Object` realisiert. So wird deutlich, daß in TL-2 kein weiterer Mechanismus außer dem dynamischen Methodenaufruf existieren muß, um algorithmische Vollständigkeit zu erreichen.

3.2.4 Die Klassen `Nil` und `Void`

Das obige Beispiel deutet bereits die Existenz des speziellen Typs `Nil` an. Er hat genau eine Instanz, den Wert `nil`. Diesen Wert können alle Variablen annehmen, um z.B. dadurch zu signalisieren, daß der Wert nicht definiert, nicht vorhanden oder nicht bekannt ist. Außerdem ist `nil` der Wert sämtlicher Variablen (*slots*) eines neuerzeugten Objekts. Die Methoden zur Initialisierung können die diesen Wert natürlich danach verändern. Der Wert `nil` kann stets typsicher zugewiesen werden. Dies liegt daran, daß er gewissermaßen als speziellster möglicher Typ Subtyp aller Klassen ist. Die Klasse `Nil` verfügt dabei allerdings über keine Methoden, so daß jede Nachricht an `nil` mit einem Laufzeitfehler beantwortet wird.

Im Gegensatz dazu steht die Klasse `Void`. Sie ist die Superklasse von `Object`; es existiert aber im Gegensatz zu `Nil` kein einziger Wert dieser Klasse. Mit dem Typ `Void` läßt sich das Fehlen eines Wertes auf Typebene ausdrücken, zum Beispiel, daß eine Methode keinen Rückgabewert besitzt oder daß ein Typparameter nicht benötigt wird. Dieser allgemeinste aller Typen läßt sich so keinem Objekt zuweisen.

3.2.5 Polymorphismus

Das Typsystem von TL-2 beruht im Gegensatz zu vielen anderen Sprachen (wie etwa C, C++ und MODULA) auf struktureller Äquivalenz und nicht auf Namensäquivalenz. Als Typ einer Klasse ist die Menge der Signaturen der Methoden und Variablen der Klasse definiert.

Die Subtypisierungsrelation, die auf strukturellen Vergleichen beruht, ist zu unterscheiden von der Vererbungsrelation. In vielen objektorientierten Sprachen werden beide identifiziert (etwa in C++) und beruhen tatsächlich nur auf der Vererbungsrelation. Eine Klasse ist dort Subtyp einer anderen, wenn sie direkt oder indirekt von der zweiten erbt. In der Regel ist aber auch in TL-2 eine Subklasse gleichzeitig Subtyp seiner Superklasse. Einzelheiten dazu sind etwa in [Ern98] zu finden.

Die Subtypisierung ist die Grundlage für eines der wichtigsten Konzepte der polymorphen Sprachen: die *Subsumption*, die es gestattet, immer dann, wenn ein Objekt des Typs A erwartet wird, eines des Typs B einzusetzen, sofern B Subtyp von A ist. Dies wird auch als *Subtyppolymorphismus* bezeichnet.

Daneben unterstützt TL-2 noch den begrenzten parametrischen Polymorphismus (*bounded parametric polymorphism*). Hierdurch lassen sich sowohl Klassen als auch Methoden und

Funktionen mit Typparametern versehen, wodurch allgemein formulierte Abstraktionen mit hoher Generizität möglich werden (dies entspricht soweit den *templates* in C++). Begrenzung des Typparameters bedeutet dabei die Angabe eines Typs als obere Schranke, der dann der Supertyp des Parameters ist. Die folgende Methode zeigt die Verwendung:

```
bestPaid(T<:Employee, employees :Sequence(T)) :T
{
  employees.sort(fun(e1 :T, e2 :T) {
    e1.salary > e2.salary
  }),
  employees[0]
}
```

Die Typbegrenzung `T<:Employee` des Typparameters `T` stellt dabei sicher, daß `p` zur Laufzeit mindestens vom Typ `Employee` ist, also die Nachricht `salary` auf jeden Fall versteht. Im Gegensatz zu anderen Sprachen (etwa C++) ermöglicht es die Begrenzung, die Typprüfung der generischen Klasse nur einmal durchführen zu müssen. Dies kann bei der häufigen Verwendung von generischen Abstraktionen sehr viel Zeit beim Übersetzungsvorgang sparen. Besonders sinnvoll lassen sich Typparameter z.B. für Behälterklassen und Iterationsabstraktionen einsetzen, die dann vollständig von dem Elementtyp abstrahieren können. Auch häufig genutzt werden generische Metaklassen.

In einer Reihe von Fällen stellt es sich als vorteilhaft heraus, wenn das Typargument nicht Subtyp eines festen Typs sein muß, sondern sich selbst wieder (selbstbezüglich) verwenden kann. Dies wird durch den sogenannten *F-bounded parametric polymorphism* erreicht. Ein klassisches Beispiel dafür ist das der sortierbaren Liste, deren Elemente sich nur mit Elementen des gleichen Typs vergleichen lassen. Das erfordert eine generische Schnittstelle zum Sortieren, hier `Sortable`. Sie definiert eine abstrakte Vergleichsfunktion, die von ihren Subklassen implementiert werden muß. Das folgende Beispiel stellt die beteiligten Klassen dar:

```
class Sortable(T <: Object)
metaclass AbstractClass
...
  "<"(t :T) :Bool deferred
...

class SortableList(T <: Sortable(T))
...

class Person
super Sortable(Person)
...
  name :String
...
  "<"(t :Person) :Bool
  {
    name < t.name
  }
...
```

So ist sichergestellt, daß jeder bei `SortableList` eingesetzte Typ die benötigte Vergleichsfunktion besitzt. Als Beispiel ist die Klasse `Person` aufgeführt. Diese Technik hilft außerdem, das Problem der kontravarianten Spezialisierungen bei Vererbung zu umschiffen.

3.3 Das TYCOON-2-System

In diesem Abschnitt sollen die wesentlichen Systemeigenschaften, die sich nicht aus der Sprachdefinition ergeben, und die zentralen Komponenten des TYCOON-2-Systems vorgestellt werden. Details insbesondere der virtuellen Maschine sind in [Wei98] zu finden.

3.3.1 Eigenschaften

Das grundsätzliche Merkmal des Systems ist die Verwendung einer virtuellen Maschine. Die TL-2-Programme werden (wie es auch z.B. bei JAVA-Programmen der Fall ist) in einen maschinenunabhängigen *Bytecode* übersetzt, der von der virtuellen Maschine interpretiert wird.

Die Maschine gestattet weiterhin die effiziente nebenläufige Ausführung leichtgewichtiger Prozesse (*threads*). Die Realisierung geschieht durch eine Abbildung auf *POSIX-threads*. Alle *threads* teilen sich dabei den gemeinsamen Objektspeicher. Dies bedingt die Notwendigkeit von Synchronisierungsmechanismen bei nebenläufigen Zugriffen auf gemeinsam benutzte Objekte. Die bereitgestellten Mechanismen sind – anders als in JAVA – nicht Bestandteil der Sprache, sondern als Klassen der Standardbibliothek definiert.

Alle Daten und der gesamte *Bytecode* (also Objekte und übersetzte Klassen) sowie die Ausführungszustände aller *threads* werden uniform im Objektspeicher gehalten. Dies ermöglicht eine effiziente implizite Freispeicherverwaltung (*garbage collection*) auf der Basis transitiver Erreichbarkeit.

Die uniforme Datenrepräsentation bedingt direkt die orthogonale Persistenz: Durch Sichern des Inhalts des gesamten Objektspeichers läßt sich der gesamte Zustand über beliebig viele Programmläufe erhalten. Das schließt laufende Prozesse und deren Zustände mit ein.

Der Kern der virtuellen Maschine ist sehr klein, da er im wesentlichen nur den Interpreter und den Objektspeicher implementiert. Alle anderen Systemkomponenten, z.B. Compiler, Klassenlader und Typüberprüfer sind vollständig in TL-2 realisiert und deshalb übersetzt im Objektspeicher abgelegt. Die reflektive Nutzung dieser im Objektspeicher abgelegten Systemkomponenten durch eigene Programme ist dadurch sehr einfach möglich. Die sehr kleine Maschine hat eine hohe Portabilität des gesamten Systems zur Folge. Der Kern ist zudem in portablem ANSI-C vollständig POSIX-konform entwickelt worden, ermöglicht also weitgehende Plattformunabhängigkeit. Die Maschine ist bereits auf verschiedenen UNIX-Systemen im Einsatz; die Portierung auf WINDOWS ist in Arbeit.

Neben der Persistenz bedingt die uniforme Datenrepräsentation auch die Portabilität und Plattformunabhängigkeit ganzer Objektspeicher und somit in TL-2 realisierter Systeme. Ohne Änderung lassen sie sich von Rechner zu Rechner verschieben und erneut aufsetzen

Noch nicht realisiert, aber grundsätzlich auch möglich (wie im alten TYCOON-System bereits geschehen [Mat96]) ist die orthogonale Mobilität von Daten, Code und Prozessen.

3.3.2 Systemkomponenten

Als wichtigste Systemkomponenten, die vollständig in TL-2 entwickelt worden sind, sind der Klassenlader, der Compiler, der Typüberprüfer sowie der sogenannte *Toplevel* zu nennen.

Deren Aufgaben sind folgende: Der Klassenlader verwaltet die Abhängigkeiten der noch immer extern in Dateien gespeicherten Klassendefinitionen. Wird eine Klasse geändert, so aktualisiert er alle abhängigen Klassen automatisch.

Dem Compiler obliegt die Übersetzung der vom Klassenlader importierten Klassendefinitionen in *Bytecode*. Dabei legt er für jede Klasse ein *Klassenobjekt* an, das Exemplar der Metaklasse ist. Die Tatsache, daß der übersetzte Code im Objektspeicher liegt, macht ihn sofort im laufenden System wirksam.

Der Typprüfer arbeitet auf den vom Compiler erstellten Syntaxbäumen, die ebenfalls im Objektspeicher gehalten werden. Dadurch erst ist ein hoher Grad an Reflexion im System möglich, der es z.B. gestattet, zur Laufzeit bereits übersetzte Klassen auf ihren Aufbau hin zu untersuchen, dynamische Typtests etc. vorzunehmen. Die Benutzung des Typprüfers ist grundsätzlich optional; der Compiler arbeitet vollständig unabhängig von ihm. Dadurch ist *rapid prototyping* möglich; tatsächlich lassen sich Klassen, die nicht statisch typisiert sind oder nicht typisierbar sind, übersetzten und ausführen.

Der *Toplevel* ist die textuelle Benutzungsschnittstelle des Systems. Er übersetzt und führt eingegebene Kommandos interaktiv aus. Gültige Kommandos sind dabei alle gültigen TL-2-Ausdrücke. Die Systemkomponenten sind durch bestimmte, ebenfalls im *Pool* bekannte Objekte repräsentiert und so erreichbar.

Nicht wirklich eine Systemkomponente ist die umfangreiche Klassenbibliothek, die sich im Laufe der Zeit gebildet hat. Entstanden aus der TOOL-Klassenbibliothek, bietet sie eine große Zahl von parametrisierbaren Behälterklassen nebst Iterationsabstraktionen für fast jeden denkbaren Zweck, Funktionalität für Synchronisierungsaufgaben in nebenläufigen Szenarien (*Monitore*, *Semaphoren*, *Conditions* etc.), Ein- und Ausgabefunktionen, Unterstützung von *Internet*-Funktionalität und vieles mehr.

Kapitel 4

Entwurf und Implementierung der *Tycoon Business Conversations*

Nachdem nun die theoretischen Grundlagen für kooperative Informationssysteme erläutert, das Modell der *Business Conversations* vorgestellt sowie die Sprache TL-2 und das zugehörige TYCOON-2-System beschrieben wurden, sollen der Entwurf und die Implementierung des *Tycoon Business Conversations*-Systems (TBC) in diesem Kapitel beschrieben werden. Die Implementierung stellt zugleich den praktischen Anteil dieser Arbeit dar.

Das Vorgehen ist dabei folgendes: Nach der Bestimmung des Umfanges der zu leistenden Arbeit in Abschnitt 4.1 folgt in Abschnitt 4.2 eine kurze Darstellung der Methodik des Vorgehens. Nach einer Übersicht über die Gesamtarchitektur der TBC in Abschnitt 4.3 stellt Abschnitt 4.4 den eigentlichen Entwurf und die Implementierung der verschiedenen Teilsysteme dar. Dabei werden jeweils die Anforderungen, die verwendeten Techniken sowie die Ergebnisse detailliert angegeben. In Abschnitt 4.5 werden weiterführende Konzepte, die u.a. die Evolution betreffen, und die zur Realisierung verwendbaren Techniken vorgestellt. Abgeschlossen wird das Kapitel mit einer Bewertung des entworfenen Systems in Abschnitt 4.6.

4.1 Umfang

Grundsätzlich soll wie in Abschnitt 2.4 dargestellt das in Abschnitt 2.2 vorgestellte Modell der *Business Conversations* die Grundlage der Interaktion des zu entwerfenden Agentensystems sein. Der Umfang des zu entwerfenden und zu implementierenden Systems läßt sich folgendermaßen abgrenzen:

- Die Konversationsspezifikationen und die Konversationsinstanzen als ihre konkreten Ausprägungen sollen durch Klassen und Objekte in TL-2 repräsentiert werden. Die beiden wichtigsten Anforderungen dabei sind, daß

    1. insbesondere die Spezifikationen eine Laufzeitrepräsentation besitzen und

2. die Spezifikationen und ihre Instanzen unabhängig vom restlichen System entworfen und implementiert werden.

Die erste Forderung ermöglicht reflektive Dienste für Spezifikationen, die zweite, daß Konversationsspezifikationen und -instanzen ggf. auch später unabhängig für weitere Projekte wiederverwendet werden können.

- Das zu entwerfende Agentensystem soll als *framework* die schnelle Realisierung von anwendungsspezifischen Agenten und Diensten mit so wenig Aufwand wie möglich gestatten. Ziel ist es, die gesamte für Agenten spezifische Funktionalität sowie alle technischen Details der Interaktion (die den *Business Conversations* gehorcht) bereitzustellen und – so weit möglich und sinnvoll – vor dem Benutzer zu verbergen.
- Zur Erprobung sowohl des Modells als auch des implementierten Systems wird ein prototypisches System, das Hotelreservierungssystem, realisiert.

Tiefergehende und speziellere Anforderungen werden in den entsprechenden Abschnitten aufgeführt. Die Realisierung des Hotelreservierungssystems wird separat im darauffolgenden Kapitel 5 beschrieben.

4.2 Methodik

Die Technik der objektorientierten Analyse, des Entwurfs und der Implementierung erfreut sich in der letzten Zeit hohen Interesses. Zur Zeit zeichnet sich sogar eine Standardisierung der in diesem Umfeld gängigen Notation ab. In Gang gesetzt wurde sie durch die vereinten Bemühungen von BOOCH, JACOBSON und RUMBAUGH, nachdem diese drei zunächst unabhängig voneinander ihre Methodik entwickelt hatten [Boo94, JCJÖ92, RBP+91]. Der von ihnen zur Zeit erarbeitete Standard der Notation nennt sich *unified modelling language* (UML) und soll hier verwendet werden [BJR96, FS97]. Insbesondere die bisher sehr uneinheitlich gebrauchte Terminologie hilft UML zu standardisieren.

Zusätzlich zur Standardisierung der Notation wird auch versucht, ein Vorgehensmodell für den *Prozeß* der Analyse, des Entwurfs und der Implementierung zu definieren. Schwerpunkt der Analysephase ist dabei häufig, die in der realen Welt ablaufenden Prozesse und dabei auftretende Interaktionen zwischen Benutzern und dem zu entwerfenden System mit Hilfe sogenannter *Szenarien* (*use-cases*) zu beschreiben [JCJÖ92]. Dies dient hauptsächlich dazu, die Anforderungen zu verstehen und genau festzulegen.

Das hier zu entwickelnde System und das TYCOON-2-System haben eine Charakteristik, die ein zum Teil anderes Vorgehen nahelegt.

Das Ausarbeiten von Szenarien ist für die Analyse von Systemen, mit denen menschliche Benutzer interagieren, günstig. Da hier aber *middleware*, also Software, die nicht direkt mit Benutzern kommuniziert, entwickelt werden soll, ist das Vorgehen per Szenario nicht gut geeignet. Insgesamt stellt sich also die Analysephase völlig anders dar, da alle wesentlichen Anforderungen bereits durch die in Kapitel 2 genannten Merkmale feststehen bzw. sich erst auf technischer Ebene bemerkbar machen. Eine klassische Analysephase findet hier deshalb nicht statt.

Eine weitere Besonderheit der Notation – die sich als Vorteil herausstellen soll – offenbart sich durch die sprachlichen Eigenschaften von TL-2. Die vielfach streng durchgeführte Trennung zwischen Entwurfs- und Implementierungsdiagrammen, insbesondere bei den Klassendiagrammen, erweist sich hier als unnötig. In der Regel dienen Entwurfsdiagramme dazu, das System auf hoher Ebene und meist sogar sprachunabhängig zu modellieren. Erst die Wahl der Implementierungssprache und -plattform erlegt zusätzliche Einschränkungen auf, die in der Regel zu einem weiteren Modellierungsschritt führen, um diesen Gegebenheiten gerecht zu werden: es entstehen Implementierungsdiagramme. Es hat sich – um dieses Ergebnis vorwegzunehmen – gezeigt, daß die sprachliche Mächtigkeit von TL-2 dazu führt, daß nahezu alle Entwürfe ohne Änderungen implementiert werden können. Zusätzlich begünstigt wird dies auch durch die relativ niedrige Art des entworfenen Systems (*middleware*). Daher entfällt die Notwendigkeit getrennter Darstellungen gänzlich. Als sprachliche Mittel, die dies ermöglichen, sind insbesondere die Mehrfachvererbung und der parametrische Polymorphismus zu nennen.

Die im folgenden wiedergegebenen Klassendiagramme stellen also im Prinzip Entwurf und Implementierung gleichzeitig dar. Die noch bestehenden und zumeist später nicht weiter angesprochenen Unterschiede sollen jetzt kurz skizziert werden. Die in Abschnitt 3.2 angesprochene *Referenzsemantik* der Objekte in TL-2 sorgt dafür, daß mache der in UML möglichen verschiedenen Arten der Beziehungen, insbesondere nämlich die Aggregation und die Assoziation, auf die gleiche Weise implementiert werden. Die Aggregation von mehreren anderen Objekten wird beispielsweise meist durch Verwendung eines *Behälters* realisiert, z.B. der Klasse `Dictionary` oder `Set`, und zwar entweder als Attribut oder durch Vererbung. Einwertige Beziehungen und Referenzen auf der einzelnen Seite einer 1:n-Beziehung sowie Aggregation einzelner Objekte lassen sich in TL-2 grundsätzlich nur als Attribute (*slots*), die jeweils eine Objektreferenz enthalten, realisieren. Diese uniforme Realisierung in neue Implementierungsdiagramme zu fassen (was lediglich ein mechanischer Schritt wäre), schafft kein besseres Verständnis für das System und wird deshalb hier unterlassen.

4.3 Architektur

In diesem Abschnitt wird die Gesamtarchitektur des zu entwerfenden Systems auf höchster Ebene dargestellt.

Abbildung 4.1 zeigt die gewählte Schichtenarchitektur. Jede Schicht hat die Aufgabe, die von ihr angebotenen Dienste unter Nutzung der darunterliegenden Schichten transparent für die darüberliegenden Schichten zu erbringen. Zugleich wird deutlich, daß die Schichtung in dem Kommunikationsmodell dafür sorgt, daß zwei auf der obersten Ebene der Anwendungssysteme angesiedelte Kommunikationspartner über eine virtuelle Verbindung, die durch die Dienste der unteren Schichten transparent erbracht wird, kommunizieren können. Die Unterscheidung zwischen *Kunde* und *Dienstleister* entfällt bei der Betrachtung der Architektur, da auf beiden Seiten die generischen Dienste für beide Belange vorhanden sind. Im einzelnen müssen die Schichten folgendes leisten:

Kommunikationsinfrastruktur: Sie sorgt unter Nutzung der vom TYCOON-2-System bereitgestellten Basisdienste für den Aufbau und den Betrieb von Kommunikationsver-

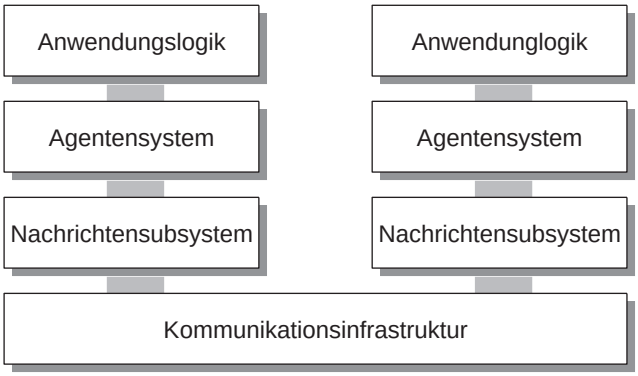


Abbildung 4.1: Architektur des Systems

bindungen zwischen zwei oder mehr verschiedenen TYCOON-2-Systemen (oder interoperablen Systemen), die auch auf unterschiedlichen Rechnern liegen können. Dazu nötig ist die Nutzung von Netzwerkdiensten, niederen Kommunikationsprotokollen (z.B. TCP/IP) und der Möglichkeit, TL-2-Objekte zu *linearisieren* (serialisieren), um sie in dieser Form versenden und wieder rekonstruieren zu können. Da letzteres im TYCOON-2-System noch nicht realisiert ist und dies den Rahmen dieser Arbeit auch bei weitem überschreiten würde, bleibt der Wirkungsbereich der Agenten zunächst auf ein System beschränkt. Natürlich wird der Rest des Systems so entworfen, als ob diese Einschränkung nicht existierte.

Nachrichtensubsystem: Dieses bildet eine Abstraktionsschicht über der Kommunikationsinfrastruktur, die für *Ortstransparenz* und *Migrationstransparenz* sorgt, indem durch die Bereitstellung und Nutzung geeigneter *Namensdienste* davon abstrahiert werden kann, wo genau sich von Agenten adressierte andere Agenten und Dienste befinden. So ist die uniforme Behandlung naher und ferner Agenten möglich. Bedingt durch das Konzept autonomer, mobiler Agenten scheiden verbindungsorientierte Kommunikationsformen als Grundlage aus. Deshalb leistet diese Schicht die gesamte Abwicklung des Nachrichtenaustausches zwischen Agenten und auch innerhalb von Agenten in asynchroner Art durch Nutzung von Warteschlangen und einem *store-and-forward*-Protokoll.

Agentensystem: Dies ist der wichtigste Bestandteil des Systems. Seine Architekturmerkmale basieren auf den Erfahrungen in [Joh97], die ihrerseits stark auf das TELESCRIPT-System sowie [Mat96] zurückgehen. Im einzelnen sind folgende Merkmale zu nennen:

- Anwendungsnahe und migrationsorientierte logische Strukturierung. Dies soll bedeuten, daß das System *Domänen* und innerhalb derer auch noch hierarchische *Orte* unterstützt, die eine räumliche Metapher (also eigentlich Namensräume) der „Agentenwelt“ darstellen.
- Anwendungsnahe Kooperation und Interaktion. Zu diesem Zwecke wird das Modell der *Business Conversations* implementiert, mit dessen Hilfe die Agenten untereinander kommunizieren.

- Unterstützung für die Erzeugung von Agenten. Dies umfaßt einerseits z.B. Basisklassen zum raschen Aufbau eigener Anwendungen in Form von Agenten und zum anderen eine einfache Schnittstelle zum Agentensystem, um die erzeugten Agenten gut administrieren zu können.

Besonders wichtig ist die geeignete Umsetzung der Anforderungen, die sich aus den in Abschnitt 2.3.1 gefundenen Kernkonzepten ableiten, z.B. der Forderung nach asynchronen, bindungsarmen und autonomieerhaltenden Kommunikationsformen.

Anwendungslogik: Ein wichtiges Entwurfsziel ist es, die Anwendungsentwicklung nicht durch die Programmierung von Agenten zu behindern. Die dazu vom Agentensystem bereitgestellten Muster von Agenten (in [Joh97] auch als „Agentenskelett“ bezeichnet) müssen demzufolge nur noch mit der gewünschten Anwendungslogik parametrisiert werden. Eine geeignete Schnittstelle dafür muß zwischen dem Agentensystem und der Anwendungslogik vorhanden sein.

Nicht exakt einer Schicht zuordnen lassen sich die wie eingangs gefordert unabhängig zu realisierenden Konversationsspezifikationen und -instanzen. Sie sind zwar von keinem der Teilsystem abhängig, werden aber selbst von fast allen Teilen verwendet.

4.4 Entwurf und Implementierung

In den nachfolgenden Abschnitten werden die Entwürfe und die Implementierung für die eben aufgeführten Teilsysteme oder Schichten eingehend dargestellt. Begonnen wird mit den Konversationsspezifikationen und -instanzen. Darauf folgen die Beschreibungen des Agentensystems und des Nachrichtensubsystems. Für alle Teilsysteme werden Klassendiagramme angegeben sowie wo nötig Interaktionsdiagramme zur Erläuterung des dynamischen Ablaufes. Abgerundet wird die Darstellung durch einige kleine Programmbeispiele in TL-2-Code.

4.4.1 Die Konversationsspezifikationen und -instanzen

An die in Abschnitt 2.2.5 definierten Konversationsspezifikationen sowie ihre konkreten Ausprägungen, die Konversationsinstanzen, sind folgende Anforderungen zu stellen: Die Konversationsspezifikationen müssen eine Laufzeitrepräsentation besitzen, die die reflektive, dynamische Inspektion des Aufbaus zuläßt, und sie müssen selbst als Objekte erster Klasse in ihrem eigenen Spezifikationssystem erscheinen. Anderenfalls wären weder Konformitätsprüfungen noch Metadienste möglich.

Die Lösung besteht darin, zwei duale, miteinander korrespondierende Klassenstrukturen zu entwerfen, die das Typsystem und das Wertesystem implementieren. Tatsächlich lassen sich zwei – bis auf die Einfügung von abstrakten Klassen, die zur Definition gemeinsamer Funktionalität dienen – isomorphe Klassenbäume definieren, die im folgenden erläutert werden (siehe Abbildung 4.2 und 4.3). Die Klassennamen in beiden Abbildungen sind dabei aus

Platzgründen zum Teil verkürzt wiedergegeben; außerdem sind der Übersichtlichkeit halber einfache Klassen, die als *mixin* dienen, weggelassen worden.

Eine Konversationsspezifikation wird durch eine Instanz der Klasse `ConversationSpec` dargestellt. Sie aggregiert eine Reihe von Dialogen (`DialogSpec`), die ihrerseits Anfragen (`RequestSpec`) und Inhalte (`ContentSpec`) aggregieren. Jede Konversationsspezifikation besitzt einen Namen sowie einen Verweis auf den initialen Dialog. Jede Anfrage besitzt neben ihrem Namen eine Liste von möglichen Folgedialogen. Eine Dialogspezifikation hat einen Namen und aggregiert benannte Inhaltstypen. Die abstrakte Klasse `ContentSpec` stellt die Wurzel der Klassenhierarchie dar, die das Typsystem der Inhalte bildet. In diesem Klassenbaum eingeschoben sind nochmals einige abstrakte Klassen, die Gemeinsamkeiten der einzelnen Inhaltstypen realisieren.

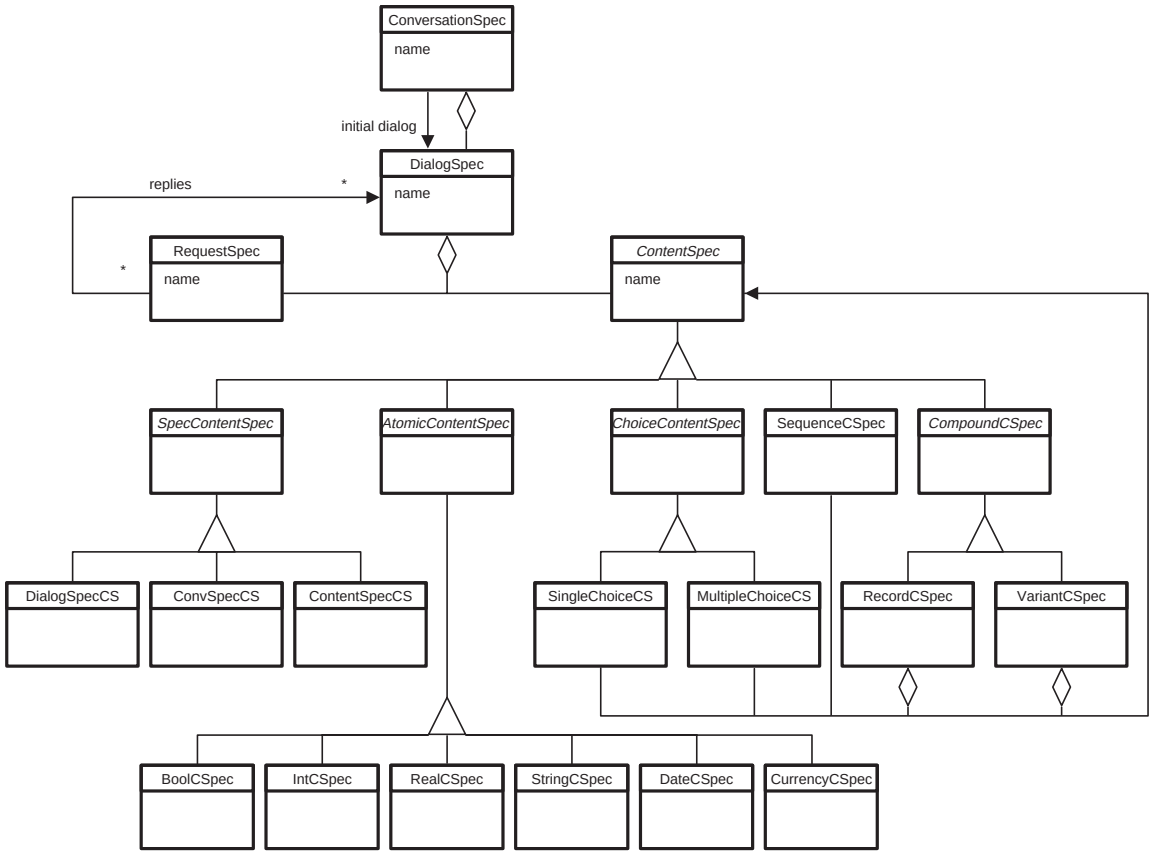


Abbildung 4.2: Klassendiagramm der Konversationsspezifikationen

Für die Subklassen von `CompoundContentSpec` wird das sogenannte *composite-pattern* verwendet [GHJV95]. Realisiert sind diese beiden Klassen durch die Ererbung der generischen Behälterklasse `Dictionary` (die nicht abgebildet ist), die das Eintragen, Auffinden und Löschen von Einträgen mit Hilfe eines Schlüsselelements gestattet. Der Schlüssel ist in diesem Fall der Name des Elements, repräsentiert als Zeichenkette (`String`), der Inhalt ist wieder vom Typ `ContentSpec`. Tatsächlich ist auch die Menge der benannten Inhaltstypen eines Dialoges durch ein Attribut vom Typ `RecordContentSpec` realisiert.

Die Auswahltypen `SingleChoiceContentSpec` und `MultipleChoiceContentSpec` sowie der Massendatentyp `SequenceContentSpec` referenzieren jeweils genau einen Typ. Dazu enthalten sie ein Attribut vom Typ `ContentSpec`, das diesen Typ referenziert. Gesetzt wird er bei der Erzeugung einer Instanz durch Übergabe an den Konstruktor `new` der Metaklasse (die gleichfalls – wie alle Metaklassen – nicht abgebildet ist). Die Subtypen von `SpecContentSpec` dienen dazu, die Spezifikationen zu spezifizieren. Im einzelnen sind Konversations-, Dialog- und Inhaltsspezifikationen als Typ möglich. Die sechs atomaren Basistypen `Bool-`, `Int-`, `Real-`, `String-`, `Date-` und `CurrencyContentSpec` sind Subtypen von `AtomicContentSpec`.

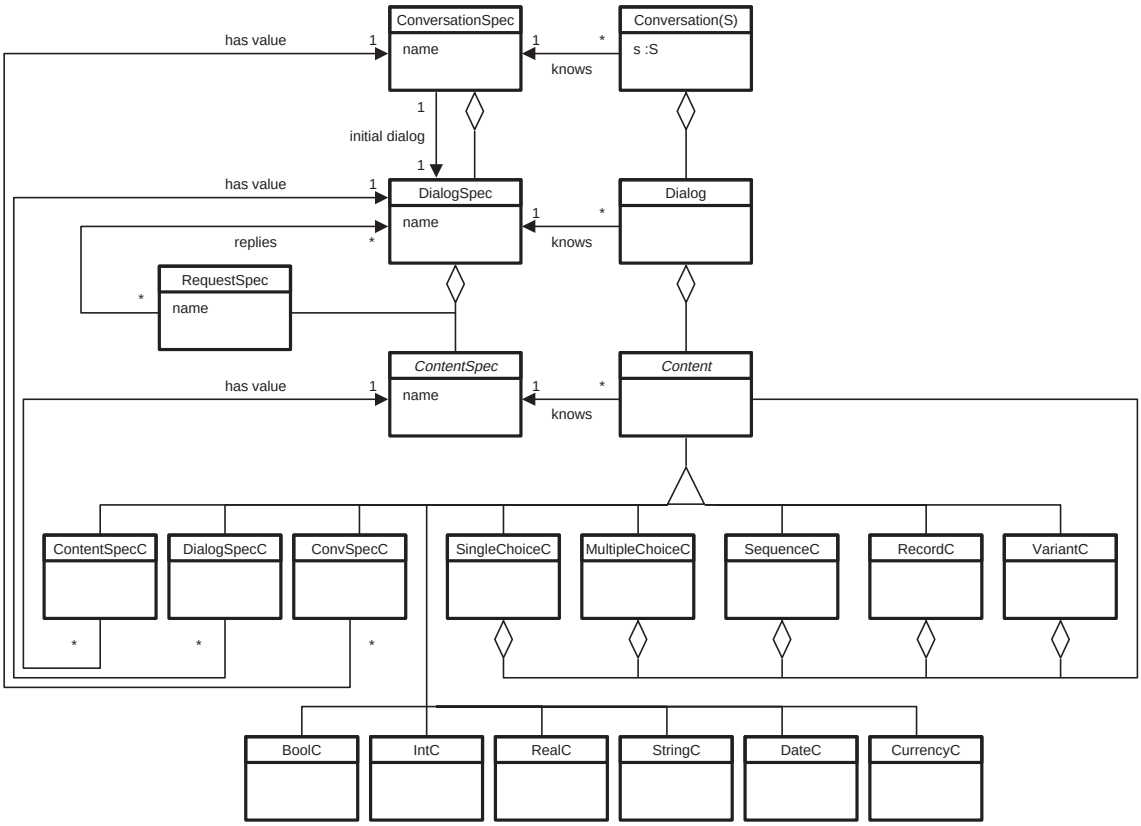


Abbildung 4.3: Klassendiagramm der Konversationsinstanzen

Die Klassenstruktur der Konversationsinstanzen ist analog aufgebaut (siehe Abbildung 4.3). Die Dualität zwischen Spezifikationen und Instanzen ist hier gut zu erkennen. Dies gilt für alle Ebenen, also sowohl die Konversationsspezifikationen und -instanzen, die Dialogspezifikationen und -instanzen, als auch die Inhaltsspezifikationen und -instanzen. Um die reflektiven Eigenschaften zu ermöglichen, „kennt“ jede Instanz auf jeder dieser Ebenen ihre eigene Spezifikation. Dies wird durch das Vorhandensein einer unidirektionale Referenz auf das Objekt der Spezifikation gewährleistet. Diese wird bei der Erzeugung eines Instanz-Objektes automatisch eingetragen (s.u.). Dadurch kann jede Konversationsspezifikation beliebig viele Instanzen besitzen.

Erzeugung und Verwendung

Das allen Inhaltsspezifikationen gemeinsame Protokoll ist in `ContentSpec` definiert. Diese Klasse ist hier auszugsweise wiedergegeben:

```
class ContentSpec
super NamedMixin
metaclass AbstractClass

public methods

  visit(visitor :ContentSpecVisitor) :Void
    deferred

  matches(other :ContentSpec) :Bool
    deferred

  instance() :Content
    deferred
  ...
```

Die Konversations- und Dialogspezifikationen haben äquivalente Methoden. Die Methode namens `matches` dient der Konformitätsprüfung zweier Spezifikationen. Das Verfahren wird in Abschnitt 4.5 erläutert. Von großer Wichtigkeit ist die Methode `instance`. Sie dient dazu, eine Inhaltsinstanz der jeweiligen Spezifikation zu erzeugen. Diese Methode muß in jeder konkreten Klasse entsprechend definiert werden. Die Erzeugung und Benutzung von Spezifikationen und ihren Instanzen wird in folgendem Beispiel illustriert:

```
let cs = IntContentSpec.new,
let c  = cs.instance,
...
```

Die erste Zeile erzeugt dabei eine Spezifikation vom Typ `IntContentSpec`, die also den Typ *Ganzzahl* repräsentiert, und die zweite Zeile instantiiert dann einen Wert dieses Typs, der dann vom Typ `IntContent` ist.

Im Sinne der Entwurfsmuster ist die Spezifikation eine *factory* (Fabrik), die die entsprechenden Werte zu erzeugen in der Lage ist. Im Sinne von Programmiersprachen und Typsystemen ist sie der *Typ* eines Wertes, und die TL-2-Instanz `cs` der Spezifikation kann als Klassenobjekt der Metaklasse `IntContentSpec` mit der Konstruktormethode `instance` aufgefaßt werden.

Da auch die Inhaltsspezifikationen von zusammengesetzten Typen für die Erzeugung der Inhaltsinstanzen verantwortlich sind, erzeugen sie per rekursiver Delegation ganze komplexe Werte. Folgendes Beispiel veranschaulicht dies:

```
let personSpec = RecordContentSpec.new,
personSpec["Name"]:= StringContentSpec.new,
personSpec["Geburtstag"]:= DateContentSpec.new,
```

```
...
let person = personSpec.instance,
person["Name"].str:= "Karin",
...
```

Zunächst wird an den Bezeichner `personSpec` eine Record-Inhaltsspezifikation gebunden, die dann zwei Komponenten unterschiedlichen Typs mit den Namen „Name“ und „Geburtstag“ erhält. Im unteren Teil dann wird eine Inhaltsinstanz der oben definierten Inhaltsspezifikation erzeugt. Die `instance`-Methode erzeugt dabei nicht nur die reine Record-Instanz, sondern auch automatisch die Instanzen aller enthaltenen Komponenten. Dazu bedient sie sich natürlich der Inhaltsspezifikation. Die Implementierung aller `instance`-Methoden ist trivial: Sie alle rufen lediglich den Konstruktor der korrelierenden Konversations- oder Inhaltsinstanz mit `self`, also der Spezifikation, als Parameter auf. Der Konstruktor ist dann für das der Spezifikation entsprechende Erstellen des Wertes verantwortlich. Als Beispiel dafür ist abschließend die Metaklasse von `RecordContent` wiedergegeben:

```
class RecordContentClass
super ConcreteClass(RecordContent)
metaclass MetaClass

public methods

  new(aRSpec :RecordContentSpec): RecordContent
  {
    let inst = _new,                              (* Objekt erzeugen *)
    inst._init,
    inst._init1(aRSpec.size),
    inst.spec := aRSpec,                      (* Spezifikation setzen *)
    aRSpec.keysAndElementsDo(                 (* Komponenten erzeugen *)
      fun(name :String, cs :ContentSpec) :Void {
        inst[name] := cs.instance()
    }),
    inst                    (* Ergebnis ist die neue Inhaltsinstanz *)
  }
```

An diesen Beispielen sieht man zusätzlich sehr gut, wie das ererbte `Dictionary` in den Klassen `RecordContentSpec` und `RecordContent` verwendet wird. Wegen der Mehrfachvererbung müssen hier zwei Initialisierungsmethoden aufgerufen werden; eine parameterlose und eine mit einem Parameter. Die letztere stammt dabei aus dem `Dictionary`. Dieser Stil stellt sich als im Nachhinein als unvorteilhaft heraus; besser wäre es tatsächlich, das Wörterbuch als Attribut von `RecordContentSpec`zu führen.

Jede Konversationsinstanz vom Typ `Conversation` einer Konversationsspezifikation enthält neben dem Verweis auf die Spezifikation ein Protokoll (*history*) der bisher ausgeführten Konversation sowie Methoden für den Zugriff darauf. Weiterhin enthält diese Klasse einige unterstützende Methoden z.B. zum Erzeugen von spezifikationsgemäßen Dialogen, wobei gewährleistet wird, daß sie bei dem aktuellen Stand der Konversation legal sind.

Zugriff auf Werte von Instanzen

Ein aufgetauchtes Problem und seine Lösung sollen hier noch dargestellt werden. Alle von `Content`, der abstrakten Basisklasse der Inhaltsinstanzen, abgeleiteten konkreten Klassen, insbesondere die atomaren, besitzen ein Attribut (*slot*), das den eigentlichen Wert enthält. Dieses Attribut ist entsprechend typisiert, also z.B. eine Ganzzahl (`Int`) bei `IntContent`. Aufgrund der statischen Typisierung in TL-2 kann aber niemals typsicher auf dieses Attribut eines Objektes zugegriffen werden, sofern man nur die allgemeine Annahme, daß es sich bei dem Objekt um eines der Klasse `Content` handelt, zugrundelegt. Diese allgemeine Annahme ist aber z.B. immer dann schon gegeben, wenn ein Element eines zusammengesetzten Wertes betrachtet wird, etwa eine Recordkomponente, da diese alle als `Content` typisiert sind. Das Wissen, daß es sich garantiert z.B. um ein `StringContent` handelt, der unter dem Namen „Name“ abgelegt ist, bleibt der statischen Typisierung natürlich verborgen. Die Lösung ist, wie bereits in dem Record-Beispiel angedeutet, in der Superklasse `Content` entsprechend typisierte Zugriffsmethoden zum Setzen und Auslesen aller Wertetypen abstrakt zu definieren. In den entsprechenden konkreten Subklassen werden diese Methoden dann erst fallweise definiert.

```
class Content
metaclass AbstractClass
  ...
  "int:="(i :Int) :Void  deferred
  int() :Int  deferred

  "str:="(s :String) :Void  deferred
  str() :String  deferred
  ...
  "cvspec:="(c :ConversationSpec) :Void  deferred
  cvspec() :ConversationSpec  deferred
  ...
  rec() :RecordContent      deferred
  variant() :VariantContent  deferred
  seq() :SequenceContent      deferred
  sch() :SingleChoiceContent   deferred
  mch() :MultipleChoiceContent  deferred
  ...
```

Zur Illustration ist oben ein Ausschnitt von `Content` und unten ein Ausschnitt aus einer der Subklassen (`StringContent`) wiedergegeben:

```
class StringContent
super Content
metaclass ContentClass(StringContent)

public methods
  ...
  "str:="(s :String) :Void {
    _value:=s
  }
```

```
    str() :String {
      _value
    }

  private

    _value :String
```

So ist jetzt auch verständlich, wieso die Zuweisung in dem zweiten Beispiel zur Wertkonstruktion `person["Name"].str:= ...` lauten mußte. Wird allerdings irrtümlich z.B. die Methode `str` eines Objekts vom Typ `IntContent` aufgerufen, so erzeugt das Laufzeitsystem eine Ausnahme, da die Methode in dieser Klasse nicht definiert ist. In gewisser Weise ist diese Lösung mit den *varianten Records* vergleichbar, die in der Vorgängersprache TL existieren. Eine weitergehende, allerdings nicht realisierte Möglichkeit wäre, all diese Methoden in allen Klassen zu definieren, um damit Konvertierungen zwischen den Werten vornehmen zu können: So müßte dann etwa `str`, angewandt auf ein `IntContent`-Objekt, den numerischen Wert als Zeichenkette zurückliefern. Die kombinatorische Explosion der Möglichkeiten und die in vielen Fällen unklare Semantik einer solchen Konvertierung aber sprachen gegen die Realisierung.

Das Besucher-Muster

Zum Schluß soll noch ein weiteres verwendetes Entwurfsmuster, das *visitor-pattern*, vorgestellt werden [GHJV95]. Sinn dieses Musters ist es, auf einer komplexen, häufig dem *composite-pattern* entsprechenden Struktur neue Operationen definieren zu können, ohne die Klassen der Struktur selbst ändern zu müssen. Die gesamte neu zu definierende Funktionalität wird dabei in nur einer neuen Klasse gekapselt, die Subtyp einer abstrakten, sogenannten *visitor*-Klasse ist. Der Aufruf geschieht über eine generische Schnittstelle, die die Struktur bereithält. Dazu definiert die Wurzelklasse der zu besuchenden Struktur eine evtl. aufgeschobene Methode `visit(v :Visitor)`, die eine Instanz des Besuchers übergeben bekommt. Alle Subklassen implementieren diese Methode durch Aufruf einer ihnen entsprechenden Methode des Besuchers. In Abbildung 4.4 sind die Zusammenhänge dargestellt.

Wenn die Besucher-Methoden für zusammengesetzte Strukturen den `visit`-Aufruf weiter an die enthaltenen Unterstrukturen delegieren, so genügt ein Aufruf der `visit`-Methode des jeweiligen Wurzelobjekts, um die gesamte Struktur zu traversieren. Dies ist in Abbildung 4.4 in der `visitRecord`-Methode skizziert. Theoretisch könnte die Delegation auch gleich in den entsprechenden `visit`-Methoden der Struktur stattfinden; dies hätte aber den Nachteil, daß die Art des rekursiven Abstiegs (*preorder*, *inorder*, *postorder* etc.) dadurch festgelegt wäre. Denkbar, aber nicht realisiert, ist die Einführung von Methoden der Struktur, die verschiedene Arten von Iteratoren erzeugen können. Mit Hilfe dieser Iteratoren ließe sich die Struktur dann auf die verschiedenen Weisen traversieren, ohne daß die Delegation Aufgabe der Besucher sein müßte.

So lassen sich also leicht Besucherklassen für verschiedene Zwecke entwickeln. Realisiert sind bereits Klassen, die Konversationsspezifikationen und Konversationsinstanzen aus-

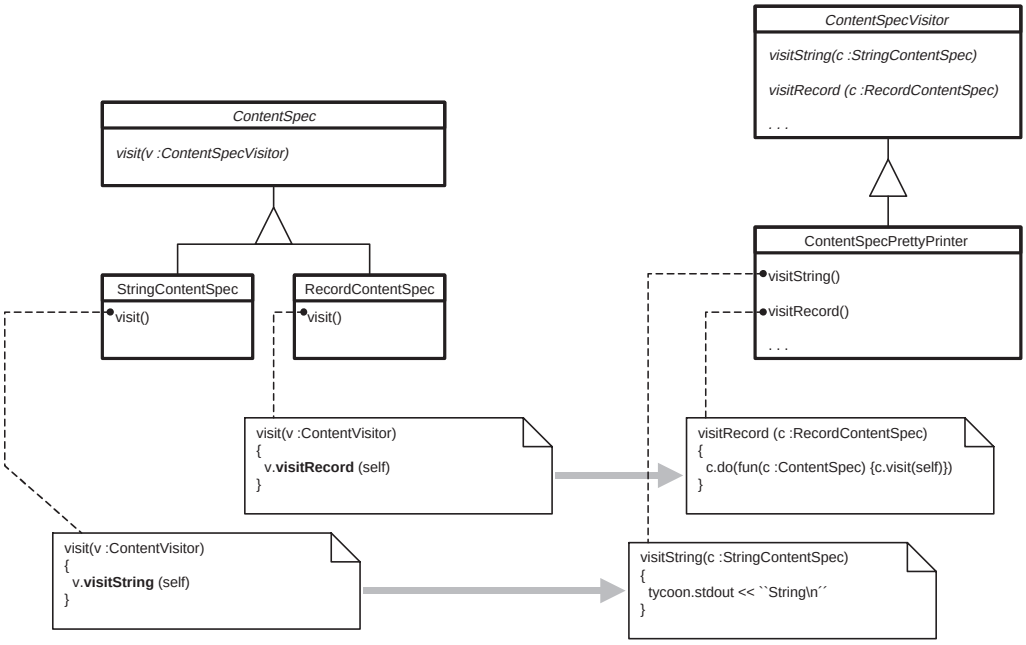


Abbildung 4.4: Das *visitor-pattern* am Beispiel der Inhaltsspezifikationen

drucken (*pretty-printer*). Dazu existieren die abstrakten Besucherklassen `ContentSpecVisitor`, `ContentVisitor` sowie `TBCSpecVisitor`. Die dritte ist durch Ererbung der beiden ersten definiert und ist so in der Lage, Spezifikationen auf jeder Ebene sowie deren Instanzen zu behandeln.

Dem Vorteil der Möglichkeit, einfach neue Funktionalität definieren zu können, steht der Nachteil gegenüber, daß bei Erweiterung der Struktur selbst alle Besucherklassen geändert und erweitert werden müssen.

4.4.2 Das Agentensystem

Das Agentensystem selbst mit seinen vielfältigen Anforderungen stellt den größten und wichtigsten Teil der *Tycoon Business Conversations* dar. Die zentralen Forderungen, denen beim Entwurf Rechnung getragen werden muß, seien im folgenden kurz zusammengestellt:

- Ein Agent soll durch eine Klasse bzw. ein Objekt dargestellt werden. Natürlich kann er weitere Objekte aggregieren. Somit ist der Agent die Einheit der Migration und der Adressierung.
- Jeder Agent soll in der Lage sein, als Kunde oder Dienstleister im Sinne der *Business Conversations* Konversationen mit anderen Agenten führen zu können.
- Jeder Agent soll zusätzlich in beiden Rollen (Kunde und Dienstleister) gleichzeitig agieren können, um so Subkonversationen zu führen. Weiterhin soll er in beliebig

vielen Rollen gleichzeitig auftreten können, um gleichzeitig verschiedene Konversationen mit anderen Agenten zu führen, die unterschiedliche Konversationsspezifikationen verwenden.

- Die Anbindung der Anwendungslogik soll möglichst einfach sein. Der Agent und das Agentensystem sollen einen Rahmen für die schnelle Entwicklung agentenbasierter Informationssysteme darstellen.

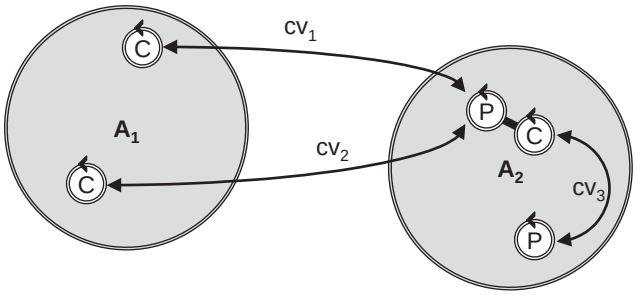


Abbildung 4.5: Struktur der Agenten

Der sich daraus ergebende grundsätzliche Aufbau von Agenten wird in Abbildung 4.5 dargestellt. Jeder Agent (A_1 und A_2) enthält Instanzen, die als Kunde (C) oder Dienstleister (P) agieren und durch Konversationen miteinander verbunden sind (cv_1, cv_2, cv_3). Wie man sieht, sind auch Konversationen innerhalb desselben Agenten möglich (cv_3). Dies kann dazu ausgenutzt werden, um Systeme gemäß den in Abschnitt 2.2.4 genannten Konzepten zu entwerfen. Weiterhin ist zu sehen, daß Subkonversationen durch Koppelung eines Kunden mit einem Dienstleister realisiert werden (dicke Linie in A_2). Die dafür nötigen Mechanismen werden in Abschnitt 4.5 erläutert.

Agenten, Rollen und Regeln

Nach diesem groben Überblick wird in Abbildung 4.6 das Klassendiagramm der Agenten vorgestellt. Die zentrale Klasse ist `Agent`. Jede Instanz aggregiert beliebig viele Rollen, nämlich Instanzen der Klassen `CustomerRole` oder `PerformerRole`, deren gemeinsame Superklasse `Role` ist. Rollen werden stets zusammen mit einer Konversationsspezifikation aggregiert, wobei eine Rolle letztere kennt.

Wie in Abschnitt 2.2.5 erklärt wurde, kann man eine Konversationsspezifikation als nichtdeterministischen endlichen Automaten begreifen. Die Zustände und der Ablauf sind durch die Menge der Dialoge und die Anfragen mit Folgedialogen festgelegt. Die Entscheidung der Zustandsübergänge läßt sich, wie ebenfalls in Abschnitt 2.2.5 angedeutet, als Funktionspaar notieren. Diese Übergangsfunktionen werden deklarativ als *Regeln* implementiert, und zwar als zwei Mengen: die der Kunden- und die der Dienstleisterregeln. Das bedeutet, daß auf der Kundenseite für jeden Dialog eine Regel existieren muß, die ihn erstens entsprechend den Anforderungen der Anwendungslogik modifiziert und zweitens eine der möglichen Anfragen des Dialoges auswählt, und auf der Seite des Dienstleisters muß für jede Anfrage

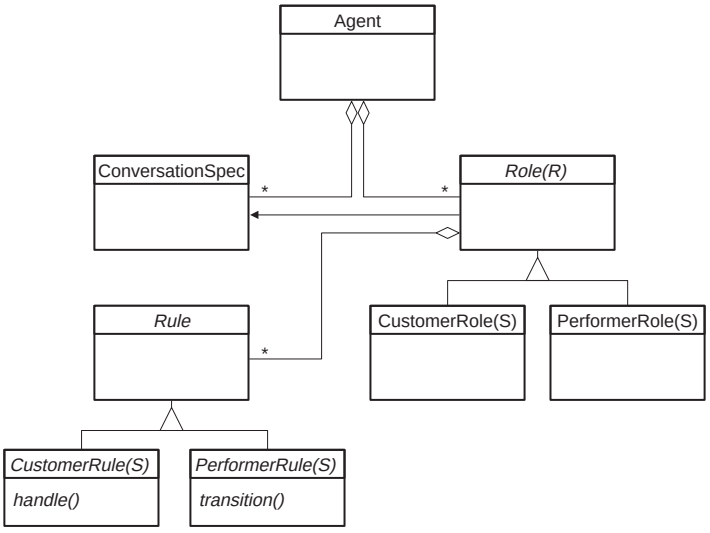


Abbildung 4.6: Klassendiagramm der Agenten

jedes Dialoges eine Regel existieren, die in Abhängigkeit von dem vom Kunden modifizierten Dialog einen der auf die Anfrage möglichen Folgedialoge generiert. Diese Funktionen müssen durch Subklassen von `CustomerRule` und `PerformerRule` implementiert werden, und zwar in den Methoden `handle` und `transition`. Ein Rollenobjekt wird deshalb neben der Konversationsspezifikation auch noch mit den Regeln parametrisiert, und zwar aggregiert eine `CustomerRole` dazu Instanzen von Subklassen von `CustomerRule` und eine `PerformerRole` Instanzen von Subklassen von `PerformerRule`. Diese Regeln erst implementieren die eigentliche Anwendungslogik. Sie zu entwickeln bleibt die Aufgabe der Benutzer des Agentensystems. Genau hier besteht die Schnittstelle zwischen Anwendungslogik und Agentensystem.

Interaktion

Mit diesen Diagrammen und Erklärungen sind erst die statischen Aspekte beschrieben. Um den dynamischen Anforderungen gerecht zu werden, ist hochgradige Nebenläufigkeit im Agentensystem notwendig. Dazu startet jedes instantiierte Agenten-Objekt sofort bei seiner Erzeugung einen eigenen *thread*. Damit kommt das System der Anforderung nach autonomer, vom restlichen System unabhängigen Ausführung nach, da fortan alle Aktionen des Agenten parallel zu den sonstigen Systemaktivitäten ausgeführt werden. Um auch mehrere Konversationen parallel ausführen zu können, wird weiterhin für jede ein weiterer *thread* vom Agenten gestartet. Die Funktionalität dafür ist aufgrund der Asymmetrie der Kommunikation in den Klassen `PerformerRole` und `CustomerRole` definiert, wobei natürlich gemeinsam nutzbare Teile in `Role` definiert sind.

Zum besseren Verständnis des dynamischen Ablaufes einer Konversation zwischen zwei Agenten ist in Abbildung 4.7 ein Interaktionsdiagramm wiedergegeben. Die Notation in dem Diagramm ist dabei gegenüber der üblichen leicht abgewandelt: Da hier sowohl *threads*

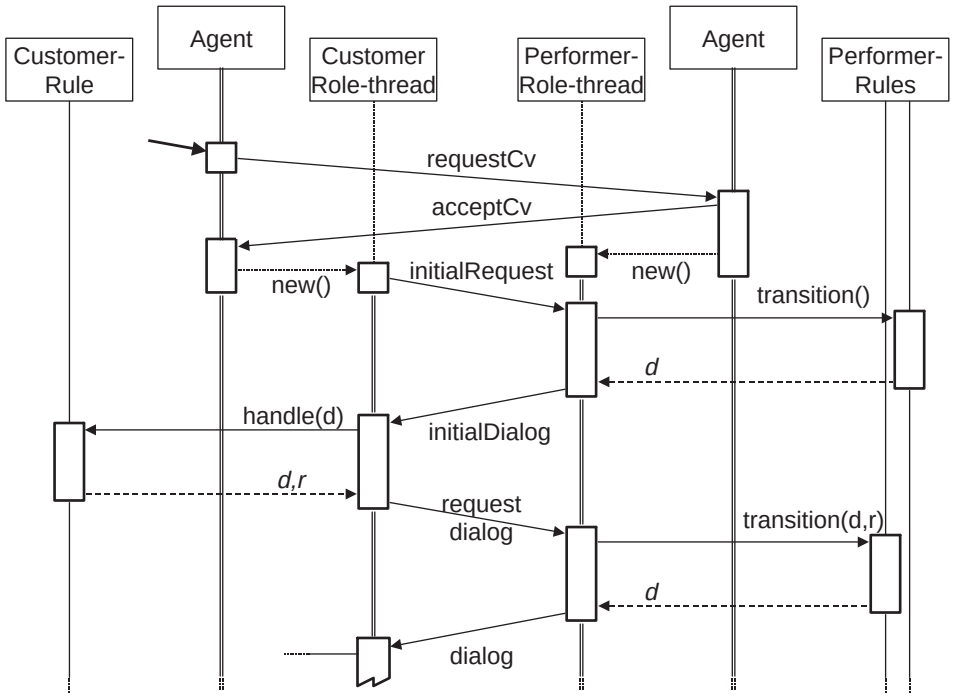


Abbildung 4.7: Interaktionsdiagramm für den Aufbau einer Konversation

miteinander kommunizieren als auch normale Objektinteraktion stattfindet, werden diese unterschiedlich notiert: Die Objektinteraktion in der üblichen Notation und die Kommunikation, die durch den asynchronen Austausch von Nachrichten über das Nachrichtensubsystem geschieht, durch schräge Linien. Die bei Objekten übliche „Lebenslinie“ wird für die *threads* doppelt durchgezogen und hat die Bedeutung, daß sie passiv auf den Eingang einer Nachricht warten. Die Aktivierung eines Objekts entspricht dem Empfang und der Verarbeitung einer Nachricht; beides wird gleich notiert.

Die Konversation wird dadurch begonnen, daß der links gezeichnete Agenten-*thread* dazu angehalten wird, eine solche zu starten – dies ist durch den kleinen, starken Pfeil angedeutet. Daraufhin setzt er sich mit dem anderen Agenten in Verbindung (Nachricht `requestCv`); dabei wird die zu verwendende Konversationsspezifikation mitgeschickt. Hat der andere Agent eine Rolle, die diese Spezifikation als Dienstleister auszuführen vermag, so beantwortet er die Frage positiv (Nachricht *acceptCv*). Beide Agenten instantiieren daraufhin jeweils einen neuen *thread*, der die Konversation im folgenden abwickelt. Genaugenommen – und im Diagramm nicht dargestellt – wird die Abwicklung der Kommunikation und die Auswertung der damit verbundenen Nachrichten von Methoden von `PerformerRole` und `CustomerRole` übernommen, auch wenn der Agentenprozeß die Nachrichten zunächst empfängt. Das verwendete Muster der Delegation wird im nächsten Abschnitt vorgestellt. Da sich die TYCOON-*threads* Funktionen höherer Ordnung bedienen und diese Funktionen innerhalb der Rollenklassen definiert sind, haben sie vollen Zugriff auf alle Attribute der sie startenden Rolle. Deswegen ist die Bezeichnung *Customer-* bzw. *PerformerRole-thread* gerechtfertigt. Der *thread* des Kunden beginnt nun die eigentliche Konversation mit der Anfrage nach dem initialen Dialog. Die Konversation folgt ab da immer dem folgendem Schema: Der

Dienstleister ruft die entsprechende Dienstleisterregel (durch die Methode `transition`) auf; diese generiert den eigentliche Dialog laut Konversationsspezifikation und belegt dessen Inhalt geeignet vor. Der Dialog wird dann vom Dienstleister an den Kunden geschickt; dieser ruft die Kundenregel für diesen Dialog auf (Methode `handle`), die den Dialog auswertet, ggf. modifiziert und eine der möglichen Anfragen wählt. Anfrage und Dialog werden dann vom Kunden zurück an den Dienstleister geleitet, und der Zyklus beginnt erneut.

Ausführungskontext

Zwei Probleme zeichnen sich hier ab: Die Regeln, die eine Rolle aggregiert, werden von allen laufenden Konversationen (bzw. von den sie repräsentierenden *threads*), die von der Rolle geführt werden, geteilt. Das heißt insbesondere, daß die Methoden in Klassen von Regeln *wiedereintrittsfest* (*reentrant*) programmiert werden müssen. Dies gilt natürlich auch für die von den Regeln evtl. benutzten weiteren Dienste wie Datenbanken etc.

Zum zweiten stellt sich das Problem eines Ausführungskontextes. Der deklarative Stil der Regeln und die Art ihrer Ausführung erschweren die Erzeugung, Verwaltung und besonders die Erreichbarkeit sowohl eines für alle Regeln gemeinsamen als auch eines sitzungsspezifischen (also nur auf eine Konversation beschränkten) Arbeitskontexts. Dieser könnte nötig sein, um z.B. weitere Dienste wie Datenbanken, Kommunikationsverbindungen etc. zu verwenden. Da – wie auch in der Abbildung angedeutet – die Möglichkeiten bestehen, der Rolle als Regeln mehrfach dasselbe Objekt, aber auch jeder Regel ein eigenes Objekt zuzuordnen, erschwert sich das Aufrechterhalten des Kontextes zusätzlich. Hier wurde folgende Lösung gewählt: Jeder Rollenprozeß erzeugt ohnehin für seine Konversation (Sitzung) eine Konversationsinstanz der Klasse `Conversation` (siehe auch Abbildung 4.3). Darin ist nun ein Attribut einer vom Verwender des Agentensystems frei wählbaren Klasse eingebettet, was durch den parametrischen Polymorphismus ermöglicht wird. Diese Konversationsinstanz wird nun bei jeder Regelausführung als Parameter an die Methoden `handle` bzw. `transition` übergeben. Dadurch haben alle Regeln, gleichgültig ob immer dasselbe Objekt oder verschiedene zum Einsatz kommen, Zugriff auf immer dasselbe eingebettete Objekt, das dennoch für jede Konversation unterschiedlich ist. Konversationsübergreifende Daten können weiterhin in den Klassen der Regeln definiert werden, wobei die Synchronisierung des Zugriffs zu beachten ist. Die eigentliche Herausforderung beim Entwurf ist nun, dies so gekapselt zu entwerfen, daß das Agentensystem keine Kenntnisse über den Typ dieses Objekts haben muß, aber alles trotzdem statisch typisierbar ist. Dazu dienen die bereits in den Klassendiagrammen angegebenen Typparameter, die dort alle `S` heißen. Zunächst muß `Conversation(S)` mit ihm versehen werden. Da die Konversationsinstanzen von den Rollen erzeugt werden, müssen auch sie ihn kennen (also z.B. `CustomerRole(S)`), und zum Schluß auch die Regeln selbst.

Die Verwendung des Agentensystems, der Rollen und der Regeln wird an einem Beispiel am besten deutlich:

```
let convSpec :ConversationSpec = ...,

(* Create an agent. *)
agent1 := Agent.new("Agent1"),
```

```
(* Create a performer for the conversation-specification above. *)
let aPerf :PerformerRole(TestSession) =
    PerformerRole(:TestSession).new(agent1, "Service", convSpec),

(* Add rules to the performer. *)
aPerf.addRule("", convSpec.initialRequestName, TestPerfRuleInit.new),
aPerf.addRule("Login", "Connect", TestPerfRuleConnect.new),
aPerf.addRule("Login", "Cancel", TestPerfRuleCancel.new),

(* Start conversation. *)
TBC.instance.startConversation("Agent2/Tester", "Agent1/Service"),
```

Die Konversationsspezifikation `convSpec` sei als gegeben angenommen. Zuerst wird ein Agent instantiiert. Dem Konstruktor muß dabei lediglich der Name mitgegeben werden. Als nächstes wird ein Rollenobjekt (`aPerf`) erzeugt, nämlich das eines Dienstleisters. Es ist vom Typ `PerformerRole(TestSession)`; dabei ist `TestSession` der besagte Typ des benutzerdefinierten Kontextes (hier nicht weiter definiert). Dem Konstruktor der Rolle müssen der Name der Rolle (in diesem Sinne also auch des *Dienstes*, den sie bereitstellt), der Agent sowie die Konversationsspezifikation angegeben werden. Es fehlen nun nur noch die Regeln. Sie werden durch mehrfachen Aufruf der Methode `addRule` der Rolle gebunden. Zusätzlich müssen dazu – da es sich hier um einen Dienstleister handelt – der Name des Dialoges und der der Anfrage übergeben werden, für die die als drittes anzugebende Regel gebunden werden soll. Die erste der drei Regeln ist dabei für die initiale Anfrage des Kunden zuständig; hier muß als Dialogname die leere Zeichenkette sowie der spezielle, fest eingebaute Name der initialen Anfrage angegeben werden. Als Regelobjekte werden in diesem Beispiel Instanzen von drei verschiedenen Klassen (die alle Subtyp von `PerformerRule` sind) angegeben. Zum Schluß wird durch einen Aufruf einer Methode des Agentensystems eine Konversation eines in diesem Beispiel nicht gezeigten Kunden mit dem gerade erzeugten Dienstleister gestartet.

Um die Programmierung und Benutzung der Regeln noch deutlicher zu machen, wird jetzt noch die Klasse einer der Regeln wiedergegeben:

```
class TestPerfRuleInit
super PerformerRule(TestSession)
metaclass SimpleConcreteClass(TestPerfRuleInit)

public methods

  transition(conv: Conversation(TestSession),
             dialog :Dialog, reqest: String) :Dialog
  {
    (* create session variable: *)
    conv.s := TestSession.new,
    conv.s.user := "<none yet>",
    (* create initial dialog "Login" *)
    conv.newDialog(conv.convSpec.initialDialogName)
  }
```

Hier ist gut zu sehen, wie die Typisierung des Kontext-Objektes `TestSession` vonstatten geht. Es ist ein Attribut des Parameters `conv` und kann beliebig typsicher verwendet werden. In diesem Beispiel wird es gerade erzeugt und initialisiert (dazu habe `TestSession` ein Attribut `user` vom Typ `String`).

Fehlerbehandlung

Die Fehlerbehandlung spielt eine wichtige Rolle in der Kommunikation. Eine Aufgabe des Agentensystems bzw. der Rollen ist, verschiedene Arten möglicher Fehler zu erkennen, abzufangen und geeignete Maßnahmen einzuleiten. Im einzelnen sind folgende Fehler möglich:

- Verwender des Systems können es versäumen, Regeln für alle möglichen Fälle anzugeben. Zur Zeit wird zur Ermöglichung des schnellen Baus von Prototypen (*rapid prototyping*) nicht geprüft, ob für alle Dialoge und Anfragen auch Regeln an die Rollen gebunden werden. Zusätzlich ermöglicht dies die dynamische Umkonfiguration der Regelbasis einer Rolle, gestattet also gewissermaßen die Selbstmodifikation der Anwendungslogik. Dies ist ein Vorteil und eröffnet weitgehende Möglichkeiten.
- Eine Regel kann – trotz der unterstützenden Methoden der Klasse `Conversation` – Folgedialoge bzw. Anfragen generieren, die nicht der Konversationsspezifikation entsprechen.
- Die Ausführung einer Regel kann eine Ausnahme (*exception*) auslösen. Der normalerweise daraufhin eintretende Abbruch des ausführenden *threads* darf natürlich nicht hingenommen werden, da sonst auch der Betrieb von anderen Konversationen gestört werden kann.

Alle oben aufgeführten Fehler werden sofort vom Agentensystem abgefangen und auf die gleiche Weise behandelt: Die betreffende Konversation wird dadurch abgebrochen, daß dem Kommunikationspartner der besondere, finale Fehlerdialog übermittelt wird (siehe Abschnitt 2.2.5). In dem Dialog werden die Gründe des Abbruchs vermerkt. Die Spezifikation des Fehlerdialoges ist implizit Bestandteil jeder Konversationsspezifikation. Er ist außerdem implizit Folgedialog jeder Anfrage. Seine Dialogspezifikation in Form eines Objekts der Klasse `DialogSpec` ist durch eine Methode von `Conversation` erhältlich, ebenso der Name der zugehörigen Anfrage (diese ist nötig, da beim Auftreten eines Fehlers beim Kunden dem üblichen Schema entsprechend der Fehlerdialog zusammen mit einer Anfrage an den Dienstleister geschickt wird). In allen Fällen ist die Konversation mit Abschicken des Fehlerdialoges für die fehlerhafte Seite und dem Empfang und der evtl. Verarbeitung für die andere Seite beendet. Dem generellen Schema entsprechend, können auch für den Fehlerdialog bzw. die Fehleranfrage Regeln an die Rollen gebunden werden. Dadurch ist eine geordnete Fehlerbehandlung auch auf der Ebene der Anwendungslogik (zumindest auf der nicht fehlerhaften Seite) möglich.

Verwandt mit der Fehlerbehandlung ist folgendes Problem, das bei der Entwicklung des Hotelreservierungssystems diskutiert wurde. Die Frage war, ob z.B. inhaltliche Fehler, die von der Anwendungslogik einer Seite in den Dialoginhalten entdeckt werden (die die andere

Seite produziert hat), durch einen zusätzlichen Mechanismus des Ablehnens und erneuten Anforderns des Dialoges (*reject*) zu behandeln wären. Dadurch könnten ohne Änderung und Aufblähung der Konversationsspezifikation viele typische Fehlersituationen erfaßt werden. Die Alternative ist, solche Ausnahmen in der Konversationsspezifikation selbst zu modellieren. Wir haben uns zunächst dafür entschieden, das zweite Verfahren beizubehalten. Der entscheidende Grund dafür ist, daß die Einführung des ersten Mechanismus bedingt, alle Regeln entsprechende Fallunterscheidungen treffen lassen zu müssen bzw. ein neues Modell sekundärer Ausnahmeregeln schaffen zu müssen. Dieser Aspekt sollte aber in zukünftigen Anwendungen aufmerksam verfolgt werden.

Agentensystem

Das Agentensystem selbst wird durch eine einzelne Klasse repräsentiert, nämlich die Klasse `TBC`. Ihr obliegt zur Zeit lediglich die Verwaltung der im Agentensystem vorkommenden Agenten. Zu diesem Zwecke melden sich instantiierte Agenten selbsttätig beim System an und ggf. wieder ab. Da die Lebensdauer eines Agenten und damit seiner *threads* potentiell unendlich ist, verfügen sie als einzige Möglichkeit zum Stoppen über eine Methode `stop`, die den Prozeß kontrolliert abbricht. Erst wenn dies geschieht, melden sich die Agenten beim System wieder ab. Zusätzlich verfügt das System ebenfalls über eine Methode `stop`, die alle laufenden Agenten beendet. Die benötigten Namensdienste für das Auffinden von Agenten und Rollen sind Bestandteil des Nachrichtensubsystems und werden im nächsten Abschnitt vorgestellt.

Abschließend sollen noch einige Überlegungen zur Mobilität angestellt werden. Da im TYCOON-2-System die Linearisierung von Objekten (Objektserialisierung) noch nicht realisiert worden ist, entfällt die Möglichkeit der Mobilität zur Zeit. Dennoch sind in Hinblick auf die spätere Realisierung bereits einige Vorkehrungen getroffen worden. Diese sowie zukünftige Anforderungen sind folgende:

- Ein *Migrationsprotokoll* für Agenten ist zu entwerfen. Dazu muß insbesondere der Agent Methoden besitzen, die vor und nach einer Migration aufgerufen werden, um etwa notwendigen Maßnahmen Raum zu geben. Dieses Protokoll ist evtl. auch auf die Ebene der Rollen auszudehnen. Die Klasse `Agent` besitzt bereits die drei abstrakt definierte Methoden `leave`, `enter` und `die`, die aufgerufen werden sollen, wenn ein Agent einen Ort verläßt, ihn betritt oder wenn er endgültig beendet wird. Zur ihrer Benutzung müssen Subklassen von `Agent` definiert werden, die diese Methoden dann implementieren.

- Die Migration setzt ein ubiquitäres Agentensystem voraus. Dies erfordert insbesondere, daß technische Mechanismen dafür bereitstehen, daß Bindungen an das Agentensystem sowie an weitere ubiquitäre Betriebsmittel bei der Versendung des Agenten gelöst und beim Empfang durch Bindungen an das dort vorhandene System ersetzt werden können [Mat96].

- Die Migration muß transaktional erfolgen, um der Gefahr von Übertragungsfehlern wie Verlust und Doubletten zu begegnen.

- Die Migration eines Agenten muß sowohl von außen als auch von innen ausgelöst werden können. Das bedeutet, daß sowohl ein Verwender des Agentensystems – etwa durch ein Administrierungswerkzeug für mobile Agenten – als auch eine Kunden- oder Dienstleisterregel aus inneren Gründen die Migration verlangen kann. Die Klasse `Agent` hat dazu die Methode `migrateTo`.

- Eine weitere Anforderung an die Migration ist die, daß sie erst dann stattfinden darf, wenn keine Regel aktiv ist. Das bedeutet, daß eine initiierte Migration aufgeschoben werden muß, bis alle aktuell in Ausführung befindlichen Regeln beendet sind. Problematisch ist hier nur das Verhalten bei einer von einer Regel initiierten Migration. Es gibt prinzipiell zwei Möglichkeiten: die Migration erst stattfinden zu lassen, wenn auch diese Regel beendet ist, oder sie – unter Berücksichtigung der obigen Forderung für andere Regeln – sofort stattfinden zu lassen, so daß gleich nach dem Aufruf der Methode `migrateTo` der Agent bereits samt Regeln am neuen Ort ist und die weitere Ausführung der Regel dort stattfindet.

- Geeignete Mechanismen für die migrationstransparente Kommunikation müssen vorhanden sein. Da zum Zeitpunkt der Migration etliche Konversationen im Gange sein können, muß das zugrundeliegende Kommunikationsprotokoll derart beschaffen sein, daß es den zwischenzeitlichen Ortswechsel eines Partners transparent für die anderen Partner bewerkstelligen kann.

In diesem System ebenfalls noch nicht realisiert ist ein hierarchisches Schema zur Benennung von Orten (*places*), an denen sich die Agenten aufhalten können. Zur Zeit ist das Agentensystem in dieser Hinsicht flach strukturiert. Der in [Joh97] realisierte Ansatz sieht darüber hinaus vor, daß alle diese Orte von jeweils einem speziellen, immobilen Agenten „bewacht“ werden, dem sogenannten *Ortewächter*. Ihm obliegt die Kontrolle des Zugangs mobiler Agenten von anderen Orten. Das oben geforderte Migrationsprotokoll wird dabei uniform in allen Fällen genutzt, gleichgültig ob ein Agent aus einer anderen Domäne (d.h. einem anderen Agentensystem) oder einem anderen Ort desselben Agentensystems stammt. Erstrebenswert wäre bei einer Realisierung dieses Orte-Konzepts eine Spezifikation des Protokolls zwischen Agenten, Agentensystem und Ortewächtern in Form von Konversationsspezifikationen. Dafür erforderlich und gewinnbringend einsetzbar sind die vorhandenen reflektiven Eigenschaften des Typsystems der Spezifikationen.

4.4.3 Das Nachrichtensubsystem

Das Nachrichtensubsystem bildet eine Abstraktionsschicht über der Kommunikationsinfrastruktur. Da letztere aufgrund der fehlenden Mobilität noch nicht verwendet werden muß, stellt das Nachrichtensubsystem die z.Z. unterste Schicht dar. Folgende Aufgaben und Anforderungen muß es erfüllen:

- Durch Bereitstellung von geeigneten Namensdiensten soll das Auffinden von Agenten, Agentensystemen und Diensten in Form von Rollen ermöglicht werden. Das Ziel ist dabei, völlige *Ortstransparenz* zu schaffen, so daß davon abstrahiert werden kann, wo genau sich die adressierten Einheiten befinden.

- Die Abwicklung des gesamten asynchronen Nachrichtenaustausches zwischen den Agenten und auch innerhalb von Agenten ist dabei primäre Aufgabe des Nachrichtensubsystems. Der nebenläufige Charakter der interagierenden Kommunikationspartner ist dabei zu beachten.
- Die Kommunikation soll bindungsarm und somit migrationstolerant sein. Das heißt, daß die o.g. Dienste weitgehend auf direkte Objektreferenzen verzichten müssen und die Adressierung hauptsächlich über symbolische Namen geschehen muß.
- Trotz der niedrigen Ebene dieser Schicht und der hohen Anzahl der zu verarbeitenden Arten von Nachrichten soll sie natürlich mit Abstraktionsmitteln hohen Niveaus, etwa generischen Klassen und Entwurfsmustern, typsicher entworfen werden.
- In Hinblick auf die umfangreichen Protokolle zur Abwicklung der Migration muß das Nachrichtensubsystem möglichst generisch und erweiterbar gehalten werden.

Ein Grundkonzept beim Entwurf des Nachrichtensubsystems ist, daß es in keiner Weise von dem darüberliegenden Agentensystem abhängig ist. Im Prinzip soll es so auch für völlig andere Anwendungen nutzbar sein.

Systemstruktur

Die grundsätzliche Struktur des eigentlichen Nachrichtensubsystems ist in Abbildung 4.8 wiedergegeben. Das hier verwendete Kürzel ITC steht für *inter-thread-communication* und ist historisch bedingt.

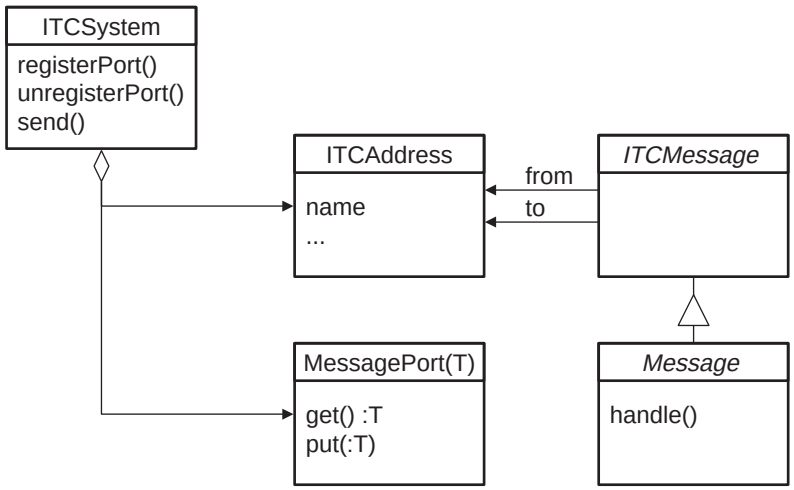


Abbildung 4.8: Klassendiagramm des Nachrichtensubsystems

Die Klasse `ITCMessage` ist dabei die abstrakte Superklasse aller jemals im System versendeten Nachrichten. Sie enthält noch keine Nutzinformation; diese wird erst durch Vererbung in den Subklassen hinzugefügt. Jede Nachricht hat einen Empfänger und optional einen Absender. Diese werden durch Objekte von Typ `ITCAddress` angegeben. Dazu hat

`ITCMessage` die zwei Attribute `to` und `from`. Die Adresse vom Typ `ITCAddress` besteht lediglich aus einem durch eine Zeichenkette repräsentierten Namen, der allerdings einen hierarchischen Aufbau in der Art der WWW-Adressen (URL) haben kann.

Um asynchrone Kommunikation führen zu können, ist die Klasse `MessagePort` entwickelt worden. Jede ihrer Instanzen stellt im Grunde eine Warteschlange (*FIFO-Queue*) dar, an die mit der Methode `put` ein Eintrag angefügt werden und von der mit der Methode `get` der älteste entfernt werden kann. Ist kein Eintrag vorhanden, so blockiert der Aufruf von `get` den Aufrufer solange, bis eine Nachricht vorhanden ist. Diese wird dann zurückgegeben. Der `MessagePort` bedient sich dabei der vom TYCOON-2-System bereitgestellten Synchronisierungsmittel des `Mutex`, der `Condition` sowie der Behälterklasse `Queue` und ist dadurch auch durch nebenläufige Prozesse sicher benutzbar. Weiterhin ist diese Klasse generisch entworfen, hat also einen Typparameter, der die Klasse der Einträge in der Warteschlange und der Parameter von `get` und `put` festlegt. Warteschlangen sind so generell einsetzbar und dabei trotzdem typsicher.

Eine bekannte Einschränkung des TL-2-Typsystems kommt hier allerdings zum Tragen. Aufgrund der kontravarianten Stellung des Parameters der `put`-Methode ist bei der angegebenen Signatur keine Subtypisierung von Warteschlangen der folgenden Art möglich: Wenn für zwei Typen (Klassen) `A<:B` gilt, so gilt `MessagePort(A)<:MessagePort(B)` leider nicht mehr. Diese Eigenschaft wäre aber insbesondere für die Verwendung in der Klasse `ITCSystem` wünschenswert. Die – zwar unschöne, aber gängige – Lösung ist, den Typ des Parameters von `put` fix auf z.B. `Object` zu setzen. Die Behälterklassen der Standardbibliothek bedienen sich häufig dieser Lösung.

Über Objekte der Klasse `MessagePort` wird in asynchroner Weise die gesamte Kommunikation der `threads` untereinander abgewickelt, die in Abbildung 4.7 durch diagonale Linien notiert ist. Träger der Nachrichten sind dabei ausnahmslos Objekte der Subklassen von `ITCMessage`.

Der Kern des Nachrichtensubsystems wird durch die Klasse `ITCSystem` realisiert. Sie hat genau eine Instanz, gehorcht also dem *singleton-pattern*. Um den Forderungen nach Sicherheit, Bindungsarmut und Migrationstoleranz nachzukommen, dürfen die Verwender des Nachrichtensubsystems so wenig Bindungen wie möglich eingehen. Gerade Bindungen (durch direkte Objektreferenzen) an die Empfängerobjekte, also z.B. deren `MessagePort` oder gar den Agenten selbst, müssen unter allen Umständen vermieden werden. Die Adressierung von Agenten etc. geschieht daher stets über symbolische Namen, nämlich Objekte der Klasse `ITCAddress`. Um nun eine Adressauflösung vornehmen zu können, muß lediglich dem Nachrichtensystem der Empfänger bekannt sein. Dazu geben alle an der Kommunikation interessierte Agenten, *threads* usw. dem Nachrichtensubsystem durch die Methode `registerPort` einen ihnen gehörenden `MessagePort` zusammen mit der zugehörigen `ITCAddress` bekannt. Nachrichten an das nun symbolisch adressierbare Objekt werden nun einfach der Methode `send` des Nachrichtensystems übergeben. Da jede Nachricht Subtyp von `ITCMessage` ist, lassen sich der Empfänger (auch vom Typ `ITCAddress`) daraus bestimmen, in der Liste der registrierten Nutzer der zugehörige `MessagePort` finden und die Nachricht diesem zustellen. Technisch gesehen aggregiert das `ITCSystem`-Objekt dazu Paare von Objekten der Typen `MessagePort(ITCMessage)` und `ITCAddress`. Weiterhin muß das System der Tatsache Rechnung tragen, daß es von nebenläufigen *threads* verwendet wird; es ist also die Synchronisierung der Zugriffe auf die internen Daten notwendig.

Nachrichtenverarbeitung

Obwohl die im folgenden geschilderten Muster eigentlich dem Agentensystem zuzuordnen sind, werden sie erst hier besprochen, da ihnen das Konzept der Nachrichten zugrundeliegt. Wie u.a. im Klassendiagramm bereits angedeutet, werden Objekte der Subklassen von `ITCMessage` zwischen den Agenten und Rollen ausgetauscht. Wesentlich für einen guten, typsicheren Entwurf ist es nun, die Fülle von unterschiedlichen Nachrichtenklassen und Objekten in einheitlicher Weise ohne Rückgriff auf überholte Muster wie endlos lange Fallunterscheidungen (*case-switch*) unterscheiden und behandeln zu können. Das Problem stellt sich sofort nach Empfang einer Nachricht. Das folgende Codefragment verdeutlicht (neben der typischen Benutzung des Systems) die Typproblematik:

```
...
private
  _address :ITCAddress,
  _port :MessagePort(Message),
  ...
methods
    ...
    _address:= ITCAddress.new(name),
    _port   := MessagePort.new,
    ITCSystem.instance.registerPort(_port,_address),
    ...
    until({_stop},{
      let msg = _port.get,
      ...
```

Das einzige, was wir nun statisch wissen können, ist, daß `msg` vom Typ `Message` ist. Den aktuellen Typ der empfangenen Nachricht geeignet zu diskriminieren und die Nachricht zu verarbeiten ist Aufgabe des dazu in dieser Arbeit entworfenen und dem *visitor-pattern* nachempfundenen Musters zur Nachrichtenverarbeitung, dem *message-handler-pattern*.

Die grundsätzliche Klassenstruktur dieses Musters ist in Abbildung 4.9 wiedergegeben. Der zu verarbeitenden Struktur im *visitor-pattern* entspricht hier die Klassenhierarchie der Nachrichten mit der Wurzelklasse `Message`. Statt der Methode `visit` besitzt sie die (abstrakte) Methode `handle`, der ein Objekt vom Typ `MessageHandler` übergeben wird. Diese Klasse definiert das abstrakte Protokoll zur Behandlung einer jeden Subklasse der Nachricht durch jeweils eine aufgeschobene Methode. Die Verwendung geschieht dann wieder analog zum Besucher; das oben begonnene Beispiel ließe sich etwa so fortsetzen:

```
      ...
      let msg = _port.get,
      msg.handle(_handler),
      ...
```

Das Objekt `_handler` muß Instanz einer Subklasse von `MessageHandler` sein, die die benötigte Funktionalität besitzt. Die konkrete Gestalt der im Agentensystem verwendeten Klassenhierarchie ist in Abbildung 4.10 dargestellt. Die rechts unten abgebildeten Klassen

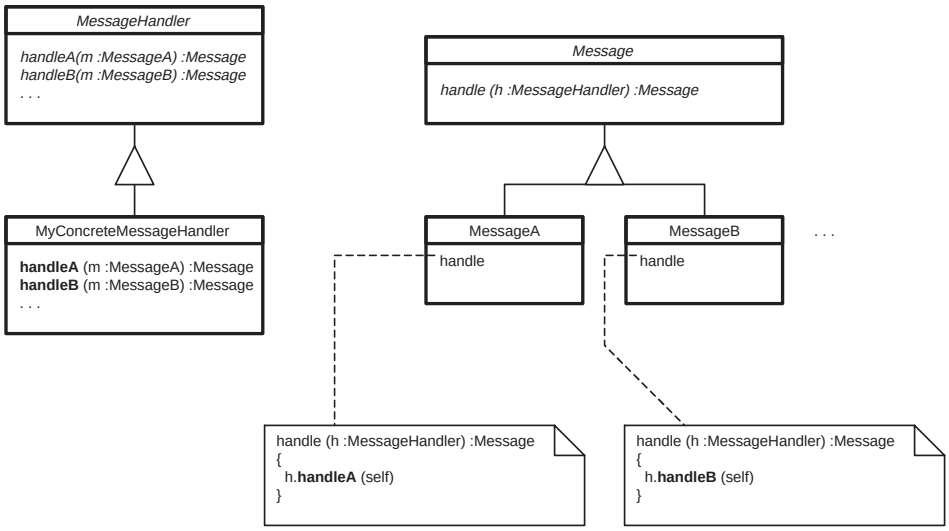


Abbildung 4.9: Klassendiagramm des *message-handler-patterns*

sind die der in Abbildung 4.7 dargestellten Nachrichten. Sie haben neben der Methode `handle` weitere ihrem Zweck entsprechende Attribute für Konversationsspezifikationen, Dialoginstanzen, Anfragen etc. sowie solche für die Steuerung der Kommunikation zwischen den Agenten und den Rollen, die allerdings nicht dargestellt sind. Das Muster der Nachrichtenbearbeitung wird dabei erst in der (abstrakten) Klasse `Message` eingeführt und nicht schon in `ITCMessage`, damit das Nachrichtensubsystem in keine Abhängigkeit von der Schicht der Agenten gerät. Die abstrakte Klasse `MessageHandler` definiert schließlich aufgeschobene Methoden für jede der konkreten Nachrichtenklassen.

Von entscheidender Bedeutung ist nun, daß die Klassen der Rollen durch Vererbung selbst `MessageHandler` sind. Da ein Agent – wie aus Abbildung 4.6 ersichtlich ist – Objekte der Klassen `CustomerRole` oder `PerformerRole` aggregiert, die die der Rolle entsprechende Funktionalität implementieren, also auch das ihnen spezifische Kommunikationsverhalten, kann der Agent, der zunächst alle eingehenden Nachrichten empfängt, bestimmen, welches seiner Rollenobjekte zuständig ist (dies läßt sich leicht aus der genauen Empfängeradresse der Nachricht erkennen) und dieser Rolle durch Verwendung des Nachrichtenbehandlungsmusters die weitere Verarbeitung der Nachricht überlassen.

Das oben fortgesetzte Beispiel, das tatsächlich der Hauptschleife des *threads* des Agenten entnommen (und gekürzt) ist, lautet daher folgendermaßen:

```
...
until({_stop},{
  let msg   = _port.get,
  let role  = lock({_roles[msg.to.nameTail]}),
  let reply = msg.handle(role),
  ITCSystem.instance.send(reply)
}),
...
```

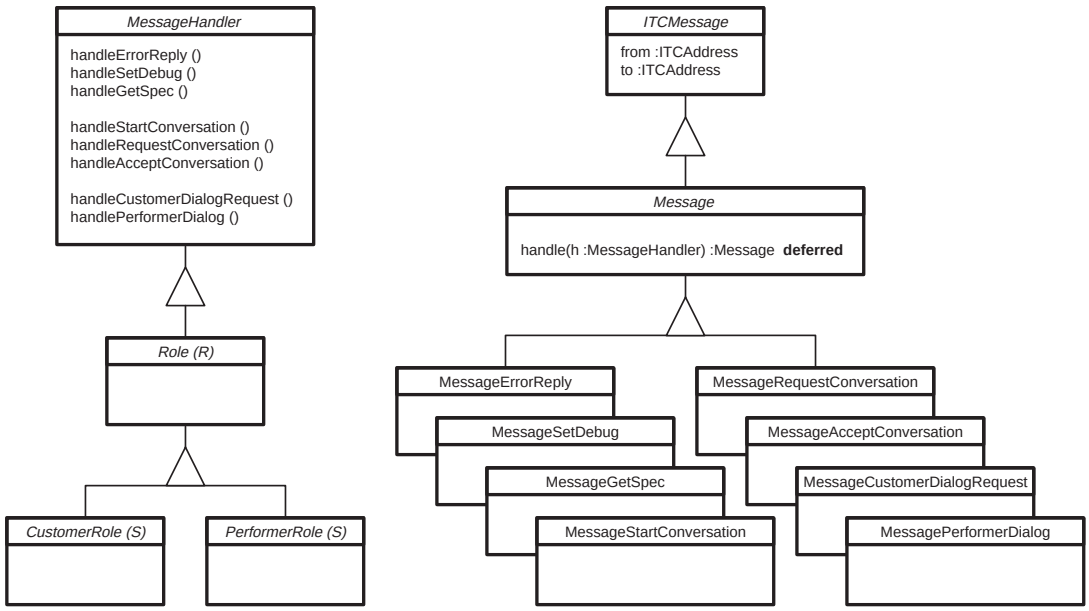


Abbildung 4.10: Klassendiagramm der Nachrichtenverarbeitung

Da alle `handle`-Methoden als Rückgabewert eine Nachricht haben, ist die Erzeugung und Versendung von Antworten besonders einfach: Das Ergebnis der Methode wird einfach dem Nachrichtensubsystem zur Weiterleitung übergeben. Gibt es keine Antwort, so wird der Wert `nil` zurückgegeben, den das System dann ignoriert.

Mediatoren

Ein wichtiger Bestandteil des Nachrichtensubsystems des in [Joh97] beschriebenen Agentensystems ist ein optionaler *Mediatorendienst*. Dieser dem Nachrichtensubsystem über eine spezielle Schnittstelle angegliederte Dienst erhält sämtliche über das Nachrichtensubsystem vermittelte Nachrichten, bevor diese dem eigentlichen Empfänger zugestellt werden. Der Mediator hat dabei das Recht, die Nachrichten zu analysieren, zu verändern oder sogar zu unterdrücken. Auf diese Weise lassen sich beispielsweise Historien (*logging*), Protokollumsetzer oder zentrale Sicherheitsmechanismen realisieren. Die in [Kru97] vorgestellten Dienste für die Verwaltung von Konversationshistorien sind durch Benutzung des dortigen Mediators implementiert.

Auch in der hier vorgenommenen Implementierung existiert eine einfache Schnittstelle für einen solchen Dienst im Nachrichtensubsystem. Mit der Methode `registerMediator` kann ihm eine Funktion höherer Ordnung bekannt gemacht werden, der jede zu versendende Nachricht vor der Zustellung vorgelegt wird. Diese Funktion kann dann die Nachricht entsprechend behandeln. Auch die Änderung des Empfängers ist noch möglich, ebenso die vollständige Unterdrückung der Nachricht (was aber nur in Grenzen ratsam ist, da dann in der Regel die betroffenen Komponenten blockiert werden). Die Ausführung dieser Funktion impliziert noch keine Einschränkung der Nebenläufigkeit, d.h. es ist zu berücksichtigen, daß

sie von parallel arbeitenden *threads* gleichzeitig ausgeführt wird, da sie immer im Kontext des `send` aufrufenden *threads* läuft.

Als prototypische Anwendung eines Mediators wurde ein generischer Mediator entwickelt, der die Anmeldung mehrerer Mediator-Unterdienste gestattet. Diese werden dann von einem eigenen *thread* in strikt sequentieller Folge ausgeführt. Die anmeldbaren Unterdienste sind dabei gleichfalls in der Form höherer Funktionen zu spezifizieren, wobei ihnen eine Priorität gegeben werden muß. Sind mehrere Unterdienste angemeldet, so wird ihnen in der Reihenfolge ihrer Priorität die zu begutachtende Nachricht vorgelegt, wobei die später aufgerufenen Funktionen der Kette bereits die Resultate der früheren übergeben bekommen, also eine unter Umständen mehrfache Filterung der Nachricht stattfindet.

Ein Nachteil dieses so realisierten Mediators ist, daß durch seine Nutzung die ansonsten mögliche Nebenläufigkeit im Nachrichtensubsystem fast völlig aufgehoben wird. Bei sehr hohem Verkehrsaufkommen kann der Mediator also als Flaschenhals wirken. Sollte dies ein ernsthaftes Problem darstellen, so muß, falls es die Art der Anwendung gestattet, ein parallel arbeitender Mediator entwickelt werden.

4.5 Evolution

In diesem Abschnitt sollen für einige der bereits in Abschnitt 2.2 vorgestellten weitergehenden Konzepte die theoretischen Ansätze, die zur Realisierung nötigen Techniken und die sie unterstützende Implementierung vorgestellt werden. Betrachtet werden hier insbesondere diejenigen Konzepte, die im Hinblick auf verbesserte Unterstützung der Systemevolution Gewinn zu versprechen scheinen.

Das grundlegende Paradigma der betrachteten Evolution ist die vorsichtige, *inkrementelle Erweiterung* laufender Dienste und Spezifikationen. Es geht hier also weniger um Analyse, Entwurf und Implementierung eines vollständigen Systems von Grund auf, als vielmehr um die behutsame, den aktuellen Erfordernissen der im stetigen Wandel begriffenen Geschäftsprozesse folgende Erweiterung und Änderung bereits bestehender Systeme.

Die im folgenden näher betrachteten Konzepte sind die von Subkonversationen, sekundären Konversationen sowie die Subtypisierung von Konversationsspezifikationen.

4.5.1 Subkonversationen

Die Subkonversationen sollen dazu dienen, bereits definierte Konversationsspezifikationen wiederverwenden zu können. Technisch gesehen ist die Wiederverwendung von Teilen der Spezifikation, z.B. Inhaltsspezifikationen innerhalb einer Konversationsspezifikation, bereits leicht möglich durch die Nutzung der Referenzsemantik von TL-2: Ein einmal mit den in Abschnitt 4.4.1 vorgestellten Mitteln konstruiertes Objekt der Klasse `ContentSpec` kann in beliebigen Dialogspezifikationen „eingehängt“, also von ihnen referenziert und benutzt werden. Dasselbe ist mit Dialogspezifikationen in verschiedenen Konversationsspezifikationen möglich. Der Nachteil dieses Vorgehens ist natürlich, daß die Wiederverwendung nicht explizit modelliert wird, sondern nur ein Produkt der Spracheigenschaften ist.

Der in [Joh97] vorgenommene Entwurf sieht Dialogspezifikationen und Subkonversationsspezifikationen als alternative Ausprägungen von *abstrakten Dialogspezifikationen* an, die von der Konversationsspezifikation aggregiert werden. Am Rande wird dort erwähnt, daß dabei sogar eine Modifizierung der Subkonversation möglich ist. Aufgrund der bislang unklaren operationalen Semantik – die Implementierung dieses Konzepts wurde auch nicht vorgenommen – soll hier ein eigener Ansatz vorgestellt werden.

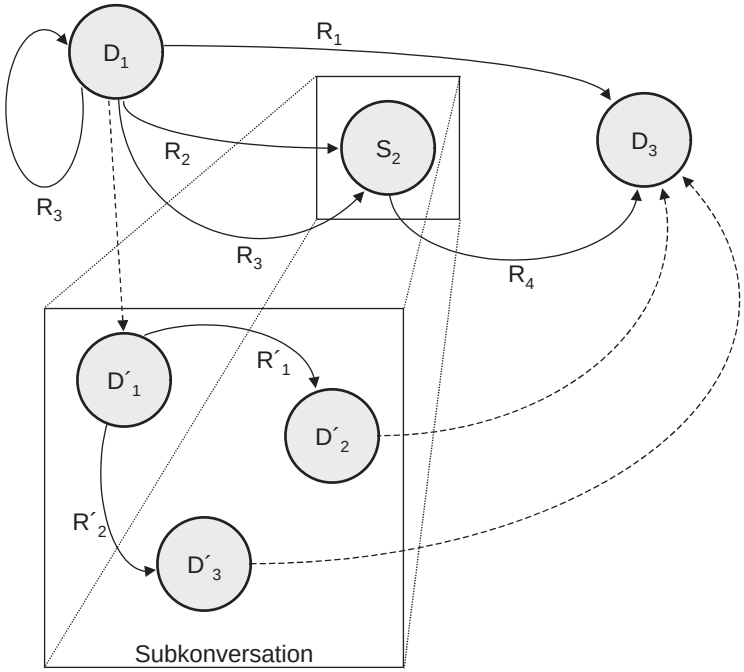


Abbildung 4.11: Verfeinerung einer Spezifikation durch eine Subkonversation

Schematisch ergibt sich das in Abbildung 4.11 dargestellte Bild. Die anstelle eines Dialoges eingesetzte Subkonversation S_2 ist in Wirklichkeit eine vollständige, hier sehr einfache, nur aus drei Dialogen bestehende Konversationsspezifikation. Zum Teil kritisch – und deswegen genau zu definieren – sind dabei unter anderem die folgende Punkte:

- Der Einstieg in die Subkonversation ist unkritisch und erfolgt bei dem initialen Dialog gemäß ihrer Konversationsspezifikation. Dieser (D_1') folgte sofort auf die Anfrage R_2 oder ggf. auch auf R_3.
- Viel problematischer ist der Ausstieg. Die Subkonversation ist definitionsgemäß mit Erreichen eines finalen Dialoges beendet (im Beispiel wären dies D_2' und D_3'). Fraglich ist jetzt, wie auf der Ebene darüber fortgefahren wird. Offensichtlich benötigt die Spezifikation der Subkonversation genau wie ein Dialog Anfragen, die auf Folgedialoge verweisen.
- Gemäß dem bisherigen Entwurf sind für die Entscheidung der Zustandsübergänge Funktionen in Form von Regeln anzugeben. Deren Verwendung ist in diesem Fall ebenfalls fraglich und muß genau definiert werden. Zu Beginn entscheidet eine Dienstleisterregel im Rahmen der Konversationsspezifikation darüber, ob der Folgedialog

eine Subkonversation ist. Auf der Ebene der Subkonversation ist der Ablauf unproblematisch und kann dem normalen Ablauf folgen. Kritisch ist wieder der Übergang zur höheren Ebene. Am sinnvollsten erscheint die Lösung, auf der Kundenseite auch eine Regel für die Subkonversation zu binden, die nach Beendigung der Subkonversation die Entscheidung über die jetzt zu stellende Anfrage trifft. Auf der Seite des Dienstleisters ist dann – analog zu Dialogen – für jede mögliche Anfrage, die aus der Subkonversation folgt, eine Regel zu binden.

- Ein weiterer, eher technischer Gesichtspunkt ist die Frage, ob und welche Dialoginstanz diesen Regeln übergeben wird. Im normalen Fall ist dies eindeutig durch die Konversationsspezifikation festgelegt. Es ist sinnvoll, daß sowohl der Kunde als auch der Dienstleister auf einfache Weise über die Resultate der Subkonversation verfügen können. Deshalb wäre eine mögliche Lösung, den finalen Dialog der Subkonversation zunächst der Kundenregel, die auf der höheren Ebene an die Subkonversation gebunden ist, vorzulegen, um ihn dann, ggf. verändert, an die entsprechende Dienstleisterregel, die zu der gewählten Anfrage gehört, weiterzuleiten. Da aber die Spezifikation des Dialoges auf der höheren Ebene formal nicht bekannt ist, ist diese Lösung nicht empfehlenswert, auch wenn die Struktur des Dialoges durch die reflektiven Möglichkeiten untersuchbar wäre. Stattdessen könnten die Regeln entweder gar keinen Dialog (wie die Regel für die initiale Anfrage auf der Dienstleisterseite) oder einen speziellen, systemdefinierten Dialog erhalten, der Aufschluß über die Subkonversation gibt. Die eigentliche Handhabung der Ergebnisse muß also in beiden Fällen auf der Ebene der Anwendungslogik geschehen.

- In [Joh97] nur angedeutet ist die Möglichkeit, die Subkonversation mit einem anderen Dienstleister als dem aktuellen zu führen. Dies ist prinzipiell mit der oben entwickelten Lösung vereinbar.

- Jede Rolle wird wie oben beschrieben an genau eine Konversationsspezifikation gebunden. Zu lösen ist also das Problem der Bindung der evtl. enthaltenen Subkonversationsspezifikationen. Sie zum Bestandteil der höheren Ebene zu machen scheidet aus, da dann die Wiederverwendbarkeit eingeschränkt ist. Die beste Lösung ist, sie an eine eigene Rolle zu binden. Diese verfügt wie jede Rolle über eigene Regeln, die dann ebenfalls nicht Bestandteil der höheren Ebene sein müssen. Auf der höheren Ebene wird sie dann nur noch symbolisch referenziert. Der Gewinn dadurch ist hoch: So sind prinzipiell beliebige Dienste als Subkonversationen nutzbar, auch solche, die zunächst überhaupt nicht als Teil eines größeren Prozesses entworfen wurden. Dies ist ein zentrales Merkmal für evolutionstaugliche Modelle.

- Ferner ist noch das technische Problem der Abwicklung der Subkonversation zu lösen. Sinnvoll wäre, daß die Ablaufsteuerung der Rollen selbst erkennt, daß ein Dienstleister als Folgedialog eine Subkonversation gewählt hat und die Kundenseite der Ablaufsteuerung veranlaßt, eine neue Konversation zu starten. Die Steuerung im Kunden muß dann die höhere Konversation solange aufzuschieben, bis die Subkonversation beendet ist und dann wie oben beschrieben fortfahren. Zusätzlich ist zu bestimmen, wo die Entscheidung darüber fällt, mit welchem Dienstleister die Subkonversation geführt wird.

Die uniforme Behandlung von Konversationen und Subkonversationen ermöglicht so die inkrementelle und auf Wiederverwendung basierende Systementwicklung von Diensten. Damit wird sie der Forderung nach Unterstützung der Evolution gerecht.

Das eben entwickelte Konzept für Subkonversationen ist zur Zeit noch nicht Bestandteil der Implementierung des Agentensystems. Sie nachzuholen bedeutet aber voraussichtlich keinen größeren Aufwand.

4.5.2 Sekundäre Konversationen

Die *Delegation* und *Koordination* genannten Mechanismen sind mit dem der Subkonversation eng verwandt. Wie in Abschnitt 2.2.4 beschrieben, dienen sie dazu, komplexe Beziehungen zwischen Kunden und Dienstleistern durch *sekundäre Konversationen* zu modellieren und aufzubauen. Der Unterschied von sekundären Konversationen zu Subkonversationen ist, daß sie nicht Bestandteil der Konversationsspezifikation sind und deshalb transparent für den Kommunikationspartner ablaufen. Auch ist hier höhere Flexibilität in Hinblick auf Nebenläufigkeit, Wahl der Rolle und Anzahl der gleichzeitigen Konversationen vorhanden.

Die Ablaufsteuerung von Subkonversationen ist Sache des Agentensystems und muß in den Rollen implementiert werden. Die Initiierung und die Ablaufsteuerung von sekundären Konversationen dagegen ist allein Sache der Anwendungslogik. Allerdings sind vom Agentensystem unterstützende Funktionen zu fordern.

Grundsätzlich gilt dasselbe wie für Subkonversationen: Zur Abwicklung einer sekundären Konversation wird eine im Prinzip unabhängige, neue Konversation einer der Anforderung entsprechenden Rolle gestartet. Die Synchronisierung und der Austausch der Informationen zwischen den einzelnen Konversationen muß in den entsprechenden Regeln implementiert werden. Dazu ist im Einzelfall zu untersuchen, wie diese zentrale Kontrolle (z.B. die eines *Brokers* oder *Koordinators*) am besten zu entwerfen ist. Als Möglichkeiten seien genannt:

1. Die Erstellung eines gemeinsamen zentralen *threads*, der von allen betroffenen Regeln durch synchrone oder asynchrone Kommunikation angesprochen wird, und der die eigentliche Funktionalität vereint. Durch Nutzung der in Abschnitt 4.4.3 vorgestellten Mittel (`MessagePort` etc.) ist so ein System sehr schnell und einfach zu realisieren.

2. Der verteilte, nebenläufige Zugriff auf den betroffenen Regeln gemeinsame Objekte. Dies bedingt häufig zusätzlichen Aufwand zur Erhaltung der Konsistenz der Objektzustände. Durch Nutzung der objektorientierten Eigenschaften ist diese Konsistenzkontrolle (realisiert durch Semaphoren etc.) aber leicht in den betroffenen Klassen zu kapseln.

Zur Unterstützung besitzt das Agentensystem in der Klasse `TBC` drei spezielle Methoden: die zuvor bereits in einem Beispiel gezeigte Methode `startConversation`, die ohne auf den Erfolg zu warten eine neue Konversation zwischen zwei durch ihre Namen anzugebende Dienstleister und Kunden initiiert; die Methode `startConversationSuccess`, die abhängig davon, ob die gewünschte Konversation erfolgreich gestartet werden konnte, den entsprechenden Wahrheitswert zurückgibt; sowie die Methode `startConversationEnd`,

die `nil` zurückliefert, wenn die Konversation nicht zustandekommen konnte, ansonsten aber wartet, bis sie beendet ist und dann den Namen des letzten, finalen Dialoges liefert. Die letzte Methode besitzt eine weitere Variante namens `startConversationEndA`, die ebenfalls im Falle des Mißlingens des Aufbaus der Konversation `nil` liefert, im Falle des Gelingens aber nicht das Ende abwartet, sondern eine Funktion zurückgibt, die erst beim Aufruf auf die Beendigung wartet und den Namen des letzten Dialoges liefert. Dadurch können zunächst nach Aufbau einer Konversation weitere Schritte unternommen werden (etwa eine andere Konversation weitergeführt werden), und erst später muß auf die Beendigung gewartet werden. Dies kann etwa von einer Regel ausgenutzt werden, die eine sekundäre Konversation initiiert, wobei die Beendigung der sekundären Konversation die Vorbedingung für die Abarbeitung einer anderen Regel ist. Das Funktionsobjekt kann dabei leicht über den Mechanismus des gemeinsamen Kontextobjektes erreicht werden. Zur Modellierung dieser Abhängigkeiten eignen sich Petrinetze gut.

4.5.3 Subtypisierung von Konversationsspezifikationen

Ein Szenario, in dem die evolutionsunterstützenden Eigenschaften von Konversationsspezifikationen gewinnbringend genutzt werden können, ist folgendes: Ein Dienstleister, der sich (organisatorisch oder räumlich bedingt) getrennt von etwaigen Kunden auf neue Anforderungen einstellt, ändert und erweitert dazu seine Geschäftsprozesse und die sie beschreibende Konversationsspezifikation. Damit ein diese Entwicklung nicht oder nicht so rasch nachvollziehender Kunde weiterhin trotz seiner unveränderten Konversationsspezifikation mit dem Dienstleister Konversationen führen kann, sind gewisse Kompatibilitätsanforderungen an die Konversationsspezifikationen beider Seiten zu stellen.

Analog zu Typen in Programmiersprachen muß die neuere, inzwischen speziellere Konversationsspezifikation des Dienstleisters *Subtyp* der alten, allgemeineren des Kunden sein. Das bedeutet, daß die neue die Möglichkeiten der alten noch voll enthält. Dies führt zum Konzept der *Subtypisierung von Konversationsspezifikationen*.

Ziel ist, daß eine Konversation fehlerfrei ablaufen kann, wenn die Konversationsspezifikation des Dienstleisters Subtyp der des Kunden ist. Dazu ist die Subtypbeziehung folgendermaßen definiert:

1. Auf der Ebene der *Ablaufbeschreibung*, also der Anfragen und der Folgedialoge, ist zu gewährleisten, daß

   - alle Dialoge der alten Spezifikation auch noch in der neuen vorhanden sind,
   - alle Anfragen in jedem Dialog der alten Spezifikation auch im neuen vorhanden sind und
   - jede Anfrage noch auf dieselben Folgedialoge verweist. Außerdem muß der initiale Dialog derselbe bleiben.

   Erlaubt ist, daß Dialoge, Anfragen und Folgedialoge neu hinzukommen. Die Zuordnung der zu vergleichenden Dialoge geschieht dabei über ihre Namen. In diesem Sinne beruht die Subtypisierung auf dieser Ebene also auf Namensäquivalenz und nicht

auf struktureller Äquivalenz. Letzteres würde wesentlich aufwendigere Algorithmen erfordern, die letztlich die Isomorphie von Graphen prüfen können müßten.

2. Auf der Ebene der Inhaltsspezifikationen der Dialogspezifikationen ist zu gewährleisten, daß die Inhaltstypen ähnlich wie in Programmiersprachen Subtypen sind, d.h. jede neue Inhaltsspezifikation muß Subtyp der alten sein. Dabei lassen sich folgende Subtypbeziehungen zwischen den Inhaltstypen definieren:

   - `SingleChoice(T) <: T`
   - `MultipleChoice(T) <: Sequence(T)`
   - Die Subtypisierung von Recordtypen geschieht analog zu Recordsubsignaturen in TL [Mat93]; dies bedeutet informell beschrieben, daß alle Komponenten erhalten bleiben müssen und jede Komponente der neuen Definition jeweils Subtyp der entsprechenden alten sein muß. Dabei werden die Subtypisierungsregeln rekursiv angewendet; ferner dürfen neue Komponenten dazukommen. Die Reihenfolge der Komponenten spielt keine Rolle, denn die Zuordnung geschieht über ihre Namen.

   Die Subtypisierung auf dieser Ebene beruht also auf struktureller Äquivalenz. Denkbar ist auch noch die Einführung von Subtypbeziehungen zwischen den atomaren Basistypen, z.B. wären die Beziehungen `String <: Real <: Int <: Bool` oder `Real <: Currency` möglich. Da äquivalente Beziehungen aber auch nicht in TL-2 bestehen, wurde von der Realisierung Abstand genommen.

Zugriffe auf nicht existente Komponenten von Inhalten können zur Laufzeit nicht auftreten, da die Dialoge immer vom Dienstleister erzeugt werden und vom Kunden lediglich die ihm bekannte Untermenge verändert wird. Da der Dienstleister diese von ihm erzeugten Dialoge auch wieder zurückerhält und der Kunde niemals selbst Dialoge instantiiert, können auf diesem Wege auch keine Fehler verursacht werden.

Abbildung 4.12 stellt die Subtypisierung von Konversationsspezifikationen auf der Ebene des Ablaufes graphisch dar. Die A' genannte Konversationsspezifikation ist dabei Subtyp von A, nicht aber A'' von A. Dies liegt daran, daß A' alle Dialoge, Anfragen und Folgedialoge von A besitzt, bei A'' aber die Kante von D_2 nach D_1 fehlt.

Die Prüfungen für die beschriebenen Subtypisierungsbeziehungen sind im Agentensystem implementiert. Die dazu nötigen Prüfungen sind in den `matches`-Methoden der die Spezifikationen implementierenden Klassen definiert, also in `ConversationSpec`, `DialogSpec`, `RequestSpec` sowie in den Subklassen von `ContentSpec`. Durch rekursive Delegation an die Methoden der enthaltenen Unterstrukturen können so durch einen Aufruf der Art `myConvSpec.matches(yourConvSpec)` vollständige Konversationsspezifikationen miteinander verglichen werden. Diesen Mechanismus verwendet die Klasse der Dienstleisterrolle `PerformerRole` zur Prüfung, ob einer Anfrage nach Eröffnung einer Konversation seitens eines Kunden entsprochen werden kann. Dies ist möglich, da diese Anfrage die Konversationsspezifikation des Kunden als Parameter enthält.

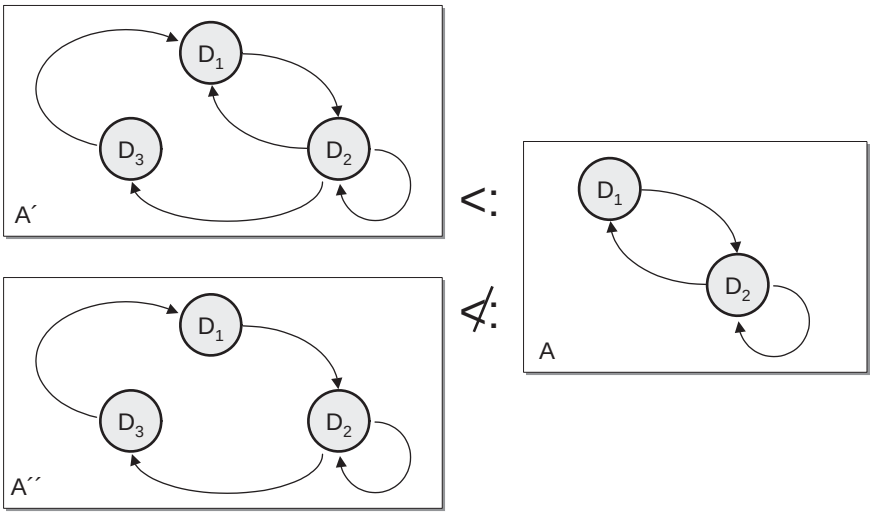


Abbildung 4.12: Subtypisierung von Konversationsspezifikationen

4.6 Bewertung

Ein Vergleich des Umfanges der alten in [Joh97] beschriebenen Implementierung mit der neuen, hier vorgenommenen liefert folgende Ergebnisse:

1. Die die Interaktion, also die *Business Conversations* selbst betreffenden Teile, sind hier im vollen Umfange realisiert worden. Die einzige Ausnahme kann in der Verwirklichung des Konzepts der Subkonversationen gesehen werden; dieses war allerdings auch nicht Bestandteil der alten Implementierung. Es ließe sich aber leicht nachrüsten; die erforderlichen Mechanismen sind zum großen Teil bereits schon vorhanden. Im Umfang über die alte Implementierung hinaus geht dagegen die Realisierung der Unterstützung für sekundären Konversationen und Subtypisierung von Konversationsspezifikationen.

2. Das Konzept der Agenten ist jetzt etwas allgemeiner als zuvor. Jeder Agent kann nun beliebig viele Rollenobjekte verschiedener Art aggregieren und in deren Rollen agieren. Die Realisierung wurde weitgehend anders gelöst. Insbesondere kommen nun wesentlich mehr *threads* zum Einsatz, nämlich für den Agenten selbst einer und für jede laufende Konversation ein weiterer. Dadurch ließ sich zum einen die Nebenläufigkeit stark erhöhen und zum anderen die Realisierung vereinfachen.

3. Nicht realisiert werden konnte die Migration von Agenten aufgrund der fehlenden Systemvoraussetzungen. Trotzdem wurde das gesamte System dahingehend entworfen. Auch das Konzept der hierarchischer Orte fehlt noch, da es eng mit der Migration verbunden ist und isoliert keinen Vorteil bietet.

Interessant ist es, die sich ergebenden Unterschiede im Programmierstil zu betrachten. Die Programmiersprache TL mit ihrem funktional-imperativen Charakter führt zu einem Stil,

der starken Gebrauch von langen, sehr tief geschachtelten Modulen, Konstrukten und Ausdrücken macht. Dies trifft in besonderem Maße auf die Nachrichten zu, die durch variante Tupel implementiert wurden. Die Realisierung einer Rolle im alten System besteht zum Beispiel hauptsächlich aus einer etwa 150 Zeilen langen Fallunterscheidung und Behandlung der zu verarbeitenden Nachrichten. Mit TL-2 hingegen läßt sich ein Stil etablieren, der sehr fein granulare Abstraktionen durch relativ viele Klassen und Methoden nutzt. Die gesamte neue Implementierung umfaßt daher gut 100 Klassen, die fast alle sehr klein sind. Gut erkennen läßt sich der Unterschied auch hier wieder am neuen Muster zur Nachrichtenverarbeitung.

Als noch wesentlich wichtiger und nützlicher haben sich die in TL-2 vorhandenen Sprachmittel der Vererbung und des parametrischen Polymorphismus erwiesen. Drastisch fällt das Ergebnis eines Vergleiches der Implementierung der Rollen aus: Während die Kunden- und Dienstleisterrollen in der alten Implementierung äußerst viele strukturelle Redundanzen aufweisen und lange Passagen sogar identisch sind, ließ sich dies in der neuen Implementierung durch die Zusammenfassung von gemeinsamer Funktionalität in eine Superklasse vollständig vermeiden.

Durch den – allerdings auch schon in TL vorhandenen und genutzten – parametrischen Polymorphismus ließ sich häufig sehr viel Funktionalität früh generalisieren und in Superklassen definieren, ohne auf typunsichere Konstrukte zurückgreifen zu müssen. Dies ist im Vergleich zu Sprachen, die dies nicht bieten (etwa JAVA), ein nicht zu unterschätzender Gewinn.

Von der Möglichkeit zur Mehrfachvererbung in `TL-2` wurde sehr häufig Gebrauch gemacht. Dabei zeigt sich, daß in vielen Fällen ein *mixin style* angebracht ist. Er ist immer dann günstig anzuwenden, wenn relativ allgemeine, nicht sehr umfangreiche Funktionalität eher schematischer Art an mehreren Stellen, also in verschiedenen Klassen, benötigt wird. Als Beispiele in der vorgenommenen Implementierung sind etwa die Klasse `Trace`, die primitive Ablaufverfolgung und globale Schalter dafür unterstützt, die Klasse `NamedMixin`, die Klassen um einen Namen bereichert sowie die Klasse `PrettyPrintMixin` zu nennen, die Unterstützung bei der Formatierung der Ausgabe von verschachtelten Strukturen bietet.

Andererseits zeigt sich an manchen Stellen, daß die Komposition von Attributen in Klassen der direkten Ererbung der entsprechenden Klassen vorzuziehen ist. Dies ist auch eine generelle These, die in [GHJV95] vertreten wird und die sich hier im wesentlichen bestätigt findet. Zwar lassen sich im zweiten Fall die Methoden einfach redefinieren (überschreiben), aber im ersten Fall führt die Ererbung schnell zu unübersichtlichen Schnittstellen, die oft viel mehr als die wirklich benötigte Funktionalität bereitstellen. Es ist also häufig abzuwägen, ob die Betrachtung von zwei Klassen unter dem Blickwinkel der „ist-eine“-Beziehung oder der „hat-eine“-Beziehung adäquater ist. Die zur Zeit durch Ererbung von `Dictionary` realisierten Klassen (z.B. `RecordContentSpec` oder auch `ConversationSpec`) wären, im Nachhinein betrachtet, besser durch Attribute vom Typ `Dictionary` realisiert worden. Als Faustregel ergibt sich mithin, daß die Klassen der Standardbibliothek normalerweise besser als Attribute verwendet, und nicht direkt ererbt werden sollten.

Als unpraktisch bei der Verwendung gerade der o.g. Klassen hat sich herausgestellt, daß die Schlüssel von `Dictionaries` nicht Bestandteil der aggregierten Elemente sein können. Dies wäre aber gerade bei diesen Klassen sehr nützlich, da eines ihrer Attribute, nämlich der

Name (vom Typ `String`), zugleich ihr jeweiliger Schlüssel in den sie enthaltenden Objekten ist. So sind jetzt immer zusätzliche Anweisungen nötig. Eine entsprechende Klasse, die `Dictionary` in diesen Fällen ersetzt oder erweitert, wäre also nützlich.

Bewährt hat sich die Nutzung und Weiterentwicklung von Entwurfsmustern. Dies zeigt sich besonders gut bei der Anwendung des Musters zur Nachrichtenverarbeitung.

Großen Anteil an der einfachen und schnellen Implementierung hat die sehr gut ausgebaute Klassenbibliothek des TYCOON-2-Systems. Zu nennen sind besonders die an Funktionalität schier unerschöpflichen Behälterklassen. Hier ist das einzige Problem tatsächlich, die benötigte Funktion zu finden, da ein benutzungsfreundlicher *classbrowser* noch nicht existiert. Die Realisierung nebenläufiger Prozesse und ihre Synchronisierung stellten sich aufgrund der wenigen, aber sehr gut entworfenen und deshalb absolut ausreichenden Mittel (`Thread`, `Mutex`, `Condition` etc.) als besonders einfach heraus. Auch die Abwicklung der Kommunikation wurde durch die (selbst vorgenommene) Entwicklung der Klasse `MessagePort` stark vereinfacht, da bei deren Nutzung von den nötigen Synchronisierungsmaßnahmen völlig abstrahiert werden kann.

Kapitel 5

Realisierung von Internet-Informationssystemen

Die Bedeutung des Internet steigt aufgrund seiner zunehmenden Kommerzialisierung auch für die Hersteller und Betreiber von Informationssystemen immer weiter an. Ihr Wunsch ist es, ihre bisher konventionellen Geschäftsprozessen gehorchenden Dienstleistungen und ihr Informationsangebot auch über das Internet verfügbar zu machen. Damit entstehen *Internet-Informationssysteme.*

Von einem technischen Standpunkt aus gesehen sind die für die Übertragung und Darstellung im Internet verwendeten Protokolle und Formate HTTP und HTML sehr interessant, da sich mit ihnen das erste Mal eine portable, plattformunabhängige und faktisch ubiquitäre Infrastruktur für die Kommunikation und die Visualisierung durchgesetzt hat, die für sehr viele Arten der Benutzerinteraktion mit Systemen genutzt werden kann und in zunehmendem Maße auch tatsächlich genutzt wird. Das Schlagwort *Intranet* steht häufig dafür.

Das eigentliche Problem ist dabei stets die Koppelung der eigentlichen, die Geschäftsprozesse beherbergenden Anwendung mit den zur Bereitstellung der Dienste im Internet nötigen Werkzeugen wie dem WWW-Server. Hierfür gibt es inzwischen eine Vielzahl von sehr unterschiedlichen Ansätzen und kommerziellen Lösungen.

Der folgende Abschnitt 5.1 stellt die Entwicklung der bisherigen Ansätze zur Realisierung von Internet-Informationssystemen kurz dar. Im nächsten Abschnitt 5.2 soll sodann ein neuerer, gleichfalls auf dem TYCOON-2-System basierender Ansatz vorgestellt werden. Um zu untersuchen, inwieweit das in dieser Arbeit entwickelte mit den *Tycoon Business Conversations* arbeitende Agentensystem zur Realisierung von Internet-Informationssystemen – oder natürlich auch Intranet-Informationssystemen – taugt, wurde ein solches realisiert. Dies soll im darauf folgenden Abschnitt 5.3 beschrieben werden. Abschließend wird in Abschnitt 5.4 eine Bewertung der vorgestellten Lösungen vorgenommen.

Die im gängigen informationstechnischen *slang* meistens uneinheitlich als *Web-* oder *WWW-Server* und *-Browser* o.ä. bezeichneten Programme wollen wir im folgenden stets HTTP-Server und HTTP-Client nennen.

5.1 Die bisherigen Ansätze

Zu Anfang der Entwicklung besaßen HTTP-Server keinerlei dynamische Möglichkeiten, stellten also eine feste Menge von unveränderlichen, statisch miteinander in Beziehung gesetzten HTML-Seiten dar, die als einzelne Dateien in einer dem Server bekanntgemachten Verzeichnisstruktur angeordnet waren.

Dynamik wurde erst durch Entwicklung der HTML-Formulare und der CGI genannten Schnittstelle (*common gateway interface*) des Servers möglich. Formulare ermöglichen dabei die Interaktion zwischen Benutzer und HTTP-Server. Um die ausgefüllten und zurückgeschickten Formulare auszuwerten, ruft der HTTP-Server externe Programme auf. Dies sind normalerweise in Skriptsprachen wie PERL oder in C geschriebene Programme. Ihre Aufgabe ist es, basierend auf den Informationen des Formulars als Antwort wiederum eine HTML-Seite oder ein HTML-Formular zu generieren. Die dazu verwendete Schnittstelle CGI soll hier nicht weiter betrachtet werden. Probleme ergeben sich u.a. daraus, daß zur Auswertung jeder Anfrage eines Formulars erneut das zugeordnete Programm gestartet werden muß, was unter dem zumeist verwendeten Betriebssystem UNIX relativ teuer ist.

Zur Realisierung von Informationssystemen müssen diese Programme dann auf geeignete Weise auf die die Dienste eigentlich leistenden Anwendungen wie z.B. Datenbanken zugreifen. Die Hauptprobleme dabei sind zum einem wieder die Konvertierung der Ergebnisse in HTML und zum anderen die Eigenschaft des Protokolls HTTP, verbindungs- und zustandslos zu sein. Beides zusammen führt häufig zu sehr kryptischen, undurchschaubaren Lösungen. Als weiterer Kritikpunkt ist anzubringen, daß eine Trennung zwischen Präsentation, also graphischer Gestaltung der HTML-Seiten, und den darin dargestellten logischen Inhalten durch das HTML-Format nicht unterstützt wird.

Als nächste Stufe der Entwicklung sind integrierte HTTP-Server und Datenbanken zu nennen. Den ersten liegt die Überlegung zugrunde, daß durch Erweiterung des HTML-Formats die o.g. Skripte auch gleich in die Seiten integriert werden und vor dem Versand durch den HTTP-Server von ihm ausgeführt werden können. Das Ergebnis ist dann wieder reines HTML. Inzwischen haben auch die Hersteller von Datenbanken den Markt entdeckt und statten ihre Produkte häufig zusätzlich mit der Funktionalität eines HTTP-Servers aus, der es gestattet, die Tabellen der Datenbank zu benutzen. Natürlich sind auch Mischformen beider Entwicklungen denkbar. Auf all diese Aspekte soll hier nun nicht weiter eingegangen werden.

5.2 STML

Im folgenden soll ein neuer Ansatz, der von der Firma HIGHER-ORDER zur Realisierung ihrer Internet-Informationssysteme verwendet wird, beschrieben werden [MW97, GW97, HOX98]. Er basiert auf dem TYCOON-2-System.

Ausgangspunkt der Entwicklung ist die orthogonale Erweiterung von SGML (der *standard general markup language*) um Variablen, Funktionsabstraktionen und Funktionsanwendung. Dies ermöglicht die nahtlose Einbindung von Programmiersprachen, um auf der einen Seite weitere Dienste zu bemühen und auf der anderen Seite dynamisch Inhalte zu generieren.

Das Vorgehen ist dabei folgendes [MW97]:

1. Gegeben sei eine Programmiersprache A, in diesem Fall TL-2, und eine strukturierte Präsentationssprache P, hier HTML, letztere durch ihre Definition (*document type definition*, DTD).
2. Durch Erweiterung von P durch die oben genannten SGML-Erweiterungen für die entsprechenden Konstrukte aus A entsteht P_A. Die neue Definition P_A kann zur werkzeugunterstützten Erstellung und Analyse von P_A-Dokumenten genutzt werden.
3. Es muß ein P_A-Sprachprozessor entwickelt werden, der Dokumente auf Typkorrektheit prüft und sie in ausführbaren Code von A übersetzt. Insbesondere gewährleistet er, daß die Ausführung des Codes zur Laufzeit Dokumente erzeugt, die P entsprechen.
4. Ein Laufzeitsystem führt den erzeugten Code aus und sorgt für die typsichere Bindung des Codes an weitere Dienste.

Die so gewonnene neue Dokumentenbeschreibungssprache (DTD) wurde STML (*standard tycoon markup language*) genannt.

Ein wichtiger Gedanke ist, daß jede eingebettete Funktion wieder HTML erzeugen kann. Durch Nutzung von Iterationsabstraktionen im Code lassen sich so z.B. auf sehr einfache Weise Massendatenstrukturen visualisieren. Hier soll ein kleines Beispiel die Nutzung von STML illustrieren. Darin wird innerhalb eines bedingen Ausdrucks, der die Verwendung der SGML-Fallunterscheidung zeigt, mittels der sogenannten `<tycoon>`-Markierung TL-2-Code eingebettet, der eine primitive Liste erzeugt:

```
...
<if true='application.project.employees.size = 0>
  Es gibt keine MitarbeiterInnen!<P>
<else>
  <B>MitarbeiterInnen:</B>
  <OL>
  <tycoon>
    application.project.employees.do(fun(e: Employee) {
      out << "<LI>" << e.name << "</LI>"
    })
  </tycoon>
  </OL>
</if>
...
```

Hier ist `out` der Ausgabestrom der Antwort. In dem eingebetteten TL-2-Code besteht der direkte Zugriff auf die HTTP-Objekte `request` und `reply` der Anfrage und der zurückzusendenden Antwort. Damit besteht z.B. Zugriff auf die Formularfelder des vorherigen Formulars. Ferner ist die Nutzung von spezialisierten Anwendungsobjekten möglich, die über den Bezeichner `application` angesprochen werden können.

In STML sind weiterhin Konstrukte zur Definition von Variablen, Methoden, Funktionen und dem Aufruf von Methoden bzw. Funktionen definiert.

Das TYCOON-2-System verfügt über einen fast vollständig in TL-2 entwickelten HTTP-Server. Dieser ist unter Nutzung von generischen und abstrakten Basisklassen um eigene *HttpResource*-Klassen und -Objekte erweiterbar, die genau spezifizierbaren HTTP-Anfragen zuzuordnen sind. Dadurch sind alle Sprachmittel von TL-2 und alle Systemeigenschaften in spezialisierten Diensten voll nutzbar, um auf HTTP-Anfragen zu reagieren.

Bereits vorhanden sind Klassen, die Anfragen nach HTML-Dokumenten etc. behandeln. Sie greifen wie gewöhnlich auf die Dateien im Dateisystem zu. Für die Anfragen nach STML-Dokumenten existiert ebenfalls eine spezielle Klasse `StmlResource`. Alle Anfragen nach Dokumenten, die mit *".stml"* enden, werden von ihr behandelt. Das Ergebnis dieser Auswertung ist dann eine HTML-Seite. Die dabei ausgeführten Schritte sind im einzelnen die folgenden:

1. Durch Nutzung der persistenten Ergebnisse der Schritte 2-4 wird bei Schritt 5 weiter fortgefahren, falls sich das STML-Dokument seit dem letzten Aufruf nicht geändert hat.

2. Ansonsten wird das STML-Dokument von einen SGML-*Parser* analysiert und der Aufbau verifiziert. Insbesondere wird dabei eine Typprüfung auf Ebene der DTD durchgeführt. Das Ergebnis ist ein abstrakter STML-Syntaxbaum, der die Gliederung des Textes widerspiegelt.

3. Wenn dies erfolgreich ist, wird der STML-Baum in eine TYCOON-2-Funktion übersetzt. Dazu wird kein TL-2 Quelltext erzeugt, sondern gleich ein abstrakter Syntaxbaum in dem Format, das der eigentliche TL-2-Übersetzer als Resultat der Syntaxanalyse erwartet.

4. Der so gewonnene Syntaxbaum wird dann vom TL-2-Übersetzer in eine von der virtuellen Maschine ausführbare Funktion übersetzt, und der TL-2-Typüberprüfer prüft parallel dazu die Typkorrektheit. Beides ist erst durch die reflektive Nutzung dieser Sprachprozessoren im System möglich.

5. Bei jeder Anfrage nach einer STML-Seite wird ein Objekt der Klasse `StmlProcessor` erzeugt, das Referenzen auf die HTTP-Anfrage, den zu erzeugenden HTML-Ausgabestrom und ein besonderes Anwendungsobjekt enthält (das bereits oben angesprochene `application`-Objekt). Ihnen obliegt die Bearbeitung der Anfrage.

6. Die in Schritt 4 erzeugte Funktion wird ausgeführt und produziert das Ergebnis, eine HTML-Seite oder ein HTML-Formular. Sollten – während der Entwicklungsphase unvermeidliche – Fehler auftreten, so werden diese als Ergebnis in Textform geliefert. In allen Fällen wird es vom HTTP-Server dann an den anfragenden HTTP-Client geschickt.

Durch Nutzung von STML-Seiten und entsprechenden selbstentwickelten Anwendungsklassen und -objekten ist die Erstellung von dynamischen, personalisierbaren Informationsdiensten möglich.

5.3 Das Hotelreservierungssystem

Wie in der Zielsetzung (Abschnitt 1.1) bereits gesagt, konnten Analyse, Entwurf und Implementierung des zur Erprobung gewählten Hotelreservierungssystems nicht Bestandteil dieser Arbeit sein. Vielmehr sind sie Inhalt der Arbeit von V. RIPP, wobei dort die Entwicklung eines bruchlosen Entwicklungsprozesses im Vordergrund steht [Rip98].

Dieser Entwicklungsprozeß läßt sich in groben Zügen folgendermaßen beschreiben: Die Analysephase nimmt ihren Anfang mit der Modellierung der möglichen Interaktionen zwischen den Benutzern des Systems und dem System. Dazu werden Bildschirmmasken in Form von HTML-Formularen entwickelt. Sie unterstützen die Modellierung der möglichen Szenarien (*use-cases*). Der Vorteil ist, daß man durch das Anlegen der Formulare bereits in diesem Stadium Klarheit über die auszutauschenden Daten auf zunächst relativ informelle Weise gewinnt. Wichtiger ist noch, daß sich daraus gleich die zugrundeliegenden Geschäftsprozesse entweder in der Form von Ereignis-Prozeß-Ketten (EPK) oder sogar als Konversationsspezifikationen im Sinne der *Business Conversations* ableiten lassen. Dabei konnte gezeigt werden, daß beide äquivalent sind. Zentral ist dabei die Beobachtung, daß die Betrachtung der Geschäftspartner als Kunde-Dienstleister-Paar möglich ist. Dieses Wissen wird dann zum Entwurf der Anwendungslogik des Systems genutzt. Auf den Inhalt und das Vorgehen beim Entwurf der Anwendungslogik soll hier nicht weiter eingegangen werden.

Die entscheidenden Entwurfsüberlegungen aber waren: daß erstens zur Realisierung der Benutzungsschnittstelle ein HTTP-Client zum Einsatz kommen soll; damit sind dann sowohl Intranet- als auch Internet-Informationssysteme möglich; und zweitens, daß zur Interaktion die *Business Conversations* dienen sollen. Durch den oben beschriebenen Entwicklungsprozeß liegen bereits implizit Konversationsspezifikationen vor, so daß dies einfach möglich ist.

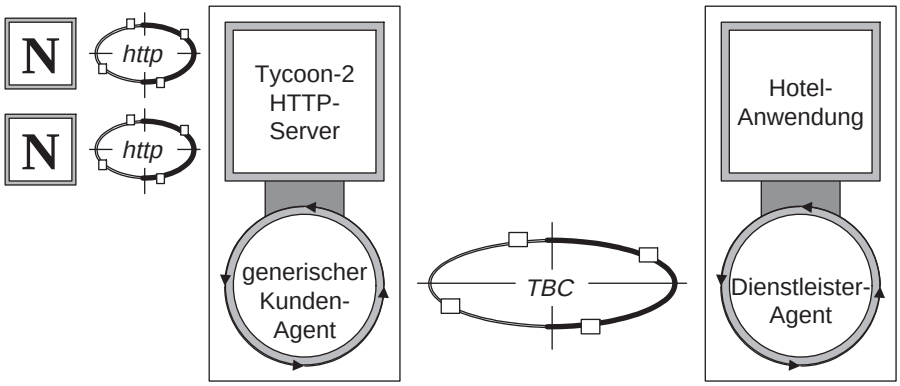


Abbildung 5.1: Architektur des Hotelreservierungssystems

Die Architektur des so entworfenen Systems ist in Abbildung 5.1 dargestellt. Ganz links, mit „N“ bezeichnet, sind zwei HTTP-Clients zu sehen. Sie stehen über das Protokoll HTTP mit dem TYCOON-2-HTTP-Server in Verbindung. Dieser kommuniziert zur Bearbeitung der eingehenden Anfragen über den sogenannten *generischen Kunden* per *Tycoon Business Con-*

versations mit einem Agenten, der die Dienste des eigentlichen Hotelreservierungssystems bereitstellt.

Die kritischen Stellen sind die Schnittstellen zwischen den einzelnen Komponenten dieses Systems. An sie sind die folgenden Anforderungen zu stellen:

HTTP-Client/HTTP-Server: Das verwendete Protokoll ist HTTP. Die serverseitige Schnittstelle ist Bestandteil des TYCOON-2-HTTP-Servers und stellt somit kein Problem dar.

HTTP-Server/Generischer Kunde: Dies ist die komplizierteste Schnittstelle. Auf der Seite des Servers gibt es die Möglichkeiten, entweder eine Subklasse von `HttpResource` zur Behandlung der HTTP-Anfragen zu entwickeln, oder aber eine STML-Seite zu schreiben, die TL-2-Code enthält, der das gleiche leistet. Das Behandeln der Anfragen besteht im wesentlichen darin, trotz des eigentlich zustandslosen Protokolls HTTP die Sitzungen zu verwalten, die durch mehrere verschiedene, gleichzeitig arbeitende Benutzer entstehen, sowie die Inhalte der HTML-Formulare geeignet zu konvertieren, sie den Regeln und Konversationen des generischen Kunden zukommen zu lassen und umgekehrt deren Resultate zu empfangen.

Generischer Kunde/Dienstleister: Da beide Seiten unter Nutzung des in Kapitel 4 beschriebenen Agentensystems entwickelt werden, ist diese Schnittstelle unproblematisch. Die Kommunikation wird vollständig vom Agentensystem abgewickelt.

Dienstleister/Hotelanwendung: Die Kommunikation geschieht über Dienstleister-Regeln, die gemäß der zugrundeliegenden Konversationsspezifikation implementiert sind.

Die zweite Schnittstelle soll im folgenden noch genauer beleuchtet werden. Der *generische Kunde* hat diesen besonderen Namen, da dieser Agent eine spezielle Kundenrolle besitzt, die – jedenfalls konzeptuell gesehen – ohne Kenntnis der Konversationsspezifikation in der Lage ist, Konversationen generisch durch Delegation der eigentlichen Bearbeitungsregel an den HTTP-Server abzuwickeln. Technisch gesehen wird dazu für jede Regel dasselbe Objekt einer speziellen Klasse gebunden, nämlich `GenericCustomerRule`. In dieser Klasse wird die Kommunikation mit dem HTTP-Server abgewickelt.

Die andere Seite (die des Servers) ist in Form einer STML-Seite implementiert. Diese Lösung wurde vorgezogen, weil sie schneller zu implementieren war. Der in der Seite eingebettete Code muß also mit der generischen Regel kommunizieren. Außerdem werden auf dieser Seite die Konvertierungen zwischen HTTP-Anfragen und HTML-Formularen einerseits und TBC-Anfragen und TBC-Dialogen andererseits vorgenommen.

An dieser Stelle wird es notwendig, die hier zur Konvertierung verwendeten Techniken zu erläutern: Wie anfangs beschrieben, wurde die Interaktion in der Analysephase bereits durch HTML-Formulare modelliert. Diese werden nun wiederverwendet, um einerseits aus ihnen maschinell die Konversationsspezifikation für das Agentensystem zu gewinnen, und um andererseits als Schablone für die Inhalte konkreter Konversationen zu dienen. Das Vorgehen ist schematisch in Abbildung 5.2 dargestellt.

Die Menge der die Konversation beschreibenden HTML-Formulare wird einmalig von dem bereits beschriebenen SGML-*Parser* analysiert. Das Resultat ist einerseits ein die SGML-

Textstruktur beschreibender Ableitungsbaum pro Formular, andererseits wird daraus insgesamt eine Konversationsspezifikation, also eine weitere Objektstruktur, erzeugt. Dabei wird jedes einzelne HTML-Formular auf genau einen Dialog abgebildet. Die Anfragen und Folgedialoge ergeben sich durch die einzelnen *Submit-Buttons* des Formulars. Die Inhaltsspezifikationen werden aus den im Formular enthaltenen Feldern gewonnen. Natürlich sind dazu einige Kunstgriffe nötig, da HTML für Formularfelder keine Typen kennt. Dieses und ähnliche Probleme werden durch geeignete Namen, die den gewünschten Typ identifizieren, gelöst. Die Analyse wird von einer speziellen Klasse geleistet, die auch die erzeugten Ableitungsbäume und die Konversationsspezifikation aggregiert und für die Benutzer, nämlich den Code der STML-Seite bzw. den generischen Kunden sowie den Dienstleister bereithält. Jedesmal, wenn sich die HTML-Formulare geändert haben, ist dieser Analyseschritt zu wiederholen.

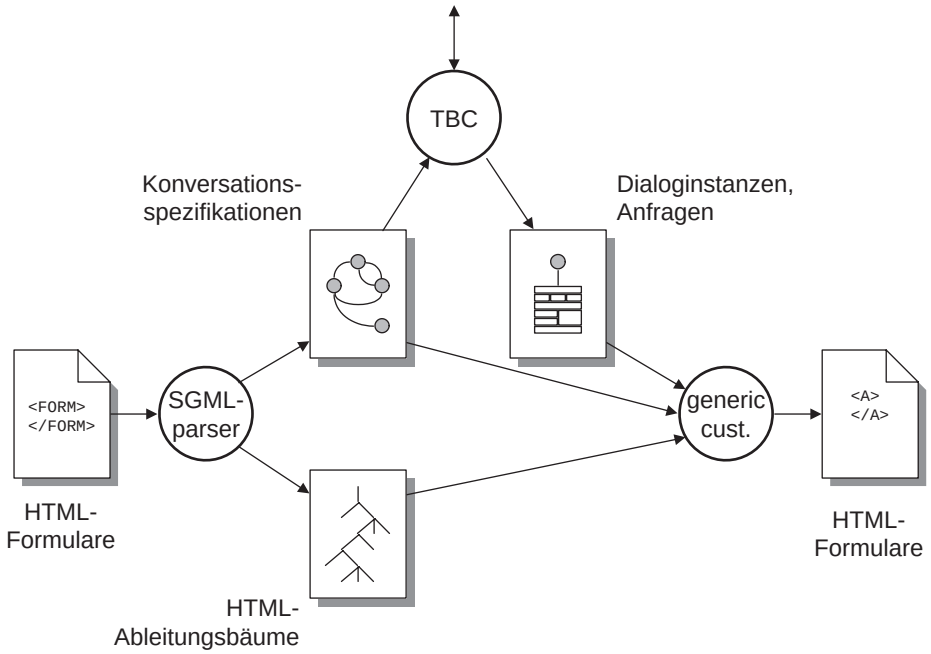


Abbildung 5.2: Schema der Wiederverwendung von Formularen

Die Konversationsspezifikation wird dann vom Agentensystem zur Abwicklung der Konversation genutzt. Die dabei erzeugten Konversationsinstanzen (also Dialoge und Inhalte) werden von dem Gespann aus generischem Kunden und der STML-Seite zur Laufzeit der Konversation unter Nutzung sowohl des abstrakten Syntaxbaums der ursprünglichen Seite als auch der Konversationsspezifikation zurück in eine HTML-Seite bzw. ein HTML-Formular umgewandelt. Diese schließlich gibt der HTTP-Server als Antwort auf eingehende Anfragen zurück.

Eine weitere Schwierigkeit stellt die dynamische Interaktion zwischen dem Code der STML-Seite und der generischen Regel dar. Beide werden jeweils von dem sie beherbergenden System, also dem HTTP-Server bzw. dem Kunden-Agenten aufgerufen, die beide als eigenständige *threads* laufen. Problematisch ist dabei, daß logisch gesehen die Ausführungszeiten von STML-Code und generischer Kunden-Regel bedingt durch die gegenseitigen

Abhängigkeiten phasenverschoben sind: Die Anfrage an den HTTP-Server liefert für die Kundenseite den ausgefüllten Dialog und die Anfrage im Sinne der *Business Conversations*, beendet also eine Regelausführung. Andererseits bedingt die Beantwortung der HTTP-Anfrage das Vorliegen der Antwort, also den Folgedialog. Der ist aber erst mit Beginn der nächsten Regelausführung bekannt. Es ist also nötig, daß sich beide Seiten asynchroner Kommunikation bedienen, um auf die jeweiligen Ergebnisse der anderen Seite zu warten. Als Mittel der Wahl stellt sich wiederum der `MessagePort` heraus.

Abbildung 5.3 zeigt die dabei entstehende Interaktion zwischen HTTP-Server, generischem Kunden und Dienstleister. Gut zu erkennen ist dabei, wie die generische Regel und der HTTP-Server aufeinander warten (hervorgehoben durch die besonders starken Doppellinien). Der diagonale, grau hinterlegte Bereich stellt den sich bei jeder Anfrage wiederholenden Ablauf dar.

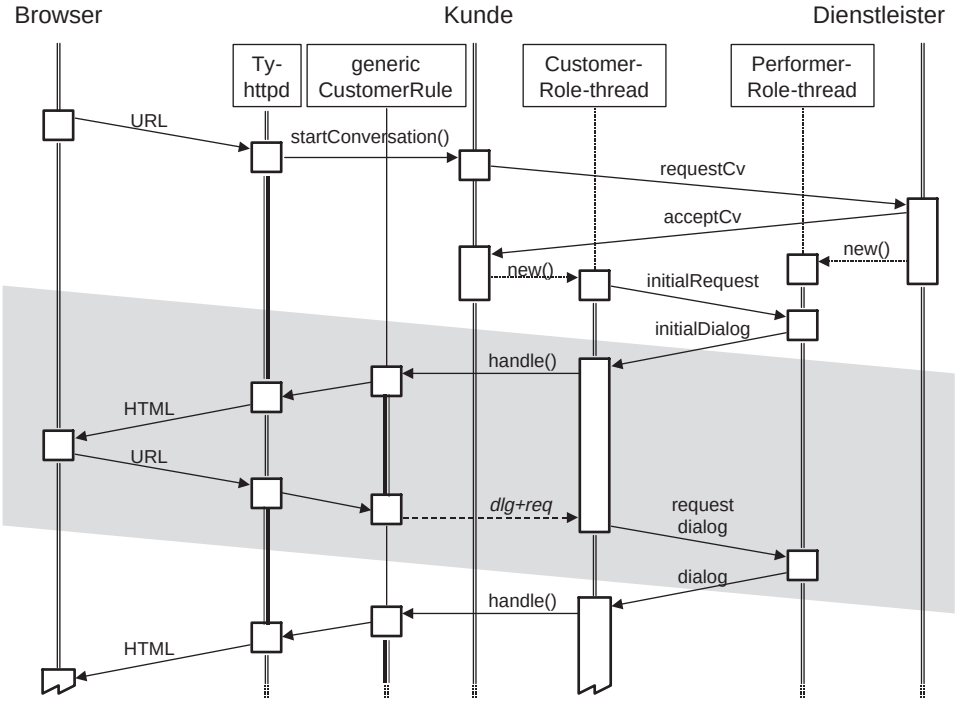


Abbildung 5.3: Interaktion zwischen Server, generischem Kunden und Dienstleister

Durch die Verwendung von versteckten Formularfeldern in den HTML-Seiten ist es möglich, Sitzungen zu simulieren, also aufeinanderfolgende Anfragen desselben Benutzers einander zuzuordnen, was HTTP selbst nicht unterstützt. Dadurch und durch ein Verzeichnis der Sitzungen in dem der STML-Seite zugeordneten Anwendungsobjekt ist es sogar möglich, mehrere Sitzungen gleichzeitig zu führen. Die Nutzung wird dadurch etwas erleichtert, daß der HTTP-Server in der vorliegenden Version alle Anfragen sequentiell abarbeitet. Bei der Sitzungsverwaltung muß weiterhin der Möglichkeit Rechnung getragen werden, daß durch eine für das Konversationsmodell inadäquate Nutzung der HTTP-Client-Historie o.ä. weiter in der Konversation zurückliegende Seiten erneut angefordert werden: So zum Beispiel muß die *Zurück*-Funktion der gängigen HTTP-Clients abgefangen werden; außerdem for-

dern manche von ihnen sogar bei Veränderung der Größe des darstellenden Fensters den Inhalt neu an. Die Lösung ist, stets die aktuelle, zuletzt erzeugte Seite nochmal zu senden. Ein interessanter Nebeneffekt ist, daß durch das Anlegen eines Lesezeichens (*bookmark*) die begonnene Konversation zu einem späteren Zeitpunkt weitergeführt werden kann.

Abschließend soll hier noch auszugsweise der Code der STML-Seite wiedergegeben werden, um die Kommunikation mit der Kundenregel zu illustrieren:

```
...
let cookie:String=formFields["cookie"],
let msg:GeCuRoMessage = nil,

cookie="" ?
{                                              (* no conversation yet *)
  let service:String = formFields["service"],
  TBC.instance.startConversation("Client/GeCuRo",service),
  cookie:=application.createContext(service),
  msg := application.getPort.get,       (* wait for initial dialog *)
  application.setContext(cookie,msg)
}:{        (* got request for next dlg + fields from dialog before *)
  ...
  msg:= application.getContext(cookie),
  msg.request := application.fillCurrentDialog(cookie,formFields),
  msg.replyPort.put(msg),
  msg := application.getPort.get,          (* wait for next dialog *)
  application.setContext(cookie,msg)
},
application.drawDialog(cookie,eout),
msg.dialog.dialogSpec.replies.size = 0 ? {        (* final dialog? *)
  application.removeContext(cookie),
}
```

Ein verstecktes Formularfeld namens `"cookie"` dient dazu, die Sitzungen zu identifizieren. Ist es leer, so stammt die Anfrage von einem initialen HTML-Formular und es muß eine neue Konversation begonnen werden. Nach dem Start der Konversation wird auf das Eintreffen des initialen Dialoges gewartet. Der Code der STML-Seite und die generische Kundenregel bedienen sich dabei der Klasse `GeCuRoMessage` zur Kommunikation. Das Anwendungsobjekt `application` enthält die zur Verwaltung der Sitzungen nötigen Verzeichnisse und Methoden, sowie die zur Konvertierung erforderlichen Methoden `fillCurrentDialog` und `drawDialog`. Nach Erhalt des initialen Dialoges wird (von `drawDialog`) das initiale HTML-Formular erzeugt. Danach ist die Ausführung dieser Seite zunächst beendet.

Die im weiteren Verlauf der Konversation nötigen Schritte des Wartens auf die Ergebnisse des Kunden und die Generierung der resultierenden Seite sind gut zu erkennen. Zunächst wird die Sitzung identifiziert (per `getContext`), dann der Dialog mit den zurückerhaltenen Formularfeldern gefüllt und die Anfrage extrahiert (dies leistet `fillCurrentDialog`), beides zusammen in der `GeCuRoMessage` an den Kunden versendet und schließlich auf den Empfang des Folgedialoges gewartet. Dieser wird dann wieder wie zuvor in ein HTML-Formular übersetzt und der Zyklus schließt sich.

5.4 Bewertung

Der traditionelle Ansatz des *common gateway interface* (CGI) erscheint technisch überholt, schlecht skalierbar und den sich dynamisch fortentwickelnden Anforderungen nur sehr ungenügend gewachsen. Dazu trägt insbesondere die schwierige Wartbarkeit bei.

Der STML-Ansatz in Verbindung mit dem in das TYCOON-2-System integrierten HTTP-Server umgeht die meisten dieser Probleme. Die Vorteile sind, daß sich die Visualisierung von komplexen Datenstrukturen damit sehr elegant und schnell bewerkstelligen läßt und daß der volle Sprachumfang von TL-2 zur Verfügung steht. Die Anbindung von weiteren Diensten fällt damit sehr leicht. Nachteilig für die Entwicklung kooperativer Informationssysteme ist, daß die Modellierung der ablaufenden Prozesse weder formal noch technisch unterstützt wird. Auch als Nachteil kann die fehlende Entkoppelung der Anwendungslogik von der Visualisierung betrachtet werden.

Diese Lücke wird gut von dem zur Realisierung des Hotelreservierungssystems erdachten Modell geschlossen. Der standardisierte Entwicklungsprozeß führt gleichermaßen zu formalen Prozeßbeschreibungen als auch zur Grundlage der Visualisierung. Letzteres läßt sich nahezu vollständig automatisieren. Ein Hauptvorteil liegt dabei in der vollständigen Entkoppelung der Anwendungslogik von der Präsentation, wobei aber beides auf einfache Weise erweiterbar bleibt, somit also der Forderung nach Evolutionsunterstützung in hohem Maße genügt. Aufgrund der in Abschnitt 4.5 geschilderten Techniken ist sogar eine unabhängige Weiterentwicklung der Anwendungslogik möglich.

Auch unter technischen Gesichtspunkten kann das realisierte System überzeugen: Die vorhandene Mehrbenutzerfähigkeit und die erreichte Geschwindigkeit sind auch für kommerzielle Anforderungen mehr als ausreichend. Hervorzuheben ist in diesem Zusammenhang auch die durch das TYCOON-2-System mögliche Portabilität.

Der Verwendung für weitere Projekte steht damit nichts im Wege; die Entwicklung neuer Dienste läßt sich auf der Grundlage des bestehenden Systems schnell vornehmen. Sinnvolle zukünftige Erweiterungen und Verbesserungen werden in Kapitel 6 angesprochen.

Kapitel 6

Schluß

Zum Schluß dieser Arbeit sollen die Ergebnisse zusammengefaßt, eine Bewertung des Erreichten vorgenommen sowie ein Ausblick auf offen gebliebene und sich neu ergebende Fragen gegeben werden.

6.1 Zusammenfassung und Bewertung

Die in dieser Arbeit entwickelte Agentenumgebung bietet einen konzeptuellen und softwaretechnischen Rahmen für anwendungsnahe Modellierung und Implementierung von verteilten Systemen und insbesondere für kooperative Informationssysteme.

Bei der Betrachtung von anderen, kommerziellen und akademischen Entwicklungen von Agentensystemen ergibt sich als wichtigste Erkenntnis, daß die Formalisierung der Koordination der Agenten ausschlaggebend für die erfolgreiche Verwendung in Informationssystemen ist. Das hierzu verwendete Modell der *Business Conversations* stellt dafür eine gut geeignete Grundlage dar.

In dieser Arbeit mußte und konnte die bisher teilweise in einigen Punkten unklare Semantik der *Business Conversations* weiter präzisiert werden. Dies betrifft besonders operationale Aspekte wie z.B. den Beginn einer Konversation oder die Fehlerbehandlung. Außerdem sind einige Weiterentwicklungen des Modells geleistet worden. So konnte die Subtypisierung von Konversationsspezifikationen nicht nur definiert, sondern auch implementiert und verwendet werden. Ferner konnte die Notation und operationale Semantik von Subkonversationen definiert sowie neue Beiträge zur Bestimmung der Bedeutung von sekundären Konversationen erbracht werden.

Als weitere wichtige Anforderungen an Agentensysteme haben sich die Aspekte Mobilität, Autonomie, Kommunikation, Asynchronizität, Benennbarkeit sowie Sicherheit erwiesen. Der Entwurf und die Implementierung wird ihnen (mit Ausnahme des ersten und des letzten Punktes) voll gerecht. Obwohl eines der wesentlichen Merkmale, die Mobilität, aufgrund fehlender Systemvoraussetzungen nicht zu realisieren war, ist der Entwurf durch migrationstolerante und ortstransparente Mechanismen darauf ausgelegt. Werden die erforderlichen Voraussetzungen in Zukunft geschaffen, so sollte die Erweiterung keinen großen

Aufwand bedeuten. Der Aspekt der Sicherheit in Agentensystemen wurde in den Betrachtungen innerhalb dieser Arbeit gänzlich ausgeklammert, da dafür sowohl konzeptionell als auch systemtechnisch noch wenige bzw. keinerlei Grundlagen bestehen.

Das prototypisch realisierte Hotelreservierungssystem hat die Leistungsfähigkeit sowohl von Modell als auch Implementierung deutlich gezeigt. Zudem verspricht die verwendete Architektur eine schnellere, wartungsfreundlichere und stärker an den wechselnden Anforderungen orientierte Entwicklung von Internet-Informationssystemen zu unterstützen. Zusammen mit dem zugrundeliegenden Analyse-, Entwurfs- und Implementierungsprozeß [Rip98] entspricht die Lösung den Anforderungen evolutionärer Systementwicklung.

Unserer Meinung nach zeigen diese Ergebnisse weiterhin folgendes: Obwohl es im Grunde nur als Agentensystem konzipiert wurde, das in erster Linie in der Lage sein soll, *mobile* Agenten zu betreiben, ist das entstandene System durch seine für ein Agentensystem notwendigen Eigenschaften (insbesondere Autonomie, Kommunikation, Asynchronizität) gerade dadurch zu einer hervorragend geeigneten Plattform für die Entwicklung von kooperativen Informationssystemen geworden. Die Synergie der Konzepte beider Seiten (Agentensystem vs. Informationssystem) erweist sich hier als überaus fruchtbar. Erst durch die Infrastruktur des Agentensystems lassen sich die Kooperations- und Kommunikationskonzepte des Konversationsmodells gewinnbringend nutzen. Aber auch das Modell der *Business Conversations* geht durch die ihm innewohnenden Abstraktions- und Beschreibungsmöglichkeiten sowohl für *Prozesse* als auch für *strukturierte Inhalte* über ein reines Kommunikationsprotokoll weit hinaus. Zukünftig wird es sicherlich als gutes Mittel für die Modellierung und Beschreibung von Geschäftsprozessen (*workflow*) einerseits und zur Definition strukturierter Dokumente im Umfeld des *information retrieval* und der damit verbundenen Anfrage- und Interaktionsmöglichkeiten andererseits verwendet werden können. Die vorhandene Medien- und Aktorenunabhängigkeit des Modells stellt die für eine breite Nutzung erforderliche Voraussetzung dar.

Aufgrund dieser Möglichkeiten stellt die fehlende Umsetzung einiger weniger Konzepte, namentlich der hierarchischen Orte und Ortewächter, keine große Einschränkung dar, zumal diesen in einem immobilen Szenario ohnehin keine große Bedeutung zukommt. Als Basis für die Entwicklung kooperativer Internet-Informationssysteme benötigt das Agentensystem die Mobilität und das Ortekonzept ohnehin nicht.

Auf der Ebene des Entwurfs und der Implementierung läßt sich rückblickend auf die weiteren in Abschnitt 1.1 genannten Ziele folgendes feststellen: Die zum Entwurf verwendeten Methoden und Notationen haben sich bewährt. Insbesondere die Verwendung von Entwurfsmustern fördert das Verständnis von objektorientiertem Entwurf und ermöglicht sehr klare, gut erweiterbare Strukturen. Die verwendete Sprache TL-2 ist sehr gut geeignet, um auf hohem Niveau entworfene Systeme zu realisieren. Tatsächlich kann durch die Mächtigkeit der vorhandenen Sprachmittel fast immer auf eine Unterscheidung zwischen Entwurfsmodellen und Implementierungsmodellen verzichtet werden.

Hervorragende Dienste hat auch das TYCOON-2-System geleistet. Weite Teile des Agentensystems konnten durch Wiederverwendung von Funktionalität der Standardbibliothek auf einfache Weise realisiert werden. Ganz besonders das Vorhandensein des HTTP-Servers und der integrierten SGML- und STML-Unterstützung haben die rasche Realisierung der Präsentationsebene des Hotelreservierungssystems entscheidend gefördert.

6.2 Ausblick

Zum Abschluß sollen offen gebliebene Punkte und neu aufgetauchte Probleme geschildert werden sowie ein Ausblick auf zukünftige Arbeiten gegeben werden.

- Den wichtigsten offenen Punkt, verglichen mit der in [Joh97] vorgenommenen Implementierung, stellt sicherlich die fehlende Mobilität der Agenten dar. Die orthogonale Mobilität in objektorientierten, persistenten Systemen stellt allerdings auch hohe Anforderungen an die zu verwendenden Mechanismen. Diese wären wahrscheinlich eng verbunden mit Problemen der Evolution von Klassen in persistenten Objektsystemen. Dies ist ein Gebiet, das noch viel Raum für weitere Untersuchungen läßt.
- Ein hierarchisch aufgebautes System von Orten zur logischen Strukturierung eines Agentensystems fehlt ebenfalls noch. Die technische Umsetzung davon ist einfach. Wünschenswert ist bei der Einführung der speziellen immobilen Agenten, der Ortewächter, daß diese gleichfalls durch Nutzung von Konversationsspezifikationen und Konversationen kommunizieren. Insgesamt betrachtet ist die Implementierung von Orten und Wächtern erst sinnvoll, wenn die Migration von Agenten möglich ist.
- Die Standardisierungsbemühungen im Umfeld der mobilen Agenten, insbesondere seitens der OMG, sollten weiter beobachtet werden. Als besonders für das TBC-System interessant könnte sich dabei die Nutzung von standardisierten Namenskonventionen herausstellen. Dieser Punkt wurde wegen der fehlenden Mobilität bisher nur prototypisch gelöst. Ob es sinnvoll ist, die in den Vorschlägen definierten Schnittstellen zur Administrierung des Agentensystems zu übernehmen, kann als fraglich bezeichnet werden. Die Interoperabilität hängt damit eng zusammen.
- Die in dieser Arbeit völlig unberücksichtigten Aspekte der Sicherheit von Agentensystemen stellen ein weites Feld für weitere Untersuchungen dar. Hier ist insbesondere das Fehlen von grundlegenden Mechanismen für Authentisierung, Autorisierung und Kryptographie innerhalb des `Tycoon-2`-Systems noch ein Mangel.
- Auf der Ebene der Realisierung von Internet-Informationssystemen wäre es sinnvoll, die bisherige Lösung so umzuändern, daß keine STML-Seite mehr verwendet wird; stattdessen ist die Entwicklung einer eigenen `HttpResource` anzustreben, die die Konversationen eines generischen Kunden mit Hilfe der bereits beschriebenen Mitteln visualisieren kann. Der Aufwand dafür wäre nicht sehr hoch, würde aber die Klarheit des Systems verbessern und es vollständig von der – ohnehin nur rudimentären – Benutzung von STML befreien.
- Das grundsätzliche Vorgehen der Wiederverwendung von HTML-Formularen hat sich bewährt. Allerdings ist die Spezifikation von Konversationen durch HTML-Formulare bislang relativ umständlich und unelegant. Hier wäre die Entwicklung einer SGML-konformen Dokumententypdefinition (DTD) sinnvoll, die dies besser unterstützt und leicht auf HTML abzubilden ist. Sie sollte dann weiterhin von den bisherigen Werkzeugen verwendet werden können.
- In diesem Zusammenhang ist auch die *generische Visualisierung* von Konversationen zu sehen. Dies wäre praktisch der umgekehrte Weg des hier vorgestellten Verfahrens.

Der generische Kunde müßte dafür ohne Konversationsspezifische Formatvorlagen aus den Konversationsinhalten HTML-Formulare erzeugen können. Durch Nutzung der reflektiven Eigenschaften und z.B. des *visitor-patterns* wäre dies sehr einfach zu bewerkstelligen. Probleme sind nur aufgrund der hohen Orthogonalität des Typsystems der Inhaltsspezifikationen zu erwarten, da dies tief geschachtelte Strukturen erlaubt, die nur aufwendig zu visualisieren sind.

- Eine zur Zeit durchgeführte Studienarbeit [Hup98] hat die Aufgabe, den Zugriff auf die Konversationshistorie in den Kunden- und Dienstleisterregeln zu verbessern. Dazu werden zum einem neue sprachliche Konstrukte eingeführt, und zum anderen wird eine statische Prüfung des Quelltextes der Regeln vorgenommen. Beides geschieht durch eine Erweiterung des Übersetzers bzw. Typprüfers. Durch die Analyse der zugrundeliegenden Konversationsspezifikation wird so eine statische Prüfung auf zur Laufzeit dynamische Korrektheit der erweiterten Regeln ermöglicht. Es ist sinnvoll, dieses System voll in das bereits vorhandene zu integrieren, so daß es etwa automatisch jede zu bindende Rolle prüft.

- Agenten sind auch ein aktuelles Teilgebiet der künstlichen Intelligenz (KI). Die dort entwickelten Modelle und die verwendeten Wissensrepräsentationssprachen basieren vielfach ebenfalls auf Erkenntnissen der Sprechakttheorie, so beispielsweise KQML. Eine vertiefende Untersuchung der Gemeinsamkeiten könnte ein erhöhtes Verständnis der Agentenkoordination bringen.

- Schließlich könnten einige Teile der Implementierung des Agentensystems dem Prozeß des *refactoring* unterzogen werden. Dies bedeutet, ein System ohne Änderung der Funktionalität inkrementell, also in kleinen Schritten, so zu verändern, daß der Gesamtentwurf verbessert wird. Dies ist vor allen Dingen bei schon lange und häufig geänderten Systemen erforderlich. Dieses Vorgehen wäre aufgrund der inzwischen gewonnenen Erfahrungen beim objektorientierten Entwurf an einigen Stellen ratsam. Ansatzpunkt wäre z.B. das in [GHJV95] genannte Prinzip, die Komposition der Vererbung gegenüber zu bevorzugen.

Auch die Ergebnisse weiterer, indirekt oder direkt mit demselben Thema befaßten Arbeiten können interessant sein. Hier sind beispielsweise Modelle für das *workflow-manangement*, die die *Business Conversations* nutzen wollen, die Integration des SAP R / 3-Systems mit Hilfe sogenannter *wrapper*, die ebenfalls zur Kommunikation die *Business Conversations* verwenden, oder die Entwicklung eines *Internet-Shops* auf Basis der hier entwickelten Umgebung für Internet-Informationssysteme zu nennen.

Danksagung

Das Gelingen dieser Arbeit wäre nicht möglich gewesen ohne die Hilfe einer ganzen Reihe von Personen, denen ich an dieser Stelle herzlich danken will: Als erstes ist natürlich die hervorragende Betreuung durch Florian Matthes und Leonie Dreschler-Fischer zu nennen. Nicht vergessen werde ich die – manchmal im wörtlichen Sinne – erschöpfenden Diskussionen mit Florian Matthes. Bei meinen ersten, unsicheren Schritten mit TYCOON-2 half mir Ingo Richtsmeier. Ihm und der Firma HIGHER-ORDER, die mir das System anfangs zur Verfügung stellte, habe ich dafür zu danken. Weiterhin muß ich allen Mitarbeitern und Studenten für die freundschaftliche Atmosphäre am Arbeitsbereich STS danken. Nur so macht Studieren Spaß! Von unschätzbarem Wert war – wie immer – die kongeniale Zusammenarbeit mit Volker Ripp. Ihm als ersten Benutzer meines Systems verdanke ich wertvolle Anregungen und Kritik. Für die gewissenhafte Durchsicht des Textes dieser Arbeit danke ich Johannes Bitterling, Karin Kettner und Dr. Arnim Wegner, meinem Vater. Ihnen ist es zu verdanken, daß noch etliche Fehler und so manches Satzungetüm unschädlich gemacht werden konnten. Zuletzt, aber am allermeisten möchte ich meiner Freundin Karin für ihre Liebe und ihr Vertrauen (sowie für all die Dinge, die ich hätte erledigen sollen) danken.

Literaturverzeichnis

[Aus62] AUSTIN, J.: How to do things with words / Oxford University Press, Oxford. 1962. – Forschungsbericht

[BF93] BARBUCEANU, M.; FOX, M.S. The Design of COOL: A Language for representing Cooperation-Knowledge in Multi-Agent Systems. http://www.ie.utoronto.ca-/eil/abs-page/abs-intro.html. 1993

[BF94] BARBUCEANU, M.; FOX, M.S.: The Architecture of a Generic Agent for Collaborative Enterprises / University of Toronto. 1994. – EIL Working Paper

[BJR96] BOOCH, G.; JACOBSON, I.; RUMBAUGH, J. The Unified Modeling Language, Version 0.91. http://www.rational.com/ot/uml.html. September 1996

[Boo94] BOOCH, G.: *Object-Oriented Design with Applications*. second edition. Benjamin/Cummings Publishing Company, Inc., 1994

[BS82] BRODIE, M.; SCHMIDT, J.W.: Final Report of the ANSI/X3/SPARC DBS-SG Relational Database Task Group. In: *ACM SIGMOD Record* 12 (1982), Nr. 4

[BW94] BEALE, Russel; WOOD, Andrew: Agent-Based Interaction. In: *People and Computers* IX (1994), S. 239–245

[CGG+97] CRYSTALIZ, INC.; GENERAL MAGIC, INC.; GMD FOKUS; IBM CORPORATION; THE OPEN GROUP: Mobile Agent System Interoperability Facilities Specification / OMG. 1997. – OMG TC Document orbos/97-10-05

[Cha97] CHAUHAN, Deepika: *JAFMAS: A Java-based Agent Framework for Multiagent Systems Development and Implementation*, ECECS Department, University of Cincinnati, Master's thesis, 1997

[DDJ+97] DE MICHELIS, Giorgio; DUBOIS, Eric; JARKE, Matthias; MATTHES, Florian; MYLOPOULOS, John; PAPAZOGLOU, Mike; POHL, Klaus; SCHMIDT, Joachim; WOO, Carson ; YU, Eric: Cooperative Information Systems: A Manifesto. In: PAPAZOGLOU, Mike P. (Hrsg.); SCHLAGETER, Gunther (Hrsg.): *Cooperative Information System: Trends and Directions*. Academic Press, New York and London, 1997

[Ern98] ERNST, Matthias: *Typüberprüfung in einer polymorphen objektorientierten Programmiersprache: Analyse, Design und Implementierung eines Typprüfers für Tycoon-2*, Fachbereich Informatik, Universität Hamburg, Studienarbeit, Februar 1998

[FGHW88] FLORES, F.; GRAVES, M.; HARTFIELD, B.; WINOGRAD, T.: Computer Systems and the Design of Organizational Interaction. In: *ACM Transactions on Office Information Systems* 6 (1988), Nr. 2, S. 153–172

[FS97] FOWLER, Martin; SCOTT, Kendall: *UML Distilled: Applying the Standard Object Modelling Language*. Addison-Wesley Publishing Company, 1997

[GHJV95] GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSADES, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995

[GHS91] GREWENDORF, Günther; HAMM, Fritz; STERNEFELD, Wolfgang: *Sprachliches Wissen: Eine Einführung in moderne Theorien der grammatischen Beschreibung*. 5. Auflage. Frankfurt / Main : Suhrkamp, 1991

[GM95] GAWECKI, A.; MATTHES, F.: TooL: A Persistent Language Integrating Subtyping, Matching and Type Quantification / FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ. 1995 (FIDE / 95 / 135). – FIDE Technical Report Series

[GM96] GAWECKI, A.; MATTHES, F.: Integrating Subtyping, Matching and Type Quantification: A Practical Perspective. In: *Proceedings of the 10th European Conference on Object-Oriented Programming, ECOOP'96*. Linz, Austria : Springer-Verlag, Juli 1996, S. 26–47

[GMSS97] GAWECKI, Andreas; MATTHES, Florian; SCHMIDT, Joachim W.; STAMER, Sören: Persistent Object Systems: From Technology to Market / Higher-Order Informations- und Kommunikationssysteme GmbH. 1997. – Forschungsbericht

[GW97] GAWECKI, Andreas; WIENBERG, Axel: STML Developers Guide / Higher-Order Informations- und Kommunikationssysteme GmbH. 1997. – Forschungsbericht

[HOX98] Higher-Order Informations- und Kommunikationssysteme GmbH. http: / / www.higher-order.de. 1998

[Hup98] HUPE, Patrick: *Zustandstypisierung*, Fachbereich Informatik, Universität Hamburg, Studienarbeit, 1998

[JCJÖ92] JACOBSON, I.; CHRISTERSON, M.; JONSON, P.; ÖVERGAARD, G.: *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley Publishing Company, 1992

[Joh97] JOHANNISSON, Nico: *Eine Umgebung für mobile Agenten: Agentenbasierte verteilte Datenbanken am Beispiel der Kopplung autonomer Internet Web Site Profiler*, Fachbereich Informatik, Universität Hamburg, Diplomarbeit, April 1997

[Kru97] KRUSE, Wolfgang: *Historienmangement für kooperative Aktivitäten*, Fachbereich Informatik, Universität Hamburg, Diplomarbeit, Dezember 1997

[LC96] LANGE, Danny B.; CHANG, Daniel T.: IBM Aglets Workbench. Programming Mobile Agents in Java: A White Paper / IBM Corporation. 1996. – Forschungsbericht

[LS87] LOCKEMANN, P.C. (Hrsg.); SCHMIDT, J.W. (Hrsg.): *Datenbank-Handbuch*. Springer-Verlag, 1987

[Mat93] MATTHES, F.: *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung*. Springer-Verlag, 1993

[Mat96] MATHISKE, Bernd: *Mobilität in persistenten Objektsystemen*, Fachbereich Informatik, Universität Hamburg, Dissertation, Mai 1996

[Mat97a] MATTHES, Florian: Business Conversations: A High-Level System Model for Agent Coordination. In: *Proceedings of the Sixth International Workshop on Database Programming Languages, Estes Park, Colorado*, Springer-Verlag, August 1997

[Mat97b] MATTHES, Florian: Mobile Processes in Cooperative Information Systems. In: *STJA'97, Smalltalk und Java in Industrie und Ausbildung*, Springer-Verlag, September 1997

[Mey88] MEYER, B.: *Object-oriented Software Construction*. Prentice Hall, Englewood Cliffs, New Jersey, 1988 (International Series in Computer Science)

[MMM93] MATHISKE, B.; MATTHES, F.; MÜSSIG, S.: The Tycoon System and Library Manual / Fachbereich Informatik, Universität Hamburg. 1993 (212-93). – DBIS Tycoon Report

[MMS94] MATTHES, F.; MÜSSIG, S.; SCHMIDT, J.W.: Persistent Polymorphic Programming in Tycoon: An Introduction / Fachbereich Informatik, Universität Hamburg. 1994 (FIDE / 94 / 106). – FIDE Technical Report

[MMS96] MATHISKE, B.; MATTHES, F.; SCHMIDT, J.W.: On Migrating Threads. In: *Journal of Intelligent Information Systems* 8 (1996), Nr. 2

[MSS97] MATTHES, F.; SCHRÖDER, G.; SCHMIDT, J.W.: Tycoon: A Scalable and Interoperable Persistent System Environment. In: ATKINSON, M.P. (Hrsg.): *Fully Integrated Data Environments*. Springer-Verlag, 1997

[MW97] MATTHES, Florian; WIENBERG, Axel: Visualizing Persistent Objects using Higher-Order Functions in SGML / Technische Universität Hamburg-Harburg, Arbeitsbereich Softwaresysteme. 1997. – Forschungsbericht

[OHE96] ORFALI, Robert; HARKEY, Dan; EDWARDS, Jeri: *The Essential Distributed Objects Survival Guide*. John Wiley & Sons, 1996

[OK97] OSHIMA, Mitsuru; KARJOTH, Guenther: Aglets Specification / IBM Corporation. 1997. – Forschungsbericht

[RBP+91] RUMBAUGH, J.; BLAHA, M.; PREMERLANI, W.; EDDY, F.; LORENSEN, W.: *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1991

[Ric97] RICHTSMEIER, Ingo: *Kommunizierende Informationssysteme am Beispiel autonomer Internet WebSiteProfiler: Vergleich objekt- und agentenbasierter Ansätze*, Fachbereich Informatik, Universität Hamburg, Diplomarbeit, Januar 1997

[Rip98] RIPP, Volker: *Verbesserung der Lokalität und Wiederverwendbarkeit von Geschäftsprozeßspezifikationen: Probleme und Lösungsansätze am Beispiel kundenorientierter Hotelgeschäftsprozesse*, Fachbereich Informatik, Universität Hamburg, Diplomarbeit, Mai 1998

[Sea69] SEARLE, J.: Speech Acts / Cambridge University Press, Cambridge. 1969. – Forschungsbericht

[Wah98] WAHLEN, Jens: *Entwurf einer objektorientierten Sprache mit statischer Typisierung unter Beachtung kommerzieller Anforderungen*, Fachbereich Informatik, Universität Hamburg, Diplomarbeit, 1998

[Wei98] WEIKARD, Marc: *Entwurf und Implementierung einer portablen multiprozessorfähigen virtuellen Maschine für eine persistente, objektorientierte Programmiersprache*, Fachbereich Informatik, Universität Hamburg, Diplomarbeit, 1998

[Whi94] WHITE, J.E.: Telescript Technology: The Foundation for the Electronic Marketplace / General Magic Inc., Mountain View, California, USA. 1994. – White Paper

[Wie97] WIENBERG, Axel: *Bootstrap einer persistenten objektorientierten Programmierumgebung*, Fachbereich Informatik, Universität Hamburg, Studienarbeit, August 1997

[Win87] WINOGRAD, T.A.: A Language / Action Perspective on the Design of Cooperative Work / Stanford University. 1987 (No. STAN-CS-87-1158). – Forschungsricht

[Wit60] WITTGENSTEIN, L.: *Philosophische Untersuchungen*. Frankfurt / Main, 1960