Diplomarbeit

Intranetunterstützung für problemlösende Arbeitsprozesse im Team

von Jan Stövesand
Fachbereich Informatik
Universität Hamburg

Betreuung:

Prof. Dr. Florian Matthes
Arbeitsbereich Softwaresysteme
Technische Universität Hamburg-Harburg

Prof. Dr. Rüdiger Valk
Theoretische Grundlagen der Informatik
Fachbereich Informatik
Universität Hamburg

28. August 1999

Inhaltsverzeichnis

1 Einleitung **3**
1.1 Inhalt und Struktur 3

2 Problemlösende Arbeitsprozesse **5**
2.1 Stellen von Aufgaben 6
2.2 Delegation von Aufgaben 6
2.3 Erweiterung von Aufgaben 7
2.4 Art der Bearbeiter 7
2.5 Nebenläufigkeit 7

3 Business Conversations **9**
3.1 Konzepte des Modells 9
3.2 Initiierung und Delegation 10
3.3 Konversationskontext 11
3.4 BC-Addons 12

4 Modellierung **15**
4.1 Grundlagen 15
4.2 Entwicklung des Modells 17
4.2.1 Konversationen zwischen Verteiler, Aufgabensteller und Bearbeitern 17
4.2.2 Initiierung 17
4.2.3 Antwort 18
4.2.4 Delegation 19
4.2.5 Konstruktion des Modells 19
4.3 Steuerung mit Objektnetzen 23
4.4 Einzelkonversationen 24
4.5 Erweiterungen des Modells und Bewertung 25

5 Entwicklungsumgebung **27**
5.1 Microsoft Transaction Server 27
5.1.1 Aktivierung von Komponenten 28
5.1.2 Transaktionskontrolle 30
5.1.3 Zugriffsschutz von Komponenten 30
5.2 Datenbankserver 31

5.3 Webserver 31
5.3.1 Active Server Pages 32
5.4 Extensible Markup Language (XML) 36
5.5 Extensible Style Language (XSL) 37

6 Entwurf und Implementierung **41**
6.1 Klassen des statischen Teils 42
6.1.1 Die Klasse *Agent* 42
6.1.2 Die Klasse *Conversation* 43
6.1.3 Die Klasse *Dialog* 43
6.2 Klassen des dynamischen Teils 44
6.2.1 Rollenklassen 45
6.2.2 Konversationsklassen 46
6.3 Interaktion mit dem System 47
6.3.1 Starten einer neuen Konversation 47
6.3.2 Fortsetzen von Konversationen 47
6.4 Komponentenstruktur und Datenbankmodell 50
6.5 Historienmanagement mittels eines *BC-Addons* 51

7 Beispiel **53**
7.1 Problemstellung und Analyse 53
7.2 Modellierung 54
7.3 Implementierung 56
7.3.1 Konversationsspezifikation, Dialoge und Rollen 56
7.3.2 Programmierung der Arbeitsprozeßkomponente 58
7.4 Zeitmanagement mittels eines BC-Addons 62

8 Zusammenfassung und Ausblick **63**
8.1 Zusammenfassung 63
8.2 Ausblick 64

A XSL-Beispiele **65**
A.1 Beispiel 1 65
A.2 Beispiel 2 66
A.3 Beispiel 3 68

B Dialogbeschreibung **71**
B.1 Syntax in Backus-Naur-Form 71
B.2 Syntax als Document Type Defnintion (DTD) 71
B.3 Beispiel für eine XSL-Datei 72

C Beispielapplikation **77**
C.1 Eingabe der Daten für die Konversationsspezifikation *aufgabestellen* 77
C.2 Aktionsfunktion 78

Literaturverzeichnis **81**

Abbildungsverzeichnis

2.1 Sternförmige Systemtopologie 7

3.1 Initiierung einer neuen Konversation 11
3.2 Delegation einer Aufgabe 11
3.3 Schnittstellen der BC-Addons 13

4.1 Ein einfaches Elementares Objektsystem 16
4.2 Systemnetz mit Aufgabensteller, Verteiler und Bearbeitern 18
4.3 Verfeinerung des Verteilers 19
4.4 Initiierung von Konversationen 20
4.5 Beantwortung durch die Bearbeiter 21
4.6 Delegation von Konversationen 22
4.7 Arbeitsprozeß als ENS 23
4.8 Erweiterter Arbeitsprozeß 24
4.9 Vollständiges Systemnetz mit Beschriftung entsprechend der Interaktionsrelation 25
4.10 Vergröberungen und Aufteilung des Objektnetzes 26

5.1 Architektur des Microsoft Transaction Server 28
5.2 Indirekter Zugriff auf Objekte über die Kontexthülle 29
5.3 Benutzersicht durch den MTS 31
5.4 Baumdarstellung von in XML codierten Daten 38
5.5 XSL-Prozessor 38
5.6 Verschiedene Visualisierungen derselben XML-Daten 39

6.1 ASP-Skript zum Einloggen in das System 42
6.2 Implementierung von *getConversations* in einer Rollenkomponente 45
6.3 Starten einer neuen Konversation 48
6.4 Fortsetzen einer Konversation 49
6.5 Klassenstruktur 50
6.6 Darstellung einer Ablaufhistorie 51

7.1 Arbeitsprozeß als Elementares Netzsystem 54
7.2 Arbeitsprozeß als erweitertes Elementares Netzsystem 55
7.3 Teilung des Arbeitsprozesses in Einzelkonversationen 57

7.4 Dialog zur Bearbeitung einer Aufgabe 61
7.5 Zeiterfassung bei der Bearbeitung von Aufgaben 62

Wenn du eine weise Antwort verlangst,
mußt du vernünftig fragen.

- Johann Wolfgang von Goethe
heute ist seinen 250. Geburtstag

Ich danke all denen, die mich bei der Erstellung dieser Arbeit mit ihrem Wissen, ihrer Hilfe und ihrer Geduld unterstützt haben und denen, die es mir ermöglicht haben so weit zu kommen.

Ich hoffe, ich habe immer vernünftig gefragt.

Jan Stövesand

Kapitel 1

Einleitung

Intranetunterstützung bei der rechnergestützten Bearbeitung von Arbeitsprozessen bietet den Vorteil einer plattformunabhängigen Benutzerschnittstelle. Browser, wie der Netscape Communicator oder der Microsoft Internet Explorer sind für die großen Betriebssysteme (MacOS, Unix, Windows) frei verfügbar. Dadurch läßt sich der Entwicklungsaufwand reduzieren und die Portierung der Clients auf mehrere Betriebsysteme entfällt.

Die Programme die die Arbeitsprozesse abbilden, laufen auf einem zentralen *Applikationsserver*. Die Verbindung zwischen dem Applikationsserver und den Browsern wird dabei über einen *Webserver* hergestellt. Die Verwaltung der zu speichernden Daten erfolgt über einen *Datenbankserver*, der vom Applikationsserver angesprochen wird. Diese Dreiteilung der Aufgaben in Darstellung durch einen Webserver, Programmlogik mit einem Applikationsserver und Datenhaltung in einem Datenbankserver wird als *3-Schichten-Architektur (three tier architecture)* bezeichnet. In dieser Arbeit wird eine solche Architektur verwendet, um ein System zur Unterstützung von *problemorientierten Arbeitsprozessen* zu erstellen. Dazu werden Arbeitsprozesse anhand eines theoretischen Modells untersucht, daß sowohl die Ablaufcharakteristik als auch die Kommunikationsstruktur in den Arbeitsprozessen erhält. Bei der Implementierung des Systems werden Aspekte wie *Transaktionssicherheit*, *Sitzungsunterstützung (Sessionmanagement)* und die *Trennung von Daten und Visualisierung* untersucht.

1.1 Inhalt und Struktur

Unter problemlösenden Arbeitsprozessen werden Abläufe verstanden, bei denen Probleme oder Aufgaben in das System gelangen und dort von einem oder mehreren Bearbeitern gelöst werden. Dazu wird das Konzept eines zentralen Verteilers verwendet, welcher Aufgaben von Aufgabenstellern entgegennimmt und Bearbeitern zuweist. Dieses Konzept des Workflowsystems wird in Kapitel 2 beschrieben.

Diesem Workflowsystemmodell wurde das Workflowmodell der *Business Conversations* zugrundegelegt. Die Kommunikation zwischen den Aufgabenstellern, dem Verteiler und den Bearbeitern folgt der Idee von Einzelkonversationen. Da ein ablauforientierter Arbeitsprozeß somit als Aggregation mehrerer Einzelkonversationen betrachtet werden kann,

entsteht die Notwendigkeit, Konversationen in einem *Konversationskontext* zusammenzufassen, der dann den Gesamtarbeitsprozeß bildet. In Kapitel 3 wird beschrieben, wie das Modell der *Business Conversations* verwendet wird.

Um ablauforientierte Arbeitsprozesse mit dem konversationsorientierten Workflowmodell zu verbinden, werden *Elementare Objektsysteme (EOS)* verwendet, da sie durch die Trennung von Systemnetz und Objektnetz gut für die Analyse geeignet sind. Die Verwendung von EOS wird in Kapitel 4 beschrieben.

Die für die Erstellung des Workflowsystems verwendete Entwicklungsumgebung wird in Kapitel 5 einführend beschrieben. Es werden die Aufgaben der Teilkomponenten voneinander abgegrenzt. Außerdem werden die Möglichkeiten der Intranetunterstützung durch diese Umgebung verdeutlicht.

Aus dem entwickelten Workflowsystemmodell entsteht zusammen mit den Möglichkeiten der Entwicklungsumgebung dann ein Workflowsystem. In Kapitel 6 wird auf die Implementierung des Systems eingegangen. Die Umsetzung des Workflowmodells durch Komponententechnologie sowie die Datenhaltung in einer relationalen Datenbankumgebung wird eingehend erläutert. Weiterhin wird in diesem Kapitel die Verwendung von XML beschrieben. XML wird verwendet, um eine intranetfähige Trennung von Daten und Visualisierung der Dialoge der *Business Conversations* zu erreichen.

Als Abschluß wird in Kapitel 7 ein Beispiel für die Verwendung des Workflowsystems gegeben. Anhand einer Aufgabenstellung aus der Praxis wird der Weg von der Analyse des Arbeitsprozesses bis zur Implementierung im Workflowsystem demonstriert.

Kapitel 2

Problemlösende Arbeitsprozesse

In diesem Kapitel werden die Gruppe der problemlösenden Arbeitsprozesse beschrieben, sowie das Workflowmodell, das den besonderen Ansprüchen dieser Arbeitsprozesse genügt. Das so entstandene Workflowsystem bildet mit seiner sternförmigen Topologie die Grundlage für alle in dieser Arbeit untersuchten Arbeitsprozesse.

Das Workflowmodell, das in dieser Arbeit erstellt wird, soll nicht für alle möglichen Arbeitsprozesse ausgelegt sein, sondern beschränkt sich vielmehr auf eine Teilmenge aller Arbeitsprozessen, die problemlösenden Arbeitsprozesse. Problemlösende Arbeitsprozesse zeichnen sich dadurch aus, daß ein Aufgabensteller mit einem Problem oder einer Aufgabe an einen oder mehrere Bearbeiter herantritt. Der Aufgabensteller wartet solange, bis er eine Antwort von einem Bearbeiter erhält. Typische Beispiele für diese Art von Arbeitsprozessen sind :

Beispiel 1: Bei einer Serviceabteilung in einer Firma können die Mitarbeiter Probleme mit der EDV-Anlage schildern. Zum Beispiel tritt das Problem auf, daß Druckaufträge von einem bestimmten Drucker nicht mehr ausgeführt werden. Der Mitarbeiter ruft also die Serviceabteilung an und schildert das Problem. Er wird an einen der Techniker weitergeleitet, der für die Drucker im Hause zuständig ist. Dieser prüft das Gerät, kann aber keinen Fehler feststellen. Daher leitet er die Aufgabe an einen Kollegen weiter, der die Druckerspooler betreibt. Dieser stellt einen Absturz im Programm fest, wodurch der Drucker nicht mehr angesprochen werden konnte. Der Fehler wird behoben und bei der Serviceabteilung als erledigt gemeldet. Der Mitarbeiter der Serviceabteilung informiert nun den Kollegen, daß das Problem mit dem Drucker behoben ist.

Beispiel 2: Ein Kunde betritt eine Autowerkstatt und möchte, daß an seinem Fahrzeug die Winterreifen aufgezogen werden. Außerdem soll das Kühlmittel überprüft werden. Die Auftragsannahme leitet dieses an die Werkstattmitarbeiter weiter. Zuerst werden die Räder abgebaut. Der Mitarbeiter gibt die Räder an einen seiner Kollegen weiter, damit die neuen Reifen aufgezogen und ausgewuchtet werden. Danach nimmt er die fertigen Räder wieder entgegen und baut diese an das Fahrzeug an. Ein anderer Mitarbeiter prüft das Kühlmittel. Dabei wird ein Defekt an den Kühl-

schläuchen entdeckt. Das wird dem Werkstattbüro gemeldet, welches sich mit dem Kunden in Verbindung setzt. Der Schaden soll ebenfalls behoben werden. Dies wird dem Mitarbeiter in der Werkstatt gemeldet, er behebt den Schaden und meldet dem Werkstattbüro, daß alle Aufgaben erledigt sind. Der Kunde wird informiert und kann sein Fahrzeug abholen.

Beispiel 3: Ein Student betritt einen Kopierladen und wünscht von einem Skript eine Kopie, die zusätzlich auch gebunden werden soll. Zuerst werden also die Kopien durch ein Kopiergerät erstellt und schließlich mit einer Bindemaschine gebunden. Die fertige Kopie wird an den Studenten gegeben. Dieser bezahlt, wodurch der Arbeitsprozeß abgeschlossen ist.

Diese drei Vorgänge sollen Beispiele für problemlösende Arbeitsprozesse geben. Dabei werden unterschiedliche Anforderung sichtbar, denen ein Workflowsystem gerecht werden muß, um solche Arbeitsprozesse unterstützen zu können.

2.1 Stellen von Aufgaben

Um einen problemlösenden Arbeitsprozeß zu starten, muß zuerst die Aufgabe formuliert werden. Ein Aufgabensteller muß also in der Lage sein, die Aufgabe an das System zu übergeben und eventuell sogar einen Bearbeiter auszuwählen. Alternativ kann aber auch ein Bearbeiter durch das System automatisch vergeben werden. Dazu wird in allen Fällen ein Verteiler verwendet, der die Aufgabe an den gewünschten Bearbeiter weiterleitet. Ist die Aufgabe beendet, wird die Antwort auch über eben diesen Verteiler wieder an den Aufgabensteller zuückgeliefert. Das System muß also in der Lage sein, Aufgaben entgegenzunehmen und an die Bearbeiter weiterzuleiten. Weiterhin werden die Aufgaben nicht einfach an die Bearbeiter abgegeben, sondern es besteht (soweit das für den Arbeitsprozeß nötig ist) die Möglichkeit, den Aufgabensteller von der Beendigung der Aufgabe zu unterrichten. Diese Anordnung von Aufgabensteller, Verteiler und Bearbeiter läßt eine sternförmige Topologie entstehen, die sich in der Modellierung, aber auch in der Implementierung wiederfinden läßt. Abbildung 2.1 verdeutlicht den Fluß der Aufgaben unter Verwendung eines Verteilers.

2.2 Delegation von Aufgaben

Ein Austausch der Aufgaben zwischen den Bearbeitern muß möglich sein. Dies umfaßt auch die Abgabe der gesamten Aufgaben wie bei dem ersten Beispiel. Hier übergibt der Techniker, der für die Drucker zuständig ist, die Aufgabe an den Kollegen, der die Druckerspooler überwacht. Das Beispiel der Autowerkstatt zeigt aber, daß es für Bearbeiter auch möglich sein muß, Unteraufgaben zu formulieren, wie hier die Abgabe der Reifen an einen Kollegen, der das Aufziehen und Auswuchten übernimmt. Den Anbau übernimmt dann wieder der ursprüngliche Kollege. Innerhalb des Systems kann man also sowohl als Bearbeiter als auch als Aufgabensteller fungieren.

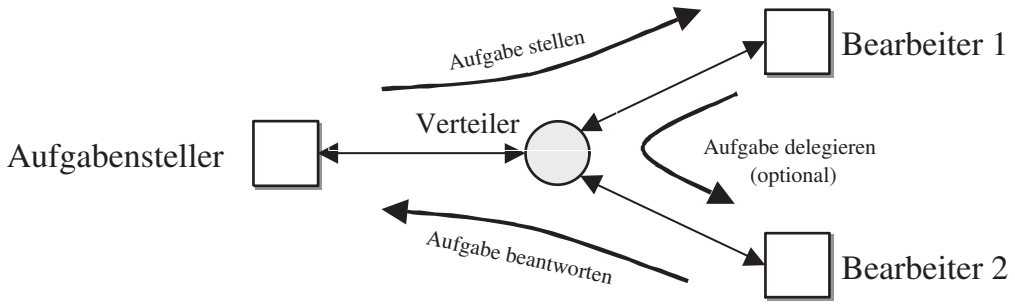


Abbildung 2.1: Sternförmige Systemtopologie

2.3 Erweiterung von Aufgaben

Die Entscheidung, ob eine Aufgabe gelöst ist, ist nicht immer eindeutig von den Bearbeitern zu treffen. Daher wird die Aufgabe an den Aufgabensteller in Form einer Antwort zurückgeleitet, der prüfen kann, ob seine Ansprüche befriedigt wurden. Sind sie es nicht, so kann die Aufgabe auch erneut an die Bearbeiter geleitet werden. Ein zusätzlicher Aspekt ist die Erweiterung der Aufgabe, wie im Beispiel der Autowerkstatt. Während der Bearbeitung der Aufgaben ist noch ein Problem aufgetreten, das nach Rücksprache mit dem Aufgabensteller in den Katalog der zu erledigenden Aufgaben aufgenommen wurde.

2.4 Art der Bearbeiter

Die Rolle der Bearbeiter wurde in den ersten beiden Beispielen immer durch Menschen übernommen. Das Beispiel des Kopierladens zeigt jedoch, daß durchaus auch Maschinen oder vielleicht Programme die Aufgaben einer Bearbeiters übernehmen können. Die Möglichkeit, zusätzlich zur Kommunikation mit Menschen, eine Kommunikation mit Hard- und Software zu ermöglichen, ist daher wünschenswert.

2.5 Nebenläufigkeit

Arbeitsprozesse sind in der Regel nicht durchgängig seriell, sondern können auch über *nebenläufige* Teilstücke verfügen. Bei dem Beispiel der Autowerkstatt ist zum Beispiel eine mögliche Nebenläufigkeit vorhanden, wenn ein Mitarbeiter das Kühlmittel prüft, während ein anderer die Reifen wechselt. Der Aspekt der Nebenläufigkeit wird in dieser Arbeit aber nur theoretisch untersucht. In der Implementierung werden die Arbeitsprozesse serialisiert.

Kapitel 3

Business Conversations

In diesem Kapitel wird die Verwendung von Business Conversations in dem im vorigen Kapitel beschriebenen Workflowmodell erläutert. Das Modell der Business Conversations wurde von Florian Matthes [Matthes 97; Matthes 98] und Nico Johannisson [Johannisson 97] in Zusammenarbeit mit Ingo Richtsmeier [Richtsmeier 97] entwickelt. Eine praxisorientierte Anwendung der *Business Conversation* ist außerdem in [Wegner 98] zu finden. In diesen Arbeiten wird das Modell genau beschrieben und untersucht, weswegen in diesem Kapitel auch nur auf die Einzelheiten eingegangen wird, die diese Arbeit direkt betreffen. Außerdem werden hier die dem Modell eigenen Begriffe eingeführt, die im Rest dieser Arbeit Verwendung finden.

In den weiteren Abschnitten werden Erweiterungen des Modells beschrieben, die sich aus den Anforderungen für die Implementierung ergeben. So werden mehrere Einzelkonversationen in einem Konversationskontext zusammengefaßt. Dies ergibt sich durch die Verwendung eines zentralen Verteilers, da hierdurch Arbeitsprozesse durch mehrere Einzelkonversationen realisiert werden.

Weiterhin werden sogenannte *BC-Addons* eingeführt, durch welche die Fähigkeiten von *Business Conversations* erweitert werden können, ohne das Modell zu verändern.

3.1 Konzepte des Modells

Integraler Bestandteil des Modells ist die Interaktion zweier Konversationspartner. Hierbei wird durch den einen Partner, den *Kunden*, ein Dienst gefordert und der andere Partner, der *Dienstleister*, versucht diesen Dienst zu erbringen. Realisiert wird die Konversation durch *Dialoge*. Diese entsprechen den Formularen in der realen Welt. In diesen Dialogen können *Inhalte* der Konversation angezeigt und geändert werden. Außerdem werden in den Dialogen mögliche Folgeaktionen vorgegeben. Auf diese Weise repräsentiert ein Dialog einen bestimmten Punkt in einer längeren Beziehung zwischen zwei Partnern. Jeder Zustand beschreibt somit einen Zustand in einer Konversation zwischen einem Kunden und einem Dienstleister.

Der Verteiler, welcher im letzten Kapitel eingeführt wurde, nimmt hierbei zum einen die Rolle eines Dienstleisters an, und zwar bei der Konversation mit dem Aufgabensteller, der

der Kunde in der Konversation ist. Bei der Weiterleitung der Aufgabe an einen Bearbeiter übernimmt der Verteiler aber selbst die Rolle des Kunden, und der Bearbeiter tritt als Dienstleister auf. Der Umfang der Inhalte einer Konversation ist im Rahmen dieser Arbeit statisch. Die Anzahl der Inhalte kann zu dem Zeitpunkt, an dem die Konversation gestartet wird, nicht mehr erweitert werden. So hat zum Beispiel eine Konversation, bei der ein Buch bestellt werden soll, in ihren Dialogen Inhalte wie Buchname, ISBN-Nummer, Autor, etc. sowie etwaige Felder, in denen der Dienstleister den Kunden informieren kann, wenn etwa eine Bestellung fehlgeschlagen ist.

Die Menge der möglichen Anfragen, die ein Kunde ausgehend von einem Zustand einer Konversation an einen Dienstleister richten kann, ist durch den Dialog bestimmt, der den aktuellen Zustand der Konversationen repräsentiert. Bei Verwendung von HTML-Formularen geben zum Beispiel die Aktionsschaltflächen vor, welche Aktion ausgelöst werden soll. Aber auch bestimmte Eingaben für die Inhalte einer Konversation können verwendet werden, um die gewünschte Aktion festzulegen. Die Auswertung der Eingaben und die daraus resultierende Aktion erfolgen durch eine sogenannte *Aktionsfunktion*. Darin werden Regeln beschrieben, die prinzipiell festlegen, von welchem Zustand man durch welche Aktion zu welchem Folgezustand gelangt. Diese werden in Abschnitt 6.2.2 im Rahmen der Konversationsklassen beschrieben, in welchen diese Funktionen implementiert werden.

Die Dialoge einer Konversation, sowie die zugehörigen Inhalte, werden in einer *Konversationsspezifikation* festgelegt. Dadurch ist sichergestellt, daß während der Konversation keine undefinierten Zustände erreicht werden können.

3.2 Initiierung und Delegation

Bei der sternenförmigen Topologie, bestehend aus dem Aufgabensteller, dem Verteiler und den Bearbeitern, die für diese Arbeit verwendet wird, werden neue Konversationen aus bestehenden Konversationen heraus gestartet. Während der Verteiler für die Aufgabensteller als Dienstleister fungiert, leitet er Aufgaben an die Bearbeiter weiter und übernimmt hier die Rolle eines Kunden.

Bei der Art, wie der Verteiler neue Konversationen mit den Bearbeitern erstellt, unterscheidet man zwischen einer *Initiierung* und einer *Delegation*. Abbildung 3.1 verdeutlicht das Vorgehen bei einer Initiierung. Die Aufgabe, die von einem Aufgabensteller kommt, führt immer zur Initiierung einer neuen Konversation mit einem Bearbeiter. Will dieser Bearbeiter, z.B. zum Lösen einer Teilaufgabe, eine Aufgabe an einen anderen Bearbeiter weiterleiten, so wird wiederum eine neue Konversation initiiert. Dies ist für den Aufgabensteller transparent. Der Bearbeiter 1 wartet auf die Antwort des Bearbeiters 2 und führt dann selber die Bearbeitung fort. Zum Schluß erfolgt die Beantwortung der Aufgabe an den Aufgabensteller. Ein Beispiel hierfür wird in Kapitel 2 gegeben. In der Autowerkstatt werden die Räder von einem Mitarbeiter (Bearbeiter 1) zuerst abgebaut und dann an einen anderen Mitarbeiter (Bearbeiter 2) zum Auswuchten gegeben. Nach Fertigstellung dieser Teilaufgabe werden die Räder wieder von dem Mitarbeiter (Bearbeiter 1) an das Fahrzeug angebaut, der sie auch abgebaut hat.

Anders gestaltet sich der Ablauf, wenn eine Aufgabe an einen anderen Bearbeiter

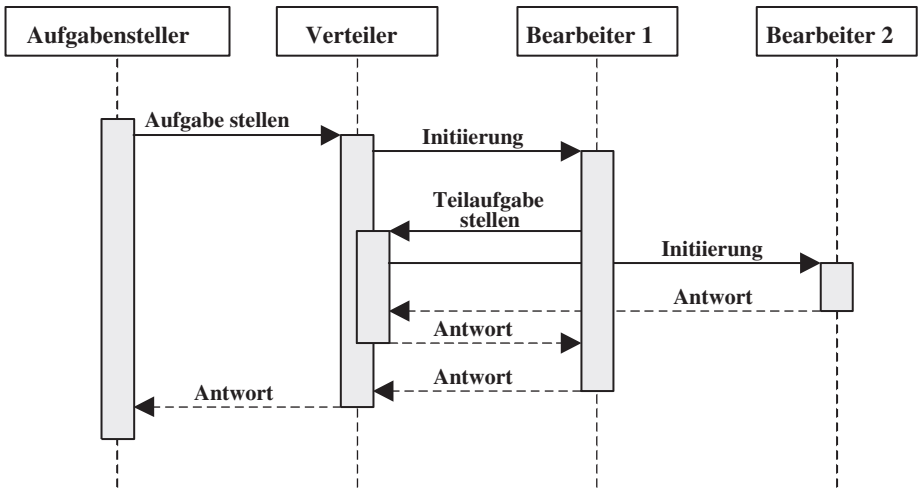


Abbildung 3.1: Initiierung einer neuen Konversation

weitergeleitet werden soll. Dies wird in Abbildung 3.2 veranschaulicht. Der Bearbeiter 1 übergibt die Aufgabe vollständig an den Bearbeiter 2, wobei dieser auch die Zuständigkeit für die Aufgaben gegenüber dem Aufgabensteller übernimmt. Für den Bearbeiter 1 ist die Bearbeitung damit abgeschlossen. Die Beantwortung der Aufgabe durch den Bearbeiter 2 wird direkt an den Aufgabensteller gegeben.

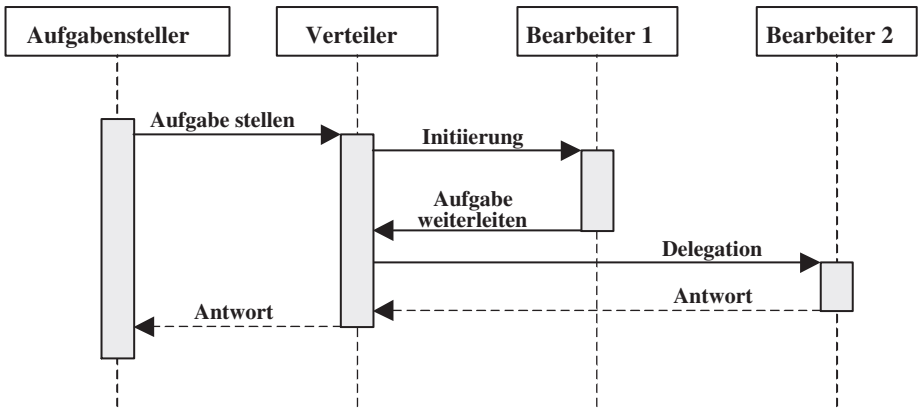


Abbildung 3.2: Delegation einer Aufgabe

3.3 Konversationskontext

Wie in Kapitel 2 beschrieben, besteht ein Arbeitsprozeß aus mehreren Einzelkonversationen. Diese sind gemäß dem Modell der *Business Conversations* jedoch unabhängig voneinander

ander und können nicht direkt in Verbindung zueinander gebracht werden. Dies ist aber wünschenswert, um Aussagen über den aktuellen Status eines Arbeitsprozesses machen zu können. Zu diesem Zweck wird ein *Konversationskontext* eingeführt, in dem mehrere Konversationen zu einer Gesamtkonversation zusammengefaßt werden. Alle Konversationen zwischen dem Aufgabensteller und dem Verteiler, sowie zwischen dem Verteiler und den beteiligten Bearbeitern sind dadurch immer miteinander in Beziehung zu setzen. Dies ist unter anderem auch für die Erzeugung einer konversationsübergreifenden Ablaufhistorie hilfreich.

3.4 BC-Addons

Zuweilen möchte man bei der Verwendung von *Business Conversations* den Funktionsumfang erweitern, ohne in das Modell einzugreifen. Ein Historienmanagement, das Auskunft über den Ablauf einer Konversation gibt, ist im Modell zum Beispiel nicht vorgesehen. Ebenso fehlt ein Zeitmodul, das protokolliert, wie lange eine Konversation gedauert hat bzw. wie lange eine Konversation in einem bestimmten Zustand verweilte. Dies kann durch Zusätze zu den *Business Conversations*, sogenannten *BC-Addons*, erfolgen. Dies sind Funktionssammlungen, die über zwei Schnittstellen verfügen. Die Funktionen der ersten Schnittstelle, der *Konversationsschnittstelle*, werden von den Aktionsfunktionen (siehe Abschnitt 6.2.2) der Konversationen aus aufgerufen. Dadurch können Daten der Konversationen gespeichert werden, die erst zu einem späteren Zeitpunkt analysiert werden sollen. Die *externe Schnittstelle* dient dazu, diese Daten in aufbereiteter Form auszugeben.

Die Trennung der Schnittstellen hat zwei Vorteile. Zum einen sind die Daten in ihrer rohen Form nicht direkt zugänglich, sondern nur über die externe Schnittstelle. Dadurch lassen sich benutzerabhängige Sichten der Daten erstellen. Gerade am Beispiel eines Historienmanagements (siehe Abschnitt 6.5) läßt sich deutlich erkennen, daß nicht jeder Benutzer auf die vollen Informationen einer solchen Historie zugreifen können soll.

Aber es gibt auch geschwindigkeitstechnische Vorteile. Bei der Zeitverfolgung von Konversationen (siehe Abschnitt 7.4) meldet sich jeweils nur die gerade aktivierte Konversation an. Dadurch entstehen in der Datenbank eine Reihe von Aktivierungsmeldungen mit den IDs der Konversation und dem Zeitpunkt der Aktivierung. Das ist zur Laufzeit äußerst schnell zu erledigen. Bei der Analyse werden diese Aktivierungsmeldungen der Reihe nach analysiert und erst hieraus läßt sich eine Gesamtlaufzeit einer einzelnen Konversation ermitteln. Wäre dies schon zu Laufzeit der Konversation erfolgt, wäre die Ausführung unnötig verzögert worden.

Aufgabe eines BC-Addons ist die Speicherung von Rohdaten zur Laufzeit der Konversationen und einer davon unabhängigen Analyse der Daten. Diese Trennung der beiden Schnittstellen wird in Abbildung 3.3 verdeutlicht.

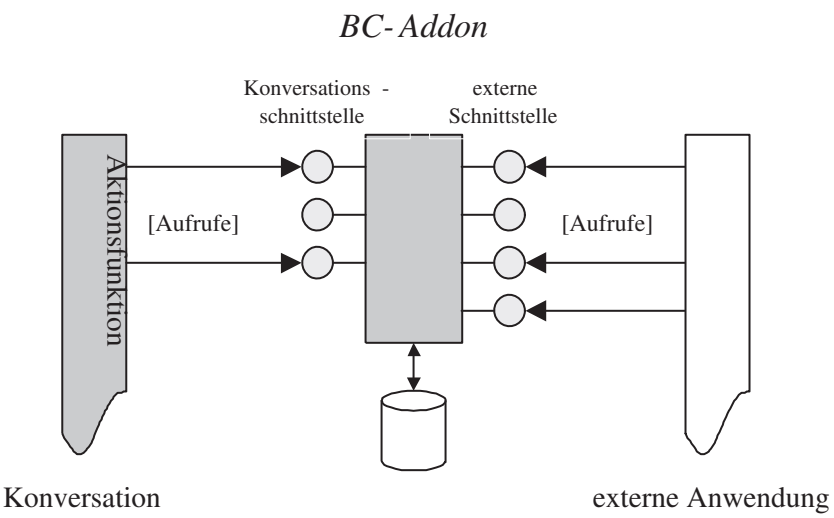


Abbildung 3.3: Schnittstellen der BC-Addons

Kapitel 4

Modellierung

Bei der Analyse von Arbeitsprozessen, wie sie in Kapitel 2 beschrieben wurde, gelangt man häufig zu sequentiellen Prozeßbeschreibungen. Eine Aufgabe wird gestellt, die Aufgabe wird an einen Bearbeiter weitergeleitet, dieser gibt die Aufgabe mittels des Verteilers an einen anderen Bearbeiter weiter, usw. Das Vorgehen erinnert an einen Laufzettel, der in einer vorgegebenen Reihenfolge abgearbeitet wird. Dadurch erhält man eine *ablauforientierte* Beschreibung eines Arbeitsprozesses. Arbeitsprozesse, die dagegen nach dem in Kapitel 3 vorgestellten Modell der *Business Conversations* beschrieben werden, beruhen auf der Identifikation von Konversationen zwischen Kunden und Dienstleistern. Die so erstellten Prozeßbeschreibungen sind daher *konversationsorientiert*. Beide Modelle haben Ihre Vorteile. Deswegen soll bei der Modellierung versucht werden, beide Ansätze beizubehalten. Dazu werden in diesem Kapitel *Elementare Objekt Systeme (EOS)* [Valk 98b] beschrieben, die eine Verbindung zwischen diesen beiden Workflowvarianten schaffen.

4.1 Grundlagen

Elementare Objektsysteme (EOS) stellen eine Erweiterung von *Elementaren Netzsystemen (EN)* dar, welche in [Thiagarajan 87] beschrieben werden.

Definition 1. Ein *Elementares Netzsystem* (ENS) ist ein Tupel $P = (S, T, F, m_0)$ mit einer endlichen Menge von Stellen S und einer endlichen Menge von Transitionen T, wobei $S \cap T = \emptyset$, einer Flußrelation $F \subseteq (S \times T) \cup (T \times S)$ und einer Anfangsmarkierung $m_0 \subseteq S$.

Hierbei werden die Marken, welche ein ENS durchlaufen, wiederum durch ein eigenständiges ENS ersetzt. Ein EOS besteht somit aus einem Systemnetz und einem Objektnetz, das die Marke bzw. Marken des Systemnetzes darstellt. Das Modell sieht auch vor, verschiedene Objektnetze durch das Systemnetz zu transportieren, was für diese Arbeit aber nicht notwendig ist und weswegen hier nur auf den Sonderfall der unären EOS eingegangen wird.

Beim Systemnetz und auch beim Objektnetz handelt es sich um ENS. Unäre EOS werden demnach wie folgt definiert:

Definition 2. Ein *Unäres Elementares Objektsystem* ist ein Tupel $EOS = (SN, ON, \rho)$ mit

- $SN = (S, T, F, M_0)$ ist ein ENS, mit $|M_0| = 1$, genannt Systemnetz
- $ON = (B, E, W, m_0)$ ist ein ENS, genannt Objektnetz
- $\rho \subseteq T \times E$ ist die Interaktionsrelation

Die Marken des Systemnetzes sollen hierbei als das Objektnetz verstanden werden, welches das Systemnetz durchläuft. Als *Bi-Markierung* wird hierbei eine Markierung (M, m) verstanden, wobei M die momentane Markierung des Systemnetzes und m die Markierung des Objektnetzes ist. Die *interaktive Schaltregel* für Elementare Objektsysteme besagt, daß ein Paar von Transitionen $[t, e] \in T \times E$ in einer Bi-Markierung (M, m) *aktiviert* ist, wenn $(t, e) \in \rho$ und t und e in M bzw. m aktiviert sind. Die weiteren sogenannten *autonomen Schaltregeln* werden hier nicht weiter vorgestellt, da sie für die weitere Betrachtung keine Relevanz haben. Eine genaue Beschreibung ist in [Valk 98b] zu finden.

Welche Rolle spielen EOS in der Modellierung des Workflowsystems. Abbildung 4.1 zeigt ein sehr einfaches EOS. Das Systemnetz soll hierbei die Ressourcen eines Copy-Shops darstellen. Es kann kopiert (t_1), geschnitten (t_2) und gebunden (t_3) werden.

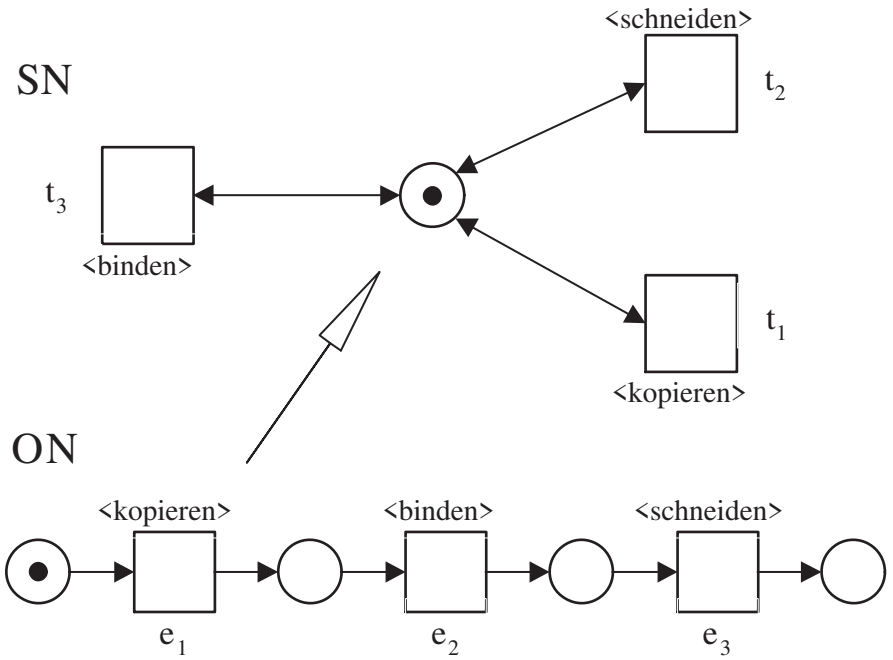


Abbildung 4.1: Ein einfaches Elementares Objektsystem

Betrachtete man das Systemnetz als eigenständiges ENS, so wären alle Transitionen aktiviert und könnten schalten. Es wäre also eine Schaltfolge möglich, bei der gebunden

wird (t_3), bevor man etwas kopiert (t_1) hat. Das Systemnetz alleine kann daher nur als Ressourcenlieferant betrachtet werden. Das dargestellte Objektnetz stellt einen typischen Arbeitsprozeß dar. Zuerst soll etwas kopiert werden (e_1), dann werden die Seiten gebunden (e_2) und schließlich werden die Kanten noch einmal sauber geschnitten (e_3). Jetzt ändert sich auch die Aktivierung des Systemnetzes. Da $[t_1, e_1] \in \rho$ und $[t_2, e_1], [t_3, e_1] \notin \rho$ ist nur noch das Paar $[t_1, e_1]$ aktiviert und kann schalten. Durch den mit dem Objektnetz repräsentierten Arbeitsprozeß kann also die Reihenfolge gesteuert werden, in der die mit dem Systemnetz dargestellten Ressourcen verwendet werden.

4.2 Entwicklung des Modells

Mit Hilfe der im letzten Abschnitt vorgestellten Elementaren Objektsysteme sollen die in Kapitel 2 beschriebenen problemorientierten Arbeitsprozesse modelliert werden. Aufgabensteller, Verteiler und Bearbeiter sind aus Kapitel 2 bekannt und werden durch ein Systemnetz dargestellt. Die Arbeitsprozesse, die durch dieses Workflowsystem bearbeitet werden sollen, werden durch Objektnetze modelliert. Dadurch erhält man auf der einen Seite die sternförmige Topologie des dialogorientierten Workflowsystems, kann aber gleichzeitig ablauforientierte Arbeitsprozesse untersuchen, ohne die Struktur anpassen zu müssen. So erhält man ein universelles Workflowsystem, an dem sich unterschiedliche Arbeitsprozesse abarbeiten und untersuchen lassen. In den folgenden Unterkapiteln wird das System sukzessive entwickelt, indem es immer weiter angepaßt wird, um alle Funktionen erfüllen zu können, welche in Kapitel 2 beschrieben werden.

4.2.1 Konversationen zwischen Verteiler, Aufgabensteller und Bearbeitern

Abbildung 4.2 zeigt ein erstes einfaches Systemnetz, daß das aus Kapitel 2 bekannte Zusammenspiel von Aufgabensteller (t_0), Verteiler (t_K), sowie zwei Bearbeitern (t_1, t_2) darstellt. Es sind auch schon die Konversationszyklen erkennbar, bestehend aus $\{t_0, s_{v0}, t_K, s_{b0}\}$, $\{t_1, s_{v1}, t_K, s_{b1}\}$ und $\{t_2, s_{v2}, t_K, s_{b2}\}$.

Allerdings kann noch keine Konversation zwischen dem Aufgabensteller und dem Verteiler stattfinden, da t_K nicht aktiviert ist. Grund dafür ist, daß sich in s_{v1} und s_{v2} keine Marken befinden. Da die Konversation zwischen Verteiler und Aufgabensteller unabhängig von den Konversationen des Verteilers mit den Bearbeitern ist, müßte t_K jedoch schalten dürfen. Daher wird eine Verfeinerung des Systemnetzes, speziell der Transition t_K benötigt. Dies wird in Abbildung 4.3 dargestellt.

Die Transition t_K wird in die Transitionen t_{K0}, t_{K1} und t_{K2} aufgeteilt. Dadurch können unabhängige Konversationen zwischen dem Verteiler und dem Aufgabensteller bzw. den Bearbeitern stattfinden.

4.2.2 Initiierung

Unter der Initiierung einer neuen Konversation wird die Möglichkeit des Verteilers verstanden, aus einer bestehenden Konversation heraus eine neue zu starten. Dies ist zum

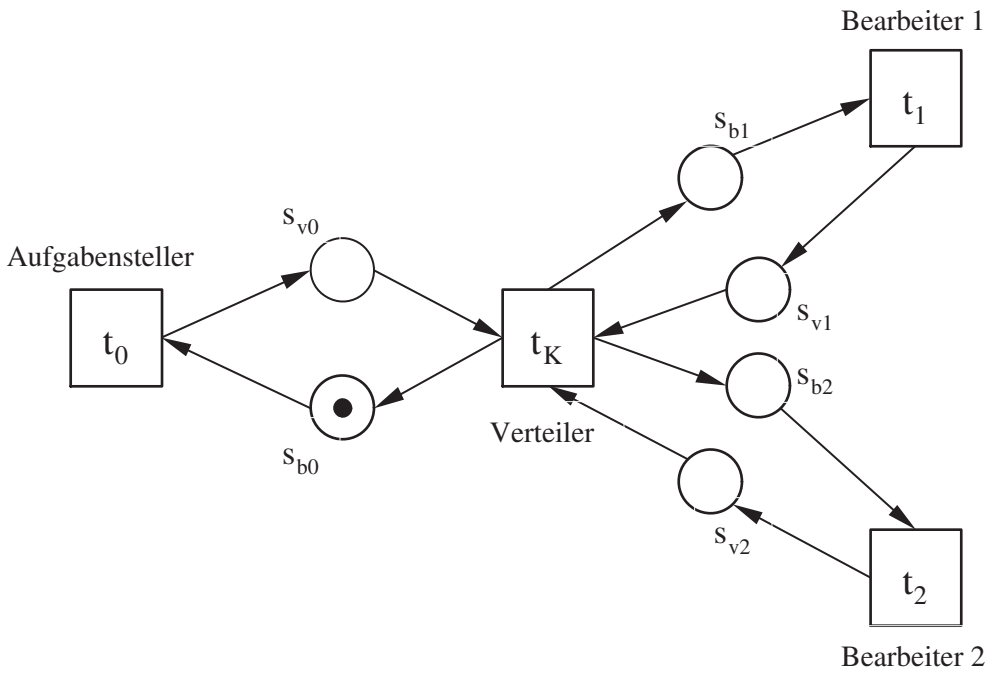


Abbildung 4.2: Systemnetz mit Aufgabensteller, Verteiler und Bearbeitern

Beispiel notwendig, wenn eine Aufgabe seitens eines Aufgabenstellers formuliert worden ist. Um diese von dem gewünschten Bearbeiter bearbeiten zu lassen, muß eine Konversation zwischen dem Verteiler und dem Bearbeiter gestartet werden.

Hierzu wird eine zusätzliche Transition hinzugefügt (Abbildung 4.4), die die Aufgabe des Verteilers repräsentiert, neue Konversationen zu initiieren. t_I wird analog zu t_K zu den Transitionen t_{I1} und t_{I2} verfeinert, um eine unabhängige Initiierung von Konversationen mit Bearbeiter 1 (t_1) und Bearbeiter 2 (t_2) zu ermöglichen. Jetzt können mit dem Systemnetz schon Arbeitsprozesse abgearbeitet werden, bei denen zuerst eine Aufgabe formuliert $[(t_{K0}, t_0)^*]$und dann durch t_{I1} oder t_{I2} an einen Bearbeiter weitergeleitet und dort bearbeitet wird.

4.2.3 Antwort

Nachdem eine Aufgabe von einem Aufgabensteller erstellt worden ist und diese durch eine Initiierung an einen Bearbeiter weitergeleitet wurde, ist im Netz von Abbildung 4.4 die Konversation zwischen dem Verteiler und dem Aufgabensteller beendet, da sich weder in s_{v0} noch in s_{b0} Marken befinden. Es ist aber durchaus sinnvoll, daß der Aufgabensteller von der Lösung der Aufgabe informiert wird. Die Konversation sollte also nur solange pausieren, bis die Bearbeiter die Aufgabe gelöst haben. Hierzu werden Transitionen eingeführt, die die Konversation des Verteilers mit dem Aufgabensteller wieder aufleben las-

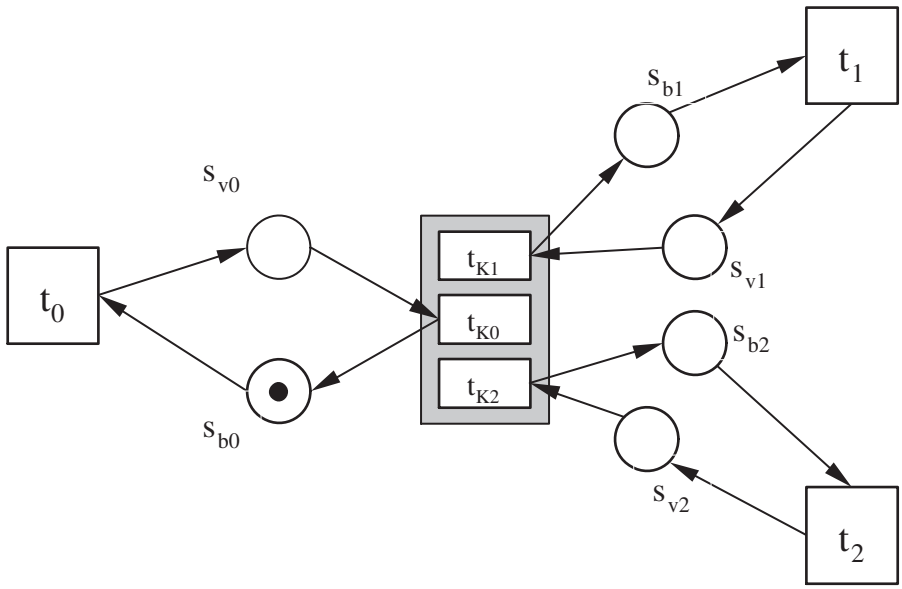


Abbildung 4.3: Verfeinerung des Verteilers

sen. Zusätzlich wird eine Stelle s_w eingeführt, durch die das Warten des Aufgabensteller dargestellt wird. Abbildung 4.5 zeigt diese Erweiterungen.

4.2.4 Delegation

Zur Lösung komplexer Aufgaben kann es notwendig sein, daß eine Aufgabe durch mehrere Bearbeiter gelöst wird. Hierzu muß es möglich sein, eine Aufgabe von einem Bearbeiter an einen anderen weiterzuleiten. Dieser Vorgang wird im weiteren als *Delegation* bezeichnet. Hierbei wird das Systemnetz derart erweitert, daß es Transitionen gibt, die formal die eigene Konversation beenden und eine Konversation zwischen dem Verteiler und dem anderen Bearbeiter beginnen. Dies wird in Abbildung 4.6 verdeutlicht. Der Ablauf, der sich hinter der Delegation verbirgt, wird in Abschnitt 3.2 detailliert beschrieben.

4.2.5 Konstruktion des Modells

Bisher wurde das Modell lediglich für zwei Bearbeiter konstruiert. Eine Erweiterung auf n Bearbeiter ist aber ohne weiteres möglich. Im folgenden werden die Konstruktionsregeln beschrieben, durch die das Systemnetz mit n Bearbeitern eindeutig bestimmt ist.

Zuerst benötigt man eine Transaktion t_0, welche den Aufgabensteller darstellt, sowie eine Transaktion für jeden Bearbeiter t_{1-n}.

$$T_{AB} = \{t_0\} \cup \{t_i | 1 \leq i \leq n\}$$

Der Verteiler besteht aus vier Transaktionsgruppen zur Initiierung (I), Konversation (K), Delegation (D) und zur Beantwortung (A). Diese Aktionen sind in den letzten

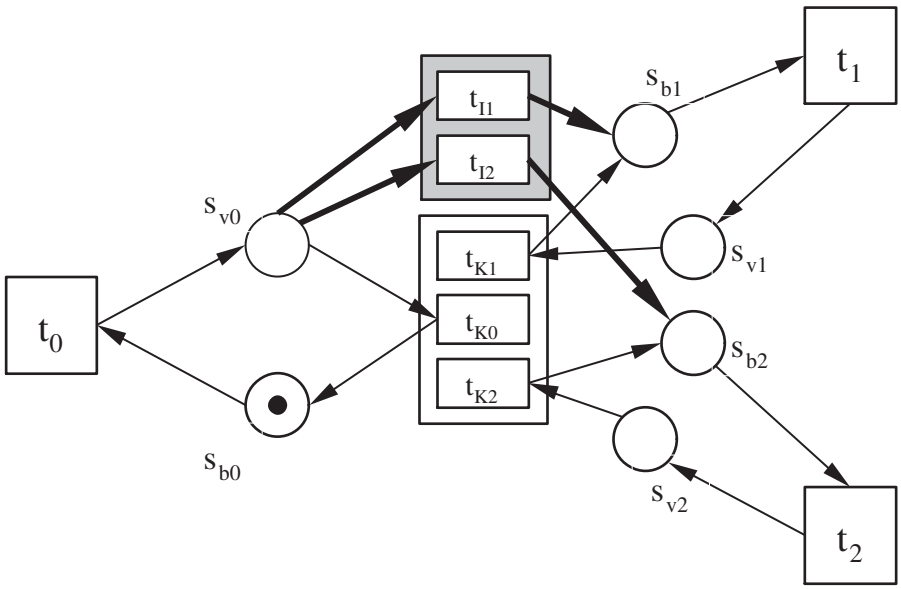


Abbildung 4.4: Initiierung von Konversationen

Abschnitten beschrieben. Da von der Konversation mit dem Aufgabensteller eine Konversation mit jedem Bearbeiter ausgelöst werden können soll, muß auch eine Transaktion für jeden Bearbeiter vorhanden sein. Dies sind die Transaktionen t_{I1} - t_{In}.

Die Transaktionsgruppe *Konversation* ist für die Fälle zuständig, in denen nur Daten einer Konversation geändert werden, ohne eine Aktion wie Initiierung oder Delegation auszuführen. Diese Transaktionen bilden also lediglich die Konversation zwischen Aufgabensteller bzw. den Bearbeitern mit dem Verteiler ab. Hierzu werden dementsprechend $n+1$ Transaktionen benötigt (t_{K0} - t_{Kn}).

Die Delegation ist der Vorgang, durch den eine Aufgabe von einem Bearbeiter an einen anderen weitergeleitet wird. Daher werden hier $n-1$ Transaktionen pro Bearbeiter i benötigt. Das Schalten der Transaktion $t_{Dj,i}$ bedeutet hierbei, daß die Aufgabe vom Bearbeiter j an den Bearbeiter i delegiert wird. Eine Delegation an sich selbst wird ausgeschlossen, indem die Kanten $(s_{vi}, t_{Di,i})$ $(t_{Di,i}, s_{bi})$ in der Flußrelation F nicht enthalten sind. Vereinfacht gesprochen, gibt es keine Delegationstransition, die die Stellen s_{vi} und s_{bi} eines Bearbeiters miteinander verbindet.

Schließlich muß es noch möglich sein, eine Aufgabe von den Bearbeitern zum Aufgabensteller zurückzuschicken und diesen aus seiner Warteposition herauszuholen. Dies geschieht mit den Transitionen t_{A1} - t_{An}. Die Transaktionen des Verteilers sind somit gegeben durch:

$$\begin{aligned} T_I &= \{t_{Ii} | 1 \leq i \leq n\} \\ T_K &= \{t_{Ki} | 0 \leq i \leq n\} \end{aligned}$$

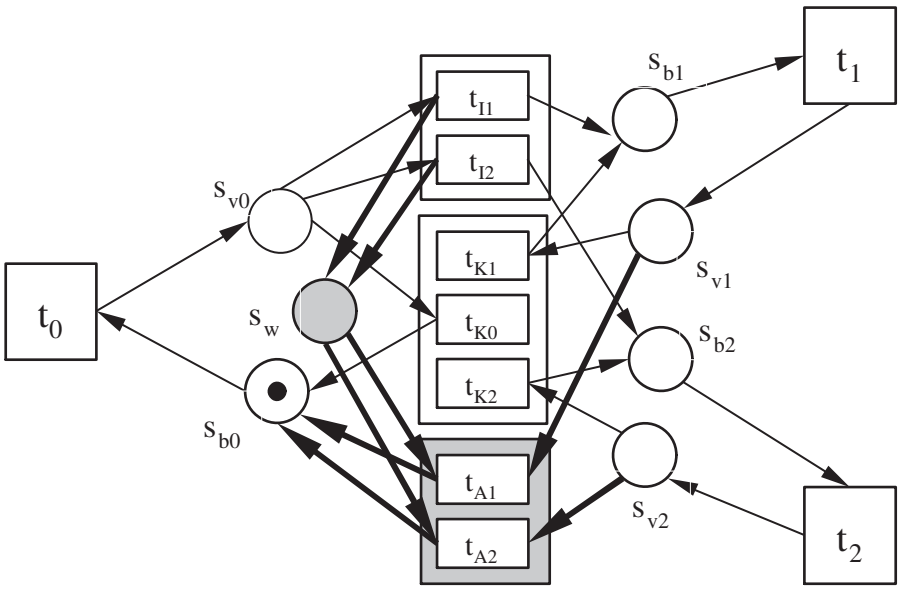


Abbildung 4.5: Beantwortung durch die Bearbeiter

$$\begin{aligned} T_D &= \{t_{Di,j} | 1 \leq i \leq n, 1 \leq j \leq n, i \neq j\} \\ T_A &= \{t_{Ai} | 1 \leq i \leq n\} \end{aligned}$$

Als nächstes benötigt man für jeden Bearbeiter, sowie für den Aufgabensteller zwei Stellen s_v und s_b. Dabei bedeutet eine Markierung einer Stelle s_{vi}, daß eine Konversation zwischen dem Verteiler und dem Bearbeiter i läuft und der Verteiler an der Reihe ist, einen Konversationsschritt durchzuführen. Dementsprechend bedeutet eine Markierung einer Stelle s_{bi}, daß der Bearbeiter i an der Reihe ist, eine Aktion durchzuführen. Hierbei sind die Stellen s_{v0} und s_{b0} zuständig für die Konversation zwischen Aufgabensteller und Verteiler. Weiterhin wird eine Stelle s_w eingefügt, welche aussagt, ob der Aufgabensteller momentan auf die Lösung der Aufgabe durch die Bearbeiter wartet. Die notwendigen Stellen sind somit gegeben durch:

$$S = \{s_{vi} | 0 \leq i \leq n\} \cup \{s_{bi} | 0 \leq i \leq n\} \cup \{s_w\}$$

Den komplexeren Teil bei der Konstruktion bildet die Flußrelation. F_{AB} beschreibt zusammen mit F_M die Konversation zwischen dem Aufgabensteller und dem Verteiler, sowie die Konversationen der einzelnen Bearbeiter mit dem Verteiler. F_I beschreibt die Initiierung. Dabei wird eine Konversation mit einem Bearbeiter gestartet und die Konversation mit dem Aufgabensteller wird in eine Warteposition versetzt (s_w). F_D beschreibt das Schalten einer Delegationstransaktion und durch F_{Dj} wird eine Delegation von einem Bearbeiter zu allen anderen Bearbeitern außer sich selbst ermöglicht. F_A schließlich ermöglicht eine Rückführung einer Aufgabe zum Aufgabensteller, wobei dieser auch aus der Warteposition s_w abgeholt wird.

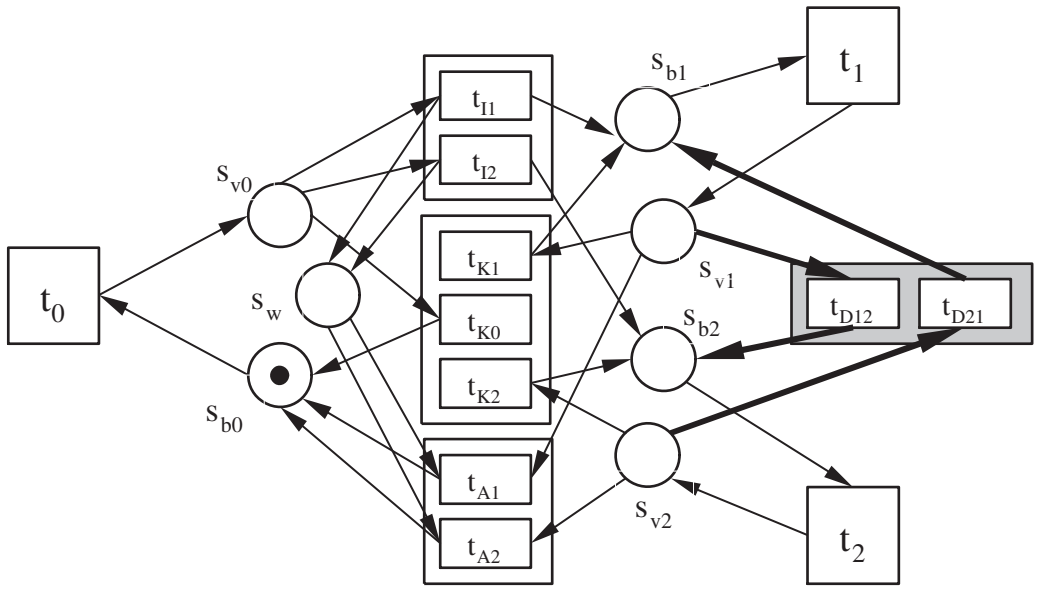


Abbildung 4.6: Delegation von Konversationen

$$
\begin{aligned}
F_{AB} &= \{(t_i, s_{vi}) | 0 \leq i \leq n\} \cup \{(s_{bi}, t_i) | 0 \leq i \leq n\} \\
F_I &= \{(s_{v0}, t_{Ii}) | 1 \leq i \leq n\} \cup \{(t_{Ii}, s_w) | 1 \leq i \leq n\} \cup \{(t_{Ii}, s_{bi}) | 1 \leq i \leq n\} \\
F_M &= \{(s_{vi}, t_{Ki}) | 0 \leq i \leq n\} \cup \{(t_{Ki}, s_{bi}) | 0 \leq i \leq n\} \\
F_{DB} &= \{(t_{Dj,i}, s_{bi}) | 1 \leq i \leq n, 1 \leq j \leq n, i \neq j\} \\
F_{VD} &= \{(s_{vj}, t_{Dj,i}) | 1 \leq i \leq n, 1 \leq j \leq n, i \neq j\} \\
F_A &= \{(s_w, t_{Ai}) | 1 \leq i \leq n\} \cup \{(s_{vi}, t_{Ai}) | 1 \leq i \leq n\} \cup \{(t_{Ai}, s_{b0}) | 1 \leq i \leq n\}
\end{aligned}
$$

Das gesamte Systemnetz SN = (S,T,F) des Workflowsystems wird somit beschrieben durch:

$$
\begin{aligned}
S &= \{s_{vi} | 0 \leq i \leq n\} \cup \{s_{bi} | 0 \leq i \leq n\} \cup \{s_w\} \\
T &= T_{AB} \cup T_I \cup T_K \cup T_D \cup T_J \\
F &= F_{AB} \cup F_I \cup F_M \cup F_{VD} \cup F_{DB} \cup F_A
\end{aligned}
$$

Wenn man davon ausgeht, daß der Aufgabensteller mit dem ersten Schritt die Aufgabe formuliert und somit den Arbeitsprozeß beginnt, ergibt sich als Anfangsmarkierung:

$$M_0 = \{s_{bo}\}$$

4.3 Steuerung mit Objektnetzen

Um das in den vorigen Abschnitten beschriebene Systemnetz zu steuern, werden die Arbeitsprozesse, die im Systemnetz laufen sollen, durch Objektnetze beschrieben. Diese entsprechen den Laufzetteln in ablauforientierten Prozeßbeschreibungen. Als erstes wird der Ablauf des Arbeitsprozesses analysiert und durch ein ENS dargestellt. Jeder Schritt in diesem Arbeitsprozeß entspricht dabei einer Transaktion des Objektnetzes. Die Zuordnung, wer für den jeweiligen Arbeitsschritt zuständig ist, geschieht mit der Interaktionsrelation, durch die jede Transaktion $e \in E$ einem Bearbeiter (oder dem Aufgabensteller) t_{0-n} zugewiesen ist. Dadurch ergibt sich

$$(t_j, e_i) \in \rho \Leftrightarrow \begin{cases} \textit{Schritt i wird vom Aufgabensteller aufgeführt} & : \quad j = 0 \\ \textit{Schritt i wird von Bearbeiter j aufgeführt} & : \quad j > 0 \end{cases}$$

Abbildung 4.7 zeigt einen Arbeitsprozeß, bei dem Schritt 0 und 3 durch den Aufgabensteller ausgeführt werden und Schritt 1 und 2 jeweils durch Bearbeiter 1 und 2. Zur vereinfachten Darstellung werden die Elemente der Interaktionsrelation benannt und in eckigen Klammern an die Transitionen geschrieben. Die Beschriftung `<ACTn>` in Abbildung 4.7 an einer Transition e_m bedeutet hierbei, daß $(t_n, e_m) \in \rho$.

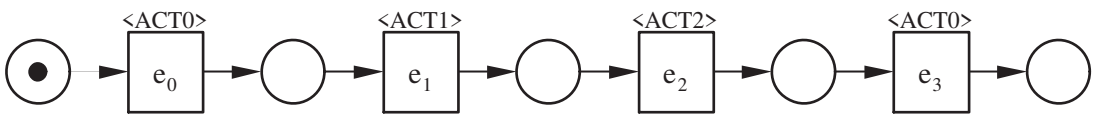


Abbildung 4.7: Arbeitsprozeß als ENS

Um das so entstandene ENS als Objektnetz in dem in Abschnitt 4.2.5 erstellten Systemnetz verwenden zu können, muß das Netz um die Aufgaben des Verteilers erweitert werden. Zwischen zwei Arbeitsschritten des Prozesses (durch Stellen verbundene Transitionen e_a und e_b des ON) wird eine neue Transition eingefügt. Werden die beiden getrennten Arbeitsschritte von demselben Bearbeiter/Aufgabensteller erledigt, so wird eine Transition eingefügt, die mit der entsprechenden Transition aus T_K interagiert. Wird die erste Transition vom Aufgabensteller ($(t_0, e_a) \in \rho$) erfüllt und die zweite durch einen Bearbeiter ($(t_i, e_b) \in \rho \Rightarrow i > 0$), so wird eine Transition eingefügt, die mit einer Transition aus T_I interagiert. Wenn es sich um Arbeitsschritte zweier Bearbeiter handelt, wird eine *Delegation* durchgeführt. Bei einer Aktion eines Bearbeiters gefolgt vom Aufgabensteller wird mit einer *Antwortstransaktion* verbunden. In Analogie zu der Zuordnung zum Aufgabensteller und zu den Bearbeitern durch die Beschriftung `<ACTm>`, werden die Aufgaben des Verteilers durch die folgenden Beschriftungen beschrieben:

- `<INITn>` bedeutet, daß eine Konversation mit Bearbeiter m begonnen wird.
- `<DELEGn>` bedeutet, daß die Aufgabe vom aktiven Bearbeiter zu Bearbeiter n delegiert wird.

- `<ANTW>` bedeutet, daß der aktive Bearbeiter die Aufgabe an den Aufgabensteller zurückgibt. Dies kann im Falle einer Lösung der Aufgaben, aber auch im Falle einer Rückfrage erfolgen.
- `<KONV>` kennzeichnet einen weiteren Iterationsschritt in einer Konversation des Verteilers mit einem Bearbeiter oder dem Aufgabensteller.

Durch diese Erweiterungen ergibt sich ein Objektnetz, wie es in Abbildung 4.8 dargestellt wird.

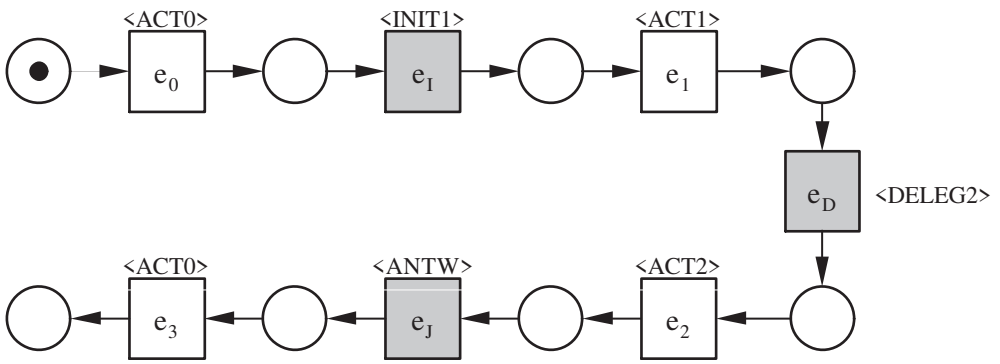


Abbildung 4.8: Erweiterter Arbeitsprozeß

Um dieses Objektnetz durch das Systemnetz zu transportieren, wird die eben eingeführte Notation für die Interaktionsrelation übernommen und im Systemnetz ebenfalls verwendet. Das vollständige Systemnetz ist in Abbildung 4.9 zu sehen.

4.4 Einzelkonversationen

Wie in Kapitel 3 beschrieben, besteht der durch das Objektnetz dargestellte Arbeitsprozeß aus mehreren Einzelkonversationen, welche in einem Konversationskontext zusammengefaßt werden. Um im Objektnetz die Struktur dieser Einzelkonversationen zu erkennen, kann man Vergröberungen des Objektnetzes erstellen, die auf die Arbeitsschritte eines einzelnen Bearbeiters oder des Aufgabenstellers reduziert sind. Die Zerlegung des Objektnetzes erfolgt an den Transaktionen des Verteilers. Eine Konversation mit einem Bearbeiter m beginnt mit einer Initiierung (`<INITm>`) oder einer Delegation (`<DELEGm>`) und endet mit einer Beantwortung (`<ANTW>`) oder einer Delegation (`<DELEGn>`). Die Konversation mit dem Aufgabensteller kann durch eine Initiierung unterbrochen werden, wird jedoch durch eine Antwort (`<ANTW>`) wieder fortgesetzt. Abbildung 4.10 zeigt die Vergröberungen für das Objektnetz aus Abbildung 4.8.

Durch die Identifikation der Einzelkonversationen lassen sich Konversationsspezifikationen bestimmen, die für die Modellierung der Business Conversations benötigt werden. Weiterhin sind diese Einzelkonversationen ein Analogon zu den benutzerdefinierten Konversationen, wie sie in Abschnitt 6.2 vorgestellt werden.

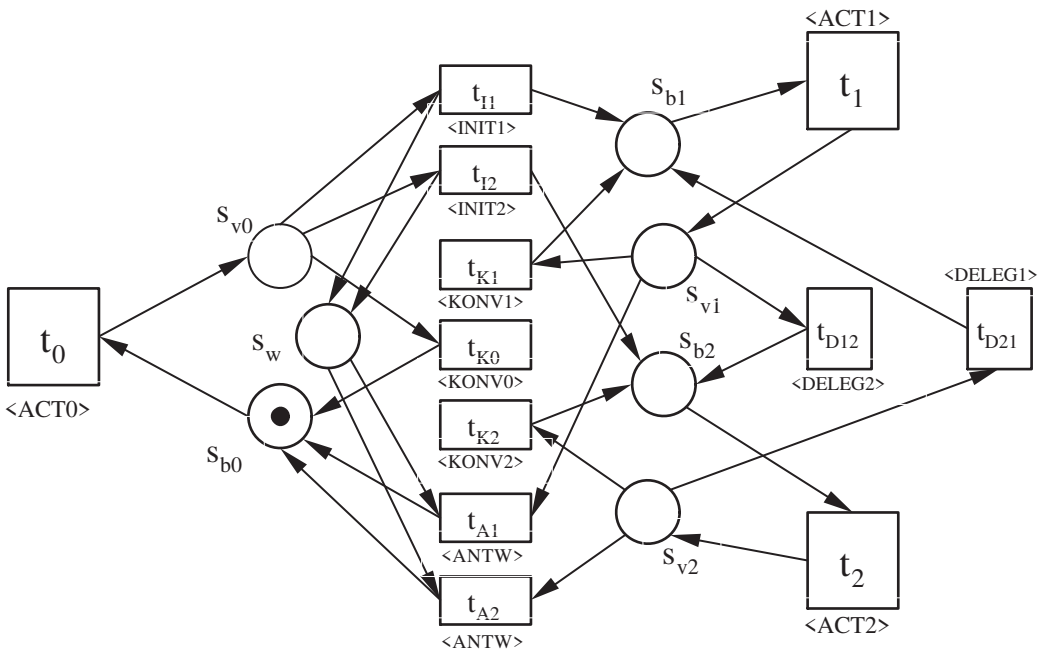


Abbildung 4.9: Vollständiges Systemnetz mit Beschriftung entsprechend der Interaktionsrelation

4.5 Erweiterungen des Modells und Bewertung

Das in den vorherigen Abschnitten vorgestellte Modell dient dazu, Konzepte des Workflowsystems vorzustellen, die später implementiert werden. Aus Gründen der besseren Verständlichkeit wurde das Modell teilweise eingeschränkt. Es ist hier nicht vorgesehen, eine neue Konversation durch einen Bearbeiter zu initiieren. Solche Fälle sind aber wünschenswert, wenn zum Beispiel eine Aufgabe in Unteraufgaben zerlegt werden soll, oder ein Bearbeiter zur Lösung der Aufgabe einen anderen Bearbeiter mit der Lösung eines Teilaspektes beauftragt und auf dessen Lösung wartet. In diesem Fall übernimmt der Bearbeiter die Rolle eines Aufgabenstellers. Diese Fälle werden in der Implementierung berücksichtigt, jedoch im Modell nicht vorgesehen, da dies zu einer unnötigen Verkomplizierung führen würde. In die Modellierung ebenfalls nicht eingeflossen sind die *Inhalte* der Konversationen, wie sie in den *Business Conversations* verwendet werden. Hier kann die Verwendung von gefärbten Petrinetzen das Modell erweitern. Nebenläufigkeiten werden von dem Modell ebenfalls unterstützt, sind in den gezeigten Beispielen aber nicht aufgetreten. Informationen zum Verhalten von EOS bei Nebenläufigkeit werden in [Valk 98a] beschrieben.

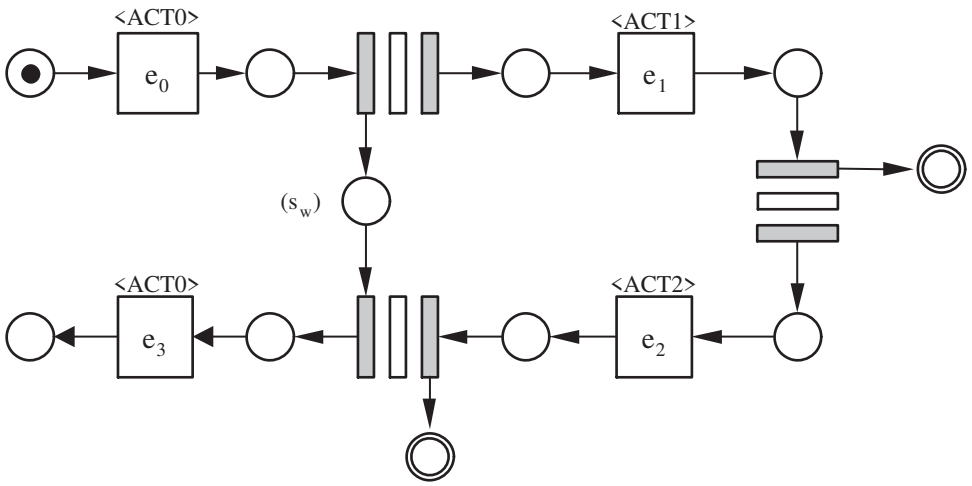


Abbildung 4.10: Vergröberungen und Aufteilung des Objektnetzes

Kapitel 5

Entwicklungsumgebung

Zur praktischen Umsetzung des bisher theoretisch beschriebenen Workflowsystems wird ein dreiteiliges System verwendet. Ein Microsoft Transaction Server (MTS) ist für die Verwaltung der Komponenten sowie für das Transaktionsmanagement innerhalb des Workflowsystems zuständig. Der MTS ist außerdem Bindeglied zwischen dem verwendeten SQL-Server, welcher die zweite Systemkomponente bildet und dem Webserver, der das Frontend des Systems darstellt. In diesem Kapitel werden diese Komponenten näher beschrieben. Dies dient dem tieferen Verständnis der eingesetzten Technologien, da hierdurch bestimmte Vorgaben für das Design und die Konstruktion des Workflowsystems entstehen.

5.1 Microsoft Transaction Server

Der Microsoft Transaction Server (MTS) gehört zur Gruppe der *Applikationsserver*. Applikationsserver stellen Funktionalität mittels Komponenten zur Verfügung, welche von Clients aus genutzt werden können. Die Ausführung der Komponentenmethoden erfolgt auf dem Server. Im Falle des MTS, handelt es sich bei diesen Komponenten um *ActiveX-Komponenten*. Weitere Beispiele für Applikationsserver sind Bea Weblogic und IBM WebSphere, welche *Enterprise Java Beans* als Komponenten verwenden.

Der MTS wird verwendet für die Entwicklung und Verwaltung von verteilten Anwendungen [Träger 98]. Besonders die Entwicklung von Intra- und Internetanwendungen wird unterstützt, da es eine enge Verzahnung mit dem Microsoft Internet Information Server (IIS) gibt.

Das Anwendungsmodell des MTS beruht auf Komponenten, die durch diesen verwaltet werden, wodurch sie auch der Transaktionskontrolle des MTS unterstellt sind. Die Nutzung dieser Komponenten kann auf zwei Arten erfolgen. Entweder können sie direkt durch Win32-Clients angesprochen werden, wobei die Kommunikation über COM [Mic 95] realisiert wird. Oder die Komponenten werden in ASP-Skripten über den IIS benutzt (siehe Abbildung 5.1). Diese Methode wird in dieser Arbeit eingesetzt werden, weswegen sie im folgenden auch alleiniger Bestandteil der Betrachtung sein wird. Die Verwendung von Komponenten in der Umgebung des MTS aus ASP-Skripten heraus wird in Abschnitt 5.3.1.2 beschrieben. Die Komponenten werden in Paketen gruppiert. Das bietet

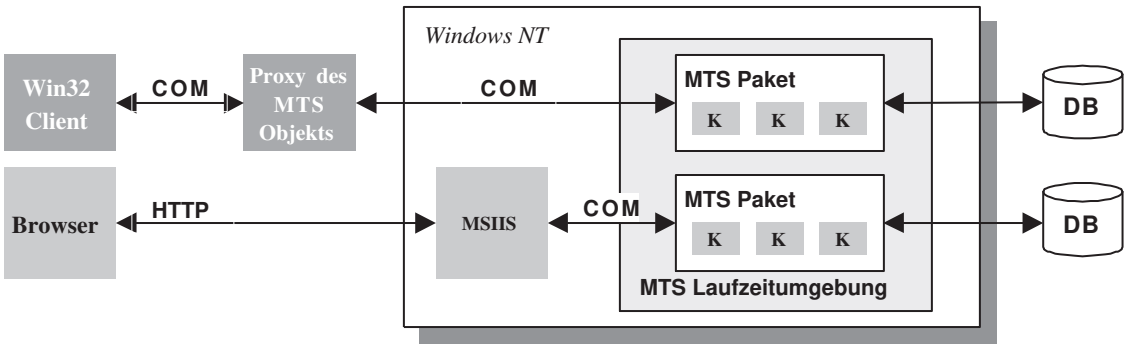


Abbildung 5.1: Architektur des Microsoft Transaction Server

die Möglichkeit Komponenten einer Applikation in einem Paket zusammenzufassen und diese in einem separaten Server-Prozeß laufen zu lassen. Durch diese Isolierung wird vermieden, daß sich Komponenten verschiedener Applikationen beeinflussen können. Ein Absturz einer Komponente einer Applikation beeinträchtigt dadurch nicht die Ausführung der Komponenten einer anderen Applikation. Außerdem können die Zugriffsrechte und Rollen pro Paket definiert werden. Bei Methodenaufrufen über Paketgrenzen hinweg, werden diese Zugriffsrechte überprüft, während Methodenaufrufe zwischen Komponenten desselben Paketes nicht der Sicherheitsüberprüfung unterliegen. Komponenten innerhalb eines Pakets müssen sich daher vertrauen.

Anwendungen, die auf der Benutzung von einer oder mehreren MTS-Komponenten beruhen, werden auch als *MTS-Applikationen* bezeichnet. Die Komponenten einer solchen MTS-Applikation laufen innerhalb der *MTS-Laufzeitumgebung*, welche auch für die Verwaltung der Systemressourcen zuständig ist. Dies beinhaltet die Aktivierung bzw. Deaktivierung der Komponenten, die Einleitung und Überwachung von Transaktionen, sowie den Zugriffsschutz von Komponenten.

5.1.1 Aktivierung von Komponenten

Bei der Verwendung von Komponenten stehen zwei traditionelle Methoden zur Verfügung, die sich in Bezug auf Zustandserhalt und Lebensdauer unterscheiden.

- Bei der ersten Methode erzeugt der Client ein Objekt, kann dies benutzen und danach wieder entfernen. Der Zustand des Objekts wird nur während eines Methodenaufrufs erhalten. Daher wird diese Art von Objekten als *zustandslos* bezeichnet. Der Vorteil liegt in der guten Ausnutzung von Ressourcen, da diese nur verwendet werden, solange das Objekt eine Methode ausführt und danach wieder freigegeben werden können. Ein Nachteil ist der Aufwand, bei jedem Aufruf ein Objekt erneut erzeugen zu müssen.

- Die zweite Variante bietet die Möglichkeit, den Zustand des Objektes zu erhalten und so Daten über mehrere Methodenaufrufe hinweg zu erhalten. Dies erspart das

häufige Erzeugen von Objekten und kann auch die Zahl der zu übertragenden Funktionsparameter reduzieren. Allerdings bleiben Ressourcen auf dem Server während der gesamten Lebenszeit des Objektes belegt und können nicht anderweitig genutzt werden. Dies schränkt die Skalierbarkeit des Systems ein.

Bei MTS-Komponenten findet eine Mischform beider Varianten Verwendung. Das Konzept wird als *Just-in-time-Aktivierung* [Reed et al. 97] bezeichnet. Komponenten sind zustandslos, genau wie in der ersten, bereits beschriebenen Variante. Nach einem Methodenaufruf wird das Objekt aber nicht entfernt, sondern die Referenz behält ihre Gültigkeit. Diese Referenz verweist jedoch nur noch auf eine Kontexthülle, wodurch dem Client vorgespiegelt wird, das Objekt sei noch vorhanden. Intern wird es jedoch entfernt und damit werden auch alle vom Objekt beanspruchten Ressourcen inklusive des Zustands freigegeben. Abbildung 5.2 zeigt den indirekten Methodenaufruf über die Kontexthülle des Objektes, welche vom MTS automatisch für jedes Objekt erzeugt wird.

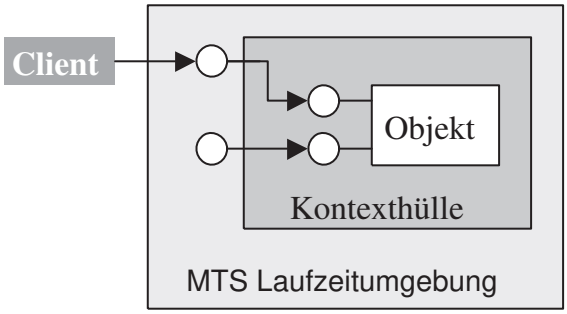


Abbildung 5.2: Indirekter Zugriff auf Objekte über die Kontexthülle

Wenn der Client mittels der noch vorhandenen Referenz eine weitere Methode des Objektes aufruft, wird ein neues Objekt innerhalb der Kontexthülle erzeugt. Dies geschieht jedoch für den Client vollständig transparent und es muß keine explizite Generierung von Objekten seitens des Clients erfolgen. Das reduziert die Netzwerklast, welches eine der positiven Eigenschaften der zweiten beschriebenen Variante ist. Die Kontexthülle bleibt solange erhalten, solange Referenzen hierauf existieren.

Bei der Entwicklung von MTS-Anwendungen ist das zustandslose Verhalten der Komponenten entscheidend, da Attribute eines Objektes nicht über zwei Methodenaufrufe hinweg erhalten bleiben. Genauer gesagt, ist der Erhalt des Zustandes an die Dauer der Transaktion gekoppelt, der eine Komponente angehört. Solange diese nicht abgeschlossen wird, bleibt auch der Zustand erhalten. Dieser Zusammenhang sollte aber nicht verwendet werden, um künstlich den Zustand zu erhalten, da lang andauernde Transaktionen viele Ressourcen beanspruchen und sogar die Entstehung von Verklemmungen begünstigen [Lockemann, Schmidt 93].

5.1.2 Transaktionskontrolle

MTS-Komponenten werden automatisch mit einem Transaktionskontext versehen [Pattison 98]. Dieser Kontext ist über ein Kontextobjekt von der Komponente aus erreichbar. Das Kontextobjekt stellt hauptsächlich vier Funktionen zur Verfügung. Zwei Funktionen, mit denen Transaktionen beendet (*setComplete*) oder abgebrochen (*setAbort*) werden können und zwei Funktionen, die den aktuellen Zustand des Objektes als konsistent bzw. inkonsistent darstellen (*enableCommit* und *disableCommit*). Ein Aufruf von *setAbort* oder *setComplete* in einer Methode, beendet die Transaktion nach Beendigung dieser Methode. Das Objekt wird innerhalb der Kontexthülle (s. Abschnitt 5.1.1) entsorgt.

Die Transaktionen, die der MTS verwaltet, sind jedoch nicht auf die Laufzeitumgebung des MTS beschränkt. Vielmehr wird das Konzept der verteilten Transaktionen unterstützt, wodurch sich der Wirkungskreis auf mehrere Ressourcengeber erstrecken kann. Besonders interessant ist hierbei die Verwendung der Transaktionslogik von Datenbanken. Zur Koordination solcher verteilten Transaktionen verwendet der MTS den Microsoft Distributed Transaction Coordinator (MSDTC). Dieser fungiert als Bindeglied zwischen den Transaktionsmanagern des MTS und des jeweiligen *Datenbanksystems (DBS)*. Voraussetzung hierfür ist natürlich, daß das DBS eine Transaktionskontrolle mittels des MSDTC unterstützt.

Datenbankzugriffe aus einer MTS-Komponente heraus werden automatisch und transparent der Transaktion der MTS-Komponente zugeordnet. Wird diese Transaktion abgebrochen, so wird auch die Transaktion auf dem DBS abgebrochen. Aber auch bei der Benutzung von MTS-Komponenten durch andere MTS-Komponenten können die Transaktionen miteinander gekoppelt werden. Wie dies geschieht, kann für die MTS-Komponenten individuell eingestellt werden. Zum einen kann eine MTS-Komponente eine neue Transaktion beginnen, unabhängig davon, ob sie von einer MTS-Komponente verwendet wird, die bereits in einer Transaktion läuft. Dadurch wird die Komponente unabhängig und ein Abbruch der Transaktion der übergeordneten Komponenten hat keinen Einfluß auf die eigene Transaktion. Die zweite Variante sieht vor, daß bei einem Aufruf durch eine Komponente, welche bereits eine Transaktion gestartet hat, die neue Komponente ebenfalls in diesem Transaktionskontext läuft. Ein Abbruch der Transaktion der übergeordneten Komponente hat in diesem Fall auch einen Abbruch der eigenen Transaktion zur Folge.

5.1.3 Zugriffsschutz von Komponenten

Der MTS definiert ein deklaratives Sicherheitsmodell, welches über ein Rollenkonzept realisiert wird [Chappell 98]. Benutzer oder Benutzergruppen können einer Rolle zugeordnet werden. Der Zugriff auf Komponenten kann nun auf Mitglieder bestimmter Rollen beschränkt werden. Dies reicht unter Umständen aber nicht aus, da es zum Beispiel möglich sein kann, daß eine Komponente von Benutzern mehrerer Rollen benutzt werden kann, die Funktionalität jedoch in Abhängigkeit von der Rolle eingeschränkt ist. Nehmen wir an, eine Komponente bietet Zugriff auf eine Datenbank via SQL. Die Komponente darf von den Rollen *Administrator* und *Bearbeiter* verwendet werden. Allerdings soll sich der Umfang der erlaubten SQL-Befehle unterscheiden. Während ein Administrator den vollen Sprachumfang von SQL nutzen darf, soll einem Bearbeiter nur die Möglichkeit gegeben werden,

Daten aus der Datenbank auszulesen. Die Entscheidung, ob eine Methode der Komponente ausgeführt werden darf, kann also erst zur Laufzeit, in Abhängigkeit vom übergebenen SQL-Befehl, getroffen werden. Daher stellt der MTS Funktionen zur Verfügung, mittels derer Komponenten explizit prüfen können ob der Benutzer einer bestimmten Rolle angehört. Bei der Implementierung des Workflowsystems in Kapitel 6 konnte dieses Sicherheitskonzept aber nicht angewandt werden. Da der Webserver als Frontend verwendet wurde und alle Anfragen an den MTS aus einem ASP-Skript heraus erfolgen, tritt stets der Webserver als Benutzer der Komponenten auf. Eine Unterscheidungsmöglichkeit, durch welchen Benutzer auf die Komponente zugegriffen wurde, steht daher nicht mehr zur Verfügung. Abbildung 5.3 zeigt diesen Verlust der Identifikation.

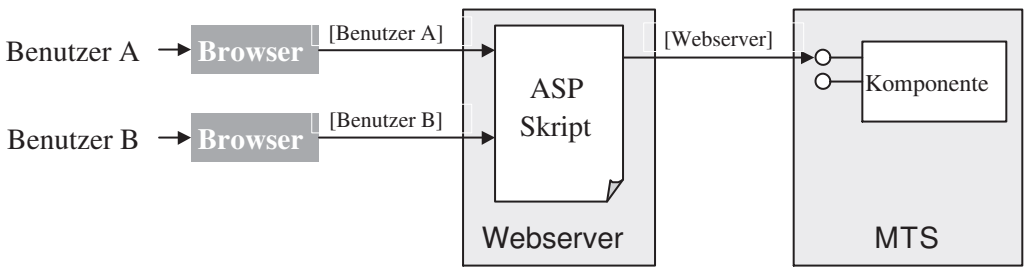


Abbildung 5.3: Benutzersicht durch den MTS

5.2 Datenbankserver

Als Datenbankserver wird für diese Arbeit der Microsoft SQL Server (MSSQL) [Petkovic 96] verwendet. Dieser ist aber nicht zwingend notwendig und kann auch durch einen anderen SQL-Server ersetzt werden. Bedingung ist lediglich, daß der SQL-Server das Transaktionsmanagement in Verbindung mit dem MTS unterstützt. Die Koordination der beiden Server erfolgt hierbei durch den Microsoft Distributed Transaction Manager (MSDTC). Zu dem Zeitpunkt, zu dem diese Arbeit entstand, kam neben dem MSSQL ab der Version 6.5 auch der Datenbankserver Oracle 8 in Betracht. Die Interaktion mit dem Datenbankserver erfolgt ausschließlich durch SQL. Hierbei muß auf Unterschiede verschiedener SQL-Dialekte geachtet werden.

5.3 Webserver

Als Webserver fungiert der Microsoft Internet Information Server 4 (IIS). Dieser ist der einzige Webserver, der momentan die microsofteigene Skriptumgebung der *Active Server Pages (ASP)* [Walther 98] unterstützt und kann daher nicht durch einen anderen Server ersetzt werden. Mit Hilfe von ASPs kann man sowohl einfache dynamische Webseiten, als auch komplexe ASP-Applikationen erstellen. Die Erstellung von Applikationen wird durch

ein integriertes Session- und Applikationsmanagement unterstützt, und es ist möglich, ActiveX-Komponenten aus den ASPs heraus zu verwenden.

5.3.1 Active Server Pages

Eine einzelne ASP kann HTML und vom Webserver auszuführende Skripte enthalten. Weiterhin können ActiveX-Komponenten [Appleman 99] aus ASPs heraus verwendet werden. Als Skriptsprachen stehen u.a. *VBScript*, *JScript* und *PERLScript* zur Verfügung [Bortniker et al. 98]. Folgende Beispiele beziehen sich hierbei immer auf VBScript, da ausschließlich diese Skriptsprache in dieser Arbeit Anwendung fand. Die Schnittstelle zum Webserver bilden Server-Objekte, die in jedem ASP-Skript automatisch zur Verfügung stehen. Diese werden im nächsten Abschnitt beschrieben. Abschnitt 5.3.1.2 demonstriert die Verwendung von MTS-Komponenten aus ASP-Skripten.

5.3.1.1 Server Objekte

Das Objekt *Request* stellt Funktionen zur Verfügung, durch die man auf die *Servervariablen* zugreifen kann. Dazu gehören auch die Daten, die von einem HTML-Formular übertragen wurden, bzw. die in der URL enthalten waren. Das Objekt *Response* dient hauptsächlich dazu, Daten in den Ausgabestrom zu schreiben. Hierdurch kann dynamischer Webseiteninhalt erstellt werden. Das folgende Beispiel zeigt eine ASP, die im HTML-Teil ein Formular mit zwei Feldern enthält. Dieses Formular ruft sich rekursiv auf. Der Skriptteil dieser ASP liest die Formularfelder aus und gibt diese mit Hilfe des *Response-Objektes* in einer Tabelle aus.

```
<html>
<body>
<form action=bsp1.asp method=post>
  <table>
  <tr>
    <td>Name</td>
    <td>
      <input type=text name=Name>
    </td>
  <tr>
    <td>Vorname</td>
    <td>
      <input type=text name=Vorname>
    </td>
  <tr>
    <td></td>
    <td>
      <input type=submit value=Abschicken>
    </td>
```

```
  </table>
</form>

<hr>
<%
  name = Request.Form("Name")
  vorname = Request.Form("Vorname")

  Response.Write "<table border><tr><th>Name</th><th>Vorname</th></tr>"
  Response.Write "<tr><td>" & name & "</td><td>" & vorname & \
    "</td></tr></table>"
%>
</body>
</html>
```

Neben den Objekten *Request* und *Response* gibt es noch die Objekte *Application* und *Session*, die in Abschnitt 5.3.1.3 gesondert beschrieben werden. Das Objekt *ObjectContext* dient der Einbindung eines ASP-Skriptes in eine Transaktion. Diese Transaktion wird dann vom MTS (siehe Abschnitt 5.1) verwaltet. Das Objekt *Server* stellt Funktionen zur Instantiierung von Komponenten (siehe Abschnitt 5.3.1.2), sowie zur String-Konvertierung zur Verfügung. Genauere Informationen zu den Standardobjekten sind in der Dokumentation des IIS zu finden.

5.3.1.2 Verwendung von MTS-Komponenten

Die Verwendung von MTS-Komponenten geschieht mit Hilfe des Objektes *Server*. Durch die Methode *Server.CreateObject* wird ein Objekt erzeugt und eine Referenz zurückgeliefert. Dabei erfolgt die Identifikation der Objektklasse über den Namen und wird als Zeichenkette an *Server.CreateObject* übergeben. Dadurch kann die Entscheidung, welche Objektklasse instantiiert werden soll, bis zur Laufzeit verzögert werden. Dies birgt natürlich Gefahren, da es ebenfalls erst zur Laufzeit feststellbar ist, ob die angeforderte Objektklasse überhaupt existiert. Diese Form des *Polymorphismus* findet in der Implementierung Verwendung. Dadurch kann eine dynamische Struktur erzeugt werden, die für die Erstellung von benutzerdefinierten Arbeitsprozessen notwendig ist. Diese werden in Abschnitt 6.2 beschrieben. Das folgende Beispiel demonstriert die Verwendung einer Datenbankkomponente. Zuerst wird ein Objekt erzeugt, das eine Methode zur Verfügung stellte, um einen SQL-Befehl an einen Datenbankserver weiterzuleiten.

```
<html>
<body>
<%
  dbaccess = Server.CreateObject("MTSDEMO.DBACCESS")
  dbaccess.execute("DELETE FROM user WHERE id=5")
  Response.Write "Der SQL-Befehl wurde übertragen."
%>
```

```
</body>
</html>
```

5.3.1.3 Session- und Applikationsmanagement

Eine Sammlung von logisch zusammenhängenden ASPs bezeichnet man als eine ASP Applikation [Fedorchek, Rensin 97]. Diese werden durch Unterverzeichnisse im Verzeichnisbaum des Webservers definiert und abgegrenzt. Wird ein Knoten im Verzeichnisbaum einer Applikation zugewiesen, so sind alle ASPs die sich in diesem Verzeichnis befinden, sowie alle ASPs, die sich in Unterverzeichnissen befinden, Bestandteil dieser ASP-Applikation. Innerhalb von Applikationen ist es möglich, Daten zu speichern, die von allen ASPs dieser Applikation gelesen und geschrieben werden können. Die Speicherung erfolgt im Objekt *Application*. Um den nebenläufigen Zugriff auf die globalen Variablen einer Applikation zu serialisieren, kann die Applikation exklusiv gesperrt werden, um die typischen Probleme, die beim nebenläufigen Zugriff auf Variablen auftreten können, zu vermeiden [Coulouris et al. 94]. Das folgende Beispiel zeigt einen einfachen Zähler, der bei jedem Aufruf der Seite erhöht wird. Gleichzeitig wird der aktuelle Zählerstand ausgegeben.

```
<html>
<body>
<%
  Application.Lock

  Response.Write "Es sind bisher " & Application("Counter") &  \
    " Zugriffe erfolgt."

  Application("Counter") = Application("Counter") + 1
  Application.Unlock
%>
</body>
</html>
```

Neben dem Austausch von globalen Daten zwischen verschiedenen ASPs einer Applikation ist es weiterhin wünschenswert, Informationen über mehrere Zugriffe eines Browser auf verschiedene ASPs hinweg zu erhalten. Da HTTP ein zustandsloses Protokoll ist, behandelt der Webserver jeden Zugriff als unabhängigen Aufruf und es besteht kein Unterschied, ob zwei Webseiten oder ASPs von demselben Browser aus aufgerufen werden oder von zwei verschiedenen. Für verbreitete Web-Applikationen, wie zum Beispiel einen Internetshop, sollen Daten aber über mehrere Zugriffe erhalten bleiben und auch einem Browser eindeutig zugeordnet werden. Bei dem Internetshop möchte man zum Beispiel einen Warenkorb realisieren, in den der Kunde Artikel, die sich auf verschiedenen Webseiten befinden, legen kann.

Das Aufrufen mehrerer Webseiten oder ASPs durch einen Benutzer wird als *Session* bezeichnet. Diese beginnt, wenn ein Benutzer eine ASP einer ASP-Applikation zum er-

stenmal aufruft. Diese bleibt solange aktiv, wie der Benutzer eine zu dieser Applikation zugehörige ASP aufruft und bevor eine festzulegende Zeitspanne (*timeout*) abgelaufen ist.

Genauso, wie man mit Hilfe des Objektes *Application* Daten speichern kann, kann man das Objekt *Session* benutzen, um Daten einer Session zu speichern. Um Daten zwischen mehreren Aufrufen zu erhalten, mußte man sich bisher der Verwendung von versteckten Feldern in HTML-Formularen bedienen. Dies hat aber den Nachteil, daß man bei der Navigation zwischen den Webseiten auf die Verwendung von Formular-Knöpfen beschränkt ist, da diese versteckten Daten auch jedesmal an den Server gesendet werden müssen. Wechselt man die Webseiten, indem man zum Beispiel manuell die URL angibt oder einem normal Link folgt, so gehen diese Informationen verloren. Bei der Verwendung des Session-Objektes bleiben die Daten auch in diesen Fällen erhalten.

Das folgende Beispiel soll den Unterschied zwischen Applikations- und Sessionvariablen verdeutlichen. Es ist eine Kombination aus den ersten beiden Beispielen. Der Name und Vorname aus dem Formular werden diesmal aber in Sessionvariablen gespeichert. Dadurch kann jeder Benutzer seinen eigenen Namen speichern; dennoch greift die ASP auf die gleiche Applikationsvariable *Counter* zu und erhöht diese bei jedem Zugriff. Dadurch werden die Zugriffe aller Benutzer und gleichzeitig der Name jedes einzelnen Benutzers gespeichert.

```
<html>
<body>
<form action=bsp3.asp method=post>
  <table>
  <tr>
    <td>Name
    <td>
      <input type=text name=Name>
  <tr>
    <td>Vorname
    <td>
      <input type=text name=Vorname>
  <tr>
    <td>
    <td>
      <input type=submit value=Abschicken>
  </table>
</form>

<hr>
<%
  Session("name") = Request.Form("Name")
  Session("vorname") = Request.Form("Vorname")

  Application.Lock
```

```
  Response.Write "Hallo " & Session("vorname") & " " & Session("name")  \
    & " !<br>"
  Response.Write "Es sind bisher " & Application("Counter") & " Zugriffe \
    erfolgt."

  Application("Counter") = Application("Counter") + 1
  Application.Unlock
%>
</body>
</html>
```

5.4 Extensible Markup Language (XML)

Eine große Schwäche von HTML ist es, daß keine Trennung von Daten und Visualisierung erfolgt. Beides wird in einer einzigen Datei gemischt und dann über das Inter-/Intranet vom Server zum Browser transportiert. Will man nun verschiedene Darstellungsformen erreichen, zum Beispiel um verschiedenen Benutzern unterschiedliche Ansichten des gleichen Datenbestandes zu ermöglichen, so ist dies nur über Umwege möglich (z.B. *Cascading Style Sheets*). Hierbei handelt es sich nicht um ein Problem, das nur durch die Verwendung von *Business Conversations* bzw. den damit verbundenen Dialogen entstanden ist. Vielmehr ist es ein Problem, das den Datenaustausch im Internet generell betrifft. Bei der Suche nach Informationen im Internet müssen Suchmaschinen immer ganze HTML-Seiten übertragen und parsen. Dabei geht viel Zeit verloren, da nicht ersichtlich ist, wo Daten aufhören und Formatierungen anfangen. Auch hier ist eine Trennung von Daten und Visualisierung wünschenswert, da sich die Suche damit auf den Datenteil beschränken könnte. Als mögliche Lösung bietet sich hier die Verwendung von XML an. XML ist ähnlich wie HTML eine Untermenge von SGML. Während jedoch HTML eine feste, nicht erweiterbare Syntax besitzt, ist die Syntax von XML erweiterbar. Die Idee bei der Entwicklung von XML war, die Teile von SGML auszusparen, die selten benutzt werden, was zu einer Reduktion des Sprachumfanges um über 90% führte [Walsh 98]. Da XML einen erweiterbaren Sprachumfang bietet, kann eine *Document Type Definition (DTD)* mitgeschickt werden, die es dem Parser erlaubt, das Dokument auf seine Korrektheit zu prüfen. Zur Steigerung der Performanz kann diese jedoch auch weggelassen werden, wodurch das Dokument in jedem Falle vom Parser als korrekt anerkannt wird.

Am Beispiel einer Anzeige von Sonderangeboten soll der Einsatz von XML verdeutlicht werden.

```
<Angebote>
   <Artikel>
      <Bezeichnung>
         <uk>milkchocolate</uk>
         <de>Milchschokolade</de>
      </Bezeichnung>
```

```
      <VK>
        1,99
      </VK>
      <EK>
        0,23
      </EK>
   </Artikel>
   <Artikel>
      <Bezeichnung>
         <de>Kaffee</de>
         <uk>coffee</uk>
      </Bezeichnung>
      <VK>
        5,99
      </VK>
      <EK>
        3,25
      </EK>
   </Artikel>
</Angebote>
```

Hierbei werden zwei Artikel als Sonderangebote ausgewiesen. Es wird jeweils eine deutsche und eine englische Bezeichnung angegeben, weiterhin ist der Einkaufs- sowie der Verkaufspreis enthalten. Eine Überschneidung von Bereichsgrenzen wie z.B. in

```
<Angebote>
   <Bezeichnung>
      <Artikel>
   </Bezeichnung>
      </Artikel>
</Angebote>
```

ist nicht erlaubt. Die Interpretation des obigen XML-Codes als Baum wird in Abbildung 5.4 gezeigt.

5.5 Extensible Style Language (XSL)

Um aus diesen Daten eine browsergerechte Darstellung zu erhalten, muß eine Umformung in HTML erfolgen. Hierzu werden Visualisierungsregeln angegeben, durch die geregelt wird, wie bestimmte XML-Elemente durch HTML repräsentiert werden sollen. Diese Regeln werden in XSL [Mic 98] formuliert. Ein *XSL-Prozessor* kann dann aus einer XML-Datei mit den Regeln aus einer XSL-Datei eine HTML-Datei erzeugen (Abb. 5.5)

XSL-Regeln bestehen immer aus zwei Teilen. Einmal aus der Angabe eines Musters und dann aus der zugehörigen Aktion. Ein XML-Baum wird durch den XSL-Prozessor

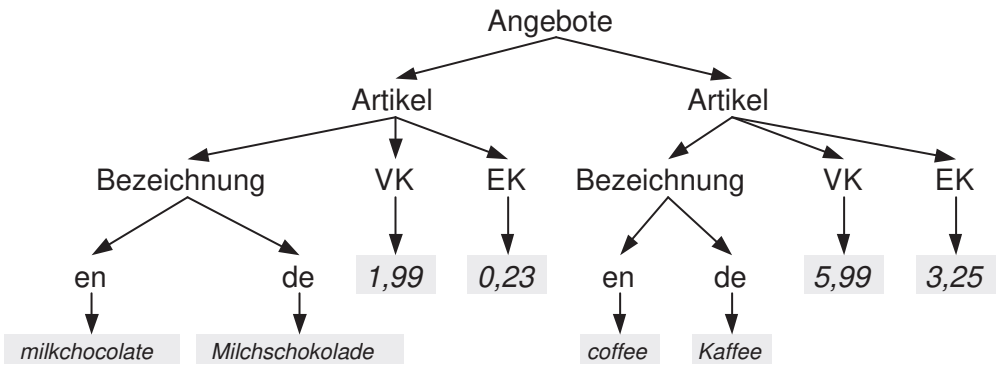


Abbildung 5.4: Baumdarstellung von in XML codierten Daten

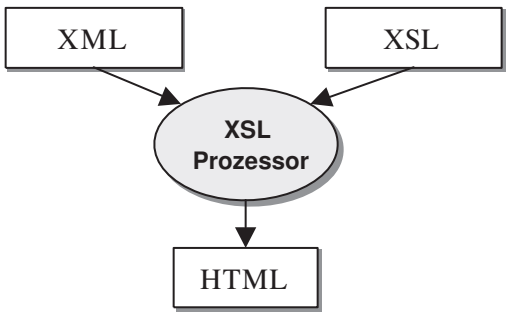


Abbildung 5.5: XSL-Prozessor

auf Vorkommnisse dieses Musters hin überprüft und danach kommt die zugehörige Aktion zur Anwendung. Bei den folgenden Beispielen dient stets der XML-Baum aus Abschnitt 5.4 als XML-Quelle. In diesem Beispiel wird etwa ein Element (`<EK>...</EK>`) eingeführt, welches die Angabe des Einkaufspreises ermöglicht. Eine Regel, durch die der Einkaufspreis immer fett und kursiv erscheint, würde zum Beispiel folgendermaßen aussehen:

```
<rule>
   <target-element type="EK"/>
   <b><i>
      <children/>
   </i></b>
</rule>
```

Dadurch würde eine Konvertierung von

```
<EK>
   "3.25"
</EK>
```

nach

```
<b><i>
   3.25
</i></b>
```

erfolgen. Bei der Formulierung von Regeln bietet XSL aber noch weitaus komplexere Möglichkeiten, auf die hier nicht näher eingegangen werden soll. Weitere Informationen sind unter [Bradley 98] zu finden. Um den praktischen Einsatz von XML und XSL zu verdeutlichen, wird das Beispiel mit den Sonderangeboten aus Abschnitt 5.4 aufgegriffen. Zum einen wird eine deutschsprachige Seite mit EK und VK erzeugt und zum anderen eine englischsprachige, auf der nur die VKs ausgewiesen sind. Die XSL-Regeln zur Erzeugung der deutschen Darstellung sind in Anhang A.1 und die XSL-Regeln für die englische Darstellung in Anhang A.2 zu finden. Durch Benutzung der jeweiligen XSL-Regelsätze entstehen unterschiedliche Darstellungen, wie in Abbildung 5.6 zu sehen ist.

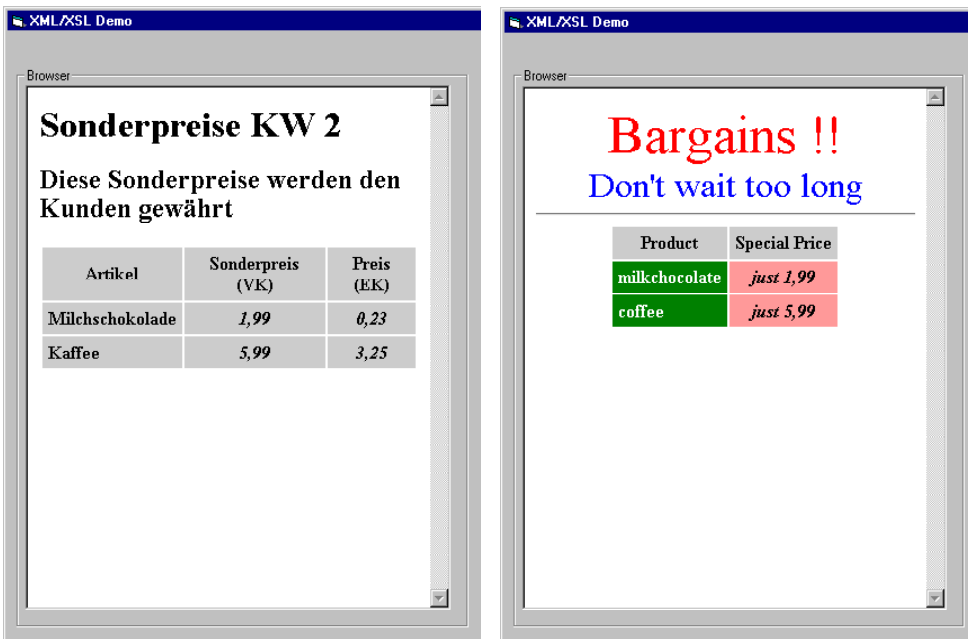


Abbildung 5.6: Verschiedene Visualisierungen derselben XML-Daten

Kapitel 6

Entwurf und Implementierung

In diesem Kapitel wird der Entwurf und die Implementierung des Workflowsystem beschrieben. Große Teile des Entwurfs ergeben sich bereits aus dem in Kapitel 4 beschriebenen Modell. Ähnlich wie bei dem Modell gibt es einen *statischen Teil*, der gemeinsam von allen Arbeitsprozessen genutzt wird und der die Ressourcen zur Verfügung stellt. Dies entspricht konzeptuell dem Systemnetz aus dem Modell. Daneben gibt es noch den *dynamischen Teil*, in dem sich die Arbeitsprozesse unterscheiden. Das sind insbesondere die Initialisierung von Inhalten einer Konversation sowie das Verhalten, also die Regeln, die festlegen, durch welche Aktionen welche Zustandsänderungen vollzogen werden. Im Modell entspricht diese Funktionalität den Objektnetzen. Auch im weiteren wird immer auf Analogien zwischen dem Modell und der Implementierung hingewiesen, um die Parallelen zwischen Modell und Implementierung darzustellen. Auch auf Abweichungen zwischen Modell und Implementierung wird hingewiesen.

Hauptaufgabe des statischen Teils ist die Anmeldung, Autorisierung und Rollenannahme. Weiterhin werden Funktionen zur Verfügung gestellt, durch die neue Konversationen begonnen, laufende fortgesetzt und der jeweils aktive Dialog einer Konversation angezeigt werden kann. Entsprechend dem Systemnetz können Konversationen andere Konversationen *initiieren*, eine Konversation kann die Aufgabe an eine andere *delegieren* und eine Konversation kann ein Ergebnis an eine andere zurückliefern und diese dadurch aus der Warteposition holen (*antworten*).

Aufgabe des dynamischen Teils ist es, die Initialisierung sowie das Verhalten von Konversationen einer Spezifikation festzulegen. Die Implementierung der Rollen ist ebenfalls Bestandteil des dynamischen Teils und kann als Verfeinerung der Komponente *Agent* angesehen werden.

Darüber hinaus wird die Administration des Systems, also das Erstellen von Konversationsspezifikationen, die Benutzerpflege und das Erstellen von Dialogen beschrieben. Die langlebige Speicherung erfolgt in einer relationalen Datenbank. Wie die Abbildung der Komponentenstruktur auf die Datenbank erfolgt, wird in Abschnitt 6.4 beschrieben.

Die Implementierung erfolgt in der in Kapitel 5 vorgestellten Entwicklungsumgebung. Die Klassen die in den nächsten Abschnitten vorgestellt werden, sind *Visual Basic* Komponenten, die in der Umgebung des Microsoft Transaction Servers laufen.

6.1 Klassen des statischen Teils

Die Aufgaben des statischen Teils lassen sich in zwei Bereiche untergliedern. Zum einen die Interaktion mit dem System durch den Benutzer und zum anderen die Behandlung und Ausführung von Konversationen. Jeder dieser Bereiche wird durch eine Klasse abgedeckt. Die Interaktion mit dem System erfolgt über die Klasse *Agent* und die Implementierung der Konversationen erfolgt über die Klasse *Conversation*.

6.1.1 Die Klasse *Agent*

Wie bereits angedeutet, bildet die Klasse *Agent* die Schnittstelle zum System. Benutzer können sich einloggen und eine Rolle übernehmen. Dies wird mit dem ASP-Skript in Abbildung 6.1 demonstriert. Zuerst wird ein Objekt vom Typ *Agent* erzeugt. Dann wird der Benutzername und die Bezeichnung der Applikation (*todo*) übergeben, in die man sich einloggen möchte. War beides erfolgreich, wird die Referenz auf den Agenten in einer Sessionvariablen gespeichert, um einen Zugriff aus den nächsten ASP-Skripten heraus zu ermöglichen. Schließlich wird noch die Rolle eines Bearbeiters angenommen.

```
username = Request("name")

set Agent = Server.CreateObject("BC.Agent_2")
ret = Agent.Login(username, "todo")

if ret = 0 then
  set Session("Agent") = Agent
  Agent.setRole "Bearbeiter"
else
  set Session("Agent") = Nothing
end if
```

Abbildung 6.1: ASP-Skript zum Einloggen in das System

Die Ermittlung der Konversationen, an denen der Benutzer in der aktiven Rolle beteiligt ist, geschieht über die Klassen, die diese Rollen implementieren. Diese *Rollenklassen* können als Verfeinerung der Komponente *Agent* angesehen werden. Da *Visual Basic* aber das Konzepte der Vererbung und Funktionsüberladung nicht unterstützt, gehen auch diese Anfragen erst über die Komponente *Agent* und werden von dort explizit aufgerufen. Auf die Rollenklassen wird bei der Beschreibung des dynamischen Teils genauer eingegangen. Auch auf die Probleme, die durch das Fehlen bestimmter objektorientierter Konzepte in *Visual Basic* entstehen, wird dort eingegangen.

Nach der Ermittlung der Konversationen können diese fortgesetzt werden, indem der aktive Dialog angezeigt wird. Dazu wird eine Methode der Komponente *Agent* aufgerufen (`displayCurrentDialogHTML`), die als Parameter nur die Identifikationsnummer (ID) der Konversation erwartet. Dieser aktive Dialog der Konversation wird als HTML-Formular

[December, Ginsburg 95] angezeigt, welches als Ziel immer auf dasselbe ASP-Skript verweist. Dieses Skript ruft die Methode *action* auf, wobei keine Parameter mehr übergeben werden müssen. Die ID der Konversation ist im Formular enthalten und der Zugriff auf die Daten des Formulars kann direkt durch den MTS erfolgen, da diese Information implizit zwischen IIS und MTS ausgetauscht werden.

6.1.2 Die Klasse *Conversation*

Die Klasse Conversation stellt die Methoden zur Verfügung, die alle Konversationen gemein haben. Dies beinhaltet das persistente Speichern des Inhaltes einer Konversation, das Anzeigen von Dialogen sowie die aus dem Modell bekannten Fähigkeiten zur Initiierung neuer Konversationen und Delegation von Aufgaben an andere Bearbeiter. Die Teile, in denen sich die Konversationen unterscheiden, werden durch *Konversationsklassen* realisiert, die als Verfeinerungen der Klasse *Conversation* angesehen werden können. Diese sind ebenfalls Bestandteil des dynamischen Teils und werden dort genauer beschrieben.

6.1.3 Die Klasse *Dialog*

Dialoge spielen eine zentrale Rolle im Modell der *Business Conversations*. Sie repräsentieren zum einen den aktuellen Zustand einer Konversation und werden außerdem zur Interaktion mit den beteiligten Konversationspartnern genutzt. Die Interaktion mit menschlichen Kommunikationspartnern soll in diesem Falle, mittels eines Browsers erfolgen. Die Verwendung von HTML [Musciano, Kennedy 94] zur Darstellung der Dialoge liegt daher nahe. Verwendet wird diese Klasse hauptsächlich von der Klasse *Conversation*, die die Visualisierung des aktiven Dialogs anfordert und zur Ausgabe weiterleitet. Wenn die HTML-Darstellung eines Dialogs angefordert wird, ist es die Aufgabe der Klasse *Dialog*, die XML-Darstellung des Dialogs mit dem aktuellen Inhalt der Konversation zu füllen und schließlich mittels eines XSL-Regelwerkes (siehe Abschnitt 5.5) in HTML zu konvertieren. Der folgende Abschnitt beschreibt, wie Dialoge in XML beschrieben werden können.

6.1.3.1 Dialogbeschreibung mit XML

In diesem Abschnitt werden die XML-Elemente beschrieben, die zur Spezifikation von Dialogen verwendet werden können. Das erste Element muß immer ein Dialog-Element sein, das alle folgenden Elemente klammert und auch als Wurzelelement (*root-element*) bezeichnet wird. Die Elemente werden im folgenden vorgestellt. In Anhang B ist die Syntax sowohl in *Backus-Naur-Form* als auch in Form einer *Document Type Definition* (DTD), angegeben. Die DTD kann vom *XML-Parser* zur Syntaxkontrolle verwendet werden. Die Elemente sind im einzelnen :

Dialog beschreibt die oberste Ebene eines jeden Dialogs.

Title dient zur Angabe eines Titels für dem Dialog.

Form fängt den Bereich der Dialogbeschreibung an, in dem die Inhalte, die in diesem Dialog angezeigt werden sollen, angegeben werden.

Data dient zum Anzeigen der Inhalte eines Dialoges. Es wird zwischen den Typen *Display*, *Input*, *Textarea* und *Select* unterschieden. *Input* und *Textarea* ermöglichen die Eingabe bzw. das Ändern eines Inhaltes, während *Display* diesen anzeigt, ohne daß er geändert werden kann. Im Falle von *Textarea* wird eine mehrzeilige Eingabe ermöglicht, wobei die Anzahl der Zeilen (*rows*) und Spalten (*cols*) angegeben werden kann. *Select* stellt eine Auswahl zur Verfügung, wobei zum Beispiel alle Benutzer angezeigt werden können, die einer bestimmten Rolle angehören.

Action markiert den Anfang des Bereichs, in dem die möglichen Aktionen, die in einem Dialog ausgeführt werden können, angegeben werden.

Do beschreibt eine mögliche Aktion in einem Dialog. In der Regel werden diese als Schaltflächen eines HTML-Formulars dargestellt.

Das folgende Beispiel demonstriert die Anwendung dieser XML-Elemente. Der so formulierte Dialog stammt aus der Beispielapplikation aus Kapitel 7. Die Darstellung des Dialogs ist in Abbildung 7.4 zu sehen.

```
<?XML version="1.0"?>
<Dialog>
  <Title>Aufgabe bearbeiten</Title>
  <Form>
    <Data Type="Display" Titel="Projekt"><Projekt/></Data>
    <Data Type="Display" Titel="Titel"><Titel/></Data>
    <Data Type="Textarea" Titel="Beschreibung" rows="10" cols="60">
      <Beschreibung/>
    </Data>
    <Data Type="Select" Titel="Bearbeiter">
      <Bearbeiter roles="Bearbeiter"/>
    </Data>
    <Action>
      <Do>ok</Do>
      <Do>Abbrechen</Do>
    </Action>
  </Form>
</Dialog>
```

6.2 Klassen des dynamischen Teils

Die Klassen des dynamischen Teils übernehmen alle Aufgaben, durch die sich die Konversationen und Agenten unterscheiden. Für jede Konversationsspezifikation gibt es eine Klasse, die das Verhalten und die Initialisierung der Konversationen dieser Spezifikation beschreibt. Diese Klassen werden als *Konversationsklassen* bezeichnet. Weiterhin gibt es für jede Rolle, die ein Agent annehmen kann, eine sogenannte *Rollenklasse*. Über diese

Rollenklassen hat der Benutzer Zugriff auf die Liste aller Konversationen, an denen er noch beteiligt ist.

6.2.1 Rollenklassen

Ein Benutzer kann in einer Rolle Konversationen einer oder mehrerer Konversationsspezifikationen führen. Ein Bearbeiter kann zum Beispiel Konversationen führen, in denen es um die Lösung einer Aufgabe geht, andererseits kann er aber auch als Aufgabensteller auftreten, wenn es um die Vergabe von Teilaufgaben an andere Bearbeiter geht. Eine Rolle ist also nicht immer nur mit einer Konversationsspezifikation in Verbindung zu setzen. Daher muß für die Rolle eine explizite Angabe erfolgen, Konversationen welcher Spezifikation geführt werden können. Hierzu wird die Funktion *getConversations*, die auch eine Methode der Komponente *Agent* ist, überladen. Abbildung 6.2 zeigt ein Codesegment aus dem Quellcode einer Rollenklasse. Durch den SQL-Befehl werden alle IDs von Konversationen zurückgeliefert, bei denen der Benutzer beteiligt ist, der Name der Konversationsspezifikation *qs* lautet und die Konversation entweder noch läuft (RunStatus=0) oder sich in Warteposition (RunStatus=1) befindet.

```
Function getConversations() As Collection

  ...

  sql$ = "select ConversationId, RunStatus from BConversations "
  sql$ = sql$ & " where Performer=" & m_parent.getUserId()
  sql$ = sql$ & " and Name='qs'"
  sql$ = sql$ & " and RunStatus<2 "

  conn.Open m_connstring
  Set rs = conn.Execute(sql$)

  ...

End Function
```

Abbildung 6.2: Implementierung von *getConversations* in einer Rollenkomponente

Aus der Sicht der Systementwicklung handelt es sich bei den Rollenklassen um eine Verfeinerung der Klasse *Agent*. Je nachdem welche Rolle angenommen wurde, muß der Aufruf von *getConversations* an die Methode aus der richtigen Klassen gebunden werden. Dies entspricht dem objektorientierten Konzept der *späten Bindung*. Dieses Konzept ist bei *Visual Basic* jedoch nicht vorhanden, weswegen man hier einen Umweg gehen muß. Die Schnittstelle zum System bietet immer die Komponente *Agent*. Wenn der Agent eine bestimmte Rolle annehmen soll, so wird überprüft, ob es eine Rollenklasse für diese Rolle gibt. Bei einem Aufruf von *getConversations* wird zuerst die Methode der Klasse *Agent*

aufgerufen, welche ausschließlich die Aufgabe hat, ein Objekt der Rollenklasse zu erzeugen und die Methode *getConversations* aufzurufen (siehe Abbildung 6.4). Dies birgt in zwei Fällen die Gefahr Laufzeitfehler zu erzeugen. Zum einen kann es sein, daß aufgrund eines Schreibfehlers eine Rolle angenommen werden soll, für die es keine Rollenklasse gibt. Wenn zum Beispiel die Zeile, in der die Rolle angenommen wird (siehe Abbildung 6.1), versehentlich in

```
...
Agent.setRole "Bearbeitr"
...
```

geändert wird, so versucht die Klasse *Agent* die Rollenklasse *Bearbeitr* zu finden, die es aber gar nicht gibt.

Die zweite Fehlerquelle ist, daß es keine Möglichkeit gibt, das Vorhandensein bestimmter Funktionen in einer Klasse zu erzwingen. Bei objektorientierten Sprachen hätte man sich einer abstrakten Basisklasse bedient und so die Existenz einer Funktion erzwingen können. In diesem Fall kann man aber eine Rollenklasse erstellen, die nicht über eine Methode *getConversations* verfügt oder eine falsche Signatur besitzt. Auch hier würde der Fehler erst zur Laufzeit auftreten.

6.2.2 Konversationsklassen

Der Großteil aller Aufgaben, die die Klasse *Conversation* zu erfüllen hat, ist für alle Konversationen gleich. Das Speichern des Inhaltes sowie das Darstellen des aktiven Dialoges einer Konversation kann durch dieselben Methoden realisiert werden. Wo sich die Konversationen aber unterscheiden, ist die Initialisierung mit Startwerten, sowie im Verhalten. Unter Verhalten ist hier das Regelwerk zu verstehen, das vorgibt, welche Zustandsänderungen durch welche Aktionen ausgeführt werden. Letzteres wird durch die sogenannte *Aktionsfunktion* realisiert. In dieser kann auf den aktuellen Zustand der Konversation zugegriffen werden. In Abhängigkeit von diesem Zustand und davon welche Aktion ausgelöst wurde, wird hier der Folgezustand festgelegt. Das folgende Codesegment zeigt einen Ausschnitt einer Aktionsfunktion.

```
Sub action(Optional ByRef parent As Object)
  ...
  Select Case currentId

  Case 17 ' bearbeiten
    Select Case action
      Case "ok"
        parent.UpdateDialogData
        parent.setCurrentDialog 49
      Case "Abbrechen"
        parent.setCurrentDialog 30
    End Select
```

```
  ...
End Sub
```

Wenn man sich in Dialog 17 befindet und der "ok"-Knopf gedrückt worden ist, so wird der neue Inhalt der Konversation gespeichert *(UpdateDialogData)* und der Dialog 49 wird zum aktuellen Dialog der Konversation. Wurde dagegen der "Abbrechen"-Knopf gedrückt, so werden die Inhalte nicht verändert und es wird in Dialog 30 gewechselt.

6.3 Interaktion mit dem System

Die einzige Schnittstelle zur Außenwelt wird durch die Klasse Agent zur Verfügung gestellt. Im folgenden wird exemplarisch die Interaktion der beteiligten Klassen anhand von Interaktionsdiagrammen vorgestellt. Gezeigt wird der Ablauf der Methodenaufrufe beim Starten einer neuen und beim Fortsetzen einer bereits laufenden Konversation. Daran läßt sich auch die Verwendung des statischen und des dynamischen Teils besser veranschaulichen.

6.3.1 Starten einer neuen Konversation

Das Starten eine neuen Konversation erfolgt immer dann, wenn eine neue Aufgabe durch einen Aufgabensteller gestellt wird. Abbildung 6.3 zeigt den Ablauf der Methodenaufrufe beim Starten einer neuen Konversation, sowie das eigentliche Führen der Konversation, bis zu dem Moment, wo der Benutzer das System verläßt *(Logout)*.

Die Aufrufe erfolgen hierbei aus den ASPs heraus. Dabei wird zuerst eine Objekt der Klasse *Agent* erzeugt. Die Referenz wird in einer Sessionvariable gespeichert, damit das Objekt auch von den anderen ASPs (*newconv.asp*, *execute.asp*, *logout.asp*) aus erreichbar bleibt. Um die Konversation zu starten, wird ein neues Objekt angelegt, das wiederum ein Objekt der gewünschten Konversationsklasse erzeugt. Das Anzeigen der aktuellen Dialoge erfolgt durch den Aufruf von *displayCurrentDialogHTML*. Zurückgeliefert wird HTML, das an den Browser weitergereicht wird. Der Aufruf von execute.asp kann beliebig oft hintereinander ausgeführt werden und steht für einen Konversationsschritt. Der Aufruf von *Logout* hat zur Folge, daß die Referenz auf das Konversationsobjekt sowie das Objekt der Konversationsklasse gelöscht werden. Das Löschen des Objektes erfolgt automatisch durch den *Garbage Collector* des MTS. Deutlich wird hierbei die Verwendung der Konversationsklasse. Beim Starten der Konversation wird die Initialisierungsfunktion *init* aufgerufen und in jedem Konversationsschritt erfolgt ein Aufruf der Aktionsfunktion *action*. Alle anderen Funktionalitäten, wie zum Beispiel das Anzeigen des aktuellen Dialoges, haben alle Konversationen gemein, wodurch sie nicht Bestandteil der Konversationsklasse sein müssen.

6.3.2 Fortsetzen von Konversationen

Ähnlich wie das Starten einer Konversation sieht das Fortsetzen einer Konversation aus. Der Unterschied hierbei ist, daß eine bestehende Konversation ausgewählt wird. Die Liste

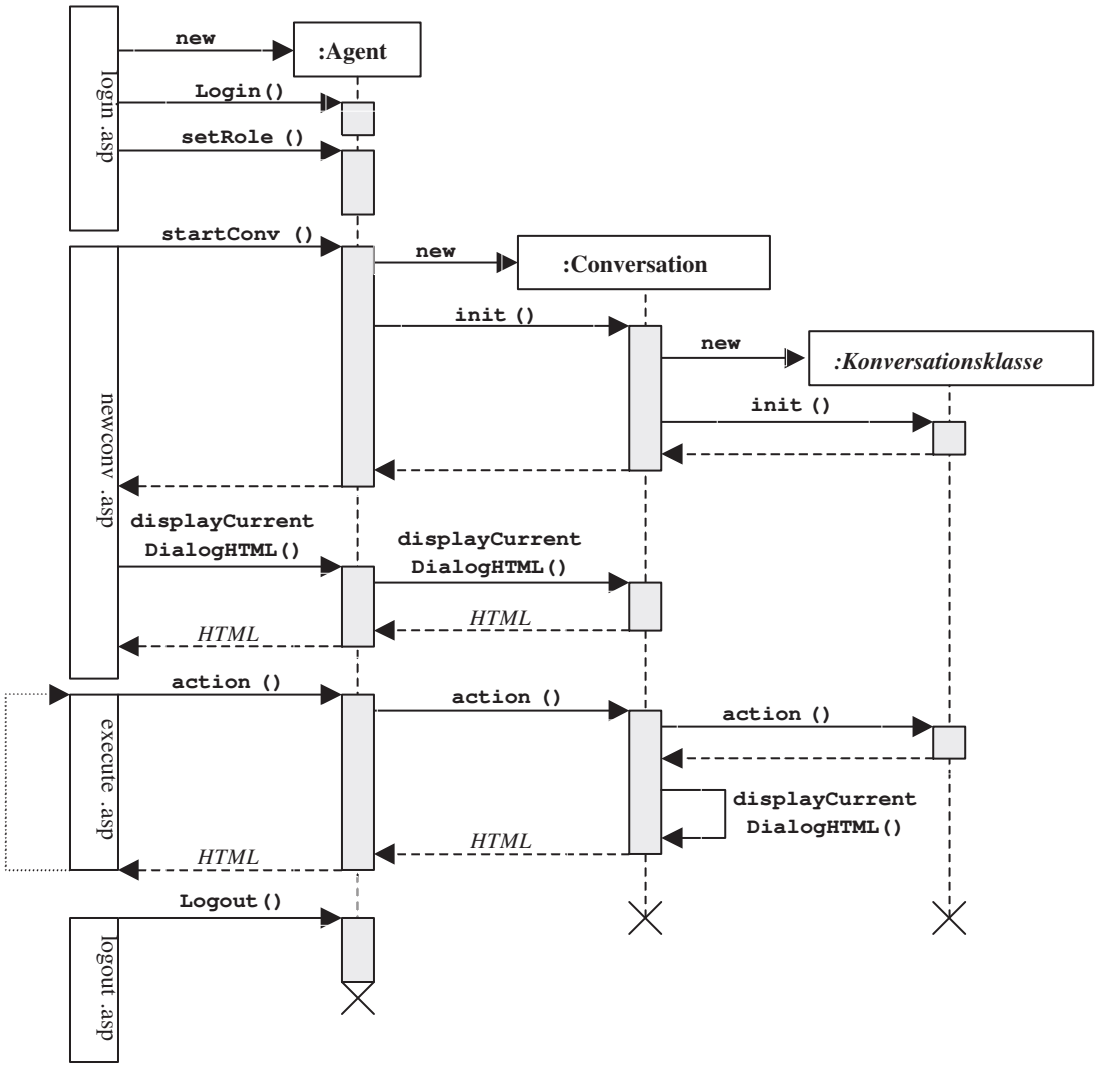


Abbildung 6.3: Starten einer neuen Konversation

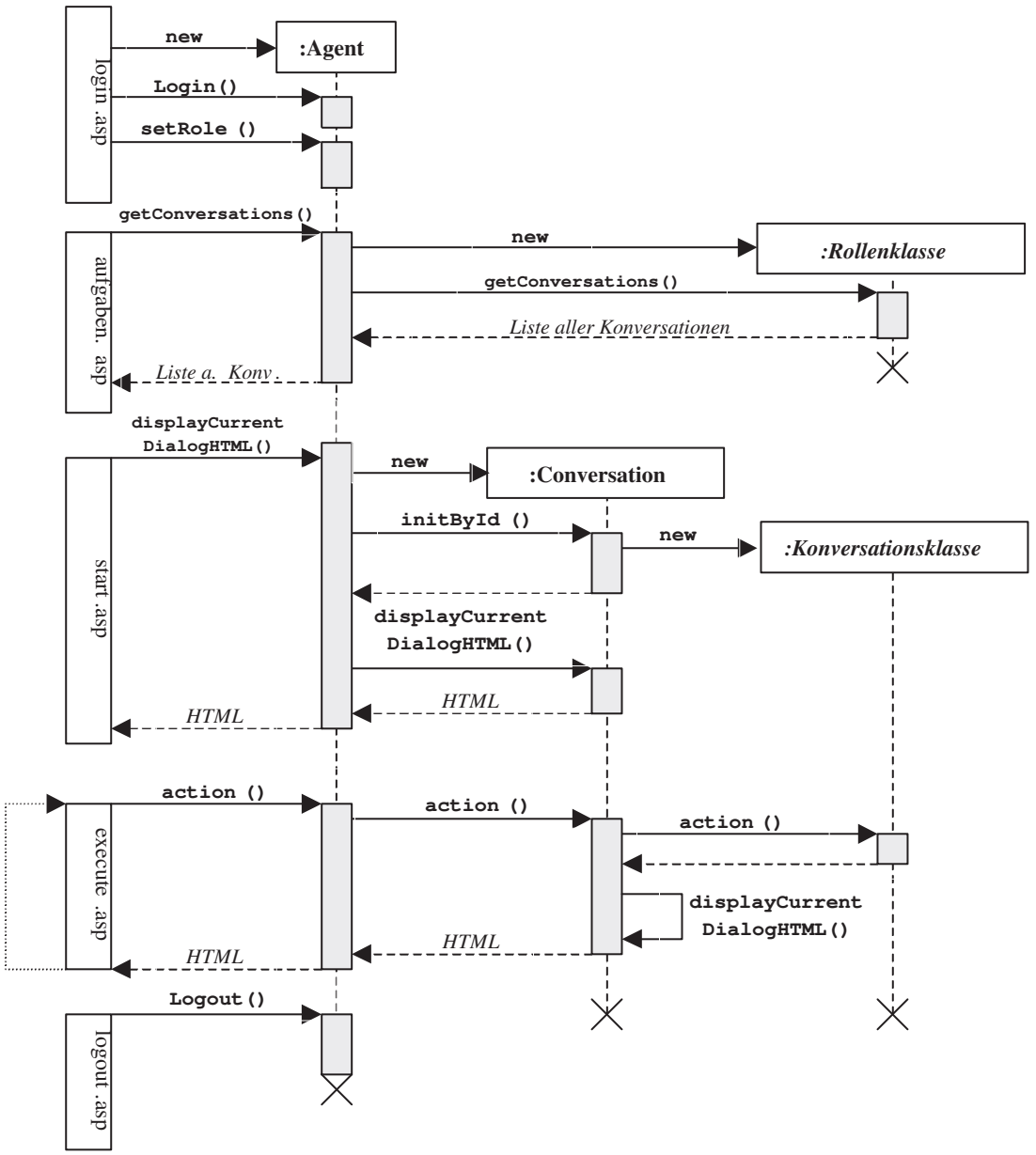


Abbildung 6.4: Fortsetzen einer Konversation

der noch aktiven Konversationen wird hierbei durch die Rollenklasse geliefert. Der Aufruf der Methode *getConversations* liefert eine Liste von Konversationen zurück. Eine dieser Konversationen kann ausgewählt und fortgesetzt werden. Abbildung 6.4 zeigt den Ablauf hierfür.

Deutlich wird hierbei auch die Funktion der Rollenklassen. Die Implementierung der Methode *getConversations*, wie sie in Abschnitt 6.2.1 beschrieben wurde, erfolgt in diesen Klassen. Dadurch kann individuell pro Rolle festgelegt werden, welche Konversationen noch laufen.

6.4 Komponentenstruktur und Datenbankmodell

Die Arbeitsprozesse innerhalb des Workflowsystems werden mit den im letzten Abschnitt beschriebenen Klassen realisiert. Die Zustände der Klassen repräsentieren somit auch den Zustand des Arbeitsprozesses und müssen daher *langlebig* gespeichert werden. Da die Klassen nicht über eine implizite Speicherungsmöglichkeit (Persistenzabstraktion) verfügen, müssen die Zustände der Klassen explizit in einer Datenbank gespeichert werden. Hierzu wird ein relationales Datenbankmodell entwickelt, auf welches sich die Klassenstruktur abbilden läßt. Abbildung 6.5 zeigt die Klassenstruktur.

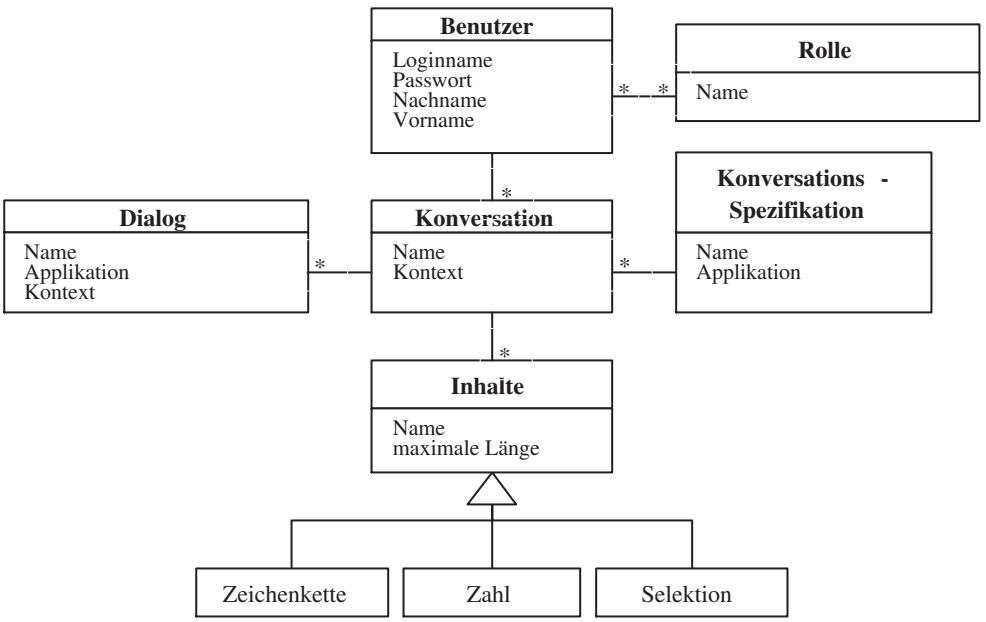


Abbildung 6.5: Klassenstruktur

Die Abbildung auf eine Datenbankmodell erfolgt durch einen einfachen Algorithmus, wie er zum Beispiel in [Rumbaugh et al. 93] beschrieben wird. Hierbei werden Klassen auf Tabellen abgebildet. *1:n*-Beziehung zwischen Klassen können direkt durch einen Fremdschlüssel realisiert werden und *n:m*-Beziehungen werden, wie in relationalen Datenbanken

üblich, mittels einer Verbindungstabelle dargestellt, die zwei Fremdschlüssel enthält.

6.5 Historienmanagement mittels eines *BC-Addons*

Um den Ablauf eines bestimmten Arbeitsprozesses nachvollziehen zu können, möchte man auf eine Ablaufhistorie zugreifen können, in der vermerkt wird wann und von wem welche Aktionen in den Konversationen ausgelöst worden sind und welche Inhalte in welchen Schritten modifiziert wurden. Hierzu kann zum Beispiel ein Protokollierungsdienst verwendet werden, wie er in [Kruse 97] beschrieben wird. Dieser Dienst ist dort direkt auf der Ebene der *Business Conversations* realisiert und protokolliert implizit den Ablauf der Konversationen. Der Ansatz, der in dieser Arbeit gewählt wurde, beruht auf der Verwendung eines BC-Addons, das explizit in der Aktionsfunktion der Konversationen verwendet wird. Dadurch werden alle Änderungen von Inhalten sowie die Zustandsübergänge protokolliert. Die Auswertung dieses Protokolls erfolgt durch Verwendung der externen Schnittstelle des BC-Addons. Abbildung 6.6 zeigt die Ausgabe einer solchen externen Anwendung. Die einzelnen Schritte der Konversation können angezeigt werden. Außerdem ist sichtbar, welche Inhalte durch die nächste Aktion verändert wurden.

Abbildung 6.6: Darstellung einer Ablaufhistorie

Kapitel 7

Beispiel

Anhand eines Beispiels wird die Verwendung des Workflowsystems verdeutlicht. Ein einfaches System zur Aufgabenverwaltung beschreibt den Weg von der Workflowanalyse bis zur Umsetzung im Workflowsystem. Daran wird auch demonstriert, wie die in Kapitel 4 beschriebenen *Elementaren Objektsysteme* zur Analyse und Modellierung eingesetzt werden und wie die Umsetzung in einen lauffähigen Arbeitsprozeß erfolgt.

7.1 Problemstellung und Analyse

Aufgabe ist es, ein System zu entwickeln, bei dem zu lösende Aufgaben eingegeben werden, die dann an einen ausgewählten Bearbeiter weitergeleitet werden. Die Formulierung der Aufgabe kann hierbei in mehreren Schritten durch den Aufgabensteller erfolgen. Der Aufgabensteller hat also die Möglichkeit, eine Aufgabe erst einmal zu formulieren, ohne daß diese gleich an einen Bearbeiter vergeben wird. Für den Fall, daß sich die Aufgabe von alleine löst, kann diese durch den Aufgabensteller auch einfach verworfen werden. Ist die Aufgabe fertig formuliert, soll ein Bearbeiter ausgewählt werden. Dieser erhält die Aufgabe und kann diese in beliebig vielen Schritten bearbeiten. Der Bearbeiter sollte außerdem die Möglichkeit haben, die Aufgabe an einen anderen Bearbeiter zu übergeben. Sieht der Bearbeiter die Aufgabe als gelöst an, oder sieht er sich außerstande sie zu lösen, kann er eine Antwort an den Aufgabensteller zurückgegeben. Dieser kann die Aufgabe jetzt als gelöst bestätigen oder aber die Aufgabe weiterbearbeiten. Dies kann zum Beispiel bedeuten, daß die Aufgabe präziser formuliert wird oder ein anderer Bearbeiter ausgewählt wird.

Es soll möglich sein, die Aufgaben mit einer Priorität zu versehen. Diese soll aus *sehr hoch*, *hoch*, *mittel*, *niedrig* und *sehr niedrig* ausgewählt werden können. Aufgaben werden einem Projekt zugeordnet und erhalten einen Aufgabentitel. Die Aufgabe selber sowie die Lösungen, Vorschläge oder Kommentare werden in einem Textfeld gespeichert.

Weiterhin soll vermerkt werden, wann der Gesamtarbeitsprozeß begonnen worden ist, und wann und durch wen eine Aufgabe gelöst worden ist. Dieser Ablauf wird als *Elementares Netzsystem* in Abbildung 7.1 dargestellt.

Die Bezeichnung der Stellen deutet bereits die benötigten Dialoge an. Ist eine Stelle markiert, bedeutet dies, daß der zugehörige Dialog den aktiven Dialog der Konversation

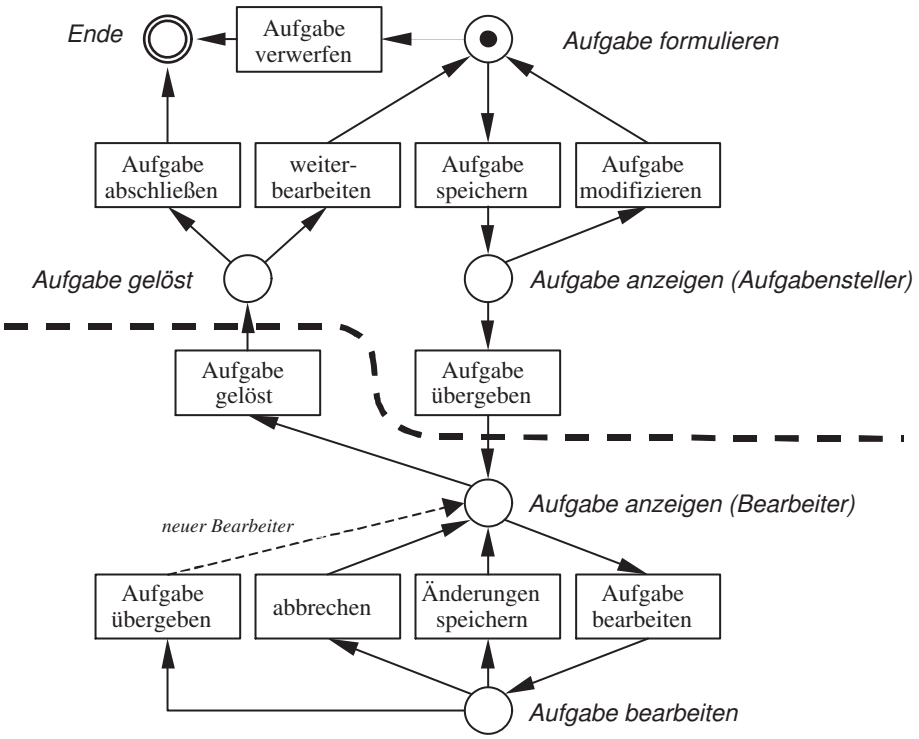


Abbildung 7.1: Arbeitsprozeß als Elementares Netzsystem

darstellt. Alle Transitionen, die durch die Belegung einer Stelle aktiviert werden, geben an, welche Aktionen in diesem Dialog ausgeführt werden können.

7.2 Modellierung

Um den eben beschriebenen Arbeitsprozeß in dem in Kapitel 4 entwickelten Systemnetz zu verwenden, muß das Netz um die Aufgaben des Verteilers erweitert werden. Dadurch erhält man ein *Elementares Netzsystem*, welches als Objektnetz verwendet werden kann. Hierzu werden, wie in Abschnitt 4.3 beschrieben, hinter jeder Transition zusätzliche Transitionen, im folgenden als *Verteilertransitionen* bezeichnet, eingefügt. Welche Aufgabe der Verteiler jeweils zu erfüllen hat, richtet sich danach, wer für die Ausführung der Transition vor und nach der neu eingefügten Verteilertransition zuständig ist. Das erweiterte Netz ist in Abbildung 7.2 dargestellt. Bei der Übergabe der Aufgabe an einen Bearbeiter kann im Modell noch nicht festgestellt werden, an welchen Bearbeiter die Aufgabe geleitet werden soll. Daher kann die Transition `<INITn>` noch nicht mit der entsprechenden Nummer des Bearbeiters versehen werden. Diese wird erst zur Laufzeit bestimmt. Gleiches gilt im Falle einer Delegation an einen anderen Bearbeiter, weswegen auch die Beschriftung dieser Verteilertransition `<DELEGn>` lautet.

Durch die Beschriftung der Verteilertransitionen ergibt sich auch die für ein *Elemen-*

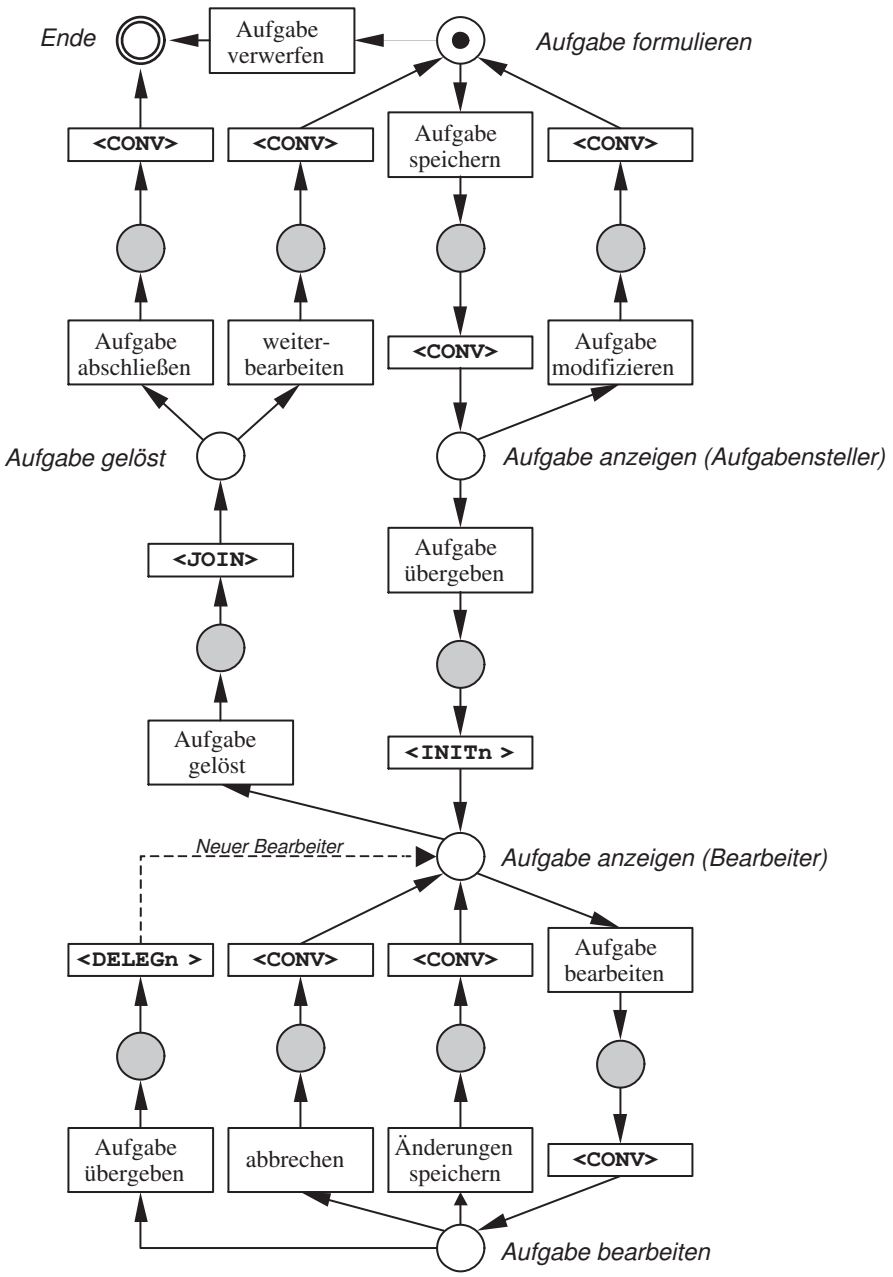


Abbildung 7.2: Arbeitsprozeß als erweitertes Elementares Netzsystem

tares Objektnetz notwendige Interaktionsrelation.

Für die spätere Implementierung ist es noch notwendig, die Einzelkonversationen zu identifizieren, in die der Gesamtarbeitsprozeß zerlegt werden kann. Dieses wird in Abbildung 7.3 gezeigt. Der obere Teil bildet die Konversation des Aufgabenstellers mit dem Verteiler. Die Spezifikation dieser Konversation wird durch *aufgabestellen* bezeichnet. Die Spezifikation der Konversationen der Bearbeiter mit dem Verteiler wird durch *bearbeiten* bezeichnet.

7.3 Implementierung

Die Implementierung besteht aus zwei Teilen. Der erste Teil ist die Erstellung der Konversationsspezifikationen. Dies umfaßt auch die Inhalte der Konversationen sowie die benötigten Dialoge. Weiterhin müssen die beteiligten Rollen identifiziert und den Benutzer des Workflowsystems zugeordnet werden. Der zweite Teil ist die Programmierung der Rollen- und Konversationsklassen in einer dem Arbeitsprozeß zugehörigen Komponente, welche später im Microsoft Transaktion Server integriert wird.

7.3.1 Konversationsspezifikation, Dialoge und Rollen

Die Analyse hat ergeben, daß zwei Konversationstypen benötigt werden. Diese zwei Konversationstypen werden durch Konversationsspezifikationen namens *aufgabestellen* und *bearbeiten* realisiert. Für jede Konversationsspezifikation wird der Name der Konversation, das Verzeichnis, in dem sich die zugehörigen Dialogspezifikationen befinden, Informationen für den Datenbankzugriff, die XSL-Datei zur Formatierung der Dialoge, sowie die ID des Startdialoges der Konversation angegeben. Dieses kann direkt per SQL-Befehl in der Datenbank erfolgen, oder aber mittels eines browsergestützten Datenbankfrontends zur Verwaltung der Spezifikationen, Dialoge und Rollen. So ein Frontend wurde im Rahmen dieser Arbeit ebenfalls entwickelt. In Anhang C.1 befindet sich eine Abbildung, die die Dateneingabe für die Konversationsspezifiaktion *aufgabestellen* zeigt. Die beschrifteten Stellen aus Abbildung 7.3 stellen die benötigten Dialoge dar. Daraus ergeben sich fünf Dialoge für *aufgabestellen* und vier Dialoge für *bearbeiten*. Die folgenden Tabellen giben alle Dialoge an, die benötigt werden, zusammen mit den IDs und den Dateinamen für die Dialogbeschreibungen in XML. Die IDs werden durch die Datenbank automatisch erzeugt. Die Eindeutigkeit wird dabei vom Datenbanksystem sichergestellt.

Konversationsspezifikation *aufgabestellen*		
ID	Titel	Dateiname
12	Aufgabe formulieren	form.xml
13	Aufgabe anzeigen	anzeigen.xml
15	Aufgabe beendet	ende.xml
22	Aufgabe gelöst	solved.xml
23	Aufgabe verworfen	cancel.xml

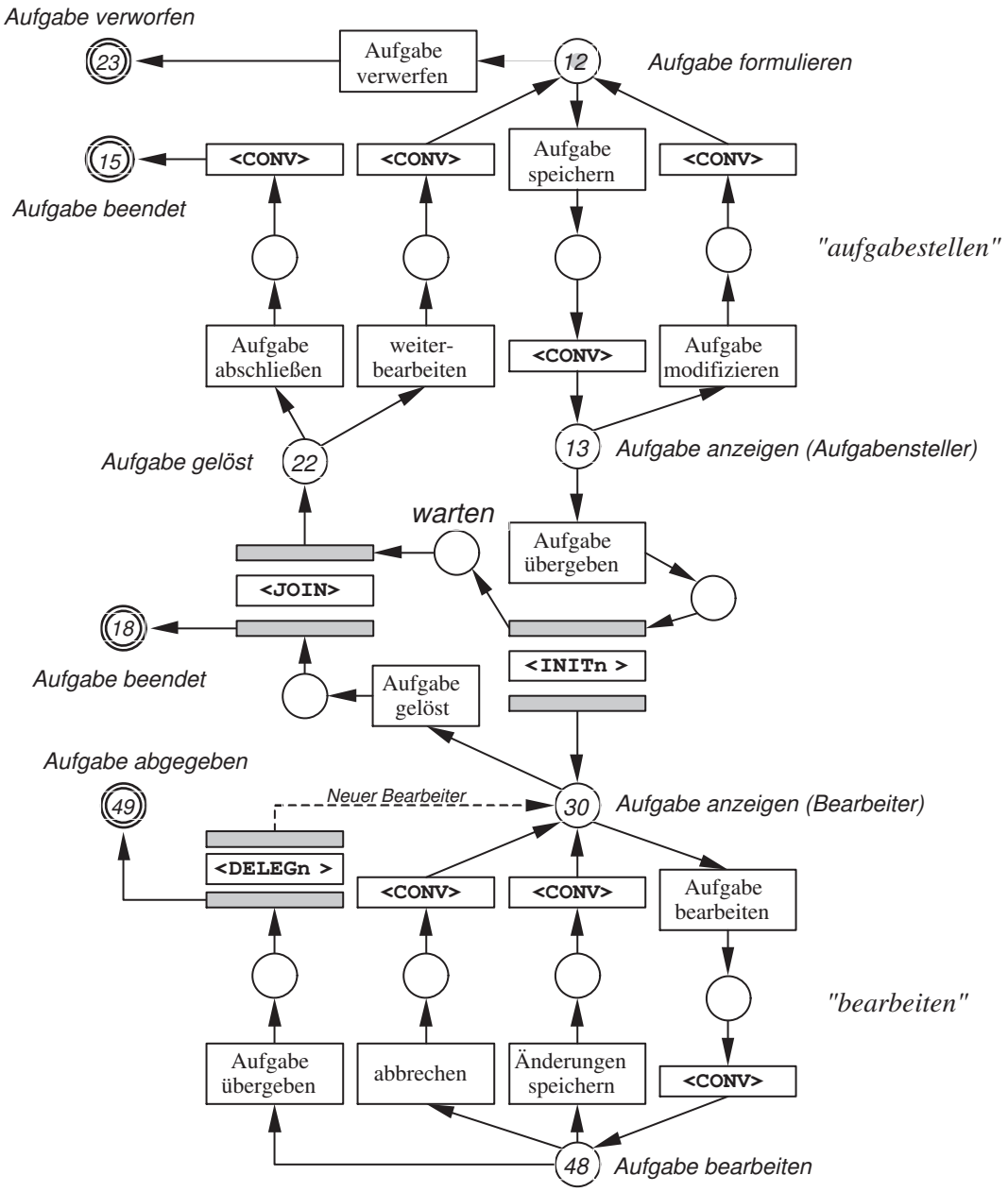


Abbildung 7.3: Teilung des Arbeitsprozesses in Einzelkonversationen

Konversationsspezifikation *bearbeiten*		
ID	Titel	Dateiname
48	Aufgabe bearbeiten	a_bearb.xml
30	Aufgabe anzeigen	a_anzeigen.xml
18	Aufgabe beendet	a_ende.xml
49	Aufgabe abgegeben	a_deleg.xml

Bei den Rollen gibt es nur einen Aufgabensteller und einen Typ von Bearbeiter, weswegen die Anzahl der Rollen auf zwei beschränkt ist. Die Rollen heißen *Submitter* für den Aufgabensteller und *Bearbeiter* für die Bearbeiter. In der Benutzerverwaltung muß nun eine Zuteilung der Rollen zu den Benutzern erfolgen. Die folgende Tabelle zeigt die Zuordnung der Rollen zu den Benutzern des Beispielsystems.

Benutzer	Kürzel	Rollen
Jan Stövesand	jst	Submitter, Bearbeiter
Alexander Gast	ag	Submitter
Thomas Meier	tm	Submitter, Bearbeiter

7.3.2 Programmierung der Arbeitsprozeßkomponente

Den komplexeren Teil der Implementierung bildet die Programmierung der *Arbeitsprozeßkomponente*. Für jede Konversationsspezifikation und jede Rolle muß eine Klasse gleichen Namens vorhanden sein. Für das Beispiel wird also eine Komponente mit den Klassen *aufgabestellen*, *bearbeiten*, *Submitter* und *Bearbeiter* benötigt.

7.3.2.1 Rollenklassen

Bei der Implementierung der Rollenklassen beschränkt sich die Arbeit auf die Angabe, welche Konversationen zu einer bestimmten Rolle gehören. Diese Konversationen werden von der Funktion *getConversations* zurückgeliefert. Diese Funktion unterscheidet sich in den unterschiedlichen Rollenklassen nur in der Konstruktion des SQL-Strings. Das folgende Codesegment zeigt die Implementierung für die Rolle *Bearbeiter*.

```
Function getConversations() As Collection '
...
  sql$ = "select ConversationId, RunStatus from BConversations "
  sql$ = sql$ & " where Performer=" & m_parent.getUserId()
  sql$ = sql$ & " and Name='bearbeiten'"
  sql$ = sql$ & " and RunStatus<2 "
...
End Function
```

Der SQL-Befehl bedeutet, daß alle Konversationen vom Typ *bearbeiten* zurückgegeben werden, die entweder noch laufen (*Runstatus=0*) oder in Warteposition sind (*RunStatus=1*). Zum Vergleich zeigt das nächste Codesegment die Implementierung für die Rolle

submitter. Es ist leicht zu sehen, daß alle Konversationen des Typs *aufgabestellen* zurückgegeben werden.

```
Function getConversations() As Collection
...
  sql$ = "select ConversationId, RunStatus from BConversations "
  sql$ = sql$ & " where Performer=" & m_parent.getUserId()
  sql$ = sql$ & " and Name='aufgabestellen' "
  sql$ = sql$ & " and RunStatus<2 "
...
End Function
```

Der Fall, daß alle laufenden und wartenden Konversationen zurückgeliefert werden, stellt den Standardfall dar. Die Schnittstelle wurde aber dennoch so gewählt, daß auch eine Auswahl bereits beendeter Konversationen möglich ist. Dies könnte bei Rollen der Fall sein, die eine Beobachterfunktion haben, oder vielleicht einen Überblick über alle bereits gelaufenen Konversationen benötigen. Zurückgeliefert wird in beiden Fällen eine Kollektion der IDs der gefundenen Konversationen.

7.3.2.2 Konversationsklassen

Konversationsklassen haben zwei Aufgaben zu erfüllen. Zum einen müssen sie eine Funktion *initialize* implementieren, mittels derer Inhalte der Konversation mit Startwerten belegt werden können. Bei Konversationen des Typs *aufgabestellen* soll zum Beispiel die Priorität der Aufgabe anfangs als mittel eingestuft werden. Die Implementierung der Funktion sieht demnach folgendermaßen aus:

```
Sub initialize(Optional parent As Object)
  parent.m_Contents("prio").cntValue = 40 ' mittlere Priorität
  m_objCtx.SetComplete
End Sub
```

Für Konversationen des Typs *bearbeiten* soll das Datum gespeichert werden, an dem die Aufgabe übergeben worden ist.

```
Sub initialize(Optional parent As Object)
  parent.SetData "SubmitDate", Format(Date, "dd.mm.yyyy")
  m_objCtx.SetComplete
End Sub
```

Hier wird durch die Verwendung der Funktionen *Format* und *Date* auch deutlich, daß man bei der Implementierung der Klassen auf den vollen Funktionsumfang der Entwicklungssprache zurückgreifen kann. Dies schließt die Verwendung externer Funktionsbibliotheken mit ein.

Die zweite Aufgabe der Konversationsklasse ist die Implementierung einer *Aktionsfunktion*, in der geregelt wird, von welchem Dialog man zu welchem mit einer gegebenen

Aktion gelangt. Hierbei kann man neben der Information, welcher Dialog gerade aktiv ist und welcher Knopf auf dem Formular gedrückt worden ist, auch auf die Inhalte des Formulars zugreifen. Dadurch lassen sich auch Aktionen bestimmen, die in Abhängigkeit zu den Eingaben des Benutzers stehen. Als Beispiel soll ein Dialog aus der Konversationsspezifikation *bearbeiten* dienen. Der Dialog *Aufgabe bearbeiten* (Dialog 17, siehe Abbildung 7.3) soll die Möglichkeit bieten, Änderungen an den Daten vorzunehmen und diese zu speichern. Bleibt das Feld, in dem der Bearbeiter gewählt wird dabei unverändert, so werden einfach nur die Daten gespeichert und es erfolgt ein Wechsel zum Dialog *Aufgabe anzeigen*. Wird der Bearbeiter aber geändert, so werden zwar auch die restlichen Daten gespeichert, zusätzlich soll aber eine Delegation der Aufgabe zu dem neu ausgewählten Bearbeiter erfolgen. Der folgende Ausschnitt aus dem Quelltext der Aktionsfunktion zeigt die Umsetzung dieses Verhaltens. Der vollständige und ausführlich kommentierte Quelltext ist in Anhang C.2 zu sehen.

```
Sub action(Optional ByRef parent As Object)
  ...
  Select Case currentId
  Case 17 ' bearbeiten
    Select Case action
      Case "ok"
        parent.UpdateDialogData
        id = parent.GetData("Bearbeiter")
        If id <> parent.m_PerformerId Then
          parent.InitiatePeer id, INIT_DELEGATE
          parent.setCurrentDialog 49
        Else
          parent.setCurrentDialog 30
        End If ' if Request ...
      Case "Abbrechen"
        parent.setCurrentDialog 30
    End Select
  ...
End Sub
```

Wenn also in Dialog 17 der Knopf mit der Beschriftung *ok* gedrückt wurde, so werden zunächst die Daten gespeichert (*UpdateDialogData*). Dann wird geprüft, ob in dem Feld, in dem die Bearbeiter angezeigt werden, eine Änderung aufgetreten ist. In dem Falle wird eine Delegation der Aufgabe zu diesem Bearbeiter vollzogen und als nächster Dialog dieser Konversation wird der finale Dialog 49 gewählt. Ist der Bearbeiter noch derselbe, wird lediglich zu Dialog 30 gewechselt. Abbildung 7.4 zeigt eine Bildschirmdarstellung des entsprechenden Dialogs.

Darin ist auch die Beschriftung der beiden Knöpfe zu sehen, welche in der Aktionsfunktion abgefragt werden, sowie das Feld mit dem die Bearbeiter ausgewählt werden können. Der folgende Quelltext zeigt den XML-Code des Dialogs.

```
<?XML version="1.0"?>
<Dialog>
  <Title>Aufgabe bearbeiten</Title>
  <Form>
    <Data Type="Display" Titel="Projekt"><Projekt/></Data>
    <Data Type="Display" Titel="Titel"><Titel/></Data>
    <Data Type="Textarea" Titel="Beschreibung" rows="10" cols="60">
      <Beschreibung/>
    </Data>
    <Data Type="Select" Titel="Bearbeiter">
      <Bearbeiter roles="Bearbeiter"/>
    </Data>
    <Action>
      <Do>ok</Do>
      <Do>Abbrechen</Do>
    </Action>
  </Form>
</Dialog>
```

Abbildung 7.4: Dialog zur Bearbeitung einer Aufgabe

Bei der Angabe des Auswahlfeldes wird auch die Einschränkung der angezeigten Bearbeiter auf eine bestimmte Rolle demonstriert. Es werden in der Liste nur die Benutzer

angezeigt, die die Rolle *Bearbeiter* innehaben.

Wenn die Aktionsfunktionen für alle Konversationstypen erstellt worden sind und alle Dialoge spezifiziert wurden, ist das System einsatzbereit. Die kompilierte Arbeitsprozeßkomponente wird beim Microsoft Transaction Server angemeldet und kann verwendet werden.

7.4 Zeitmanagement mittels eines BC-Addons

Eine Anforderung an das System ist es, nachvollziehen zu können, wieviel Zeit eine bestimmte Konversation bereits aktiv war. Dazu wird eine BC-Addon verwendet, das bei jedem Fortsetzen einer Konversation diese zur aktiven Konversation macht und somit die Zeitverfolgung aktiviert. Eine andere gerade aktive Konversation wurde hiermit inaktiv im Sinne der Zeitverfolgung. Über die externe Schnittstelle läßt sich für jede Konversation die bereits verbrauchte Zeit ermitteln. Dadurch erhält man einen Überblick, wie lange das Lösen einer Aufgabe gedauert hat. Dies ist zum Beispiel für die Aufwandsabrechnung hilfreich, die dem Aufgabensteller damit in Rechnung gestellt werden kann. Abbildung 7.5 zeigt die Übersicht eines Benutzer über die laufenden Konversationen und wieviel Zeit auf die jeweilige Aufgabe bereits verwendet worden ist.

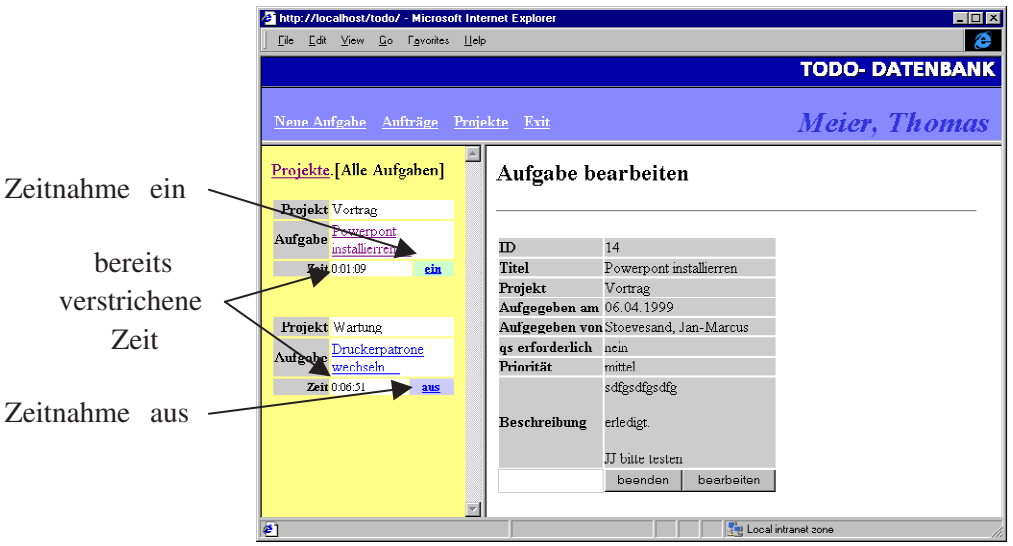


Abbildung 7.5: Zeiterfassung bei der Bearbeitung von Aufgaben

Kapitel 8

Zusammenfassung und Ausblick

Abschließend wird noch einmal zusammengefaßt, was in dieser Arbeit entwickelt und untersucht worden ist. In der Zusammenfassung wird erläutert, welche Anforderungen gestellt wurden und wie sie in der Praxis umgesetzt werden konnten. Im Ausblick wird auf Alternativen und neue Technologien hingewiesen, die ähnliches leisten. Er soll als Anregung für weitere Arbeiten auf diesem Gebiet dienen.

8.1 Zusammenfassung

In dieser Arbeit wurde ein Workflowsystem auf Basis des Microsoft Transaction Server entwickelt. An diesem System wird verdeutlicht, wie unter Verwendung von Komponententechnologie Arbeitsprozesse auf ein intranetbasiertes Informationssystem abgebildet werden können. Dies beinhaltet Aspekte wie Transaktionssicherheit, Visualisierung von Dialogen und Persistenz der Konversationen sowie das Führen von Sessions.

Es wurde gezeigt, daß die Transaktionssicherheit durch die Komponenten gewährleistet wird, da dies eine Eigenschaft ist, die die Komponenten inhärent mittels des Microsoft Transaction Servers zur Verfügung stellen.

Das Führen von Sessions ist notwendig, da das verwendete HTTP einen Erhalt von Informationen bei einem Seitenwechsel nicht vorsieht. Hier bietet der verwendete Webserver die entsprechenden Möglichkeiten.

Ein weiteres Ziel dieser Arbeit war es, eine Modellierungstechnik zu finden, die es erlaubt, ohne Modellwechsel von der Analyse bis zur Implementierung zu gelangen. Ein Problem bei der Analyse ist es, daß Arbeitsprozesse oft als serielle Folge von Aktionen und Ereignissen beschrieben werden. Vergleichbar mit einem Laufzettel, der Punkt für Punkt abgearbeitet wird. Andere Modelle wie etwa die *Maps, Tabs und Tix (MTX)* [Ziemer 98] haben sich hieran orientiert und behalten diese ablauforientierte Form bei. Will man aber einen konversationsorientierten Ansatz wählen, wie dies bei der Verwendung von *Business Conversations* geschieht, muß man die Analyse anders ansetzen und verliert die Information über den Ablauf eines Arbeitsprozesse. Dafür kann man aber Aussagen über die Kommunikationsstruktur innerhalb eines Arbeitsprozesses treffen, der aus den reinen Abläufen nicht ersichtlich ist.

Mit dem Modell, das in dieser Arbeit unter Verwendung von *Elementaren Objektsystemen* entwickelt worden ist, gelingt es, beide Ansätze zu verbinden. Es fungiert als Bindeglied zwischen der ablauforientierten Analyse und der konversationsorientierten Implementierung. Die verwendeten Objektnetze bilden die Abläufe eines Arbeitsprozesses ab. Diese werden in einem Systemnetz verarbeitet, das anzeigt, welche Konversationen zwischen welchen Partnern stattfinden. Dadurch kann man die Vorteile beider Modellierungs- und Analysevarianten verbinden.

8.2 Ausblick

Die in dieser Arbeit verwendeten Technologien beruhen zumeist auf Produkten der Firma Microsoft. Transaktionssichere Komponenten, sessionfähige Webserver, Transaktionskoordination mit Datenbankservern, etc. werden aber auch von anderen Plattformen unterstützt. *Enterprise Java Beans* [Vogel, Rangarao 99] stellen zum Beispiel einen anderen Ansatz dar, der ähnliches leistet. Im Vergleich zu den meist proprietären Standards, die in dieser Arbeit Verwendung finden, kann man hier ein größeres Spektrum an Alternativprodukten erwarten. Eine Untersuchung, wie diese Produkte geeignet sind, Arbeitsprozesse intranetfähig abzubilden, ist daher interessant. Aufgrund der Tatsache, daß zur Kommunikation neben RMI auch CORBA [Orfali, Harkey 99] Anwendung findet, bietet sich die Möglichkeit, Vergleiche der Leistungsfähigkeit von Protokollen wie RMI und CORBA im Vergleich zu DCOM anzustellen.

Anhang A

XSL-Beispiele

A.1 Beispiel 1

```
<xsl>

   <rule>
      <root/>
      <HTML>
         <BODY>
            <h1>Sonderpreise KW 2</h1>
            <h2>Diese Sonderpreise werden den Kunden gew&auml;hrt</h2>
<![CDATA[
<table border=2 bgcolor="#cccccc" bordercolor="white" cellspacing="0"
cellpadding="5" >
      <tr><th>Artikel<th>Sonderpreis (VK)<th>Preis (EK)</tr>
]]>
<children/>
<![CDATA[
</table>
]]>
         </BODY>
      </HTML>
   </rule>

   <rule>
      <target-element type="Artikel"/>
      <tr>
         <children/>
      </tr>
   </rule>
```

```
   <rule>
      <target-element type="Bezeichnung">
         <attribute name="lang" value="DE"/>
      </target-element>
      <td>
         <b><children/></b>
      </td>
   </rule>

   <rule>
      <element type="Artikel">
         <target-element type="VK"/>
      </element>
      <th>
         <i><children/></i>
      </th>
   </rule>

   <rule>
      <target-element type="Bezeichnung">
         <attribute name="lang" value="UK"/>
      </target-element>
      <empty/>
   </rule>

   <rule>
      <element type="Artikel">
         <target-element type="EK"/>
      </element>
      <th>
         <i><children/></i>
      </th>
   </rule>

</xsl>
```

A.2 Beispiel 2

```
<xsl>
   <rule>
      <root/>
      <HTML>
         <BODY>
```

```
            <center>
               <font size="+5" color="red">Bargains !!</font><br/>
               <font size="+3" color="blue">Don't wait too long</font>
               <hr/>
<![CDATA[
<table border=2 bgcolor="#cccccc" bordercolor="white" cellspacing="0"
cellpadding="5" >
      <tr><th>Product<th>Special Price</tr>
]]>
<children/>
<![CDATA[
</table>
]]>
            </center>
         </BODY>
      </HTML>
   </rule>

   <rule>
      <target-element type="Artikel"/>
      <tr>
         <children/>
      </tr>
   </rule>

   <rule>
      <target-element type="Bezeichnung">
         <attribute name="lang" value="UK"/>
      </target-element>
      <td bgcolor="green">
         <font color="white"><b><children/></b></font>
      </td>
   </rule>

   <rule>
      <element type="Artikel">
         <target-element type="VK"/>
      </element>
      <th bgcolor="#ff9999">
         <i>just <children/></i>
      </th>
   </rule>

   <rule>
```

```
      <target-element type="Bezeichnung">
         <attribute name="lang" value="DE"/>
      </target-element>
      <empty/>
   </rule>

   <rule>
      <element type="Artikel">
         <target-element type="EK"/>
      </element>
      <empty/>
   </rule>

</xsl>
```

A.3 Beispiel 3

```
<xsl>

   <define-script>
   <![CDATA[

   function displayall(p) {
      var i;
      var t=0;
      var ch;
      var output = "";

      output = "<tr><th>Artikel</th>"
      for(i=0;i<p.children.length;i++){
         if (t == 0) {
           output = output + "<td bgcolor=#bbffbb><font size=+2>"
         } else {
           output = output + "<td bgcolor=#ffbbbb><font size=+2>"
         }
         t=1-t;
         output = output + p.children.item("Artikel",i) _
                     .children.item("Bezeichnung",1).text;
         output = output + "</font></td>"
      }
      t=0;
      output = output + "<tr><th>Preis</th>"
      for(i=0;i<p.children.length;i++){
```

```
         if (t == 0) {
           output = output + "<th bgcolor=#bbffbb><font size=+2>"
         } else {
           output = output + "<th bgcolor=#ffbbbb><font size=+2>"
         }
         t=1-t;
         output = output + p.children.item("Artikel",i) _
                     .children.item("VK",0).text;
         output = output + "</th>"
      }

      return output;
   }

   ]]>
   </define-script>

   <rule>
      <root/>
      <HTML>
         <BODY>
            <center>
               <font size="+5" color="#008800">Aufgepasst<br/> _
                     zugefasst ...</font><br/>
               <font size="+2" color="blue">... haufenweise Sonder_
                     preise</font>
               <hr/>
<![CDATA[
<table border=2 bgcolor="#cccccc" bordercolor="white" _
      cellspacing="0"
cellpadding="5" >
]]>
         <eval>displayall(this)</eval>
<![CDATA[
</table>
]]>
            </center>
         </BODY>
      </HTML>
   </rule>

</xsl>
```

Anhang B

Dialogbeschreibung

B.1 Syntax in Backus-Naur-Form

$$
\begin{aligned}
\text{DIALOG} &::= <Dialog> \text{ TITEL FORM } </Dialog> \\
\text{TITEL} &::= <Titel> Dialogtitel </Titel> \\
\text{FORM} &::= <Form> \text{ DATA* [ACTION] } </Form> \\
\text{ACTION} &::= <Action> \text{ DO* } </Action> \\
\text{DO} &::= <Do> Aktion </Do>
\end{aligned}
\tag{B.1}
$$

B.2 Syntax als Document Type Defnintion (DTD)

```
<!DOCTYPE Dialog [
<!ELEMENT Dialog (Titel,Form)>
<!ELEMENT Form (Data*,Action?)>
<!ELEMENT Data (#PCDATA)*>
<!ATTLIST Data
 Type (Display|Input|Textarea|Select)
 Titel (#PCDATA)
>
<!ELEMENT Action (do+)>
<!ELEMENT do (#PCDATA)*>
]>
```

B.3 Beispiel für eine XSL-Datei zur Darstellung von Dialogen

```
<xsl>

<rule>
  <root/>
    <html>
      <head>
        <select-elements>
          <target-element type="Redirect"/>
        </select-elements>
        <select-elements>
          <target-element type="Caption"/>
        </select-elements>
      </head>
      <body bgcolor="white">
        <select-elements>
          <target-element type="Title"/>
        </select-elements>
        <select-elements>
          <target-element type="Form"/>
        </select-elements>
      </body>
    </html>
</rule>

<rule>
<target-element type="Title"/>
  <h2><children/></h2>
  <hr/>
</rule>

<rule>
<target-element type="Redirect"/>
  <META HTTP-EQUIV="refresh" CONTENT='= getAttribute("seconds")  \
    + "; URL=" + getAttribute("url")' />
</rule>

<rule>
<target-element type="Caption"/>
  <title><children/></title>
</rule>
```

```
<rule>
<target-element type="Form"/>
  <form action="execute.asp" method="post">
    <![CDATA[
    <table border=2 bgcolor="#cccccc" bordercolor="white"   \
        cellspacing="0" cellpadding="0" >
    ]]>
      <children/>
    <![CDATA[
    </table>
    ]]>
  </form>
</rule>

<rule>
<element type="Form">
  <element type="Action">
    <target-element type="Do"/>
  </element>
</element>
      <input type="submit" name="BCAction"  \
             value="=this.children.item(0).text" />
</rule>

<rule>
<element type="Form">
  <target-element type="Action"/>
</element>
<![CDATA[
  <tr>
    <td bgcolor="white">
    </td>
    <td align="center" bgcolor="white">
]]>
      <children/>
<![CDATA[
    </td>
  </tr>
]]>
</rule>

<rule>
<element type="Form">
```

```
  <target-element type="Data">
  <attribute name="Type" value="hidden"/>
  </target-element>
</element>
  <input    type="hidden"
            name='=getAttribute("name")'
            value='=getAttribute("value")' />
</rule>

<rule>
<element type="Form">
  <target-element type="Data">
  <attribute name="Type" value="Input"/>
  </target-element>
</element>
  <tr>
    <td>
      <b><eval>getAttribute("Titel")</eval></b>
    </td>
    <td>
      <input
            size='=getAttribute("size")'
            maxlength='=this.children.item(0).getAttribute("size")'
            type="text"
            value="=this.children.item(0).text"
            name="='BC_' + this.children.item(0).tagName" />
    </td>
  </tr>
</rule>

<rule>
<element type="Form">
  <target-element type="Data">
  <attribute name="Type" value="Display"/>
  </target-element>
</element>
  <tr>
    <td>
      <b><eval>getAttribute("Titel")</eval></b>
    </td>
    <td>
      <children/>
    </td>
  </tr>
```

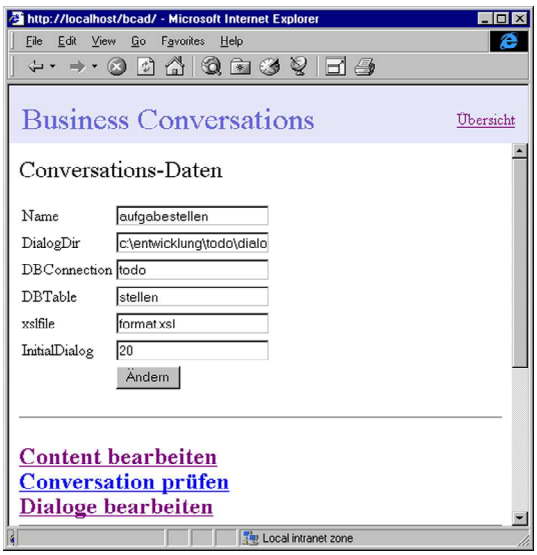
```
</rule>

<rule>
<element type="Form">
  <target-element type="Data">
  <attribute name="Type" value="Textarea"/>
  </target-element>
</element>
  <tr>
    <td valign="top">
      <b><eval>getAttribute("Titel")</eval></b>
    </td>
    <td>
      <textarea name="='BC_' + this.children.item(0).tagName"
                cols='=getAttribute("cols")'
                rows='=getAttribute("rows")' >
      <children/>
      </textarea>
    </td>
  </tr>
</rule>

<rule>
<element type="Form">
  <target-element type="Data">
  <attribute name="Type" value="Select"/>
  </target-element>
</element>
  <tr>
    <td valign="top">
      <b><eval>getAttribute("Titel")</eval></b>
    </td>
    <td>
      <select  name="='BC_' + this.children.item(0).tagName">
        <children/>
      </select>
    </td>
  </tr>
</rule>

<rule>
<element type="Data">
  <target-element type="Option">
  <attribute name="selected" value=""/>
```

```
  </target-element>
</element>
      <option value='=getAttribute("value")' selected="">
        <children/>
      </option>
</rule>

<rule>
<element type="Data">
  <target-element type="Option"/>
</element>
      <option value='=getAttribute("value")' >
        <children/>
      </option>
</rule>

</xsl>
```

Anhang C

Beispielapplikation

C.1 Eingabe der Daten für die Konversationsspezifikation *aufgabestellen*



C.2 Aktionsfunktion der Konversationsspezifikation *bearbeiten*

```
Sub action(Optional ByRef parent As Object)
  Dim currentId As Integer
  Dim action As String
  Dim data As Collection
  Dim id As Integer
  Dim timetrace As Object

  On Error GoTo errorhandler:

  currentId = parent.m_currentDialogID

  If m_objCtx Is Nothing Then
    action = ""
  Else
    action = m_objCtx("Request")("BCAction")
  End If


  Select Case LCase(currentId)
Case 40
    Select Case action
      Case "erledigt"
        ' wie oben ausführlich berichtet, werden hier Daten zum Austausch
        ' mit einer anderen Conversation vorbereitet
        Set data = New Collection
        data.Add parent.GetData("Beschreibung"), "Beschreibung"
        data.Add parent.m_PerformerId, "ErledigtVon"
        data.Add Format(Date, "dd.mm.yyyy"), "ErledigtAm"

        ' hier kommt endlich mal ein join. Bedeutet, da diese C. ja von einer
        ' anderen C. von Typ "aufgabestellen" mit dem Flag INIT_WAIT gestartet
        ' wurde, hängt diese solange fest, bis wir sie hier mit join() wieder
        ' zum leben erwecken
        parent.Join data

        ' das kennen wir schon ...
        parent.setCurrentDialog 18
        Set data = Nothing

        ' wieder mal die externen tools ...
        Set timetrace = m_objCtx.CreateInstance("bcaddons.timetrace")
        timetrace.Deactivate parent.m_PerformerId, parent.m_id
```

```
        Set timetrace = Nothing

      Case "weiterbearbeiten"
parent.setCurrentDialog 30
    End Select
  Case 17, 48 ' bearbeiten
    Select Case action
      Case "ok"
            ' s.o.
            parent.UpdateDialogData

            ' hier wird die ID des Bearbeiters ausgelesen
            id = CInt("0" & m_objCtx("Request")("BC_Performer"))

            ' wurde der Bearbeiter auf einen anderen Namen gestellt, geschieht
            ' folgendes ...
            If id <> parent.m_PerformerId Then
              ' ... es wird eine Neue Conversation desselben Typs mit einer
              ' Kopie aller Daten erstellt. Das ist sozusagen ein kompletter
              ' Clone der eigenen C. Ethisch allerdings nicht so bedenklich wie
              ' beim Schaf Dolly.
              ' Wichtig ist hierbei INIT_DELEGATE. Dadurch übernimmt die C.
              ' alle Rechte und Pflichten von dieser C. z.B. ist die neue C.
              ' jetzt dafür zuständig evtl. schlafende Conversationen des Typs
              ' aufgabestellen mittels join() wieder aufzuwecken.
              parent.InitiatePeer id, INIT_DELEGATE

              ' jetzt lassen wir den Clone die Arbeit machen und legen uns
              ' wieder hin. Also einfach einen finalen Dialog aufrufen ...
              parent.setCurrentDialog 49 ' finished

              ' Zeitverfolgung abschalten
              Set timetrace = m_objCtx.CreateInstance("bcaddons.timetrace")
              timetrace.Deactivate parent.m_PerformerId, parent.m_id
              Set timetrace = Nothing

            Else
              ' wie gehabt
              parent.setCurrentDialog 30
            End If ' if Request ...
    End Select
  Case 30 ' anzeige
    Select Case action
      Case "bearbeiten"
```

```
        parent.setCurrentDialog 48
      Case "beenden"
          ' mal wieder daten für den Austausch vorbereiten
          ' hier sieht man aber sehr schön, daß man alles mögliche
          ' an eine neue Conversation übergeben kann. Also nicht so wie
          ' bisher, wo nur eigene contents übergeben worden sind.
          Set data = New Collection
          data.Add parent.GetData("Beschreibung"), "Beschreibung"
          data.Add parent.m_PerformerId, "ErledigtVon"
          data.Add Format(Date, "dd.mm.yyyy"), "ErledigtAm"

          ' so'n join haben wir oben schon gesehen. Erklärt sich aber
          ' irgendwie auch von alleine.
          parent.Join data
          Set data = Nothing

          ' und ab in den nächsten Dialog
          parent.setCurrentDialog 18
    End Select
  End Select

  m_objCtx.SetComplete

  Exit Sub

errorhandler:
  App.LogEvent "todo.aufgabe.action : " & Err.Description
  Err.Raise Err.Number, "todo.aufgabe.action", Err.Description

End Sub
```

Literaturverzeichnis

Appleman 99: Appleman, Daniel. *COM/ActiveX Komponenten. Mit Visual Basic 6 entwickeln.* Addison Wesley Longmann, Massachusetts, 1st. edition, 1999.

Bortniker et al. 98: Bortniker, Matthew, Maharry, Dan, und Federov, Alex. *ASP 2.0 Programmer's Reference.* Wrox Press, 1st. edition, 1998.

Bradley 98: Bradley, Neil. *The XML Companion.* Addison Wesley Longmann, Massachusetts, 1 edition, 1998.

Chappell 98: Chappell, David. *Einstieg in ein neues Programmiermodell.* Microsoft System Journal, Jg. 98, März 1998, Nr. 3, S. 70–74.

Coulouris et al. 94: Coulouris, George, Dollimore, Jean, und Kindberg, Tim. *Distributed Systems. Concepts and Design.* Addison Wesley Longmann, Massachusetts, 2nd edition, 1994.

December, Ginsburg 95: December, John und Ginsburg, Mark. *HTML & CGI Unleashed.* Sams.net Publishing, 1st. edition, 1995.

Fedorchek, Rensin 97: Fedorchek, Andrew M. und Rensin, David K. *ASP: The Power Guide to Developing Server-Based Web Applications.* IDG Books Worldwide Inc, 1st. edition, 1997.

Johannisson 97: Johannisson, Nico. *Eine Umgebung für mobile Agenten: Agentenbasierte verteilte Datenbanken am Beispiel der Kopplung autonomer Internet Website Profiler.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, 1997.

Kruse 97: Kruse, Wolfgang. *Historienmanagement für kooperative Aktivitäten.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, 1997.

Lockemann, Schmidt 93: Lockemann, P.C. und Schmidt, J.W. (Hrsg.). *Datenbankhandbuch.* Springer Verlag Berlin Heidelberg, 1 edition, 1993.

Matthes 97: Matthes, Florian. *Mobile Processes in Cooperative Information Systems.* In: *Proceedings STJA'97, Erfurt.* Springer-Verlag, Berlin u.a., September 1997.

Matthes 98: Matthes, Florian. *Business Conversations. A High-Level System Model for Agent Coordination.* 1998.

Mic 95: Microsoft Corporation. *The Component Object Model Specification (http://www.microsoft.com/)*, 1995.

Mic 98: Microsoft Corporation. *XSL Tutorial (http://www.microsoft.com/xml/xsl/)*, 1998.

Musciano, Kennedy 94: Musciano, Chuck und Kennedy, Bill. *HTML. The Definitive Guide.* O'Reilly & Associates, Inc., 1st edition, 1994.

Orfali, Harkey 99: Orfali, Robert und Harkey, Dan. *Client/Server Programming with Java and CORBA, Second.* John Wiley & Sons, 2nd edition, 1999.

Pattison 98: Pattison, Ted. *MTS-Transaktionen über das Web.* Microsoft System Journal, Jg. 98, Mai 1998, Nr. 5, S. 76–87.

Petkovic 96: Petkovic, Dusan. *Microsoft SQL Server 6.5. Das Datenbanksystem im Back-Office.* Addison Wesley Longmann, Bonn, 1st. edition, 1996.

Reed et al. 97: Reed, Dave, Trewin, Tracey, und Tomsen, Mai-Jan. *Der Microsoft Transaction Server.* Microsoft System Journal, Jg. 97, Juni 1997, Nr. 6, S. 24–40.

Richtsmeier 97: Richtsmeier, Ingo. *Vergleich objekt- und agentenbasierter verteilter Datenbankprogrammierung am Beispiel der Kopplung autonomer Internet Website Profiler.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, 1997.

Rumbaugh et al. 93: Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., und Lorensen, W. *Objektorientiertes Modellieren und Entwerfen.* Carl Hanser Verlag München Wien, 1993.

Thiagarajan 87: Thiagarajan, P.S. *Elementary Net Systems.* In: Brauer, W., Reisig, W., und Rozeberg, G. (Hrsg.). *Petri Nets: Central Models and their Properties.* Springer Verlag, Berlin, 1987.

Träger 98: Träger, Michael. *Der Microsoft Transaction Server in der Praxis.* Microsoft System Journal, Jg. 98, Juni 1998, Nr. 6, S. 89–95.

Valk 98a: Valk, Rüdiger. *Concurrency in Communicating Object Petri Nets.* 1998.

Valk 98b: Valk, Rüdiger. *Petri Nets as Token Objects.* In: *Proceedings 19th International Conference, ICATPN'98. Lisbon, Portugal.* Springer-Verlag, June 1998.

Vogel, Rangarao 99: Vogel, Andreas und Rangarao, Madhavan. *Programming with Enterprise JavaBeans, JTS, and OTS.* John Wiley & Sons, 1st. edition, 1999.

Walsh 98: Walsh, Norman. *A Guide to XML.* XML.XOM, http://www.xml.com, Jg. 98, 1998.

Walther 98: Walther, Stephen. *Active Server Pages.* Sams Publishing, 1st. edition, 1998.

Wegner 98: Wegner, Holm. *Objektorientierter Entwurf und Realisierung eines Agentensystems für kooperative Internet-Informationssysteme.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, 1998.

Ziemer 98: Ziemer, Stephan. *Eine transportorientierte Workflow-Sprache: Definition, Anwendung und Bewertung.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, 1998.