Johann Wolfgang Goethe-Universität
Frankfurt am Main
Fachbereich Informatik und Biologie
Institut für Informatik

Diplomarbeit

Kooperative Verarbeitung strukturierter Textdokumente mit adaptiver Konfliktgranularität

eingereicht bei:

Prof. Dr. Oswald Drobnik (Erstgutachter)
Professur für Architektur und Betrieb Verteilter Systeme
und
Prof. Dr. Jörg M. Haake (Zweitgutachter)
Professur Praktische Informatik VI - Verteilte Systeme
FernUniversität Gesamthochschule Hagen

vorgelegt von:

Laura Dietz
Matrikel-Nr: 1262915

Abgabedatum:

21. September 2002

Erklärung:

Ich, Laura Dietz, versichere, daß ich die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

_____

Laura Dietz

Danksagung

Für ihre instensive Betreuung möchte ich zunächst Herrn Prof. Dr. Jörg M. Haake, der mir diese hochinteressante Aufgabe stellte, und Herrn Prof. Dr. Oswald Drobnik, der es mir erlaubte, diese Aufgabe für eine Diplomarbeit zu verwenden, danken.

Ebenfalls herzlichen Dank an die Herren Dr. Daniel A. Tietze und Herrn Michael Lauer für ihre fachliche Beratung.

Weiterhin gilt mein Dank Steffen Spendel für seine orthographischen Hilfestellungen, Sebastian Radestock für seine konstruktive Kritik und Sebastian Pape für seine Literaturhinweise und die fachlichen Diskussionen mit ihm.

Schließlich möchte ich meinen Eltern Dr. Eberhard Dietz und Rotraud Dietz-Richter danken, da sie mich während meiner Diplomarbeit sowie meines gesamten Studiums stets ermuntert und unterstützt haben.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Der Anbeginn der modernen Menschheit wird von vielen Anthropologen mit der Sesshaftwerdung der Hominiden festgelegt. Seit dieser Zeit zieht der Mensch die Vorteile aus der gemeinsamen Bewältigung von Problemen. Die dabei entstehenden Meinungsverschiedenheiten resultieren in verschiedensten Formen von Konflikten. Oftmals handelt es sich dabei um kleinere Mißverständnisse, die dem Menschen oft gar nicht als Konflikt bewußt werden. In anderen Fällen werden sie ausdiskutiert und danach ein Kompromiß geschlossen. Ist eine Zusammenarbeit derart konfliktreich, daß die Arbeit der einzelnen Gruppenmitglieder für sich genommen effizienter ist, als in Zusammenarbeit, so löst sich die Gruppe auf.

Eine Form der Gruppenarbeit ist das gemeinsame Erstellen von Dokumenten. Ein gemeinsam erstelltes Dokument bündelt das Wissen der Gruppe und macht es so anderen Personen zugänglich. Neben dem Ergebnis ist häufig auch ein anderer Aspekt wichtig: Versucht eine Gruppe ein Thema auszuarbeiten, erfahren die Gruppenmitglieder von dem Wissen, der Einstellung und den Argumenten der anderen Teilnehmer und sind weiterhin dazu gezwungen, einen Konsenz zu bilden, der eine intensive Auseinandersetzung mit dem Thema erfordert.

Im beruflichen Alltag werden oftmals gemeinsame Dokumente ohne spezielle Unterstützung durch den Computer erstellt. Dazu kann eine Person zum Schriftführer ernannt werden, die die Gedanken der anderen Teilnehmer bündelt und festhält. In anderen Fällen wird das Dokument in einer Laufmappe herumgereicht und reih um modifiziert.

	selbe Zeit	verschiedene Zeit
gleicher Ort	Schriftführer	Laufmappe
verschiedener Ort	—	Briefverkehr

Tabelle 1: Verfahren zur gemeinsamen Dokumentenerstellung ohne Rechnerunterstützung

Diese Verfahren sind häufig langwierig, da auf eintreffende Dokumente gewartet werden muß. Versucht man Teile zu parallelisieren, indem Aufgaben verteilt werden, so können Redundanzen entstehen, da keiner der Beteiligten eine vollständige Version besitzt. Weiterhin ist es für mehrere Personen schwierig auf dem gleichen Blatt Papier zu schreiben, da Papier, Schreibwerkzeug und menschliche Körper einen gewissen Raum benötigen. Außerdem ist das Einfügen in bestehenden Text auf Papier ein mühsamer Prozeß.

Diese Nachteile kann der Computer ausgleichen, da durch ihn die aktuelle Dokumentversion jederzeit abrufbar ist. Weiterhin existiert das Dokument nur virtuell, so daß physikalische Ausprägungen keine Rolle spielen.

Sind die Gruppenmitglieder an verschiedenen Orten, so ist ein Dokumentenaustausch ohne Rechner nur mittels Briefverkehr oder Faxgerät möglich. Da Briefe eine Spezialform von Laufmappen darstellen, steigen die Wartezeiten durch die längeren Transportvorgang noch weiter an. Eine gleichzeitige Bearbeitung des aktuellen Dokuments ist ausgeschlossen.

Der Vorteil des Computers in dieser Situation liegt darin, daß die Daten über das Internet in sehr kurzer Zeit über weite Distanzen transferiert werden können. Der Unterschied zu Sitzungen in denen alle Teilnehmer am gleichen Ort sind, liegt darin, daß alle Teilnehmer ausschließlich über Computer und Telefon kommunizieren müssen.

In der Einschränkung auf den Rechner als Kommunikationsmedium liegt eine der größten Schwächen rechnergestützter Gruppenarbeit: Zum Auflösen von Konflikten in alltäglichen Situationen benötigen Menschen eine große Menge an Informationen. Werden diese durch den Computer nicht bereitgestellt oder in unverständlicher Form präsentiert, so wird der eigentliche Arbeitsfluß durch Mißverständnisse und zusätzlichen Koordinationsaufwand gebremst.

Die zentrale Frage dieser Diplomarbeit ist, wie ein Rechner gleichzeitiges, koordiniertes Arbeiten unterstützen kann und dabei den zusätzlichen Aufwand, den Benutzer zur Koordination und Auflösen von Konflikten aufwenden müssen, möglichst gering hält.

1.2 Aufgabenstellung

Ein grundlegendes Problem bei der Realisierung kooperativer Systeme (Groupware) ist das Gewährleisten einer möglichst hohen Nebenläufigkeit (Concurrency) von den Aktionen verschiedener Bearbeiter beim gemeinsamen Bearbeiten strukturierter Datenräume. Hierbei stehen sich zwei Anforderungen gegenüber: (1) Maximierung der unbeeinflußten, parallelen Arbeit an demselben Datenraum vs. (2) Maximierung der gegenseitigen Koordination, um z.B. widersprüchliche Modifikationen vermeiden zu können oder Synergie zwischen Bearbeitern nutzen zu können. Um während der Arbeit nicht durch unnötige Informationen über die Aktivitäten anderer Benutzer unterbrochen zu werden, gilt es die notwendigen Informationen für dem Benutzer aufzubereiten und in möglichst wenig störender Weise darzustellen.

In dieser Diplomarbeit soll eine Lösung entwickelt werden, die beide Anforderungen möglichst gut erfüllt. Als Beispiel für ein kooperatives System wird eine kollaborative Textverarbeitungssoftware (ein sog. Shared Texteditor) gewählt. Dieser Texteditor soll einfache Formatierungsmöglichkeiten wie Auswahl der Schriftart, -größe und -farbe anbieten. Ziel dieser Diplomarbeit ist der Entwurf und die Entwicklung einer kollaborativen Textverarbeitungssoftware, die einerseits die Nebenläufigkeit von parallelen Aktionen auf dem gemeinsamen strukturierten Dokument maximiert, und andererseits eine möglichst gute Koordi-

nation der parallelen Bearbeiter ermöglicht. Die in der Arbeit entwickelte Lösung soll auf Basis des DyCE Frameworks für komponentenbasierte kollaborative Umgebungen und der Java GUI Bibliothek Swing realisiert werden. Zum Nachweis der Funktionalität soll ein mit den Standardmechanismen von DyCE realisierter kooperativer Texteditor mit dem in dieser Arbeit entwickelten kooperativem Texteditor bezüglich des Ausmaßes an Nebenläufigkeit und des Ausmaßes an Koordination zwischen den Bearbeitern verglichen werden.

Der in dieser Arbeit entwickelte Ansatz soll nach Möglichkeit auf die kooperative Bearbeitung von Datenräumen mit allgemeiner Graphenstruktur erweitert werden können.

1.3 Gliederung

In dieser Arbeit werden Ansätze zur kooperativen Verarbeitung strukturierter Textdokumente mit adaptiver Konfliktgranularität besprochen.

Kapitel 2 gibt zunächst eine Übersicht über das Thema „rechnergestützte Gruppenarbeit“. Dazu werden allgemeine Arbeitsparadigmen und Groupwarekategorien besprochen und diese anschließend anhand von Beispielen diskutiert. Weiterhin werden anhand eines Szenarios die Anforderungen der Benutzer an eine kooperative Textverarbeitung erarbeitet und mit aus der Literatur bekannten Ansätzen verglichen. Abschließend wird in Kapitel 2 das Groupware-Framework DyCE beschrieben, auf dem (laut Aufgabenstellung) der Prototyp implementiert werden soll.

Kapitel 3 beschäftigt sich mit der Optimierung der Nebenläufigkeitskontrolle in DyCE. Dazu werden verschiedene Verfahren zur Adaption der Konfliktgranularität vorgestellt, wie die Fragmentierung, das Absperren von Bereichen und eine Methode, die auf der freiwilligen Koordination der Benutzer untereinander basiert.

In Kapitel 4 wird eine Datenstruktur vorgestellt, die dazu geeignet ist, Daten innerhalb von DyCE zu modellieren. Es wird skizziert, wie diese Struktur in XML repräsentiert werden kann. Außerdem wird gezeigt, wie die Verfahren aus Kapitel 3 in dieser Struktur angewendet werden können.

Kapitel 5 spezifiziert diese Datenstruktur mit Methoden der Informatik und stellt Operationen vor, die eine konsistente Manipulierung der Daten ermöglichen. Weiterhin behandelt Kapitel 5 implementierungstechnische Problemstellungen bei der Umsetzung der Datenstruktur auf die DyCE-Plattform.

Kapitel 6 erläutert eine Anwendung der Datenstruktur auf kooperative textverarbeitende Systeme und die Umsetzung der Benutzungsschnittstelle. Abschließend wird gezeigt, in welcher Weise der Anforderungskatalog erfüllt wird.

Eine Zusammenfassung wird in Kapitel 7 gegeben. Abschließend werden in der Arbeit offengebliebene Fragen wiederholt und neue Ausblicke gegeben.

1.4 Schreibweisen

Im folgenden Text werden Definitionen oft durch das vorgestellte Wort „Definition“ zusammen mit dem Namen in runden Klammern gekennzeichnet. Das Ende einer Definition ist durch einen Absatz gekennzeichnet.

Zur besseren Lesbarkeit werden neu vorgestellte Begriffe unterstrichen.

Bei boole’schen Ausdrücken werden die Begriffe wahr, true und erfüllt, ebenso wie die Negationen falsch, false und nicht erfüllt, synonym verwendet. Steht ein boole’scher Ausdruck in einem Wenn-Dann-Kontext, so muß dieser erfüllt sein, damit die Aussage gilt. An manchen Stellen wird durch ein vorgestelltes Ausrufezeichen die Erfüllung der Negation symbolisiert.

Im Kapitel zur Implementierung wird oft anstatt des einfachen Gleichheitszeichens „=“, das doppelte „==“ vergleichende Gleichheitszeichen oder das Zuweisungssymbol „:=“ verwendet.

Weiterhin werden folgende mathematische Symbole verwendet: $\forall$ (für alle); $\exists$ (es gibt mindestens ein); $\in$, $\ni$ (ist Element von); $\notin$ (ist nicht Element von); $\subseteq$, $\supseteq$ (ist Teilmenge von); $\subset$, $\supset$ (ist echte Teilmenge von); $\cap$ (Schnitt zweier bzw. mehrerer Mengen); $\cup$ (Vereinigung zweier bzw. mehrerer Mengen); $\varnothing$, {} (leere Menge); $\circ$ (Konkatenation von Zeichenketten oder Hintereinanderausführung von Operationen); x (Kreuzprodukt zweier Mengen).

Funktionsnamen werden durch Kleinbuchstaben gekennzeichnet, Namen von Mengen beginnen mit einem großen Buchstaben.

Variablen, Methoden, Klassen, Konstanten und Paketnamen werden in der Java-typischen Schreibweise notiert. Dazu beginnen Variablen- und Methodennamen mit einem Kleinbuchstaben, an den sich weitere Begriffe ohne Leerzeichen mit großem Buchstaben anschließen. Ähnliches gilt für Klassennamen, nur daß diese mit einem Großbuchstaben beginnen. Namen von Konstanten bestehen nur aus Großbuchstaben, mehrere Begriffe werden durch Unterstreichungszeichen getrennt. Paketnamen bestehen hingegen nur aus Kleinbuchstaben. Eine Methode ist eindeutig definiert durch <Paketpfad>.<Klassenname>.<Methodenname>(<Typ des 1. Parameters>, <Typ des 2. Parameters>, ...). Typ bezeichnet dabei eine Klasse oder primitiven Typ wie int für Integer oder boolean für Boole’sche Ausdrücke. Der Paketpfad setzt sich aus den Paketnamen, die durch Punkte getrennt werden, zusammen. Beispiel: java.util.Vector.add(int, Object). Ist klar welche Klasse gemeint ist, kann der Paketpfad weggelassen werden.

2 Grundlagen und alternative Ansätze

In diesem Kapitel sollen die Grundlagen zu Groupware und dem Forschungsfeld „rechnergestützte Gruppenarbeit“ erarbeitet werden. Die verschiedenen Kategorien von Groupware werden anhand von Systemen, die in der Praxis eingesetzt werden, diskutiert. Um die Systeme bewerten zu können, wird ein Einblick in allgemeine Arbeitsweisen von Gruppen gegeben.

Die Anforderungen an das zu entwickelnde System werden anhand eines fiktiven Szenarios herausgearbeitet und mit aus der Literatur bekannten Ansätzen verglichen.

Anschließend wird das Groupware-Framework DyCE näher vorgestellt und die Defizite und Vorteile durch einen Vergleich mit den Anforderungen verdeutlicht.

Zuletzt wird der in DyCE existierende Texteditor an den Anforderungen gemessen.

2.1 Was ist rechnergestützte Gruppenarbeit?

Die weltweit zunehmende Verbreitung von Computern und deren Vernetzung macht es möglich, daß Menschen, die nicht am gleichen Ort sind, miteinander arbeiten können. Dies nennt man rechnergestützte Gruppenarbeit. Andere Begriffe dafür sind u.a. Computer Supported Cooperative Work (CSCW), Computer Conferencing oder Computer-Mediated Communication (Johannson 1991, zit.n. [BS98]).

Die Rolle des Computers bzw. der entsprechenden Programme besteht aus der Unterstützung der Kommunikation zwischen Gruppenmitgliedern, der gegenseitigen Koordination und dem Zugang zu gemeinsamen Artefakten.

Die unterstützende Software nennt man Groupware. Ellis, Gibbs und Rein definieren Groupware als „computer-based systems that support groups of people engaged in a common task (or goal) and that provide an interface to a shared environment.“ (Ellis, Gibbs und Rein 1991, zit.n. [BS98]) (computerbasierte Systeme, die eine Gruppe von Personen unterstützen, die an einer gemeinsamen Aufgabe (oder Ziel) arbeiten und eine Schnittstelle zu einem gemeinsamen Arbeitsraum bereitstellen.)

Da viele Aufgaben in jeder Groupware gemeistert werden müssen, werden Groupwareframeworks bereitgestellt. Diese übernehmen die Verwaltung der Benutzer, Verteilung des gemeinsamen Arbeitsraums und sorgen für die Konsistenzerhaltung der Daten bei paralleller Modifikation.

Definition (Sitzung, Gruppe, Server, Client, Teilnehmersystem): In einem allgemeinen Groupwareszenario gibt es eine Sitzung, an der eine Gruppe von Benutzern teilnimmt. Jeder Teilnehmer arbeitet an jeweils einem Computer. Die Computer sind über das Netzwerk mit einem Server verbunden. Den Computer eines Benutzers nennt man Client. Die Einheit aus einem Benutzer zusammen

mit seinem Client – genauer gesagt, dem Benutzerprozeß – nennt man Teilnehmersystem[1].

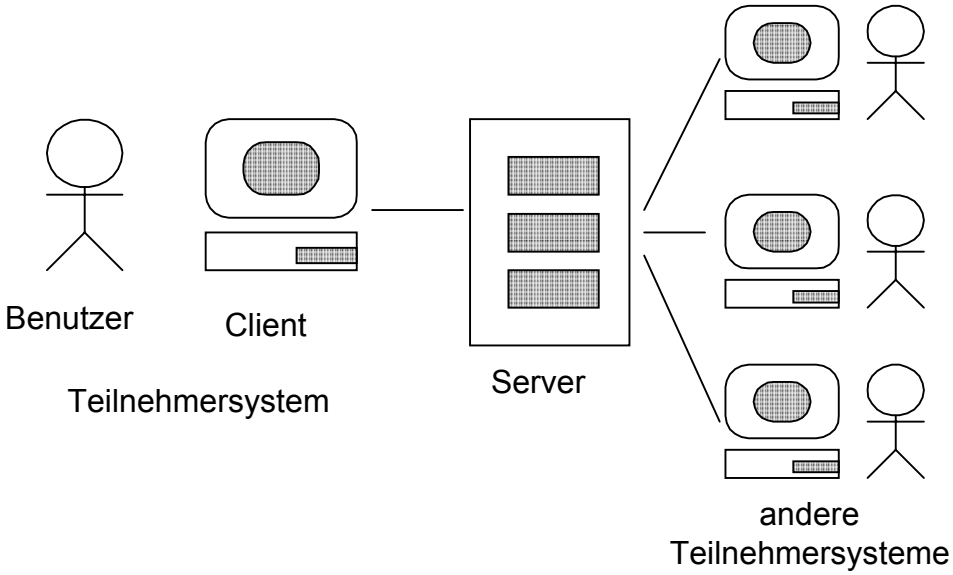


Abbildung 1: Allgemeines Groupwareszenario

In einigen Fällen existiert kein ausgezeichneter Server – die Clients sind direkt miteinander verbunden. In anderen Fällen gibt es mehrere Server.

Es ist möglich, daß sich mehrere Benutzer den gleichen Client teilen oder daß ein Benutzer mehrere Clients verwendet. In beiden Fällen spricht man von mehreren Teilnehmersystemen. Für Groupwaresysteme ist häufig der Begriff Teilnehmersystem entscheidender als der des Benutzers oder des Clients.

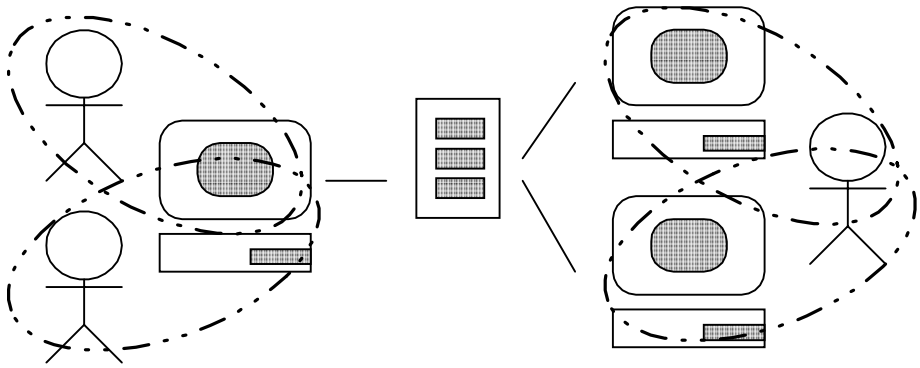


Abbildung 2: Ein Szenario mit vier Teilnehmersystemen.
Jede Kombination aus Benutzer und Client repräsentiert ein Teilnehmersystem.

[1] englisch: site

Anhand der Kommunikationsstruktur der Gruppe unterscheidet man verschiedene Systeme, die der Zusammenarbeit zuträglich sind [BS98]. Informationssysteme dienen lediglich der Verteilung von Informationen. Der Informationsfluß ist hier unidirektional gerichtet. Ein Beispiel für Informationssysteme ist das Worldwide Web. Koordinationssysteme lassen Absender und Empfänger im direkten Kontakt stehen, um Informationen und Aktionen zu koordinieren. Arbeiten die Teilnehmer auf ein gemeinsames Ziel hin und unterhalten sie unregelmäßige Interaktion, spricht man von Kollaboration. Jedoch erst bei häufiger, regelmäßiger Interaktion und einer Unterordnung der persönlichen Ziele unter die gemeinsame Aufgabe spricht man von Kooperation. Nur Systeme, die die Kooperation unterstützen, heißen nach [BS98] Groupware.

Herkner schreibt „Es läßt sich [...] zeigen, daß bei Personen mit ähnlichen Einstellungen (mit ähnlichen Wertesystemen) im allgemeinen reibungslose und für beide Teile erfreuliche Interaktionsmuster zu erwarten sind.“ [Her81] Damit die Gruppe miteinander kommunizieren kann, ohne daß dabei Mißverständnisse entstehen, ist also ein gemeinsamer Hintergrund[2] nötig. Dieser besteht aus einer gemeinsamen Terminologie, gemeinsamen Normen und gemeinsamen Vorstellungen. Gemeinsam heißt in diesem Kontext, daß es nicht ausreicht, wenn zwei Teilnehmer einen bestimmten Begriff kennen, sondern er muß für beide auch die gleiche Bedeutung haben. Groupware sollte es daher den Teilnehmern erlauben, eine gemeinsame Arbeitsumgebung und einen gemeinsamen Kontext aufzubauen. [DB]

2.2 Arbeitsparadigmen

Aufgabe der Groupware ist es, die kooperative Arbeit sinnvoll zu unterstützen. Doch nach welchen Paradigmen arbeiten Personen zusammen?

Die einfachste Weise besteht darin, daß nur eine Person zur Änderung von Daten berechtigt ist. Die anderen Gruppenteilnehmer diskutieren mögliche Änderungen und diktieren sie ihm, wobei der Autor gleichzeitig Schiedsrichter ist und im Konfliktfall entscheidet, welche Änderungen er übernimmt und welche nicht. (Schriftführerverfahren[3]). [PMB96] Dieses Verfahren ist aus dem Schulalltag weitläufig bekannt, wobei der Lehrer die Gedanken der Schüler aufnimmt und an der Tafel notiert. Voraussetzung ist, daß alle Beteiligten gleichzeitig bei der Sitzung anwesend sind.

Eine zweite Möglichkeit besteht im iterativen Verbessern[4] [MR96] der Daten. Das erste Gruppenmitglied beginnt mit der Modifizierung oder Editierung von Daten, nach Beendigung seiner Arbeit geht das Änderungsrecht auf das nächste Gruppenmitglied über. Dieser Vorgang wiederholt sich solange, bis alle

[2] englisch: Common Ground

[3] englisch: Scribe

[4] englisch: Turn Taking oder Draft Passing

Gruppenmitglieder mit dem Ergebnis einverstanden sind. Ohne Computer wird dieses Verfahren meist mit Laufmappen implementiert, die von Person zu Person weitergereicht werden. Es ist ein zeitlich weitgehend entkoppeltes Verfahren das durch die Sequentialisierung einen längeren Zeitraum benötigt.

Ein drittes Arbeitsparadigma ist das Verteilverfahren[5] [MR96]. Für dieses Verfahren ist ein Überblick über die Struktur und den Inhalt der zu erstellenden Daten notwendig, z.B. in Form einer Gliederung. Die Arbeit wird zunächst auf alle Gruppenmitglieder verteilt, von diesen durchgeführt und die Teilergebnisse schließlich gemeinsam überarbeitet. Das Verteilverfahren wird z.B. in Seminaren praktiziert. Nach einer kurzen Vorbesprechung, in der die Themen verteilt werden, beginnt eine längere entkoppelte Phase, bis sich zuletzt alle Teilnehmer treffen um die Ergebnisse zu diskutieren.

Neben der Möglichkeit die Verteilung nach Teilproblemen vorzunehmen (Horizontale Division), kann auch ein Rollenkonzept verwendet werden (Vertikale Division oder Stratifikation). [MR96]. Um einen wissenschaftlichen Artikel in Druck geben zu können, braucht es einen Experten, der sich in dem Problemgebiet auskennt und Informationen darüber liefern kann, einen Autor, der die Erkenntnisse des Experten bündelt und festhält und einen Korrektor, der die schriftlichen Ausarbeitungen des Autors auf Fehler überprüft. Die verschiedenen Rollen müssen nicht zwangsläufig von verschiedenen Personen übernommen werden.

Schließlich besteht noch die Möglichkeit, daß jedes Gruppenmitglied eine vollständige Bearbeitung der Aufgabe vornimmt (Exploration). Anschließend werden alle Ausarbeitungen von allen Gruppenmitgliedern gelesen und eine gemeinsame Version unter Verwendung der besten Stellen der einzelnen Gruppenmitglieder erstellt (Konsolidierung) [Sch98]. Der Arbeitsaufwand für den einzelnen Teilnehmer ist hier zwar deutlich höher als beim Verteilverfahren, dafür wird die Aufgabe auf tiefergehendere Weise bearbeitet.

Das Prozeßmodell SECAI (Summarization, Evaluation, Comparison, Argumentation and Integration) geht z.B. nach diesem Verfahren vor. SECAI wurde insbesondere für Lernsituationen entwickelt, die wissenschaftliche Artefakte behandeln. In der Explorationsphase arbeitet jedes Gruppenmitglied eine Zusammenfassung und eine Bewertung für sich aus. In der Konsolidierungsphase finden zunächst Vergleich und Diskussion, schließlich die Integration der verschiedenen Sichten statt. [Sch98]

Häufig richtet sich die von einer Gruppe gewählte Arbeitsweise nicht streng nach einem dieser Paradigmen aus, sondern vereinigt mehrere Ansätze oder wechselt gar das Verfahren. Zusätzlich kann sich zur Ausarbeitung eines Teilproblems eine Untergruppe bilden, die ihre eigene Arbeitsweise wählt.

Verfahrenswechsel erfordern ein hohes Maß an Koordination. Wird die Arbeitsweise häufig gewechselt, so spricht man unabhängig von den Eigenschaften der Verfahren von einer eng gekoppelten Arbeitsweise.

Jo Ann Oravec betont, daß es der Produktivität hinderlich ist, die Arbeitsform strikt von außen vorzugeben. Sie schreibt: „the groups would fare better in measures of productivity by letting productive members head off in their pre-

---

[5] englisch: Horizontal Division

ferred direction and pace“ [Ora96] (Gruppen würden, gemessen an ihrer Produktivität, günstiger fahren, wenn man ihre Teilnehmer in deren bevorzugte Richtung und Tempo vorstoßen lassen würde). Um dies zu erfüllen, bedeutet es für ein Groupwaresystem, Unterstützung für alle möglichen Formen der Arbeitsweisen bereitstellen zu müssen.

2.3 Technische Kategorien von Groupware

Die Groupwaresysteme klassifizieren sich u.a. nach folgenden sechs Eigenschaften: Verteilungsstruktur, temporale Korrelation des Arbeitsvorgangs, Förderung des Gruppenbewußtseins, Grad der Kopplung der Sichten, Art der Datenverteilung und Art der Koordination nebenläufigen Arbeitens.

Bei der Verteilungsstruktur unterscheidet man zwischen zentralen und replizierten Architekturen. Während bei einer zentralen Architektur alle Daten auf einem ausgezeichneten Rechner, dem Server, liegen und auf diesem von den Gruppenmitgliedern bearbeitet werden, liegen bei einer replizierten Architektur Kopien der Daten auf den jeweiligen Clients. Durch Abgleich von Datenmodifikationen an alle Rechner wird erreicht, daß jedem Gruppenmitglied die gleichen Daten zur Bearbeitung vorliegen.

Die temporale Korrelation des Arbeitsvorgangs unterscheidet zwischen einer synchronen und einer asynchronen Kooperation der Gruppenmitgliedern. Die synchrone Kooperation besteht aus der gleichzeitigen Bearbeitung von Daten durch die Gruppenmitglieder, wohingegen die asynchrone Kooperation erfordert, die Daten zu unterschiedlichen Zeiten zu ändern. Angestrebt werden Systeme, die beide Arten inklusive Übergänge dazwischen und Mischformen daraus unterstützen. Dazu gehört das Zulassen von Teilnehmern die zu spät kommen oder die Sitzung frühzeitig verlassen möchten.

Die Förderung des Gruppenbewußtseins geschieht durch Mitteilungen über die Arbeit anderer Gruppenmitglieder. Gruppenbewußtsein bezieht sich dabei nicht nur auf die Information welche Person an welcher Aufgabe arbeitet, sondern auch auf die Frage, wer gerade ansprechbar und offen für Diskussionen ist [PMB96]. Diese Kategorie unterteilt sich in die folgenden Bereiche: Bewußtsein seiner Selbst („Wo/Was bin ich?“), Bewußtsein der Anderen („Wer ist wo? Wer tut was? Wer ist ansprechbar?“) und Kommunikationsmöglichkeiten. Kommunikation findet häufig nicht nur im Austausch von schriftlichen oder gesprochenen Nachrichten statt. Z.B. ist es auch nötig, eine Verbindung von Nachrichten zum gemeinsamen Datenraum herzustellen („Hier liegt das Problem!“) In diesem Zusammenhang fällt häufig der Begriff Telepointer, der einen Zeigestock im verteilten Datenraum repräsentiert.

Der Grad der Förderung legt fest, wie stark ein Gruppenmitglied über die Tätigkeiten der anderen Gruppenmitglieder informiert wird. Ist dieser Grad zu niedrig, wird Arbeit ggf. mehrfach verrichtet und die Teilnehmer können sich nur schwer gegenseitig synergetisch beeinflussen.

Der Grad der Kopplung der Sichten legt fest, wie unabhängig die Teilnehmer voneinander in der Anwendung navigieren können. Dies reicht von der totalen

Synchronizität der Bildschirmdarstellung (eng gekoppeltes WYSIWIS[6]) bei den Gruppenmitgliedern bis zum unabhängigen Navigieren jedes Mitgliedes (lose Kopplung).

Für die Datenverteilung bestehen drei Möglichkeiten. Zunächst besteht die Möglichkeit der Verteilung herkömmlicher, auf einen einzelnen Benutzer zugeschnittener Anwendungen auf mehrere Benutzer (Application Sharing). Die Daten liegen nur auf dem Rechner, der die herkömmliche Applikation gestartet hat. Die anderen Teilnehmer erhalten lediglich Bilder des Bildschirminhalts. Eine weitere Möglichkeit ist das Starten einer speziellen Applikation auf allen Clients, deren Daten durch das propagieren von Eingabeereignissen an alle Teilnehmer konsistent gehalten werden (GUI Event Sharing). Die dritte Variante ist das Verteilen der Datenstruktur (Data Sharing). Auch hier starten alle Teilnehmer die gleiche Gruppenanwendung, die mit der verteilten Datenstruktur arbeitet.

Die Koordination von nebenläufiger Arbeit kann ebenfalls auf drei unterschiedliche Arten erfolgen. Die einfachste Möglichkeit besteht in der Verwendung von Sperrmechanismen, die es ermöglichen, Datensegmente für Gruppenmitglieder zu reservieren und damit ein gleichzeitiges Bearbeiten dieser Daten durch mehrere Benutzer unmöglich machen. Eine Sonderform eines Sperrmechanismus ist ein Tokenmechanismus [BS98], bei dem die Sperre auf den gesamten gemeinsamen Datenraum gesetzt wird, so daß jeweils nur ein Gruppenmitglied in der Lage ist Daten zu verändern. Sperrverfahren können mithilfe von Transaktionen automatisiert werden, die mehrere Operationen zu einer Einheit verbinden, die entweder komplett oder gar nicht ausgeführt wird. Vor der Ausführung einer Transaktion werden die nötigen Sperren gesetzt, dann die Modifikationen ausgeführt und die Sperren wieder aufgehoben. Wurden benötigte Sperren bereits von einer anderen Transaktion gesetzt, wird die Transaktion verworfen oder zu einem späteren Zeitpunkt ausgeführt.

Neben diesen pessimistischen Verfahren zur Nebenläufigkeitskontrolle, auch Konfliktvorbeugungsmechanismen[7] genannt [Tie01], finden speziell in Systemen mit Echtzeitanforderungen optimistische Verfahren, die Konflikte im nachhinein auflösen, Anwendung. In replizierten Architekturen wird die Modifikation auf den lokalen Daten ausgeführt, bevor diese auf Konflikte überprüft wurden. Im Falle eines Konfliktes werden die lokalen Daten so modifiziert, daß wieder Konsistenz herrscht.

Zur Implementierung von optimistischen Verfahren können wiederum Transaktionen verwendet werden. Siehe hierzu auch Kapitel 2.7.3.

Ebenfalls in diese Klasse fallen die sogenannten Transformationsverfahren. Gleichzeitig vorgenommene Operationen, die miteinander in Konflikt stehen, werden dabei so transformiert, daß sie ihre Gültigkeit behalten [BS98].

---

[6] Abkürzung für What You See Is What I See; deutsch: Alle Benutzer sehen das Gleiche.

[7] englisch: Conflict Prevention

2.4 Vor- und Nachteile der verschiedener Groupwaresysteme

In diesem Kapitel sollen exemplarisch einige Groupwaresysteme und -frameworks vorgestellt werden und nach den im vorigen Kapitel genannten Merkmalen klassifiziert werden. Anhand der Systeme sollen die verschiedenen Vor- und Nachteile der Merkmale, sowie deren Implementierung, diskutiert werden.

Netmeeting

Verteilungsstruktur	zentral
synchron / asynchron	nur synchron
Gruppenbewußtsein	nicht unterstützt
Grad der Kopplung der Sicht	enges WYSIWIS
Art der Datenverteilung	Application Sharing
Nebenläufigkeitskontrolle	Tokenmechanismus

Application Sharing wird z.B. von Microsoft Netmeeting unterstützt. Diese Art der Datenverteilung setzt den Tokenmechanismus als Nebenläufigkeitskontrolle, eine zentrale Verteilungsstruktur, eine enge Kopplung der Ansichten und eine vorwiegend synchrone Arbeitsweise voraus. Es werden keine expliziten Mechanismen zur Förderung des Gruppenbewußtseins angeboten. Da aber stets nur ein Teilnehmer die Daten modifizieren kann, während alle anderen zusehen, ist die ganze Gruppe über seine Beiträge informiert.

Vorteil des Application Sharing ist, daß die Teilnehmer in der Regel keine neue Bedienung erlernen müssen. Die Netzlast ist bei diesem Verfahren besonders hoch, da neben systemnahen Eingabeereignissen die Bildschirmrepräsentation als Rastergrafik übertragen wird.

Application Sharing eignet sich besonders für eine Arbeitsweise mittels Schriftführer, da hier keine Transitionen zwischen synchronem und asynchronem Arbeiten nötig sind und nur eine Person zur Zeit die Daten modifiziert.

CORBA

Verteilungsstruktur	zentral
synchron / asynchron	beides
Gruppenbewußtsein	implementierungsabhängig
Grad der Kopplung der Sicht	sehr gering
Art der Datenverteilung	Data Sharing
Nebenläufigkeitskontrolle	externe Synchronisationservices

Bei CORBA (Common Object Request Broker Architecture) handelt es sich um eine programmiersprachenunabhängige Architektur zum objektorientierten Austausch von Daten und Nachrichten über mehrere Rechner hinweg. Diesen objektorientierten Austausch von Daten nennt man Remote Procedure Call (RPC).

Für den Anwendunsprogrammierer bedeutet das eine scheinbare Übertragung des entfernten Objekts auf den lokalen Rechner, wo dann entsprechende Methoden darauf aufgerufen werden. In Wirklichkeit verbleibt das Objekt auf dem Server, es werden lediglich die Methoden mit Parametern sowie anschließend die Ergebnisse kodiert und zwischen lokalem Rechner und Server übermittelt.

Das Transportprotokoll wird durch spezielle Schnittstellen, die in der CORBA-spezifischen Sprache IDL (Interface Definition Language) verfaßt sind, gekapselt. Zwischen Client und Dienst steht ein spezieller Mittelsmann, der ORB (Object Request Broker), bei dem Anfragen der Clients eintreffen. Der ORB leitet die Anfrage dann entweder an direkt den Dienst oder über das spezielle IIOP (Internet Inter-ORB Protocol) an den nächsten ORB weiter. Durch das Einsetzen von mehreren ORBs erreicht CORBA trotz der zentralen Architektur eine gute Skalierung.

CORBA bietet keine spezielle Form der Nebenläufigkeitskontrolle. Es steht der Anwendung frei, sich über spezielle Dienste zu synchronisieren. Ein spezieller CORBA-Dienst, der dafür angeboten wird, ist der „Concurrency Service“.

Ebenso überläßt es CORBA der Anwendung, Daten zur Förderung des Gruppenbewußtseins zu verwalten.

CORBA ist ein vielversprechender Ansatz, da im Gegensatz zu anderen RPC-Varianten die Protokolle plattform- und programmiersprachenunabhängig sind und viele Standardkomponenten für verschiedene Dienste dafür angeboten werden. Einer der Hauptnachteile von CORBA jedoch ist die zentrale Architektur und die damit verbundene hohe Netzlast bei der Kommunikation zwischen Client und entferntem Objekt, der insbesondere bei synchroner Kooperation zum Tragen kommt.

Lotus Notes

Verteilungsstruktur	repliziert
synchron / asynchron	asynchron
Gruppenbewußtsein	ja
Grad der Kopplung der Sicht	beliebig
Art der Datenverteilung	Data Sharing
Nebenläufigkeitskontrolle	Replizierungsprotokoll, Sperrmechanismen

Lotus Notes von IBM implementiert Groupware durch replizierte, relationale Datenbanken. Lotus Notes Applikationen sind vorwiegend für die asynchronen Kooperation vorgesehen, in neuerer Zeit wurden auch synchrone Groupwareappli-

kationen mit Notes erstellt (Lotus Sametime). Das in Lotus Notes implementierte Rollenkonzept dient einerseits zur Implementierung des Arbeitsparadigmas Stratifikation, andererseits dient es zur Errichtung von Modifikationssperren. Die Nebenläufigkeitskontrolle besteht aus mehreren Strategien: durch rollenbasierte Modifikationssperren, durch Tokenverfahren, aus der Kommutativität des Hinzufügens von Daten und durch Replizierungsprotokolle. Ein übliches Replizierungsprotokoll geht folgendermaßen vor: Nehmen zwei Benutzer gegenläufige Modifikationen am gleichen Dokument vor, so wird eine der beiden Modifikationen in das Hauptdokument, die andere in ein sogenanntes Antwortdokument übernommen. Das Antwortdokument wird an das Hauptdokument angehängt. Anschließend ist es Aufgabe des Datenbank-Administrators oder der Benutzer, die beiden Dokumentversionen zu integrieren.

Die Nachteile von Lotus Notes ergeben sich hauptsächlich aus den Konfliktbehandlungsstrategien. Fügt man stets nur Daten hinzu, so kommt es leicht vor, daß interessante Informationen aus der Menge nicht mehr erkennbar sind. Das von Notes verwendete Replizierungsprotokoll erfordert ein manuelles Nachbessern (siehe hierzu auch 3.1). Rollenbasierte Modifikationssperren sind ebenfalls nicht geeignet, einen ungehinderten Arbeitsfluß zu ermöglichen. Häufig muß ein Kollege, der spezielle Rechte besitzt, gebeten werden, zusätzliche Arbeit zu verrichten.

Sperrmechanismen bergen grundsätzlich das Problem, daß Teilnehmer erst dann mit ihrer Arbeit fortfahren können, wenn die Sperre freigegeben wurde. Auch wenn sich die betroffenen Personen extern koordinieren und somit keine Konflikte auftreten würden.

[ND], [Hil91], [IBM01], [BS98]

DyCE

Verteilungsstruktur	repliziert
synchron / asynchron	beides
Gruppenbewußtsein	abhängig von der Implementierung
Grad der Kopplung der Sicht	beliebig
Art der Datenverteilung	Data Sharing
Nebenläufigkeitskontrolle	Transaktionsbasiert

Durch die replizierte Verteilungsstruktur ist DyCE CORBA bezüglich der Performanz überlegen. Gleichzeitig verbindet es dichte Replikationszyklen und transaktionsbasierte Mechanismen, die der Gruppe eine parallele Arbeitsweise ermöglichen. Obwohl DyCE speziell für synchrone Kooperation ausgelegt ist, erlaubt es auch asynchrone Modi, sowie das verspätete Eintreffen von Teilnehmern und die Wiederaufnahme einer früheren Sitzung.

Im Gegensatz zu Lotus Notes Anwendungen, müssen Teilnehmer im DyCE System niemals Konflikte manuell auflösen. Bei gleichzeitigen gegenläufigen

Modifikationen geschieht etwas schlimmeres: Eine der beiden Versionen geht verloren. Es wird versucht, dieses Defizit durch kurze Replikationszyklen auszugleichen. Aber auch die Anwendung kann hier viel zur Vorbeugung und Vermeidung von Konflikten betragen. Solche Mechanismen sind insbesondere das Ziel dieser Diplomarbeit.

2.5 Anforderungen an eine kooperative Textverarbeitung

In dieser Diplomarbeit soll eine DyCE Applikation entwickelt werden, die es der Gruppe ermöglicht, gemeinsam ein Textdokument zu erstellen. Die Anforderungen an die Applikation werden anhand eines Szenarios herausgearbeitet, in dem die Benutzer A, B und C gemeinsam einen Bericht über eine Tagung verfassen sollen.

Anforderung 1 (Untergliederung des Dokuments in Kapitel und andere logische Einheiten)

Da der Bericht mehr als fünf Seiten umfassen wird, bietet sich die Gliederung des Dokuments in verschiedene Kapitel an, um die Lesbarkeit zu erhöhen. Aus der Kapitelstruktur soll anschließend ein Inhaltsverzeichnis automatisch generierbar sein. Evtl. müssen logische Informationen über Tabellen und Bilder ebenfalls erfaßt werden.

Anforderung 2 (Formatierungen)

Durch den Umgang mit üblichen Textverarbeitungssystemen sind es die Benutzer gewöhnt, ihren Text mit verschiedenen Formatierungsmöglichkeiten, wie Schriftarten, -größen und -farben aber auch Merkmalen wie Fettsatz, Kursivschrift und Unterstreichungen auszuschmücken. Desweiteren heben sich Überschriften durch Formatierungen besser vom Text ab.

Anforderung 3 (XML-Export)

Nach Fertigstellung soll das Dokument nach XML exportiert werden, um so in einer elektronischen Bibliothek archiviert zu werden und um die Bearbeitung mit anderer Software zu ermöglichen.

Anforderung 4 (Parallele Modifikation)

Es wurde festgelegt, daß Benutzer A die Einleitung des Dokuments, C das Schlußwort formuliert. Sie möchten dabei möglichst unabhängig voneinander arbeiten.

Anforderung 5 (Unterschiedliche temporale Korrelation)

Während Benutzer A und B schon nach Hause gegangen sind, möchte Benutzer C weiterarbeiten. Trotz seiner Verspätung am nächsten Morgen, arbeiten Benutzer A und B bereits mit der von ihm modifizierten Version. Als Benutzer C endlich im Büro eintrifft, kann er einfach der Sitzung beitreten ohne daß Benutzer A und B dazu ihre Arbeit unterbrechen müssen.

Anforderung 6 (Mehrere Ansichten)

Benutzer A strukturiert die Gliederung neu. Dabei muß er oftmals zwischen verschiedenen Kapiteln navigieren. Er erwartet eine zweite Ansicht des Dokuments, die nur Kapitelüberschriften beinhaltet. Durch einen Mausklick auf die entsprechende Überschrift kann er schnell an die entsprechende Stelle im Dokument navigieren.

Anforderung 7 (Schnelle Anzeige fremder Modifikationen)

Während Benutzer A und B am gleichen Absatz arbeiten, diskutieren sie die Änderungen, die sie am Dokument vornehmen. Sie erwarten dabei, daß die vorgenommenen Änderungen nach sehr kurzer Zeit beim Kollegen sichtbar sind.

Anforderung 8 (Telepointer)

Während der Diskussion, möchte sich Benutzer A auf eine Stelle im Dokument beziehen. Dazu schaltet er seinen Telepointer ein und zeigt damit auf die gemeinte Stelle. Benutzer B weiß durch den Telepointer, worauf sich die Aussagen von A beziehen. Der Telepointer ersetzt die aufwendige verbale Kommunikation über den Bezug zum Inhalt, die oft zu Mißverständnissen führt.

Anforderung 9 (Gruppenbewußtsein)

Da Benutzer C in der deutschen Grammatik sehr versiert ist, überarbeitet er den bereits von A und B verfaßten Text. Dabei sind die Stellen im Text, die von den Teilnehmern gerade modifiziert werden, durch Hervorhebungen in bestimmter Farbe gekennzeichnet. Dadurch weiß Benutzer C, welche Textteile noch häufigen Veränderungen unterworfen sind und bei welchen sich eine Korrektur lohnt.

Anforderung 10 (Vermeidung von Konflikten)

Die Benutzer sind sich darüber bewußt, daß es in einem kooperativen System zu Konflikten kommen kann. Sie erwarten jedoch, daß diese von der Software möglichst vermieden werden ohne sie zum manuellen Eingreifen zu nötigen.

Anforderung 11 (Hinweis auf unvermeidbare Konflikte)

Ist ein Konflikt nicht automatisiert vermeidbar, möchten die Benutzer möglichst frühzeitig über das Konfliktpotenzial informiert werden, sich selbstständig zu koordinieren. Die Benutzer könnten dann das parallele Arbeitsparadigma verlassen und einen Schriftführer wählen oder ein Benutzer könnte dem anderen den Vortritt lassen.

2.6 Stand auf dem Arbeitsgebiet

In diesem Abschnitt sollten zwei aus der Literatur bekannten Ansätze vorgestellt werden. Beide werden anschließend mit dem Anforderungskatalog verglichen, um ihre Defizite herauszuarbeiten.

2.6.1 GROVE

Verteilungsstruktur	verteilt
synchron / asynchron	synchron
Gruppenbewußtsein	ja
Grad der Kopplung der Sicht	striktes WYSIWIS
Art der Datenverteilung	Data Sharing
Nebenläufigkeitskontrolle	Transformationsverfahren

Mit GROVE (Group Outline View Editor) entwickelten Ellis, Gibbs und Rein einen Gruppeneditor für synchrone, verteilte Rechnerkonferenzen.

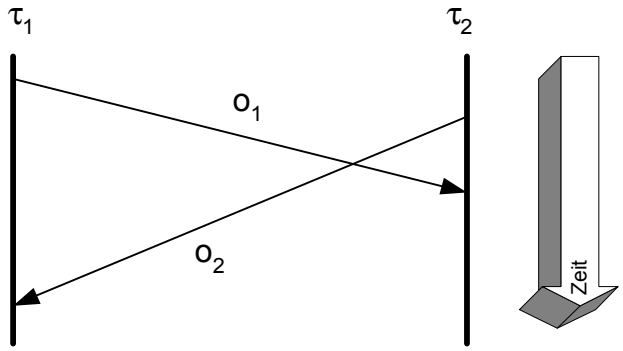


Abbildung 3: Szenario zur Übertragung von Operationen in GROVE
Teilnehmersystem τ_1 nimmt die Operation o_1 vor. Bevor diese zu Teilnehmersystem τ_2 übermittelt werden kann, initiiert dieses eine weitere Operation o_2. Das Ergebnis der Ausführung beider Operationen muß bei beiden Teilnehmersystemen äquivalent zur Ausführung von $o_1 \circ o_2$ sein.

Durch die Verwendung von Transformationsverfahren, können die Gruppenmitglieder relativ unabhängig von einander ihre Eingaben tätigen (vgl. Anforderung 4 (Parallele Modifikation). Der speziell dafür entwickelte GROVE-Algorithmus kann zu zwei Operationen o_1 und o_2 transformierte Transformationen o_1‘ und o_2‘ generieren, so daß für die Hintereinanderausführung gilt:

$o_2' \circ o_1 = o_1' \circ o_2$

Angenommen das Teilnehmersystem τ_1 initiiere zuerst Operation o_1. Anschließend, jedoch vor Erhalt von o_1, intiiere Teilnehmersystem τ_2 die Operation o_2. Die resultierende Operation soll auf beiden Teilnehmersystemen äquivalent zu $o_1 \circ o_2$ sein. Dazu wird bei τ_1 neben der bereits ausgeführen Operation o_1 die zu o_2 transformierte Operation o_2‘ ausgeführt, auf τ_2 wird zusätzlich zu o_2 die transformierte Operation o_1‘ angewendet.

Durch diesen Algorithmus werden alle Konflikte automatisch aufgelöst, womit Anforderung 10 (Vermeidung von Konflikten) in einer Weise erfüllt wird, die Anforderung 11 (Hinweis auf unvermeidbare Konflikte) überflüssig macht.

Anforderung 7 (Schnelle Anzeige fremder Modifikationen) ist sicherlich erfüllt, da die zu übertragenen Datenmengen relativ klein sind.

Da sich die in GROVE verwendeten Operationen nur auf rein textuelle Informationen beziehen, wird die Anforderung 2 (Formatierungen) nicht unterstützt. In GROVE wird ebenfalls keine Informationen über die Gliederung des Dokuments verwaltet, so daß Anforderung 1 (Untergliederung des Dokuments in Kapitel und andere logische Einheiten) ebenfalls nicht erfüllt werden kann.

Da der GROVE-Algorithmus eine feste Anzahl von Teilnehmersystemen annimmt, wird Anforderung 5 (Unterschiedliche temporale Korrelation) ebenfalls nicht unterstützt.

Weiterhin erlaubt GROVE nur eng gekoppeltes WYSIWIS, so daß die Anforderung 4 (Parallele Modifikation) nur in so lange unterstützt wird, wie die Benutzer die gleiche Bildschirmseite manipulieren.

Eine Anbindung an XML-Applikationen wurde nicht besprochen.

[BS98]

2.6.2 ShrEdit

Verteilungsstruktur	zentral
synchron / asynchron	synchron
Gruppenbewußtsein	teilweise
Grad der Kopplung der Sicht	lose Kopplung, aber auch striktes WYSIWIS mit einzelnen Teilnehmern möglich
Art der Datenverteilung	Data Sharing
Nebenläufigkeitskontrolle	Sperrmechanismen

Olson, Olson und McGuffin entwickelten ShrEdit (Shared Editor) um Erkenntnisse über den Ablauf von Gruppenprozessen zu gewinnen [OO96]. Sie modifizierten dazu einen vom Betriebssystem MacOS bereitgestellten Texteditor, der der Anforderung 2 (Formatierungen) genügen würde.

In ShrEdit markieren die Benutzer den Bereich, den sie zu Bearbeiten gedenken, vor der Manipulierung. Über Sperrmechanismen wird dieser dann für sie reserviert. Andere Benutzer können diese Teile einsehen, jedoch nicht modifizieren. Hierdurch werden, wie bei GROVE, alle Konflikte vermieden (Anforderung 10 (Vermeidung von Konflikten)) und Anforderung 11 (Hinweis auf unvermeidbare Konflikte) ist nicht weiter zu beachten.

Obwohl die Sichtkopplung normalerweise dem losen WYSIWIS entspricht, erlaubt der sogenannte Tracking-Mechanismus die strikte WYSIWIS-Kopplung. Dazu können Benutzer die Ansicht, die ein anderer Teilnehmer auf das gemeinsame Dokument hat, verfolgen. Auf diese Weise erfüllt ShrEdit Anforderung 9 (Gruppenbewußtsein).

Der Einfachheit halber wird eine zentrale Architektur verwendet, so daß Benutzer zu beliebigem Zeitpunkt in einer Sitzung beitreten oder sie verlassen können (Anforderung 5 (Unterschiedliche temporale Korrelation)). Vollständig asynchrones Arbeiten wird mittels „privaten Fenstern“ ermöglicht, deren Inhalt von anderen Gruppenmitgliedern weder eingesehen noch modifiziert werden kann.

Die Zeitverzögerung wird von [DB92] als gering beschrieben, so daß Anforderung 7 (Schnelle Anzeige fremder Modifikationen) als erfüllt angesehen werden kann.

[DB92] kritisiert, daß ShrEdit weder Telepointer zu Verfügung stellt, noch Informationen zur Position des Einfügecursors der anderen Teilnehmer zugänglich macht (Anforderung 8 (Telepointer)).

Weiterhin ist es in ShrEdit zunächst nicht möglich logische Strukturinformationen zu speichern (Anforderung 1 (Untergliederung des Dokuments in Kapitel und

andere logische Einheiten) oder die Daten mit speziellen Werkzeugen zu bearbeiten, wie in Anforderung 6 (Mehrere Ansichten) gefordert wird.

Eine Anbindung an XML-Applikationen wurde ebenfalls nicht versucht.

[OO96], [DB92]

2.7 DyCE

Durch die Aufgabenstellung ist die Verwendung von DyCE als Framework festgelegt. In diesem Kapitel wird eine Einführung in DyCE gegeben. Es wird besprochen, in welcher Weise es Benutzer und Anwendungsprogrammierer unterstützt. Dabei wird der Aufbau von Anwendungskomponenten und die Architektur von DyCE, sowie Nebenläufigkeits- und Verteilungsmechanismen, die zu Verfügung gestellt werden, beleuchtet.

2.7.1 Unterstützung durch DyCE

DyCE (Dynamic Collaboration Environment) ist ein in Java implementiertes Groupware-Framework „zur Entwicklung kooperativer, wiederverwendbarer Komponenten“ [TS00]. Durch die Komponenten erhalten die Benutzer Zugang zu gemeinsamen Artefakten. Jeder Benutzer erhält einen persönlichen Arbeitsplatz in der virtuellen Umgebung, der kollaborative Werkzeuge, Referenzen auf die von ihm bearbeiteten Artefakte und Informationen über gleichzeitig im System angemeldete Personen bereitstellt. Benötigt man in einer Sitzung ein noch nicht vorhandenes Werkzeug, so kann dieses im laufenden Betrieb beim System registriert werden und steht danach allen Benutzern zu Verfügung. [Tie01]

DyCE unterstützt insbesondere synchrone Kooperation, indem Datenmodifikationen in quasi Echtzeit zu den anderen Teilnehmern übertragen werden. Es liefert Mechanismen, die zur Förderung des Gruppenbewußtseins geeignet sind. Weiterhin werden Kommunikationswerkzeuge für audiovisuellen und textbasierten Nachrichtenaustausch und Koordinierungsmöglichkeiten in Form von Tokenverfahren bereitgestellt.

Neben der rein synchronen Kooperation ist auch das asynchrone Arbeiten möglich, da alle Artefakte von einem Server persistent gespeichert werden. Weiterhin sind noch Mischformen möglich, die den Teilnehmern einen verspäteten Einstieg in die Sitzung oder ein frühes Verlassen erlauben.

DyCE unterstützt den Entwickler durch die transparente Verteilung der Artefakte. Die Datenmodelle der Artefakte sind leicht wiederverwendbar im Sinne der Objektorientierten Programmierung. Es ist möglich, von einem Artefakt auf andere Artefakten zu referenzieren und sie so miteinander zu kombinieren.

Die Artefakte können gemeinsame und lokale Daten enthalten. Der Entwickler hat somit die Möglichkeit, persönliche Einstellungen im Artefakt zu speichern oder Zwischenergebnisse zu puffern. Der Entwickler erhält Zugang zu Informa-

tionen über die Kooperation, mithilfe derer er Komponenten erstellen kann, die das Gruppenbewußtsein fördern.

Weiterhin bietet DyCE Intergrationsmöglichkeiten zu externen Architekturen. Ein Beispiel dafür sind die sogenannten DyCE-Servlets, die in der Lage sind, Informationen aus Artefakten zu extrahieren und im Webbrowser darzustellen.

2.7.2 Architektur von DyCE

Analog zum Programmierparadigma Model-View-Controler[8] setzt DyCE eine Trennung in Artefakt (Model) und Darstellungskomponente (View und Controler) voraus.

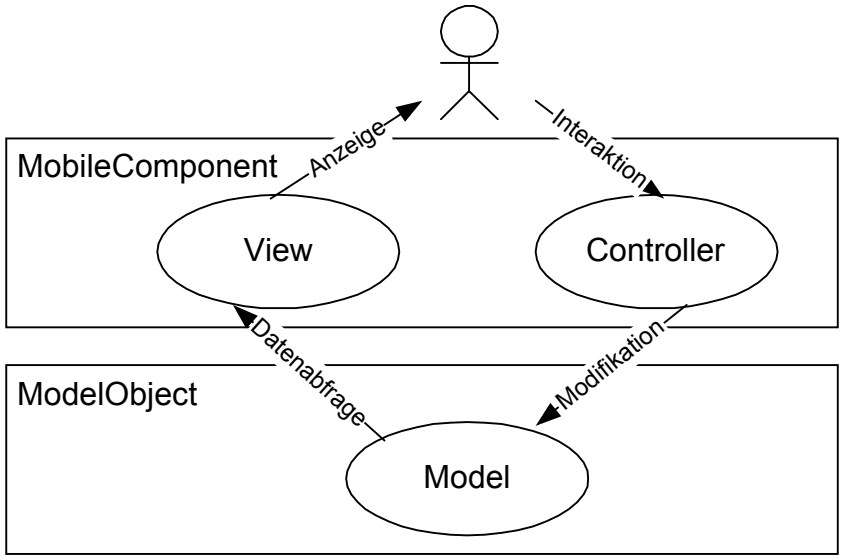


Abbildung 4: Model-View-Controler Paradigma in DyCE

Dazu muß das Datenmodell eine Instanz von ModelObject und alle Komponenten und Aggregate müssen serialisierbar sein.

Die Darstellungskomponente ist eine Ableitung von MobileComponent. Bei ihrer Instanziierung erhält sie eine Referenz auf das Artefakt.

[8] englisch für: Modell, Darstellung und Steuerung

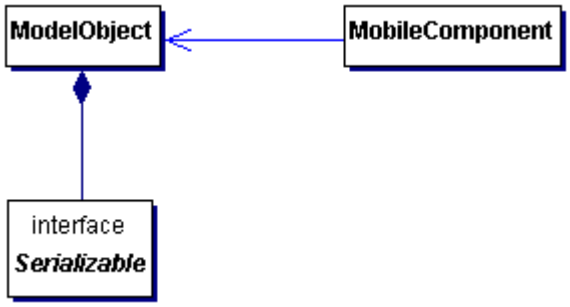


Abbildung 5: Grundstruktur einer DyCE Komponente als UML-Klassendiagramm

DyCE bietet die Möglichkeit, mehrere Darstellungskomponenten mit dem gleichen Datenmodell zu verknüpfen, wobei jede Darstellungskomponente eine bestimmte Sicht auf das Modell liefert und angepaßte Möglichkeiten zur Manipulation bietet. Dabei können sowohl verschiedene Komponenten auf dem gleichen Teilnehmersystem gestartet werden, als auch verschiedene Teilnehmer mit verschiedenen Darstellungskomponenten arbeiten.

Die Architektur von DyCE ist Client-Server basiert. Der Server verwaltet Benutzer, Werkzeuge und Sitzungen. Er speichert alle Artefakte, repliziert sie zu den Clients und sichert nebenläufige Modifikationen darauf.

Die Clients erhalten Artefakte und erzeugen sich die passende Komponente, um sie zu modifizieren.

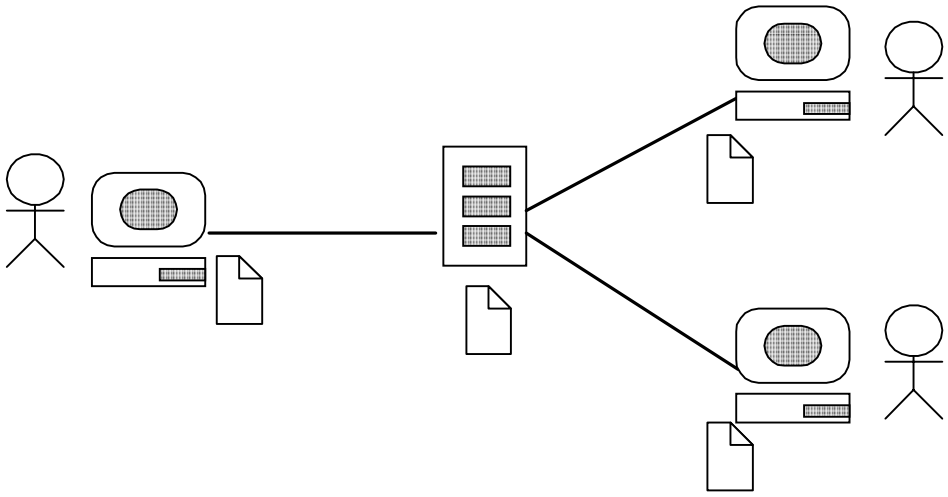


Abbildung 6: Replizierung der Artefakte auf Clientrechner und Server
Die Dokumente deuten die Replikate des Artefakts an, die auf Client- und Server-Rechner vorhanden sind. Die Pictogramme für Client, Rechner, Benutzer und Netzwerkverbindungen sind wie in Abbildung 1 gewählt. Das Artefakt wird durch das Blatt Papier mit gefalteter Ecke symbolisiert.

Um die Netzlast im laufenden Betrieb gering zu halten, wird bei Änderungen nicht das gesamte Artefakt erneut übertragen, sondern die Modifikation an die anderen Clients publiziert, die diese auf ihrem replizierten Modell ausführen.

Um das Artefakt auch an verspätete Teilnehmer replizieren zu können und um die Artefakte persistent zu speichern, hält der Server ebenfalls ein Replikat des Artefakts und aktualisiert es selbständig.

Im DyCE-Framework ist es möglich, spezielle serverseitige Komponenten zu integrieren, die bestimmte Modifikationen auf dem Artefakt automatisiert ausführen.

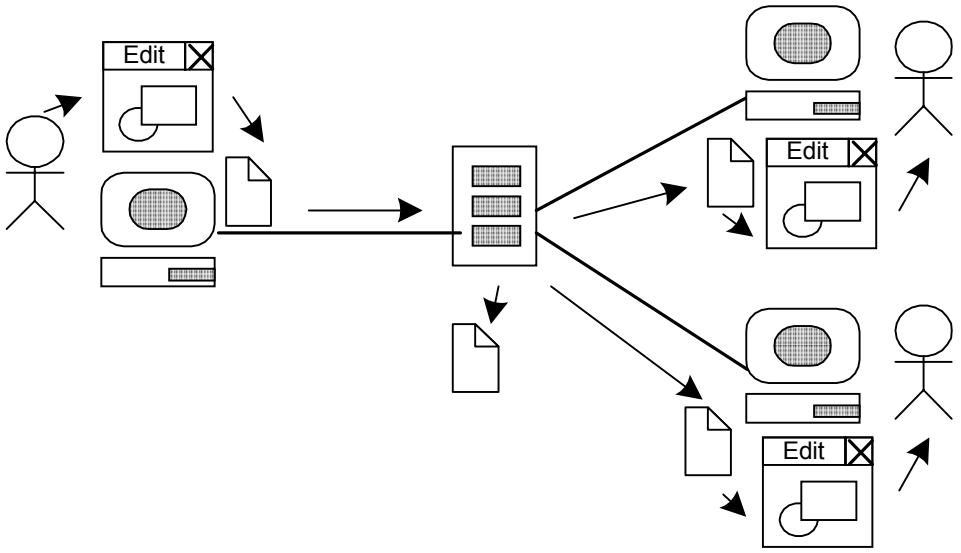


Abbildung 7: Informationsfluß und -speicherung bei Benutzerinteraktionen

Ein Benutzer interagiert mit seiner Darstellungskomponente. Dadurch wird das Artefakt auf seinem Rechner modifiziert und publiziert die Änderung an den Server. Der aktualisiert sein Replikat und reicht die Änderung an alle anderen Teilnehmersysteme der Sitzung weiter. Die Aktualisierung der Artefakte führt zu einer Änderung in der Darstellungskomponente und wird so den anderen Teilnehmern angezeigt.

Neben den in Abbildung 7 verwendeten Symbolen, wird die Darstellungskomponente durch ein Fenster mit dem Titel „Edit“ dargestellt. Die Pfeile geben den Informationsfluß an.

Um Artefakte über das Netzwerk zu replizieren, wird folgender Mechanismus verwendet. Das Artefakt selbst ist eine Instanz von ModelObject. Ein ModelObject speichert seine Daten in sogenannten Slots eines replizierungsfähigen Objekts, dem RObject. Das RObject und seine Slots sind serialisierbar, d.h. sie können über das Netzwerk verschickt werden. Wird einem Client ein RObject übermittelt, ist er in der Lage, daraus eine Kopie des ursprünglichen ModelObjects herzustellen. Die in den Slots enthaltenen Daten müssen ebenfalls serialisierbar sein.

Jedes RObject hat eine eindeutige ObjectID. Über sie kann das entsprechende RObject geladen und daraus ein Replikat des ModelObjects instanziiert werden. Da ObjectIDs serialisierbar sind, können auf diese Weise ModelObjects Aggregate anderer ModelObjects sein.

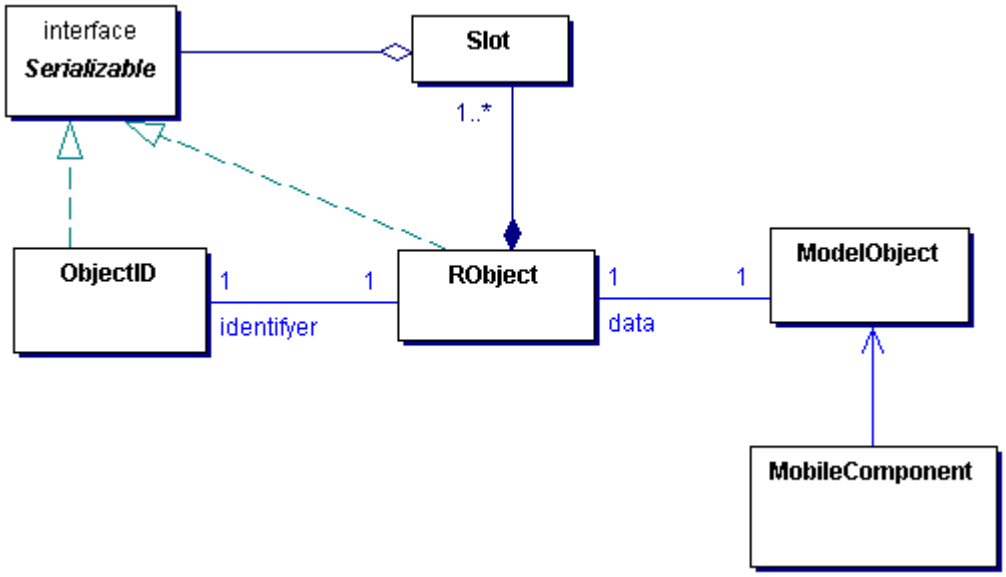


Abbildung 8: Replizierungsarchitektur in DyCE (UML-Klassendiagramm).

Da zwischen ModelObjects, RObjects und ObjectIDs jeweils bijektive Zuordnungen bestehen, werden diese Begriffe im Folgenden synonym gebraucht. Der Zugriff auf Slotinhalte geschieht analog zum Property Design Pattern [Rie97], so daß im weiteren Text nicht explizit zwischen Daten eines ModelObjects und Slotinhalten unterschieden wird.

2.7.3 Nebenläufigkeitskontrolle in DyCE

DyCE stellt die Konsistenz der modifizierten Daten durch die Kapselung von Operationen in Transaktionen sicher. Diese werden gemäß der optimistischen Nebenläufigkeitskontrolle verwendet.

Alle Transaktionen in DyCE arbeiten auf ModelObjects. Eine Operation auf einem ModelObject modifiziert einen der Slots des RObjects. Eine Transaktion gruppiert eine Reihe von Operationen auf beliebigen ModelObjects. Dabei bezieht sie sich auf die Zustände der ModelObjects.

Definition (Konflikt, Konflikteinheit, Konfliktgranularität): Zwei Transaktionen konfliktieren, wenn es in beiden eine Operation gibt, die im gleichen ModelObject den gleichen Slot modifiziert und sich dabei auf den selben Zustand des ModelObjects bezieht. Ein ModelObject bzw. Slot wird daher auch Konflikteinheit genannt. Je größer die Konflikteinheiten, desto gröber die Konfliktgranularität.

Definition (veraltete Transaktion): Eine Transaktion ist veraltet, wenn sie eine Operation enthält, die auf einem ModelObject operiert und sich die Transaktion dabei auf einen Zustand bezieht, der älter ist, als der Zustand, der dem Server bekannt ist.

Eine Transaktion muß die sogenannten ACID[9]-Eigenschaften erfüllen. Das sind Atomarität, Konsistenz, Isolation und Dauerhaftigkeit [HS00]. Atomarität heißt, daß eine Transaktion vollständig oder überhaupt nicht ausgeführt wird. Konsistenz besagt, daß die Transaktion einen gültigen Datenzustand hinterläßt. Isolation bedeutet, daß zwei parallel ausgeführte Transaktionen sich nicht gegenseitig beeinflussen können: Jede Transaktion kann davon ausgehen, daß sie im Sinne einer Ein-Benutzer-Anwendung ausgeführt wird. Dauerhaftigkeit heißt, daß die Wirkung einer Transaktion dauerhaft im System vorhanden ist.

Der Lebenszyklus einer Transaktion sieht wie folgt aus: Die Transaktion wird auf einem Teilnehmersystem erstellt. Sie wird zunächst lokal auf der Datenbasis ausgeführt und anschließend zum Server übermittelt. Dieser prüft, ob die Transaktion veraltet ist, und ob sie mit anderen Transaktionen in Konflikt steht. Ist dies nicht der Fall, wird sie an alle weiteren Teilnehmersysteme verteilt und auf deren Datenbasis ausgeführt.

Ist die Transaktion veraltet, so wird sie auf dem erstellenden Client zurückgenommen. Konfliktiert sie mit einer weiteren Transaktion, so wird eine der beiden Transaktionen ausgewählt und diese auf dem Client zurückgenommen, der sie erstellt hat.

[9] Atomicity, Consistency, Isolation, Durability

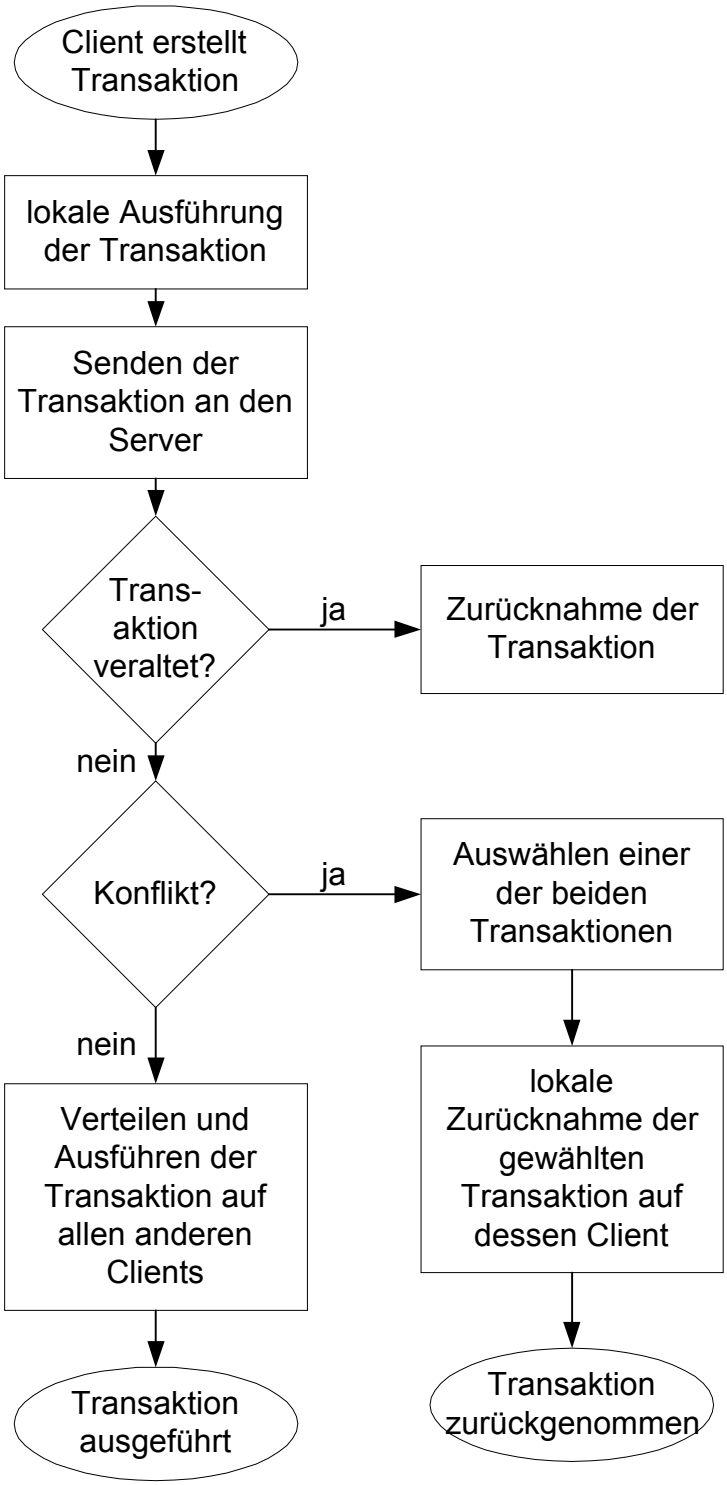


Abbildung 9: Lebenszyklus einer Transaktion als Steuerdiagramm. Start und Ende des Diagramms sind durch Ovale gekennzeichnet. Anweisungen werden durch Rechtecke, Entscheidungen durch Rauten dargestellt.

Definition (Zeitpunkt): Der Zustand eines ModelObjects wechselt zu bestimmten Zeitpunkten t_i, $i \in \{1, 2, ... \}$ in den Zustand q_i. Mit dem Zustand eines ModelObjects zum Zeitpunkt t ist der Zustand q_i gemeint wenn $t_i \leq t$ und $t_{i+1} > t$ ist.

Die Formulierung „zwei Teilnehmersysteme nehmen gleichzeitig eine Modifikation am gleichen ModelObject vor“ bedeutet, daß beide Modifikationen auf den gleichen Zustand bezugnehmen und damit einen Konflikt auslösen.

2.7.4 Konfliktbehandlungen mit undo-redo

Wie soll die Software auf die Zurücknahme einer Transaktion reagieren?

Eine Möglichkeit ist es, die Zurücknahme zu akzeptieren und den Benutzer entscheiden zu lassen, ob er die Aktion wiederholt oder ob er mit einer anderen Aktion fortfahren möchte (z.B. weil die alte Aktion nun unangebracht ist).

Ein von DyCE-Programmierern häufig eingeschlagener Weg ist es, die Transaktion erneut in Auftrag zu geben und zwar so oft, bis keine Zurücknahme mehr erfolgt. Dieser Weg funktioniert in den meisten Fällen. Seien z.B. drei konfliktierende Transaktionen in Auftrag gegeben, wird im ersten Schritt eine ausgeführt und zwei zurückgenommen. Diese zwei werden wiederholt, dabei kommt eine zur Ausführung, eine wird zurückgenommen. Spätestens im nächsten Schritt wird auch die letzte der drei Transaktion ausgeführt und das System kommt zur Ruhe.

Der Nachteil dieser Lösung ist zum Einen eine erhebliche Laufzeiteinbuße, da bei n gleichzeitigen Transaktionen die letzte bis zu (n-1)-mal auf dem Client ausgeführt und zurückgenommen wird, zum Anderen eine Stagnation des Systems, wenn pro Zeiteinheit genau so viele Transaktionen initiiert werden, wie gerade vom Server verwaltet werden können. Dann nämlich potenzieren sich die konfliktierenden Transaktionen, so daß die Anzahl der beim Server eingehenden Transaktionen größer ist, als vom Server verwaltet werden kann.

Fazit ist, daß die Konfliktbehandlung durch Verwenden der undo-redo-Methode problematisch ist und man sie nicht anwenden sollte, wenn es Alternativen gibt. Einzelne Konflikte löst sie jedoch auf einfache Weise.

2.7.5 Defizite von DyCE

DyCE löst bereits eine Reihe von Anforderungen an eine kooperative Textverarbeitung (siehe Kapitel 2.5). Dazu gehört Anforderung 5 (Unterschiedliche temporale Korrelation) sowie Anforderung 7 (Schnelle Anzeige fremder Modifikationen).

Dyce unterstützt das Gruppenbewußtsein zunächst nur durch eine Liste von Namen der an der Sitzung teilnehmenden Personen. Diese Unterstützung ist nicht ausreichend, um sich der Tätigkeiten der anderen Teilnehmer bewußt zu werden, wie in Anforderung 9 (Gruppenbewußtsein) gefordert wird. Zur Erfüllung der Anforderung 8 (Telepointer) bietet die DyCE-Umgebung bereits eine Komponente an, die von Dyce-Applikationen verwendet werden kann.

Die Anforderung 6 (Mehrere Ansichten) kann von einer DyCE-Komponente leicht erfüllt werden, indem eine weitere Darstellungskomponente implementiert wird, die mit dem gleichen Datenmodell arbeitet. DyCE übernimmt die Verwaltung und die Konsistenzsicherung automatisch.

Anforderung 4 (Parallele Modifikation) wird von DyCE prinzipiell unterstützt, jedoch werden Konflikte nicht ausgeschlossen. Modifikationen können daher nicht vollkommen unbeeinflußt vorgenommen werden. Wie im Kapitel 2.7.4 beschrieben, kann Anforderung 10 (Vermeidung von Konflikten) durch undo-redo-Mechanismen zwar kurzfristig erreicht werden, die Nebenläufigkeitsmechanismen von DyCE sind jedoch grundsätzlich verbesserungsfähig. Für Anforderung 11 (Hinweis auf unvermeidbare Konflikte) ist zur Zeit in DyCE keine Unterstützung vorgesehen.

2.8 Ein kooperativer Texteditor in DyCE

In diesem Abschnitt wird ein mittels Standardmechanismen von DyCE realisierter kooperativer Texteditor namens „Notepad“ vorgestellt und mit den Anforderungen aus Kapitel 2.5 bewertet.

Abbildung 10: Beispiel zur Verwendung der DyCE-Komponente Notepad (Screenshot) Benutzer b fügt den Text „Erster Streich“ ein. Die Modifikation erscheint mit geringer Zeitverzögerung bei Benutzer a.

Der Texteditor Notepad verwendet ein ModelObject, das in einem Slot den gesamten Text als eine Zeichenkette speichert. Wie in Kapitel 2.7.5 diskutiert, bie-

ten Standardmechanismen von DyCE keine ausreichende Unterstützung für Anforderung 10 (Vermeidung von Konflikten). Durch die Verwendung von einem einzigen Slot konfliktieren alle parallel ausgeführen Operationen. Da zusätzlich die Cursorpositionen in Anzahl der Zeichen zum Dokumentanfang gespeichert werden, können sich Einfügeoperationen eines Benutzers auf die Cursorposition eines anderen auswirken. Unabhängiges Arbeiten im Sinne von Anforderung 4 (Parallele Modifikation) ist daher nicht möglich.

Die Benutzer können verschiedene Textteile betrachten, da der Ansatz des losen WYSIWIS verfolgt wird. Eine weitgehende Förderung des Gruppenbewußtseins (Anforderung 9 (Gruppenbewußtsein)) wird nicht geboten: Fremde Cursor werden nicht angezeigt und auch ein Telepointer wurde nicht eingebunden. Es gibt keine Möglichkeit herauszufinden, welcher Benutzer welchen Abschnitt gerade bearbeitet. Anforderung 11 (Hinweis auf unvermeidbare Konflikte) wird ebenfalls nicht unterstützt.

Notepad bietet keine Unterstützung für Formatierungen, wie sie in Anforderung 2 (Formatierungen) gefordert werden. Auch eine logische Gliederung entfällt, so daß Anforderung 1 (Untergliederung des Dokuments in Kapitel und andere logische Einheiten) nicht erfüllt wird. Es wurde nur eine Präsentationskomponente implementiert. Da keine Strukturinformationen verwaltet werden, würde die Anwendung für Anforderung 6 (Mehrere Ansichten) fehlen.

Fazit ist, daß der vorhandene kooperative Texteditor in DyCE nicht geeignet ist, umfangreichere Dokumente parallel zu bearbeiten.

3 Optimierung der Nebenläufigkeitsmechanismen für DyCE

In diesem Kapitel werden, als erster Teil zur konzeptionellen Lösung, Strategien zur Verbesserung der Nebenläufigkeitskontrolle in DyCE erarbeitet, um die Anforderung 10 (Vermeidung von Konflikten) und die Anforderung 11 (Hinweis auf unvermeidbare Konflikte) zu erfüllen. Diese basieren meist auf einer Verringerung der Konfliktanzahl. Neben der Einführung eines Systems zur automatischen Verfeinerungen von Konflikteinheiten wird versucht, den Benutzer in die Konfliktbehebung bzw. -vermeidung einzubeziehen.

3.1 Akzeptanzprobleme durch Konflikte

Erstellt ein Client eine Transaktion, die später zurückgenommen wird, beobachtet der Benutzer folgenden Effekt: Durch das lokale Ausführen der Transaktion, werden die Änderungen zunächst auf dem Bildschirm angezeigt, doch kurz darauf verschwinden sie wieder, da der Server die Zurücknahme der Transaktion veranlaßte. Dies führt einerseits zu Verwirrung beim Benutzer („Warum geschieht dies?“) zum anderen zu Frustration, weil seine Arbeit zurückgenommen wurde und er sie manuell wiederholen muß.

Die automatisierte Wiederholung mittels undo-redo kann dieses Frustrationspotenzial kurzfristig kompensieren, kann dann jedoch zu starken Laufzeiteinbußen führen.

Bewirken die Interaktionen von zwei Teilnehmersystem τ_1 und τ_2 gleichzeitig modifizierende Operationen auf dem gleichen ModelObject x, so entsteht ein Konflikt[10]. In DyCE werden Konflikte gelöst, indem eine der beiden Transaktionen zurückgenommen wird.

Ist die Konfliktgranularität zu niedrig, entstehen viele Konflikte. Ist sie zu hoch, so steigt der Verwaltungsaufwand immens.

Durch die Zurücknahme von Transaktionen entsteht ein Frustrationspotenzial beim Benutzer, das zu Akzeptanzproblemen gegenüber der Software führt.

Dieses Potenzial läßt sich vermindern, wenn die betroffenen Benutzer von τ_1 und τ_2 frühzeitig auf einen bevorstehenden Konflikt hingewiesen werden. Es besteht die Hoffnung, daß beide sich selbständig koordinieren, so daß kein Konflikt auftritt.

Es gibt jedoch noch weitere Frustrationsquellen:

- Zu früh gegebene Warnungen, also dann, wenn die betroffenen Benutzer gar nicht die Absicht hatten einen Konflikt zu produzieren.

[10] Anmerkung: Genauer gesagt, entsteht der Konflikt nur dann, wenn die beiden Operationen den gleichen Slot manipulieren. Der Verständlichkeit halber wird in diesem Kapitel angenommen, daß alle Operationen den gleichen Slot betreffen und somit alle Operationen auf dem gleichen ModelObject einen Konflikt auslösen würden.

- In inadäquater Weise dargestellte Warnungen, z.B. durch modale Dialogfelder, die in großer Zahl im Benutzungsinterface erscheinen, so daß der Benutzer in seinem Arbeitsfluß gehindert wird.
- Für den Benutzer nicht nachvollziehbare Warnungen.
- Warnungen, die gegeben werden, obwohl gar keine Konflikte entstehen können.
- Konflikte, die durch einen einfachen Automatismus vermeidbar sind.

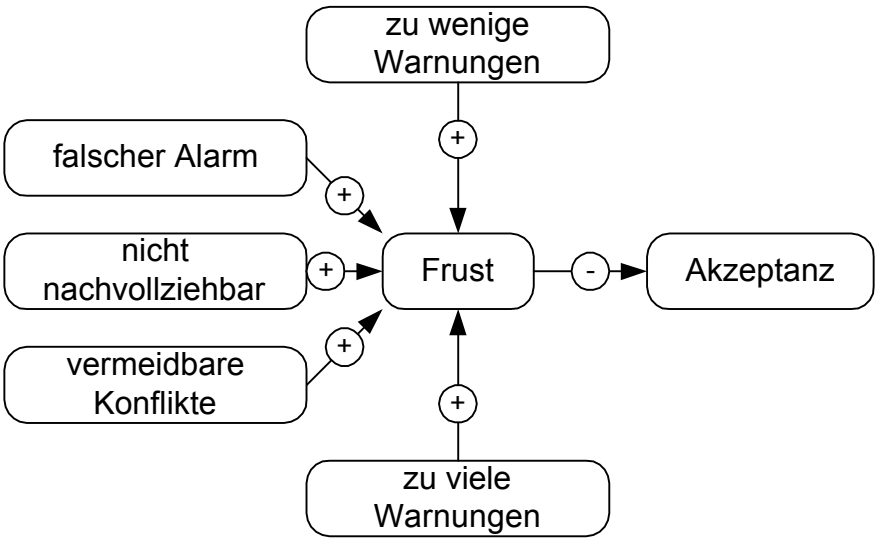


Abbildung 11: Frustrationsquellen der Benutzer in Notation nach [Dör96]. Variablen werden durch abgerundete Rechtecke dargestellt. Davon abhängige Variablen werden über einen Pfeil verbunden, wobei ein Plus, einen proportionalen, ein Minus einen antiproportionalen Zusammenhang darstellt.

Die ersten beiden Quellen hängen stark zusammen. Frühe Warnungen sollten in einer Weise dargestellt werden, die nicht die volle Aufmerksamkeit der Benutzers verlangt, z.B. als Meldung in der Statuszeile. Bei hoher Konfliktgefahr sollten Warnungen vom Benutzer sicher wahrgenommen werden. Trotzdem muß der Benutzer die Möglichkeit haben, die Warnung zu ignorieren.

Grundsätzlich sollten die Warnungen nicht derart früh gegeben werden, daß sie für den Benutzer nicht nachvollziehbar sind.

Benutzer können Warnungen insbesondere dann nicht nachvollziehen, wenn die Datenstruktur so ungünstig auf ModelObjects verteilt wurde, daß sie nicht der für den Benutzer logischen Aufteilung entspricht. Ein Beispiel dafür ist die Verteilung der Darstellung eines ModelObjects auf verschiedene Stellen in der Darstellungskomponente.

Werden Warnungen gegeben, wo keine Konflikte entstehen können, zeugt das von einer schlechten Konfliktbewertungsstrategie.

Die fünfte Frustrationsquelle bezieht sich insbesondere auf das Fehlen selbst einfachster Automatismen. Die im Kapitel 2.3 genannten Nebenläufigkeitskontrolle durch Transformationsverfahren stellt ein weitreichender Ansatz solcher Automatismen dar, der ausführlich in [BS98] besprochen wird. Meist reicht es jedoch aus, die Konfliktgranularität lokal zu modifizieren, um den Konflikt zu bannen. Dies kann eine räumliche Fragmentierung, die gemäß von Distanzen in der Darstellung vorgenommen wird, oder eine Zerlegung in die verschiedenen Eigenschaften einer Konflikteinheit sein.

Da DyCE automatisch den modifizierenden Zugriff auf verschiedene Eigenschaften[11] eines ModelObjects konfliktfrei unterstützt, wird im nächsten Abschnitt dargestellt, wie eine räumliche Fragmentierung vorgenommen wird.

3.2 Minimierung der Konfliktwahrscheinlichkeit durch Fragmentierung

Durch Spalten von ModelObjects kann die lokale Konfliktgranularität angepaßt werden. In welcher Weise sollte eine solche Fragmentierung vorgenommen werden? In diesem Kapitel wird versucht, Aussagen darüber zu treffen, wann ein ModelObject zerteilt werden sollte und wann nicht.

Sei T die Menge aller Teilnehmersysteme.

Definition (lastModified): Die Funktion lastModified: ModelObjects $\rightarrow$ (Zeitpunkte x T) ordne einem x $\in$ ModelObjects den Zeitpunkt seiner letzten Modifikation und das Teilnehmersystem, das diese Modifikation ausführte zu.

Definition (bearbeitungsstelleVon): Zu einem Zeitpunkt t_1 und zu gegebener Zeitlänge L bildet die Funktion $\text{bearbeitungsstelleVon}_{[t_1\text{-}L,\ t_1]}$: ModelObjects $\rightarrow$ Potenzmenge(T) ein ModelObject x auf die Menge der Teilnehmersysteme ab, die x innerhalb des Zeitintervalls $[t_1\text{-}L, t_1]$ bearbeitet haben.

Definition (Bearbeitungsstelle): Gilt für ein ModelObject x und Teilnehmersystem τ: $\tau \in \text{bearbeitungsstelleVon}_{[t_1\text{-}L,\ t_1]}(x)$ so ist x eine Bearbeitungsstelle von τ.

Im weiteren gelte: $\text{bearbeitungsstelleVon}_{[t_1\text{-}L,\ t_1]}(x) \subseteq \{\tau \in T \mid \text{lastModified}(x) = (t,\tau) \text{ und } t \in [t_1\text{-}L, t_1]\}$

Die genaue Definition von Bearbeitungsstellen bleibt der Anwendungsdomäne überlassen. Da L ein globaler Parameter ist, wird im folgenden auch kurz $\text{bearbeitungsstelleVon}_{t_1}$ geschrieben.

---

[11] Werden die verschiedenen Eigenschaften durch verschiedene Slots modelliert, löst es in DyCE keinen Konflikt aus, wenn diese parallel manipuliert werden.

Definition (Bearbeiten von ModelObjects): Ein $\tau \in T$ bearbeitet ein ModelObject x zum Zeitpunkt $t_1 \Leftrightarrow \tau \in$ bearbeitungsstelleVon$_{t1}$(x)

DyCE-Transaktionen laufen konfliktfrei ab, wenn alle Teilnehmersysteme auf verschiedenen ModelObjects operieren. Somit gilt: Falls zum Zeitpunkt t_1 gilt: $\exists$ Zeitlänge L, $\forall$ x $\in$ ModelObjects: |bearbeitungsstelleVon$_{[t1-L,\ t1]}$ (x)| $\leq 1 \Rightarrow$ Im Zeitintervall [t_1-L, t_1] traten keine Konflikte auf.

Durch statistische Verfahren läßt sich, in Abhängigkeit von Anzahl der Model-Objects, Anzahl der Benutzer und Länge des betrachteten Zeitintervalls L, die Wahrscheinlichkeit bestimmen mit der Konflikte auftreten.

Die Konfliktwahrscheinlichkeit läßt sich durch Veränderung der Konfliktgranularität senken. Dabei unterscheiden sich verschiedene Fragmentierungsverfahren:

Ein ModelObject x wird fragmentiert, wenn

- zwei Teilnehmersysteme in x navigieren und damit potenziell in der Lage sind einen Konflikt auszulösen (Vorbeugende Fragmentierung)
- ein Konflikt an x auftritt. Die zurückgenommene Transaktion wird in geeigneter Weise wiederholt. (Optimistische Fragmentierung)
- x Bearbeitungsstelle von einem Teilnehmersystem τ_1 ist und ein zweites Teilnehmersystem τ_2 x ebenfalls modifizieren will. Die Fragmentierung geschieht bevor die Transaktion von τ_2 auf x ausgeführt wird. (Verzögerte Fragmentierung)

Die vorbeugende Fragmentierung minimiert die Konfliktwahrscheinlichkeit und führt zu vielen kurzen Transaktionen. Allerdings wird auch bei lesendem Zugriff die Datenstruktur stark fragmentiert.

Die optimistische Fragmentierung fragmentiert die Datenstruktur am schwächsten. Ist die Anzahl der durchführbaren Transaktionen pro Zeiteinheit viel kleiner als die Anzahl der tatsächlichen Transaktionen, reicht allein die geeignete Wiederholung der Transaktion (ohne zusätzliche Fragmentierung), um diesen Konflikt aufzulösen (siehe hierzu auch 2.7.4).

Auf den ersten Blick scheint sich die verzögerte Fragmentierung nur durch den Zeitpunkt von der optimistischen Fragmentierung zu unterscheiden, nicht in der Anzahl der Fragmentierungen. Tatsächlich hat die parallele Modifikation einer Granularitätseinheit nur dann einen Konflikt zur Folge, wenn die Transaktionen zum gleichen Zeitpunkt eintreffen. Die verzögerte Fragmentierung fragmentiert bereits zu Beginn einer parallelen Modifikation – die optimistische Fragmentierung erst beim ersten Konflikt.

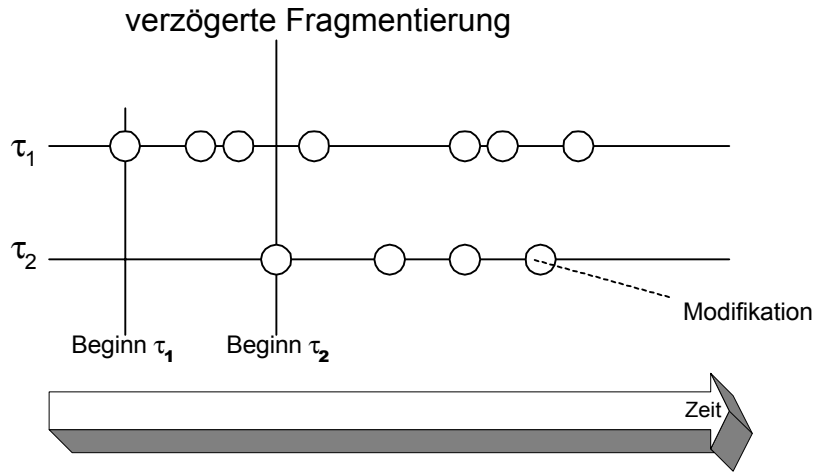


Abbildung 12: Zeitpunkt der Fragmentierung beim verzögerten Verfahren
Modifikationen auf den selben Konflikteinheiten werden durch kleine Kreise symbolisiert. Die Fragmentierung wird vorgenommen, sobald mehr als ein Teilnehmersystem innerhalb kurzer Zeitspanne eine Einheit bearbeiten.

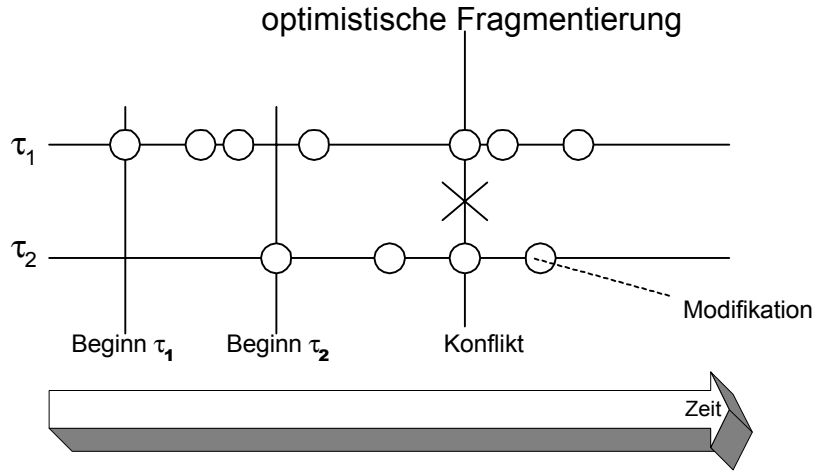


Abbildung 13: Zeitpunkt der Fragmentierung bei der optimistischen Variante
Modifikationen werden durch kleine Kreise, ein Konflikt durch ein Kreuz dargestellt. Die Fragmentierung wird erst bei vorgefallenen Konflikt vorgenommen.

Auch durch eine beliebig hohe Fragmentierung kann die Konfliktwahrscheinlichkeit nicht beliebig gesenkt werden. Hierzu ein Beispiel aus der Textverarbeitung: Das Dokument sei zeichenweise fragmentiert, so daß der Inhalt eines ModelObjects x genau ein Zeichen lang ist. Die Operation x.addZeichen(a) fügt das Zeichen a an den Inhalt an. Ein Teilnehmersystem τ_1 navigiere seinen Cursor hinter x und gebe das Zeichen a ein. Nun wird x.addZeichen(a) aufgerufen. Befinden sich alle Cursor der Teilnehmersysteme an verschiedenen Stellen, so treten keine Konflikte auf. Gibt es jedoch zwei Teilnehmersysteme, deren Cursor an der gleichen Stelle (also hinter dem gleichen ModelObject x) stehen, und beide geben gleichzeitig ein Zeichen ein, entsteht ein Konflikt, obwohl die Granularität der Fragmentierung feinstmöglich ist.

Unabhängig von der Wahl des Fragmentierungsverfahren sollten zerteilte Knoten wieder zusammengeführt werden, wenn der Umstand, der zur Fragmentie-

rung geführt hat, nicht mehr besteht. Je nach Anwendung kann es sinnvoll sein, vor dem Wiederverschmelzen noch eine gewisse Zeitspanne abzuwarten.

Würde die Fragmentierung nicht wieder aufgehoben werden, könnte eine langfristige Bearbeitung des Dokuments durch die Autoren zu einer vollständigen Fragmentierung führen, die einerseit sich in schlechter Performanz niederschlägt, andererseits keine Rückschlüsse von Fragmentierungsstatus auf aktuelle Bearbeitungsintensität zuläßt, die im Kapitel 3.5 besprochen wird.

3.3 Konfliktfreiheitsgarantie / Reservierungen

Anstatt von einer bestehenden Situation optimistisch auf die Zukunft zu schließen, kann man durch Reservierungszwang eine pessimistische Haltung einnehmen. Teilnehmersysteme können dann nur die ModelObjects modifizieren, die für sie reserviert sind.

Definition (Reservierungsfunktion): Sei die Funktion reservierung: ModelObjects $\rightarrow$ (Potenzmenge(T) $\cup$ {nichtreserviert}) gegeben. reservierung heißt die Reservierungsfunktion.

Definition (Reservierung): Für ein x $\in$ ModelObjects gebe reservierung(x) die Menge der Teilnehmersysteme an, die x modifizieren dürfen. Ist $\tau \in$ reservierung(x) so heißt x für τ reserviert und τ heißt in der Reservierung von x enthalten.

Die Reservierung muß sich mit den Bearbeitungsstellen vertragen, sprich: für ein für die Reservierungsfunktion global festgelegtes L muß zum Zeitpunkt t1 gelten: $\text{bearbeitungsstelleVon}_{[t1,\ t1-L]}(x) \subseteq$ reservierung(x).

Definition (Reservierungsinvariante): Sei x $\in$ ModelObjects. Für alle modifizierenden Operationen o auf x gilt: Teilnehmersystem τ darf x.o() nur aufrufen, wenn $\tau \in$ reservierung(x) oder wenn reservierung(x) = nichtreserviert.

Ist reservierung(x) = nichtreserviert, so darf jedes Teilnehmersystem x modifizieren – Konflikte müssen gesondert abgefangen werden.

Im folgenden werde die Reservierungsinvariante respektiert.

Definition (Konfliktfreiheitsgarantie): $\forall$ x $\in$ ModelObjects gilt: die Konfliktfreiheitsgarantie gilt an x zum Zeitpunkt t $\Leftrightarrow$ an x treten zum Zeitpunkt t keine Konflikte auf.

Lemma: Sei x $\in$ ModelObject. |reservierung(x)| $\leq$ 1 und reservierung(x) != nicht-reserviert zum Zeitpunkt t $\Leftrightarrow$ für x gilt zum Zeitpunkt t die Konfliktfreiheitsgarantie.

Definition (Reservierungsheuristik): Eine Reservierungsheuristik ist ein Algorithmus, der dafür sorgt, daß zu jedem Zeitpunkt die Reservierungsfunktion auf allen x $\in$ ModelObjects definiert ist und das Verträglichkeit mit den Bearbeitungsstellen besteht.

Da das Zerteilen (und das Wiederzusammensetzen) von ModelObjects modifizierende Operationen sind, muß einer Fragmentierungsänderung in x eine Reservierung von x für das entsprechende Teilnehmersystem vorausgehen.

Dabei ist es sinnvoll, die ModelObjects y_0, y_1, .. y_n zum ModelObject x zusammenzusetzen, wenn $\exists\ \tau \in T$ so daß $\forall\ i \in \{0, 1, .. n\}$ gilt reservierung(y_i) = $\{\tau\}$ oder nichtreserviert. Andernfalls gäbe es ein anderes Teilnehmersystem $\tau' \in T$ welches eines der y_i bearbeiten würde. Kurz nach dem Verschmelzungsvorgang wären konfliktierende Operationen durch τ und τ' hochwahrscheinlich, die wiederum zu einem Split führen würden.

Analog zu den Fragmentierungsverfahren unterscheiden sich auch hier verschiedene Reservierungsverfahren am Zeitpunkt der Reservierungsänderung.

- Reservierung beim Navigieren (vorbeugende Reservierung)
- Reservierung unmittelbar bevor die Reservierungsinvariante verletzt würde (Just-In-Time-Reservierung)
- Anpassung der Reservierung nach jeder Modifikation der Datenstruktur (aktualisierende Reservierung)

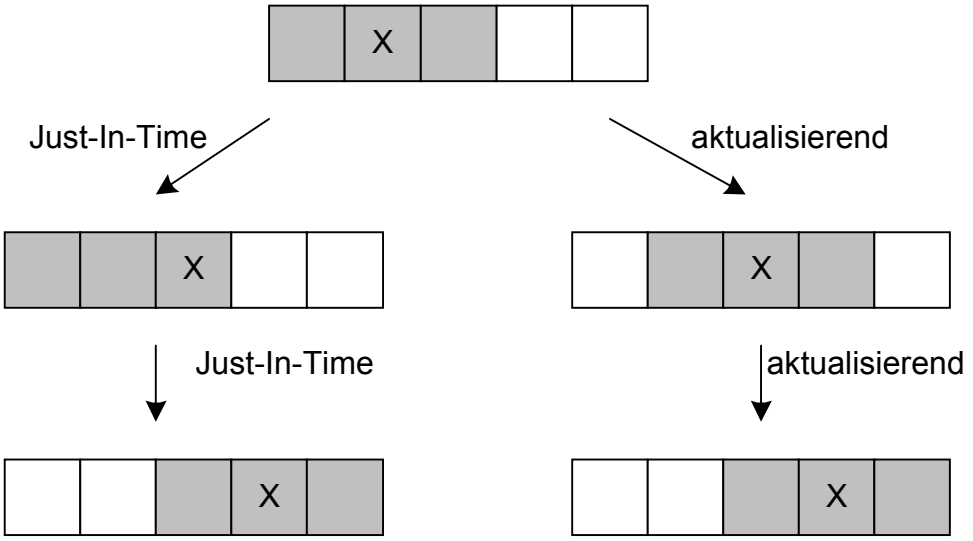


Abbildung 14: Unterschied zwischen Just-In-Time und aktualisierender Reservierung
Aneinanderhängende Rechtecke stellen ModelObjects dar. Ein Kreuz kennzeichne die aktuelle Bearbeitungsstelle, eine graue Hinterlegung die Reservierungen. Der Unterschied wird im mittleren Schritt deutlich: Die Just-In-Time Reservierung nimmt keine Reservierungsänderung vor, während die aktualisierende Reservierung sich bereits an die Situation anpaßt.

Das Just-In-Time-Verfahren liefert im Gegensatz zur aktualisierenden Reservierung ein zeitlich stabileres Verfahren: die Reservierungsfunktion wird nur dann modifiziert, wenn ein ModelObject Bearbeitungsstelle wird, das noch nicht reserviert wurde. Dadurch können jedoch sprunghafte Änderungen in der Reservierung auftreten. Die aktualisierende Reservierung sorgt hier für einen sanften Übergang.

Das Analogon zur optimistischen Fragmentierung – also eine Reservierung im Konfliktfall – widerspricht der Reservierungsinvariante und wird deshalb nicht weiter behandelt.

Bedingung	Automatismus	Ziel
Für x gilt die Konfliktfreiheitsgarantie		keine (DyCE-) Konflikte
Für x gilt die Konfliktfreiheitsgarantie nicht (|reservierung(x)| > 1)		
|bearbeitungsstelleVon(x)| > 1	Splitten, wenn möglich	ein unnötiger Konflikt wird gebannt / hoffentlich Konfliktfreiheit auf den neuen Knoten
x wurde seit längerer Zeit nicht reserviert	Zusammenfassen der Kindknoten von x, wenn möglich	Systemperformance verbessern

$\tau \in$ bearbeitungsstelleVon(x)	setze $\tau \in$ reservierung(x)	Hoffnung: τ wird x weiter bearbeiten
$\tau \in$ reservierung(x) $\tau \notin$ bearbeitungsstelleVon(x)	τ kann modifizierende Operationen auf x ausführen	Hoffnung: τ wird in naher Zukunft nur die reservierten Objekte bearbeiten

Tabelle 2 : Fragmentierungs- und Reservierungsautomatismen

Das Löschen der Reservierungsvermerke erfolgt analog zur Defragmentierung.

3.4 Reservierungsheuristiken

Neben dem Zeitpunkt der Reservierung, stellt sich die Frage nach der optimalen Reservierungsfunktion, die durch die Reservierungsheuristik angepaßt wird.

Die Antwort hängt im starken Maße von der Anwendungsdomäne ab. Hier seien ein paar grundsätzliche Ansätze dargestellt.

Definition (Grund): Für ein x $\in$ ModelObjects und ein $\tau \in$ T sei $G_{x,\tau}$ ein Ereignis, warum τ in die Reservierung von x aufgenommen werden muß. $G_{x,\tau}$ heiße Grund der Reservierungsänderung.

Je nach Fragmentierungsverfahren kann das Eintreffen von $G_{x,\tau}$ bedeuten, daß

- x Bearbeitungsstelle eines Teilnehmersystems τ wurde
- oder ein Teilnehmersystem τ in x navigiert

Im folgenden sei $G_{x,\tau}$ gerade eingetroffen.

Die einfache Reservierung sorgt in minimalistischer Weise dafür, daß die Reservierungsinvariante erfüllt ist. D.h. für alle ModelObjects x, auf die die einfache Reservierung angewandt wird, gilt: trifft $G_{x,\tau}$ ein, wird τ zu reservierung(x) hinzugefügt.

Diese Reservierungsheuristik ist natürlich wenig vorausschauend. Die weiteren Heuristiken versuchen aus dem Grund der Reservierung von x vorherzusehen, für welche ModelObjects y in naher Zukunft ein Grund vorliegen wird.

Existiert eine Nachbarschaftsrelation N auf den ModelObjects, dann reserviert die Nachbarschafts-Reservierung neben x all die ModelObjects y die in der unmittelbaren Nachbarschaft von x liegen. Also sei $Nachbarschaft_x$:= { y $\in$ ModelObjects | y N x} dann folgt aus dem Eintreffen von $G_{x,\tau}$, daß τ zu reservierung(x) und zu reservierung(y) $\forall$ y $\in$ $Nachbarschaft_x$ hinzugefügt wird.

Existiert ein Distanzmaß[12] D auf den ModelObjects, dann verallgemeinert die Umgebungs-Reservierung den Nachbarschaftsansatz in folgender Weise: Zu

[12] Für das Distanzmaß D gelte Positivität, Symmetrie, Transitivität und D(x, y) = 0 $\Rightarrow$ x = y.

gegebenem Abstand d sei $Umgebung_x$:= {y ∈ ModelObjects | D(x, y) ≤ d}. Aus dem Eintreffen von $G_{x,\tau}$ folge das Hinzufügen von τ zu reservierung(y) ∀ y ∈ $Umgebung_x$.

Die allgemeinste Heuristik ist ein Einflußheuristik.

Definition (Einfluß): Sei G = { $G_{x,\tau}$ | ∀ x ∈ ModelObjects und ∀ τ ∈ T} die Menge alle möglichen Gründe. Die Funktion einfluß: (ModelObjects x G) -> R^+ weise jedem Paar aus Grund und ModelObject eine Wertigkeit zu, die den Einfluß des Grundes auf das ModelObject angibt. Diese Funktion kann neben dem Verhältnis der beiden ModelObjects und der Art des Grundes auch andere Parameter berücksichtigen, wie die Länge der Zeitspanne, die seit dem Eintreffen des Grundes vergangen ist, oder die „Wichtigkeit" des Teilnehmersystems.

Definition (Einflußsumme): Zu einem ModelObject y und gegebener Menge M von Gründen berechnet die Einflußsumme $summe(y,M) = \sum_{G \in M} einfluß(y,G)$ die Summe aller Einflüsse von eingetroffenen Reservierungsgründen für y.

Sei M die Menge aller bisher eingetroffenen Reservierungsgründe. Für ein ModelObject y fügt die Einflußheuristik τ zu reservierung(y) genau dann hinzu, wenn summe(y, {g ∈ M | M = $G_{x,\tau'}$, τ = τ', x ∈ ModelObject}) ≥ E, wobei E ∈ R^+ ein festgelegter Parameter ist. Sinkt die Einflußsumme zu einem späteren Zeitpunkt unter den Schwellenwert E, so wird τ aus reservierung(y) entfernt.

3.5 Konflikt-Awareness

Durch Fragmentierung allein kann man nicht alle Konflikte vermeiden. Ist es stets nur einem Teilnehmer erlaubt, ein ModelObject zu reservieren, so gilt damit in jedem Fall Konfliktfreiheit, doch damit werden Sperrmechanismen nachgebildet, was die unter 2.4 beschriebenen Nachteile mit sich bringt. Kann man den Formalismus für Reservierungen dazu nutzen, dem Benutzer auf geeignete Weise mitzuteilen, in welchem Konfliktstatus sich die Datenstruktur an der von ihm bearbeiteten Stelle gerade befindet, wie in Kapitel 3.1 vorgeschlagen?

Awareness Informationen geben dem Benutzer Hinweise auf die Tätigkeiten der anderen Teilnehmersysteme. Konflikt-Awareness weist den Benutzer auf die Gefahr von möglichen Konflikten hin, so daß er sich frühzeitig koordinieren kann. Diese Koordination findet bei Sitzungen im selben Raum mündlich statt, sonst über Audio-/Videokommunikation oder textbasierte Diskussionskomponenten (z.B. Chat und Instant Messaging).

Durch die Reservierungsinvariante ist jedes Teilnehmersystem dazu gezwungen, einen Reservierungsvermerk an dem ModelObject zu setzen, das es zu bearbeiten gilt. Dabei können mehrere Vermerke auf dem gleichen ModelObject vorhanden sein. Die Reservierungsvermerke dienen dem System als Information darüber, welches Teilnehmersystem welche Stelle modifiziert. Diese Informationen werden ausgewertet und dem Benutzer in adäquater Weise prä-

sentiert. Werden z.B. Reservierungen durch farbliche Hervorhebungen in der Darstellungskomponente angezeigt, so deuten Färbungen in verschiedenen Farben eine Konfliktgefahr an.

Weiterhin könnten die Daten über Bearbeitungsstellen und Fragmentierungsänderungen zur Unterstützung des Gruppenbewußtseins und damit der Entwicklung neuer Synergien zwischen den Benutzern verwendet werden.

Wird ein ModelObject x fragmentiert, geschieht dies, um einen Konflikt zu vermeiden. Folglich war das Konfliktpotenzial vorher an x hoch und ist nun verringert. Wird ein ModelObject y für ein Teilnehmersystem τ reserviert, können andere Benutzer davon ausgehen, daß τ y in Kürze modifizieren wird. Ist y darüber hinaus noch Bearbeitungsstelle von τ, so läßt sich daraus folgern, daß y in einem (semantisch) unvollständigen Zustand ist, um nur einige Beispiele zu nennen.

Tabelle 3 zeigt eine vollständige Auflistung.

Situation	Awareness-Aspekt
Für x gilt die Konfliktfreiheitsgarantie	keine Konfliktgefahr
Für x gilt die Konfliktfreiheitsgarantie nicht (|reservierung(x)| > 1)	mittlere Konfliktgefahr
|bearbeitungsstelleVon(x)| > 1	hohe Konfliktgefahr
x wurde fragmentiert	zwei Teilnehmersystem „nähern“ sich einander ⇒ steigende Konfliktgefahr
x wurde zusammengefaßt	zwei Teilnehmersystem „entfernen“ sich von einander ⇒ sinkende Konfliktgefahr

Tabelle 3: Konflikt-Awareness

3.6 Synergie und Koordination

Die Informationen über den Konfliktstatus bewirken nicht nur eine Steigerung der Akzeptanz, sondern verstärken auch die Synergien zwischen den Benutzern. Durch Konfliktwarnungen werden sie über die Tätigkeit der anderen Benutzer informiert und können direkt darauf reagieren. Z.B. könnte ein Benutzer eine Diskussion mit dem Konfliktpartner führen und sich so mit ihm koordinieren. Je nach Anwendung sollte die Benutzungsschnittstelle komfortable Möglichkeiten bereitstellen, solche Interaktionen durchzuführen.

Neben Hinweisen auf Konflikte sind noch weitere Awareness-Informationen hilfreich, die sich aus dem Formalismus ableiten lassen.

Situation	Awareness-Aspekt
$\tau \in$ bearbeitungsstelleVon(x)	τ hat x bearbeitet / x ist möglicherweise in semantisch unvollständigem Zustand
$\tau \in$ reservierung(x) $\tau \notin$ bearbeitungsstelleVon(x)	τ bearbeitet x hochwahrscheinlich in naher Zukunft[13]

Tabelle 4: Group-Awareness

Weiterhin können die Werte von lastModified als Indizien für folgende Interpretationen dienen:

Information	Awareness-Aspekt
Zeitpunkt von x.lastModified	Wie alt bzw. semantisch abgeschlossen ist der Inhalt in x?
Teilnehmersystem von x.lastModified	Wer ist ein potenzieller Ansprechpartner für den Inhalt in x?

Tabelle 5: Mögliche Informationen über Zuständigkeit und Abgeschlossenheit

Beide Interpretationen sind in gewisser Weise gefährlich: Zwar werden Inhalte, die semantisch abgeschlossen sind, längere Zeit unmodifiziert überdauern, jedoch könnte der Bearbeiter auch seine Arbeit für die Mittagspause oder einen Kurzurlaub unterbrochen haben. Ebenso wird die für den Abschnitt verantwortliche Person irgendwann das zuletzt bearbeitende Teilnehmersystem sein. Wenn allerdings ein weiterer Benutzer einen Kommafehler korrigiert, macht ihn das noch nicht für den Inhalt verantwortlich.

3.7 Kombination der Verfahren

Die sinnvolle Verwendung der Reservierungs- und Fragmentierungsinformationen zur Interpretation von Konfliktgefahren setzt voraus, daß diese Interpretationen mit hoher Wahrscheinlichkeit zutreffen.

Dies stellt den Anspruch an die Reservierungsheuristik, nicht unnötig viele ModelObjects zu reservieren. Es sollte keine vorbeugende Reservierung vorgenommen werden, denn bei reinem lesendem Zugriff können keine Konflikte entstehen.

Durch die sprunghafte Reservierungsänderung beim Just-In-Time-Verfahren werden Konfliktpotenziale zunächst möglicherweise falsch bewertet, was zu verspäteten Konfliktwarnungen führt. Da die aktualisierende Reservierung stets

[13] unter der Voraussetzung, daß die Reservierungsstrategie die Interpretation als Konfliktfrühwarnsystem zuläßt.

die optimale Reservierungsfunktion herzustellen versucht, ist sie zur Bewertung des Konfliktpotenzials besser geeignet.

Eine Reservierungsheuristik sollte grundsätzlich versuchen, zu jedem Zeitpunkt eine für Modifikationen in naher Zukunft optimale Reservierung vorzunehmen. Je größer die Wahrscheinlichkeit, daß die Vorhersage eintritt, desto weniger ModelObjects werden unnötigerweise reserviert und desto geeigneter ist die Interpretation als Konfliktfrühwarnsystem. Für die Umgebungsheuristik muß daher der Umgebungsparameter d und für die Einflußheuristik der Schwellenwert E korrekt gewählt sein. Da diese Werte stark von der Arbeitsweise der Gruppe, bzw. der Benutzer abhängt, sollten diese Werte statistisch adaptiert werden.

Die Fragmentierung sollte für den Benutzer nachvollziehbar sein. Es empfiehlt sich daher nicht die optimistische Fragmentierung anzuwenden, da diese stark von der zeitlichen Folge der Modifikationen abhängt. Sie weckt beim Benutzer den Eindruck, bei gleichen Modifikationen unterschiedlich zu reagieren.

Ähnliches gilt für die vorbeugende Fragmentierung. Der Benutzer wird möglicherweise nicht verstehen, warum sich die Anzeige ändert, obwohl keine Modifikationen an den Daten vorgenommen werden.

Weiterhin sollte eine Reservierungsanpassung erst nach einer Fragmentierungsadaption stattfinden, da sonst Warnungen formuliert werden, obwohl das System die Konfliktfreiheit garantieren könnte (siehe hierzu auch 3.1)

Reserv.\Fragment.	**vorbeugend** - hohe Fragmentierung	**verzögert**	**optimistisch** - Ergebnis erscheint zufällig
vorbeugend	- Änderung bei Nur-Lese Zugriff	- unnötige Warnungen	- unnötige Warnungen
aktualisierend			- unnötige Warnungen
Just-In-Time - evtl. falsche Bewertung			

Tabelle 6: Vergleich von verschiedenen Fragmentierungs- und Reservierungskombinationen
Die Nachteile der einzelnen Verfahren sind im Tabellenkopf, die aus der Kombination von Verfahren resultierenden Nachteile im Tabellenrumpf notiert. Die grau hinterlegten Zellen deuten an, welche Kombinationen problematisch sind. Allein die Kombination aus verzögerter Fragmentierung und aktualisiernder Reservierung hat keine Nachteile.

Wie die Gegenüberstellung in Tabelle 6 zeigt, ist eine Kombination aus verzögerter Fragmentierung und aktualisierender Reservierung bei der Implementierung von Konflikt-Awareness am geeignetsten.

Aus den starken Auswahl-Einschränkungen ergibt sich die Frage, warum man nicht eine Fragmentierungs- und Reservierungs-Kombination zur Anzeige des

Konfliktpotenzials und eine weitere für die tatsächliche Datenfragmentierung bzw. Modifikationssperre verwendet. Ein Argument gegen ein solches System sind Akzeptanzprobleme, die entstehen, wenn das System nicht in der Weise reagiert, wie der Benutzer es vermutet. Dies könnte leicht auftreten, wenn unterschiedliche Systeme zur Anzeige und zur Durchführung verwendet werden.

3.8 Zusammenfassung der Nebenläufigkeitstrategien

Das Kapitel beschäftigte sich mit Vorgehensweisen, mit denen die Nebenläufigkeit verbessert werden kann. Dabei wurden drei Methoden genannt: Zuerst die Adaption der Konfliktgranularität, mit der eine dynamische Anpassung der Granularität an die Bedürfnisse des Benutzers erreicht wird, um Anforderung 10 (Vermeidung von Konflikten) zu erfüllen. Des weiteren die Reservierung einzelner Segmente für bestimmte Benutzer, wodurch ein Schreibschutz für Segmente entsteht, die gerade bearbeitet werden, wodurch die Vermeidung der Konflikte garantiert ist, jedoch auf Kosten erhöhter Wartezeit der Benutzer. Zuletzt wurde noch die Verwendung der Reservierungfunktion, allerdings ohne Schreibschutz, zur Bewertung von Konfliktpotenzialen als Möglichkeit zur Konfliktfrühanzeige diskutiert, mit der Anforderung 11 (Hinweis auf unvermeidbare Konflikte) unterstützt wird.

Man geht bei der Verwendung der Reservierung mit Schreibschutz davon aus, daß die Benutzer des Systems keine Absprache untereinander halten, während die Verwendung der Reservierungsverfahren nicht zum Schreibschutz, sondern zur Anzeige von Konfliktpotenzialen, eine synergetische Zusammenarbeit der Benutzer annimmt.

Eine grundsätzliche Verbesserung, für die hier vorgestellten Algorithmen kann durch die Verwendung einer Datenstruktur erreicht werden, in der DyCE-Konflikte möglichst genau dann auftreten, wenn semantisch gegenläufige Modifikationen vorgenommen wurden. Dann erst funktioniert die Bewertung des Konfliktpotenzials wie in 3.5 beschrieben.

4 Datenstruktur für gemeinsame Datenräume

Dieses Kapitel beschäftigt sich mit dem zweiten Teil der konzeptionellen Lösung. Der besteht aus einer Datenmodellierung, die eine möglichst hohe Rückschlußmöglichkeit von DyCE-Konflikten auf gegenläufige Modifikationen zuläßt. Ein Rückschluß ist um so plausibler, je mehr das Datenmodell die Struktur der Problemdomäne widerspiegelt, da dann semantisch eng korrelierende Datensegmente gruppiert werden.

Ist dies erfüllt, wird Anforderung 11 (Hinweis auf unvermeidbare Konflikte) durch die im vorigen Kapitel vorgestellten Verfahren zur Konfliktpotenzialbestimmung zuverlässig unterstützt und es wird möglich Anforderung 4 (Parallele Modifikation) zu erfüllen, wenn die Segmentierung der Daten günstig gewählt wird.

Im Folgenden wird ein allgemeines Paradigma erarbeitet, das sich leicht an verschiedene Problemdomänen anpassen läßt.

Es wird beschrieben, wie die in Kapitel 3 vorgestellten Verfahren zur Verbesserung der Nebenläufigkeitskontrolle angewendet werden können und eine Anbindung an XML skizziert.

4.1 Strukturgraphen

Im Folgenden werden einige grundlegende Definitionen und Sätze der Mengen- und Graphtheorie vorausgesetzt. Sie können in „Anhang A: Grundbegriffe der Mengen- und Graphentheorie“ nachgelesen werden.

Da hier nur gerichtete Graphen betrachtet werden, wird anstatt des präziseren Begriffes „Bogen“ von gerichteten Kanten oder Kanten gesprochen.

4.1.1 Theorie der Strukturgraphen

Die Verwendung von gerichteten Graphen als Datenmodellierung bietet die größtmögliche Flexibilität zur Strukturierung von unbekannten Problemdomänen. Die gerichteten Graphen werden zur hierarchischen Gliederung des Problems herangezogen.

Definition (Strukturgraph, Knotenmenge, Kantenmenge): Ein Strukturgraph ist ein gerichteter Graph G = (V,E) mit der Knotenmenge V und der Kantenmenge $E \subseteq V \times V$.

Ein Knoten v entspricht einer logischen Struktureinheit. Aus XML-Sicht ist ein Knoten des Strukturgraphen äquivalent zu dem Inhalt eines XML-Knotens zusammen mit seinen XML-Tags. Aus Sicht der Textverarbeitung entspricht ein Knoten v einer logischen Struktureinheit wie Kapitel, Absatz, Wort oder Bild.

Definition (Inhalt, content): Ein Knoten $v \in V$ eines Strukturgraphen trägt im Allgemeinen Inhalt. Dieser wird durch die an v inzidenten Kanten zu den Inhalten der anderen Knoten des Graphen in Beziehung gesetzt. Sei I die Menge aller gültigen Knoteninhalte. Definition (Inhaltsfunktion): Die Funktion content: $V \rightarrow I$ ordnet jedem Knoten seinen Inhalt zu.

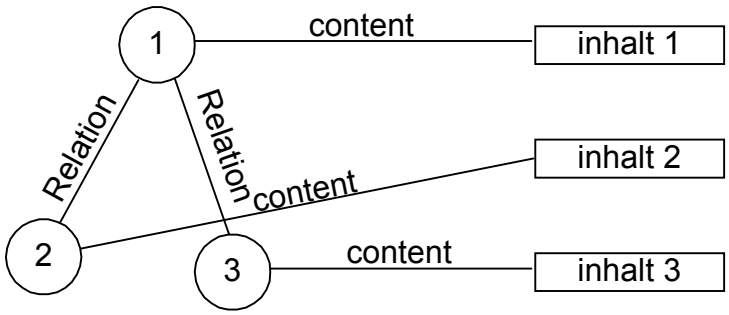


Abbildung 15: Relationen zwischen Knoten und deren Inhaltsobjekte
Die Knoten werden durch Kreise, die damit assoziierten Inhaltsobjekte werden durch Rechtecke dargestellt. Kanten zwischen Knoten repräsentieren bestimmte Relationen, die sich auf Inhaltsobjekte übertragen lassen.

Die Kanten beschreiben die Beziehungen zwischen den Knoten respektive deren Inhalten. Eine essentielle Beziehung ist die „Enthält"- Relation. Gibt es eine Kante e = (u, v) so ist der Inhalt von v in u enthalten. In einer Anwendung auf Textverarbeitungen könnte u ein Kapitel repräsentieren und v einen Absatz dieses Kapitels. In XML-Dokumenten wird diese Semantik durch Schachtelung ausgedrückt. Die „Enthält"-Relation setzt Reflexivität und Transitivität voraus.

Alle anderen Kanten sind sogenannte Referenzkanten. Eine Referenzkante (u, v) stellt einen Verweis von Knoten u auf Knoten v dar. Die genaue Semantik ist nicht festgelegt. Eine Anwendung von Referenzkanten sind Querverweise, also Texteinträge wie „siehe Kapitel 2" oder Hyperlinks, die eine aktive Schaltfläche repräsentieren, mittels derer man an eine andere Stelle im Dokument navigieren kann.

Definition (Enthältkanten, Referenzkanten): Die Kantenmenge $E = EC \cup ER$ untergliedert sich in die Menge der Enthältkanten EC und die Menge der Referenzkanten ER. Es gilt $EC \cap ER = \varnothing$.

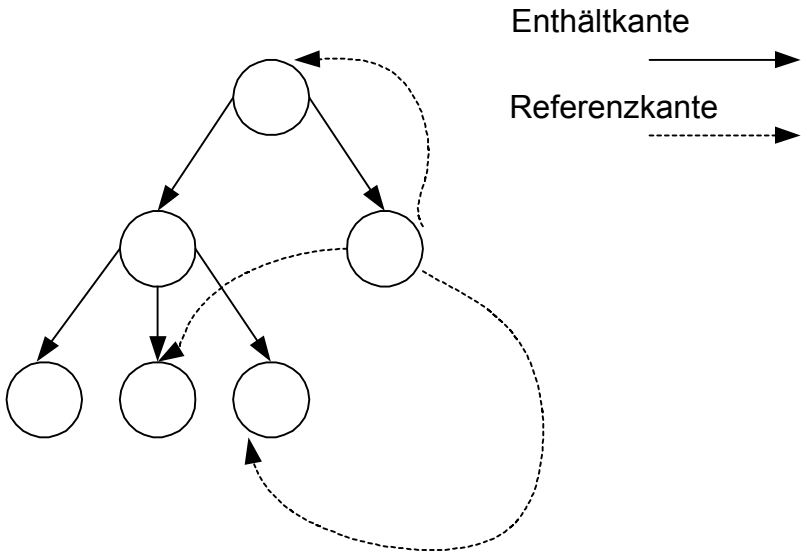


Abbildung 16: Aufbau eines Strukturgraphen mittels Enthält- und Referenzkanten
Die Knoten werden durch Kreise repräsentiert. Wie aus der Legende ersichtlich, werden Enthältkanten als normale Pfeile, Referenzkanten als gestrichelte Pfeile dargestellt. Diese Notation wird im restlichen Dokument beibehalten.

Definition (direktes bzw. mittelbares Enthaltensein): Für die Knoten $u,v,w \in V$ seien die Kanten (u,v) und $(v,w) \in EC$. v heiße in u mittelbar oder direkt enthalten, w heiße in u indirekt oder mittelbar enthalten. Die mittelbare Enthaltenseinsrelation ist transitiv.

Definition (ausgehende Kanten): Für alle $v \in V$ sei die Menge der ausgehenden Kanten $Ev = \{ e \in E \mid e = (v, u) \in E, u \in V \}$.

Definition (Kind): Ev enthält alle Kanten, deren Endpunkte die Menge der Kinder $Kv = \{ w \in V \mid (v,w) \in Ev\}$ von v beschreiben.

Definition (Blattmenge): Die Menge $B = \{ v \in V \mid Ev = \emptyset \}$ heiße Blattmenge, Menge der Blätter oder Menge der Terminalknoten. V\B heißt Menge der inneren Knoten oder Menge der Nichtterminalknoten.

Eine besondere Art der Beziehung ist „Attribut[14] von“ $\subseteq V \times V$. Ein Knoten u ist Attributkind von v, wenn u ein kompositioneller Bestandteil von v ist. Alle weiteren Kinder w von v sind dagegen aggregative Bestandteile von v. Die nähere Unterscheidung wird der Anwendung überlassen.

[14] von lat. atribuere (zuweisen, beigeben, verleihen, zuschreiben). Attribut wird hier nicht mit dem deutschen Wort „Eigenschaft“ gleichgesetzt.

Definition (Attributkanten, Attributkind): EA $\subset$ E heiße Menge der Attributkanten. Ein Kind u von v heiße Attributkind von v genau dann, wenn (v,u) $\in$ EA ist.

Die Menge der Attributkanten kann sowohl Elemente von EC als auch von ER enthalten.

Dazu ein Beispiel aus der Textverarbeitung: Der Knoten v steht für ein Kapitel. v besitzt die Kinder u und w. u repräsentiert eine Überschrift, w einen Absatz. Ein Kapitel kann ohne seine Überschrift nicht bestehen[15]. Weiterhin gelten für Überschriften besondere Einschränkungen. z.B. gibt es genau eine Überschrift zu einem Kapitel, die vor allen anderen Kindern von v aufgeführt wird. Daher ist u ein Attribut von v, bzw (v,u) $\in$ EA der Attributkanten. w ist kein Attribut, denn w ist ein nicht näher spezifizierter Kindsknoten von v.

Definition (Wurzelknoten): In jedem Strukturgraphen gebe es genau einen Knoten mit Innengrad null. Dieser heißt Wurzelknoten.

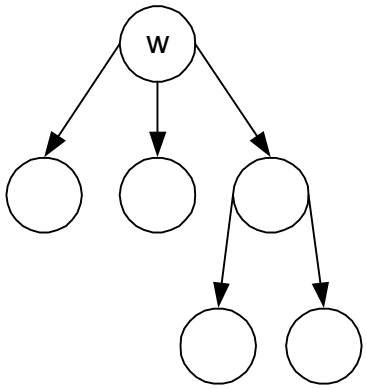


Abbildung 17: Wurzelknoten w eines Strukturgraphen.

Definition (Primärstruktur): Für jeden Strukturgraphen G gelte zusätzlich, das der Teilgraph T = (V, EC) eingeschränkt auf die Menge der Enthältkanten einen G aufspannenden Baum bildet. T heißt Baum der Primärstruktur. (Genauer gesagt, ist die Primärstruktur eine Arboreszenz, da sie alle Knoten enthält, von denen jeder Zielknoten von maximal einer Kante ist, es eine eindeutige Wurzel, den Wurzelknoten des Strukturgraphen, gibt, und T zusammenhängend ist.)

Sei R die „Enthält“-Relation. Aus der Primärstruktur folgt damit, daß für den Wurzelknoten w $\in$ V und $\forall$ Knoten v $\in$ V gilt: w R v. Für den Inhalt der Knoten v heißt daß, das der direkte bzw. mittelbare Inhalt von v auch mittelbarer Inhalt von w ist.

---

[15] In Microsoft Word wird als Kapitelinhalt der Teil des Textes definiert der sich zwischen zwei Überschriften der gleichen Ordnung befindet.

4.1.2 Definition geordneter Strukturgraph

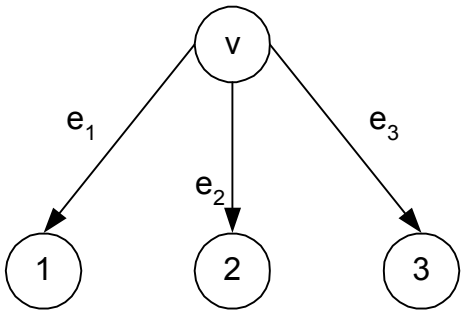
Die meisten Anwendungen fordern eine Betrachtung der Datensegmente in einer bestimmten Reihenfolge. Dies bedeutet, daß zwischen den Kindern v eines Vaterknotens u eine lineare Ordnung existieren muß.

Sei G ein Strukturgraph mit G = (V,E)

Definition (Kantenordnung): Eine Kantenordnung $<_v \subseteq Ev \times Ev$ ist eine vollständige, lineare Ordnung zwischen Kanten, die aus dem gleichen Knoten ausgehen.

e = (v, u) heißt i-te aus v ausgehende Kante :$\Leftrightarrow$ $|\{e' \in Ev \mid e' <_v e\}| == i+1$

Die Kantenordnung überträgt sich auf die Ordnung der Kindsknoteninhalte. So gilt für die Knoten u, v, w $\in$ V mit (u, v) und (u, w) $\in$ EC und (u, v) $<_v$ (u, w): v steht vor w.



$e_1 <_v e_2 <_v e_3$

Abbildung 18: Kantenordnung

Aus Sicht der Textverarbeitung ergibt sich durch die Kantensortierung die Reihenfolge der Darstellung der Geschwister. Für allgemeine XML-Dokumente speichert die Ordnung die Reihenfolge der unter dem gleichen Knoten verschachtelten Einträge gleicher Ebene.

Werden neue Kanten in einen Strukturgraphen eingefügt, so muß die Kantenordnung angepaßt werden, um die Vollständigkeit der Ordnung zu erhalten.

Definition (Index): Sei e $\in$ E eine Kante. Sei $<_v \cap (Ev\backslash\{e\} \times Ev\backslash\{e\})$ eine vollständige, auf $<_v$ jedoch nur eine teilweise definierte Ordnung. Ein Index s bildet (e, $<_v$) auf $<_v'$ ab, so daß gilt $<_v \subset <_v'$ und $<_v'$ ist eine vollständige lineare Ordnung. Man sagt, der Index s sortiert die Kante e in die bestehende Menge der aus v ausgehenden Kanten ein.

Definition (Anfangsindex): Der Index $s_{begin}(e, <_v) := (<_v \cap (Ev\backslash\{e\} \times Ev\backslash\{e\})) \cup (\{e\} \times Ev\backslash\{e\})$ reihe e als das erste Element in die Ordnung ein.

Definition (Endindex): Der Index $\underline{s_{end}}(e, <_v) := (<_v \cap (Ev\backslash\{e\} \times Ev\backslash\{e\})) \cup (Ev\backslash\{e\} \times \{e\})$ reihe e als das letzte Element in die Ordnung ein.

Definition (i-ter Index): Der Index $\underline{s_i}(e, <_v) := <_v \cup (\{e' \in Ev \mid e' \text{ ist j-te aus v ausgehende Kante, } j \leq i\} \times \{e\}) \cup (\{e\} \times \{e' \in Ev \mid e' \text{ ist j-te aus v ausgehende Kante, } j > i\})$ reihe e nach der i-ten Kante ein.

Definition (initialer Index): Der initiale Index $\underline{s_\emptyset}(e, <_v) := \emptyset$ kann nur gewählt werden, wenn e die erste Kante ist, die an v angehängt wird.

Definition (geordneter Strukturgraph): Sei G = (V,E) ein Strukturgraph. $\forall\, v \in V$ gibt es eine eindeutige Kantensortierung $<_v$. Sei $< := \bigcup_{\forall v \in V} <_v$ die Vereinigung der Ordnungen. Dann ist (G, <) ein <u>geordneter Strukturgraph</u>.

Lemma (geordneter Primärbaum): Ist T = (V, EC) der Baum der Primärstruktur von G und (G, <) ein geordneter Strukturgraph, dann ist (T, <) ein geordneter Baum.

Beweis:

Nach der Definition von [App95] ist ein Baum $T = (r: T_1, ..., T_m)$ (r ist die Wurzel von T und T_1 bis T_m seine Teilbäume) genau dann ein geordneter Baum, wenn es eine Ordnung <‘ gibt mit $T_1 <' T_2 <' ... <' T_m$ und die Teilbäume selbst wieder geordnete Bäume sind.

Sei $\mathrm{T_V}$ die Menge aller (Teil-)Bäume mit Knoten aus V.

Sei gesuchte Ordnung $<' \subset \mathrm{T_V} \times \mathrm{T_V}$ folgendermaßen definiert: $\forall$ T, T‘ $\in \mathrm{T_V}$ sei T <‘ T‘ :$\Leftrightarrow$ T = (r; ...) und T‘ = (r‘; ...) und r < r‘.

Damit folgt die Behauptung.

4.1.3 Split-Operation

Durch die Aufteilung der Daten sinkt die Konfliktwahrscheinlichkeit von gleichzeitigen Modifikationen. Es wird eine statische Konfliktgranularität modelliert. Hierbei stellt sich die Frage nach dem Grad der Granularität. Als Beispiel aus der Textverarbeitung ist für Bücher eine Segmentierung auf Kapitelebene vollkommen ausreichend, während für Briefe eine Zerteilung nach Sätzen passender ist.

Eine zu grobe Konfliktgranularität führt weiterhin zu Konflikten, die keinen Rückschluß auf gegenläufige Modifikationen zulassen. Eine zu feine Konfliktgranulariät, bewirkt eine Steigerung der Antwortzeit des Systems.

Eine adaptive Granularität ermöglicht eine optimale Anpassung des Systems an die Bedürfnisse der Benutzer. Zusätzlich begünstigt eine optimale Granularität die Entstehung von Synergieeffekten zwischen den Benutzern (siehe 3.8).

Die Adaption geschieht durch dynamisches Zerteilen und Wiedervereinigen von Knoteninhalten.

I ist die Menge aller gültigen Knoteninhalte, die durch die Funktion content: $V \rightarrow I$ assoziiert wird.

Definition (Partitionierbarkeit): Gibt es eine invertierbare Transformationsfunktion $Tsplit_n$: $I \rightarrow I^n$ mit $Tsplit_n(inhalt) = (inhalt_0, inhalt_1, ... inhalt_{n-1})$, wobei inhalt, $inhalt_0, inhalt_1, ... inhalt_{n-1} \in I$ sind, so

a) heißt inhalt partitionierbar

b) heißt es, der Knoten v mit content(v) = inhalt unterstützt die Split-Operation.

Die eindeutig bestimmte invertierte Funktion heißt $Tjoin_n := Tsplit_n^{-1}$.

4.1.4 Definition inhaltspartitionierter Strukturgraphen

Durch die Transitivität der „Enthält“-Relation ergeben sich direkte und mittelbare Knoteninhalte. Direkter Knoteninhalt kann durch Zerteilen in mittelbaren Knoteninhalt gewandelt werden. Eindeutiges Wiedervereinigen von Knoten läßt sich dann besonders leicht definieren, wenn nur Blätter direkten Inhalt tragen und es eine wie oben definierte Kantenordnung gibt.

Definition (inhaltspartitionierter Strukturgraph): Sei G = (V,E) Strukturgraph wobei $\forall v \in V \setminus B$ gilt: content(v) = null. G heiße inhaltspartioniert.

In inhaltspartitionierten Graphen besitzen Nichtterminalknoten keinen direkten Inhalt.

Bemerkung: Jeder Strukturgraph G, dessen innere Knoten direkten Inhalt tragen, läßt sich bijektiv in einen inhaltspartitionierten Strukturgraphen G‘ transformieren. Dazu wird der inhalt des Nichtterminalknotens v in ein spezielles Attributkind u von v ausgelagert. Je nach Anforderungen der Problemdomäne muß die Kante (v, u) an geeigneter Stelle in die Kantenordnung $<_v$ eingefügt werden. Durch die Einschränkung auf inhaltspartitionierte Strukturgraphen wird die Menge der damit lösbaren Probleme nicht verringert.

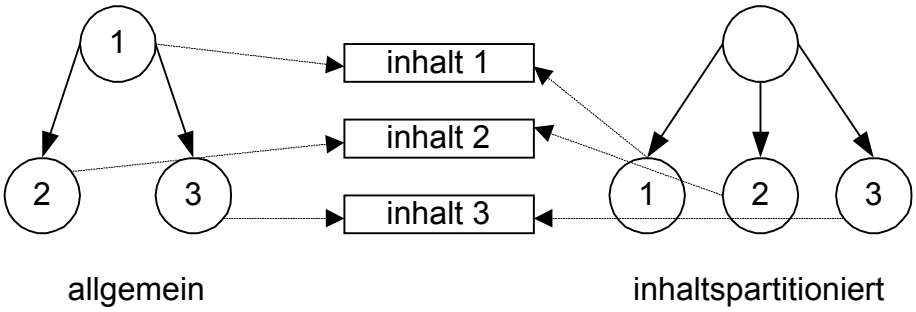


Abbildung 19: Zusammenhang zwischen allgemeinem und inhaltspartitioniertem Strukturgraph

Für den Rest der Arbeit werden nur geordnete, inhaltspartitionierte Strukturgraphen betrachtet, da diese leichter zu handhaben sind.

4.2 XML und Strukurgraphen

In Kapitel XX wurde die Anforderung 3 (XML-Export) beschrieben, in der die Möglichkeit des Exportierens nach und des Importieres von XML gefordert wurde. Dieses Kapitel beschreibt, wie diese Anforderung erfüllt werden kann.

XML ist ein Format, um strukturierte Daten so zu speichern, daß sie sowohl auf plattformunabhängige Weise maschinell verarbeitet, als auch von Menschen gelesen werden können. Die Datensegmente werden durch sogenannte Tags ausgezeichnet, die ineinander verschachtelt werden können, so daß sich baumartige Strukturen ergeben. Ein Datensegment zusammen mit seinen Tags wird Element genannt. Ein Element kann, neben verschachtelten Kindelementen, Inhalt eines bestimmten Datentyps wie Zeichenkette, ganze Zahlen oder gekapselte Daten (CData) enthalten. Elemente können zusätzlich mehrere Attribute besitzen, wobei ein Attribut eine Zeichenkette zu einem einfachen Datentyp zuordnet. Weitere Verschachtelungen sind bei Attributen nicht möglich.

4.2.1 Ansatz von Beech, Malhotra und Rys

Beech, Malhotra und Rys beschreiben in [BAM99], wie man die Komponenten von XML-Dokumenten und die Beziehungen dazwischen als einen gerichteten Graphen repräsentieren kann. Sie geben Operationen an, um auf die Daten im graphtheoretischen Sinne zuzugreifen und sie zu manipulieren.

Dabei wird in der einfachsten Form ein Baum aufgebaut, der die hierarchische Struktur der XML-Tags darstellt. Eine solche Schnittstelle wurde bereits durch die XML-Erweiterung „Document Object Model“ (DOM) implementiert [W3C]. Die Kanten eines solchen Graphens werden Element beinhaltende Kanten[16] ge-

[16] englisch: element containment edges

nannt. Die Autoren führen weiterhin spezielle Kanten zu den Attributen eines Elements ein (Attributkanten) und geben den Elementen eindeutige Identifikationsnummern (ID), über die sie darauf verweisen können. Diese Verweise nennen sie Referenzkanten.

Da die Referenzkanteninformationen über Attribute im Element gespeichert werden, enthält der Graph für jede Referenzkante eine redundante Attributkante.

Das folgende Beispiel demonstriert die Transformation des XML Quellcodes in den Graphen:

Gegeben sei ein XML-Fragment. Dieses definiere ein Element mit dem Namen A, daß die Elemente B, C und D enthält. B definiert ein Attribut „attr“, dessen Wert die Zeichenkette „b“ ist. C definiert eine Identifikationsnummer im Attribut „No“. D enthält ein Attribut mit dem Namen „link“ daß eine Referenz auf die Identifikationsnummer von C enthält.

```
<A>
        <B attr="b" />
                some content
        </B>
        <C No::ID="55" />
        <D link::IDREF="55" />
</A>
```

Quellcode 1: Beispiel für einen Graphen in XML

Die Beschriftungen der Elemente und Attribute gehen in die Beschriftung der Kanten über. Fettgedruckte Kantenbeschriftungen indizieren Kanten, die ursprüngliche Daten referenzieren, nicht fettgedruckte Beschriftungen deuten auf Kanten hin, die der Mechanismus erforderlich macht. Die Knoten enthalten die Werte der Attribute, bzw. der Elementinhalte. Die Nichtterminalknoten repräsentieren Elemente und wurden der Lesbarkeit halber mit an den Tag angelehnten Variablennamen beschriftet.

Mit breiter Strickstärke gezeichnete Kanten sind Element beinhaltende Kanten, Kanten mit dünner Strichstärke sind Attributkanten und gestrichelte Kanten sind vom Typ Referenzkante.

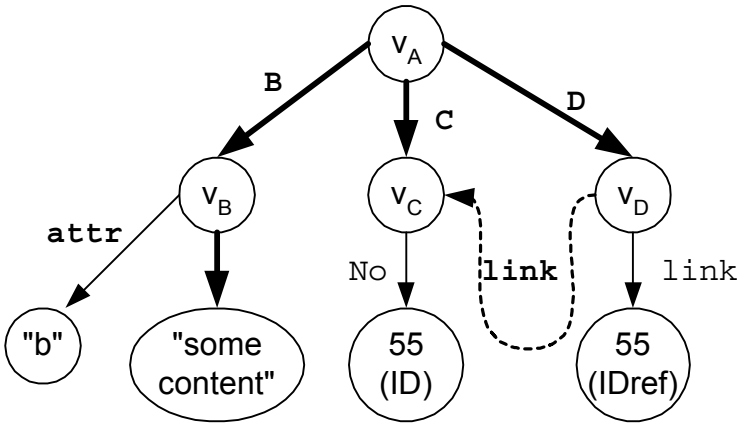


Abbildung 20: Graphrepräsentation von Quellcode 1

Zu Beachten ist das doppelte Auftreten der Kantenbeschriftung „link“. Die Referenzkante ergibt sich aus der Attributkante „link“ zusammen mit der rückwärts aufgelösten Attributkante „No“ von v_C.

[BAM99] extrahieren aus diesem Modell eine lineare Ordnung auf Kanten gleichen Typs, die aus dem gleichen Knoten ausgehen, die vollständig definiert ist. Die Ordnung basiert auf der Reihenfolge im XML-Dokument mit der die Knoten definiert werden. Da Attributkanten zu einem früheren Zeitpunkt definiert werden als Element enthaltende Kanten und Referenzkanten aus den Attributkanten hervorgehen, ist die Ordnung nicht geeignet, Element enthaltende Kanten und Referenzkanten zueinander in Reihenfolge zu setzen.

4.2.2 Ordnungen auf Kanten verschiedenen Typs

Dieses Defizit läßt sich ausgleichen, indem man spezielle Elemente einfügt, die die Referenzkanten speichern. Die Auflösung einer Referenzkante erfolgt dann über die Element enthaltende Kante zum referenzspeichernden Element, dann über dessen Attributkante zur ID-Referenz und darüber zum verwiesenen Element.

```
<A>
        <B No::ID=“44“ />
        <C>
                <c1 link::IDref=“44“>
                <c2>
                        c2
                </c2>
                <c3 link::IDref=“55“>
        </C>
        <D No::ID=“55“ />
</A>
```

Quellcode 2: Beispiel für einen Graphen in XML mit einer Kantenordnung zwischen Kanten verschiedenen Typs.

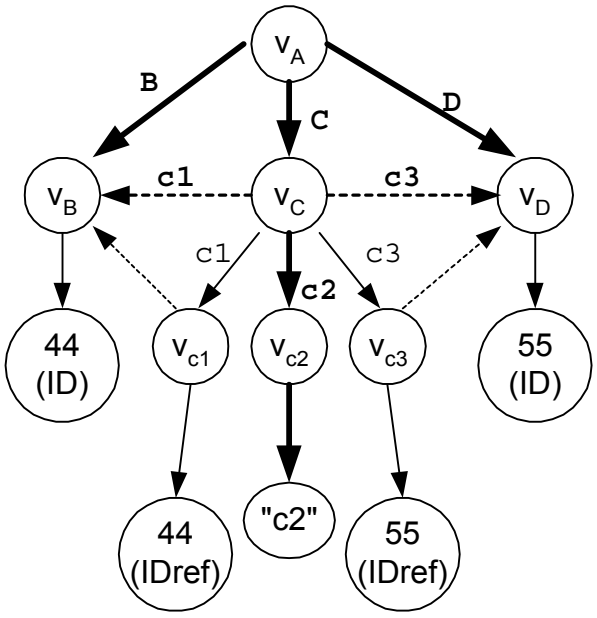


Abbildung 21: Graphrepräsentation von Quellcode 2

Die Ordnung der auf diese Weise neu definierten Referenzkanten c1 und c3 zusammen mit der Element enthaltenden Kante c2 ergibt sich aus der Ordnung der aus v_C ausgehenden Element enthaltenden Kanten c1, c2 und c3.

Die speziellen Elemente v_{c1} und v_{c3} müssen durch Tags oder Attribute eindeutig gekennzeichnet sein.

Definition (XML-Graph): Graphstrukturen wie von [BAM99] beschrieben, zusammen mit der oben genannten Erweiterung werden im Folgenden XML-Graphen genannt. Der Begriff Referenzkante bezieht sich auf die Neudefinition. Knoten und Kanten, die nur der Verwaltung der Graphstruktur dienen, können weggelassen werden.

4.2.3 Anwendung auf Strukturgraphen

Wie kann dieser Ansatz zur Speicherung von geordneten Strukturgraphen verwendet werden? (Eine gültige Speicherung von geordneten Strukturgraphen läßt sich ohne Einschränkung für die Speicherung von inhaltspartitionierten, geordneten Strukturgraphen verwenden.)

Zur Wiederholung: Strukturgraphen bestehen aus Knoten $v \in V$, die den Inhalt content(v) besitzen. Zwischen den Knoten gibt es gerichtete Kanten E, die sich zum einen in die disjunkten Mengen der Enthältkanten EC und Referenzkanten ER untergliedern, zum anderen in die disjunkten Mengen der Attributkanten EA und der restlichen Kanten E\EA. Es gibt einen ausgezeichneten Knoten, den Wurzelknoten $w \in V$, der keine eingehenden Enthältkanten besitzt.

Ist eine Kante $e \in E$ eine Attributkante (also $e \in EA$), so wird dies in dem Zielknoten von e vermerkt. Die weitere Speicherung von Attributkanten verläuft genau wie bei den restlichen Kanten E \ EA.

Im Folgenden wird beschrieben, wie sich ein Strukturgraph in einen XML-Graphen umkehrbar transformieren läßt. Dabei gilt:

- Eine Enthältkante im Strukturgraphen entspricht einer Element enthaltenden Kante im XML-Graphen. Der XML-Graph kann darüber hinaus weitere Element enthaltende Kanten besitzen, die z.B. die Struktur der Inhaltsobjekte widerspiegeln.
- Eine Referenzkante in Strukturgraphen entspricht einer Referenzkante im XML-Graphen. (Sollte die Struktur eines Inhaltsobjektes es erforderlich machen, können im XML-Graph darüber hinaus noch weitere Referenzkanten eingesetzt werden. Abgesehen davon entspricht jede Referenzkante im Strukturgraphen genau einer Referenzkante im XML-Graphen.)
- Der Wurzelknoten des Strukturgraphen entspricht dem obersten Element der XML-Hierarchie.

Eine Voraussetzung der XML-Graphen ist, daß der Graph eingeschränkt auf Element enthaltende Kanten ein Wald ist. Da die Primärstruktur des Strukturgraphen (V, EC) nach Definition ein Baum ist, wird diese Voraussetzung durch die angegebene Transformation erfüllt.

Die folgende Grafik soll die Transformation eines Knotens v demonstrieren.

Der Knoten $v \in V$ hat den Primärvater $p \in V$, sowie weitere Väter pa und pb, die durch Referenzkanten auf v verweisen. Die Kinder von v seien c1, c2 und c3 mit der Kantensortierung $(v,c1) <_v (v, c2) <_v (v, c3)$. Das mit v assoziierte Inhaltsobjekt content(v) sei durch die drei Einträge x, y und z strukturiert.

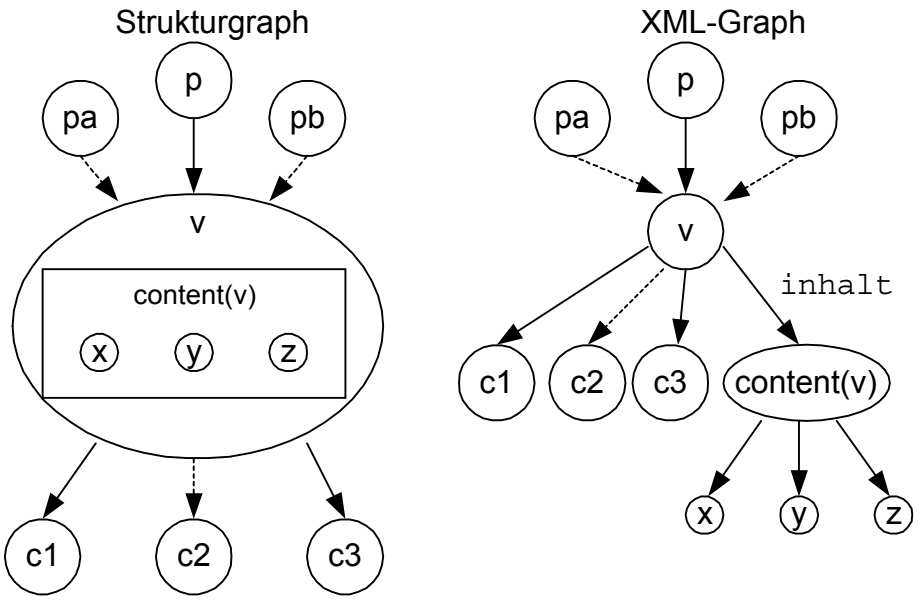


Abbildung 22: Beispiel für die Transformation zwischen Strukturgraph und XML-Graphrepräsentation

Der Übersichtlichkeit halber sei die Beschriftung der Kanten hier weggelassen. Sie stimmt mit der Beschriftung der Knoten überein. Einzige Ausnahme ist das Element mit dem Tag „inhalt“. Dieser ist bei allen Knoten v der Hinweis auf das Inhaltsobjekt content(v).

Zur Verdeutlichung sei der XML-Quellcode für den Knoten v aufgeführt. Das Kind c2, sowie die Väter pa und pb sind an anderer Stelle im Quellcode definiert.

```
<p>
        <v No::ID=“???“>
                <c1>...</c1>
                <c2 link::IDref=“...“ \>
                <c3>...</c3>
                <inhalt>
                        <x>...</x>
                        <y>...</y>
                        <z>...</z>
                </inhalt>
        </v>
        ...
</p>
```

Quellcode 3: XML-Repräsentation von Abbildung 22

Die Wahl der Tag-Namen ist bis auf „inhalt“ von der Implementierung frei wählbar. Werden verschiedene Typen von Knoten verwendet, muß aus Attributinformationen des Knotens oder dessen Tagnamen hervorgehen, zu welchem Typ der Knoten gehört, um ein Reinitialisieren des Strukturgraphen aus dem XML-Quellcode zu gewährleisten.

Da über die Struktur des Inhaltsobjekts an dieser Stelle nichts ausgesagt werden kann, muß die Anwendung der Strukturgraphen für die Umkehrbarkeit der XML-Repräsentation des Inhaltsobjekts Sorge tragen.

4.3 Nebenläufigkeit auf Strukturgraphen

Nach der Einschränkung auf inhaltspartitionierte Strukturgraphen können nur Blätter Inhalt tragen. Da gerade diese Elemente von gleichzeitigen Modifikationen betroffen sind, sollte hier eine Reservierungsheuristik verwendet werden, die speziell auf die Anforderungen der Problemdomäne eingeht.

Für manche Anwendungen kann es schwierig sein, zufriedenstellende Reservierungsheuristiken für innere Knoten zu finden, die neben dem aktuell betroffenen Knoten eine Prognose für die Modifikation weiterer Knoten zu erstellen. Es kann angenommen werden, daß die meisten Modifikationen an den Inhaltsobjekten vorgenommen werden. Daher muß insbesondere auf inhaltstragenden Knoten (bei inhaltspartitionierten Strukturgraphen sind dies ausschließlich Blätter) eine aussagekräftige Reservierungsheuristik implementiert sein. Damit die Reservierungsinvariante auf inneren Knoten erfüllt ist, muß hier zumindest die einfache Reservierungsheuristik (siehe 3.4) verwendet werden.

Eine Möglichkeit stellt daher die folgende Heuristikkombination für Strukturgraphen dar:

- $\forall\ b \in B$ wird eine speziell auf die Problem-Domäne zugeschnittene Heuristik verwendet
- auf alle $v \in V \setminus B$ wird die einfache Reservierungsheuristik angewandt.

Die Reservierungsinvariante ist damit für alle Knoten des Strukturgraphen erfüllt.

Weiterhin wird folgende Kombination aus aktualisierender Reservierung und einer modifizierten Version der verzögerten Fragmentierung empfohlen (Fragmentierungs- und Reservierungskombination für Strukturgraphen):

1) Zunächst normaler Durchlauf der Reservierungsheuristik

2) Für alle $v \in B$, für die die Konfliktfreiheitsgrarantie nicht gilt:

   2.1) Wenn v durch eine geeignete Split-Operation Kinder v_0, v_1, .. v_n enthält, so daß der Reservierungsvorgang angewendet auf v und seine Kinder die Konfliktfreiheitsgrarantie für alle Kinder aussprechen würde,

      2.1.1) wird v.split() ausgeführt

      2.1.2) und die Reservierungsheuristik darauf angewandt

Diese Kombination stellt sicher, daß nur Knoten fragmentiert werden, an denen ein Konfliktpotenzial besteht (also die Konfliktfreiheitsgarantie nicht gilt). Es wird weiterhin nicht fragmentiert, wenn durch die Fragmentierung die Konfliktgefahr nicht gebannt werden würde (Bedingung in Anweisung 2.1). Das Verfahren sorgt dennoch dafür, daß wenn für einen Knoten die Konfliktfreiheit garantiert werden kann, er auch entsprechend fragmentiert wird.

4.4 Zusammenfassung Datenstruktur für gemeinsame Datenräume

Mit den Strukturgraphen wurde eine Datenstruktur eingeführt, die sich einerseits flexibel an viele Problemstrukturen anpassen kann, andererseits bereits den Grundstein für die adaptive Konfliktgranularität – die Splitoperation – legt.

Die in Kapitel 3 beschriebenen Optimierungen der Nebenläufigkeitskontrolle wurden auf Strukturgraphen angewandt und eine spezielle Fragmentierungs- und Reservierungskombination für Strukturgraphen entwickelt.

Durch die adaptive Konfliktgranularität werden insbesondere Anforderung 10 (Vermeidung von Konflikten) und Anforderung 11 (Hinweis auf unvermeidbare Konflikte) erfüllt, aber auch Anforderung 4 (Parallele Modifikation) maßgeblich unterstützt.

Die Primärstruktur kann von einer Anwendung der Strukturgraphen leicht zur Implementierung von Anforderung 1 (Untergliederung des Dokuments in Kapitel und andere logische Einheiten) herangezogen werden.

Durch die von [BAM99] vorgestellten Mechanismen ist es möglich eine XML-Sicht auf Strukturgraphen zu entwerfen. Anforderung 3 (XML-Export) kann erfüllt werden ohne aufwendige Transformationen von XML-Applikationen implementieren zu müssen. Umgekehrt folgt daraus, daß viele Probleme, die sich mittels XML lösen lassen, auf Strukturgraphen übertragbar sind.

5 Implementierung von Strukturgraphen

In diesem Kapitel soll der abstrakte Datentyp „Strukturgraph“ näher spezifiziert werden. Die Wirkungsweise der Operationen wird in Pseudocode angegeben und ihre Konsistenz mit den Eigenschaften der Strukturgraphen wird motiviert.

Anschließend wird auf die Anforderungen von DyCE an seine Komponenten eingegangen und die Lösung einiger implementierungstechnischer Probleme erläutert.

Dazu wird zuerst auf die Frage nach der Speicherung der Graphelemente in DyCE eingegangen, wobei ein besonderer Augenmerk auf der Wahl einer Datenstruktur liegt, die keine unnötigen Konflikte verursacht.

Das zweite Problem ist die Implementierung von Attributkindern, da diese einerseits in der Graphstruktur verwaltet werden müssen, andererseits eng mit dem Inhaltsobjekt korreliert sind. Weiterhin können sie in der XML-Darstellung nicht direkt als Attribut übernommen werden, da Attribute in XML keine weitere Schachtelung zulassen.

Es wird eine Implementierung von verzögerter Fragmentierung und aktualisierendem Reservierungsverfahren besprochen. In diesem Zusammenhang soll auch eine abstrakte Implementierung von Reservierungsheuristiken vorgestellt werden.

Abschließend wird diskutiert, welcher Aufwand der Entwickler entsteht, der Strukturgraphen in DyCE für seine Anwendung einsetzen möchte.

5.1 Spezifikation von Strukturgraphen

In diesem Kapitel sollen die mathematische Definition von geordneten, inhaltspartitionierten Strukturgraphen in informatischer Weise spezifiziert werden.

Zur Spezifikation werden UML Diagramme verwendet. Die verwendete UML Notation wird in „Anhang B: Verwendete UML Notation“ beschrieben.

Die Knotenmenge V des Strukturgraphen G = (V,E) bestehe aus Instanzen v der Klasse Vertex, die Kantenmenge aus Instanzen e der Klasse Edge.

5.1.1 Objektorientierung

Inhalt der Knoten

Zugriff auf den Inhalt des Knotens v erhält man über die Methode v.getContent(). Um den Strukturgraphen nicht auf bestimmte Inhalte zu be-

schränken ist der Rückgabetyp möglichst allgemein (in Java ist dies die Klasse Object).

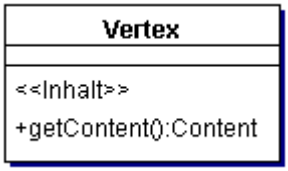


Abbildung 23: Zugriff auf den Inhalt eines Knotens (UML-Klassendiagramm)

Nach der Definition der Partitionierbarkeit muß der Inhalt von Klassen, die die Split-Operation unterstützen, partitionierbar sein.

Implementierung der Vater-Sohn-Beziehung

Die Knoten (und damit auch ihre Inhaltsobjekte) werden über Kanten in Beziehung zueinander gesetzt. Da in Strukturgraphen nur gerichtete Kanten auftreten, spricht man auch von einer Vater-Sohn-Beziehung zwischen Knoten.

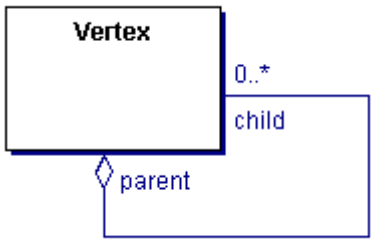


Abbildung 24:Implementierung der Vater-Sohn-Beziehung (UML Klassendiagramm)

Die Kanten E eines Strukturgraphens unterteilen sich in die Menge der Enthältkanten und der Referenzkanten. Eine Enthältkante e wird durch eine Instanz der Klasse ContainEdge, eine Referenzkante r durch eine Instanz der Klasse ReferenceEdge repräsentiert. ContainEdge und ReferenceEdge sind Ableitungen der gemeinsamen Oberklasse Edge.

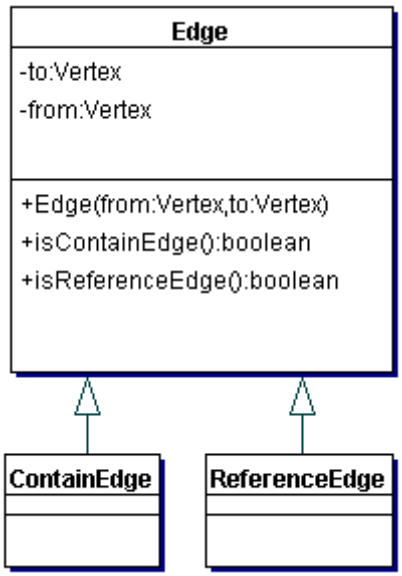


Abbildung 25: Klassenhierarchie von Kanten (UML Klassendiagramm)

Die Kantenobjekte verweisen direkt auf ihre inzidenten Knoten.

Knoteninstanzen von v greifen auf die inzidenten Kanten über die Klassen EdgeVector bzw. EdgeSet zu. EdgeVector beinhaltet alle aus v ausgehenden Kanten, EdgeSet die nach v eingehenden Kanten.

Die Benennung soll die enge Verwandtschaft mit den Javaklassen java.util.Vector und java.util.Set andeuten. Vector implementiert ein in der Länge variierendes Array von verschiedenen Objekttypen, Set eine unsortierte Menge.

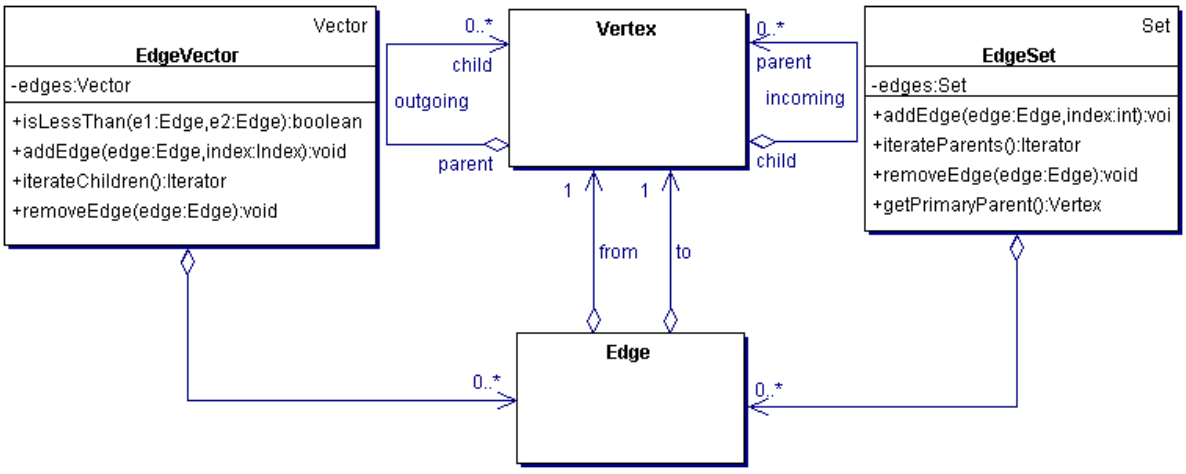


Abbildung 26: Verwaltung von Knoten und Kanten im Strukturgraph (UML Klassendiagramm)(UML Klassendiagramm)

Die Klasse Vertex implementiert einige Methoden um Informationen über die Graphstruktur abzurufen, bzw. den Strukturgraphen zu modifizieren.

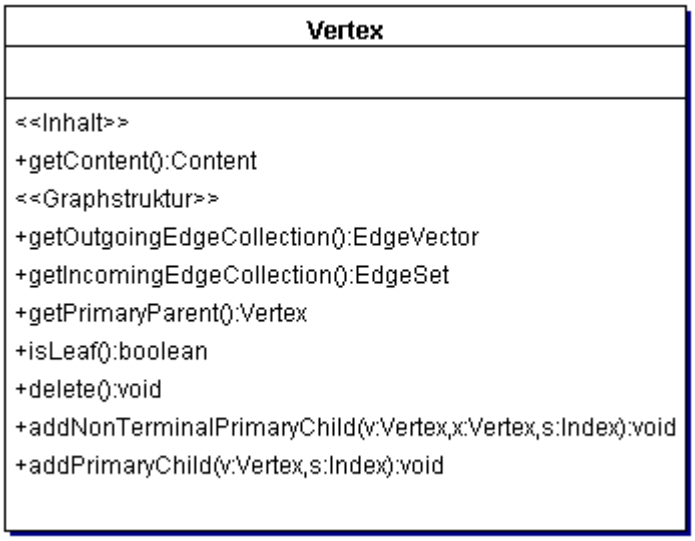


Abbildung 27: Zugriff und Modifikation der Graphstruktur ausgehend von einem Knoten (UML-Klassendiagramm)

Ein Knoten enthält die mit ihm assoziierten EdgeVector und EdgeSet Objekte über die Methoden v.getOutgoingEdgeVector() und v.getIncomingEdgeSet().

Über die eingehenden Kanten werden Väterknoten und über die ausgehenden Kanten die Kindknoten referenziert.

v.getPrimaryParent() verwendet diesen Mechanismus, um den Vater in der Primärstruktur von v zu liefern. Da es nur eine eingehende Enthältkante gibt, ist dieser eindeutig bestimmt.

v.isLeaf() zeigt an, ob v Kinder besitzt.

Die Primärstruktur des Strukturgraphen kann erweitert werden, indem neue Kinder an bestehende Knoten durch Enthältkanten angehängt werden. Dies leistet die Methode v.addPrimaryChild(u:Vertex, s:Index) wobei u das neue Kind ist. Der Index s sortiert die neue Enthältkante in die Ordnung der aus v ausgehenden Kanten ein.

Soll ein neues Kind eingefügt werden, das den Blattstatus nicht erlaubt, so muß dieses zusammen mit einem weiteren Knoten eingefügt werden. Dazu wird die Methode v.addNonTerminalPrimaryChild(u:Vertex, x:Vertex, s:Index). u und s haben die gleiche Bedeutung wie bei der Methode addPrimaryChild. x wird zusätzlich über eine Enthältkante als Kind von u eingetragen.

Knoten v können aus der Graphstruktur mit der Methode v.delete() gelöscht werden. Diese Methode stellt sicher, daß alle inzidenten Kanten ebenfalls entfernt werden.

Kantenordnung

Die Kantenordnung wird durch die Methode isLessThan von EdgeVector modelliert.

Sei e=(u,v) $\in$ E eine Instanz von Edge. Für e gelte e.from = u und e.to = v mit u und v Vertexinstanzen.

Für eine weitere Kanteninstanz e2 = (u,v‘) ist das Ergebnis der Methode e.isLessThan(e2) := $(u,v) <_u (u,v')$. isLessThan angewendet auf Kanten die aus verschiedenen Knoten ausgehen, liefert ein ungültiges Ergebnis.

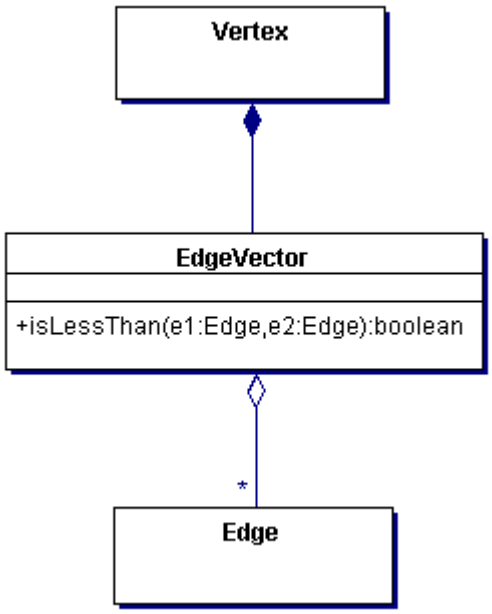


Abbildung 28: Implementierung der Kantensortierung (UML-Klassendiagramm)

Freiheitsgrade

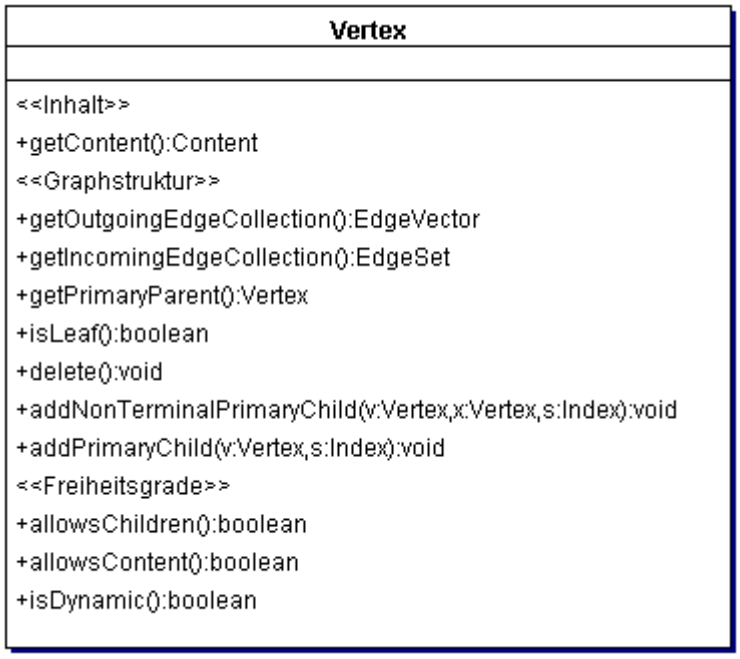


Abbildung 29: Zugriff auf Informationen der Freiheitsgrade des Knotens (UML-Klassendiagramm)

Über die Methoden v.allowsChildren() und v.allowsContent() kann ein Knoten bestimmen, ob er potenziell Kinder bzw. Inhalt erlaubt oder grundsätzlich ausschließt.

Die Knoten v, die die Anfrage v.isDynamic() mit wahr beantworten, tragen partitionierbaren Inhalt und unterstützen damit potenziell die Split-Operation. Gibt v.isDynamic() falsch zurück, so unterstützt v niemals die Split-Operation.

Da sich die Rückgabewerte dieser Methoden zur Laufzeit nicht ändern, ist keine Konsistenzprüfung nötig.

Split- und Join-Operation

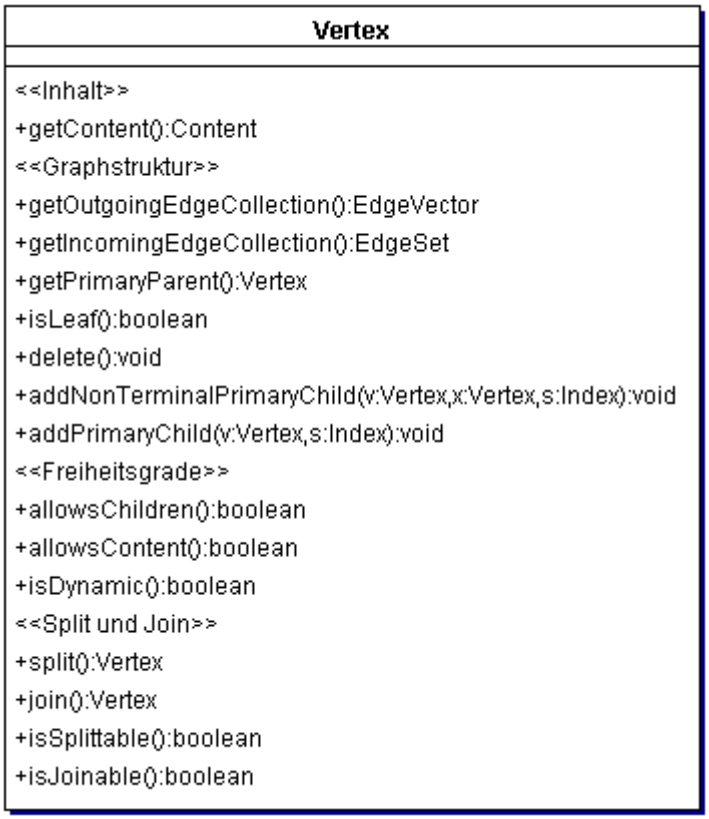


Abbildung 30: Split- und Join-Operation (UML-Klassendiagramm)

Durch Aufruf der Methode v.split() führt ein Knoten v auf sich selbst die Split-Operation aus. D.h. er partitioniert seinen Inhalt, kreiert entsprechende Kindknoten auf die er seinen direkten Inhalt verteilt. v selbst enthält anschließend keinen unmittelbaren Inhalt mehr und wird zum Nichtterminalknoten.

Durch Aufruf der Methode v.join() führt v die Join-Operation auf seinen Nachfahren aus. Darin werden die Inhalte der Kinder in der Reihenfolge der Kantensortierung zum Inhalt von v konkateniert und die Kindknoten anschließend gelöscht. v ist nach der Operation ein Blatt.

Hat v Kinder, die keine Blätter sind, so ist die Join-Operation nicht direkt möglich.

Ist es nicht möglich die Split- bzw. Join-Operation auszuführen, führt der Aufruf von v.split() bzw. v.join() zu keiner Veränderung.

Die Methoden v.isSplittable() und v.isJoinable() geben an, ob ein anschließender Aufruf der Methoden v.split() bzw. v.join() erfolgreich ausgeführt wird (und damit zu einer Veränderung führt).

Gleichheit

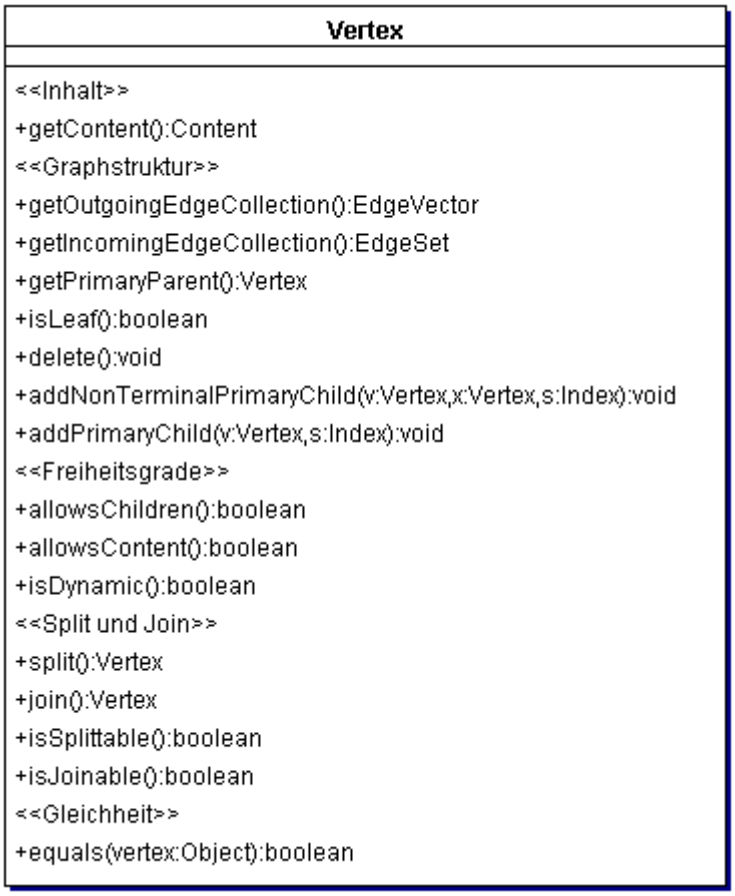


Abbildung 31: Modellierung von Gleichheit auf Knoten (UML-Klassendiagramm)

Gleichheit auf Knoten wird über gleiches äußeres Verhalten definiert. Liefern für zwei Knoten u und v alle Methoden das gleiche Ergebnis, so heißen u und v gleich. In Java wird dies durch die Methode equals modelliert.

Zwei Kanten e = (u, v) und e‘ = (u‘, v‘) heißen gleich $\Leftrightarrow$ u = u‘ und v = v‘. Sollte es in einer Strukturgraphimplementation nötig sein, mehrere Kanten mit gleichen Start- und Endknoten zu verwalten, muß ein zusätzliches Identifizierungsmerkmal verwaltet werden.

5.1.2 Bedingungen und Eigenschaften der Methoden

Seien u, v, w Vertexinstanzen und e1,e2 Edgeinstanzen. Es gilt:

1. v.getContent() == content(v)
2. v.getOutgoingEdgeVector.isEmpty() $\Leftrightarrow$ v.isLeaf == true $\Leftrightarrow$ v $\in$ B

3. v.allowsChildren == false $\Rightarrow$ Zu jedem Zeitpunkt[17] t gilt: v $\in$ B und v.getContent() != null.
4. v.allowsContent == false $\Rightarrow$ Zu jedem Zeitpunkt[18] t gilt: v.getContent() == null und mit der Einschränkung auf inhaltspartitionierte Strukturgraphen $\Rightarrow$ v $\notin$ B
5. aus 3 und 4 $\Rightarrow$ v.allowsContent() oder v.allowsChildren() muß true sein.
6. v.isSplittable() $\Rightarrow$ v.isDynamic()
7. v.isJoinable() $\Rightarrow$ v.isDynamic()
8. v.isDynamic() $\Rightarrow$ v.allowsChildren() und v.allowsContent()
9. v.isDynamic() $\Rightarrow$ v.getContent() ist partitionierbar
10. v.isSplittable() $\Rightarrow$ v.split() wird erfolgreich sein
11. unmittelbar nach erfolgreichem Ausführen von v.split() gilt: !v.isLeaf(), !v.isSplittable(), v.isJoinable()
12. v.isJoinable() $\Rightarrow$ v.join() wird erfolgreich sein
13. unmittelbar nach dem erfolgreichem Ausführen von v.join() gilt: v.isLeaf(), v.isSplittable(), !v.isJoinable()
14. Für v Wurzelknoten gilt v.getPrimaryParent() == null. Für alle anderen v $\in$ V gilt: v.getPrimaryParent() != null.
15. Für ein v $\in$ V sei $PrimaryParents_v := \{ p \mid (p, v) \in EC \}$. $\forall$ v $\in$ V gilt $|PrimaryParents_v| \leq 1$, denn der Graph (V, EC) bildet einen Baum. v.getPrimaryParent() $\in$ $PrimaryParents_v$ ist damit eindeutig bestimmt.
16. Sei e1 mit e1.to = u und e1.from = v. Sei e2 mit e2.to = w und e2.from = v. Es gilt: v.getOutgoingEdgeVector().isLessThan(e1,e2) $\Leftrightarrow$!v.getOutgoingEdgeVector().isLessThan(e2, e1)
17. u == v $\Leftrightarrow$ u.equals(v) $\Leftrightarrow$ v.equals(u)
18. v.isSplittable() $\Rightarrow$ v.isLeaf()
19. v.isJoinable() $\Rightarrow$!v.isLeaf() und $\forall$ Kinder u von v gilt: u.isJoinable() und u.isLeaf()
20. Durch die Methoden split() und join() gehen keine Inhaltsdaten verloren.

5.1.3 Strukturveränderungen

Neben Modifikationen von Inhaltsobjekten führen Benutzeraktionen auch zu Strukturveränderungen des Graphen.

---

[17] ausgenommen sind temporär inkonsistente Zustände.

[18] ausgenommen sind temporär inkonsistente Zustände.

Strukturveränderungen unterscheiden sich in Operationen, die grundlegendes Verhalten implementieren, und konsistente Operationen. Letztere garantieren, daß der Strukturgraph stets die in der Definition von geordneten, inhaltspartitionierten Strukturgraphen geforderten Eigenschaften erfüllt.

Grundlegende Operationen

Durch Operationen auf dem Strukturgraphen kann sich dessen Struktur ändern. Die grundlegendsten Operationen auf dem Strukturgraphen G sind:

- G.addVertex(u): Hinzufügen eines Knotens u in zu G
- G.delVertex(u): Löschen eines Knotens u aus G
- G.addEdge(e, s): Hinzufügen von einer Kante e=(u,v) zu G, wobei die bestehende Kantenordnung $<_u$ durch s(e, $<_u$) erweitert wird.
- G.delEdge(e): Löschen einer Kante e aus G
- G.delVertexWithEdges(u): Löschen des Knotens u und alle in u eingehenden und aus u ausgehenden Kanten.

Basisoperationen auf einem Knoten v sind das Setzen des Inhalts und der Rückgabewerte für isSplittable() und isJoinable().

Definition Konsistenz

Für einen Strukturgraphen G = (V, E) gilt stets

- Es gibt genau einen Wurzelknoten. (Eigenschaft des Strukturgraphen)
- Für alle anderen Knoten gilt, sie haben genau einen Vater gemäß der Primärstruktur. (Eigenschaft der Primärstruktur)
- Jedes Blatt hat Inhalt, jeder Nichtterminalknoten besitzt keinen direkten Inhalt. (Voraussetzung für Inhaltspartitioniertheit)
- $\forall$ Knoten v ist die Kantenordnung $<_v$ stets vollständig definiert. (Voraussetzung für geordnete Strukturgraphen)
- $\forall\ v \in V$: v.isSplittable() $\Leftrightarrow$ Split-Operation auf v ist erfolgreich und führt damit zu einer Veränderung. (Voraussetzung der Split-Operation)
- $\forall\ v \in V$: v.isJoinable() $\Leftrightarrow$ Join-Operation auf v ist erfolgreich und führt damit zu einer Veränderung. (Voraussetzung der Join-Operation)
- Für die Methoden split() und join() ist zusätzlich die Gültigkeit ihrer Invertierbarkeit zu zeigen.

Definition (Operation): Sei H die Menge aller Strukturgraphen. Die Funktion o: H -> H heißt Operation auf einem Strukturgraphen G $\in$ H die G in G‘ $\in$ H überführt.

Definition (konsistente Operation): Eine Operation o heißt konsistent $\Leftrightarrow$ wenn o ausgeführt auf einem beliebigen Strukturgraphen G wiederum zu einem Strukturgraphen G‘:=o(G) führt.

Der Erfolg der Split-Operation hängt nicht nur vom Knoten, auf den sie angewandt wird, selbst ab, sondern auch von Knoten in der Umgebung. Um v.isSplittable() und v.isJoinable() einfacher verwalten zu können, werden die lokalen Eigenschaften v.localIsSplittable() und v.localIsJoinable() eingeführt.

Definition (localIsJoinable): Es gilt: v.isJoinable() $\Leftrightarrow$ v $\notin$ B und $\forall$ Kinder u von v gilt u.localIsJoinable() ist wahr, u besitzt keine eingehenden Referenzkanten und u.isLeaf() ist wahr.

Definition (localIsSplittable): Es gilt v.isSplittable() $\Leftrightarrow$ v.localIsSplittable() und v.isLeaf() ist wahr.

In beiden Fällen gelte automatisch: v.isSplittable() und v.isJoinable() sind false, wenn !v.isDynamic().

Bei dynamischen Knoten ist v.localIsSplittable und v.localIsJoinable normalerweise auf wahr gesetzt. Falls v jedoch ein Kind hat, das nicht dynamisch ist, oder der Knoten aus anderen Gründen nicht verschmolzen werden darf, wird v.localIsJoinable auf false gesetzt. Ist die feinste Granularität erreicht, so wird v.isSplittable auf false gesetzt.

Operation v.addPrimaryChild(u, s)

Diese Operation fügt den Knoten u als ein Kind des bestehenden Knotens v in den Graph ein, so daß die Primärstruktur erhalten bleibt.

Vorbedingungen:

- v $\in$ V ist ein in G bestehender Knoten. v besitzt keinen Inhalt.
- u ist eine Knoteninstanz mit u $\notin$ V.
- u.getContent() != null.
- s ist ein Index, der die Kante (v,u) in die bestehende linearen Kantenordnung $<_v$ einsortiert

dann führt die Operation v.addPrimaryChild(u, i) folgende Schritte aus:

1) G.addVertex(u).

2) Kreiere eine neue Enthältkanteninstanz e = (v,u) mit e $\in$ EC.

3) G.addEdge(e, s).

4) hat v Inhalt, so setze Content(v) := null.

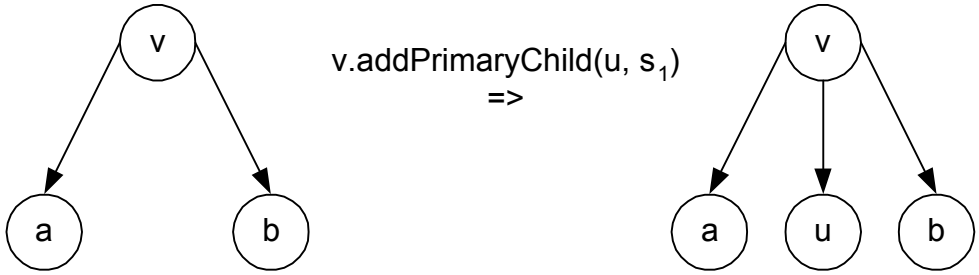


Abbildung 32: Hinzufügen eines Kindes u in der Primärstruktur bei vorhandenen Geschwistern a und b.

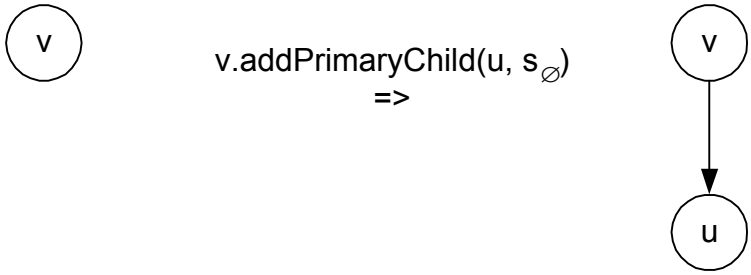


Abbildung 33: Hinzufügen eines Kindes u in der Primärstruktur ohne vorhandene Geschwister.

Behauptung: Falls u Inhalt besitzt, und die Vorbedingungen erfüllt sind, ist v.addPrimaryChild(u, s) konsistent

Plausibilität der Konsistenz:

Kann G‘ die Strukturgrapheigenschaften verlieren?

Das neue Blatt u hat Inhalt. Der Knoten v, der vor der Operation möglicherweise ein Blatt war, hat in G‘ keinen Inhalt. Es sind keine neuen Wurzelknoten hinzugekommen, da u zusammen mit einer eingehenden Kante eingefügt wurde.

Weitere Modifikationen wurden nicht vorgenommen.

Operation v.addNonTerminalPrimaryChild(u,x,s)

Instanzen u der Klasse StructuringVertex liefern v.allowsContent() == false. Daraus folgt mit der Einschränkung auf inhaltspartitionierte Strukturgraphen G, daß stets u $\in$ V\B gelten muß, damit G Strukturgraph ist. v.addPrimaryChild(u,s) auf u $\in$ StructuringVertex ist inkonsistent.

Um dieses Defizit auszugleichen fügt die Operation v.addNonTerminalPrimaryChild(u, x, s) u als Kind in v ein und hängt gleichzeitig x als ein Kind in u an.

Vorbedingungen:

- u,x sind instanziiert, aber u,x $\notin$ V
- x besitzt Inhalt
- v.allowsChildren() == true

Auszuführende Schritte

1) v.addPrimaryChild(u, s)

2) u.addPrimaryChild(x, $s_{\varnothing}$)

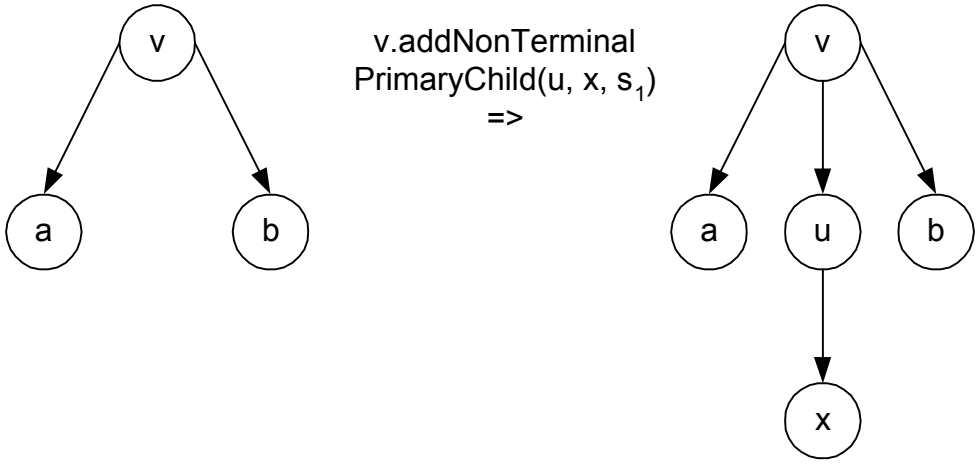


Abbildung 34: Hinzufügen eines Kindes u, welches den Blattzustand nicht erlaubt bei vorhanden Geschwistern a und b.

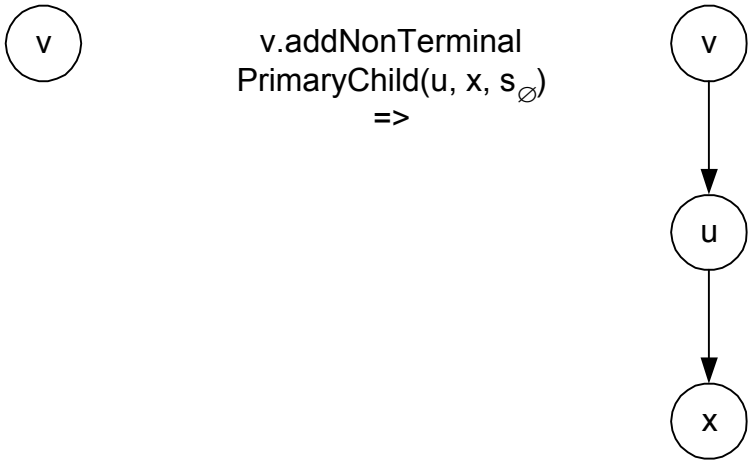


Abbildung 35:Hinzufügen eines Kindes u, welches den Blattzustand nicht erlaubt ohne vorhandene Geschwister.

Behauptung: v.addNonTerminalPrimaryChild(u, x, s) ist konsistent, wenn die Vorbedingungen erfüllt sind.

Plausibilität der Konsistenz:

Ist der neue Graph G‘ ein Strukturgraph?

Das neue Knoten u erlaubt zwar keinen Inhalt, hat aber ein Kind. Das neue Blatt x hat Inhalt nach Voraussetzung. Der Knoten v, der vor der Operation möglicherweise ein Blatt war, hat in G‘ keinen Inhalt. Es sind keine neuen Wurzelknoten hinzugekommen, da u und x zusammen mit einer eingehenden Kante eingefügt wurde.

Weitere Modifikationen wurden nicht vorgenommen.

Operation v.delete()

Durch Benutzerinteraktionen können Blätter oder ganze Teilbäume obsolete werden. Durch die Operation delete wird ein Knoten v konsistent gelöscht. Damit sichergestellt ist, daß dabei keine neuen Wurzelknoten entstehen, darf v keine ausgehenden Enthältkanten besitzen.

Diese Operation kann leicht auf das Löschen ganzer Teilbäume der Primärstruktur erweitert werden.

Sei u der Vater (nach der Primärstruktur) von v.

Vorbedingungen

- $v \in V$
- v ist kein Wurzelknoten (also u existiert)
- u.allowsContent() == true oder die Anzahl der Kinder von u > 1
- v.getOutgoingEdges() $\cap$ EC = $\varnothing$

Sind die Vorbedingungen erfüllt, werden folgende Schritte ausgeführt.

1) G.delVertexWithEdges(v)
2) Falls u nun Blatt in der Primärstruktur ist, wird u der leere Inhalt zugewiesen.

Abbildung 36: Löschen eines Knotens v, bei weiteren vorhandenen Geschwistern a und b.

Abbildung 37: Löschen des Knotens v, der das letzte vorhandene Kind von u war.

Behauptung: v.delete() ist konsistent, wenn die Vorbedingungen erfüllt sind.

Plausibilität der Konsistenz:

Ist der durch neu entstandene Graph G‘ Strukturgraph?

Es können keine neuen Wurzelknoten entstanden sein, da delete() nur auf Blättern der Primärstruktur ausgeführt werden darf. Der Wurzelknoten kann nach Voraussetzung nicht gelöscht werden.

Falls u durch die Operation zum Blatt wird, wird ihm der leere Inhalt zugewiesen.

Operation v.split()

Zur adaptiven Anpassung der Konfliktgranularität wird die Split-Operation verwandt. Durch den Aufruf v.split() wird sie auf den Knoten v angewandt.

Vorbedingungen:

- $v \in V$
- v.isSplittable()

v.split() führt nur zu einer Veränderung, wenn v.isSplittable() wahr ist (dies setzt vorraus, daß v.isLeaf() und v.allowsChildren() wahr sind und das v.getContent() partitionierbar ist).

Sind diese Vorbedingungen erfüllt, geschieht folgendes:

1) inhalt = content(v)

2) $Tsplit_n$(inhalt) führt zu ($inhalt_0$, $inhalt_1$, ... $inhalt_{n-1}$)

3) content(v) := null

4) for i=0 to n-1 do

    4.1) Instanziiere einen neuen DynamicVertex u_i

    4.2) Setze u_i.content = $inhalt_i$

    4.3) Füge u_i in den Graph ein mit v.addPrimaryChild(u_i, s_{end})

4.4) Ist $inhalt_i$ die feinste zugelassene Partitionierungsgranularität, so setze u_i.localIsSplittable := false andernfalls u_i.localIsSplittable := true

4.5) u_i.localIsJoinable := true

5) ende

Sind die Vorbedingungen nicht erfüllt, so geschieht nichts.

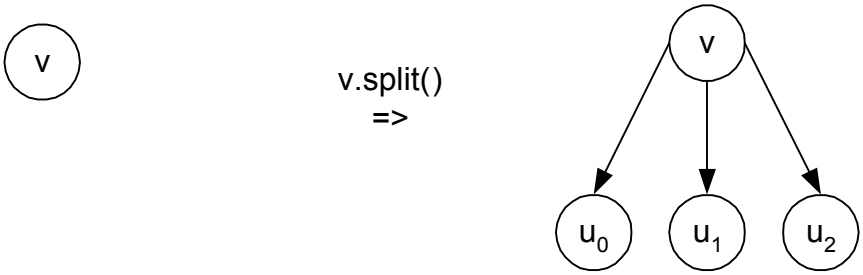


Abbildung 38: Fragmentieren des Knotens v in die Subknoten u_0, u_1 und u_2.

Behauptung: Die Splitoperation v.split(), ausgeführt auf einem Knoten $v \in V$, ist konsistent.

Plausibilität der Konsistenz:

Ist der neue Graph G‘ Strukturgraph?

Es entstehen keine neuen Wurzelknoten, da alle neuen Knoten zusammen mit einer eingehenden Kante eingefügt werden. Knoten v, der vor der Operation ein Blatt war, enthält nun keinen (unmittelbaren) Inhalt. Für alle Kinder u_i gilt: Content(u_i) != null.

Der mittelbare Inhalt von v ändert sich nicht, wenn die Funktion $Tsplit_n$, die auf dem Inhaltsobjekt operiert, die Eigenschaften aus 4.1.3 erfüllt.

In G‘ darf v.split() nicht erfolgreich sein. Dies ist erfüllt, denn v.isSplittable() ist nach der Definition von localIsSplittable false, weil v ist kein Blatt mehr ist und damit die Vorbedingung für v.split() nicht erfüllt ist.

Für ein Kind u von v darf in G‘ u.split() genau dann erfolgreich sein, wenn die feinste Granularität für u noch nicht erreicht ist. Dies gilt, da u ein Blatt ist, und u.localIsSplittable() $\Leftrightarrow$ die feinste Granularität ist nicht erreicht (Anweisung 4.4)

In G‘ muß v.join() erfolgreich sein, wenn v.localIsSplittable() bereits in G true war. Dies gilt genau dann, wenn v.isJoinable() true ist. $v \notin B$ und alle Kinder von v sind Blätter und deren localIsJoinable Eigenschaft ist true. Nach Definition von localIsJoinable ist damit v.isSplittable() true.

Für alle Kinder u von v ist u.isJoinable() zunächst false, da sie alle Blätter sind.

Operation v.join()

Ist die Konfliktgranularität im Teilbaum unter dem Knoten v höher als benötigt, so erlaubt die Methode v.join das Wiederverschmelzen eines einstufigen Teilbaums. Die Erweitung auf allgemeine Teilbäume kann damit leicht implementiert werden.

Vorbedingungen:

- $v \in V$
- v.isJoinable()

Sind die Vorbedingungen erfüllt, werden folgende Operationen ausgeführt.

1) Sei n die Anzahl der Kinder von v.

2) for i = 0 to (n-1) do

   2.1) Sei u_i der Knoten zu dem die i-te aus v ausgehende Kante $(v,u_i) \in$ Ev führt

   2.2) $inhalt_i$:= content(u_i)

   2.3) G.delVertexWithEdges(u_i)

3) Sei inhalt := $Tjoin_n(inhalt_0, inhalt_1, .. inhalt_{n-1})$

4) Setze content(v) := inhalt

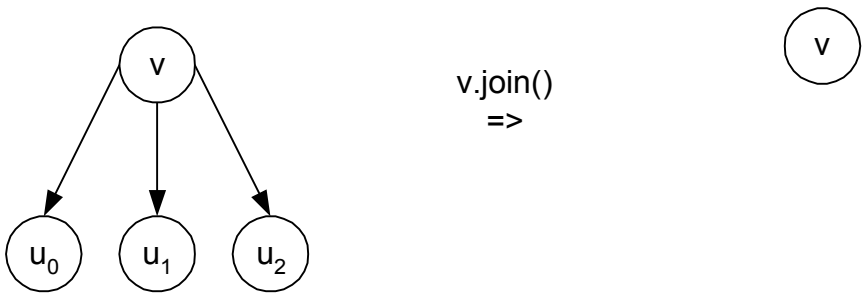


Abbildung 39: Verschmelzen der Knoten u_0, u_1 und u_2 zum Vaterknoten v.

Behauptung: Die Splitoperation v.join() ausgeführt auf einem Knoten $v \in V$ ist konsistent.

Plausibilität der Konsistenz:

Ist der neue Graph G‘ Strukturgraph?

Aus v.isJoinable() folgt $v \notin B$ und alle Kinder u von v sind Blätter. Daraus folgt, daß keine neuen Wurzelknoten entstehen, da nur Blätter entfernt werden. Für das neue Blatt v gilt: Content(v) != null.

Der mittelbare Inhalt von v ändert sich nicht, wenn die Funktion $Tjoin_n$, die auf dem Inhaltsobjekt operiert, die Eigenschaften aus 4.1.3 erfüllt.

Wenn v.localIsSplittable gilt, muß v.split() auf dem resultierenden Strukturgraph erfolgreich sein. Das ist sie, wenn v.isSplittable() true ist. Dies gilt nach Definition von localIsSplittable, da v ein Blatt ist.

Da $v \in B$ folgt mit der Definition von localIsJoinable daß die v.join() auf dem resultierenden Strukturgraph nicht erfolgreich ist.

Invertierbarkeit von split und join

Neben der Konsistenz von Split- und Join-Operation ist das Wissen über deren Invertierbarkeit essentiell.

Sei G0=(V, E) ein Strukturgraph mit $v \in B$. v.split() überführe G0 nach G1. Ein anschließendes v.join() überführe G1 nach G2.

Behauptung G0 = G2

Plausibilität der Konsistenz:

Fall 1: In G0 ist v.isSplittable()

In G1 ist $v \notin B$. $\forall$ Kinder u von v gilt: $u \in B$ und u.localIsJoinable() $\Rightarrow$ v.isJoinable() $\Rightarrow$ die v.join() führt in G1 zu einer Veränderung.

In G2 ist $v \in B$, die Kinder u sind gelöscht.

Da an v.localIsJoinable() und v.localIsSplittable() keine Änderung vorgenommen wird $\Rightarrow$ v.isSplittable

Da bereits bewiesen wurde, daß durch Split- und Join-Operation sich der (mittelbare) Inhalt von v nicht ändert, ist auch der content(v) in G0 der gleiche wie in G2.

Fall 2: In G0 ist nicht v.isSplittable()

Damit führt v.split() zu keiner Veränderung $\Rightarrow$ G0 = G1.

Da $v \in B$ nach Voraussetzung folgt, v.isJoinable() ist false. $\Rightarrow$ v.join() führt zu keiner Veränderung $\Rightarrow$ G1 = G0

Anmerkung: $v \in B$ muß gefordert werden, da sonst v.split() zu keiner Veränderung führt und v.join() evtl. eine vorausgegangene Split-Operation invertiert.

Anmerkung: Die umgekehrte Reihenfolge, also v.join() mit anschließendem v.split(), führt nicht unbedingt zum gleichen Ergebnis, da zum einen $Tsplit_n$ nicht deterministisch sein muß, zum anderen vorangegangene Modifikationen an den Kindern von v auch bei deterministischem Tsplitn zu verschiedenen Ergebnissen führen können.

Beispiel: $Tsplit_n$ teile die Eingabe s in n gleichlange Stücke.

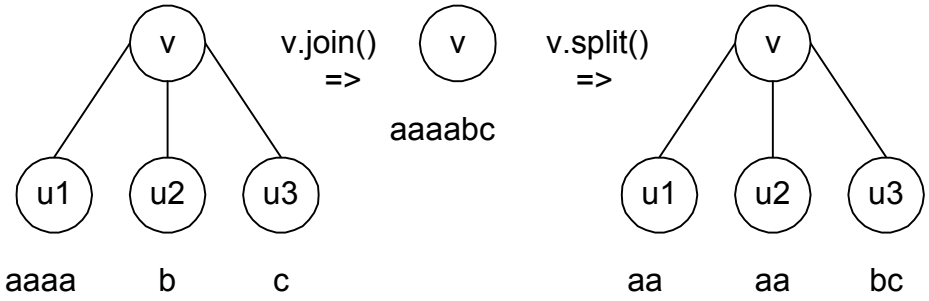


Abbildung 40: Demonstration, daß Split- und Join-Operation nur bedingt invertierbar sind.

G.addReferenceEdge(Vertex v, Vertex u, Index s)

Bevor eine Referenzkante in den Strukturgraphen eingefügt werden kann, müssen die betroffenen Knoten im Strukturgraph vorhanden sein.

Vorbedingungen:

- $v, u \in V$
- $(v, u) \notin E$

Sind die Vorbedingungen erfüllt, werden folgende Operationen ausgeführt.

1) Kreiere eine neue Referenzkanteninstanz e = (v,u) mit $e \in ER$

2) G.addEdge(e, s)

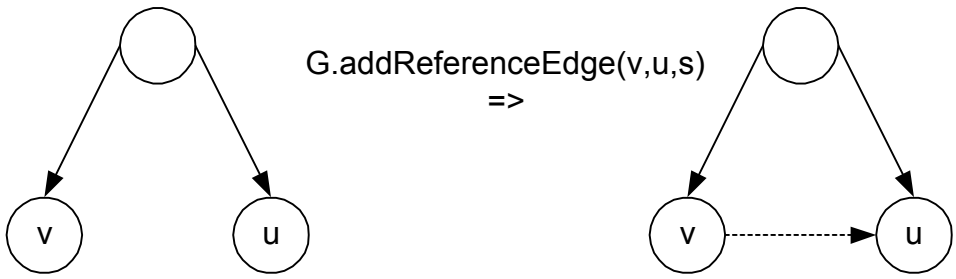


Abbildung 41: Hinzufügen einer Referenzkante zwischen beliebigen Knoten v und u.

Behauptung: Die Operation G.addReferenceEdge(v, u, s) ist konsistent.

Plausibilität der Konsistenz:

Ist der neue Graph G‘ Strukturgraph?

Ja, denn ein Strukturgraph kann beliebig viele Referenzkanten besitzen.

G.delReferenceEdge(Vertex v, Vertex u)

Vorbedingungen:

- $v, u \in V$
- $(v, u) \in ER$

Sind die Vorbedingungen erfüllt, wird folgende Operation ausgeführt.

2) G.delEdge((v, u))

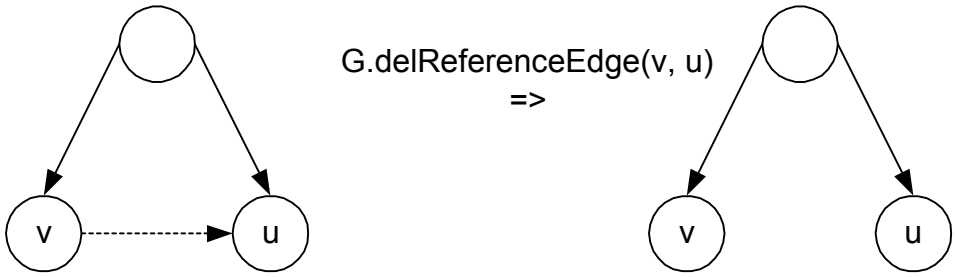


Abbildung 42: Entfernen einer Referenzkante zwischen v und u.

Behauptung: Die Operation G.delReferenceEdge(v, u) ist konsistent.

Plausibilität der Konsistenz:

Ist der neue Graph G‘ Strukturgraph?

Ja, denn ein Strukturgraph kann eine beliebige Anzahl an Referenzkanten besitzen.

Initialisation von Strukturgraphen

Üblicherweise besteht am Anfang eines Arbeitsprozesses keine Datenbasis. Wie wird ein Strukturgraph initialisiert, so daß die konsistenten Operationen in erwarteter Weise darauf arbeiten können?

Der Strukturgraph-Konstruktor erstellt einen Strukturgraphen.

1) Initialisiere die Datenstruktur für Strukturgraphen G

2) Instanziiere den Wurzelknoten w

3) G.addVertex(w)

4) Da RootVertex keinen kindlosen Zustand erlaubt, Instanziiere einen Knoten v, der diesen Zustand erlaubt.

5) Setze content(v) auf den leeren Inhalt.

6) w.addPrimaryChild(v, $s_{\varnothing}$)

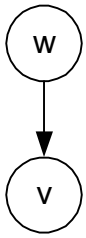


Abbildung 43: Initialisierung eines Strukturgraphen

Der auf diese Weise erstellte Graph G ist Strukturgraph, denn

- er besitzt genau einen Wurzelknoten w.
- Der andere Knoten v besitzt w als Vater der Primärstruktur.
- Die Kantenordnung $<_w$ ist vollständig definiert und die Endpunkte der Kante e = (w, v) liegen beide innerhalb des Graphen.
- w ist Nichtterminalknoten und besitzt keinen Inhalt
- v ist Blatt und besitzt leeren Inhalt.

5.1.4 Knotentypen

Die Knoten $v \in V$ eines Strukturgraphens lassen sich nach verschiedenen Typen klassifizieren. Es gibt Knoten, die lediglich als Container für andere Knoten dienen und nach der Einschränkung auf inhaltspartitionierte Strukturgraphen niemals Blätter werden können. Andere Knoten speichern einen bestimmten Typ von Inhaltsobjekten und sind somit in einem Strukturgraphen immer als Blätter enthalten. Die dritte Art speichert partitionierbaren Inhalt. Durch sie wird die adaptive Konfliktgranularität implementiert.

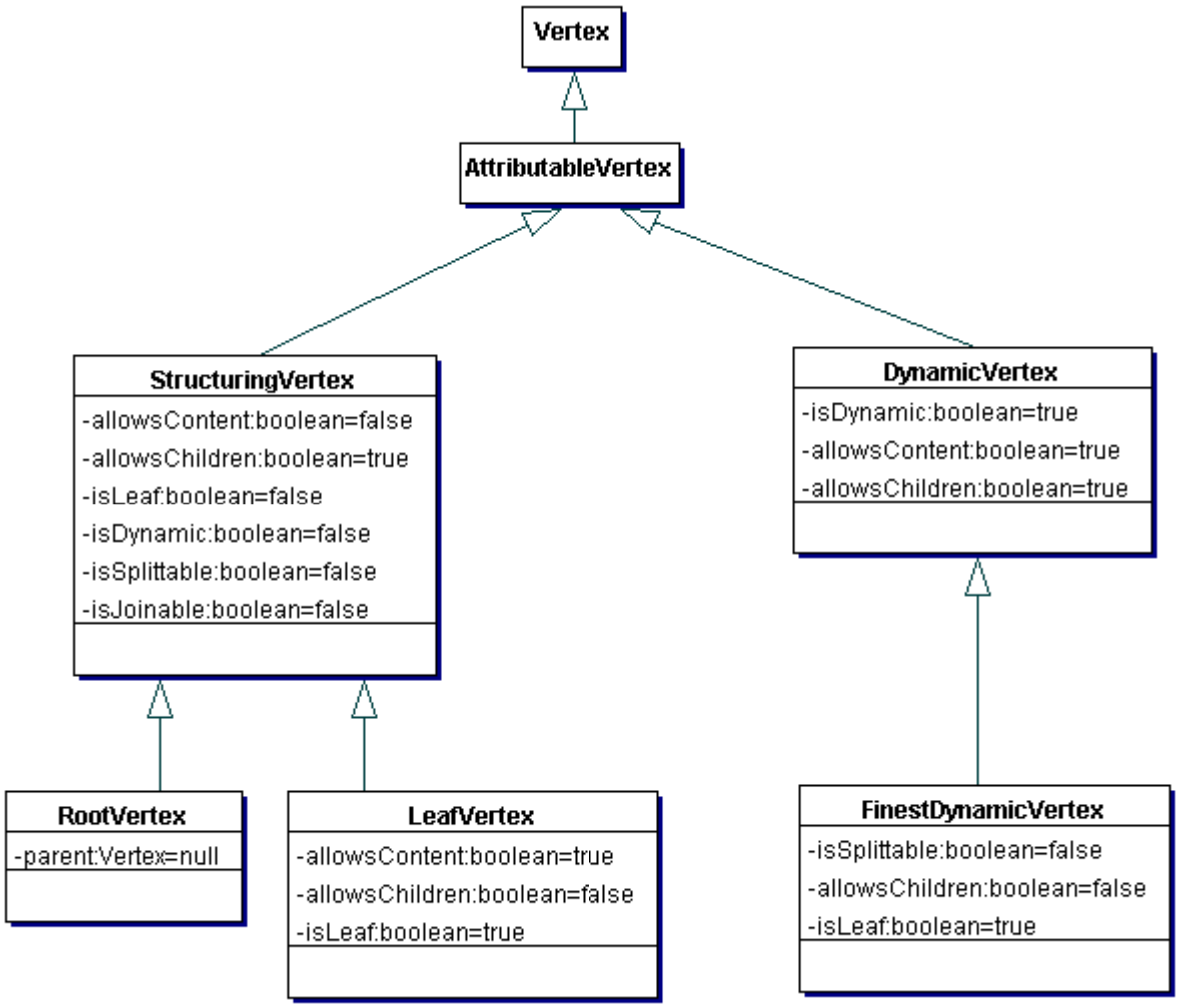


Abbildung 44: Hierarchie der Knotentypen (UML-Klassendiagramm)

Der Typ StructuringVertex und seine Ableitungen eignen sich zur Strukturierung der Datenstruktur. Sie können nur als Nichtterminalknoten auftreten und besitzen demensprechend niemals unmittelbar Inhalt. Daraus folgt, daß sie die Split-Operation nicht unterstützen können.

Ein Spezialfall ist der Typ RootVertex. Dies ist der einzige Knotentyp, der keinen Vater in der Primärstruktur besitzt und damit den Wurzelknoten modelliert.

Durch die Klasse LeafVertex kann man Knoten modellieren, die nicht partitionierbaren Inhalt besitzen und aus diesem Grund keine Kinder haben können.

Nur Ableitungen der Klasse DynamicVertex unterstützen potenziell die Split- und Join-Operationen und erlauben damit sowohl Kinder als auch direkten Inhalt zu haben.

Knoten des Typs FinestDynamicVertex modellieren die feinste Fragmentierungsgranularität. Daher erlauben sie keine weiteren Splits, obwohl ihr Inhalt partitionierbar ist.

Die im UML-Diagramm aufgeführten Attribute geben die Rückgabewerte der entsprechenden Methoden in den verschiedenen Vertextypen an. Abgeleitete

Klassen sollten diese nicht überladen, da sie die typischen Eigenschaften der Knotentypen widerspiegeln.

5.2 Entwicklung von DyCE Komponenten

Wie in Kapitel 2.7.2 beschrieben, besteht die Grundstruktur einer DyCE-Komponente aus einem ModelObject und einer MobileComponent, die Benutzerschnittstelle zum ModelObject. Das ModelObject selbst hat ein oder mehrere Slotobjekte, in denen verteilte Daten gespeichert werden. Zur Verteilung über RMI (Remote Method Invocation) müssen diese serialisierbar sein, was in Java bedeutet, das Interface java.lang.Serializable zu implementieren. Da ModelObjects ebenfalls serialisierbar sind, können Slots auch Referenzen auf ModelObjects beinhalten[19].

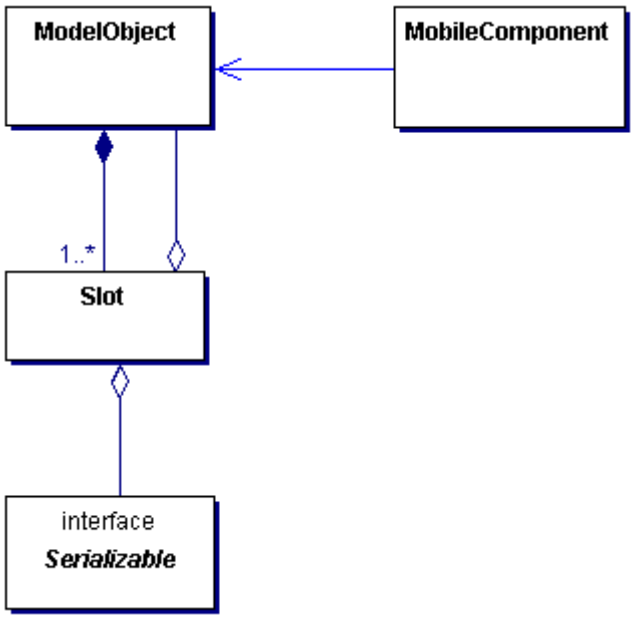


Abbildung 45: Grundstruktur einer DyCE-Komponente.

Auf Slotinhalte kann nur innerhalb von DyCE-Transaktionen (siehe 2.7.3) zugegriffen werden, damit keine inkonsistenten Zustände entstehen. Modifizieren verschiedene Teilnehmersysteme den gleichen Slot, so entsteht ein Konflikt.

Eine Datenstruktur für DyCE sollte daher Objekte, die keiner atomischen Semantik unterliegen, möglichst entkoppeln. Als Beispiel sei folgender Baum in einem DyCE-Datenmodell zu speichern.

---

[19] DyCE sieht nicht vor, daß Slots die Java-Objektreferenzen von ModelObjects direkt beinhaltet. Vielmehr sollten hier eindeutige Identifikationsnummern von ModelObjects, die sogenannten ObjectIDs gespeichert werden, über die man an das ModelObject gelangt.

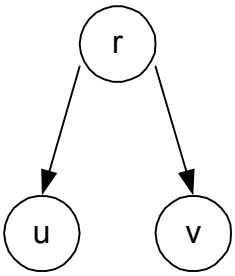


Abbildung 46: Beispiel für einen Baum mit den Knoten r, u und v.

Eine Möglichkeit der Implementierung ist es, alle Knoten in einem Slot zu sammeln, alle Kanten in einem anderen. Da ein Slot immer nur ein Objekt enthalten kann, verweist der Slot auf ein Vector-Objekt, daß eine Liste von Objekten verwaltet. Die Assoziation zwischen Knoten und Kanten erfolgt über deren Position im Vector.

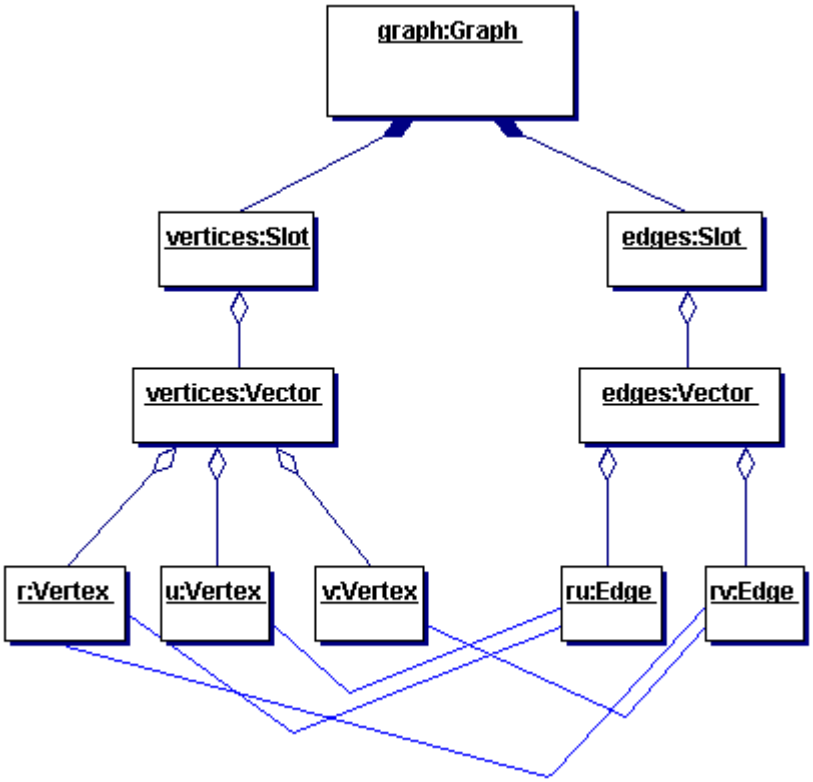


Abbildung 47: Konfliktträchtige Datenstruktur für den in Abbildung 46 angegebenen Baum. (UML-Objektdiagramm)

Wenn nun ein neuer Knoten u‘ unter u und ein neuer Knoten v‘ unter v eingefügt werden soll, muß in beiden Fällen der Vector namens „vertices“ modifiziert werden. Es kommt zu einem Konflikt.

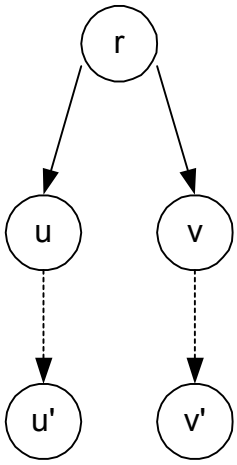


Abbildung 48: Modifikation des Baumes aus Abbildung 46.

Dieser Konflikt tritt jedoch nicht in folgender Modellierung auf. Hier referenziert das Graphobjekt nur die Wurzel des Baumes r in einem eigenen Slot. Jeder Knoten hat einen Slot in dem alle ausgehenden Kanten gespeichert werden, die Kanten wiederum besitzen je einen Slot, der ihren Zielknoten referenziert.

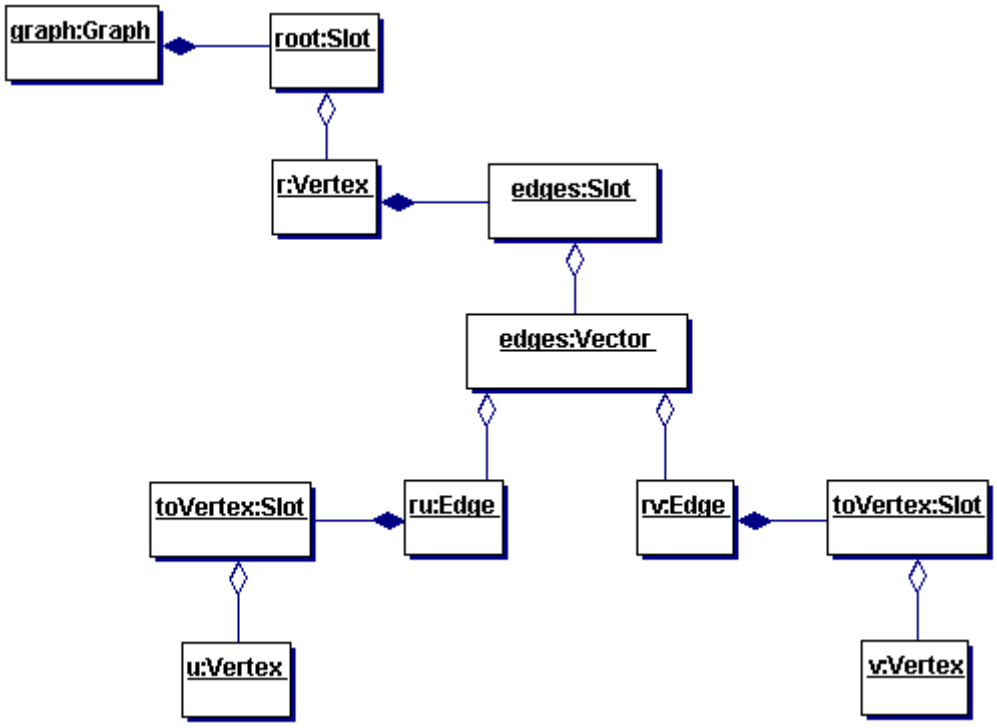


Abbildung 49: Verbesserte Struktur zur Implementierung von Bäumen in DyCE (UML-Objektdiagramm).

In dieser Modellierung betrifft das Hinzufügen des Knotens u‘ nur das Model-Object des Knotens u und das Hinzufügen von v‘ betrifft nur das ModelObject von Knoten v. Es tritt also kein Konflikt auf.

5.3 Speichern der Knoten- und Kanteninstanzen

Dieses Kapitel wendet die im vorigen Abschnitt besprochene Modellierungsweise auf Strukturgraphen an.

Der Strukturgraph, seine Knoten und Kanten werden allesamt durch Model-Objects repräsentiert.

Da die Modifikation von Knoten verschiedener Teilbäume möglichst entkoppelt sein soll, werden die Knoteninstanzen nicht direkt im Strukturgraphobjekt gespeichert.

Der Strukturgraph speichert allein eine Referenz auf den Wurzelknoten. Der Wurzelknoten w hat eine Referenz auf seine OutgoingEdgeCollection. Eine Kante e, die darin gespeichert ist, hält sowohl die Referenz auf ihren Startknoten (hier: w) als auch auf den Zielknoten, der ein Kind von w ist.

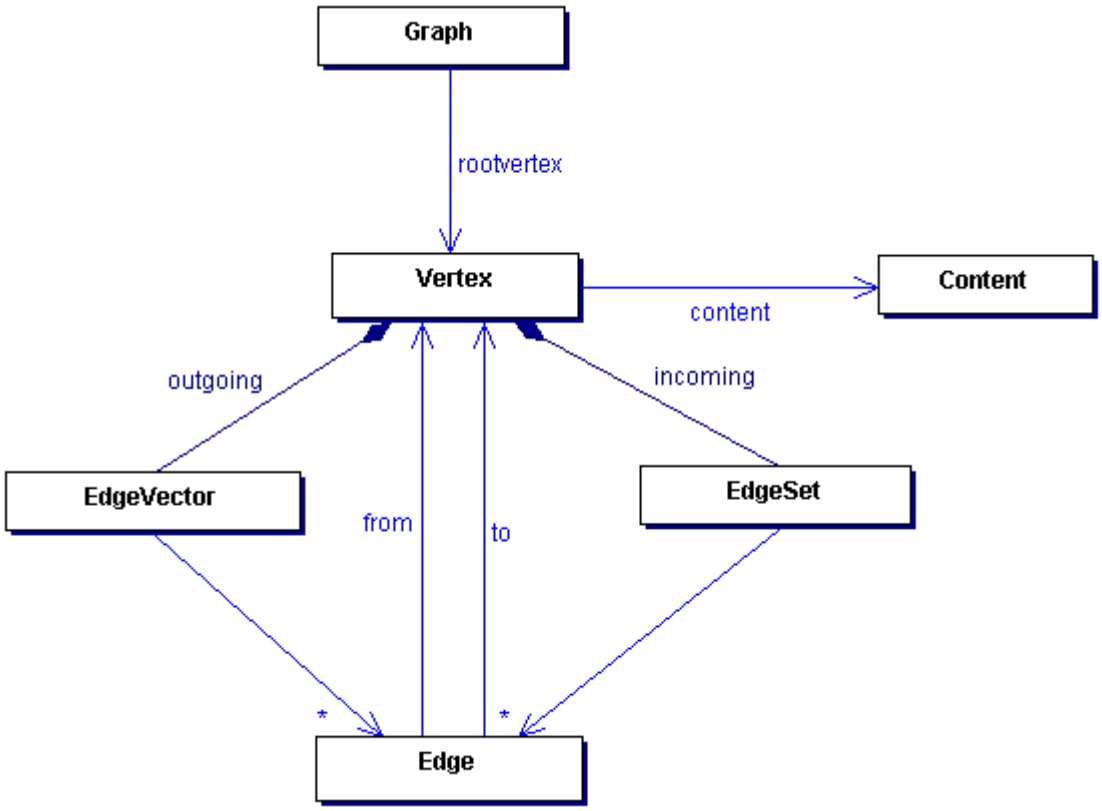


Abbildung 50: Implementierung von Strukturgraphen in DyCE (UML-Klassendiagramm).

Da ein Strukturgraph G = (V, E) zusammenhängend ist, und der einzige Knoten mit Innengrad null der Wurzelknoten ist, können auf diese Weise alle Knoten v $\in$ V und Kanten e $\in$ E erreicht werden.

Zur vollständigen Traversierung aller Knoten des Strukturgraphen reicht es aus, die Primärstruktur zu betrachten, denn diese spannt den Strukturgraphen auf.

5.4 Attribute, Kinder und Inhalt

Ein Knoten $v \in V$ kann Daten auf verschiedene Weisen speichern – direkt über das Inhaltsobjekt oder mittelbar über seine Nachfahren.

Oftmals möchte man zu einem Knoten nicht nur unspezifische Daten assoziieren, sondern konkrete Attribute angeben. In objektorientierten Programmiersprachen wie Java geschieht dies durch Erweitern der Klasse um Membervariablen und darauf operierenden Methoden.

Doch wie kreiert man ein Attribut, das eine Vertexinstanz referenziert, ohne die Strukturgrapheigenschaften zu verletzen?

Sei v eine Vertexinstanz. Sei a eine Vertexinstanz, die in v als Attribut gespeichert werden soll. Sei e eine Kante von v nach a. Je nach Semantik kann e eine Enthältkante oder eine Referenzkante sein. e wird in den EdgeVector der aus v ausgehenden Kanten eingefügt.

Die Vertexklasse von v wird um ein Attribut edgeToAttributeA:Edge erweitert. Dieses referenziert die Kante e.

Die Vertexklasse von v wird erweitert um die Methoden

- v.getAttributeA():Vertex und
- v.setAttributeA(Vertex a),

die die Kante edgeToAttributeA auflösen bzw. modifizieren.

Da a ein spezielles Kind (ein sogenanntes Attributkind) von v ist, möchte man die „echten“ Kinder von a unterscheiden. Dazu wird die neue Methode

- v.getNonAttributeOutgoingEdgeVector():EdgeVector

implementiert, die alle ausgehenden Kanten speichert, die keine Attribute referenzieren.

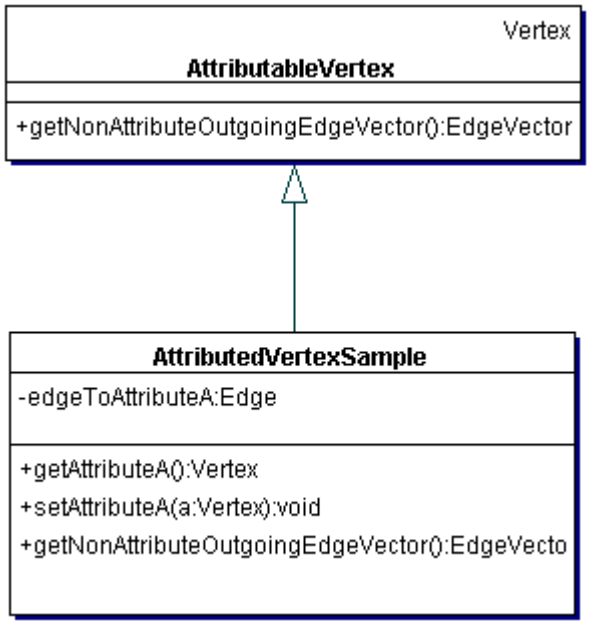


Abbildung 51: Implementierung von Knoten, die Attributkanten verwalten (UML-Klassendiagramm).

Gekapselt wird dies durch die Klasse AttributableVertex, deren Implementierung von getNonAttributeOutgoingEdgeVector() das gleiche Ergebnis liefert wie getOutgoingEdgeVector().

Vertexableitungen, die Attribute haben, überladen die Methode getNonAttributeOutgoingEdgeVector, um das Attributkind außen vor zu lassen.

Trotz dieser Implementierungsdetails entspricht ein Attributkind immer noch dem Property Pattern [Rie97]. Und wird daher in der üblichen UML Syntax beschrieben.

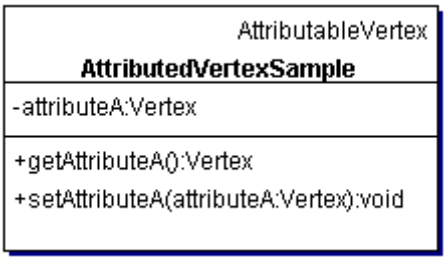


Abbildung 52: Knoten mit einem Attribut A, daß einen auf einen Knoten verweist, als Property Pattern (UML-Klassendiagramm).

5.5 Realisierung von Fragmentierung und Reservierung in DyCE

Sowohl verzögerte Fragmentierung als auch aktualisierende Reservierung werden unmittelbar vor einer Modifizierung aktiv. Es wird daher vorgeschlagen, modifizierende Methoden im Knoten folgendermaßen zu kapseln: Vor der eigentlichen Modifikation wird das Reservierungsmodul befragt, ob der Knoten bereits für das modifizierende Teilnehmersystem reserviert ist. Falls nicht, wird geprüft, ob es noch Vermerke für andere Teilnehmersysteme gibt und der Knoten ggf. fragmentiert. Der alte Reservierungsvermerk des fremden Teilnehmersystems muß nun in geeigneter Weise auf eines der neuen Fragmente übertragen werden. Anschließend wird ein Vermerk für das modifizierende Teilnehmersystem gesetzt, ein Timer gestartet und die Modifikation vorgenommen. Bei weiteren Modifikationen des Teilnehmersystems wird der Timer stets neu gestartet. Läuft dieser aus, hat das entsprechende Teilnehmersystem also längere Zeit keine Modifikation an dem Knoten vorgenommen. Der Reservierungsvermerk wird gelöscht, und, wenn möglich, Fragmente zusammengefaßt.

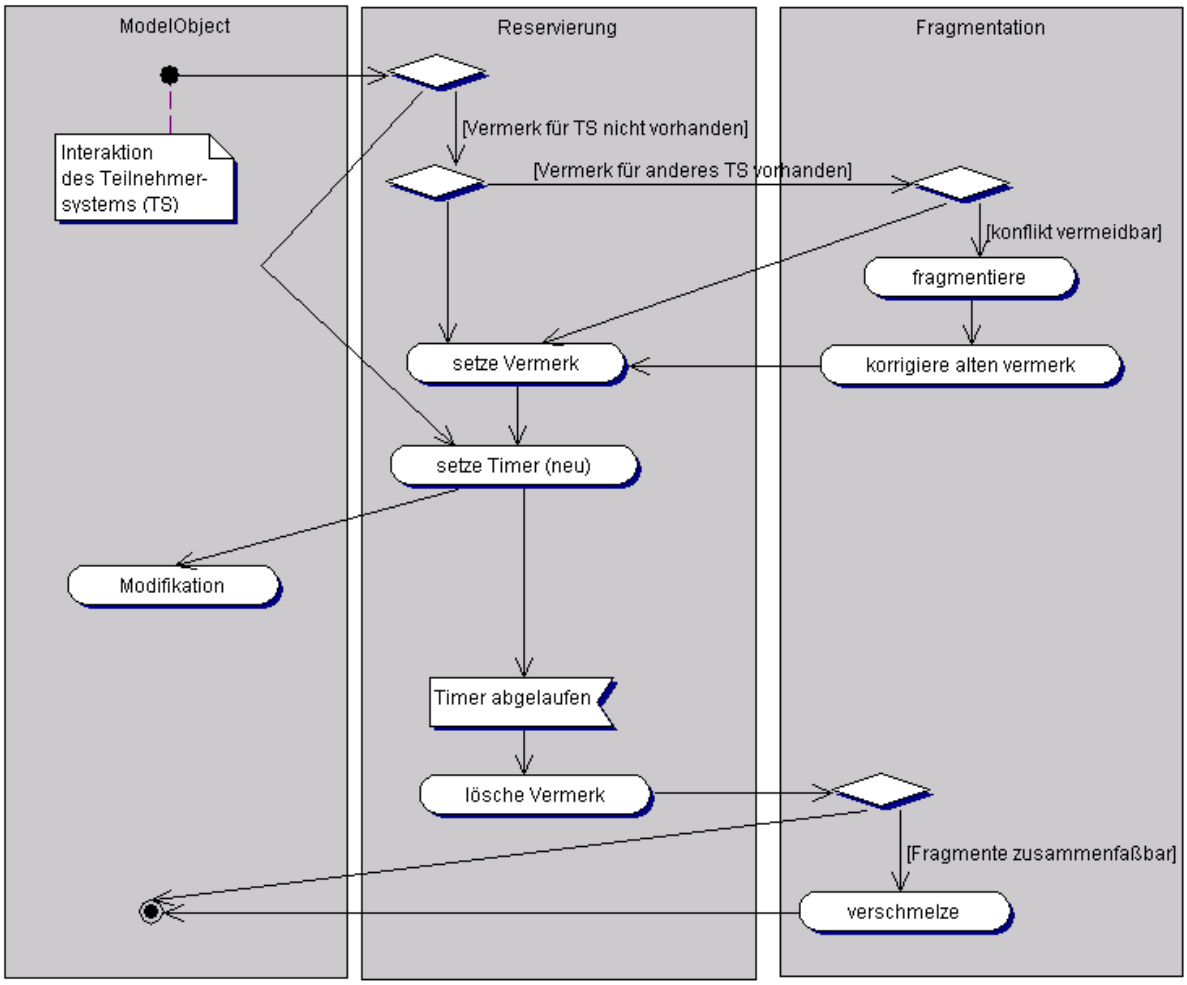


Abbildung 53: Implementierung der Heuristikkombination für Strukturgraphen für die einfache Reservierungsheuristik (UML-Aktivitätsdiagramm).

Um Sperrmechanismen zu implementieren, kann das Reservierungsmodul den kompletten Vorgang abbrechen, wenn es – selbst nach Fragmentierung – nicht möglich ist, den Knoten für das modifizierende Teilnehmersystem allein zu reservieren.

Die Abbildung 53 zeigt dieses Vorgehen für eine einfache Reservierungsheuristik. Zur Implementierung einer weiterreichenden Heuristik ersetze man die Aktivität „setze Vermerk“ durch das Hinzufügen des Grundes in die Heuristik (siehe 3.4) und Neuberechnung der Reservierungsfunktion. In diesem Schritt werden dann die Vermerke auf einzelne Knoten gesetzt oder gelöscht. Die Heuristik verwendet ihren eigenen Timer um veraltete Reservierungsgründe aus der Menge der beachteten Gründe zu eliminieren.

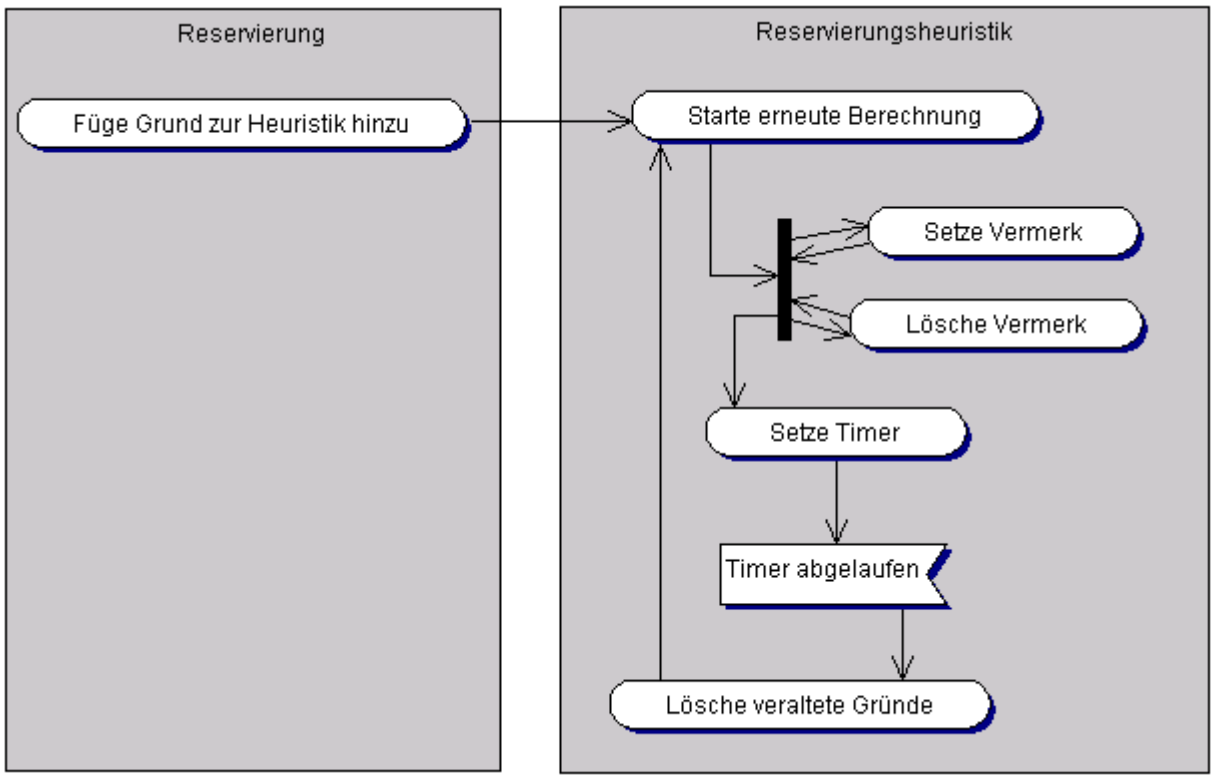


Abbildung 54: Implementierung einer Reservierungsheuristik in den in Abbildung 53 angegebenen Ablauf (UML-Aktivitätsdiagramm).

Um das Just-In-Time-Verfahren zu implementieren wird die beschriebene Kombination nur dann an einem Knoten aufgerufen, wenn dieser modifiziert werden soll, aber noch kein Reservierungsvermerk für das modifizierende Teilnehmersystem vorliegt.

Für die aktualisierende Reservierung wird die Kombination stets dann aufgerufen, wenn ein Knoten Bearbeitungsstelle wird.

5.6 Aufwand für den Anwendungsentwickler

In diesem Abschnitt wird diskutiert, welche Voraussetzungen erfüllt sein müssen, damit Strukturgraphen angewendet werden können, und welche Teilprobleme dem Anwendungsentwickler überlassen werden.

Ein Problem eignet sich zur Anwendung von Strukturgraphen, wenn es eine Struktur hat, die mittels Graphen abgebildet werden kann. Der Fragmentierungsmechanismus funktioniert nur dann, wenn vorwiegend Daten gespeichert werden, für die es eine Transformationsfunktion $Tsplit_n$ gibt, die die in der Definition der Partitionierbarkeit genannten Eigenschaften besitzt.

Zur Anwendung von Strukturgraphen ist es zunächst nötig, den semantischen Zusammenhang zwischen Knoten und Kanten des Strukturgraphen und der Problemdomäne herzustellen. Dazu muß der Entwickler entscheiden, für wel-

che Konzepte er Enthält- oder Referenzkanten verwendet und welche er über Attribute modelliert. Anschließend muß er Knotenklassen von den jeweiligen Typen StructuringVertex, RootVertex, LeafVertex, DynamicVertex und FinestDynamicVertex ableiten, um damit die für seine Anwendung nötige Funktionalität zu implementieren. Für inhaltstragende Knoten muß er spezielle Inhaltsobjekte implementieren, die die mit dem Knoten assoziierten Daten speichern. Die Inhaltsobjekte von dynamischen Knoten müssen die Transformationsfunktion $Tsplit_n$ auf geeignete Weise implementieren.

Für den XML-Export (und -Import) muß eine XML-Transformation für Inhaltsobjekte implementiert werden, die die Struktur aus 4.2.2 verwendet.

Es muß entschieden werden, ob die Reservierung zur Implementierung von Sperrmechanismen herangezogen werden soll, oder ob sie nur der Anzeige des Konfliktpotenzials dienen soll, mittels derer sich die Teilnehmer selbständig koordinieren. Abhängig davon wird der Zeitpunkt gewählt, an dem die Fragmentierungs- und Reservierungskombination für Strukturgraphen ausgeführt wird und eine Reservierungsheuristik implementiert.

Zum Schluß muß der Entwickler eine geeignete Präsentationskomponente entwerfen, über die der Benutzer mit dem Strukturgraphen interagiert. Diese sollte, neben der Unterstützung des Gruppenbewußtseins und Kommunikationsmöglichkeiten zwischen den Teilnehmern, auch konfliktbezogene Awareness-Informationen in geeigneter Weise bereitstellen. Dabei sind die unter 3.1 beschriebenen Aspekte zu bedenken.

Der Entwickler erhält mit den Strukturgraphen eine Datenstruktur, die einerseits hierarchische Informationen speichert, aber auch die Verwaltung von allgemeinen Graphstrukturen ermöglicht.

Allein durch die Implementierung der Transformationsfunktionen $Tsplit_n$ und $Tjoin_n$ auf den zu verwendeten Inhalten ist die Datenstruktur selbständig in der Lage die Konfliktgranularität an die Situation anzupassen. Informationen über das Konfliktpotenzial sind leicht extrahierbar und die Grundlage für Sperrmechanismen ist ebenfalls enthalten.

In Kapitel 6 wird gezeigt, wie Strukturgraphen für die kooperative Textverarbeitung herangezogen werden können.

5.7 Zusammenfassung Implementierung von Strukturgraphen

Es wurden Operationen zur Modifikation der Graphstruktur eingeführt, die sicherstellen, daß die Eigenschaften der Strukturgraphen erfüllt bleiben. Dazu gehören sowohl Operationen zum Hinzufügen und Löschen von Knoten und Referenzkanten, als auch die Split- und Joinoperation zur Fragmentierung und Defragmentierung von Knoten. Es wurden verschiedene Knotenklassen eingeführt, die sich leicht zur Erfüllung von Anforderung 1 (Untergliederung des Dokuments in Kapitel und andere logische Einheiten) heranziehen lassen.

Es wurde besprochen, wie eine Implementierung von Strukturgraphen Knoten und Kanten gespeichert werden sollten, damit keine unnötigen Konflikte ausge-

löst werden. Weiterhin wurde ein Mechanismus beschrieben, mittels derer Attributkinder von der Strukturgraphimplementierung verwaltet werden können.

Auf den implementierungstechnischen Aspekt der Fragmentierungs- und Reservierungsverfahren wurde eingegangen.

Abschließend wurde beschrieben, welche Problemstellung der Anwendung einer Strukturgraphimplementierung überlassen werden.

6 Anwendungsfall: Kooperative Textverarbeitung

Im Folgenden wird beschrieben, wie auf der Basis einer vorhandenen Implementierung der Strukturgraphen eine DyCE-Komponente zur kooperativen Verarbeitung strukturierter Textdokumente erstellt werden kann. Dabei müssen die in 2.5 vorgestellten Anforderungen erfüllt werden.

6.1 Anwendung auf Strukturgraphen

In einer Textverarbeitung müssen verschiedene Arten von Inhalt präsentiert werden: Text, Tabellen und Bilder, wobei Bilder mit externen Applikationen bearbeitet und in der Textverarbeitung nur angezeigt werden. Der Text muß nach Anforderung 2 (Formatierungen) mit Schriftart, -größe und -farbe und Merkmalen wie Fettsatz, Kursivschrift und Unterstreichungen formatierbar sein.

Textdokumente haben einen eindimensionalen Charakter. Alle Buchstaben und Bilder sind „von links nach rechts“ angeordnet. Diese Ordnung wird im Strukturgraph durch die Kantensortierung wiedergegeben. Zur besseren Lesbarkeit werden die Zeilen in der Darstellung umgebrochen.

Um Anforderung 4 (Parallele Modifikation) und Anforderung 10 (Vermeidung von Konflikten) zu erfüllen, müssen die zu bearbeiteten Inhalte partitionierbar sein und dabei die Distanzen zwischen Textteilen berücksichtigen. Ist F die Menge aller Formatierungsmöglichkeiten und Σ der Zeichensatz, so besteht formatierter Text aus Elementen der Menge $\Sigma \times F$. Sei der formatierte Text $t \in (\Sigma \times F)^*$, wobei $t = r \circ s$ mit $r, s \in (\Sigma \times F)^*$ dann gilt $Tsplit_2(t) = (r, s)$ und $Tjoin_2(r,s) = t$. Formatierte Textinhalte können folglich in dynamischen Knoten gespeichert werden. Zur Implementierung empfiehlt sich eine speichereffizientere Verwaltung der Formatierungsinformationen, zumal es hochwahrscheinlich ist, daß auf einen mit $f \in F$ formatiertes Zeichen ein weiteres mit der gleichen Formatierung f folgt.

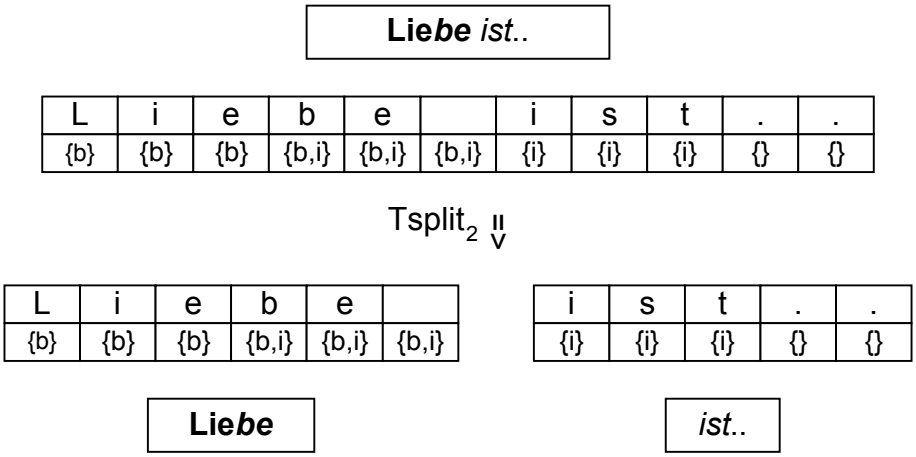


Abbildung 55: Beispiel für die Fragmentierung von formatiertem Text.
In oberen Rechtecken sind jeweils die textuellen Informationen, in unteren Rechtecken die Menge der Formatierungsinformationen angegeben. „b“ steht dabei für Fettdruck und „i“ für kursiv.

Um die Anforderung 3 (XML-Export) vollständig zu erfüllen, muß eine umkehrbare Abbildung zwischen den Inhaltsobjekten und XML-Quellcode definiert werden. Hierfür empfiehlt sich eine zu HTML (Hypertext Markup Language) äquivalente Speicherung, die mit den Vorgaben für XML weitgehend verträglich ist.

Tabellen sind Container von gitterartiger Struktur, die in ihren Zellen formatierten Text, Bilder und weitere Tabellen beinhalten können. Daher ist es sinnvoll, die Partitionierung nicht im Tabellencontainer selbst vorzunehmen, sondern in den Zellen die entsprechenden Vertexinstanzen zu referenzieren und die Partitionierung an sie zu delegieren.

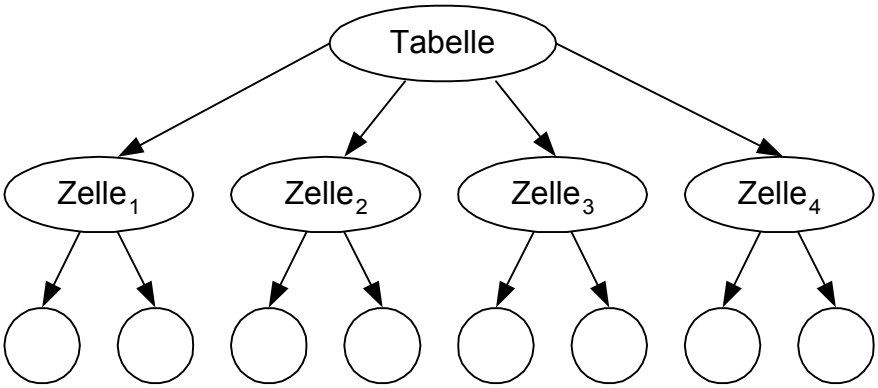


Abbildung 56: Implementierung einer Tabelle.

Neben den syntaktischen Strukturen, wie Text, Tabellen und Bilder, sind nach Anforderung 1 (Untergliederung des Dokuments in Kapitel und andere logische Einheiten) Informationen über die Dokumentstruktur zu speichern.

Dies sind:

- die Kapitelhierarchie,
- Absatzinformationen,

- dokumentweite Attribute wie beteiligte Autoren und Erstellungsdatum und
- inhaltsbezogene logische Informationen wie Überschriften der Kapitel oder Titel des Dokuments.

Die Bildung der Kapitelhierarchie wird durch Knoten des Typs StructuringVertex übernommen. Ihnen wird ein Attributknoten hinzugefügt, der die Überschrift symbolisiert. Dieser ist unter allen Kindsknoten des Kapitelknotens der Erste.

Alle dokumentweit gültigen Eigenschaften werden in einer Ableitung der Klasse RootVertex gespeichert. Der Titel wird analog zur Überschrift in einem Attributknoten verwaltet.

Die Absatzinformationen werden nicht gesondert gespeichert, da sie sich aus den Paragraphenzeichen rekonstruieren lassen[20].

6.2 Knoten der Textverarbeitung

Im Folgenden wird beschrieben, wie die Knotentypen des Strukturgraphen für die logischen Einheiten einer Textverarbeitung verwendet werden.

ArbitraryText

Formatierter Text wird grundsätzlich durch eine Instanz der Klasse ArbitraryText oder deren Ableitungen dargestellt. Arbitrary Text speichert ein beliebiges Stück zusammenhängenden Textes, unabhängig von Interpunktion oder Absatzzeichen. ArbitraryText speichert keine logische Struktur, implementiert dafür die Operationen split() und join().

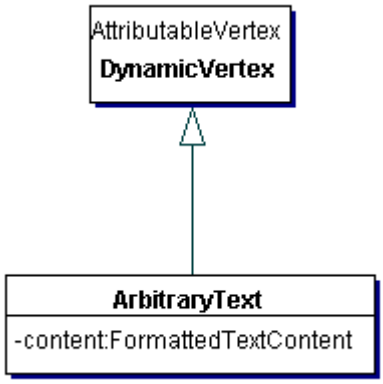


Abbildung 57: Die Klasse ArbitraryText (UML-Klassendiagramm)

[20] besondere Zeilenumbruchsarten werden hier nicht behandelt.

Paragraph, Sentence und Word

Paragraph und Sentence sind Ableitungen von ArbitraryText, die ihren Inhalt auf vollständige Absätze bzw. Sätze beschränken.

Ein vollständiger Absatz wird durch zwei Absatzmarken „\n“ markiert, wobei nur das letzte „\n“ zum Absatz gehört.

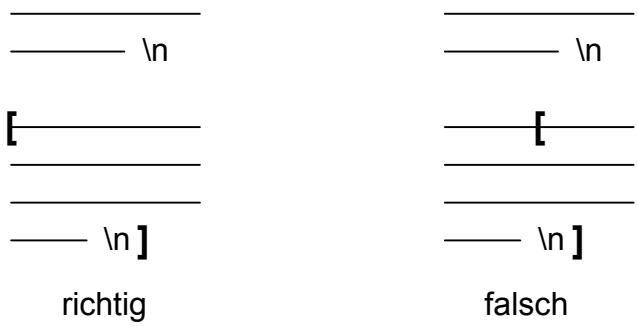


Abbildung 58: Definition eines Absatzes

Ein vollständiger Satz beinhaltet den Punkt „.“ am Ende des beinhalteten Textes. In Sätzen können weitere Vorkommen des Punktes, wie in Abkürzungen, nicht ausgeschlossen werden. Links an den Satz schließt sich ebenfalls ein Punkt an.

Word beinhaltet genau ein Wort mit Interpunktionszeichen und Abstandszeichen wie Leerzeichen und Tabulatoren. Der textuelle Inhalt entspricht dem Regulären Ausdruck $(a+i)^+ \circ w^+$, wenn a für einen Buchstaben, i für ein Interpunktionszeichen und w für Abstandszeichen steht.

Da Paragraph und Sentence Ableitungen von ArbitraryText sind, implementieren sie die Split-Operation. Ein Paragraph spaltet sich dabei bevorzugt in Sentence-Instanzen, Sentence-Objekte in Word-Instanzen.

Auch wenn das Inhaltsobjekt für formatierten Text eine Partitionierung auf Zeichenebene zuläßt, ist es wenig hilfreich, wenn Wortteile für verschiedene Teilnehmersysteme reserviert werden. Daher ist eine Wordinstanz die feinstmögliche Fragmentierungseinheit. Sie unterstützt daher die Split-Operation nicht.

Die Restriktionen der Klassen Paragraph, Sentence und Word bergen einige Effizienznachteile. Ist es notwendig, einzelne Absätze, Sätze oder Wörter zu bestimmen, lassen sich diese auch nach Speicherung in der effizienteren Datenstruktur ArbitraryText extrahieren. Diese Klassen werden daher nicht zur Speicherung von allgemeinem Text verwendet. Sie finden ihre Anwendung vorwiegend in Vertex-Attributen wie z.B. Überschriften, die genau einen Absatz enthalten dürfen.

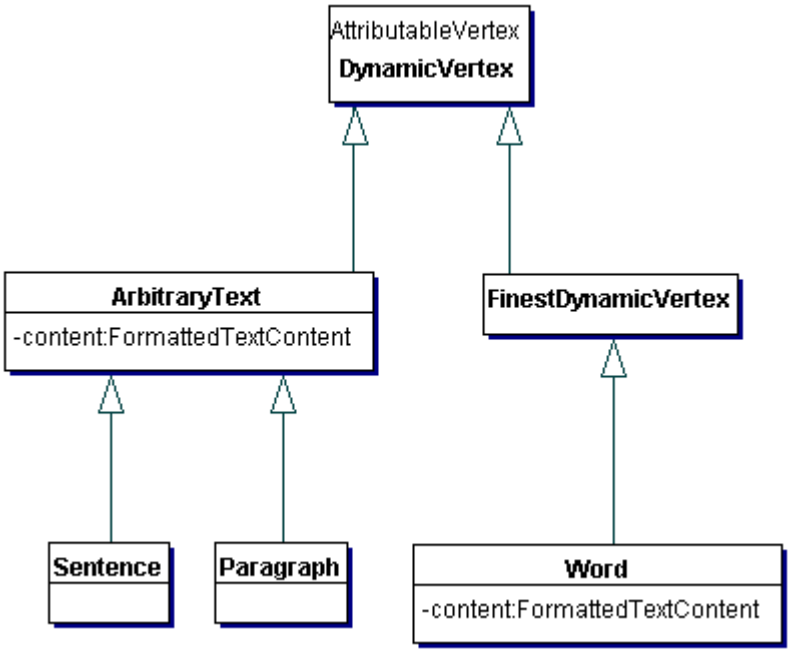


Abbildung 59: Die Klassen Sentence, Paragraph und Word (UML-Klassendiagramm)

Document und Chapter

Document ist die Vertexinstanz, die den Wurzelknoten repräsentiert. Pro Strukturgraph gibt es genau einen Document-Knoten. Document hat ein Attribut namens „title“, das auf eine Paragraphinstanz verweist.

Chapter hat ein Attribut namens „heading“, das ebenfalls auf eine Paragraphinstanz verweist und die Überschrift des Kapitels bezeichnet.

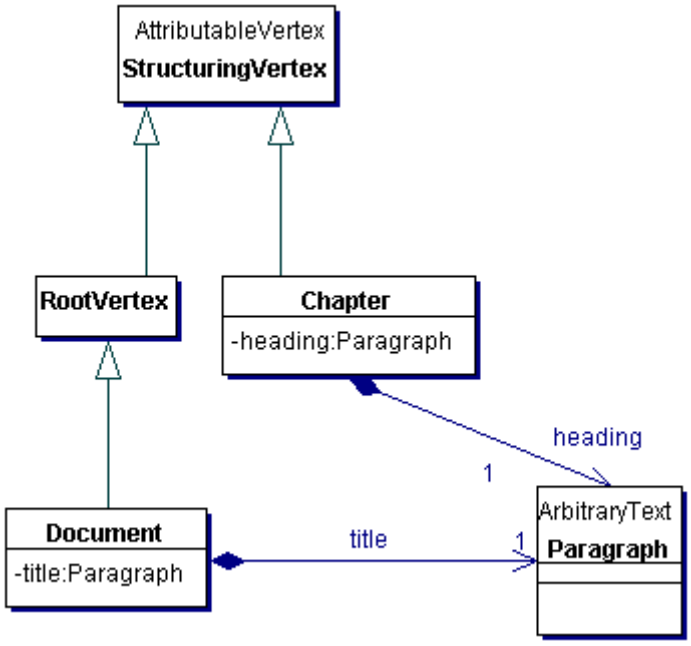


Abbildung 60: Die Klassen Document und Chapter und deren Titel bzw. Überschrift (UML-Klassendiagramm)

Table

Eine Tabelle habe r Zeilen und c Spalten. Die entsprechende Table-Instanz besitzt genau r*c viele Kinder, die alle vom Typ TableCell sind. Eine TableCell-Instanz gruppiert die Objekte, die zu einer Zelle gehören.

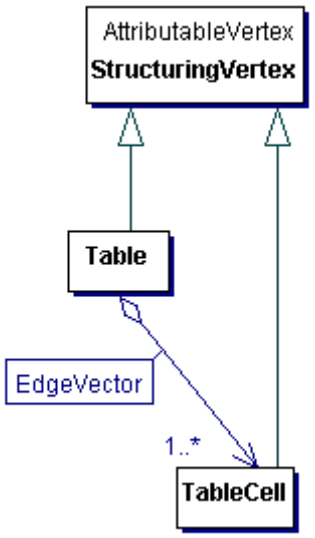


Abbildung 61: Die Klassen Table und TableCell (UML-Klassendiagramm)

Picture

Pictureobjekte sind Repräsentanten für eingebettete Bilder. Diese Knoten sind nicht dynamischer Natur und immer kinderlos.

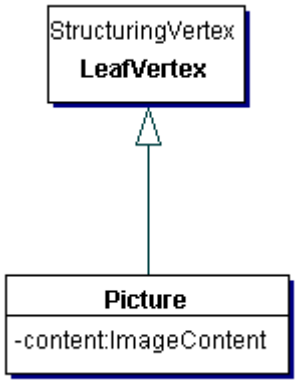


Abbildung 62: Die Klasse Picture (UML-Klassendiagramm)

Beispiel

Hier nun ein Beispiel für die Verwendung der Knoteninstanzen. Chapter verweist auf eine Paragraphinstanz, die die Kapitelüberschrift im Sinne eines Attributkindes beinhaltet. Die „anderen“ Kinder beschreiben den Inhalt des Kapitels. In diesem Beispiel ist das ein Textblock in dessen Mitte ein Bild eingebettet ist.

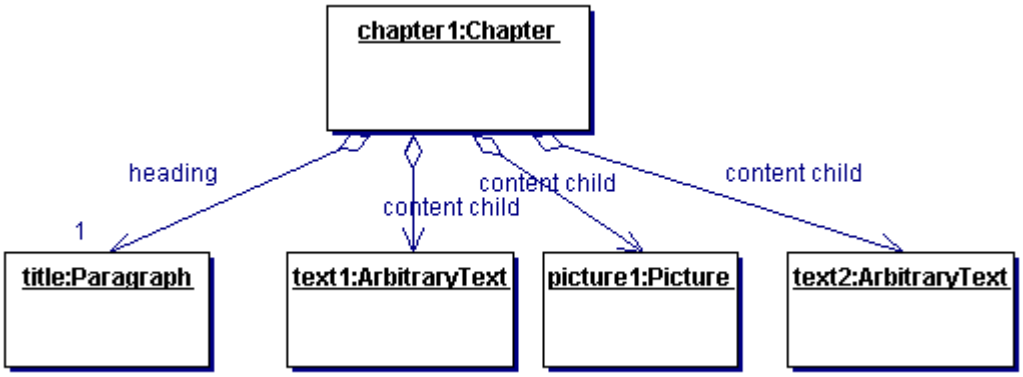


Abbildung 63: Beispiel für ein Dokument als Strukturgraph (UML-Objektdiagramm).

Die Darstellung davon könnte so aussehen:

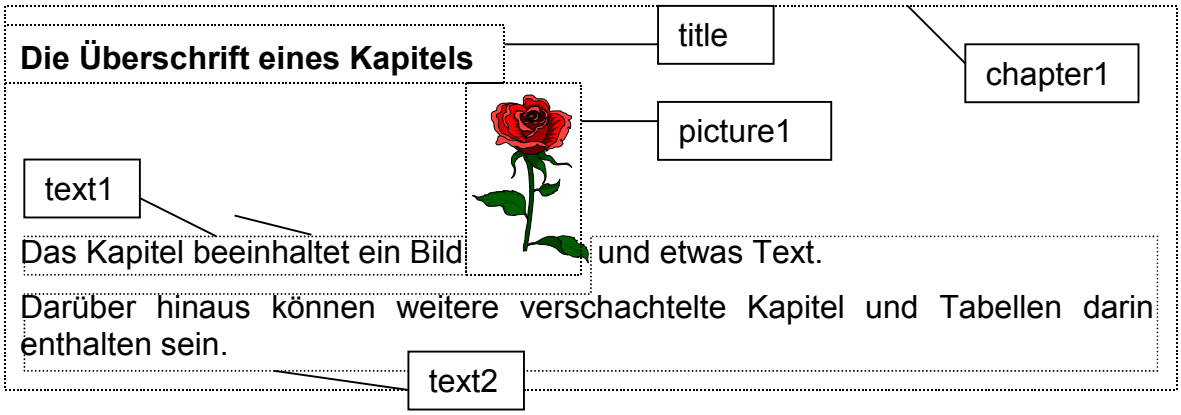


Abbildung 64:Beispiel zum Strukturgraphen aus Abbildung 63.

6.3 Bearbeitungsstellen

Reservierungsheuristik und Fragmentierung basieren auf der Definition der Bearbeitungsstelle. Doch wann kann man sagen „Teilnehmersystem τ bearbeitet einen Knoten x“? Eine Antwort ist: „das Teilnehmersystem, daß x zuletzt modifiziert hat“. Dabei werden Modifikationen, die übermäßig lange zurückliegen, nicht gewertet. Werden Knoten parallel modifiziert, führen mehrere Teilnehmersysteme kurz hintereinander Modifikationen darauf aus. Dann sollte man sagen „alle diese Teilnehmersysteme bearbeiten x“.

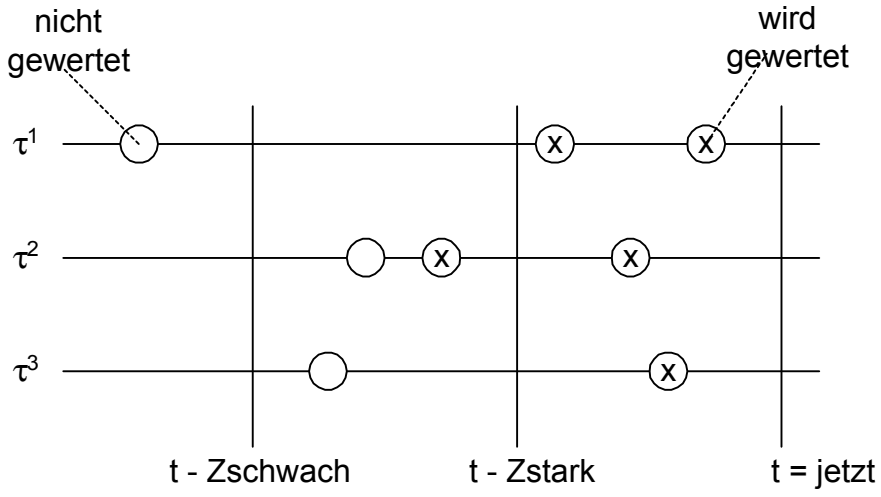


Abbildung 65: starke und schwache Bearbeitungsstellen

Zu global vorgegebenen Zeitlängen Zstark und Zschwach, mit Zstark $\leq$ Zschwach ist

$Schwach_t(x) = \{ \tau \in T \mid lastModified(x) = (t_1, \tau), t_1 \in [t - Zschwach, t] \text{ und } \forall \tau' \text{ mit } lastModified(x) = (t', \tau') \text{ gilt } t_1 \geq t' \}$

$Stark_t(x) = \{ \tau \in T \mid lastModified(x) = (t_1, \tau), t_1 \in [t - Zstark, t] \}$

$$bearbeitungsstelleVon_t(x) = \begin{cases} Stark_t(x) & Stark_t(x) \neq \{\} \\ Schwach_t(x) & \text{sonst} \end{cases}$$

6.4 Reservierungsheuristiken

Wie in Kapitel 4.3 beschrieben, ist eine Möglichkeit, bei Strukturgraphen auf inneren Knoten stets die einfache Reservierungsheuristik anzuwenden, die nur den aktuell betroffenen Knoten reserviert. In diesem Kapitel werden daher zunächst verschiedene Heuristiken besprochen, die nur auf Blätter angewandt werden. Zum Schluß wird eine Heuristik vorgestellt, mit der es – in einigen Fällen – möglich wird, auch innere Knoten im voraus zu reservieren.

Insbesondere die Ausprägung auf Blätter angewandte Reservierungsheuristiken hängt stark von der Anwendungsdomäne ab. Eine sehr einfache Strategie ist es, neben den Knoten, der aktuell bearbeitet wird, beide Nachbarn im voraus zu reservieren (Nachbarschaftsheuristik). Ein Vorteil dieser Heuristik ist, daß wenn die Fragmentierung in dem aktuellen Bereich hoch ist, sprich: mehrere Teilnehmersysteme mit wenig Abstand voneinander arbeiten, die tatsächliche Reservierung weniger umfangreich ausfällt, als wenn die Fragmentierung niedrig ist.

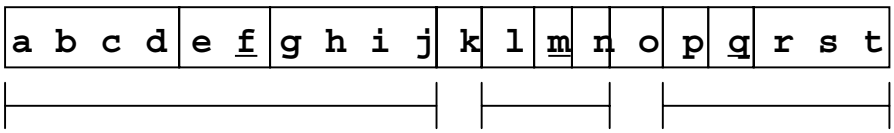


Abbildung 66: Anpassung der Nachbarschaftsheuristik an den Fragmentierungsgrad. Benachbarte Rechtecke symbolisieren Knoten, auf denen die Nachbarschaftsrelation erfüllt ist. Die Zeichen darin stellen die Knoteninhalte dar. Das unterstrichene Zeichen repräsentiert die Bearbeitungsstelle. Durch eine gemeinsame Linie unterstrichene Knoten deuten Knoten mit dem gleichen Reservierungsvermerk an. Man sieht, daß der mittlere Verbund deutlich weniger Zeichen umfaßt als der erste.

In manchen Fällen ist die Hinzunahme der Nachbarn nicht ausreichend. Deshalb stellt die Verwendung einer statistisch adaptierenden Umgebungsheuristik eine Verbesserung dar. Damit stellt sich jedoch die Frage nach dem Distanzmaß. Wählt man die Anzahl der Buchstaben als Distanzmaß, unterliegt die Größe der Reservierung nur kleinen Schwankungen, die aus der statistischen Adaption herrühren. Dann jedoch berücksichtigt die Reservierungsgröße nicht den Abstand der Bearbeitungsstellen verschiedener Teilnehmersysteme von einander. Weiterhin müßten reservierte Knoten freigegeben werden, wenn durch das Wachsen von Fragmenten, diese Knoten die gewünschte Distanz überschreiten. Verwendet man die Anzahl der Fragmente als Distanzmaß, än-

dert sich die Größe der Reservierung sprunghaft, wenn mehrere Knoten zusammengefaßt werden.

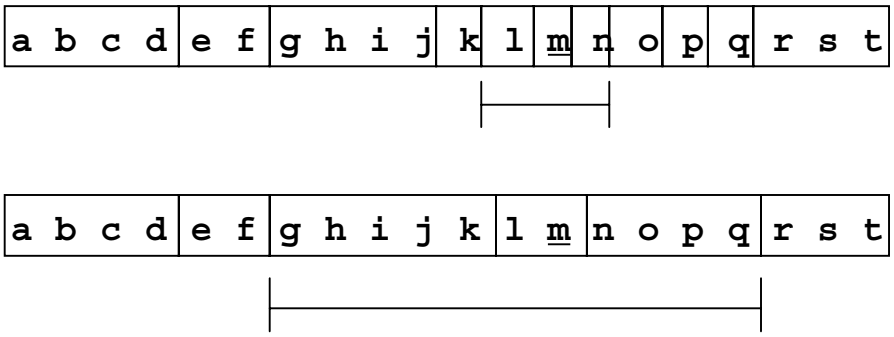


Abbildung 67: Änderung der Umgebung beim Zusammenfassen von Fragmenten. In beiden Fällen liegt die Bearbeitungsstelle am Zeichen m, doch die Anzahl der reservierten Zeichen unterscheidet sich stark.

Leider ignorieren beide Ansätze logische Grenzen innerhalb von Dokumenten, wie etwa Kapitel oder Absatzgrenzen. So kann es durchaus auftreten, daß eine Reservierung einen Teil eines Kapitels und noch drei Zeichen des vorigen Kapitels beinhaltet.

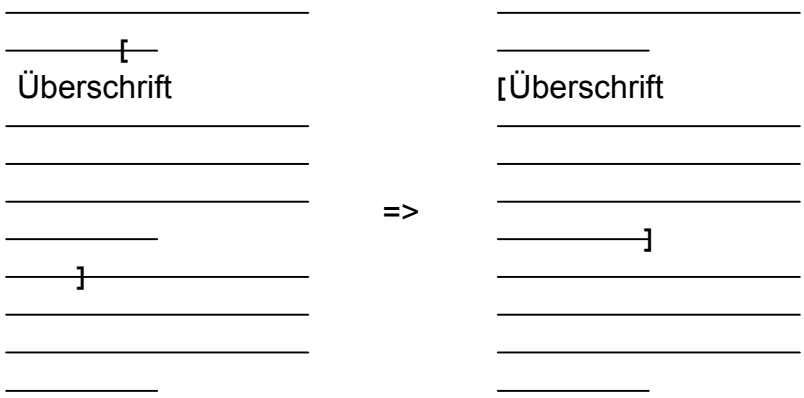


Abbildung 68: Reservierung unter Beachtung von logischen Grenzen.

Eine Lösung für dieses Problem ist die Erweiterung einer Heuristik, die nur auf Blättern operiert um den sogenannten Wasserspiel-Ansatz. Dazu wird die logische Struktur des Dokuments herangezogen um Knoten zu „Wasserwannen“ zu gruppieren. So, wie das Wasser in einer Wanne erst dann überläuft, wenn diese vollständig gefüllt ist, so werden weitere Knoten nur dann entsprechend reserviert, wenn alle Knoten der Wanne für dieses Teilnehmersystem reserviert sind.

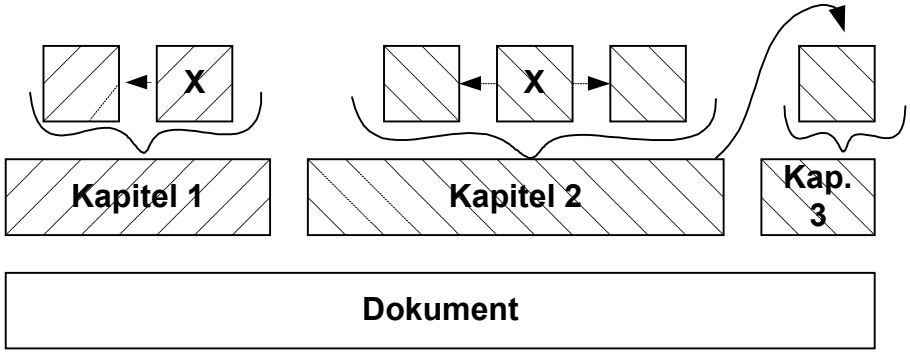


Abbildung 69: "Fluß" der Reservierung im Wasserspiel-Ansatz.

Die Rechtecke repräsentieren logische Einheiten im Dokument. Übereinander angeordnete Rechtecke stellen die Enthaltenseinsbeziehung dar. Die mit einem Kreuz markierten Rechtecke sind Bearbeitungsstellen (genauer: Reservierungsgründe) verschiedener Teilnehmersysteme. Auf Knoten gleicher Ebene wird die ursprüngliche Heuristik angewandt. Das gleiche gilt für die Situation nach einem Überlauf.

Anmerkung: Wird ein strukturiertes Dokument in einem Strukturgraphen gespeichert, so ist die logische Struktur des Dokuments durchaus mit der des Strukturgraphen verwandt, jedoch nicht identisch. Der Unterschied ergibt sich im wesentlichen durch die Fragmentierung.

Durch diesen Ansatz fallen die resultierenden Reservierungen zwar kleinflächiger aus als in der ursprünglichen Heuristik, doch dies läßt sich durch eine Adaption der Größenparameter bei Umgebungs- und Einflußheuristik ausgleichen.

6.5 Benutzungsschnittstelle in DyCE

Eine kooperative Textverarbeitung muß neben der Funktionalität einer herkömmlichen Textverarbeitung auch Werkzeuge zur Förderung des Gruppenbewußtseins anbieten.

Dabei kommt es besonders auf die Benutzungsschnittstelle an: sie muß gleichzeitig informativ sein und eine intuitive Bedienung zulassen. Sie darf den Benutzer nicht mit unnötigen Informationen und Funktionen überfrachten oder ihn in seinem Arbeitsfluß einschränken.

Verwendung als Textverarbeitung

Die zur Bearbeitung von Textdokumenten nötige Funktionalität wird durch die Werkzeugleiste am oberen Bildschirmrand zu Verfügung gestellt. Dies sind (von links nach rechts)

- Einfügen eines neuen Kapitels,
- Ausschneiden und Kopieren in die, sowie Einfügen von Text aus der Zwischenablage,

- Änderung der Schriftartfamilie und -größe,
- Wählen von Fettsatz, Kursivschrift oder Unterstreichungen,
- Änderung der Schriftfarbe
- und Ausrichten des Absatz am linken Rand, zentriert oder am rechtem Rand

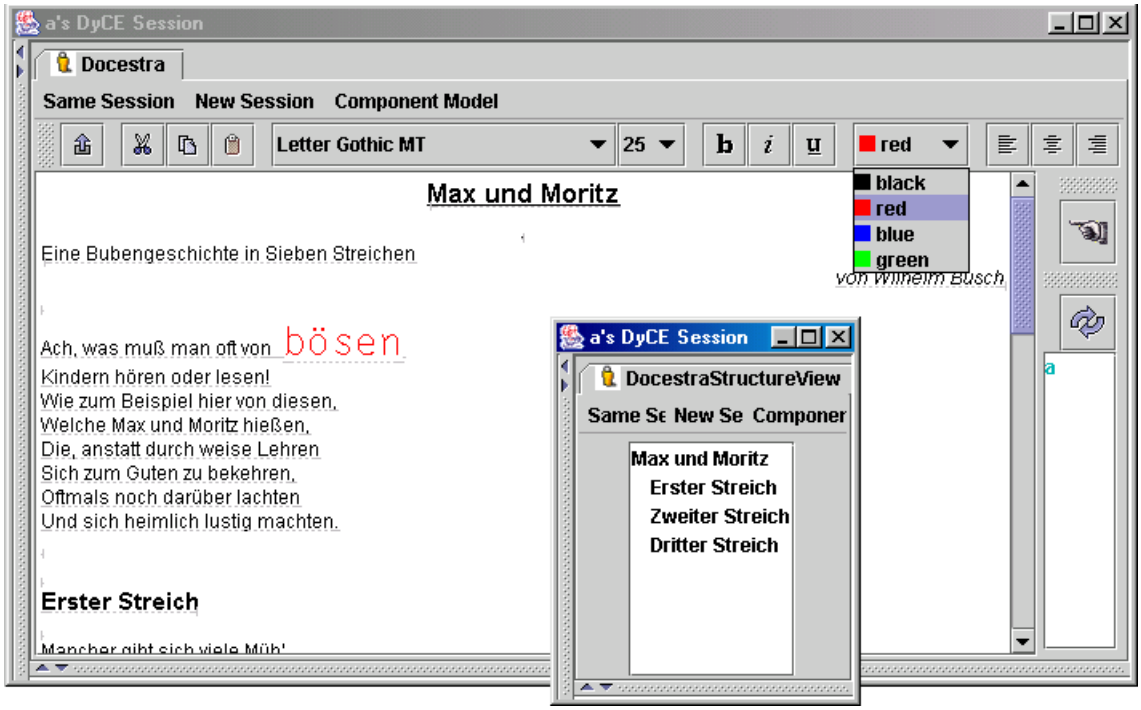


Abbildung 70: Formatierung und Bearbeitung von Text, Aufbau und Anzeige der logischen Struktur. (Screenshot aus dem Prototyp)
Durch die Werkzeugleiste am oberen Fensterrand kann der Text bearbeitet und mit Formatierungen ausgeschmückt werden. Mit dem ersten Knopf können Kapitel mit Überschriften in das Dokument eingefügt werden. Durch die damit aufgebaute Gliederung kann mit dem Strukturansicht navigiert werden.

Als Navigationshilfe wurde eine Strukturansichtskomponente entwickelt, die nur die Gliederung zeigt. Durch Auswählen einer Überschrift in der Strukturansicht springt der Cursor in der Textverarbeitung an die entsprechende Stelle.

Unterstützung der kooperativen Arbeit

Als Zeigewerkzeug in Diskussionen hat jeder Teilnehmer einen Telepointer, der mit seinem Namen gekennzeichnet ist. Dieser kann bei Bedarf angeschaltet und verschoben werden.

Wird Text automatisch am Fensterrand umgebrochen, so kann das gleiche Wort bei verschiedenen Teilnehmersystemen an verschiedenen Stellen ange-

zeigt werden. Da der Telepointer auf einem pixelbasiertem Koordinatensystem beruht, dürfen keine automatischen Zeilenumbrüche erfolgen.

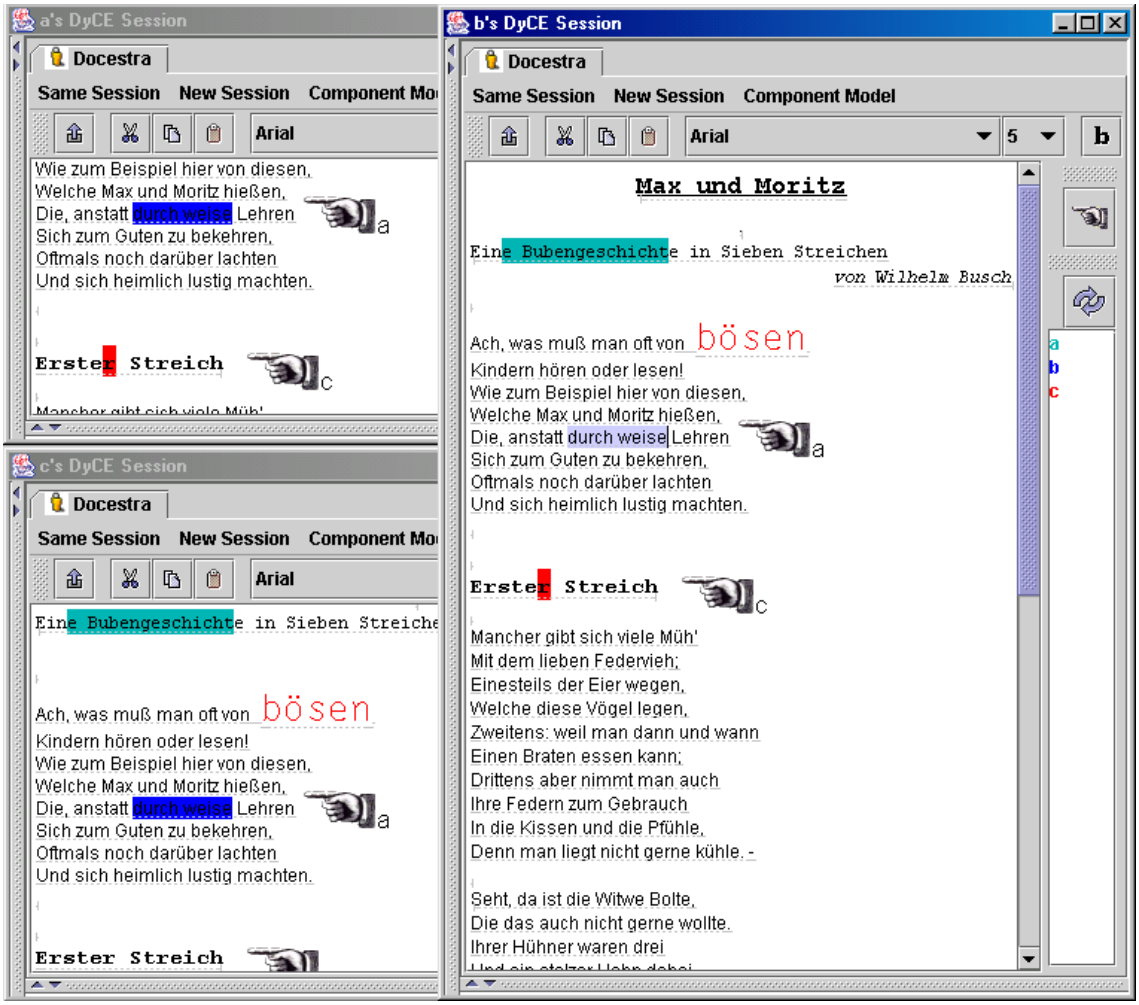


Abbildung 71: Verwendung von Telepointer und Anzeige von Cursor und Textselektionen (Screenshot vom Prototyp)

Eine Sitzung der Benutzer A, B und C. Benutzer a positioniert seinen Telepointer auf das Vorwort und fragt: „Das hier sollten wir umformulieren.“ Benutzer B markiert einen Teil des Textes um ihn zu Löschen. Die anderen Benutzer erkennen dies an der blauen Hinterlegung.

Durch die Kennzeichnung von Cursorpositionen und Textselektionen der anderen Teilnehmer durch farbige Hinterlegungen, hat ein Benutzer jeder Zeit einen Überblick, wer an welcher Stelle arbeitet.

Welcher Benutzer durch welche Farbe symbolisiert wird, zeigt die Liste am rechten Fensterrand. Seine eigene Farbe kann jeder Benutzer selbst bestimmen. In das Farbauswahlmenü gelangt er durch einen Mausklick auf das Symbol mit den beiden Pfeilen. Die Zuordnung wird zusammen mit den Dokumentdaten gespeichert, so daß ein Benutzer bei einer Wiederaufnahme der Sitzung die Farbwahl nicht wiederholen muß.

Die gewählten Farben werden weiterhin zur Anzeige des Konfliktpotenzials benutzt. Dazu werden Bearbeitungsstellen eines Teilnehmersystems mit einer

durchgehenden Linie unterstrichen. Teile, die zusammen mit einer Bearbeitungsstelle reserviert wurden, sind mit einer gestrichelten Linie markiert.

Modifiziert ein Benutzer ein Fragment, daß für ein anderes Teilnehmersystem reserviert wurde, so versucht die Applikation zunächst eine weitergehende Fragmentierung durchzuführen, um den Konflikt zu bannen. Ist die feinste Fragmentierung bereits erreicht, in dieser Applikation entspricht sie einem Wort, so müssen beide Benutzer mit Konflikten rechnen. Durch eine mehrfache Unterstreichung wird den Benutzern signalisiert, welche anderen Teilnehmersysteme hier eine Modifikation anstreben könnten.

Zu Verdeutlichung der Konflikteinheiten sind die Ausmaße der Fragmente durch kleine senkrechte Striche an den Enden der Unterstreichung gekennzeichnet. Die geben weiterhin Aufschluß über den Fragmentierungsgrad.

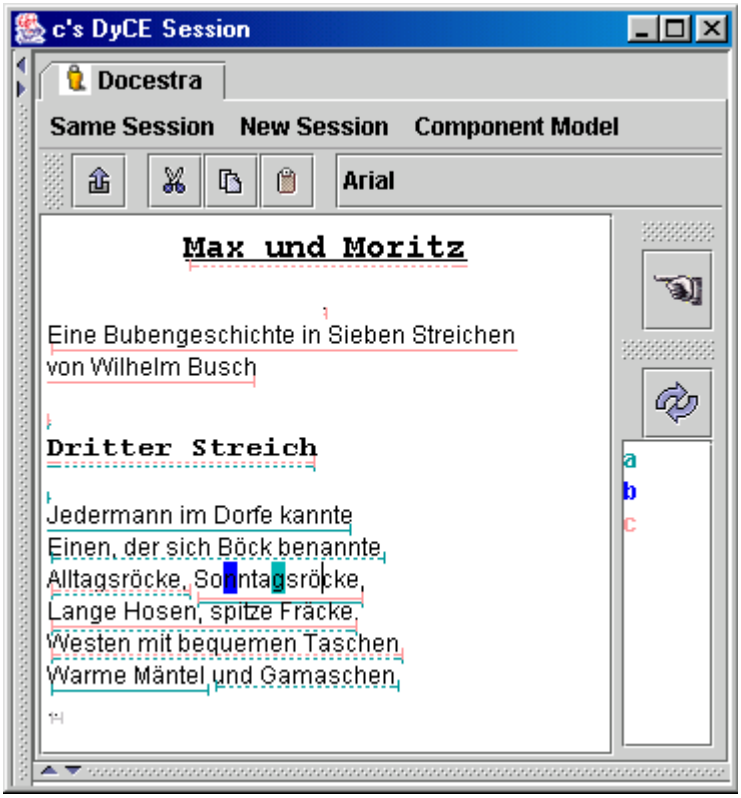


Abbildung 72: Anzeige des Konfliktpotenzials. (Screenshot vom Prototyp)
Benutzer a und c überarbeiten ein Kapitel. Benutzer b hat seinen Cursor ebenfalls in diesen Bereich bewegt. Die Bearbeitungsstellen sind durch durchgezogene Linien, Reservierungsvermerke durch geschrichelte Linien gekennzeichnet. Am Wort „Sonntagsröcke“ ist die Konfliktgefahr am höchsten, da sowohl a als auch c hier Modifikationen vorgenommen haben.

Durch diese Benutzungsschnittstelle werden die Benutzer in einer Weise auf Konflikte hingewiesen, die dem Betrachter schnell zugänglich ist und ihn nicht in seinem Arbeitsfluß einschränkt. Es ist sehr wahrscheinlich, daß der Benutzer Hinweise, die ihn betreffen, wahrnehmen wird, da sie durch optische Merkmale in dem Text, den er bearbeitet, gegeben werden. Er wird durch Hinweise die ihn

nicht betreffen, nicht abgelenkt. Schließlich steht es ihm frei, die Warnung zu ignorieren und einen Konflikt in Kauf zu nehmen.

6.6 Erfüllung der Anforderungsanalyse

Die in 2.5 beschriebenen Anforderungen werden in folgender Weise unterstützt:

Anforderung 1 (Untergliederung des Dokuments in Kapitel und andere logische Einheiten)

In Kapitel 6.2 wurde festgelegt, wie die Knoten eines Strukturgraphen zur Speicherung der logischen Informationen verwendet werden. Diese Informationen werden einerseits in der Benutzungsschnittstelle angezeigt, sie können aber auch in einfacher Weise zur Generierung von Inhaltsverzeichnissen genutzt werden.

Anforderung 2 (Formatierungen)

Die Möglichkeit der Akzentuierung des Textes durch Formatierungen wird durch die Anwendung gewährleistet. Die genaue Vorgehensweise wurde in Kapitel 6.1 beschrieben.

Anforderung 3 (XML-Export)

Der Export von Dokumenten nach XML wird durch den Ansatz von Beech, Malhotra und Rys stark vereinfacht. Der Ansatz und seine Anwendung durch Strukturgraphen wurde in Kapitel 4.2 vorgestellt. Auf die Speicherung des anwendungsspezifischen Inhaltsobjektes wurde in Kapitel 6.1 eingegangen.

Anforderung 4 (Parallele Modifikation)

Die unbeeinflußte, parallele Modifikation verschiedener Datensegmente wird, wie in Kapitel 2.7.3 beschrieben, durch DyCE sichergestellt, wenn die Implementierung der Speicherung von Knoten und Kanten des Strukturgraphen, wie in Kapitel 5.3 erläutert, gestaltet wird. Dabei wird vorausgesetzt, daß Inhaltsteile, die in enger räumlicher Beziehung stehen, in die gleichen Datensegmente eingeteilt werden. Die Anwendung der räumlichen Fragmentierung, wie in 3.2 beschrieben, erlaubt zusätzlich eine dynamische Anpassung der Segmentgröße an die aktuelle Situation.

Anforderung 5 (Unterschiedliche temporale Korrelation)

Die hier geforderten Transitionen und Mischformen aus asynchroner und synchroner Kooperation werden durch DyCE bereits vollständig unterstützt.

Anforderung 6 (Mehrere Ansichten)

Da DyCE mehrere Präsentationskomponenten unterstützt, konnte, wie in Kapitel 6.5 beschrieben, auf einfache Weise eine Gliederungsansicht erstellt werden, die zur Navigation in der Dokumentenansicht verwendet werden kann.

Anforderung 7 (Schnelle Anzeige fremder Modifikationen)

Diese Anforderung wird durch DyCE erfüllt.

Anforderung 8 (Telepointer)

Zur Unterstützung dieser Anforderung ist der in DyCE enthaltene Telepointer ausreichend.

Anforderung 9 (Gruppenbewußtsein)

DyCE zeigt eine Namensliste an, in der die an einer Sitzung teilnehmenden Benutzer aufgeführt werden. Dies trägt zur Bildung des Gruppenbewußtseins nur in schwachem Maße bei, da die Gruppenmitglieder zwar wissen, wer an der Sitzung teilnimmt, aber keine Informationen über deren Arbeit innerhalb der Sitzung erhalten.

Die Anwendung unterstützt die Förderung des Gruppenbewußtseins, indem Textteile, die gerade bearbeitet wurden, farblich hervorgehoben werden, wie in 6.5 ausgeführt. Weiterhin können durch ein Kontextmenü Informationen über die letzte Bearbeitung abgerufen werden. Die Informationen hierzu werden, wie in Kapitel 3.6 ausgeführt, extrahiert.

Anforderung 10 (Vermeidung von Konflikten)

Die Vermeidung von Konflikten erfolgt durch die Adaption der Segmentgröße. Dieses Verfahren wurde in Kapitel 3.2 beschrieben. Zusammen mit der in 6.1 ausgeführten Split-Transformation verbessert dies die momentanige Situation in DyCE.

Die Erforschung der tatsächlichen Zufriedenheit der Benutzer würde den Rahmen dieser Diplomarbeit übersteigen.

Anforderung 11 (Hinweis auf unvermeidbare Konflikte)

Die Informationen über das Konfliktpotenzial ergeben sich aus der in 3.3 besprochenen Reservierungsfunktion. Ihre Verwendung als Konfliktfrühwarnanzeige wurde in 3.5 diskutiert.

Die Anzeige des Potenzials wurde in Kapitel 6.5 beschrieben.

6.7 Prototyp

Viele Ansätze dieser Diplomarbeit wurden in einem Prototyp namens „DOCestra“ umgesetzt. Dazu gehören insbesondere die Umsetzung der Datenstruktur Strukturgraph, zusammen mit der in 5.5 beschriebenen Kombination aus Fragmentierungs- und Reservierungsverfahren. Desweiteren wurden die Reservierungsheuristiken „Simple“ und „Nachbarschaftsheuristik“ aus Kapitel 3.4 implementiert.

Leider hat der Prototyp nicht die nötige Reife erhalten, als daß hiermit ein sinnvoller Feldversuch zur Erforschung der Benutzerakzeptanz möglich wäre.

Die während der Implementierung aufgetretenen Schwierigkeiten, sind nicht durch den in dieser Arbeit vorgestellten Ansatz bedingt, sondern resultieren vielmehr aus der Verwaltung der kommutativen Transaktionen in DyCE. Hierzu verwendet DyCE den unter 2.7.4 vorgestellten undo-redo-Mechanismus, der zu einem Aufstau von Transaktionen, z.B. zur Anpassung der Reservierungsfunktion, führt.

Der Prototyp benötigt weiterhin einige Überarbeitung in Hinblick auf Geschwindigkeit und optimale Abstimmung der Transaktionen.

Weiterhin fehlt ein Mechanismus, der weitere Modifikationen eines Knotens während eines Splits verhindert und dafür sorgt, daß die zerteilende Transaktion in jedem Fall ausgeführt wird.

6.8 Zusammenfassung Kooperative Textverarbeitung

In diesem Kapitel wurde beschrieben, wie Strukturgraphen in DyCE zur Implementierung einer kooperativen Textverarbeitung verwendet werden können.

Es wurde eine Benutzungsschnittstelle vorgestellt, die neben der Anzeige des Dokuments und seiner Struktur auch Informationen über potenzielle Konflikte und die Arbeit der anderen Sitzungsteilnehmer darstellt.

Probleme, die bei der Implementierung des Prototyps auftraten wurden diskutiert.

Abschließend wurde wiederholt, wie die in Kapitel 2.5 aufgestellten Anforderungen von der kooperativen Textverarbeitung erfüllt wurden.

7 Zusammenfassung und Ausblick

7.1 Zusammenfassung

Im Rahmen dieser Diplomarbeit wurden Mechanismen entwickelt, mittels derer eine Komponente zur kooperativen Verarbeitung strukturierter Textdokumente im Groupwareframework DyCE realisiert wurde.

Diese Mechanismen umfassen zunächst zwei Methoden zur Verbesserung der Nebenläufigkeitskontrolle von DyCE. Das sind Fragmentierungsverfahren, die Konflikteinheiten bei Bedarf zeitweilig verfeinern, und Reservierungsautomatismen, mittels derer entweder Modifikationssperren errichtet oder Informationen über bevorstehende Konflikte gewonnen werden können. Für die zweite Methode wurden einige Heuristiken angegeben, die versuchen, aus dem aktuellen Benutzerverhalten auf deren Aktivitäten in naher Zukunft zu schließen.

Diese Verfahren setzen eine Datenstruktur voraus, die eine Aufteilung der zu verwaltenden Daten nach logischen und räumlichen Zusammenhängen erlaubt. Zu einer gewählten Partitionierung müssen Segmente unabhängig von anderen Teilen verfeinerbar sein, um keine unnötigen Konflikte zu verursachen.

Diese Eigenschaften werden von den, in dieser Arbeit entwickelten, Strukturgraphen erfüllt. Darüber hinaus eignen sich Strukturgraphen zur Abbildung von Baum- und Graphstrukturen, wie etwa die logische Gliederung eines Dokuments in Kapitel und Unterkapitel.

Durch die hierarchische Gliederung von Strukturgraphen stellte sich die Frage nach der Kompatibilität zu XML. Dazu wurde mithilfe des Ansatzes von Beech, Malhotra und Rys eine Anbindung skizziert.

Schließlich wurde, als Anwendung von Strukturgraphen, die kooperative Textverarbeitung vorgestellt. Diese wurde, wie die aus der Literatur bekannten Ansätze GROVE und ShrEdit und der mit Standardmechanismen von DyCE entwickelte Texteditor Notepad, mit einem Anforderungskatalog verglichen, der zu Beginn der Arbeit aus einem Szenario gewonnenen wurde.

7.2 Offene Fragen

Die Qualität einer Reservierungsheuristik ist als hoch einzustufen, wenn die zu den aktuellen Bearbeitungsstellen reservierten Segmente mit hoher Wahrscheinlichkeit in naher Zukunft von den jeweiligen Teilnehmersystemen modifiziert werden. Die vorgestellten Reservierungsheuristiken sind nun auf ihre Qualität als Konfliktfrühwarnsystem zu prüfen und gegebenenfalls bessere Heuristiken zu kreieren

Es ist notwendig, die Benutzerakzeptanz bezüglich der in dieser Diplomarbeit vorgestellten kooperativen Textverarbeitung zu evaluieren. Besonders nützlich

wäre die Bestimmung der Effektivität der Mittel, die zur Förderung des Gruppenbewußtsein vorgeschlagen wurden. Es müßte überprüft werden, ob allein die Warnung vor Konflikten deren Entstehung verhindert oder ob das Einrichten von Sperrmechanismen unerläßlich ist.

Mechanismen zur benutzerseitigen Zurücknahme von Änderungen („undo-Funktion“) wurden in dieser Arbeit außen vor gelassen. In einer weiterführenden Arbeit könnte überprüft werden, ob die gängigen Ansätze aus der CSCW-Forschung (z.B. von Alan Dix) auf die in dieser Arbeit vorgestellte kooperative Textverarbeitung anwendbar sind bzw. ob die Implementierung eines „Papierkorbes“, in dem gelöschte Textteile zunächst aufbewahrt werden, ausreicht.

In dem verwendeten Framework DyCE ist der Mechanismus zur Auflösung von kommutativen Transaktionen zu überdenken. Zur Zeit werden kommutative Transaktionen auf gleichen Konflikteinheiten in geeigneter Weise wiederholt. Dies ist eine Anwendung des undo-redo Mechanismus und führt zu den beschriebenen Problemen, wie quadratische Laufzeit und Aufstau von Transaktionen.

Weiterhin müßte ein Verfahren gefunden werden, mit dessen Hilfe die Transaktion, die die Splitoperation ausführt, Vorrang gegenüber anderen Transaktionen erhält. Andernfalls können Konflikte auftreten, die die zurückgenommene Fragmentierung verhindert hätte.

Aus der Ähnlichkeit zwischen Strukturgraphen und XML-Graphen kann vermutet werden, auf welche Probleme sich Strukturgraphen anwenden lassen. Z.B. könnte man den Ansatz einer kooperativen Textverarbeitung leicht auf ein System zur kooperativen Generierung von Hypermedia erweitern. Andere Anwendungen wären die kooperative Bearbeitung von CAD-Diagrammen (Computer Aided Design), wie sie in dem Ingenieurswesen häufig verwendet werden, kooperative Bearbeitung von Rastergrafiken oder kooperative UML-Editoren.

Eine weiterführende Aufgabe wäre es, Kriterien zu finden, durch die sich die Klasse der auf Strukturgraphen anwendbare Probleme eindeutig abgrenzen lassen.

Anhang A: Grundbegriffe der Mengen- und Graphentheorie

Die in Kapitel 4.1 verwendeten Grundbegriffe der Graphentheorie wurden größtenteils [Grö01] entnommen. Der Vollständigkeit halber sind die benötigten Abschnitte hier zitiert.

> „[...] Ein Graph G ist ein Tripel (V,E, Ψ) bestehend aus einer nicht-leeren Menge V, einer Menge E und einer Inzidenzfunktion Ψ : E -> V^2. Hierbei bezeichnet V^2 die Menge der ungeordneten Paare von (nicht notwendigerweise verschiedenen) Elementen von V. Ein Element von V heißt Knoten (oder Ecke oder Punkt oder Knotenpunkt; englisch: vertex oder node oder point), ein Element aus E heißt Kante (englisch: edge oder line). Zu jeder Kante e $\in$ E gibt es also Knoten u, v $\in$ V mit Ψ (e)) uv = vu. [...]
>
> Die Anzahl der Knoten eines Graphen heißt natürliche Ordnung des Graphen. [...]
>
> Gilt Ψ (e) = uv für eine Kante e $\in$ E, dann heißen die Knoten u,v $\in$ V Endknoten von e, und wir sagen, daß u und v mit e indizieren oder auf e liegen, daß e die Knoten u und v verbindet, und daß u und v Nachbarn bzw. adjazent sind. Wir sagen auch, daß zwei Kanten indizent sind, wenn sie gemeinsame Endknoten haben.[...]
>
> Die Benutzung der Inzidenzfunktion Ψ führt zu einem relativ aufwendigen Formalismus. Wir wollen daher die Notation etwas vereinfachen. Dabei entstehen zwar im Falle von nicht-einfachen Graphen gelegentlich Mehrdeutigkeiten, die aber i.a. auf offensichtliche Weise interpretiert werden können. Statt Ψ (e) = uv schreiben wir von nun an einfach e = uv (oder äquivalent: e = vu) und meinen damit die Kante e mit den Endknoten u und v. Das ist korrekt, solange es nur eine Kante zwischen u und v gibt. Gibt es mehrere Kanten mit den Endknoten u und v, und sprechen wir von der Kante uv, so soll das heißen, daß wir einfach eine der parallelen Kanten auswählen. Von jetzt an vergessen wir also die Inzidenzfunktion Ψ und benutzen die Abkürzung G = (V,E), um einen Graphen zu bezeichnen. Manchmal schreiben wir auch E_G oder E (G) bzw. V_G oder V (G) zur Bezeichnung der Kanten- bzw. Knotenmenge eines Graphen G.[...]
>
> Sind W eine Knotenmenge und F eine Knotenmenge in G = (V,E), dann bezeichnen wir mit E(W) die Menge aller Kanten von G mit beiden Endknoten in W und mit V (F) die Menge aller Knoten, die Endknoten mindestens einer Kante aus F sind.[...]
>
> Sind G = (V,E) und H = (W,F) Graphen, so daß W $\subseteq$ V und F $\subseteq$ E gilt, so heißt H Untergraph (oder Teilgraph) von G. [...] Ein Untergraph H = (W,F) von G = (V,E) heißt aufspannend, falls V = W gilt. [...]
>
> Digraphen:
>
> Die Kanten eines Graphen haben keine Orientierung. In vielen Anwendungen spielen aber Richtungen eine Rolle. Zur Modellierung solcher Probleme führen wir gerichtete Graphen ein. Ein Digraph (oder gerichteter Graph) D = (V,A) besteht aus einer

(endlichen) nicht-leeren Knotenmenge V und einer (endlichen) Menge A von Bögen (oder gerichteten Kanten; englisch: arc). Ein Bogen a ist ein geordnetes Paar von Knoten, also a = (u, v), u ist der Anfangs- oder Startknoten, v der End- bzw. Zielknoten von a; u heißt Vorgänger von v, v Nachfolger von u, a indiziert mit u und v. [...]

Falls D = (V,A) ein Digraph ist und $W \subseteq V$, $B \subseteq A$, dann bezeichnen wir mit A(W) die Menge de Bögen, deren Anfangs- und Endknoten in W liegen und mit V(B) die Menge der Knoten, die als Anfangs- oder Endknoten mindestens eines Bogens in B auftreten. Unterdigraphen, induzierte Unterdigraphen, aufspannende Unterdigraphen, Vereinigung und Durchschnitt von Digraphen, das Entfernen von Bogen und Knotenmengen und die Kontraktion von Bogen- oder Knotenmengen sind genau wie bei Graphen definiert.

Ist D = (V,A) ein Digraph, dann heißt der Graph G = (V,E), der für jeden Bogen $(i,j) \in A$ eine Kante ij enthält, der D unterliegende Graph.[...]

Der Außengrad (Innengrad) von v ist die Anzahl der Bögen mit Anfangsknoten (Endknoten) v. Die Summe von Außengrad und Innengrad ist die Grad von v. [...]

Ketten, Wege, Kreise, Bäume:

In einem Graphen oder Digraphen heißt eine endliche Folge $W = (v_0, e_1, v_1, e_2, v_2,..., e_k, v_k)$, $k \geq 0$, die mit einem Knoten beginnt und endet und in der Knoten und Kanten (Bögen) alternierend auftreten, so daß jede Kante (jeder Bogen) e_i mit den beiden Knoten v_{i-1} und v_i indiziert, eine Kette. Der Knoten v_0 heißt Anfangsknoten, v_k Endknoten der Kette; die Knoten $v_1,..., v_{k-1}$ heißen innere Knoten. W wird auch $[v_0, v_k]$-Kette genannt. Die Zahl k heißt Länge der Kette (= Anzahl der Kanten bzw. Bögen in W). Falls (in einem Digraphen) alle Bögen e_i der Kette W der Form (v_{i-1}, v_i) (also gleichgerichtet) sind, so nennt man W gerichtete Kette bzw (v_0, v_k)-Kette. Ist $W = (v_0, e_1, v_1,..., e_k, v_k)$ eine Kette und sind i,j Indizes mit $0 \geq i \geq j \geq k$, dann heißt die Kette $(v_i, e_{i+1}, v_{i+1},...,e_j, v_j)$ das $[v_i, v_j]$-Segment (bzw. (v_i, v_j)-Segment, wenn W gerichtet ist) von W. Jede (gerichtete) Kante, die zwei Knoten der Kette W miteinander verbindet, die aber nicht Element von W ist, heißt Diagonale (oder Sehne) von W. [...]

Eine Kette, in der alle Knoten voneinander verschieden sind, heißt Weg. Eine Kette, in der alle Kanten und Bögen verschieden sind, heißt Pfad. Ein Weg ist also ein Pfad, aber nicht jeder Pfad ist ein Weg. Ein Weg oder Pfad in einem Digraphen, der eine gerichtete Kette ist, heißt gerichteter Weg oder gerichteter Pfad. Wie bei Ketten sprechen wir von [u, v]- Wegen, (u, v)-Wegen etc. [...]

Eine Kette heißt geschlossen, falls ihre Länge nicht Null ist und falls ihr Anfangsknoten mit ihrem Endknoten übereinstimmt. Eine geschlossene (gerichtete) Kette, in der der Anfangsknoten und alle inneren Knoten voneinander verschieden sind, heißt Kreis (gerichteter Kreis). Offensichtlich enthält jede geschlossene Kette einen Kreis. [...]

Ein Wald ist ein Graph, der keinen Kreis enthält. Ein zusammenhängender Wald heißt Baum. Ein Baum in einem Graphen

heißt aufspannend, wenn er alle Knoten der Graphen enthält. Ein Branching B ist ein Digraph, der ein Wald ist, so daß jeder Knoten aus B Zielknoten von höchstens einem Bogen von B ist. Ein zusammenhängendes Branching heißt Arboreszenz. Eine aufspannende Arboreszenz ist eine Arboreszenz in einem Digraphen D, die alle Knoten von D enthält. Eine Arboreszenz enthält einen besonderen Knoten, genannt Wurzel, von dem aus jeder andere Knoten auf genau einem gerichteten Weg erreicht werden kann. Arboreszenzen werden auch Wurzelbäume genannt. Ein Digraph, der keinen gerichteten Kreis enthält, heißt azyklisch.

Ein Graph heißt zusammenhängend, falls es zu jedem Paar von Knoten s,t einen [s,t]-Weg in G gibt. [...]“

[KS91] definieren Relation und Ordnungsbegriff wie folgt:

„(1.12) DEFINITION: (1) Es seien M und N Mengen; es sei $K \subset M \times N$. Dann heißt K eine Korrespondenz zwischen M und N.
(2) Es sei M eine Menge, es sei $R \subset M \times M$ eine Korrespondenz. Dann heißt R eine Relation auf M. Man schreibt statt $(x, y) \in R$ auch x R y. [...]

(1.14) DEFINITION: Es sei M eine Menge, es sei R eine Relation auf M,
(1) R heißt reflexiv, wenn für jedes $x \in M$ gilt: Es ist x R x. [...]
(3) R heißt antisymmetrisch, wenn für alle $x, y \in M$ mit x R y und mit y R x gilt: Es ist x = y.
(4) R heißt transitiv, wenn für alle $x, y, z \in M$ mit x R y und mit y R z gilt: Es ist x R z.
(5) R heißt alternativ, wenn für alle $x, y \in M$ gilt: Es ist x R y oder y R x.

(1.15) DEFINITION: Es sei M eine Menge, es sei R eine Relation auf M. [...]
(2) R heißt (teilweise) Ordnung, wenn R reflexiv, antisymmetrisch und transitiv ist.
(3) R heißt lineare Ordnung, wenn R alternativ und eine Ordnung ist.“

Der verwendete Begriff des geordneten Baumes ist [App95] entnommen:

„1. Ein gerichteter Graph $G = (V_G, E_G)$ heißt Baum, wenn er keine Zyklen enthält und wenn es einen Knoten $r \in V_G$ gibt, so daß für alle $v \in V_G$ genau ein Weg von r nach v existiert. r heißt die Wurzel von G

2. Ein Graph heißt Wald, wenn seine Komponenten Bäume sind. Ein geordnetes m-Tupel vom Bäumen $(T_1, T_2, ..., T_m)$ heißt geordneter Wald.

3. Jeder Baum T läßt sich in der Form $(r; T_1, ..., T_m)$ schreiben, wobei r die Wurzel von T ist und $T_1, ..., T_m$ die an r ‚hängenden‘ Teilbäume von T sind. $T = (r; T_1, ..., T_m)$ heißt geordnet, wenn eine Anordnung $T_1 < T_2 <... < T_m$ vorgegeben ist, und wenn für $i \in \{1, ..., m\}$ die T_i selbst geordnet sind.“

Anhang B: Verwendete UML Notation

Dieser Anhang gibt einen kurzen Einblick in die verwendete UML (Unified Modeling Language) Syntax. Für die genaue Spezifikation, siehe [OMG].

Die folgenden Zitate stammen aus [Dum].

Klassen

„Klassen werden durch Rechtecke dargestellt, die den Namen der Klasse und/oder die Attribute und Operationen der Klasse enthalten. Klassenname, Attribute und Operationen werden durch eine horizontale Linie getrennt. Der Klassenname steht im Singular und beginnt mit einem Großbuchstaben. Attribute können näher beschrieben werden, z.B. durch ihren Typ, einen Initialwert und Zusicherungen. Sie werden aber mindestens mit ihrem Namen aufgeführt. Operationen können ebenfalls durch Parameter, Initialwerte, Zusicherungen usw. beschrieben werden. Auch Sie werden mindestens mit ihrem Namen aufgeführt."

„Attribute sind Informationen bzw. Daten die ein Element einer Klasse näher beschreiben. Sie werden mindestens durch ihren Namen beschrieben, [...]"

„Eine Nachricht besteht aus dem Selektor (ein Name) und einer Liste von Parametern. Sie wird an genau einen Empfänger gesendet. Eine Operation setzt sich zusammen aus dem Namen der Operation, den Parametern (falls vorhanden) und einem evt. Rückgabewert. Die Parameter einer Operation entsprechen in ihrer Definition den Attributen. Eine Operation ist innerhalb einer Klassendefinition eindeutig identifizierbar. [...] Der Name einer Operation beginnt mit einem Kleinbuchstaben. Der Name des Argumentes beginnt ebenfalls mit einem Kleinbuchstaben. Das Argument wird durch Nennung seines Typs näher beschrieben, außerdem kann ein Initialwert angegeben werden. Argumentname und Argumenttyp werden durch einen Doppelpunkt getrennt. [...] Ein abstrakte Operation kann man [...] kursiv schreiben. [...] Klassenoperationen werden durch Unterstreichung gekennzeichent und die äußere Sichtbarkeit von Operationen durch voranstellen des Sichtbarkeitskennzeichen.

klassenoperation()

+publicOperation()

#protectedOperation()

-privatOperation()

Innerhalb des Klassenrechteckes werden Operationen im unteren Teil aufgeführt."

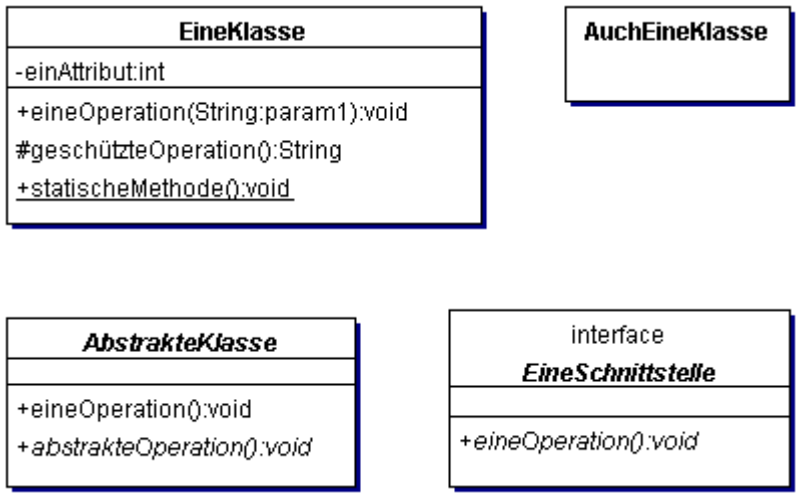


Abbildung 73: Darstellungen von Klassen in UML.

Vererbung

„Die Vererbung wird mit einem nicht ausgefüllten Pfeil, der von der Unterklasse zur Oberklasse zeigt dargestellt."

„Die Nutzung einer Schnittstelle durch andere Klassen wird durch eine Abhängigkeitsbeziehung (gestrichelter Pfeil) notiert."

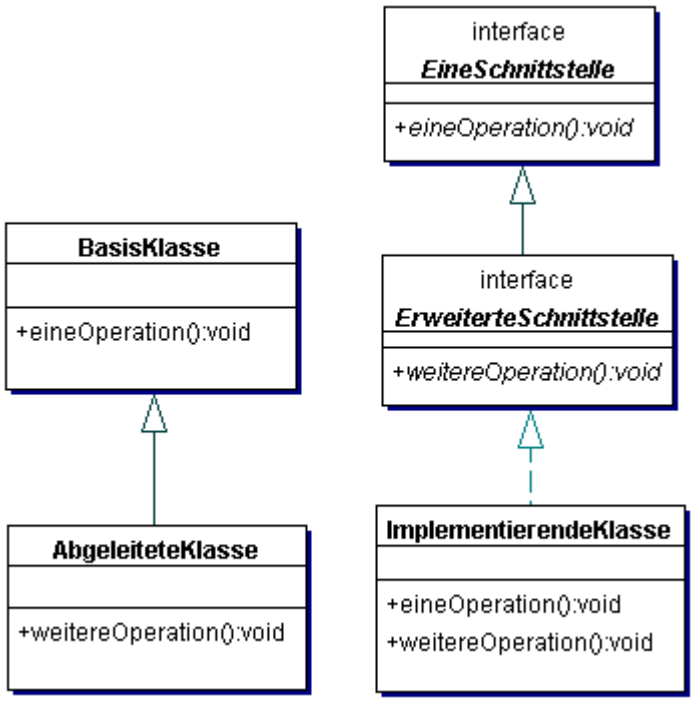


Abbildung 74: Darstellungen von Vererbung und Implementierung in UML.

Aggregation und Komposition

„Assoziationen werden durch eine Linie zwischen den beteiligten Klassen dargestellt. Die Linie wird mit einem Namen versehen [...], der beschreibt, worin und warum diese Beziehung besteht. [...] An den Enden der Verbindungslinie kann die Multiplizität der Beziehung angegeben werden. Sie wird als einzelne Zahl oder als Wertebereich auf jeder Seite der Assoziation notiert. Der Wertebereich wird wie folgt notiert: Angabe des Minimums und des Maximums, getrennt durch zwei Punkte. Mit einem * wird der Joker beschrieben, also "viele". Mit einem Komma können unterschiedliche Möglichkeiten der Multiplizität aufgezählt werden. [...] Es existiert auch eine Form in der die Assoziation selbst über Attribute verfügt. Diese Assoziationsattribute sind dann existenzabhängig von der Assoziation.“

„Die Aggregation wird als Linie zwischen zwei Klassen dargestellt, und zusätzlich mit einer Raute versehen. Die Raute steht auf der Seite des Aggregats (des Ganzen) und symbolisiert das Behälterobjekt, in dem die Teile gesammelt sind. Die Kardinalitätsangabe auf der Seite des Aggregats ist häufig 1, so das ein Fehlen der Angabe standardmäßig als 1 interpretiert wird. [...]

Die Komposition wird als Linie zwischen zwei Klassen gezeichnet. Die Linie wird auf der Seite des Aggregates mit einer ausgefüllten Raute versehen. Kompositionsbeziehungen können mit einer Multiplizitätsangabe, mit einem Beziehungsnamen und mit Rollennamen notiert werden. Mehrere Kompositionsbeziehungen zu einem Ganzen können baumartig zusammengefaßt werden.“

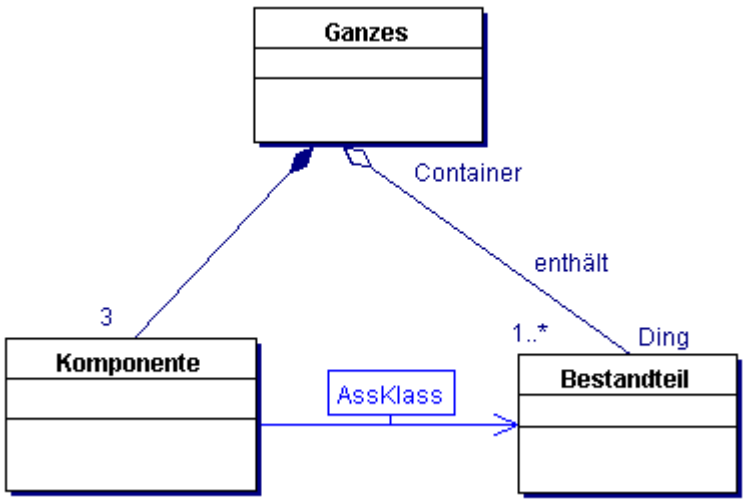


Abbildung 75: Darstellung von Komposition, Aggregation und Assoziation im UML-Klassendiagramm.

Objektdiagramme

“Objekte werden durch Rechtecke visualisiert. Diese beeinhalten den Namen des Objektes und eventuell zusätzlich den Na-

men der Klasse des Objektes.“

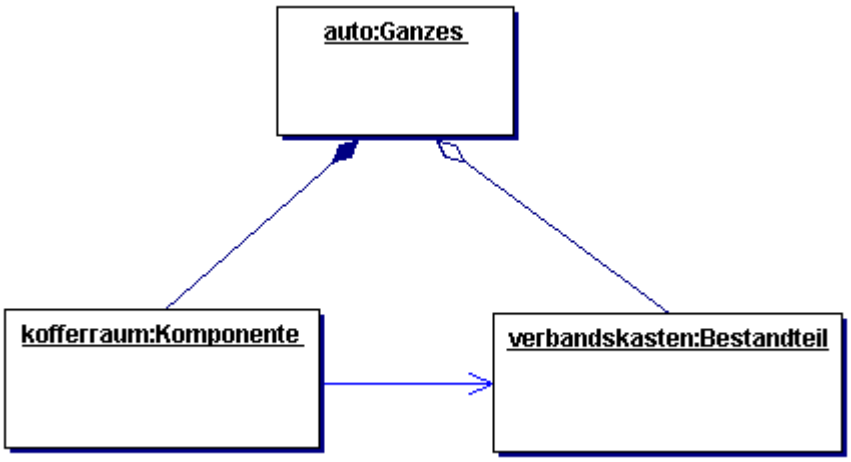


Abbildung 76: Darstellung von Komposition, Aggregration und Assoziation im UML-Objektdiagramm.

Aktivitätsdiagramme

„In einem Programmablauf durchläuft ein Modellelement eine Vielzahl von Aktivitäten, d.h. Zuständen, die eine interne Aktion und mindestens eine daraus resultierende Transition enthalten. Die ausgehende Transition impliziert den Abschluss der Aktion und den Übergang des Modellelementes in einen neuen Zustand bzw. eine neue Aktivität. Diese Aktivitäten können in ein Zustandsdiagramm integriert werden oder besser aber in einem eigenen Aktivitätsdiagramm visualisert werden. Ein Aktivitätsdiagramm ähnelt in gewisser Weise einem prozeduralem Flußdiagramm, jedoch sind alle Aktivitäten eindeutig Objekten zugeordnet,d.h. sie sind entweder einer Klasse, einer Operation oder einem Anwendungsfall eindeutig untergeordnet. [...] Eine Aktivität wird durch ein ‚Rechteck‘ mit konvex abgerundeten Seiten visualisiert. Sie enthält eine Beschreibung der internen Aktion. Von der Aktivität aus gehen die Transitionen, die den Abschluss der internen Aktion und den Übergang zur nächsten Aktivität darstellen.“

„Start- und Endzustand eines Objektes sind als besondere Zustandstypen anzusehen, da zu einem Startzustand kein Übergang stattfinden und dem Endzustand eines Objektes keine Zustandsänderung folgen kann. [...] Ein Zustand kann Bedingungen mit dem Ereignis verbinden, die erfüllt sein müssen, um den den Folgezustand zu erreichen, bzw. um zu entscheiden, welchen Folgezustand das Objekt einnimmt. [...] Ein Startzustand wird als gefüllter Kreis visualisiert, während man einen Endzustand als nicht gefüllten Kreis, in dem sich ein kleinerer voller Kreis befindet, darstellt. [...] Ein Zustandübergang als Folge eines Ereignisses wird als Pfeil zwischen zwei Zuständen symbolisiert. Der Pfeil kann auch zum Ausgangszustand zurückführen.“

Gehen mehrere Transitionen aus einer Aktivität aus, so müssen alle bis auf eine mit einem in eckigen Klammen angegebenen Boole'schen Ausdruck versehen werden, der nur für eine der Transition wahr ist. Die unbeschriftete Transition wird dann ausgeführt, wenn keiner der angegebenen Ausdrücke erfüllt ist.

Ereignisse sind eine Spezialform von Aktivitäten. Sie werden durch Rechtecke mit ausgeschnittener Splitze symbolisiert.

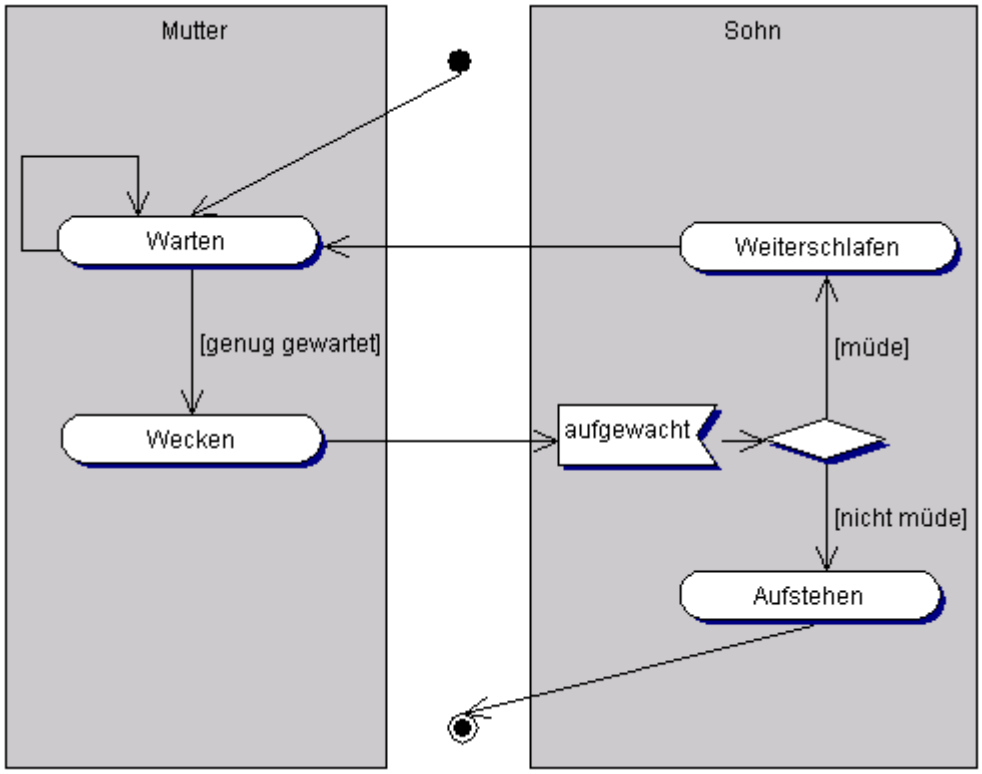


Abbildung 77: Aktivitätsdiagramme in UML.

Glossar

ACID-Eigenschaften	Atomarität, Konsistenz, Isolation und Dauerhaftigkeit. Eigenschaften, die jede Transaktion erfüllen muß.
Aktualisierende Reservierung	Paßt die Reservierungsfunktion nach jeder Modifikation der Datenstruktur an.
Application Sharing	Verteilen der Ansicht einer herkömmlichen Ein-Benutzer-Anwendung an verschiedene Teilnehmersysteme und übertragen von Maus- und Tastaturereignissen auf dem umgekehrten Weg. Dabei kann nur ein Benutzer zur gleichen Zeit Daten modifizieren.
Artefakt	Gegenstand oder Arbeitsprodukt einer Sitzung.
asynchrone Kooperation	Alle Benutzer sind zu verschiedenen Zeiten am Arbeitsvorgang beteiligt.
Attributkante	Kante, die auf ein Attribut verweist.
Attributkind	Knoten, der ein Attribut seines Vaterknotens (gemäß der Primärstruktur) darstellt und daher mit ihm über eine Attributkante verbunden ist.
bearbeitet ein ModelObject	Ein Teilnehmersystem bearbeitet ein ModelObject, wenn es das ModelObject vor Kurzem modifiziert hat.
Bearbeitungsstelle	Knoten, der vor Kurzem modifiziert wurde.
bearbeitungsstelleVon	Funktion von der Menge der Knoten in die Menge der Teilnehmersysteme, die den Knoten vor Kurzem modifiziert haben.
Benutzungsschnittstelle	Teil einer Anwendung, durch die der Benutzer die Daten einsieht und manipuliert.
Client	Vernetzter Computer, an dem ein Benutzer arbeitet
content	Funktion, die jedem Knoten seinen Inhalt zuordnet.
CSCW	Arbeitsgebiet der rechnergestützten Gruppenarbeit. Abkürzung für Computer Supported Cooperative Work.
Data Sharing	Verteilung der modifizierten Daten im Gegensatz zu darstellungsspezifischen GUI Informationen. Zusätzlich wird meist ein spezielles Verfahren zur Nebenläufigkeitskontrolle verwendet.
Datenverteilung	Verfahren mittels derer Manipulationen an den Daten an andere Teilnehmersysteme übertragen werden. Beispiele: Application Sharing, GUI Event Sharing,

	Data Sharing.
direkt enthalten	Der Inhalt c ist direkt in einem Knoten v enthalten, wenn content(v) = c.
DyCE	Ein Groupware-Framework. Abkürzung für Dynamic Collaboration Environment.
DyCE-Komponente	Anwendung für DyCE.
DynamicVertex	Spezialisierter Knoten eines Strukturgraphen, dessen Inhalt partitionierbar ist und damit eine adaptive Konfliktgranularität modelliert.
„enthält“-Relation	Ein Knoten v ist in einem Knoten u enthalten, wenn der Inhalt von v ebenfalls ein Teil von u ist. Die „enthält“-Relation ist transitiv. (Sie wird auch „enthält“-Semantik genannt.)
einfache Reservierung	Reserviert nur das ModelObject des Grundes.
Einflußheuristik	Reservierung neben dem ModelObject des Grundes weitere ModelObjects, auf die ein gewisser Einfluß vorhanden ist.
Element beinhaltende Kanten	Kante im XML-Graphen, die von einem XML-Element auf ein darunter geschachteltes Element zeigt.
Enthältkante	Kante im Strukturgraphen die der „enthält“-Relation entspricht.
Ereignis	Mit einem Ereignis kann das Betriebssystem oder andere Objekte eine Anwendung über das Eintreten eines bestimmten Zustands informieren. Beispiele: Tastaturereignis, wenn eine Taste gedrückt wurde, oder Mausereignis, wenn die Maus verschoben wurde.
Exploration	Erste Phase des Prozeßmodells SECAI, indem jedes Teilnehmersystem die komplette Aufgabe zunächst für sich erledigt.
FinestDynamicVertex	Spezialform eines DynamicVertex‘ der die feinste Konfliktgranularität modelliert.
Fragmentierungs-verfahren	Bestimmt, wann eine Fragmentierung ausgeführt wird. Beispiele: Vorbeugende, optimistische oder verzögerte Fragmentierung.
gegenläufige Modifikationen	Aktionen, die auf der gleichen Datenbasis inkompatible Änderungen bewirken würden.
geordneter Struktur-graph	Ein Strukturgraph der eine Kantenordnung besitzt.
Groupware	Software zur Unterstützung von rechnergestützer Gruppenarbeit.

Groupware-Frameworks	Software, die dem Entwickler grundlegende Aufgaben bei der Erstellung von Groupware abnimmt.
Grund	Grund für eine Reservierungsänderung. Beispiele: ein Knoten wird Bearbeitungsstelle oder es wird in einem Knoten navigiert.
Gruppe	mehrere Benutzer, die an einer Problemstellung gemeinsam arbeiten.
Gruppenbewußtsein	Wenn der Einzelne sich der Tätigkeiten und Anwesendheit der Anderen bewußt ist.
GUI	Benutzungsschnittstelle. Abkürzung für Graphical User Interface.
GUI Event Sharing	Publizieren aller Eingabeereignisse (von Tastatur und Maus) an alle Teilnehmersysteme.
Heuristikkombination für Strukturgraphen	Kombination aus Fragmentierungs- und Reservierungsverfahren, die sich speziell für Strukturgraphen eignet.
Index	Gibt an, an welcher Stelle der Kantenordnung eine neue Kante eingefügt werden soll.
Inhalt	Mit einem Knoten assoziiertes Datensegment.
inhaltspartioniert	Ein Strukturgraph heißt inhaltspartitioniert, wenn kein Nichtterminalknoten direkten Inhalt besitzt.
iterativen Verbessern	Arbeitsparadigma, bei dem alle Teilnehmersysteme nacheinander zur Änderung der gemeinsamen Daten berechtigt sind. Das Änderungsrecht geht so lange auf den nächsten Teilnehmer über, bis alle mit dem Ergebnis einverstanden sind. Beispiel: Laufmappenverfahren.
Just-In-Time-Reservierung	Modifiziert die Reservierungsfunktion unmittelbar bevor die Reservierungsinvariante verletzt werden würde.
Kante	Verbindung zwischen zwei Knoten im Strukturgraph, die eine Beziehung zwischen diesen widerspiegelt.
Kantenordnung	Ordnung zwischen aus dem gleichen Knoten ausgehenden Kanten.
Knoten	Einheit im Strukturgraph, die ein Datensegment repräsentiert und durch seine Kanten das Datensegment mit anderen Daten in Beziehung setzt..
Konfliktauflösung	Auflösung von Konflikten nach ihrer Entstehung. Beispiele: optimistische Transaktionsverfahren, Transformationsverfahren.
Konflikteinheit	Einheit, auf der gleichzeitige Modifikationen einen

	Konflikt auslösen. In DyCE: ModelObject, bzw. Slot. In Strukturgraphen: Knoten.
Konfliktfreiheitsgarantie	Garantiert unter bestimmten Bedingungen, daß keine Konflikte auftreten.
Konfliktgranularität	Gibt an, wie viele Konflikteinheiten eine bestimmte Datenmenge aufgeteilt ist. Bei wenigen spricht man von einer groben Granularität, bei vielen von einer feinen.
konfliktierende Transaktionen	Transaktionen konfliktieren, wenn sie zur selben Zeit Operationen auf der gleichen Konflikteinheit ausführen sollen.
Konfliktprävention	Verhinderung von Konflikten vor ihrer Entstehung. Beispiel: Sperrmechanismen, pessimistische Transaktionsverfahren.
konsistente Operation	Eine Operation auf einem Strukturgraphen heißt konsistent, wenn sie den Strukturgraphen in einer Weise modifiziert, daß er danach wiederum alle Strukturgrapheigenschaften erfüllt.
Konsolidierung	Zweite Phase des Prozeßmodells SECAI, bei der die Teilnehmer aus den Ergebnissen der Explorationsphase eine optimale Version zu bilden versuchen.
Kontextmenü	Funktionsmenü in der Benutzungsschnittstelle, daß zu bestimmten Daten (dem sogenannten Kontext) aufgerufen werden kann.
Kopplung der Sichten	Gibt an, in wie weit die Ansichten der Benutzer von einander abhängen. Beispiele: striktes oder loses WYSIWIS.
lastModified	Funktion, die den Zeitpunkt und das Teilnehmersystem bestimmt, an dem ein Knoten zuletzt bearbeitet wurde.
Laufmappenverfahren	Ist ein iteratives Verfahren, in dem Artefakte in einer speziellen Mappe herumgereicht werden. Auf der Mappe ist gekennzeichnet, wer wann die Mappe erhält und damit den Inhalt modifizieren darf.
LeafVertex	Spezialisierter Knoten eines Strukturgraphen, der einem Terminalknoten entspricht, der die Split-Operation nicht unterstützt.
loses WYSIWIS	Alle Benutzer sehen die gleiche Anwendung. Position von Scrollbalken, Textcursor und Mauszeiger können unterschiedlich sein.
mittelbar enthalten	Der Inhalt c ist mittelbar in einem Knoten v enthalten, wenn content(u) = c für einen Knoten u und u in v

	enthalten ist.
MobileComponent	Präsentationskomponente einer DyCE-Anwendung.
ModelObject	Datenmodellierung einer DyCE-Komponente. Entspricht einer Konflikteinheit.
Nachbarschafts-Reservierung	Reservierung, die neben dem ModelObject des Grundes die benachbarten ModelObjects ebenfalls reserviert.
Nebenläufigkeits-kontrolle	Sichert die Datenkonsistenz bei paralleler Datenmodifikation. Diese Verfahren unterteilen sich in Konfliktprävention und Konfliktauflösung.
ObjectID	Systemweit eindeutige Identifizierungsnummer eines ModelObjects.
Operation	Modifizierungsanweisung an ein Objekt.
Optimistische Fragmentierung	Führt eine Fragmentierung aus, sobald ein Konflikt aufgetreten ist.
optimistische Verfahren	Manipulation der lokalen Daten, bevor eine Bestätigung der Konfliktfreiheit vom Server vorliegt. Tritt ein Konflikt auf, muß die lokale Änderung wieder zurückgenommen werden.
Primärstruktur	Von der Menge der Enthältkanten induzierter Teilgraph des Strukturgraphen.
rechnergestützte Gruppenarbeit	Unterstützung der Zusammenarbeit von Personen, die sich nicht unbedingt am gleichen Ort befinden müssen, durch Computer. Auch CSCW genannt.
Referenzkante	Kante, über die ein Knoten auf einen anderen verweisen kann.
replizierten Architektur	Verteilungsstruktur, bei der Kopien der zu bearbeitenden Daten auf den jeweiligen Clients liegen und dort bearbeitet werden. Ein Abgleich der Daten erfolgt automatisch.
reserviert	Ein ModelObject ist reserviert, wenn die Reservierungsfunktion an dieser Stelle nicht gleich der leeren Menge oder „nichtreserviert“ ist.
reservierung	Funktion, die zu einem ModelObject die Menge der Teilnehmersysteme bestimmt, für die das ModelObject reserviert ist.
Reservierungsfunktion	Gemeint ist hiermit die Funktion reservierung.
Reservierungsheuristik	Versucht aus einer Bearbeitungsstelle (bzw. einem anderen Grund) abzuleiten, welche weiteren ModelObjects mit hoher Wahrscheinlichkeit in Kürze modifiziert werden.

Reservierungs-invariante	Gilt, wenn nur ModelObjects modifiziert werden dürfen, für die eine Reservierung des entsprechenden Teilnehmersystems vorhanden ist.
Reservierungs-verfahren	Bestimmt, wann die Reservierungsfunktion neu berechnet wird. Beispiele: vorbeugende Reservierung, Just-In-Time-Reservierung, aktualisierende Reservierung.
RObject	Hilfsobjekt, mittes dessen Dateninhalte und -änderungen eines ModelObjects über das Netzwerk übertragen werden.
RootVertex	Spezialisierter Knoten eines Strukturgraphen, der dem Wurzelknoten entspricht.
Schriftführerverfahren	Arbeitsparadigma, bei dem nur ein einzelnes Teilnehmersystem zur Änderung der gemeinsamen Daten berechtigt ist. Dieses Teinnehmersystem erhält Modifizierungs- oder Editierungsvorschläge der anderen Teilnehmer und entscheidet dann, welche es ausführt.
SECAI	Prozeßmodell, das insbesondere für Lernsituationen entwickelt wurde, die wissenschaftliche Artefakte behandeln. Abkürzung für Summarization, Evaluation, Comparison, Argumentation and Integration.
Server	Computer, der die Daten in einem Netzwerk mit zentraler Architektur verwaltet.
Servlet	Auf einem Webserver laufende Java Komponente, die dynamisch Webseiten generiert.
Sitzung	Arbeitsvorgang, an dem mehrere Benutzer teilnehmen.
Slots	Container für replizierbare Daten eines Model-Objects. Konflikteinheit.
Sperrmechanismen	Sicherung der Konsistenz durch Setzen eines Manipulationsschutzes. (Konfliktprävention).
Split-Operation	Operation zur Verfeinerung der Konfliktgranularität.
Stratifikation	Arbeitsparadigma, bei der die zu erledigende Arbeit strikt nach Fachgebieten getrennt wird, d.h. es findet eine Aufgabenteilung nach Spezialisierung statt (z.B. Autor, Korrektor, Layouter).
striktes WYSIWIS	Alle Benutzer sehen genau den gleichen Bildschirmausschnitt.
StructuringVertex	Spezialisierter Knoten eines Strukturgraphen zur Strukturierung der Daten.

Strukturgraph	Baum- bzw. graphähnliche Datenstruktur, die leicht zur Implementierung von kooperativen Anwendungen verwendet werden kann.
synchrone Kooperation	Alle Benutzer sind gleichzeitig in einer Sitzung anwesend.
Teilnehmersystem	Einheit aus Benutzer und Client. Genauer: Benutzerprozeß.
temporale Korrelation	Gibt an, wie die zeitliche Abhängigkeit zwischen den Benutzern ist. Beispiele: synchron, asynchron.
Textverarbeitung	Software, die dem Benutzer bei der Erstellung, Modifikation und Speicherung von Texten unterstützt.
Tokenmechanismus	Speziallfall von Sperrmechanismen in dem stets nur ein Benutzer Daten modifizieren darf.
Transaktionen	Zusammenfassung von mehreren Operationen, für die die ACID-Eigenschaften gelten. Dient der Sicherung der Nebenläufigkeitskontrolle.
Transformationsverfahren	Nebenläufigkeitskontrolle, in der konfliktierende Aktionen durch Ausführen entsprechend transformierter Aktionen aufgelöst werden.
Umgebungs-Reservierung	Reserviert neben dem ModelObject des Grundes die ModelObjects aus der lokalen Umgebung.
UML	Diagrammsprache zum Objektorientierten Softwareentwurf. Abkürzung für Unified Modeling Language.
Undo-Redo	Einfacher Konfliktauflösungsmechanismus in DyCE, bei dem auf Grund von Konflikten zurückgenommene Transaktionen wiederholt werden.
veraltete Transaktion	Transaktion, die sich auf einen Datenzustand bezieht, der nicht aktuell ist.
Verteilungsstruktur	Art der Datenaufbewahrung und -verwaltung, bei Groupwaresystemen. Man unterscheidet zwischen zentralen und einer replizierten Architekturen.
Verteilverfahren	Arbeitsparadigma, bei dem die zu erledigende Arbeit zunächst, gemäß einer Grobstruktur, auf alle Teilnehmer verteilt wird. Nach der Erledigung der aufgeteiten Arbeit wird das Ergebnis gemeinsam überarbeitet.
Verzögerte Fragmentierung	Fragmentierungsverfahren, das fragmentiert, sobald mehr als ein Teilnehmersystem einen Knoten bearbeitet.
Vorbeugende Fragmentierung	Führt die Fragmentierung während der Navigation aus.

vorbeugende Reservierung	Modifiziert die Reservierungsfunktion während der Navigation.
Werkzeugleiste	Spezieller Teil der Benutzungsschnittstelle, die aus einer Sammlung von Knöpfen besteht, mittels derer der Benutzer mit der Software interagiert.
Wurzelknoten	Ausgezeichneter Knoten des Strukturgraphen, der keinen Vater in der Primärstruktur besitzt.
WYSIWIS	Abkürzung für „What You See Is What I See“[21].
XML-Element	Datensegment im XML-Quellcode. Entspricht einem Knoten im XML-Graphen.
XML-Graphen	Graphsicht auf einen XML-Quellcode. Verwendet im Ansatz von Beech, Malhotra und Rys.
XML-Tags	Kennzeichnung eines XML-Elements.
Zeitpunkt	Zwei Transaktionen werden zum selben Zeitpunkt initiiert, wenn sie auf dem gleichen Zustand basieren. Der Begriff Zeitpunkt bezeichnet in dieser Arbeit den Zeitraum zwischen zwei Zustandswechseln von beliebigen ModelObjects im System.
zentrale Architektur	Verteilungsstruktur, bei der alle Daten auf einem Server liegen. Manipulationen werden nur dort vorgenommen.

[21] deutsch: Was du siehst, sehe auch ich.

Quellen

[App95] Appelrath, Hans-Jürgen; Ludewig, Jochen: „Skriptum Informatik – eine konventionelle Einführung“ – Stuttgart, Teubner, 1995

[BAM99] David Beech, Ashok Malhotra, Michael Rys, „A Formal Data Model and Algebra for XML“, W3C XML Query Working group note, September 1999

[BS98] Borghoff, Uwe M.; Schlichter, Johann H.: „Rechnergestützte Gruppenarbeit. Eine Einführung in Verteilte Anwendungen“ – Berlin, Heidelberg: Springer, 1998

[DB] Barr, Dale; Keysar, Boaz: „The Collaborative Model and Psychological Evidence: Some Principles for Designing Cooperative Systems“ URL: ftp://zenon.inria.fr/acacia/Clark-Workshop/Papers/BarrAndKeysar.ps (28. August 2002)

[DB92] Dourish, Paul; Bellotti, Victoria: „Awareness and Coordination in Shared Workspaces“ In: „Proceedings of the Conference on Computer Supported Cooperative Work“ – 1992

[Dum] Dumke, Reiner R. „UML-Tutorial“ http://www-ivs.cs.uni-magdeburg.de/~dumke/UML/ (23. August 2002)

[Dör96] Dörner, Diertrich: „Die Logik des Mißlingens. Strategisches Denken in komplexen Situationen“ – Hamburg: 1996, Rowohlt

[Grö01] Prof. Dr. Martin Grötschel, „Graphen. Hypergraphen, Matroide: wichtige Definitionen und Bezeichnungen“

[Her81] Herkner, Werner: „Einführung in die Sozialpsychologie“, 2. Auflage – Bern: Huber, 1981

[Hil91] Hill, Daniel: „What's NEW with Lotus Domino for iSeries in OS/400 V5R1“ (2001) URL: http://www-919.ibm.com/servers/eserver/iseries/developer/domino/documents/newv5r1/newv5r1r.pdf (28. August 2002)

[HS00] Heuer, Andreas / Saake, Gunter „Datenbanken: Konzepte und Sprachen“ – Landsberg, 2000, mitp

[IBM01] IBM: „Lotus Whitepaper: Real-time Collaboration with Lotus Sametime“ (2001) URL: http://www-8.ibm.com/software/au/downloads/real-time_collab.pdf (28. August 2002)

[JavaWorld99] „XML for the absolute beginner“, Author: Mark Johnson, URL: http://www.javaworld.com/javaworld/jw-04-1999/jw-04-xml.html

[KS91] Kiyek, Karl-Heinz / Schwarz, Friedrich :„Mathematik für Informatiker 1“ – Stuttgart, 1991, Teubner.

[MR96] Michailidis, Antonios; Rada, Roy: „A Review of Collaborative Authoring Tools“ In: Rada, Roy: „Groupware AND Authoring“ – San Diego: Academic Press Ltd, 1996

[ND] NotesDesign: „Why is Domino the best platform in which to develop flexible, collaborative systems ?“ URL: http://www.notesdesign.com/ndhtml/ndintro4.htm (28. August 2002)

[OMG] UML Spezifikation von OMG

[OO96] Olson, Gary M.; Olson, Judith S.: „The Effectiveness of Simple Shared Electronic Workspaces“ In: Rada, Roy: „Groupware AND Authoring“ – San Diego: Academic Press Ltd, 1996

[Ora96] Oravec, Jo Ann: „A Portrait of the Author as an Interacting Group“ In: Rada, Roy: „Groupware AND Authoring“ – San Diego: Academic Press Ltd, 1996

[PMB96] Posner, Ilona; Mitchell, Alex; Baecker, Ronald: „Learning to Write Together“ In: Rada, Roy: „Groupware AND Authoring“ – San Diego: Academic Press Ltd, 1996

[Rie97] Riehle, Dirk: „A Role-Based Design Pattern Catalog of Atomic and Composite Patterns Structured by Pattern Purpose“ (1997) URL: http://www.berlin-consortium.org/docs/papers/ubilab-pattern-catalog.pdf (29. Mai 2002)

[Sch98] Schuhmann, Josefine: „Eine Infrastruktur für kollaboratives Lernen auf der Basis dreidimensionaler Wissensrepräsentation“ – Frankfurt: 1998

[Tie01] Tietze, Daniel A.: „A Framework for Developing Component-based Cooperative Applications“ – Sankt Augustin: GMD – Forschungszentrum Informationstechnik, 2001

[TS00] Daniel A. Tietze, Ralf Steinmetz: „Ein Framework zur Entwicklung komponentenbasierter Groupware“ In: Proceedings der Fachtagung D-CSCW 2000 - München, September, 2000

[W3C] „Document Object Model (DOM) Level 1 Specification“ (1998) URL: http://www.w3.org/TR/REC-DOM-Level-1/ (28. August 2002)

Abbildungsverzeichnis

Tabellen- und Quellcodeverzeichnis