# Flexible Versionierung semistrukturierter Dokumente im Kontext von Autonomie und Kooperation

*Diplomarbeit*

Torsten Lenk
Fachbereich Elektrotechnik
Technische Universität Hamburg-Harburg

Betreuung:

Prof. Dr. Joachim W. Schmidt,
Arbeitsbereich Softwaresysteme
Technische Universität Hamburg-Harburg

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

27. Feb 2001

# Inhaltsverzeichnis


**1 Einleitung und Motivation** **1**

1.1 Problemstellung . . . . . 1

1.2 Ziele der Arbeit . . . . . 3

1.3 Struktur der Arbeit . . . . . 3

**2 Versionsverwaltung** **5**

2.1 Überblick und Begriffe . . . . . 5

2.1.1 Versionierung und Versionsverwaltung . . . . . 5

2.1.2 Glossar . . . . . 6

2.2 Versionierung und ihre Anwendungsgebiete . . . . . 8

2.2.1 Ursprünge: Versionierung in der Produktion . . . . . 8

2.2.2 Versionierung in der Softwareentwicklung . . . . . 8

2.2.3 Die Rolle der Versionierung für die Dokumentenerstellung . . . . . 9

2.3 Anforderungen an eine Versionsverwaltung . . . . . 10

2.3.1 Allgemeine Anforderungen . . . . . 10

2.3.2 Anforderungen für die Versionierung semistrukturierter Dokumente . . 12

2.3.3 Anforderungen im Kontext von Kooperation und Autonomie . . . . . 13

2.3.4 Mehrwertdienste . . . . . 14

2.4 Bewertungskriterien . . . . . 16

2.4.1 Quantitative Evaluierung . . . . . 16

2.4.2 Qualitative Evaluierung . . . . . 18

**3 Konzepte und Verfahren** **19**

3.1 Konzepte der Versionierung . . . . . 19

3.1.1 Zustandsorientierte Versionierung . . . . . 20
3.1.2 Änderungsorientierte Versionierung . . . . . 22
3.2 Konzepte der Differenzenspeicherung . . . . . 23
3.2.1 Ansätze zur Differenzenspeicherung . . . . . 24
3.2.2 Wiederherstellung von Versionen . . . . . 25
3.2.3 Implementierungsansätze . . . . . 26

**4 Verwendete Sprachen und Werkzeuge 29**
4.1 XML . . . . . 29
4.2 Java . . . . . 30
4.3 XTree . . . . . 31

**5 Entwurf einer Versionsverwaltung 33**
5.1 Allgemeine Kriterien für die geplante Anwendung . . . . . 33
5.1.1 Abspeicherung . . . . . 34
5.1.2 Anreicherung mit Metainformationen . . . . . 36
5.1.3 Navigierbarkeit . . . . . 38
5.1.4 Mehrwertdienste . . . . . 41
5.2 Architekturüberblick . . . . . 43
5.3 Abläufe und Prozesse . . . . . 45
5.3.1 Extraktion . . . . . 45
5.3.2 Archivierung . . . . . 46
5.3.3 Deltabildung . . . . . 47
5.4 Das Verfahren zur Differenzenbildung . . . . . 49
5.4.1 *XTree* . . . . . 51
5.4.2 Inhaltsvergleich . . . . . 51
5.5 Präsentation der Änderungen . . . . . 53

**6 Ausgewählte Aspekte der Realisierung 55**
6.1 Grobarchitektur der Realisierung . . . . . 55
6.1.1 Erstellung eines strukturierten Inhaltsdeltas (SCDelta) . . . . . 55
6.1.2 Recovery eines Dokuments . . . . . 56
6.2 Verwendete Algorithmen . . . . . 57

6.2.1 Strukturvergleich . . . . . 58
6.2.2 *SCDelta* . . . . . 63
6.3 Bewertung des Verfahrens . . . . . 68
6.3.1 Evaluierungsumgebung . . . . . 68
6.3.2 Auswertung . . . . . 68
6.3.3 Fazit . . . . . 73

**7 Zusammenfassung und Ausblick** **75**
7.1 Bewertung des entwickelten Systems . . . . . 76
7.2 Verbesserungsmöglichkeiten . . . . . 77

**A DTD von XTree** **79**

**B DTD von SCDelta** **81**

**C DTD zur Versionsarchivbeschreibung** **83**

**D Quantitative Evaluierung** **85**

**E Zustand- und Änderungsorientierte Versionierung** **87**
E.1 Zustandorientierte Versionenunterstützung . . . . . 87
E.2 Änderungsorientierte Versionenunterstützung . . . . . 87

# Abbildungsverzeichnis


2.1 Beispielszenario in einem Versionenarchiv . . . . . 7

2.2 Autonomie und Kooperation in der Dokumentenerstellung . . . . . 9

2.3 Fortlaufende Numerierung in der Versionenverwaltung . . . . . 11

2.4 Darstellung eines Versionenarchivs mit abgelegten Änderungen statt Vollversionen . . . . . 11

2.5 Darstellung eines Merging-Vorgangs anhand eines Dokuments mit verschiedenen Datentypen . . . . . 16

3.1 Symmetrische Deltas . . . . . 24

3.2 Gerichtete Deltas [Rei94] . . . . . 24

4.1 Aktuelles Beispiel eines XML Dokuments mit Webeinbindung . . . . . 29

5.1 Die Darstellung einer Versionsverwaltung unter Berücksichtigung der verschiedenen Komponenten . . . . . 34

5.2 Rückwärtsdeltas in verschiedenen Strängen . . . . . 36

5.3 Darstellung des Drei-Dimensionenmodells nach Reichenberger . . . . . 39

5.4 Verschiedene Archivierungsformen . . . . . 43

5.5 Darstellung der Schnittstellen . . . . . 44

5.6 Delta-Erstellung auf struktureller Ebene . . . . . 48

5.7 Delta-Erstellung als Aktivitätsdiagramm . . . . . 49

5.8 Aufteilung in Struktur und Inhalt . . . . . 50

6.1 Grobarchitektur: Deltaerstellung . . . . . 56

6.2 Architektur des Systems zur Erstellung eines Editskripts . . . . . 57

6.3 Grobarchitektur: Recovery . . . . . 58

6.4 *XTree* Baumdarstellung . . . . . 59

6.5 Editskript von *XTree* . . . . . 60
6.6 Der *Update*-Befehl . . . . . 60
6.7 Der *Insert*-Befehl . . . . . 61
6.8 Der *Delete*-Befehl . . . . . 61
6.9 Der *Move*-Befehl . . . . . 62
6.10 Der *Copy*-Befehl . . . . . 62
6.11 Der *Glue*-Befehl . . . . . 62
6.12 Hashtabelle und Linktabelle . . . . . 64
6.13 Editskript von *SCDelta* . . . . . 65
6.14 Vergleich *SCDelta*: Unix-*diff* . . . . . 70

D.1 Vorteil von *SCDelta* gegenüber Unix-*diff* in Zeichen . . . . . 85

# Tabellenverzeichnis

2.1 Anforderungsprofil nach [Haa96] (Stand:1996) . . . . . 18

6.1 Vergleich der Größe der resultierenden Editskripte . . . . . 69

# Kapitel 1

# Einleitung und Motivation

In der sich entwickelnden Informationsgesellschaft gewinnt die Erstellung von Informationsartefakten im kooperativen Kontext immer mehr an Bedeutung. Eine zusätzliche Herausforderung stellen die örtliche und zeitliche Autonomie der partizipierenden Gruppen dar. Autorenteams sind durch die neuen Möglichkeiten der Vernetzung nicht mehr auf lokale Gruppen beschränkt, sondern setzen sich nach Verfügbarkeit und Kompetenz auch international zusammen. Insofern soll allen Parteien die Möglichkeit gegeben werden, jederzeit und an jedem Ort an einem Projekt zu arbeiten, um die Fertigstellung des Informationsartefakts nicht zu verzögern. Zusätzlich müssen gleichzeitig getätigte Änderungen koordiniert bzw. verarbeitet werden.

In dieser Arbeit werden Konzepte zur kooperativen Erstellung semistrukturierter Dokumente erörtert. Eine flexible Versionsverwaltung spielt dabei eine zentrale Rolle und bildet die Basis für wichtige Mehrwertdienste wie Notifikation, *Merging* und Änderungsvisualisierung. Der in dieser Arbeit entworfene und implementierte Ansatz für eine Versionsverwaltung stellt eine Grundlage dar, der die Möglichkeiten im Bereich der Übersicht erweitert und Unterstützung in den Bereichen Autonomie und Kooperation schafft.

## 1.1 Problemstellung

Eine traditionelle Aufgabe der Versionsverwaltung ist es sicherzustellen, daß Änderungen nicht verloren gehen und eine Evolution des betrachteten Dokuments nachvollziehbar ist. Ältere Versionen müssen erhältlich und vom Kontext her einzuordnen sein. Die Speicherung der einzelnen Versionen sollte effizient vonstatten gehen, sprich möglichst wenig redundante Daten dürfen die Speicherkapazität belasten.

Die Erstellung größerer Dokumente findet selten in einem einzelnen Arbeitsgang statt. Sie zieht sich meist über einen längeren Zeitraum hin und ist durch mehrere Entwicklungsstufen gekennzeichnet. Hier kann die Versionsverwaltung helfen, diese Entwicklungsschritte zu koordinieren, zu beobachten und zu optimieren.

Die Projektkoordination wird mit vielen verschiedenen Teilaspekten konfrontiert. Da Projekte vielfach in Teams durchgeführt werden, gewinnen kollaborative Dienste zunehmend an Be-

deutung. Zum einen müssen Aufgaben in verschiedene Bereiche aufgeteilt werden, die kompetenten Autoren zugeordnet werden (koordinativ inhaltlich). Zum anderen muß gewährleistet sein, daß verschiedene Arbeitsumgebungen (z.B. *mobile computing*) genutzt werden können (koordinativ logistisch). Dabei muß die Kopplung der einzelnen Teilprojekte berücksichtigt werden. Außerdem sind flexible Transparenzgrade nötig, die es den Teilnehmern erlauben, Teile zu bearbeiten, zu sperren oder auch freizugeben. Alle Aktionen müssen koordiniert und kommentiert und den übrigen Teammitgliedern zur Verfügung gestellt werden.

Neben der Kollaboration ist die Autonomie ein weiterer wichtiger Punkt. Im Zusammenhang mit kreativer Arbeit ist das autonome Arbeiten sehr wichtig. Dem einzelnen Anwender muß Zugriff zu den Dokumenten gewährleistet werden. Außerdem muß eine Konsistenz der getanen Arbeit gewährleistet werden und der autonome arbeitende Autor muß in der Lage sein, sich seinen Autonomiegrad zu erhalten. So kann es von Vorteil sein, nicht alle Ergebnisse gleich allen Teammitgliedern zugänglich zu machen.

Es wird deutlich, daß Kollaboration und Autonomie in einem engen Zusammenhang zueinander stehen. Der kollaborative Aspekt fördert den aktiven Informationsaustausch, der koordiniert und kanalisiert den teilnehmenden Parteien die Möglichkeit bietet, örtlich unabhängig zu arbeiten. Die Ausweitung der unter Versionsverwaltung zusammengefaßten Dienste hat inzwischen ein völlig eigenständiges Forschungsfeld geschaffen, welches sich mehr und mehr an den Bedürfnissen von kollaborativen Arbeitsumgebungen orientiert und dabei Rücksicht auf die Autonomie der Anwender nimmt.

Einer dieser Basisdienste ist die Versionsspeicherung. Wird in einer Versionsverwaltung eine neue Version abgespeichert, so wird weitestgehend versucht, auf das Abspeichern redundanter Daten zu verzichten. Es soll nur der Anteil neuer Informationen gespeichert werden, der sich verändert hat, um unnötige Ressourcenverschwendung zu vermeiden. Hier wird meist mit sogenannten Differenzen (Deltas) gearbeitet, die die Unterschiede zwischen zwei Versionen beschreiben, welche es ermöglichen, nicht komplett abgespeicherte Dokumente zu rekonstruieren.

Deltas für andere Zwecke als die Speicherung bzw. Rekonstruktion von Versionen zu verwenden spielte lange eine untergeordnete Rolle [HVT98]. Relevante Auswahlkriterien für Differenzenverfahren waren Speicheraufwand und Laufzeit (sowohl für die Berechnung eines Deltas, als auch für die Rekonstruktion). Neuere Ansätze aus der Arbeitsplanung und -unterstützung [NCK+92] zeigen auf, daß bestimmte Bedürfnisse in der kollaborativen Dokumentenerstellung mit Hilfe von Änderungsinformationen erfüllt werden können. An einem Delta läßt sich die Entwicklung eines Dokuments anhand der einzelnen Änderungsschritte explizit nachvollziehen. Allerdings erstellen die Standardverfahren der Deltaerstellung didaktisch nur unzureichend aussagekräftiges Material. Um den Anwendern als Unterstützung dienen zu können, müssen diese Deltas erst aufwendig aufgearbeitet werden.

Ziel dieser Arbeit ist nun einen Ansatz zu untersuchen, der die späteren Anwendungsmöglichkeiten der Deltas schon in der Entstehung berücksichtigt und somit eine höhere Konsistenz hinsichtlich der kollaborativen Dokumentenerstellung erreicht. Hierzu werden bei der Erstellung von Deltas die gleichen Strukturen wie bei den betrachteten Dokumente verwendet. Diese strukturierten Inhalte können nun auf vielerlei Arten (zusätzlich zur effizienten Speicherung) weitergenutzt werden. Das betrachtete Format (XML) stellt einen Standard dar, für den schon viele Anwendungen und Dienste existieren bzw. in der Entstehung sind. Die

Deltas können sowohl direkt (als Änderungsinformationen), als auch für weitere Versionsverwaltungsdienste, wie z.B. Evolutionsverfolgung oder Notifikation genutzt werden.

## 1.2 Ziele der Arbeit

Die Zielsetzung dieser Arbeit besteht zunächst in der Identifikation der Anforderungen einer Versionsverwaltung zur Unterstützung der kollaborativer Erstellung semistrukturierter Dokumente. Verschiedene Konzepte sollen untersucht und ein Anforderungsprofil erstellt werden. Aus dem Anforderungsprofil wird eine Versionsverwaltung entworfen.

Entsprechend den Anforderungen der entworfenen Versionsverwaltung wird eine Versionsspeicherung implementiert, welche einen Basisdienst darstellt. Im Vordergrund stehen Flexibilität und Anwenderunterstützung. Der Dienst sollte eine aussagekräftige Präsentation der Daten ermöglichen, welche Schlüsse auf die Dokumentenentwicklung zuläßt. Dies soll den kollaborativen Charakter der entworfenen Versionsverwaltung unterstreichen.

Wichtig in dieser Arbeit ist die Offenheit und Flexibilität. Der hier verwendete Dienst und auch die eigene Implementation arbeiten mit einem generischen Ansatz und legen Wert auf Offenheit. Die Flexibilität bezieht sich sowohl auf den Einfluß der Anwender selbst auf die zu erstellenden Deltas bzw. Evolutionshistorien, als auch auf Kompatibilität des Dienstes mit bestehenden Anwendungen.

## 1.3 Struktur der Arbeit

In Kapitel 2 werden die in dieser Arbeit verwendeten Begriffe vorgestellt und erläutert. Desweiteren wird ein Überblick über die Anwendungsgebiete der Versionierung bzw. deren historische oder auch applikationsabhängige Entstehung gegeben. Im weiteren werden allgemeine Anforderungen an eine Versionsverwaltung im Kontext von Kollaboration und Autonomie aufgestellt. Für die die Evaluierung eines Dienstes, sowohl in quantitativer, als auch in qualitativer Hinsicht werden Bewertungskriterien aufgestellt.

Auf dem Gebiet der Versionsverwaltung existieren schon mannigfaltige Ansätze und Konzepte, die in Kapitel 3 klassifiziert und dargestellt werden. Die Differenzenspeicherung stellt einen Basisdienst der Versionsverwaltung dar. Auch hier gibt es verschiedene Konzepte die vorgestellt werden.

Die für diese Arbeit verwendeten Technologien sind XML als Standard für semistrukturierte Dokumente und Java als Programmiersprache für die Realisierung und Implementation des vorgestellten Ansatzes. Weiterhin wird ein existierendes Vergleichsverfahren für XML Dokumente in dem in dieser Arbeit implementierten Dienst Verwendung finden. Kapitel 4 gibt einen Überblick über diese Technologien.

In Kapitel 5 wird eine Versionsverwaltung entworfen. Zuerst wird auf die Anforderungen eingegangen, um dann einen groben Architekturüberblick zu geben. Nach der Darstellung typischer Abläufe und Prozesse wird der implementierte Basisdienst im Detail beschrieben.

In Kapitel 6 werden neben der Architektur die verwendeten Algorithmen bzw. deren Imple-

mentation vorgestellt. Hierbei werden die modularen Teile einzeln beschrieben. Es folgt eine Bewertung der implementierten Komponenten nach quantitativen und qualitativen Maßstäben (gemäß Kapitel 2).

Im letzten Kapitel werden nochmal die Hauptmerkmale des implementierten Dienstes zusammengefaßt. Desweiteren werden Schlüsse für die Verwendbarkeit des Verfahrens gezogen und mögliche Verbesserungen und Erweiterungen diskutiert.

# Kapitel 2

# Versionsverwaltung

Nach der eingehenden Darstellung der grundsätzlichen Problematik ist dieses Kapitel der Begriffseinordnung und den Versionierungsanforderungen gewidmet. Nach einem kurzen historisch angelegtem Überblick, wird der Bezug der speziellen Anforderung zu dem in dieser Arbeit gewählten Szenario hergestellt. Die verschiedenen Anforderungen legen die Grundlage für den Entwurf einer Versionsverwaltung. Auch werden Bewertungskriterien festgelegt, die zur Evaluierung der implementierten Teile verwendet werden.

## 2.1 Überblick und Begriffe

### 2.1.1 Versionierung und Versionsverwaltung

Der Begriff der Version bzw. der Versionierung spielt in vielen Gebieten und Disziplinen eine Rolle. Eine Definition für die Versionierung vom World Wide Web Consortium (www.w3.org) lautet:
„The storage and management of previous copies of a piece of information, for security, diagnostics, and interest.“ Eine allgemeingültige Definition zu finden ist schwierig, da es sich um ein dynamisches und vielfältiges Feld handelt. Die Versionierung kümmert sich darum, daß der Benutzer in der Lage ist, alle relevanten Zwischenschritte eines Bearbeitungsvorganges abzuspeichern und wiederzufinden.

Bei der Erstellung von Dokumenten entstehen oft eine Vielzahl von Entwicklungszyklen und auch Einzelkomponenten, die alle verwaltet werden müssen. Hierbei ist es wichtig, diese Dokumente oder auch nur die einzelnen Komponenten in ihrer kompletten Entwicklungsvielfalt (Versionen, Revisionen, Varianten) und den bestehenden Abhängigkeiten sichtabar und verfügbar zu machen.

Eine Versionsverwaltung ist ein System, welches dem Benutzer beim Umgang mit den Versionen unterstützt. Dies umfaßt nicht nur die Numerierung der einzelnen Versionen und die Speicherung bzw. Extraktion. Hier kommen noch weitere Dienste hinzu, wie z.B. die Gewährleistung der Übersichtlichkeit über das Versionsarchiv und das schnelle und konfortable Auffinden einer gewünschten Version. Die Unterstützung in Bereichen wie Evolutionshistorien, Änderungsmanagement, Notifikationsservice, Wiederzusammenführen von verschiedenen Ver-

sionssträngen, etc. gehört ebenfalls zur Versionsverwaltung.

### 2.1.2 Glossar

In diesem Abschnitt werden die später verwendeten Begriffe zugeordnet und definiert. In der Literatur wird oft von Version, Revision, Konfiguration, Variante, Komponenten oder auch Objekten gesprochen. Diese Begriffe sind intuitiv meist schon belegt, können aber durchaus abweichend interpretiert werden und somit zu Irritationen führen.

Um diese Begrifflichkeiten für diese Arbeit festzulegen, wird die begriffliche Einordnung von Reichenberger [Rei94] zugrunde gelegt:

**Komponente (*component*)**: Als Komponenten werden alle nicht mehr aufspaltbaren Einheiten eines Gesamtkonstrukts bezeichnet. Hierbei geht es nicht um die theoretisch mögliche Aufspaltung, sondern um die praktisch relevante. Weitere verwandte Begriffe sind, je nach Kontext, Teil oder auch Atom.

**Objekt (*object*)**: Als Objekt wird eine Version einer Komponente bezeichnet, sprich es können mehrere Objekte einer Komponente existieren.

**Variante (*variant*)**: Obwohl Variante zumeist als Substantiv Verwendung findet, ist der Begriff aber genaugenommen nur als Relation zu verstehen. Und zwar in dem Kontext *ist-Variante-von*. Aus dem Wortstamm ergibt sich, daß eine Variante in dieser Rolle nur in Bezug auf ein anderes Objekt definiert ist.

**Revision (*revision*)**: Auch in diesem Fall handelt es sich um die Relation *ist-Revision-von*. Allerdings hat sich im allgemeinen Sprachgebrauch durchgesetzt, Komponenten auch als Revisionen zu bezeichnen, selbst wenn es sich um die Ur-Version handelt. Hierbei ist wichtig, daß die Relation *ist-Revision-von* transitiv ist und nicht gleichberechtigte (sondern durch Revision auseinander hervorgegangene) Objekte beschreibt.

**Version (*version*)**: Version ist quasi die kontextuelle Einordnung eines Objekts. Versionen implizieren ein Versionsarchiv in das sie einordbar sind, z.B. durch Vorgänger- oder Nachfolgerversion. Sowohl bei Revisionen und Varianten, als auch bei Ur-Objekten spricht man von Versionen. In der Praxis verschwimmen die Begriffe ein wenig. Allerdings reicht es in dieser Arbeit, wenn auf die verfeinerten Betrachtungen verzichtet wird und nur noch von Versionen gesprochen wird.

**Extrahieren (*check-out*)**: Das Kopieren eines Objektes zur Weiterbearbeitung wird als Extrahieren bezeichnet.

**Archivieren (*check-in*)**: Das Rückkopieren eines Objektes nach der Weiterbearbeitung [1] in das Versionsarchiv wird als archivieren bezeichnet.

**Sperren (*lock*)**: Um zu verhindern, daß ein bestimmtes Objekt extrahiert (oder auch nur archiviert) wird, kann man es sperren. Dies kann zur Vermeidung von Konflikten dienlich sein, schränkt aber parallele Arbeit von Autoren ein.

**Freigeben (*unlock*)**: Wurde ein Objekt gesperrt, so muß es auch wieder freizugeben sein.

[1] Hierbei muß nicht notwendigerweise etwas geändert worden sein.

Enthält eine Versionsverwaltung z.B. einen automatischen Sperrmechanismus beim Extrahieren, so ist es üblich, die Freigabe auch automatisch mit der Archivierung des Objektes zu verknüpfen.

**Konfiguration (*configuration*)**: Als Konfiguration wird die Sammlung einzelner Objekte zu einem größeren Ganzen bezeichnet, in diesem Fall das gesamte Dokument.

**Deltaspeicherung (*delta storage*)**: Um Ressourcen zu sparen, werden nicht die kompletten Versionen abgespeichert, sondern nur die jeweiligen Änderungen bzw. Differenzen. Wichtig ist hierbei, daß es in jedem Fall möglich ist, alle archivierten Komplettfassungen durch Nutzung der Deltas zu rekonstruieren.

**Editierfolge (*edit script* = Editskript, *delta*)**: Die Editierfolge beschreibt die Befehle, die nötig sind, um aus einer Komplettfassung einer Version eine andere zu rekonstruieren.

Das folgende Szenario faßt die wichtigsten Begriffe noch einmal zusammen und zeigt sie in ihrer Einbindung in einen typischen Vorgang im Bereich der Versionsverwaltung: Aus den in einem Versionenarchiv gespeicherten Komponenten kann ein Objekt extrahiert werden und in einer privaten (autonomen) Umgebung bearbeitet werden (siehe Abb. 2.1). Gleichzeitig kann das Objekt gesperrt werden, um die Konsistenz zu wahren. Hat der Nutzer seine Änderungen abgeschlossen, so muß das Objekt wieder archiviert werden, um eine neue Version erzeugen zu können. Mit der Archivierung erfolgt gleichzeitig (im Falle eines vorangegangenen Sperrens) die Freigabe des Objekts für andere Benutzer.

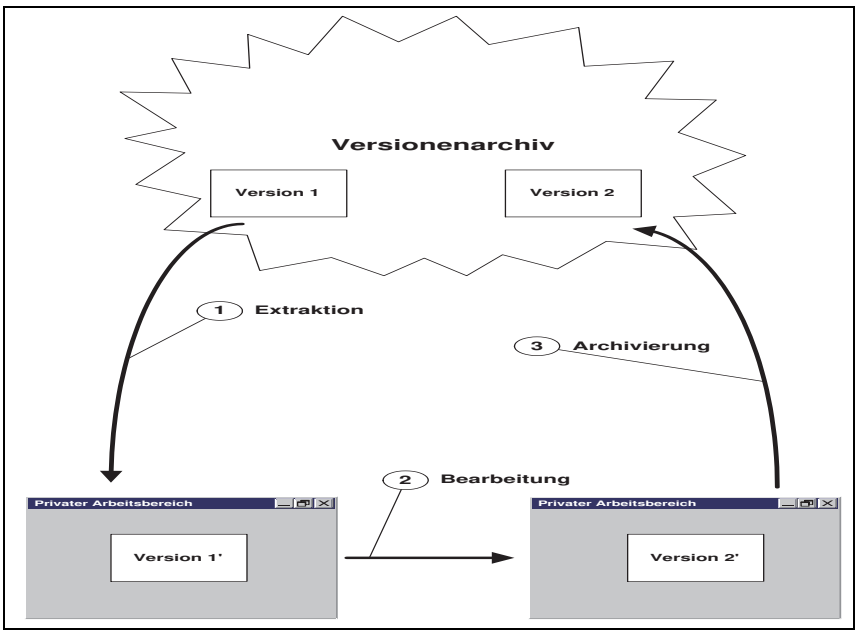


Abbildung 2.1: Beispielszenario in einem Versionenarchiv

## 2.2 Versionierung und ihre Anwendungsgebiete

### 2.2.1 Ursprünge: Versionierung in der Produktion

Der Begriff der Versionierung spielte in der Fertigungsindustrie schon lange eine wichtige Rolle. Sogenannte Änderungsprotokolle sind schon seit Anfang des 20. Jahrhunderts Normen unterworfen und werden in Produktentwicklungsordnern abgelegt. Der Gedanke ist es daher, die Entwicklungsphase genau zu dokumentieren und somit sowohl die Möglichkeit zu haben Neuentwicklern den Einstieg zu ermöglichen, als auch die Option des rückwärtigen Aufarbeitens oder gar des Rückschritts in eine zuverlässige, weil bewährte Entwicklungsphase, z.B. zur Korrektur von Fehlentwicklungen, zu besitzen.

Das Ziel dieses, sich auf die industrielle Anfertigung von Einzelteilen beziehenden Prozesses kann analog auf Anwendungen im IT - Bereich übertragen werden. Es geht darum Entwicklungsprozesse zu koordinieren und nachvollziehbar zu machen und stellt einen elementaren Bestandteil der Dokumentation solcher Prozesse dar. Der Begriff des *change management* findet sich heute in den verschiedensten Branchen wieder.

In der fertigenden Industrie wurden Historienspeicherungen oder manuelle Versionsmechanismen schon früh eingesetzt und fanden auch gleich Einzug in die computer-unterstützte Entwicklung von Fertigungsgütern (CAD[2]).

In der Entwicklung von Softwareprodukten wurden diese Mechanismen zuerst eher individuell und manuell eingesetzt. Inzwischen sind sie integraler Bestandteil von Werkzeugen wie Software-Entwicklungsumgebungen (SDE[3]). Gerade diese leisten wichtige Beiträge für die neuesten Fortschritten auf dem Gebiet der Versionsverwaltung.

### 2.2.2 Versionierung in der Softwareentwicklung

Die Entwicklung von Software ist ein langfristiger Änderungsprozeß. Dieser umfaßt nicht nur den Zeitraum bis zur eigentlichen Markteinführung, sondern erstreckt sich meist über einen weit längerer Zeitraum. Selbst bei Erreichen einer gewissen Reife und Stabilität werden weitere Ergänzungen oder Verbesserungen nötig, um den Anforderungen des Marktes gerecht zu werden und konkurrenzfähig zu bleiben. Dies setzt ein gutes Versionsmanagement voraus.

Sämtliche relevante Zwischenversionen müssen erhalten werden und die Entwicklungsstufen müssen nachvollziehbar sein. Nur die veröffentlichten Versionen zu erhalten, würde bedeuten, daß die Ideen der Programmentwicklung verloren gehen und nicht reproduzierbar sind. Dies kann gerade bei nachträglich erkannten Fehlern erhebliche Verzögerungen bedeuten. Dies stellt erhöhte Anforderungen an das Speichermanagement. Wird zudem noch die Anzahl der gespeicherten Versionen und Verzweigungen sehr groß, so läßt die Variante der fortlaufenden Numerierung sehr schnell den Überblick verlieren. Hier gilt es geeignete Dienste bereitzustellen, die den Benutzer bei der Navigation unterstützen.

Um den Entwicklern genügend Autonomie zugestehen zu können wird eine flexible zugriffskontrollierte Benutzerverwaltung benögtigt. Steigt die Anzahl der an einem Projekt beitei-

[2] *Computer Aided Design*
[3] *Software Development Environment*

ligten Personen an, reichen koordinative Maßnahmen auf persönlicher Ebene nicht mehr aus, insbesondere wenn örtliche Synchronität abnimmt.

Eine Versionsverwaltung, die jedem zu jeder Zeit und an jedem Ort die Möglichkeit gibt, seine Arbeitskraft für das betrachtete Projekt einzusetzen und sich über alle relevanten Änderungen und Vorkomnisse zu informieren, ist das Ziel. Hierbei sind Anwenderfreundlichkeit und Übersichtlichkeit elementare Voraussetzungen.

Entsprechend den immer komplexer werdenden Konstrukten, die verwaltet werden müssen, wächst auch die Versionsverwaltung zu einer immer größeren Sammlung einzelner Dienste, die alle in Einklang miteinander stehen müssen.

### 2.2.3 Die Rolle der Versionierung für die Dokumentenerstellung

Bereits in einem Szenario, welches nur einen Autor und einen Text vorsieht, werden die Vorteile von Versionierung deutlich. Durchgeführte Änderungen sind nachvollziehbar und revidierbar. Dies ist beispielsweise im Falle eines inhaltlichen oder auch gliederungstechnischen Irrweges sehr dienlich.

Heutige Szenarien sind allerdings dahingehend komplexer, daß Dokumente aus vielen Einzelteilen zusammengesetzt sind, die von vielen Beteilligten erstellt und überarbeitet werden (siehe Abb. 2.2). Auf sämtliche Dokumententeile muß der Zugriff gewährleistet sein und eine ausreichende Identifizierung der einzelnen Versionen ist unabdinglich. Die Benennbarkeit der Versionen ist nicht nur für die Speicherungs- und Navigationsmechanismen wichtig, sondern auch eine Basis für die Kommunikation über das Dokument.

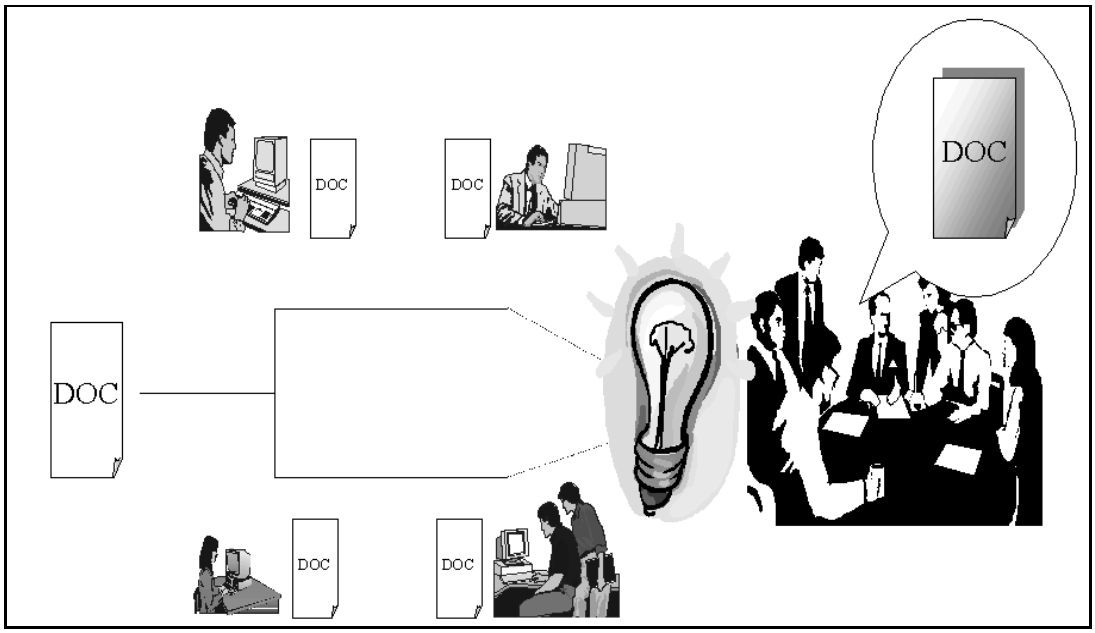


Abbildung 2.2: Autonomie und Kooperation in der Dokumentenerstellung

Außerdem sind für das kooperative Arbeiten mehrbenutzerfähige Systeme Voraussetzung, um

parallele Arbeit nicht zu behindern.

Im Zuge der Arbeitsteilung darf auch nicht vergessen werden, daß eine ausreichende Begleitdokumentation zu den einzelnen Versionen nicht fehlen darf. Es muß möglich sein festzustellen, wer der jeweilige Autor bestimmter Versionen ist, und zu welchem Zeitpunkt sie erstellt wurden. Weitere Informationen wie Kommentare zur Motivation oder weitere Zielsetzungen sind auch maßgeblich für den Erfolg eines Projekts.

Ist die Entstehungsgeschichte eines Dokuments nachvollziehbar, können zusätzlich Erkenntnisse über z.B. Arbeitsstil und Effizienz erworben werden. Optimale Abläufe können als Anleitungen oder Muster genutzt werden, vollzogene Irrwege durch Präventivmaßnahmen verhindert werden.

Kooperative Systeme können als 2-mal-2-Matrix [Sto96] mit folgenden Skalen klassifiziert werden:

- räumliche Übereinstimmung bzw. Trennung
- synchrones bzw. asynchrones Arbeiten

Je weiter sich der Arbeitspunkt vom Ursprung entfernt, desto mehr Augenmerk muß auf die unterstützende Umgebung gelegt werden. Diese zwei-dimensionale Matrix stellt eine Vereinfachung der Thematik dar, macht aber die Problematik kooperativer Systeme klar. Insofern ist es wichtig die Schnittstellen für Werkzeuge und Dienste so offen zu gestalten, daß Erweiterungen möglich sind.

## 2.3 Anforderungen an eine Versionsverwaltung

### 2.3.1 Allgemeine Anforderungen

Die Grundaufgaben der Versionsverwaltung liegen in erster Linie in der Archivierung von Versionen. Hauptaugenmerk liegt auf der Persistenz der Daten. Hierbei ist zu beachten, daß die meisten Versionsverwaltungen keine automatische Archivierung anbieten, sondern eine Archivierung der Daten immer auf Initiative des Benutzers erfolgt.

Die Speicherung sollte einem Mechanismus folgen, der es erlaubt, Versionen leicht wiederzufinden und in den Entwicklungskontext einordnen zu können. Der klassische Ansatz ist die fortlaufende Numerierung von Versionen (siehe Abb. 2.3). Wird versucht das gleiche Objekt, z.B. von zwei Autoren, die gleichzeitig an dem Objekt gearbeitet haben, zu archivieren, so wird ein neuer Zweig (paralleler Versionsstrang) eröffnet. Dies resultiert in einer komplexeren Semantik der Numerierung.

Dadurch wird eine weitere Aufgabe der Versionsverwaltung erschwert: Das Navigieren durch das Versionsarchiv. Die Versionsverwaltung muß gewährleisten, daß der Anwender in der Evolutionshistorie einer Komponente beliebig vor- und zurücknavigieren kann. Desweiteren ist es von großem Vorteil, wenn Navigationshilfen vorhanden sind, die die Orientierung im Versionenarchiv erleichtern.

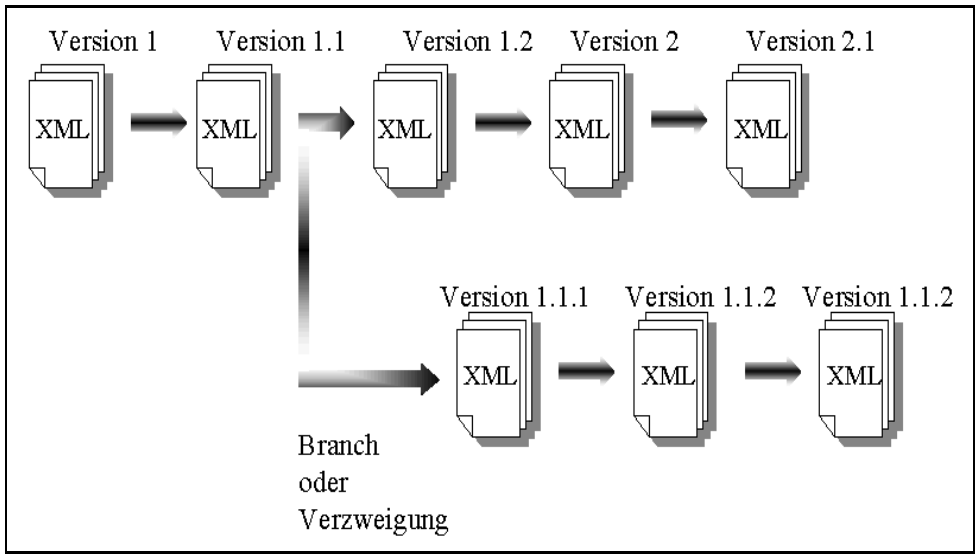


Abbildung 2.3: Fortlaufende Numerierung in der Versionenverwaltung

Der Navigation kommt eine besondere Bedeutung zu, da die einzelnen Objekte zumeist nicht in ihrer Gesamtheit abgespeichert werden, sondern nur in Form von Änderungen bezüglich ihres Nachbarobjektes. Die jeweils komplette Abspeicherung einer neuen Version würde die Speicherplatzressourcen zu stark belasten. Somit entspricht die Versionenarchivverwaltung intern mehr einer Deltaverwaltung. Der Anteil der vorhandenen Metainformationen spielt für

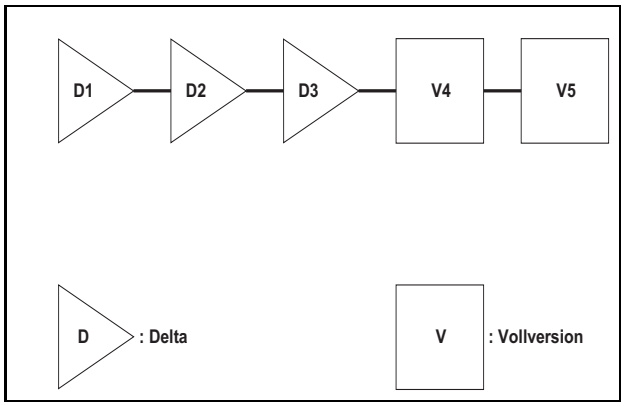


Abbildung 2.4: Darstellung eines Versionenarchivs mit abgelegten Änderungen statt Vollversionen

den suchenden Benutzer eine große Rolle. Für eine bessere Übersicht sollten die Versionen mit Metainformationen angereichert sein, die den Benutzer bei der Navigation unterstützen und weitergehend informieren. Hier sind sowohl automatisch generierte Informationen wie z.B. Datum und Autor wichtig, als auch die manuell eingegebenen Daten über den Status der Version und die Motivation des Autors.

Zusätzlich zu einer Navigation sollten auch Suchfunktionen vorhanden sein. Ein Nutzer mag

über den augenblicklichen Status oder Kontext nicht vollständig im Bilde sein, aber dennoch auf der Suche nach einer bestimmten Version sein. Hier bieten die erweiterten Metainformationen (in einem strukturierten Datenformat) Möglichkeiten eine Eingrenzung innerhalb eines Archivs aufzustellen, die das Auffinden einer bestimmten Version ermöglichen.

Zusammenfassend lassen sich folgende Kernfunktionalitäten für eine Versionsverwaltung aufstellen:

- Sichere Speicherung
- Identifizierbarkeit
- Navigierbarkeit
- Unterstützung von Archivierung und Extraktion
- Anreicherung mit Metainformationen

Mehrwertdienste wie Informationsservice, Notifikationservice, Änderungsmanagement (*change management* und das Wiederzusammenfügen von verschiedenen Zweigen spielen gerade in genauer definierten Arbeitsumgebungen eine Rolle und werden in den folgenden Abschnitten vorgestellt.

### 2.3.2 Anforderungen für die Versionierung semistrukturierter Dokumente

Semistrukturierte Dokumente sind nicht streng typisiert, zeichnen sich aber durch eine gewisse Struktur aus. Freitext löst sich mit Struktur- bzw. Formatierinformationen ab. Hierin liegen die besonderen Anforderungen begründet. Da sich semistrukturierte Dokumente in ihren Strukturanteil und ihren eigentlichen Inhalt aufteilen lassen, sollte dies auch in der Versionierung berücksichtigt werden.

Die Strukturierung der Dokumente kann als Grundlage für die systematische Betrachtung der Änderungen in einem Dokuemnt verwendet werden. Wichtig ist es auch zwischen Struktur- und Inhaltsänderungen zu unterscheiden. Legt man außerdem die Änderungen von semistrukturierten Dokumenten im selben Format ab wie die eigentlichen Dokumente, lassen sich weitere Vorteile für die Weiterverarbeitung und für den Nutzen von Mehrwertdiensten erwarten.

Ein großes Anwendungsgebiet für semistrukturierte Dokumente ist das Verbinden von verschiedenen Datenformaten (z.B. multimediale Formaten). Hier müssen Werkzeuge bereitgestellt werden, um auf die einzelnen Formate zugreifen und sie bearbeiten zu können. Gerade bei der Abspeicherung von multimedialen Formaten werden die Ablagekonzepte vor besondere Aufgaben gestellt. Hier müssen spezielle und meist sehr speicherintensive Datenformate verwendet werden. Deshalb ist es besonders wichtig, daß bei der Archivierung von Versionen nicht jeweils die kompletten Dokumente abgespeichert werden, sondern nur die relativen Änderungen [Haa96].

Ein weiteres Charakteristikum semistrukturierter Dokumente ist die mögliche Referenzierung zwischen einzelnen Teilen. Eine Versionenverwaltung muß auch in der Lage sein dies zu berücksichtigen und Änderungen von Referenzen explizit angeben können. Freitextteile

können nach bewährten konventionellen Methoden behandelt werden. Um die Aktualität der Versionsverwaltung, gerade in Bezug auf die Trennung von Strukturinformationen bzw. Inhalten, zu bewahren, ist es wichtig die einzelnen Programmteile mit austauschbaren Programmodulen zu implementieren.

### 2.3.3 Anforderungen im Kontext von Kooperation und Autonomie

Neben erweiterten Kernfunktionalitäten sind im Kontext von Kooperation und Autonomie weitere Mehrwertdienste erforderlich, die die kollaborative Arbeit an Dokumenten erleichtern.

Die angesprochene Sicherheit der Speicherung ist besonders für das autonome Arbeiten von großer Bedeutung. Um autonomes Arbeiten zu gewährleisten ist der ständig mögliche Zugriff auf das Dokument und eine Gewährleistung der Datenkonsistenz und Änderungssicherheit notwendig. Weiterhin muß ein Mehrautorensystem gewährleisten, daß ein Objekt gefunden und im gewünschten Format bereit gestellt wird. Um die individuellen Arbeitsweisen zu ermöglichen, sollten in der Bearbeitungsumgebung verschiedene Formate unterstützt werden [Haa96]. Die Identifizierbarkeit stellt einen wesentlichen Teil des kooperativen Aspekts dar. Zum einen muß der Benutzer die Möglichkeit haben die gewünschte Version eigenständig zu erkennen, um die Teile seines Verantwortungsbereichs bearbeiten zu können. Auf der anderen Seite ist sowohl die interne, als auch die externe Kommunikation abhängig von eindeutigen und plausiblen Bezeichnern.

Navigierbarkeit ist ein Punkt, der im Bereich der Kooperation besonderes Augenmerk verdient. Oft ist nur ein Teil des Versionenarchivs tatsächliches Betätigungsfeld des Benutzers. Dennoch muß die Möglichkeit bestehen, Versionen durchzusehen, um relevante Kontextinformationen zu erhalten. Hier kann eine Metainformationsebene Zeitersparnis bringen. Ebenso profitieren Archivierung/Extraktion von Metainformationen, um Dokumente besser in einen Kontext einordnen zu können. Der Benutzer hat nicht immer die Möglichkeit in die Dokumente hineinzuschauen. Gerade ältere Versionen sind nicht als Vollversionen abgelegt, sondern müssen erst generiert werden. Dies verlangsamt den Prozeß der Versionssuche stark. Der Benutzer muß in der Lage sein, eine Suche nach einer bestimmten Version aufgrund der Metadaten durchführen zu können [Emp99]. Hierzu sind sowohl eine automatische Generierung von Basisdaten, wie z.B. Datum und Autor wichtig, als auch ein vom Benutzer auszufüllendes Statusprotokoll über Inhalt, Intention und Motivation. Der manuell bei jeder Archivierung zu erstellende Teil sollte einen vorstrukturierten Anteil enthalten, um passende Mehrwertdienste zur Informationsverarbeitung kreieren zu können bzw. um eine einheitliche Dialektik zu erhalten. Zusätzlich können aber auch individuell erstellte Kommentare eingefügt werden.

Im Falle einer kollaborativen Arbeitsumgebung ist die Mehrbenutzerfähigkeit wichtig. Altmann [Alt99] postuliert in seinem Aufsatz über die kooperative Softwareentwicklung: „Voraussetzung für eine Automatisierung eines Entwicklungsprozesses ist die vollständige Beschreibbarkeit solcher Entwicklungen. [Es wurde beobachtet] daß sich eine rigide Ablaufsteuerung und Automatisierung des Entwicklungsprozesses nachteilig auf eine kooperative Arbeit zwischen den Projektmitarbeitern auswirkt.“ Es gilt also die Kooperation zu unterstützen, aber nicht an stringente Vorgaben bis zum Projektende zu binden.

Für kooperative Entwicklungsumgebungen sind oftmals nur die Änderungen von Teilen von Dokumenten interessant [Haa96]. Bestimmte Evolutionshistorien beschäftigen sich nur mit

einzelnen Teilaspekten. Hierbei ist es wichtig, daß die Speicherungsstrategien den Erfordernissen der jeweiligen Aufgabe angepaßt sind. Ist z.B. nur eine bestimmte Passage oder ein Teilaspekt von Relevanz, so sollte es möglich sein, dessen Entwicklung gesondert zu verfolgen.

Wichtig für die Autonomie der Nutzer sind auch die verschiedenen Rollen der Anwender. Diese sollten klar definierbar sein und damit das Verantwortungsgefüge beschreiben. Es ist für den Benutzer auch wichtig aus verschiedenen Transparenzstufen wählen zu können. Nicht immer soll es jedem gestattet sein Einblick in alle Dokumentteile zu haben. Dies ist ein wichtiger Beitrag zur Autonomie, die eine ungestörte kreative Arbeit von einzelnen Dokumentteilen ermöglicht.

### 2.3.4 Mehrwertdienste

Zusätzlich zu den Kernfunktionalitäten lassen sich Mehrwertdienste implementieren, die den Anwender im Bereich der Versionenverwaltung weitergehend unterstützen. Wichtig hierbei ist, daß eine bestehende Verwaltungsumgebung Möglichkeiten zur Erweiterung bietet.

**Weiterverwertung der Metainformation:** Eine geeignete Auswahl der Metainformationen kann für die Arbeits- und Projektplanung eingesetzt werden. Aus der Betrachtung eines gesamten Projektstatus lassen sich beispielsweise Schlüsse über Effizienz oder Fehlentwicklungen ziehen.

Notifikationdienste sind ein weiterer Verwerter von Metainformationen [Thu00]. Hier werden Informationen nicht nur abgefragt, sondern können automatisch verarbeitet und übermittelt werden. Wird z.B. ein Objekt an dem ein Autor gearbeitet hat verändert, so ist er unter Umständen daran interessiert, sofort benachrichtigt zu werden, wenn an der Version weitergearbeitet wird. Hier ist eine wunschgerechte Aufbereitung und Präsentation dieser Information notwendig, um die Übersicht gewährleisten zu können. Nicht immer sind für alle Beteiligten alle Informationen von gleicher Relevanz. So kann es z. B. dem Anwender überlassen werden, wie und in welchem Umfang er über Vorkomnisse informiert werden möchte. Vielfach sollte aber eine Voreinstellung des Informationsdienstes ein unnötiges Aufkommen von nicht benötigter Information einschränken. Dies läßt sich beispielsweise über die Hierarchiestufen, gemäß der Bearbeitungsrechte, die sich aus den Teamstrukturen ergeben, regeln. In der Arbeit von Karsten Thurow [Thu00] finden sich genauere Ausführungen über die Zusammenhänge von Hierarchien und Kooperation.

**Selektive Transparenz:** Nicht jede Änderung eines Dokumentbestandteils sollte in dem nutzbaren Material (Versionsarchiv) festgehalten werden. Oft machen auch erst eine Reihe von Änderungen in einem bestimmten Kontext Sinn. Hier ist von Nöten, daß selektiert werden kann, welche Ausgangspunkte für ein Weiterarbeiten oder auch Zurückgehen von Vorteil sein können.

Vor allem ist die Flexibilität hinsichtlich der Berücksichtigung der spezifischen Teamstrukturen wichtig. Hierarchiestufen zu erkennen ist von elementarer Bedeutung und sollte auch in den Bereich der Bearbeitungsrechte eingebunden werden.

**Aufbereitung der Änderungen (Visualisierungswerkzeuge):** Im Hinblick auf kooperatives Arbeiten spielt die Aussagekraft der Unterschiede von Dokumenten eine große Rolle [NCK+92]. Hier muß ein möglichst repräsentiver Überblick über die Änderungen darge-

stellt werden können und auch kontextabhängig (z.B. auch für das Merging, siehe unten) zur Verfügung stehen. Editierfolgen (Deltas) sollten möglichst in einem Format generiert werden, welches eine Visualisierung unterstützt. Hier muß die Versionsverwaltung frühzeitig die Grundlagen für eine kooperative Unterstützung legen.

Einen weiteren Punkt stellt die Aufnahmefähigkeit der Benutzer dar. Auf der einen Seite soll keine Information verloren gehen, um einen genauen Überblick zu gewährleisten. Auf der anderen Seite muß darauf geachtet werden die Informationen so zu präsentieren, daß die bereitgestellte Information auch aufgenommen werden kann („Moreover, a reader with limited meta-cognitive capacity may fail to detect distraction or may reccover from the problem only after considerable misspent time.“ [NCK+92]). Muß die Aufmerksamkeit des Betrachters auf zwei Dokumente aufgeteilt und die Änderungsinformation selbstständig generiert werden, so kann es zu einer Überforderung kommen. Hier gilt es, die Dienste möglichst flexibel zu gestalten, damit anwenderspezifisch nachgeregelt werden kann. Neuwirth et al [NCK+92] stellen in ihrer Arbeit Hilfswerkzeuge zur kollaborativen Dokumentenentwicklung vor, standen aber immer vor dem Problem vorhandene Informationen umstrukturieren zu müssen. Eine möglichst standardisierte Ablage der benötigten Versionen erleichtert dagegen die Weiterverarbeitung, nicht nur im Falle der Visualisierung.

**Konflikthilfen und Merging:** Reiner Informationsfluß löst noch keine Konflikte. Diese entstehen, wenn Objekte parallel bearbeitet werden und archiviert werden sollen. Um die Datenpersistenz zu erhalten, werden neue Zweige eröffnet. Für ein gemeinsames Endergebnis müssen nun oft parallele Zweige wieder zusammengeführt werden. In Abbildung 2.5 ist als Beispiel die Bearbeitung eines Reiseprospekts dargestellt. Von der Urversion werden zwei neue Versionen geschaffen. Im oberen Zweig wurde der Text verändert und im unteren die Bildgestaltung. In diesem simplen Fall wäre bei geeigneten Strategien (vgl. [MD94]) sogar eine automatische Zusammenführung denkbar.

Diese parallelen Weiterentwicklungen zu einem kohärenten Bild zu verschmelzen, wird als Merging bezeichnet. In der Praxis geschieht das meist manuell oder semi-automatisch. Für ausführlichere Darstellungen sei auf [Jac01] verwiesen.

**Sozialwissenschaftliche Anforderungen:** Ein weiterer wichtiger Aspekt ist der sozialpsychologische Ansatz. In diversen Studien wurde beobachtet [Sha93], daß der Extraaufwand für Koordination und Kommunikation oft die angestrebten Vorteile der Arbeitsteilung übertrifft. Zusätzlich wirkt Druck, aufgrund von Gruppendynamiken und unklarem Verantwortungsgefüge bzw. Eigentumsansprüchen, kontraproduktiv. Gleichzeitig wurde aber auch nachgewiesen [FRK89], daß der Einsatz von automatisierten Autorensystemen viele dieser Faktoren berücksichtigt und somit (erfolgreich) zu entschärfen versucht.

Die Flexibilität der Verwaltung erhält somit eine grosse Bedeutung: Es gilt, gute Automatismen zur Verfügung zu stellen, wobei darauf zu achten ist, daß nicht mehr Arbeit generiert wird, als für den zu erwartenden Vorteil zu verantworten ist. Sharples [Sha93] zeigt auf, daß der Einsatz von komplizierten Werkzeugen (selbst nach Anwendungszeiträumen von über einem Jahr) oft an Frustrationsgrenzen scheitert.

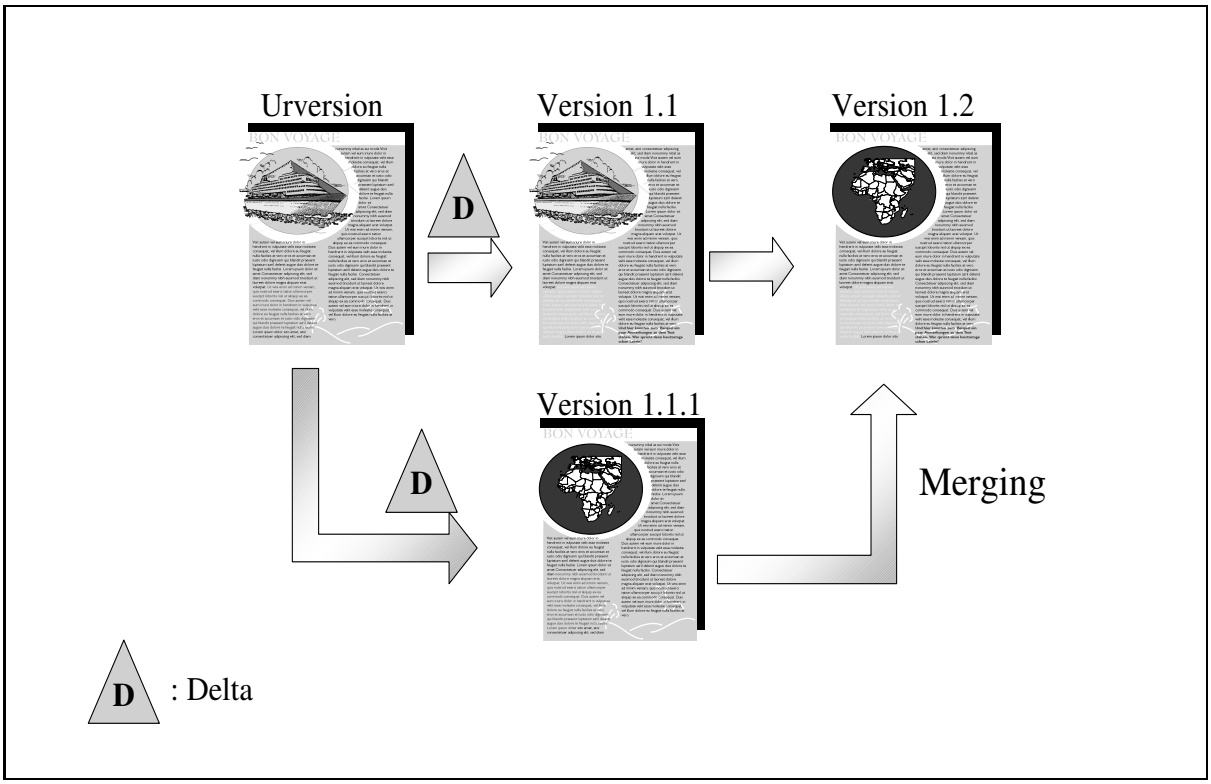


Abbildung 2.5: Darstellung eines Merging-Vorgangs anhand eines Dokuments mit verschiedenen Datentypen

## 2.4 Bewertungskriterien

In diesem Kapitel wurden Anforderungen für eine Versionsverwaltung zusammengetragen. Schwierig wird es, den Grad der Umsetzung dieser Anforderungen einer Bewertung zu unterziehen. Oft werden Laufzeiten- oder Speicheraufwandkriterien zu Rate gezogen. Bei der Vielzahl von möglichen Anwendungen innerhalb der Versionierung ist es aber schwierig, Datensätze zu finden, die zumindest für einen Teil der Anwendungen charakteristisch sind.

Ein weiteres Problem stellt die Betrachtung der Problemlösung aus verschiedenen Perspektiven dar. Die Kriterien aus Sicht der Automoie und der Kooperation sind teils überschneidend und teils auch inkohärent. Die Evaluierung selber gliedert sich in eine quantitative und eine qualitative Betrachtung.

### 2.4.1 Quantitative Evaluierung

Quantitative Bewertungen der Deltabildung messen zumeist Geschwindigkeit bzw. Komplexität des betrachteten Algorithmus und die Größe des resultierenden Deltas. Zwar kann die benötigte Zeit direkt gemessen werden, allerdings ist hier zu beachten, daß diese stark von der Performanz und Architektur abhängen. Auch die Größe der zu betrachtenden Dateien hängt von vielerlei Faktoren ab. Zum Beispiel: Wie groß ist ein Delta, wenn die Differenz der

betrachteten Dokumente klein ist? Wie kann die Größe der betrachteten Dateien in Relation gesetzt werden? Hunt [HVT98] bietet in seinem Artikel eine Metrik an, die in dem betrachtetem Fall nicht anwendbar ist. Sie geht von einer *Longest Common Sequence* aus, die als Referenz gewählt wird. Erste Versuche zeigten schon, daß in dem betrachteten Anwendungsfall mehrere längere gemeinsame Teilstücke in den Dateien enthalten waren. Weiterführend soll auch auf den inhaltlichen Kontext Rücksicht genommen werden, was auch in der Metrik nicht berücksichtigt wird.

Hunt weist allgemein darauf hin, daß „The first problem encountered when defining a benchmark [is] [...] finding an apppropriate data set that is both large enough for the results to be statistically significant and representative of real world applications.“. Dies kann in dieser Arbeit nicht geleistet werden. Der Algorithmus wird stattdessen an einem realen Fallbeispiel getestet.

Eine weiterer Evaluierungsansatz ist der Vergleich mit existierenden Werkzeugen, welche die gleichen Ziele verfolgen. Allerdings existieren im Bereich der strukturellen Vergleiche nur wenige Dienste, die semistrukturierte Dokumente behandeln und dabei auch die Strukturen berücksichtigen. Im Falle von HTML-Vergleichsalgorithmen (z.B. [Inc99]) zum Beispiel wird die limitierte Menge von Tags für die Laufzeitoptimierung der Algorithmen ausgenutzt. Somit sind HTML Vergleichswerkzeuge nicht mit solchen für XML Dokumente vergleichbar. Auch die Werkzeuge, die ein *Merging* für XML Strukturen anbieten und somit Vergleichstools (wie z.B. *xdiff* von Sudarshan S. Chawathe [Cha00]) integriert haben, benutzen meist Unix-*diff*, bevor die Dokumentstrukturen von weiterführenden Diensten wieder herausgearbeitet werden.

Der kommerzielle Markt für dergestalte Softwareprodukte scheint sich noch in der Aufbauphase zu befinden. Ein Projekt für die Entwicklung eines *xmldiff*-Werkzeugs bei IBM wurde nach längerer Forschung nicht fortgeführt [BSF99]. Von der Firma SCHEMA [Pre00] wurde für Januar 2001 ein neues XML Delta Modul angekündigt, welches aber zur Erstellung dieser Arbeit noch nicht zur Verfügung stand. Auch waren nähere Auskünfte bezüglich der Funktionsweise nicht erhältlich.

Eines der führenden Vergleichswerkzeuge *xdelta* [Mac00], ebenso wie *vdelta* [HVT98], ist nicht signifikant besser in der Performanz als Unix-*diff* (siehe Vergleichsstudie von Hunt [HVT98]). Dagegen sind diese Werkzeugen im Dokumentenbereich schwieriger mit dem hier entwickelten Vergleichswerkzeug in Beziehung zu stellen, da sie binäre Dateien verarbeiten. Sowohl bei *xdelta*, als auch bei *vdelta* werden die Editskripte bei der Erstellung der Deltas komprimiert, was die Lesbarkeit weiter erschwert, da die Dekomprimierung erst wieder bei der Wiederherstellung von Dateien zum Tragen kommt. Sollten Deltas selber auch genutzt werden können, so ist es sinnvoll Komprimierungsmechanismen unabhägig von der Deltaerstellung zu implementieren.

Für Unix-*diff* stehen Weiterentwicklungen zur Verfügung, die die textuelle Darstellung verbessern und Möglichkeiten zur Weiterverarbeitung bieten. Gerade unter der Prämisse, die erstellten Deltas weiterbenutzen zu können (siehe Kapitel 2.4.2), erscheint der Vergleich mit Unix-*diff* sinnvoll. Unix-*diff* untersucht Textdateien zeilenweise und unterstützt somit eine lesbare Ausgabe. Die resultierenden Daten können zeilenweise in einen Kontext eingeordnet werden. Ein dynamisches flexibles Anpassen an die Anwendung ist allerdings nicht möglich. Vorhandene Strukturen werden nicht beachtet oder genutzt.

### 2.4.2 Qualitative Evaluierung

| **Anforderung** | **Erfüllungsstatus** |
|---|---|
| Abwärtskompatibilität | nicht erfüllt |
| Aufwärtskompatibilität | nicht erfüllt |
| Versionen aller Dokumentbestandteile | nicht erfüllt |
| Zugriff auf Versionen über Attribute und Beziehungen | bedingt erfüllt |
| Zugriff auf Versionen über Entstehungszeitpunkt und Autoreninformation | erfüllt |
| Zusammenhängende Änderungen (Festhalten und Identifikation) | erfüllt |
| Wiedernutzung von Inhalten | bedingt erfüllt |

Tabelle 2.1: Anforderungsprofil nach [Haa96] (Stand:1996)

Da qualitative Ansprüche nicht direkt gemessen werden können, wird ein Kriterienkatalog zur Bewertung erstellt. In der Arbeit *Versionenunterstützung für strukturierte Hyperdokumente im elektronischen Publizieren* [Haa96] stellt Anja Haake einen Anforderungenkatalog auf. Um die Möglichkeiten der hier betrachteten Versionenverwaltung abzuschätzen, werden relevante Katalogpunkte extrahiert (siehe Tab. 2.1). Hierbei wird darauf geachtet, daß eine Bewertung hinsichtlich der einzelnen Punkte (Spalte eins in Tabelle 2.1) möglich ist, da Haakes Anforderungsprofil auf vollständige Versionierungsumgebungen konditioniert ist. Desweiteren wird versucht, auf die von ihr als „mangelhaft realisiert“ bezeichneten Aspekte besonderes Augenmerk zu legen. In der zweiten Spalte der Tabelle ist dargestellt, welche Anforderungen von den von Haake untersuchten Werkzeugen schon erfüllt bzw. nicht erfüllt werden. Zugriffe über Attribute wie z.B. den Autor gehören zwar schon zum Standard, wurden aber trotzdem in den Katalog aufgenommen. Gerade weil sie zum Standard gehören, muß auf die Erfüllung geachtet werden. Und desweiteren soll auch auf die Vorteile hingewiesen werden, die aus der Verwendung des XML-Formats ergeben, da hier besonders auf den Zugriff von Metadaten geachtet wird. In Kapitel 6.3.2 wird der implementierte Dienst unter den genannten Aspekten untersucht. Der Anforderungskatalog ist für vollständige Versionierungsumgebungen erstellt worden, allerdings lassen sich Aussagen über die Möglichkeiten des implementierten Dienstes unter den betrachteten Aspekten treffen.

# Kapitel 3

# Konzepte und Verfahren

In diesem Kapitel geht es darum, existierende Konzepte und Verfahren der Versionierung darzustellen. Die vorgestellten Verfahren enthalten Ansätze, die den Anforderungen in Kapitel 2 entsprechen und Alternativen aufzeigen. Aus den Ansätzen bzw. Limitierungen lassen sich Schlüsse für den in dieser Arbeit vorgestellten Ansatz einer Versionsverwaltung ziehen.

Im Folgenden wird eine Trennung in zustandsorientierte und änderungsorientierte Verfahren vorgenommen, um eine bessere Einordnung und Abgrenzung der verschiedenen Ansätze zu ermöglichen. Dies entspricht der Einteilung in der einschlägigen Literatur [Haa96] [CW98] und dient der besseren Orientierung. Im Vordergrund stehen allerdings, für den Entwurf (siehe Kapitel 5) interessante, Teilaspekte auf die näher eingegangen wird.

Im weiteren Verlauf wird im Detail auf Differenzenverfahren eingegangen, die die Grundlage für die meisten Speichermechanismen in der Versionsverwaltung bilden. Die Grundfeste dieser Verwaltungen sind Ablage, Suche und Extraktion der Versionen. Hier interessieren sowohl die Konzepte, als auch die verschiedenen Anwendungsmöglichkeiten. Aus den vorgestellten Konzepten sollen die Möglichkeiten klar werden, die sich im Bereich der Versionierung ergeben. Im Zusammenhang mit denen in Kapitel 2 vorgestellten Anforderungen wird in Kapitel 5 eine Versionsverwaltung entworfen werden.

## 3.1 Konzepte der Versionierung

Die Entwicklung im Bereich der Versionierungs- und Konfigurationsmanagementsysteme reicht bis in die frühen siebziger Jahre zurück und wurde anfangs von sehr pragmatischen Gesichtspunkten getragen. In den Anfangsjahren ging es darum, Werkzeuge zu schaffen, die in bestehende Umgebungen eingebunden werden konnten. Hier konnte es sich um Dateisysteme handeln oder auch um Datenbankumgebungen. Heutzutage werden immer mehr sogenannte Entwickungsumgebungen geschaffen, die eine Versionsverwaltung beinhalten. Die Funktionalität der einzelnen Ansätze wird stark von dem konkreten Anwendungsgebiet beeinflußt.

In der folgenden Vorstellung verschiedener Ansätze wird eine grobe Unterteilung in zustandsorientierte und änderungsorientierte Versionierung gemacht. Während änderungsorientierte Verfahren zur Versionenunterstützung die Erzeugung von Versionen komplexer Objekte aus

der Perspektive des Änderns des komplexen Objekts betrachten, betrachten zustandsorientierte Verfahren das Erzeugen von Versionen komplexer Objekte als Zusammenstellung- oder Konfigurationsprozeß von Komponentenversionen[1]. In Bezug auf semistrukturierte Dokumente sind beide Ansätze denkbar. Auf der einen Seite existieren Strukturen, die eine zustandsorientierte Versionierung begünstigen, auf der anderen Seite steht bei der kooperativen Dokumentenerstellung der Änderungsaspekt stark im Vordergrund.

### 3.1.1 Zustandsorientierte Versionierung

Zustandsorientierte Verfahren betrachten die Erzeugung von Versionen als einen Zusammenstellungsprozeß verschiedener Komponentenobjekte ([Haa96]: siehe Definitionen in Anhang E). Durch die Auswahl der Komponentenobjekte wird explizit eine Version erzeugt. Somit verändert das Austauschen eines Komponentenobjekts die Version des Gesamtobjekts. Der Unterschied zwischen zwei Versionen läßt sich durch die Änderung von Komponentenobjekten beschreiben. Hierbei kann zwischen statischen und dynamischen Referenzierungen unterschieden werden [CW98]: Bei einer statischen Referenz muß einer Version die neueste Version eines bereits eingebundenen Komponentenobjekts nach einer Veränderung manuell wieder zugewiesen werden. Im dynamischen Fall werden bei Änderungen die Referenzen automatisch auf die neueste Version umgesetzt.

Bei der zustandsorientierten Versionierung sind alle Komponentenobjekte eindeutig identifizierbar, ansprechbar und somit wiederverwendbar. Hier liegt allerdings auch die Limitierung: Verknüpfte Änderungen in unterschiedlichen Komponenten sind nicht in einer gemeinsamen Historie hinterlegt. Jedes Komponentenobjekt hat seine eigene Evolutionshistorie. Somit ist es schwierig zu erkennen, ob Zusammenhänge in der Evolution verschiedener Komponentenobjekte bestehen.

Die einzelnen Komponentenobjekte besitzen Attribute. In der zustandsorientierten Versionierung sind Funktionalitäten zum Erhalt von Versionen so konzipiert, daß diese Attribute für die Suche nach Versionen und die Extraktion derselben genutzt werden können. Die Möglichkeiten der Versionszusammenstellung sind vielfältig, stellen aber erhöhte Ansprüche an die Reglementierung der möglichen Konfigurationen. In technischen Anwendungen kann es z.B. sinnvoll sein, die gemeinsame Nutzung von Objekten in verschiedenen Artefakten (Produkten, Baureihen) einzuschränken. In rein logischen Zusammenhängen ist eine solche Limitierung nicht nötig. Hier können die Komponentenobjekte frei miteinander zu komplexen Objekten (Versionen) verknüpft werden. Allerdings wird die Vielfältigkeit mit einem Mangel an Übersicht erkauft. So ist es nicht einfach zu erkennen, welche Kompositionen wirklich Sinn machen und welche nicht.

Im Folgenden sollen nun einige ausgewählte Verfahren und Systeme vorgestellt werden, die die zustandsorientierte Versionierung verwenden.

**SPRITE** SPRITE [Hop92] wurde für die Unterstützung der Dokumentation technischer Enwicklungsprozesse entwickelt. Es koppelt seinen Versionsmechanismus an den planungstechnisch definierten Entwicklungszyklus der zu dokumentierenden Produkte. Die Zyklen

[1] Definitionen finden sich in Anhang E.

ergeben sich aus Erfahrungswerten, die in die Arbeitsplanung einfließen und für jedes Produkt individuell angepaßt werden.

Die Dokumente entstehen parallel zur Produktentwicklung und beschreiben die möglichen Produktvarianten und ihre Bestandteile. Ein Konfigurationsdokument, welches für das gesamte Produkt steht, besteht dabei aus verschiedenen sogenannten *Baustein*-Dokumenten (*building-blocks*). Diese können Varianten einer technischen Spezifikation eines Produktes symbolisieren, wie zum Beispiel Diesel- oder Ottomotor. Dabei können in den einzelnen *Baustein*-Teilen durchaus Referenzen auf weitere *Bausteine* gesetzt werden.

Die Dokumentation ist nicht nur für die technischen Spezifikationen von Belang, sondern zum Beispiel auch für Bedienungsanleitungen: Werden in einer neuen Produktionslinie neue Funktionalitäten eingebaut, so müssen diese in der gesamten Dokumentation ergänzt werden. Komplementär verhält es sich mit Charakteristiken, die nicht mehr Bestandteil eines Produkts sind. Gilt es, aus einer anderen Produktlinie Teile zu übernehmen, so kann einfach darauf verwiesen werden. Blöcke aus anderen Versionsarchiven können ohne Probleme eingebunden werden.

SPRITE arbeitet außerdem mit einer Multimedia-Datenbank, die es ermöglicht, die benötigten Texte und Graphiken gemeinsam abzulegen. Es wird Wert darauf gelegt, betrachtete Graphiken durch eine *scanning and recognition component* in ein einheitliches Format zu bringen. Dies ist inbesondere für die Speicherung von Deltas interessant.

Ein Browser ermöglicht es weiterhin entlang der Referenzen durch die einzelnen *Baustein*-Dokumente im Versionsarchiv zu navigieren.

**Hyperform** Das von Wiil und Legget [WL92] entwickelte System *Hyperform* ist ein objektorientiertes Werkzeug, welches Objektklassen für die Erstellung von Hypertexten anbietet. Die Versionen der einzelnen Hypertextobjekte werden in Baumstrukturen abgelegt. Für die Abspeicherung der einzelnen Objekte wird ein gerichtetes Deltaverfahren (siehe Abschnitt 5.3.3) verwendet.

Der Anwender kann aus den zur Verfügung stehenden Objekten in beliebiger Weise ein Hypertext erstellen. Dies bietet die Möglichkeit der kreativen Textzusammenstellung, liefert aber keine Hilfen, wenn z.B. ein Autor (autonomes Arbeiten) keine Vorkenntnisse über einen schon erstellten Text (kollaboratives Arbeiten) hat. Außerdem wird kein Linkmodell unterstützt.

In einer Nachfolgearbeit [WL93] wurde ein stärkeres Augenmerk darauf gelegt eine offene Plattform für ein Gruppe von zusammenarbeitenden Anwendern (*multiple collaborative users*) zu schaffen. *Hyperbase* konzentriert sich darauf, durch verbesserte Notifikation und neugestaltete Sperrmechanismen die Zusammenarbeit zu unterstützen, im Gegensatz zur Unterstützung von Mehrteilnehmersystem (*multi-user system*), die ihr Augenmerk auf die benutzerfreundlichen Zugriffsrechte legen. Ein betonter Aspekt ist hierbei „[the] versioning of both the data and the structure“. Dies erscheint insofern sinnvoll, als daß die Strukturen auch einen Beitrag zur (im kollaborativem Kontext) Übersicht leisten.

**Generischer Ansatz** Ahmed und Navathe [AN91] beschäftigen sich vornehmlich mit Klassifikation und Typisierung von Objekteigenschaften und dem Problem der Versionenprolife-

ration. Das entworfenen Modell ist ein fein differenziertes Objektmodell, welches nicht versionierbare, generische und versionierbare Objekte unterscheidet. Ein generisches Objekt faßt die gemeinsamen Eigenschaften aller Versionen eines versionierten Designobjekts in ein Objekt zusammen und stellt damit sicher, daß alle Versionen des versionierten Desingnobjekts die gleichen Werte bezüglich dieser invarianten Egenschaften aufweisen. Versionen entsprechen versionierbaren Objekten, die die zustandsabhängigen (varianten) Eigenschaften des versionierten Designobjektes beschreiben

Das generische Objekt umfaßt alle Versionen des versionierten Designobjekts, wobei die Versionsmenge durch gerichtete azyklische Versionsgraphen strukturiert werden kann.

Zur Erzeugung neuer Versionen unterscheiden Ahmed und Navathe [AN91] noch einmal in invariante, versionenrelevante und versionenirrelevante Eigenschaften: Invariante Eigenschaften sind Schnittstellen und versionenirrelevante Eigenschaften haben meist beschreibenden Charakter. Nur wenn versionenrelevante Eigenschaften geändert werden, führt dies automatisch zur Generierung einer neuen Version.

Die Neuschaffung von Versionen bei jeder Änderung führt meist zu einer ungewollten Proliferation, die das Versionsarchiv schnell unübersichtlich macht. Um dies in den Griff zu bekommen haben [Kat90] einen sogenannten *group-check-in*-Operator vorgeschlagen, der erst eine neue Version erzeugt, wenn durch die Änderung einer Gruppe von Objekten ein neuer Entwicklungsstand erreicht wird. Dies ist in der Idee der zustandsorientierten Versionierung, wie zu Anfang des Kapitels erwähnt, nicht vorgesehen, sondern schon im Bereich der änderungsorientierten Versionierung anzusiedeln.

### 3.1.2 Änderungsorientierte Versionierung

Im Gegensatz zur zustandsorientierten Versionierung, die Versionen von Komponentenobjekten betrachtet, beschäftigen sich änderungsorientierte Verfahren mit der Erzeugung von Versionen aus der Sicht der Änderungen eines komplexen Objekts. Ausgegangen wird von einer Urversion, von der alle Änderungen abgeleitet werden. Hiermit ergibt sich eine hierarchische Struktur. Änderungen zwischen Versionen werden explizit durch Deltas angegeben, die die Unterschiede zwischen zwei Versionen beschreiben.

Charakteristisch für das änderungsorientierte Modell ist, daß die Versionen meist nicht persistent vorliegen, sondern als Änderungsbeschreibung bezüglich einer anderen Version gespeichert sind und somit erst berechnet werden müssen. Im Folgenden werden einige für diese Arbeit relevante Beispiele vorgestellt.

**Personal Information Environment (PIE)** Das von [BG81] vorgestellte PIE unterteilt die Versionscharakteristika in Schichten und Kontexte. Schichten repräsentieren eine Menge von Änderungen, die sich über verschiedene Objekte erstrecken können. Diese Schichten beziehen sich immer auf eine benachbarte Version und sind somit dem hierarchischen Prinzip unterworfen. Dies hat den Vorteil, daß die einzelnen Knoten in einem semistrukturierten Dokument nicht unabhängig voneinander betrachtet werden müssen, sondern zusammenhängende Knotenteile in einer Änderung zusammengefaßt werden können.

Kontexte sind Folgen von Schichten, die quasi eine Reihe von Änderungen beinhalten. Hierbei

ist es nicht notwendig, daß die Entstehungsreihenfolge der Änderungen eingehalten wird. PIE wurde zum ersten Mal von Vassili Prevelakis [Pre90] auf Hypertextdokumente angewendet. Es bietet die Möglichkeit Textstrukturen flexibel zu versionieren und auch die Änderungen in anderen Kontexten zusammenzufassen und weiterzuverwenden. Das Ziel der Arbeit war es neue Möglichkeiten zu finden die kollaborative Arbeit zu unterstützen („[...] providing versioning facilities that not only support but enhance the special features provided by hypertext."). Auf Möglickeiten die sich aus solch flexibler Versionierung ergeben wird im folgenden noch eingegangen werden.

**Interleaf** Interleaf [Haa96] stellt die Verfolgung der Änderungen in den Vordergrund. Die betrachteten Dokumente können aus Texten, Graphiken oder Bildern bestehen. Der Delta-Algorithmus steht vergleicht komplette semistrukturierte Dokumente und beschreibt deren Änderungen. Abgespeichert werden die Unterschiede (Deltas) und in einem hierarchisch aufgebauten Baum abgelegt. Die Form der Deltas ist ein proprietäres Format der Firma *Interleaf*. Das Produkt selber wird nicht mehr *supported*.

Um die Änderungen explizit nutzen zu können, besteht die Möglichkeit, diese zu visualisieren. Interleaf versucht so, die kollaborativ arbeitenden Autorenteams zu unterstützen. Durch verbessertes Änderungsmanagement soll eine effektivere Zusammenarbeit gewährleistet werden.

**SCCS und RCS** SCCS[2] verwaltet Versionen von Textdateien. Die Versionen der Dateien werden in Baumstrukturen abgelegt. Soll an einer Version gearbeitet werden, wird sie in eine Arbeitsumgebung kopiert und später als neue Version wieder archiviert. SCCS in der Urform benutzt Vorwärtsdeltas (siehe Kapitel 2).

RCS[3] ist eine Unix-basierte zeilenorientierte Versionierung mit kooperativen Diensten. Der Standard, von Walter F. Tichy [Tic85] Anfang der achtziger Jahre entwickelt, stellt den Nachfolger von SCCS dar.

Für die Hauptversionswege werden Rückwärtsdeltas verwendet und die Attributliste der Versionen ist gegenüber SCCS erweitert. Diese können sogar für die Suche nach älteren Versionen verwendet werden und bieten somit flexiblere *Retrieval*-Mechanismen.

## 3.2 Konzepte der Differenzenspeicherung

Die persistente Speicherung der Versionen ist eine Kernfunktionalität der Versionsverwaltung. Die komplette Speicherung jeder Version besitzt eine geringe Speichereffizienz, da viel redundante Information abgelegt wird.

Als Alternative zur kompletten Abspeicherung greift man darauf zurück, nur die Unterschiede zwischen einzelnen Versionen abzulegen. Diese abgelegten Differenzen werden als *Deltas* bezeichnet. Handelt es sich um Editieranweisungen zur Überführung von einer in die andere Version, so ist auch der Ausdruck Editskript gebräuchlich. Im nachfolgenden Abschnitt sollen zunächst einige grundlegende Ansätze erläutert werden.

[2] SourceCode Control System
[3] Revision Control System

### 3.2.1 Ansätze zur Differenzenspeicherung

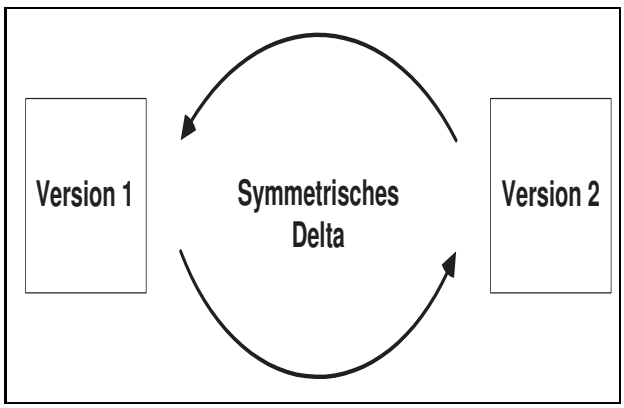


Abbildung 3.1: Symmetrische Deltas

Fast alle Konzepte im Versionsmanagement basieren auf einer Abspeicherung von Deltas zwischen den einzelnen Versionen. Allerdings existieren hier verschiedene Ansätze. Bestehende Verfahren [CW98] unterscheiden zwischen:

- Symmetrischen Deltas,
- Gerichteten Deltas,
- und eingebetteten (*embedded*) Deltas.

Symmetrische Deltas (siehe Abb. 3.1) haben keine Präferenzausrichtung und beschreiben genau die Unterschiede zwischen zwei Dokumenten. Sie können in beide Richtungen angewandt werden. Gerichtete Deltas (siehe Abb. 3.2) enthalten eine Editierfolge (Editskript), die ausgeführt ein Dokument in das andere überführt.

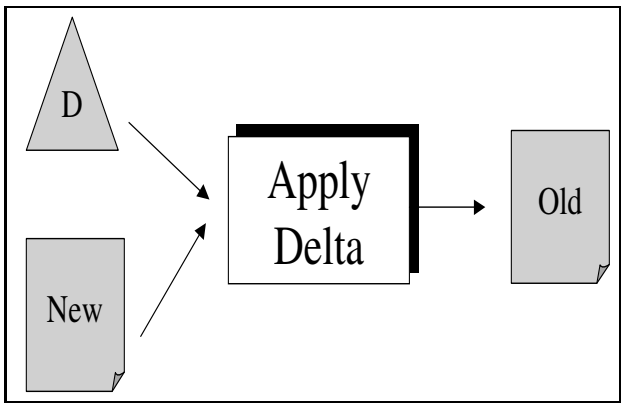


Abbildung 3.2: Gerichtete Deltas [Rei94]

Eingebettete Deltas wiederum beruhen auf dem Prinzip, daß alle Versionen in einer überlappenden Art und Weise (kann auch eine Dokumentenstruktur sein) abgespeichert sind. Eine

einzelne Version setzt sich aus einer Menge von Fragmenten zusammen. Die Menge dieser Fragmente wird benötigt, um eine komplette Version zu erhalten. Dies kann entweder durch Zeigerstrukturen realisiert werden, die auf die benötigten Fragmente verweisen, oder durch Kontrollausdrücke mit denen die jeweiligen Fragmente versehen sind. Diese machen deutlich, welche Teile für eine bestimmte Version benötigt werden. Sie fungieren quasi als An/Aus-Schalter. Bei diesem Ansatz sind sind also alle Deltas in ein Dokument eingebettet. Der Nachteil hier ist, daß Evolutionshistorien extra erstellt werden müssen.

### 3.2.2 Wiederherstellung von Versionen

Liegen die abgespeicherten Versionen nicht explizit vor, wie es bei der Deltaspeicherung der Fall ist, obliegt es der Versionsverwaltung, für eine Wiederherstellung kompletter Versionen zu sorgen. Hierbei spielt es eine Rolle, in welcher Art und Weise die Deltas zur Abspeicherung genutzt werden. Auch hier gibt es verschiedene Konzepte:

- Vorwärtsdeltas,
- Rückwärtsdeltas,
- Strategisch plazierte Vollversionen.

Die einfachste Technik ist das (gerichtete) Vorwärtsdelta, d.h. die Differenz zwischen zwei Files wird so gebildet, daß das Delta angewandt werden muß, um die alte Version in die neue zu überführen. Als Vollversion wird also nur die erste Version abgespeichert, da alle andern inkrementell erhalten werden können. Man kann allerdings auch alle Deltas von einer Version bilden, womit man vermeidet mehrere Deltas hintereinander ausführen zu müssen, allerdings geht das zu Lasten der Größe der Deltas.

Das Prinzip der Rückwärtsdeltas verläuft komplementär: Hier ist die neueste Version als Vollversion abgelegt und kann sofort extrahiert werden, während ältere Dokumente erst durch Anwendung von Deltas generiert werden müssen. In der Vielzahl der Anwendungen sind aktuelle Versionen weit häufiger gefragt als ältere Versionen, weshalb sich die Variante der Rückwärtsdeltas größerer Beliebtheit erfreut. Meist werden jedoch Hybridlösungen angewendet, um eine größtmögliche Effizienz zu erreichen.

Eine wichtige zusätzliche Option stellt die Plazierung von Vollversionen an strategisch wichtigen Punkten dar. Der Grundgedanke hierbei ist, die Zahl der auszuführenden Deltas zu reduzieren. Ein Beispiel ist die Ablage von Vollversionen an Verzweigungspunkten.

Die Effizienz von Deltas kann somit in zwei Kategorien gemessen werden, die miteinander konkurrieren:

1. Schnellstmöglicher Erhalt der verlangten Version.
2. Speichereffizienz.

Aufgrund der vielfältigen Anwendungsgebiete mit charakteristischen Anforderungen fällt es jedoch schwer allgemeingültige Vergleichsmaßstäbe (*benchmarks*) zu finden. Zusätzliche Anforderungen ergeben sich, wenn Handhabbarkeitskriterien der Deltas bewertet werden sollen

(siehe Kapitel 2.4.2). Wie in Kap. 3.1.2 besprochen, legen einige Ansätze schon Prioritäten jenseits der Effizienz, z.B. Interleaf auf verbesserte Visualisierung von Änderungen in den zugehörigen Kontexten. Dies stellt eine wichtige Unterstützung in den Bereichen der kollaborativen Dokumentenerstellung dar. Allerdings ist eine vergleichende Bewertung in diesen Bereichen ziemlich schwierig.

### 3.2.3 Implementierungsansätze

In der Deltabildung existieren verschiedene Ansätze. Im Folgenden werden die gängigsten Ideen vorgestellt. Nach einer Empfehlung des World Wide Web Consortiums [vHP00] werden für die Erstellung von Deltas nur Kopier- und Anfügebefehle verwendet. Das Ziel der Deltaerstellung ist es möglichst viele Kopierbefehle zu verwenden. Ein Kopier-Befehl benötigt weniger Speicherplatz im Delta und ist bei der Überführung von einer Version zur anderen schneller auszuführen. Kopierbefehle entsprechen also dem Prinzip keine redundanten Daten abzuspeichern. Anfügebefehle enthalten dagegen die Unterschiede zwischen zwei Dateien und müssen diese neuen Inhalte mit abspeichern.

Im Folgenden sind gängige Delta-Ansätze qualitativ beschrieben. Meist unterscheiden sich Neuentwicklungen nicht wesentlich in der Grundidee, sondern in der konkreten Implementierung (siehe z.B. [HVT98]).

**Unix-*diff*** Unix-*diff* [Tic84] ist der wohl am weitesten verbreitete Textvergleichsalgorithmus. *diff* vergleicht zwei Files zeilenweise und produziert eine Liste mit den Zeilen die geändert, eingefügt oder gelöscht werden müssen. Unix stellt noch ein weiteres Programm (*patch*) zur Verfügung, das die Überführung des einen Files mit Hilfe des entstandenen Deltas in das zweite ermöglicht. Die erhaltenen Deltas sind nur in einer Richtung anwendbar, d.h. es handelt sich um gerichtete Deltas.

**Longest Common Sequence (LCS)** Diese Methode beruht darauf, möglichst lange Übereinstimmungen in zwei Texten zu identifizieren. Als eine der ersten Implementationen arbeitet RCS [Tic84] (siehe Abschnitt 3.1.2) mit einem solchen Ansatz. Um möglichst viele Kopierbefehle zu erhalten, wird nicht zeilenweise verglichen, sondern nach den längsten Übereinstimmungen gesucht. Kopiervorgänge wirken sowohl kapazitätensparend, als auch deltaverkürzend. Es ist von Vorteil eine Mindestlänge vorzugeben. Zum einen müssen weniger Entitäten miteinander verglichen werden und zum anderen können Änderungen in einen Kontext gestellt werden. Dies macht die Deltas zwar etwas länger, erleichtert aber die Lesbarkeit. Die meisten LCS Algorithmen arbeiten nur mit den beiden Operationen *append* und *copy*, womit sie in die Kategorie gerichtete Deltas einzuordnen sind.

**Heaviest Common Sequence (HCS)** Die Suche nach der am stärksten zu gewichtenden Übereinstimmung ist eine Weiterentwicklung der LCS-Methodik [Che00]. Hier wird davon ausgegangen, daß es sich um Texte handelt, die nach inhaltlicher Zusammengehörigkeit gewichtet werden können. Somit wird die absolute Länge der Gemeinsamkeiten durch die syntaktisch erweiterte Länge ersetzt. Während die LCS-Methode alle Entitäten gleich wichtet, bewertet die HCS-Methode die Bedeutung von Worten in Zusammenhängen. Beispielsweise

haben Verben und Nomen eine größere Wichtung als Konjunktionen oder Artikel. Hierzu ist eine Bewertungsbibliothek notwendig[4]. HCS bietet die Möglichkeit Änderungen gezielter in einen inhaltlichen Kontext einzubinden. In dieser Hinsicht verspricht HCS genau die von Neuwith et al [NCK+92] (siehe auch Abschnitt 2.3.4) geforderten Unterstützungen in der Kollaboration schon in der Deltaerstellung, da Repräsentation der Änderungen nicht erst noch kontextuell aufgearbeitet werden muß. Für ein detailliertes Beispiel sei auf [TCHN99] verwiesen.

[4] Aus diesem Grund findet der Algorithmus in dieser Arbeit keine Anwendung.

# Kapitel 4

# Verwendete Sprachen und Werkzeuge

Nach der Einführung in die grundsätzliche Problematik der Versionsverwaltung und der Vorstellung einiger existierender Ansätze, werden in diesem Kapitel die im weiteren Verlauf der Arbeit verwendeten Sprachen und Werkzeuge erläutert. In Kapitel 6 werden diese verwendet, um einen Teil einer Versionenverwaltung zu implementieren und zu evaluieren.

## 4.1 XML

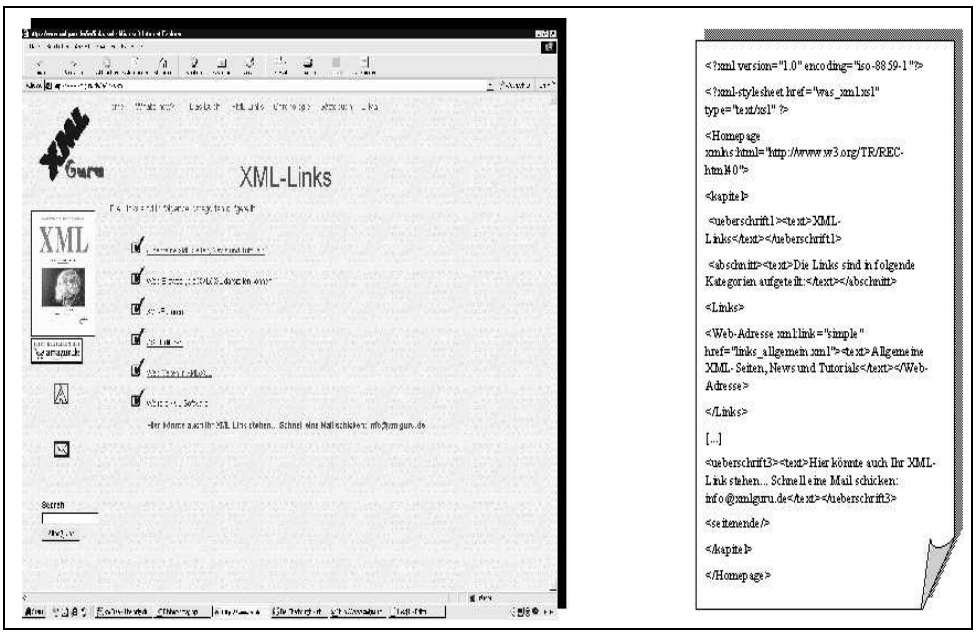


Abbildung 4.1: Aktuelles Beispiel eines XML Dokuments mit Webeinbindung

In dieser Arbeit wird als Dokumentenstandard XML (eXtensible Markup Language) betrach-

tet, der sich durch seine universelle und selbstbeschreibende Art und Erweiterbarkeit auszeichnet. XML wurde vom W3C [Con00] als Standard empfohlen, mit dem sich eine Web-Auszeichnungssprache anwendungsspezifisch definieren läßt. Er stellt eine eingeschränkte Variante der Standard Generalized Markup Language (SGML) dar [KPM99]. XML stellt also eine geeignete Möglichkeit dar, hierarchische, semistrukturierte Dokumente zu beschreiben.

Aus dem Internet ist als Beschreibungssprache HTML bekannt, welche auch von SGML abgeleitet ist, aber im Vergleich zu XML einen geringeren Struktur- und Flexibilitätsgrad besitzt. XML enthält eine Metasprache, welche in der Lage ist, logische Strukturen sehr unterschiedlicher Daten abhängig von den Anforderungen der jeweiligen Anwendungsdomäne zu beschreiben.

XML ist darüberhinaus als Standard für den Datenaustausch etabliert. Die Möglichkeit Inhalte zu strukturieren erhöht die Übersicht. Dies vereinfacht die automatische Weiterbearbeitung (z.B. Sortierung oder Ablage). Mehrwertdienste profitieren von der Plattformunabhängigkeit und der Offenheit des Standards. Gezielte Informationssuche wird durch die Meta-Attributierung begünstigt. Standards wie „Topic Maps“[Emp99] oder „RDF[1]“[W3C00] verwenden XML und dienen zu besserem Zugriff sowie besserer Modellierung von Informationsbeständen.

In ihrer Arbeit über die Klassifizierung von Versionierungsmodellen sagen Conradi und Westfechtel [CW98]: „In the future, we expect that more and more [...] systems will be built with the help of database technology. Having gained a better understanding of version models, versioning can be pulled out of SCM [Software Configuration Model] systems and moved into database systems. In particular, [...] versioning will benefit from the powerful facilities of an underlying deductive system.“ Für XML stehen schon Datenbanken (z.B. Tamino [AG00]) zur Verfügung und auch Conklin [Con87] postuliert, daß signifikante Verbesserungen in den Konzepten erwartet werden, wenn als Datenform semistrukturierte Dokumente verwendet werden. Neben Datenbanken existieren noch weitere Anwendungen wie das *Document Object Model* (DOM), welches eine plattform- und sprachunabhängige Schnittstelle darstellt, die es Programmen und Skriptsprachen erlaubt, dynamisch auf den Inhalt, die Struktur und das Layout eines XML Dokumentes zuzugreifen und zu verändern. Besonders im Rahmen von *Electronic Commerce* Anwendungen findet DOM starken Einsatz. Gerade die weite Verbreitung und Nutzung in vielen Bereichen der Informationstechnologie machen XML zu einem attraktiven Standard, weshalb XML in dieser Arbeit zur Beschreibung semistrukturierter Dokumente gewählt wurde.

## 4.2 Java

Bei der Frage nach der Implementierung der Versionsverwaltung gibt es einige Dinge zu beachten. Auf der einen Seite soll ein offener Standard gewählt werden, der die Weiterentwicklung und Erweiterung durch Mehrwertdienste erleichtert, auf der anderen Seite soll eine Unabhängigkeit von der Plattform gewährleistet sein.

Aus folgenden Überlegungen wird die objekt-orientierte Programmiersprache Java [Sun00] gewählt:

[1] *Ressource Description Frameworks*

- Java ist frei verfügbar und plattformunabhängig.
- Für das Parsen von XML Dokumenten gibt es bereits frei verfügbare Dienste, die auch in dieser Arbeit verwendet werden.
- Für die Erweiterung bietet Java als objektorientierte Sprache große Vorteile, gegenüber nicht objekt-orientierten Sprachen.

Gerade der letzte Punkt liefert genügend ausschlaggebende Gründe: Der in dieser Arbeit verwendete XML - Vergleichsalgorithmus liegt schon in Java vor und muß nur noch angepaßt werden. Diese Anpassung bzw. Erweiterungen können in Java implementiert werden. Um zukünftigen Entwicklern die Möglichkeit zu geben in ebensolcher Weise weiterzuverfahren, wird auch der in dieser Arbeit implementierte Vergleichsalgorithmus in Java geschrieben.

Zwar resultiert die Implementation, eines Algorithmus im Speziellen, in Java in höheren Laufzeiten, aber dieser Nachteil wird bei der Entwicklung eines Prototypen durch die Möglichkeiten der leichten Erweiterbarkeit aufgewogen.

Die in dieser Arbeit verwendete Baumvergleichsumgebung XTree [Spr00] verwendet eine graphisch orientierte Benutzeroberfläche, die die im JDK[2] 1.2 enthaltene Klassenbibliothek *SWING* nutzt. Die bestehende Oberfläche wird weitestgehend weitergenutzt, wobei der implementierte Dienst auch ohne graphische Oberfläche nutzbar ist.

## 4.3 XTree

XTree ist ein von Christoph Spreen [Spr00] am Arbeitsbereich Softwaresysteme in Java entwickeltes Werkzeug, welches den Kern eines Dienstes zum Änderungsmanagement darstellt. XTree ist ein generischer Dienst zum strukturorientierten Vergleich semistrukturierter Daten, die als XML Dokumente vorliegen.

Zwei XML Dokumente können miteinander verglichen werden und das Resultat wird in einem Änderungsprotokoll abgelegt. Dieses kann als ein Delta (Editskript) interpretiert werden, da es die Informationen enthält, wie ein Dokument in das andere überführt werden kann.

Die Besonderheit von XTree liegt im erweiterten Operationenrepertoire. Durch diese höhere Auflösung hinsichtliche der Änderungsbeschreibung, lassen sich auch gezieltere Aussagen über die Evolutionshistorie machen.

Die Funktionalität Dokumenten durch Anwendung von Deltas wiederherzustellen muß noch implementiert werden. Desweiteren beschreibt XTree XML Dokumente als Baumstrukturen und führt sämtliche Operationen auf dieser Klasse durch. Um nach Anwendung eines Deltas wieder ein XML Dokument zu erhalten, muß auch noch ein *XMLDocumentWriter* implementiert werden. Weitere Details der Implementierung finden sich in Kapitel 6.

[2] Java TM Development Kit

# Kapitel 5

# Entwurf einer Versionsverwaltung

In Kapitel 2 werden Kriterien, die an eine Versionsverwaltung zu stellen sind, zusammengetragen. In Kapitel 3 sind die existierenden Ansätze aufgezeigt worden. Diese erfüllen teilweise die gestellten Ansprüche und geben desweiteren Anregungen für Ergänzungen oder Implementierungsansätze. In diesem Kapitel wird ausgehend von dieser Betrachtung ein Entwurf gestaltet. Der Entwurf bildet die Grundlage für die Implementierung von Diensten zur Versionsverwaltung in dieser Arbeit, die in Kapitel 6 beschrieben wird.

## 5.1 Allgemeine Kriterien für die geplante Anwendung

Eine Versionsverwaltung teilt sich in verschiedene Komponenten auf. In Abbildung 5.1 werden die wichtigsten Komponenten und deren Zusammenhang aufgezeigt. Hierbei ist insbesondere die Modularität der einzelnen Komponenten wichtig. Es soll möglich sein, je nach Anwendung einzelne Komponenten anzupassen oder auszutauschen. In diesem Abschnitt wird auf die Grundfunktionalitäten der einzelnen Blöcke eingegangen.

Die eigentliche Speicherung der einzelnen Versionen wird von einem Kerndienst übernommen, der sowohl mit einer Datenbank, als auch mit portablen Speicher (siehe Abbildung 5.4) arbeiten kann. Von Vorteil ist das einheitliche Format von Dokumenten und Deltas (XML). Der eigentlich Aufbau des Versionsarchivs wird von der Deltaverarbeitung verwaltet. Soll eine Version extrahiert werden, so wird die Datei geladen und gegebenenfalls mit Hilfe einer Vollversion (bzw. weiterer Deltas) generiert.

Die Metadatenebene verwaltet die Metadaten der einzelnen Dokumente. Außerdem ist es möglich Metadaten zusammenzufassen und mit Änderungsinformationen anzureichern.

Die Navigation und Suchunterstützung hilft dem Anwender bei der Suche nach einer bestimmten Version. Hierbei wird stark auf die Unterstützung der Metadatenebene zurückgegriffen, um Extraktionsprozesse in ihrer Zahl zu minimieren.

Evolutionshistorien-, Notifikations- und Mergingdienste sind Mehrwertdienste, die vom vorgestellten Aufbau einer Versionsverwaltung profitieren.

Sicherheitsaspekte werden in dieser Arbeit nicht berücksichtigt. Die Ausrichtung des Mo-

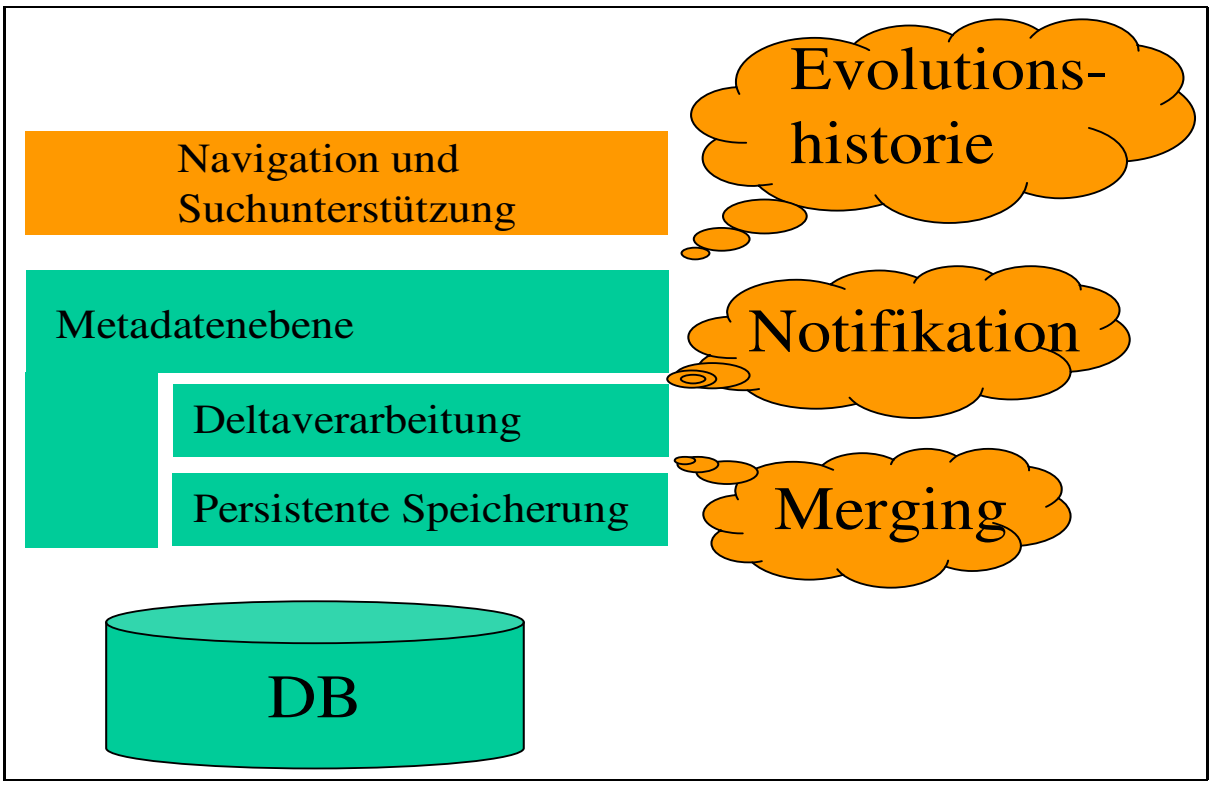


Abbildung 5.1: Die Darstellung einer Versionsverwaltung unter Berücksichtigung der verschiedenen Komponenten

dells ermöglicht aber Erweiterungen in dieser Richtung. In den weiteren Abschnitten werden die Teile Abspeicherung, Metadatenebene, Navigierbarkeit und Mehrwertdienste genauer beschrieben. Weiterhin wird auf die Punkte Flexibilität und Mehrwertdienste eingegangen.

### 5.1.1 Abspeicherung

Eines der Hauptkriterien der Versionierung ist die persistente Speicherung. Herauszustreichen ist hierbei, daß die automatische Speicherung von Änderungen nicht in den Bereich der Versionsverwaltung fallen, sondern Aufgabe der lokalen Arbeitsumgebungen ist. Diese Umgebungen unterstützen den Anwender bei der Bearbeitung des Dokuments. Die Archivierung einer geänderten Version muß immer vom Anwender oder der Anwendung selbst initiiert werden. Der persistenten Speicherung ist die Deltaverarbeitung angeschlossen. Diese ist sowohl für die Deltagenerierung, als auch für die Wiederherstellung von Dokumenten zuständig. Sie greift zur Speicherung der Deltas und der strategischen Vollversionen (siehe Abschnitt 3.2.2) auf die Dienste zur persistenten Speicherung zurück.

Somit stellt die Deltaverarbeitung die Zugriffsebene auf das Versionsarchiv dar. Ob die Speicherung in einer Datenbank oder einem (portablen) Filesystem (siehe Abbildung 5.4) geschieht, ist für den Anwender nicht einzusehen. Vorteilhaft für die Organisation der Speiche-

rung ist es, wenn alle Dateien über das gleiche Format verfügen (siehe auch Abschnitt 3.1.1). Von der Deltaverarbeitung wird eine weitere Verantwortung übernommen: Die Verwaltung der getätigten Änderungen. Einmal abgespeicherte Änderungen dürfen nicht verloren gehen, und somit ist ein Überschreiben auszuschließen. Das Archivieren einer geänderten Version muß immer die Generation eines neuen Identifizierers nach sich ziehen.

Um die Abspeicherung redundanter Daten zu vermeiden, werden zumeist nur die Änderungen (Deltas) abgespeichert. Daraus ergibt sich, daß ein nicht komplett abgespeichertes Dokument zur Extraktion erst generiert werden muß. Dies ist bei der Extraktion einer älteren Version zwar zeitintensiver, spart aber Speicherressourcen. Ein weiterer Vorteil der Deltas ist die Möglichkeit der Weiterverwendung von Änderungsinformationen für zusätzliche Dienste.

Wie Haake [Haa96] in ihrer Arbeit postuliert, ist eine „wesentliche Tätigkeit beim Überarbeiten von Dokumenten [...] das Vergleichen von Dokumentstukturen und -inhalten der verschiedenen Versionen." Gemäß dieser Maxime geschieht die Abspeicherung in dem vorgestellten Konzept in zwei Stufen (nach [WL92], Kapitel 3):

1. Deltabildung für die Dokumentstrukturen.
2. Deltabildung der Dokumenteninhalte.

Zusätzlich zur Strukturierung der Dokumente, werden auch die textuellen Inhalte der Dokumente vor der Deltabildung vorstrukturiert. Diese Vorstrukturierung ist von ihrer Granularität (Zeichenketten verschiedener Länge) und Form (Worte, Sätze, Absätze etc.) her flexibel und kann an die Arbeits- und Projektplanung angepaßt werden. So können für verschiedene Situationen im selben Projekt durchaus verschiedene Vorstrukturierungen von Vorteil sein. Diese können vom Benutzer eingestellt werden. Die Einteilung der Inhalte in die gewählten Entitäten ist auch mit einer Aussage über den Grad der Gleichverteilung des Textes verbunden. Diese sind ein Parameter für den zu erwartenden Aufwand der Deltabildung.

Die hier betrachteten Deltas sind sogenannte gerichtete Deltas, d.h. sie enthalten die Information wie aus einer Vollversion eine andere generiert werden kann. Da die meisten Zugriffe auf jüngere Versionen erfolgen[1], erscheint es sinnvoll im Hauptstrang mit sogenannten Rückwärtsdeltas (siehe auch Kapitel 3) zu operieren. In dieser Form liegt immer die neueste Version als Vollversion vor und ältere müssen generiert werden.

Mit einem wachsenden Versionsarchiv wird der sukzessive Wiederherstellungsprozeß neuerer Versionen immer aufwendiger. Somit ist es günstiger Vollversionen an strategisch wichtigen Punkten, wie Verzweigungspunkten oder Strangenden zu plazieren, um lange Wiederherstellungsprozesse zu vermeiden.

In dieser Arbeit steht die Erstellung von Dokumenten im Vordergrund. Somit wird auf repräsentative Änderungsprotokolle und einen möglichst schnellen Erhalt aktueller Versionen Wert gelegt. Aus dieser Überlegung heraus erscheinen Rückwärtsdeltas sinnvoll. Sie haben sowohl Geschwindigkeitsvorteile bei der Extraktion neuerer Versionen (liegen als Vollversionen vor), als auch bei der Archivierung (da neu-archivierte Versionen als Vollversionen abgelegt werden und im Hintergrund eine ältere Version als Delta „umgespeichert" wird). Weitere Plazierung von Vollversionen und Vermeidung von inkrementellen Wiederherstellungsprozessen

[1] Laut [K"o92] beziehen sich 95 Prozent aller Zugriffe auf zuletzt archivierte Version.

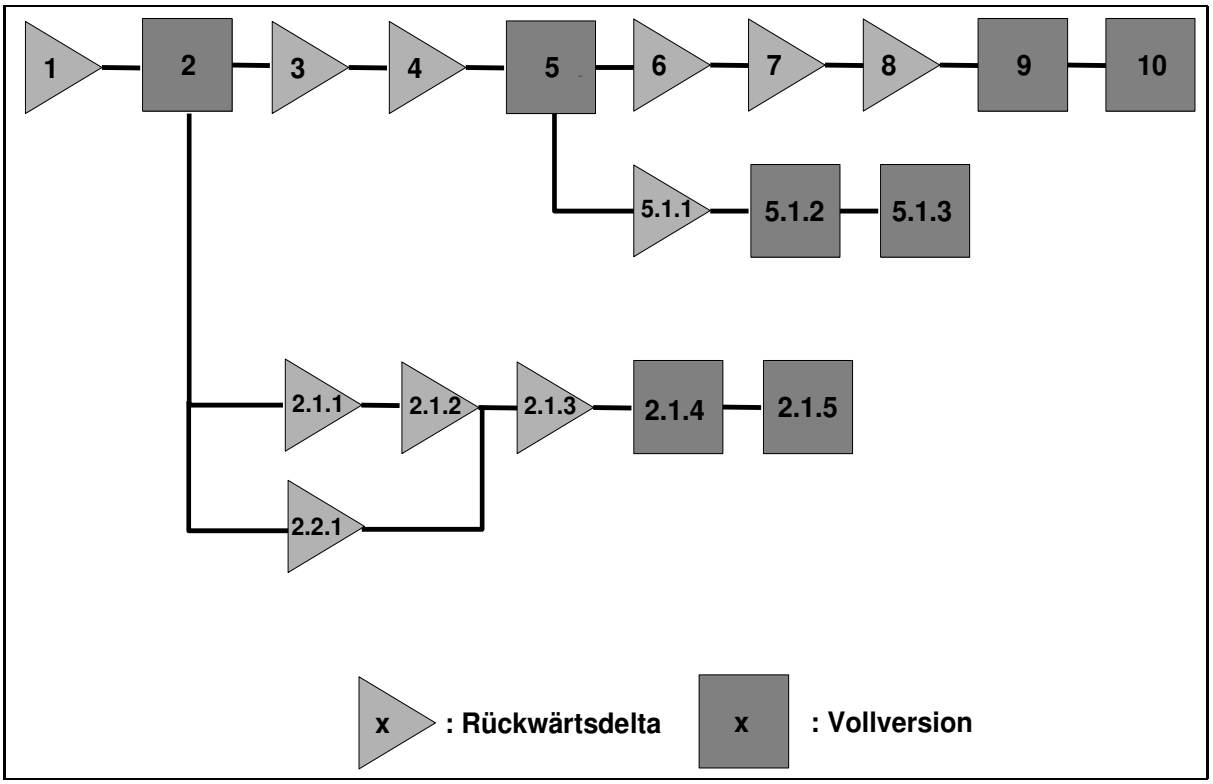


Abbildung 5.2: Rückwärtsdeltas in verschiedenen Strängen

können, abhängig von Bedarf und Kontext, integriert werden. Als Faustregel kann gelten, daß die letzten zwei oder drei Versionen eines Stranges und Versionen an Knotenpunkten als Vollversionen abgelegt werden sollten. Da nicht immer eindeutig ist, welcher Strang als Hauptstrang deklariert werden kann, sollte von einer Aufteilung in Haupt- und Nebenstränge abgesehen werden.

Die oben genannten Punkte lassen sich folgendermaßen zusammenfassen:

1. Abspeicherung der meisten Versionen im Archiv als Änderungsinformationen (Delta).
2. Die Deltabildung für die Dokumentstrukturen und die Deltabildung der Dokumenteninhalte geschehen in einzelnen Algorithmen. Die erzeugten Delta-Informationen werden aber in einem semistrukturierten Dokument abgelegt.
3. Verwendung von Rückwärtsdeltas mit mindestens zwei Vollversionen am Ende eines jeden Stranges.
4. Plazierung von Vollversionen an Knotenpunkten.

### 5.1.2 Anreicherung mit Metainformationen

Liegt eine angeforderte Version nur als Änderungsfolge (Editskript) vor, so muß die Vollversion aus einer gespeicherten Version und einem Delta (oder einer Folge von Deltas) regeneriert

werden. Um derart aufwendige Generationsprozesse (sprich überflüssige Extraktionen) in ihrer Zahl zu minimieren ist eine übersichtliche und aussagekräftige Navigation wichtig. Diese profitiert von der Metadatenebene, die sowohl Daten über die Dokumente selber verwaltet und speichert, als auch Projekt- und Arbeitsplanungsinformationen koordiniert.

Die Versionsverwaltung muß auch in der Lage sein, Verzweigungen zu kreieren,wenn es die Kooperationssituation erfordert. Wichtig ist hierbei, daß der Anwender nicht die Möglichkeit hat Versionen zu überschreiben, um Verzweigungen zu vermeiden. Dies würde den Persistenzgrundsatz der Versionsverwaltung verletzen.

In der kollaborativen Arbeit stellen Informationsdienste einen Kernpunkt dar. Die Versionsverwaltung unterstützt diese Dienste und sorgt für eine offene und flexible Anbindung. Sowohl der manuelle Zugriff, als auch der kontrollierte Fluß von Informationen muß sichergestellt sein. Navigationsdienste und Suchunterstützungen benötigen Metainformationen, um dem Anwender bei der Orientierung im Versionsarchiv behilflich zu sein. Dies stellt besondere Anforderungen an die Qualität der Metainformationsebene. Um über einen möglichst aussagekräftigen Katalog zu verfügen, muß die Anreicherung mit Metadaten von der Versionsverwaltung vorangetrieben werden. Vor jeder Archivierung wird der Benutzer angehalten, möglichst detailliert Auskunft über das zu archivierende Dokument zu geben.

All diese benötigten Informationen teilen sich auf in automatisch generierte und manuell eingegebene Daten. Zur Verwaltung dieser Informationen muß die Versionsverwaltung einen geeigneten Informationsdienst unterstützen, der auch die Informationsprotokolle verwaltet.

**Informationsprotokoll** Unter das Stichwort Informationsprotokoll fallen sowohl die Informationen zu jeder einzelnen Version, als auch die globalen Informationen, die die Gesamtkonfiguration beschreiben. Diese müssen aufgenommen und in Bezug zu den Versionen gespeichert werden. Ein vom Anwender auszufüllendes Protokoll wird, sowohl bei der Archivierung (*check-in*), als auch bei der Extraktion (*check-out*) eines Dokuments erstellt. Zusätzlich werden Parameter wie Datum oder Benutzername vom System ergänzt. Im Folgenden sind einige Basisinformationen dargestellt:

- Liste der datierten *check-ins* bzw. *check-outs*.
- Erfaßbarer quantitativer Umfang der Änderungen.
- Art der Änderungen[2].
- Der augenblickliche (Projekt-) Status des Dokuments (z.B. *checked-out*, „abgeschlossen“, etc.).
- Beschreibung der existierenden Stränge und Verzweigungen aus der Perspektive der betrachteten Version.
- ...

[2]Hier ist eine Vorklassifizierung (Formular mit Checkbox) von Vorteil.

Für die Arbeitsplanung existieren wichtige Dienste (z.B. können kooperative Effizienz und Entwicklungstatus bestimmt werden), die bestimmte Informationen von der Versionsverwaltung benötigen bzw. aus existierenden Projektdaten Bewertungen erstellen. Es ist wichtig, dies bei der Definition der Metadatenstrukturen zu berücksichtigen. Weiterhin steht die Offenheit bezüglich der Erweiterung der Metadaten im Vordergrund, da die Projektplanung, wie auch die Dokumentenerstellung, dynamischen Änderungen unterworfen ist. Als Standard wurde XML gewählt, weil die Struktur und die Erweiterungsmöglichkeiten der DTDs dynamischen Projektentwicklungen entgegenkommen. Alle diese Informationen müssen über längere Zeiträume hinweg zur Verfügung stehen, um auch innerhalb eines größeren Kontextes evaluiert werden zu können.

### 5.1.3 Navigierbarkeit

Die Navigierbarkeit ist ein wesentlicher Gesichtspunkt, da es unökonomisch ist, die einzelnen Versionen in ihrer Gänze durchzusehen. Ab einer gewissen Größe des Versionenarchivs kann eine Volldokumentsuche sehr zeitaufwendig sein. Das vorgestellte Modell arbeitet mit einer erweiterten Metainformationsebene, die die Versionenbeschreibung unterstützen soll. Die Suche nach einer bestimmten Version wird über die Metainformationsebene geführt und ist somit schneller und übersichtlicher. Suchdienste übernehmen die Vorauswahl und der Benutzer bekommt einen gefilterten Überblick.

Im Bereich der Dokumentenerstellung ist die Übersichtlichkeit der Archive besonders wichtig, da erfahrungsgemäß ein starkes Änderungspotential besteht. Einen Aspekt stellt die Identifikation der Versionen dar. Sobald eine Version archiviert wird, muß eine Identifizierer-Generierung automatisch erfolgen. Zusätzlich sollte eine semantische Zuordnung in Form von anwenderfreundlichen Bezeichnern vorhanden sein. Die Ablage des Versionsarchivs wird in Abbildung 5.2 dargestellt. Diese Form hat den Vorteil, daß schon existierende Navigationsmechanismen benutzt werden können. Sie stellt eine Erweiterung des RCS Archivs (siehe Abschnitt 3.1.2) dar.

Der Anwender muß die Möglichkeit haben, sich frei durch das Versionsarchiv zu bewegen und alle Stränge in beide Richtungen zu durchlaufen. Hier erscheint es sinnvoll, bei der Navigation intensiv auf die Metainformationsebene zurückzugreifen, um überflüssige Extraktionsprozesse zu vermeiden und dem Anwender einen besseren Überblick zu verschaffen.

Eine weitere Aufgabe der Versionshistorie ist die genaue Klassifizierung [AN91] einer einzelnen Version. Eine Unterteilung in verschiedene Versionsarten kann den Überblick erleichtern. In dem von Reichenberger [Rei94] beschriebenen Verfahren der orthogonalen Versionsverwaltung werden drei Dimensionen (Revision, Variante, Version) genutzt, um die Navigation im Versionenarchiv zu erleichtern. Die Beschreibungen der einzelnen Dimensionen sind in Kapitel 2 im Glossar zu finden. Die Identifizierung innerhalb der einzelnen Dimensionen findet über ein fortlaufendes numerisches Modell statt. Dies läßt sich sehr gut in einem kartesischen Koordinatensystem (Abbildung 5.3) darstellen und legt damit die Grundlage für visuelle Ansätze im Bereich der Navigation.

Für die Beschreibung der Versionenhistorie wird XML verwendet. Hier kann sowohl die Struktur des Versionenarchivs nach Abschnitt 5.1.3 in die DTD eingebunden werden, als auch der drei-dimensionale Ansatz nach Reichenberger (Abbildung 5.3). Desweiteren wird erreicht,

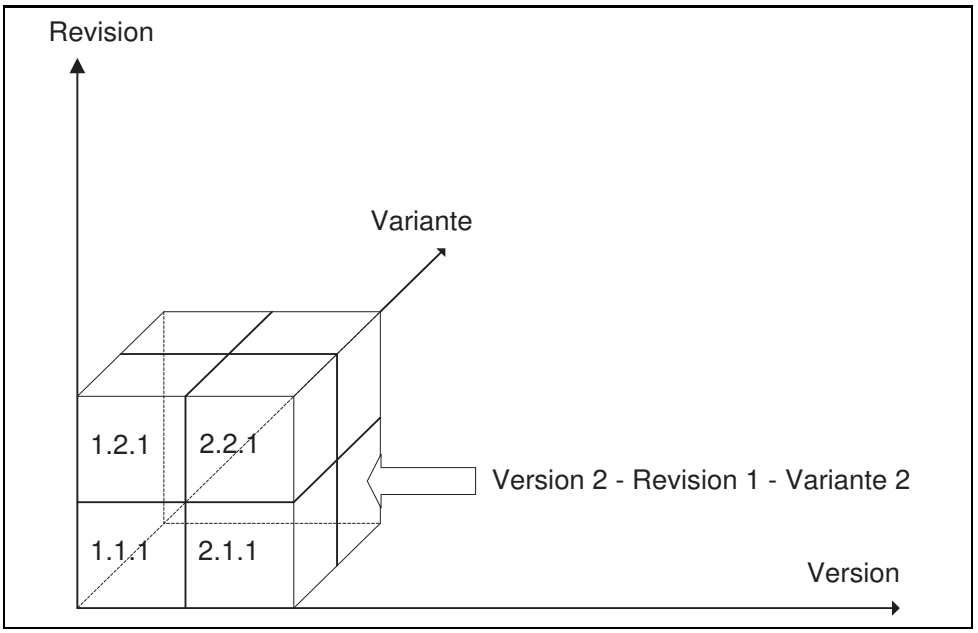


Abbildung 5.3: Darstellung des Drei-Dimensionenmodells nach Reichenberger

daß eine semantische Identifizierung erleichtert wird, da alle Informationen im Dokument strukturiert sind. Die einzelnen Elemente (Strukturbeschreibung in der DTD) können sowohl Namen als auch Attribute enthalten. Um große Informationsbestände zu klassifizieren und die Navigation solcher zu unterstützen existieren schon Konzepte und Ideen (z.B. *TopicMaps* [Emp99]), die auf XML beruhen. Diese können, aufgrund der offenen und flexiblen Schnittstellen als Mehrwertdienste eingesetzt werden.

Die Identifikation muß durch verschiedene Indikatoren möglich sein. Das heißt, sowohl durch einen numerischen Indikator, als auch durch Attribute (z.B. Autor oder Erstellungsdatum), die der jeweiligen Version zugeordnet sind.

Eine DTD (Dokument Typ Definition) zur Beschreibung der Evolutionshistorie eines Dokuments könnte folgendermaßen aussehen[3]:

*<?xml version="1.0"encoding="ISO-8859-1"?>*
*<!ELEMENT DOKUMENT (Version|Revision|Variante)>*
*<!ATTLIST DOKUMENT Identifizierer ID #REQUIRED>*

*<!ELEMENT Version(Attributliste, Data)>*
*<!ATTLIST Version Nummer ID #REQUIRED*
*Vorgänger IDREFS #IMPLIED*
*Nachfolger IDREFS #IMPLIED*
*...*
*>*

[3]Ein weiteres Beispiel findet sich in Anhang C.

*<!ELEMENT Revision (Attributliste, Data)>*
*<!ATTLIST Revision Nummer ID #REQUIRED*
*Vorgänger IDREFS #IMPLIED*
*Nachfolger IDREFS #IMPLIED*
*...*
*>*

*<!ELEMENT Variante (Attributliste, Data)>*
*<!ATTLIST Variante Nummer ID #REQUIRED*
*Vorgänger IDREFS #IMPLIED*
*Nachfolger IDREFS #IMPLIED*
*...*
*>*

*<!ELEMENT Attributliste EMPTY>*
*<!ATTLIST Attributliste Name CDATA #REQUIRED*
*Datum CDATA #REQUIRED*
*Autor CDATA #REQUIRED*
*...*
*>*

*<!ATTLIST Nachfolger Identifizierer ID #REQUIRED>*

*<!ELEMENT Data (#PCDATA)>*

Wichtig hierbei ist, daß wiederzusammengefügte Zweige (siehe auch Abbildung 5.2) dargestellt werden können. In dem angeführten Beispiel verfügt jedes Dokument über die Information seiner direkten Vorgänger und Nachfolger.

Um die Übersichtlichkeit zu erleichtern, ist der Einsatz von Filtern (siehe auch [BG81] in Kapitel 3) in hochfrequentierten Archivteilen. Hier wird dem Benutzer die Möglichkeit gegeben, durch Parametervorgaben auf bestimmte Archivteile zu fokussieren. Die Option einer manuell einstellbaren Auflösung erleichtert die Betrachtung des Versionsarchivs. Als ein Beispiel wäre die Ausblendung von Zwischenversionen zu nennen. So kann es von Vorteil sein, nur die Versionen zu ganz bestimmten Projektstufen zu betrachten und gegebenenfalls zu bewerten. Ausblenden oder Fokussieren kann bei der Nutzung von XML z.B. durch Stylesheets gelöst werden.

Eine weitere Möglichkeit in Bereichen mit hohen Zugriffsraten, in der Regel den zuletzt erstellten, ist die Aufteilung in aktuelle, neueste und Arbeitsversion. Die aktuelle Version entspricht derjenigen Version, die vom Evaluierungsstatus am weitesten bzw. die der Veröffentlichung am nächsten ist. Die neueste Version ist die, absolut gemessen, jüngste Version. Die Arbeitsversion befindet sich noch in der Entwicklungsphase und ist diejenige, die nach der aktuellen Version den höchsten Evaluierungsgrad hat.

Zusammenfassend läßt sich sagen:

1. Das Versionenarchiv wird mit Hilfe von XML beschrieben, um die Übersichtlichkeit zu erhöhen und vorhandene Präsentationsformen nutzen zu können.

2. Freie Navigation ist möglich und wird durch die Metadatenebene unterstützt.

3. Die Versionen werden mit Metadaten angereichert, um das Auffinden von Versionen zu erleichtern. klassifiziert (Version, Revision, Variante), um eine bessere Zuordnung zu erhalten.

4. Filtern (oder auch Suche nach aktueller, neuester oder Arbeitsversion) des Versionsarchivs erleichtert die Fokussierung auf bestimmte Bereiche.

5. Verwendung einer Dokument Type Definition (DTD) zur Beschreibung des Versionsarchivs, da DTDs erweiterbar sind und dem Konzept der informellen Beschreibung von Dokumenten sehr nahe kommen (Elementen können sowohl Namen als auch Attribute zugewiesen werden).

### 5.1.4 Mehrwertdienste

Weitere Zusatzdienste, die von einem Änderungsmanagement profitieren (siehe Abschnitt 2.3.4) sind z.B. Merging- und Notifikationsdienste, oder auch Verwaltung und Visualisierung von Evolutionshistorien. Beim Entwurf muß somit darauf geachtet werden, die Schnittstellen der Kerndienste offen und flexibel zu gestalten. Merging- und Notifikationsdienste profitieren von einer Struktur der Dokumente, die ihnen Zugriffe auf einzelne Knoten oder Metainformationen erleichtern. Die Evolutionshistorie kann sowohl für einen groben Überblick (Metadatenebene), als auch für einen detaillieren Fokus (Deltaebene) nützliche Anwendungen bereitstellen.

Eine geeignete Versionsverwaltung bietet die Möglichkeit zusätzliche Dienste zu integrieren. Anforderungsbeispiele sind Planungs-, Inhalts- und Rethorikunterstützung (Näheres in [Haa96]). Planungsunterstützung ist nötig, um den kollaborativen Charakter von Projekten zu unterstützen. Planungsvorgaben oder -strukturen können auch in die Versionierung miteinbezogen werden, indem z.B. Planziele als Attribute in den Identifikationprozeß integriert werden.

Die geplante Anwendung legt starkes Augenmerk auf die Beschreibung der Versionenen (Metadatenebene), um Transparenz für die Beteiligten zu schaffen. Entwicklungsprozesse und Bewertungsdienste können so leichter integriert werden. Die Autoren können flexibel auf die Anforderungen reagieren und ihre Arbeitsweise anpassen. Thurow [Thu00] spricht in seiner Arbeit von komfortableren Formen der Notifikation, die weitere Informationen über die „Arten der Modifikationen“ enthalten sollen. Hierzu gilt es die Voraussetzungen zu schaffen. Die im folgenden vorgestellten Mehrwertdienste stellen nur einen Ausschnitt dar und sollen die Vielfalt der verschiedenen Dienste repräsentieren. Der Fokus liegt dabei auf der notwendigen Unterstützung auf der Seite der Versionsverwaltung.

***Merging*** Ein wichtiger Punkt in der Dokumentenbearbeitung ist das Zusammenführen verschiedener Dokumentversionen. Dokumente werden zwar oft parallel erstellt, aber meist als ein Dokument veröffentlicht. Haake [Haa96] geht darauf im Detail ein (siehe auch *Interleaf* in Kapitel 3). Wird ein Dokument von zwei Benutzern gleichzeitig extrahiert und bearbeitet, so entsteht eine Verzweigung im Archiv. Da beide Versionen von einem Vorgänger abstammen werden zwei Nachfolgeversionen abgelegt. In vielen Fällen ist es möglich die getätigten

Änderungen zu einem späteren Zeitpunkt zu kombinieren und die beiden Nachfolgeversionen zu einer einzelnen zusammenzuführen (siehe Abbildungen 2.5 und 5.2). Dies wird als *Merging* bezeichnet. Automatisches *Mergen* hat sich als problematisch erwiesen, da selbst bei Änderungen an unterschiedlichen Dokumentteilen Abhängigkeiten bestehen können, die von den Autoren von Hand nachgebessert werden müssen.

Zur Unterstützung eines automatischem oder semi-automatischem *Mergings* sind Änderungsreporte wichtig. Diese beinhalten sowohl die Metadaten zur Änderung (Datum, Autor, Kommentare usw.), als auch eine geeignete[4] Präsentation der relevanten Änderungen. Außerdem ist ein wichtiger Punkt, daß auch hier eine projektbezogene Anpassung möglich ist: „A writing group's needs for detecting changes may be different at different times during the lifetime of a project. [Haa96]“ Es muß darauf geachtet werden, daß flexible Schnittstellen bereit stehen, die zusätzliche Dienste zulassen. Die Erfahrung zeigt, daß automatische Hilfswerkzeuge sehr individuell vom Benutzer bewertet und verwendet werden. Die Austauschbarkeit der Dienste ist somit notwendig.

Weiterhin ist wichtig, daß die Benutzerumgebung das Zusammenführen von Versionen oder Dokumentteilen anbietet. Anja Haake fordert ihrer Arbeit [Haa96]: „Wird beispielsweise ein früherer, zwischenzeitlich verworfener Dokumentzustand als neuer Ausgangspunkt für das Weiterarbeiten gewählt, so muß es möglich sein, bereits auf Grundlage eines späteren Zustands überarbeitete Teile, als konsistente, zusammengehörige Änderung in diesen, als neuen Ausgangspunkt dienenden, Zustand einzubringen.“. Dies macht deutlich, daß das *Merging* durchaus andere Motivationen als das Resultat zeitlich paralleler Arbeit haben kann.

XML ist in diesem Fall als Informationsbeschreibungsformat geeignet. Für die Bearbeitung existieren Editoren, die es ermöglichen beliebige Knoten zu einem Dokument zusammenzufügen. Die Hauptaufgaben des *Merging*-Dienstes sind die Bereitstellung einer (visuellen) Unterstützung für den Anwender, die ihm eine möglichst gute Übersicht liefert und die Erstellung von Strategien, die ein flexibles halbautomatisches *Mergen* [Jac01] unterstützen.

Als einer der Basisdienste kann die Deltabildung bezeichnet werden. Hunt et al [HVT98] schreiben in ihrer Arbeit über die Evaluierung verschiedener Deltadienste: „Other well known applications are the display of differences between files and the merging together of the changes in two different files relative to a common base.“ Wie später in diesem Abschnitt noch berücksichtigt werden wird, ist es also wichtig, schon bei den Kerndiensten der Versionsverwaltung spätere Weiterverwendungen im Auge zu behalten.

**Notifikationsdienst** Ein Notifikationsdienst ist ein zentrales Kommunikationswerkzeug des Informationsmanagement. Er kann in vielen Kommunikationsformen präsent sein. Beispiele sind sogenannte *Whiteboards*[5] oder Dienste die Änderungsmitteilungen über E-mail verschicken. Die Präsentation der Informationen ist jeweils dienstabhängig und wird in diesem Rahmen nicht behandelt. Vielmehr sollen die Voraussetzungen geschaffen werden, Notifikationsdienste zu unterstützen. Dazu ist im Zusammenhang mit einer Versionsverwaltung die Bereitstellung der Informationen über Änderungen und der problemlose Austausch zwischen den einzelnen Diensten elementar.

[4] „[...] reporting all changes is inappropriate for some tasks [...].“[Haa96]
[5] Ein Whiteboard entspricht der häuslichen Pinwand in digitalisierter Form.

Ein Notifikationsdienst beinhaltet meist mehrere Komponenten. Operiert wird oft mit automatisch generierten Informationen. Außerdem gibt es sogenannte Sender, die Informationen, z.B. über ein Eingabeprotokoll oder Vermerke innerhalb der Dokumente, generieren und hierfür Freigabemechanismen definieren. Auf der anderen Seite obliegt es dem Empfänger der jeweiligen Information die relevanten Teile herauszufiltern oder uninteressante Teile (automatisch) auszublenden. Dafür sorgen Filter, die individuell eingestellt werden können. Solche Filter werden von den meisten Notifikationsdiensten zur Verfügung gestellt [Thu00].

Eine vollständige Beschreibbarkeit meint nicht, daß sämtliche verfügbaren Informationen auch immer für alle Beteiligten sichtbar sein müssen. Hier sorgen Filter dafür, daß kein Überfluß an Information zur Unübersichtlichkeit führt. Auch Notifikationsdienste [Thu00] arbeiten mit verschiedenen Autonomiemodellen. Diese reichen von absoluter Autonomie, die keine oder individuell kontrollierte Freigabe der Informationen vorsieht, bis zur vollständigen Notifikation im Falle enger Kooperation. In diesem Fall ist es wichtig, daß von der Versionsverwaltung die Parameter und Benutzerdaten erzeugt und bereitgestellt werden, die eine Klassifizierung ermöglichen.

## 5.2 Architekturüberblick

Der Kern der entworfenen Versionsverwaltung läßt sich im wesentlichen in die drei Teile, Speicherverwaltung, Delta-Anwendung/Erstellung, sowie die benötigte Identifikation/Protokollerstellung gliedern. Mit entsprechend gewählten Schnittstellen wird der flexible Charakter der Versionsverwaltung gewahrt. Im Bereich der Speicherverwaltung und Speichererstellung müssen die Schnittstellen zum Versionsarchiv gewährleisten, daß es sowohl möglich ist mit den gängigen Datenbanken zu kommunizieren und zu kooperieren, als auch andere Archivierungsformen zu nutzen, wie z.B. portable Speicher (siehe Abbildung 5.4). Portable Speicher sind besonders im kooperativen Kontext wichtig. Nicht immer steht der Zugriff auf das Versionshauptarchiv (zentraler Server) zur Verfügung. Die Grundlagen für die Versionsverwaltung in einer privaten Arbeitsumgebung müssen auch gewährleistet werden können.

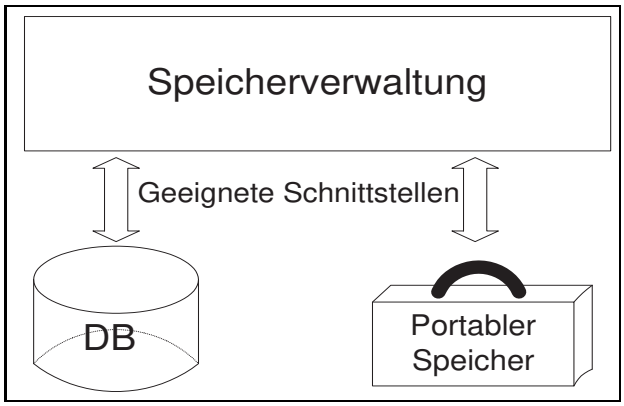


Abbildung 5.4: Verschiedene Archivierungsformen

Das Deltamodul ist dafür verantwortlich XML Dokumente zur Archivierung entgegenzunehmen, diese an die Speicherverwaltung weiterzureichen und das Archiv an die neue Situation

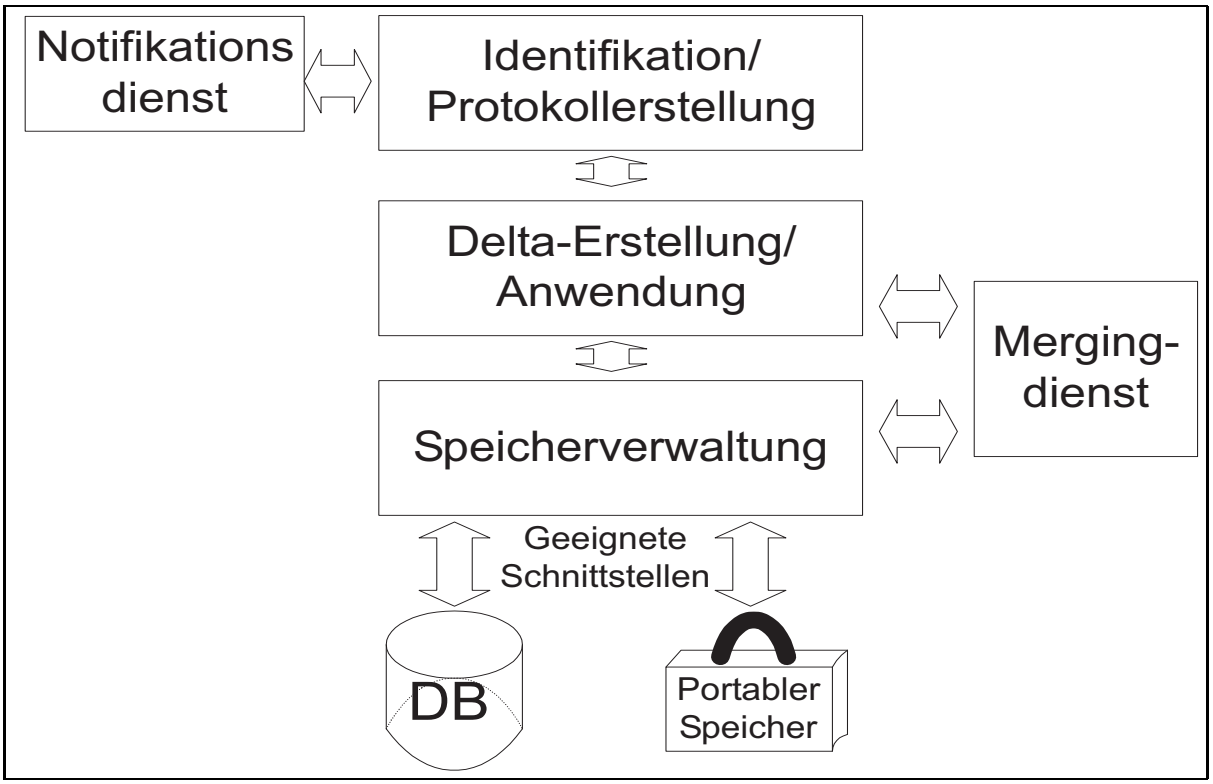


Abbildung 5.5: Darstellung der Schnittstellen

anzupassen, d.h. gegebenenfalls ein Delta aus einer älteren Verion zu erzeugen[6]. Weiterhin erzeugt das Delta-Modul auf Anfrage ältere Versionen aus Editskripten und einer Vollversion. Diese Daten werden von der Speicherverwaltung abgefragt.

Die Identifikation und Protokollerstellung zeigt sich verantwortlich für die Verwaltung der Metadaten zu den versionierten Dokumenten. Diese werden bei jedem Archivierungs- oder Extraktionsprozeß aktualisiert. Sie dienen als Suchunterstützung. Ist die gewünschte Version gefunden, so wird die Anfrage an das Deltamodul weitergeleitet, welches dann den Extraktionsprozeß übernimmt. Soll eine Version eingecheckt werden, so muß vom Benutzer ein Protokoll ausgefüllt werden, welches in der Metadatenebene abgelegt wird.

Mehrwertdienste wie z.B. Notifikationsdienste sollten austauschbar sein und die Schnittstellen einen Standard zum Datenaustausch verwenden. Weitere Vorteile für Identifikation und Protokollbeschreibung ergeben sich, wenn das Format *Content Management*-Qualitäten aufweist. In dieser Arbeit wird aus diesem Grunde XML als Datenaustauschformat gewählt.

[6]Dies ist die Vorgehensweise bei der Verwendung von Rückwärtsdeltas, wie sie in dieser Arbeit verwendet werden.

## 5.3 Abläufe und Prozesse

In diesem Kapitel werden einige beispielhafte Abläufe dargestellt. Diese entsprechen Szenarien, die in einer Versionsverwaltung integriert sind. Da im weiteren Verlauf der Arbeit die Abspeicherung neuer Versionen implementiert wird, sind die Prozesse zur Extraktion, Archivierung und Deltabildung ausgewählt worden.

Eine Version beschreibt den Status eines Dokuments zu einem bestimmten Zeitpunkt. Wurde eine Version archiviert, so ist sie nicht mehr zu ändern. Die Extraktion einer Version reserviert diese Version („sperrt“ sie) und erstellt eine Kopie für den Nutzer. Ein Wieder-Archivieren zieht unweigerlich die Schaffung einer neuen Version nach sich. Wurde eine Version mehrfach extrahiert, so führt dies beim Archivieren immer zu einer Verzweigung in der Versionshistorie.

### 5.3.1 Extraktion

Um die zu betrachtenden Abläufe zu gewährleisten und zu optimieren wird die Extraktion folgendermaßen beschrieben:

1. Auswahl der gewünschten Version.
2. Erhalt des Protokolls der Version.
3. Überprüfung der Nutzerrechte.
4. Sperren (bzw. Markieren) der Version.
5. Bereitstellung einer Arbeitskopie.

Das im zweiten Punkt erwähnte Versionsprotokoll umfaßt die in der Folge dargestellten Information:

- Datum an dem die Version zuletzt extrahiert wurde.
- Zustand der Version (z.B. gerade von anderem Nutzer extrahiert).
- Sperr-Status der Version.
- Klassifizierung als aktuelle/neueste/Arbeits-Version.
- Position der Version in der Historie (z.B. Hauptstrang/Nebenstrang).
- Benutzerprotokolle vorhanden.
- Notifikationsmechanismen werden/wurden initiiert.
- ...

Die hier vorgestellten Protokolldaten beziehen sich auf die betrachtete Dokumenterstellungsumgebung und können bei anderen Anwendungsgebieten variieren. Um genügend Flexibilität und Erweiterbarkeit zu gewährleisten kann eine Typisierung (DTD) gewählt werden, die die Strukturen wahrt und gleichzeitig nicht einschränkt. Die Erstellung einer solchen DTD ist ein kritischer Punkt der Arbeits- und Ablaufplanung eines Projekts und bedarf einiger Erfahrungszyklen.

### 5.3.2 Archivierung

Bevor eine Version archiviert werden kann, müssen einige Systemparameter überprüft und Informationen vom Benutzer abgefragt werden. Dies geschieht mit Hilfe einer Protokollroutine, die Teil des Archivierungsprozesses ist. Ein möglicher Ablauf einer Archivierung ist im folgenden beschrieben:

1. Anfrage zur Archivierung einer neuen Version.
2. Betrachtung der Vorgängerversion.
3. Eingabe eines Benutzerprotokolls.
4. Überprüfung der Nutzerrechte.
5. Überprüfung des Zustand/Status der Version.
6. Gegebenenfalls Erstellung einer Verzweigung.
7. Kreation eines eindeutigen Identifizierers.
8. Archivierung der Version[7].
9. Freigabe[8] der Version.

Für den Eintrag in das Protokoll sind verschiedene Informationen relevant. Einige Informationen werden vom System generiert (Autor, Datum usw.), andere müssen vom Benutzer bereitgestellt werden. Hier kann es von Vorteil sein, wenn der Nutzer explizit seine Motivation niederschreibt („Freitext“). Allerdings erleichtert ein strukturiertes Protokoll die automatische Weiterverwertung der Informationen.

In der freien Rede neigen Benutzer dazu, aus einem bestimmten (persönlichen) Fokus heraus zu schreiben. Dies ist vorteilhaft, wenn der Empfänger mit der speziellen Problematik vertraut ist. Gleichzeitig setzt es ein erhöhtes Verständnis voraus. Hier ist der Notifikationsservice im Sinne der Kooperation gefordert, eine eindeutige Dialektik in der Kommunikation zu fördern.

Der Navigationsdienst benötigt Informationen für die Suche von Versionen, die strukturiert und möglichst eindeutig sein sollten. Der Benutzer wird in der Protokollerstellung entlastet. Und es wird ein automatisch weiterverwendbares Informationsspektrum generiert. Folgende Fragen (im Rahmen des Benutzerprotokolls) können die Archivierung unterstützen:

[7] (Die genauere Beschreibung dieses Prozesses findet sich in Abschnitt 5.3.3)

[8] Inverse Operation zum „Sperren der Version“; Hier können verschiedene Freigabestufen implementiert werden.

- In welchem Maße (vorgegebene Skala) ähnelt das neue Dokument dem alten?
- Soll das Dokument als vollständige Kopie abgespeichert werden?
- Sollen bestimmte Zugriffsrechte gesetzt werden?
- Wurde ein Arbeitszyklus/-phase abgeschlossen[9]?

Auch hier stellen die vorgestellten Abfragen nur Beispiele dar. Ein geeignete DTD für die Metadaten muß immer projektbezogen erstellt werden. Vorteilhaft ist es, wenn die Projektstrukturen flexibel gehalten sind, so daß Heuristiken in der Parameteranpassung anwendbar sind.

### 5.3.3 Deltabildung

Eine veröffentlichte Studie [MDFK97] die sich mit der Effizienz von Datenspeicherung semistrukturierter Dokumente beschäftigt, vergleicht drei Differenzenverfahren in einer Fallstudie:

1. Eine zeilenweise Differenzenbildung.
2. Eine komprimierte zeilenweise Differenzenbildung.
3. Ein weiterentwickelter Typ der LCS-Differenzenbildung (*Longest Common Sequence* - längste Übereinstimmung).

Munson et al [MNW98] erkannten in ihrer Arbeit über die *Principles of Digital Document Processing*: „The common view on structured documents distinguishes between the structure of a document and its actual content (flat text).“ Von den betrachteten Verfahren nutzt keines die Vorstrukturierung der Dokumente. Auch das weiterentwickelte LCS-Verfahren durchsucht das gesamte Dokument.

Einige Produkte auf dem freien Markt vergleichen HTML Dokumente miteinander indem sie gleichartige durch entsprechende Tags gekennzeichnete Strukturelemente miteinander in Relation setzen. Allerdings wird dabei ausgenutzt, daß HTML über einen vordefinierten endlichen Pool an Beschreibungselementen verfügt. Diese werden sukzessive in beiden Dokumenten abgearbeitet, um die korrespondierenden Knoteninhalte in Relation zu setzen.

In der vorliegenden Arbeit soll nun getestet werden, welche Möglichkeiten in dem Ansatz stecken, die Dokumente gemäß ihrer Struktur miteinander zu vergleichen und effizient zwischenzuspeichern. Allerdings soll keine Festlegung auf einen beschränkten Standard wie HTML erfolgen, sondern die Offenheit von XML berücksichtigt werden. Ziel ist es die Flexibilität in der Deltagenerierung zu erhöhen, und damit das Anwendungsgebiet der erstellten Historien zu erweitern. Diese zusätzlichen Möglichkeiten können genutzt werden, um den Umfang der Dienste in der Versionsverwaltung zu erweitern bzw. zu verbessern.

Als Kernpunkt der Implementation wird in dieser Arbeit die Deltabildung ausgewählt. Der Großteil der Archivierungen erfolgt in Form von Deltas und somit stellt die Deltabildung quasi

[9] Wichtig für Arbeitsplanungsdienste die immer mehr Verwendung finden.

die Basis für alle beschriebenen Dienste dar. Die Archivierung aus Sicht der Deltabildung läßt sich folgendermaßen aufschlüsseln:

1. Ähnlichkeitsprognose mit der Vorgängerversion. Bei zu großer Differenz wird die Vollversion abgespeichert.
2. Vergleich der beiden Dokumente (Deltabildung).
3. Abspeichern des Deltas inklusive der Protokollinformationen.

Der Vergleich der Dokumente geschieht in mehreren Schritten. Zuerst werden die strukturellen Gemeinsamkeiten der beiden Dokumente idenfiziert. Dies geschieht durch einen von Christoph Spreen [Spr00] implementierten Algorithmus, der in 5.4.1 näher beschrieben wird. Er stellt die Grundlage für die Suche nach strukturellen Gemeinsamkeiten dar. Strukturänderungsbefehle halten die Überführung von einem in das andere Dokument fest. In Abbildung 5.6 wird ein XML Dokument in einer Repräsentation als Baumstruktur betrachtet. Das Strukturdelta ($S$-$D$) beschreibt nun, welche Änderungen struktureller Art an dem Dokument durchgeführt werden müssen, um es in Bezug auf die Struktur in das zweite beim Vergleich betrachtete Dokument zu überführen.

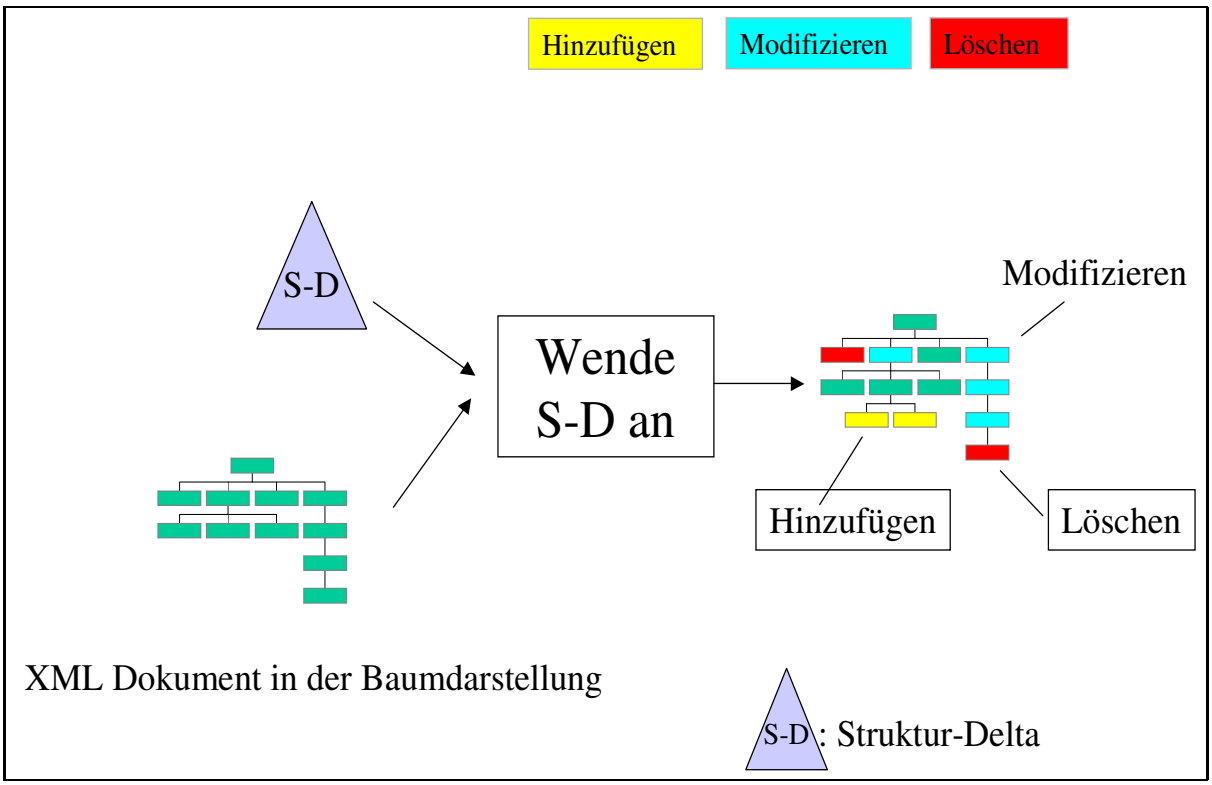


Abbildung 5.6: Delta-Erstellung auf struktureller Ebene

Stößt man auf strukturelle Gemeinsamkeiten mit inhaltlichen Differenzen, werden letztere durch ein inhaltliches Editskript beschrieben (siehe auch Abbildung 5.8). Beispielsweise kann

so durch die Operationen Verschiebung und Änderung eine realistische Dokumentevolution beschrieben werden, nämlich ein Kapitel, welches an eine andere Stelle verschoben und verändert wurde. Die Entdeckung von Gemeinsamkeiten hat den Vorteil, daß nur korrespondierende Inhalte miteinander verglichen werden.

Die Beschreibung der inhaltlichen Differenzen gliedert sich nach der *Generic Diff Format Specification* [vHP00] in das Einsetzen bzw. Löschen und das Ändern von Inhalten. Diese Spezifikation bezieht sich allgemein auf das generische File-Format für die Repräsentation der Differenz zwischen zwei Files. Die Änderung von Elementinhalten kann, je nach Anwendung, von verschiedenen Algorithmen durchgeführt werden. In dieser Arbeit wird ein Vergleichsalgorithmus entwickelt, der syntaktische Textstrukturen berücksichtigt und somit eine adäquate Präsentation der Deltas unterstützt. Abbildung 5.7 stellt die erläuterte Prozedur anhand eines Aktivitätsdiagramms dar.

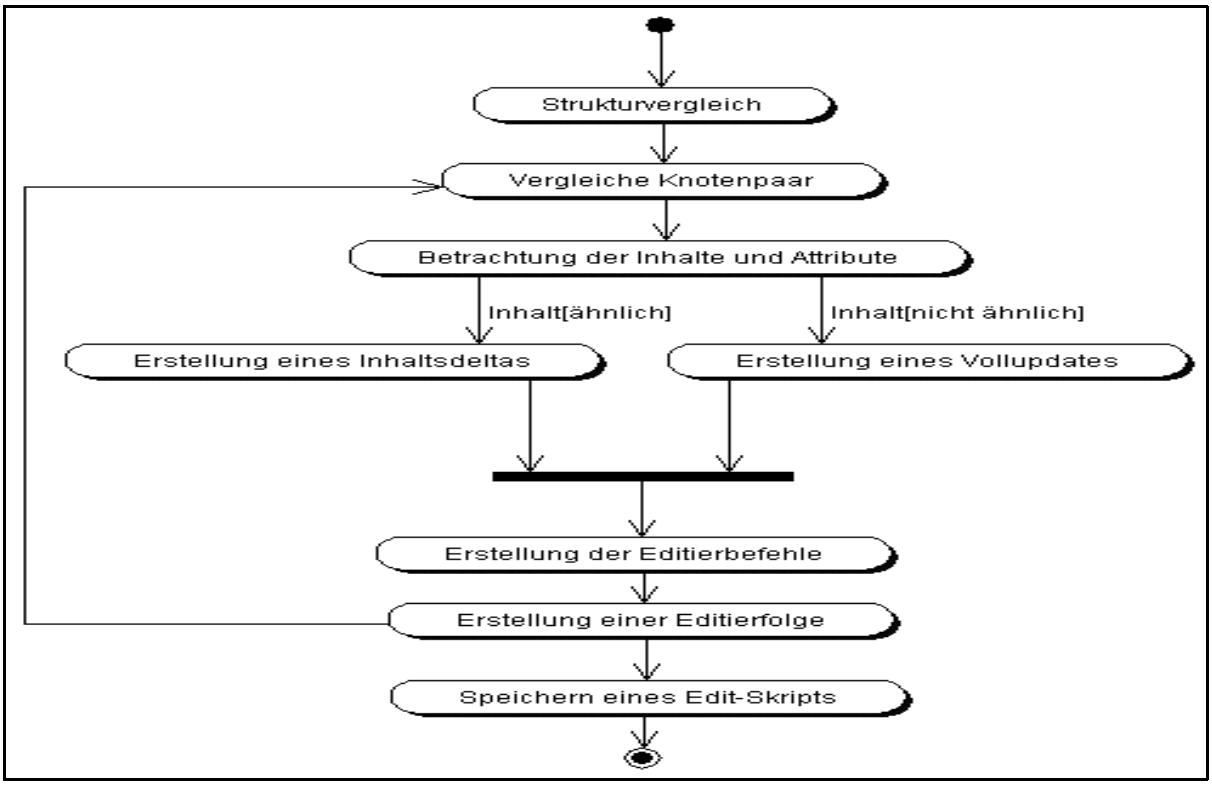


Abbildung 5.7: Delta-Erstellung als Aktivitätsdiagramm

## 5.4 Das Verfahren zur Differenzenbildung

Die Umsetzung der Deltabildung kann auf verschieden Arten erfolgen. Anstatt das Dokument als einen reine Zeichenfolge zu interpretieren, wird ein sukzessiver Ansatz verwendet. Wie im vorigen Kapitel dargestellt nutzt das angewandte Verfahren die Struktur der Dokumente aus. Die textuellen Inhalte werden durch das Verfahren von Christoph Reichenberger [Rei94] vorbearbeitet. Die Erstellung eines Deltas läßt sich folgendermaßen unterteilen (siehe auch

Abb. 5.8):

1. Einlesen der XML Dokumente durch einen SAX (Simple API for XML) Parser.
2. Vergleich der resultierenden Baumstrukturen (ohne die Inhalte).
3. Erstellung einer effizienten Strukturediterfolge, die es ermöglicht eine Struktur in die andere zu überführen.
4. Vergleich der Inhalte ähnlicher Knoten (siehe auch Abbildung 5.7):
   (a) Überschlagsweise Berechnung der Differenz der Inhalte (bei zu großer Differenz: Abspeicherung des gesamten Inhalts).
   (b) Aufteilung der Inhalte in kleinere (flexibel einstellbare) Entitäten.
   (c) Suche nach den längsten Gemeinsamkeiten im Volltext.
   (d) Beschreibung der Unterschiede[10].
   (e) Erstellung einer Editierfolge für die Inhalte und Integration dieser in die Strukturediterfolge.

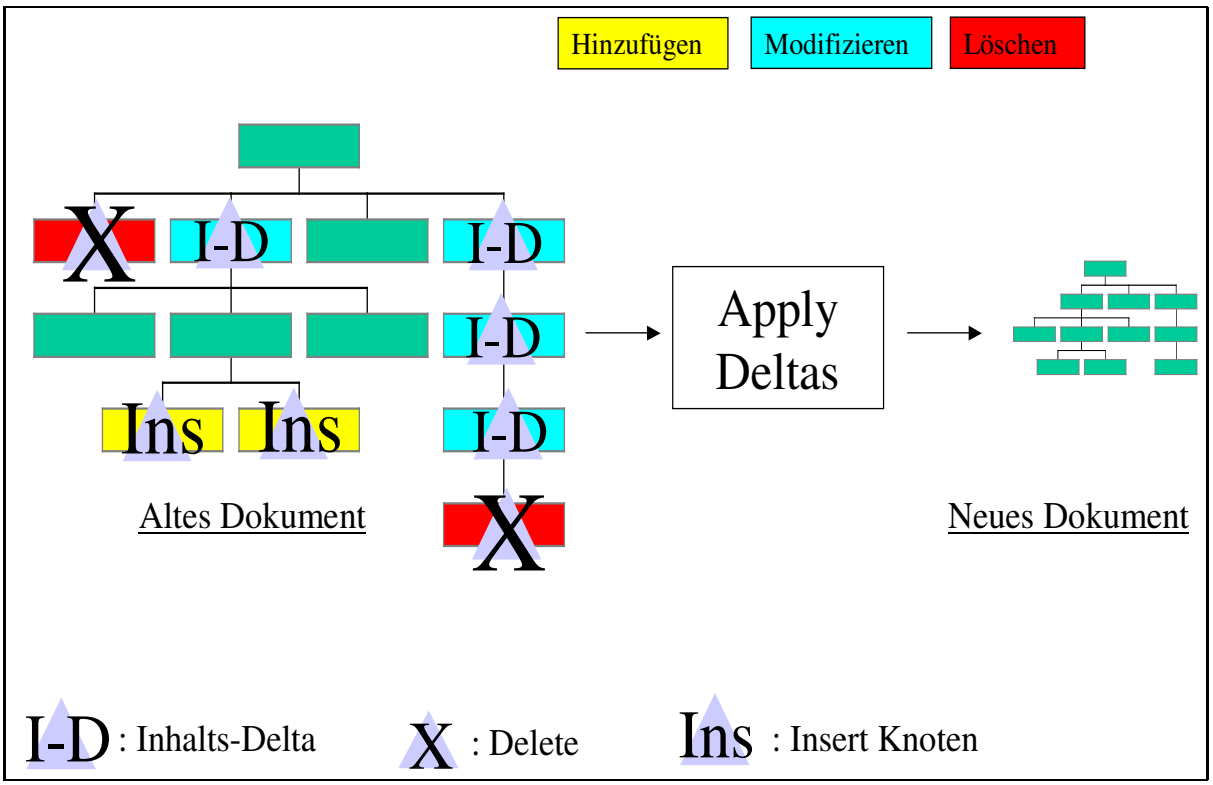


Abbildung 5.8: Aufteilung in Struktur und Inhalt

[10] Im Falle gerichteter Deltas reichen Anfügeoperationen (siehe [vHP00])

Zum Parsen der Dokumente stehen erprobte Parser zur Verfügung. Der in dieser Arbeit benutzte SAX Parser erstellt eine Baumstruktur der Elemente und stellt einige Methoden zur Verarbeitung des Baums bereit.

Nach Erzeugen der Baumstruktur werden die beiden Bäume miteinander verglichen. Christoph Spreen [Spr00] entwickelte hierzu in seiner Arbeit mit *XTree* einen Dienst, der die Strukturen von XML Dokumenten vergleicht.

### 5.4.1 *XTree*

*XTree* stellt insgesamt sechs Beschreibungselemente (oder auch Editieroperationen) zur Verfügung. Durch diese große Zahl wird die Problemstellung feingliedriger bearbeitet, was zu einer genaueren Evolutionsbeschreibung führen soll. Die von *XTree* benutzten Operationen sind näher in Kapitel 6.2.1 und im Detail in [Spr00] beschrieben. Insbesondere die Befehle *move* und *glue* ermöglichen laut Spreen [Spr00] eine größere Beschreibungstiefe. Diese höhere Auflösung der Änderungsbeschreibung wird auch von Neuwirth et al [NCK+92] in ihrer Arbeit „Flexible Diff-ing in a Collaborative Writing System“ gefordert. Sie erweitern das Beschreibungsspektrum allerdings nur um die *move*-Operation. Diese Operation wird realisiert indem nach *del-ins* Kombinationen gesucht wird, die gleiche Inhalte als Argumente enthalten.

Der Befehl *move* ermöglicht eine klarere Darstellung unter den Aspekten des Änderungsmanagement. Da der betrachtete Teil auch in der Realität meist nur verschoben wurde, ist die Darstellung gleichzeitig ein Beitrag zur realistischen Evolutionshistorie.

Der Befehl *glue* stellt eine weitere Stufe hinsichtlich der genaueren Änderungsbeschreibung dar. Die Operation entfernt einen Teil des Dokumentes, ähnlich wie *del*. Allerdings geschieht dies aus einer anderen Motivation heraus. *glue* zeigt eine Redundanz an: Sind zwei Elemente hinreichend ähnlich, so werden sie miteinander verschmolzen. „Diese Operation entspricht gewissermaßen einer umgekehrten Ausführung der *copy*-Operation. [Spr00]“

*XTree* ist ein XML Vergleichswerkzeug, welches die strukturellen Unterschiede zwischen XML Dokumenten beschreibt. In dieser Arbeit wurde der Dienst genutzt, um ein Editskript zu erstellen, welches auf ein Dokument angewandt werden kann, um ein neues zu generieren. Somit können die betrachteten Deltas zur Abspeicherung genutzt und von anderen Diensten weiterverwendet werden. Außerdem beschäftigt sich *XTree* ausschließlich mit Strukturen und deren Elementen bzw. Parametern. Für einen vollständigen Dokumentenvergleich müssen zusätzlich noch die Inhalte verglichen werden. Dieser Punkt wird im folgenden Abschnitt ausführlich dargestellt.

### 5.4.2 Inhaltsvergleich

Für einen Inhaltsvergleich ist primär wichtig, daß gleiche Teile erkannt werden, um Redundanzen in der Abspeicherung zu vermeiden. Die Editskripte der einzelnen Inhalte müssen einerseits möglichst kurz gehalten sein und andererseits über ein Maximum an Aussagekraft verfügen. Diese beiden Anforderungen stehen in der Realität in einem gewissen Konflikt zueinander. Hier gilt es einen geeigneten Mittelweg zu finden.

Der gewählte Inhaltsvergleich ist nicht eingeschränkt in Bezug auf Text oder Zeichenketten

und er ist flexibel bezüglich der Granularität einstellbar. Dies schränkt den Verwendungsgrad nicht zu stark ein, da Dokumente nicht nur textuelle Inhalte verwalten[11]. Die Flexibilität bezüglich der Granularität erleichtert die z.B. visuelle oder auch automatische Aufbereitung der Deltas in erheblichem Maße. In dem betrachteten Fall wird die Granularität an die sprachliche Semantik angepaßt, da in den betrachteten Dokumenten textuelle Inhalte vorherrschen. Somit werden als kleinste Entitäten Worte betrachtet. Allerdings können gröbere Einteilungen durchaus von Vorteil sein, um den Kontext in der Änderungsbeschreibung wiederzufinden.

Zuerst wird der Inhalt in kleinere Entitäten von einstellbarer Granularität eingeteilt. Diese Entitäten fungieren als kleinste zu vergleichende Einheiten. Sie werden in eine Hash-Tabelle (siehe auch Abbildung 6.12 in Kapitel 6) eingeordnet, um einen schnelleren Zugriff zu gewährleisten und eine unabhängige Identifizierung zu erreichen. Die gewählte Hashfunktion ist ein Parameter für die Performanz des Algorithmus. Diese Vorstrukturierung verkürzt die Suchdauer nach den längsten gemeinsamen Textstellen. Reichenberger [Rei94] stellt in seiner Arbeit Benchmark-Ergebnisse zur Verfügung, die bestimmte Hashfunktionen oder Granularitäten herausheben. Hierauf wird im nächsten Kapitel noch ein wenig genauer eingegangen.

Die Hashfunktion ist projekt- bzw. kontextabhängig wählbar. Alle beteiligten Programmkomponenten oder deren Parameter sind variabel und austauschbar gehalten. Durch die Vorstrukturierung wird erreicht, daß nur Entitäten miteinander verglichen werden, die mit Übereinstimmungen beginnen. Die Suche nach den größten Gemeinsamkeiten resultiert in einer größeren Effizienz der Editskripte. Zum einen benötigt ein *copy*-Befehl weniger Speicheraufwand und zum anderen ist er bei einer Rekonstruktion einer Version schneller in der Ausführung. Da gerichtete Deltas beschreiben wie eine Version in eine andere überführt werden kann, besagt ein *copy*-Befehl zumeist, daß der betreffende Teil in dem Dokument unverändert bleibt und zieht keine wirkliche Aktion nach sich. Nicht übereinstimmende Textteile werden durch den *add*-Befehl repräsentiert. Hier werden bei der Rekonstruktion die zu ergänzenden Inhalte eingefügt. In der Auswahl der Befehle wurde sich an dem ausgewiesenen Standard von van Hoff und Payne [vHP00] orientiert, um eine Offenheit der Anwendung zu gewährleisten.

**Hashfunktionen** Die Hashfunktion ist verantwortlich für die Gleichverteilung des Textes. Jedem Element wird ein numerisch eindeutiger Hashwert zugeordnet. Dieser fungiert als Zugriffsschlüssel. Um die Zugriffe zu optimieren, ist es notwendig eine annähernde Gleichverteilung der Schlüssel zu erreichen. Hier kann es sinnvoll sein, für ein spezielles Projekt eine eigene Hashfunktion zu kreieren.

Die Vorklassifizierung durch die Hashwerte wird in einer separaten Programmroutine durchgeführt und beinhaltet eine Bewertung (Niveau) der Gleichverteilung. Da der Implementationsaufwand für Hashfunktionen relativ gering ist, ist es ratsam mehrere Hashfunktionen zur Verfügung zu stellen und eine jeweils geeignete auszuwählen. Für die betrachtete Anwendung wurden zwei Funktionen getestet (siehe Kapitel 6.3).

[11] Hunt et al [HVT98] schreiben „Today, binary differencing capability has become mandatory.“

## 5.5 Präsentation der Änderungen

In kollaborativen Arbeitsumgebungen erlangt die Präsentation der Änderungen eine immer größere Bedeutung (siehe auch [Haa96] und Kapitel 3). Gerade bei immer komplexer werdenden Dokumentstrukturen ist es schwierig den Überblick über die Evolutionshistorie zu behalten. Eine Studie an der Universität Aalborg [BNLINLSP97] beschäftigte sich mit der kooperativen Dokumentenentwicklung. Eine entworfene Unterstützungsumgebung wurde über einen längeren Zeitraum evaluiert und nach Befragung der Nutzer ergaben sich folgende Wünsche in Bezug auf die kollaborative Unterstützungsumgebung:

- Eine Evolutionshistorie des gesamten Dokuments sollte zur Verfügung stehen.
- Einzelne Entwicklungsschritte sollten nachzuverfolgen sein.
- Falls Editskripte bei der Speicherung verwendet werden, so sollten sie einfach verständlich und einsehbar sein.
- Die Betrachtung ausgesuchter Evolutionsintervalle sollte möglich sein.

Aus den Wünschen wird deutlich, daß eine starke Nachfrage nach einer flexiblen Evolutionsbeschreibung besteht. Durch die Ablage der Versionen als Änderungsinformationen liegt es nahe, diese so zu gestalten, daß sie in verschiedenen Kontexten verwendet werden können.

Neuwirth et al [NCK+92] schreiben, daß ein Änderungsprotokoll die Zerstreuung der Leser minimieren muß. Wie in Abschnitt 2.3.4 zitiert, fällt es Betrachtern schwer zwei Dokumente mit gleicher Aufmerksamkeit zu betrachten und gleichzeitig die Unterschiede zu erkennen. Ziel ist es, dem Benutzer vom eigentlichen Vergleich der Dokumente zu entlasten, so daß er sein Augenmerk auf die Änderungen legen kann. Im Bereich semistrukturierter Dokumente kommt es vor, daß ganze Elementknoten (z.B. zur Neustrukturierung) verschoben werden, aber auch daß innerhalb von Elementen nur kleine Änderungen durchgeführt werden. Die Präsentation sollte einer Strukturveränderung und einer expliziten Inhaltsveränderung unterscheiden können (vgl. Abschnitt 5.4.1).

Für die ausgewählten Aspekte der Realisierung spielen all diese Überlegungen eine große Rolle. Mit dem dargestellten Ansatz lassen sich sowohl Evolutionshistorien erstellen und darstellen, als auch Änderungsreporte mit Standardwerkzeugen visualisieren.

# Kapitel 6

# Ausgewählte Aspekte der Realisierung

Nachdem im vorangegangenen Kapitel der Kern einer Versionsverwaltung entworfen wurde, stellt dieses Kapitel die in dieser Arbeit implementierten Teile vor. Der Fokus liegt dabei auf einem Delta-Erstellungsdienst. Zuerst wird ein grober Architekturüberblick gegeben und im weiteren Verlauf werden die einzelnen Komponenten des Dienstes näher erläutert. Das Kapitel schließt mit einer Evaluierung des Dienstes die auf den Bewertungskriterien aus Kapitel 2.4 beruht.

## 6.1 Grobarchitektur der Realisierung

In diesem Abschnitt werden die Hauptaspekte der Realisierung grob erläutert. Kernpunkte sind die Deltaerstellung und die Recovery eines Dokumentes aus Delta und einem anderen Dokument.

### 6.1.1 Erstellung eines strukturierten Inhaltsdeltas (SCDelta)

Die Deltaerstellung gliedert sich in verschiedene Klassen. *XTreeCaller* ruft die benötigten Klassen zur Deltaerstellung auf und stellt eine graphische Oberfläche zur Verfügung. *XTree* ist der Basisdienst der zur Deltaerstellung erweitert wurde. Der *Parser* erzeugt die vom *Comparer* benötigten Strukturen, um ein Strukturdelta zu erzeugen. Von *SCDelta* wird die Erstellung der inhaltlichen Deltas (*structured content delta*) übernommen. Diese werden durch die entworfenen Änderungsbefehl-Objekte abgebildet. Abbildung 6.1 zeigt die Grobarchitektur der Systemkomponenten. *XTree*, sowie *XHandler* und *Comparer* werden nahezu unverändert aus der Arbeit von Spreen [Spr00] übernommen. Sie sind für den strukturellen Vergleich zuständig. Für eine ausführliche Darstellung dieser Programmkomponenten sei auf [Spr00] verwiesen.

*XTree* stellt in der ursprünglichen Fassung [Spr00] ein Vergleichswerkzeug dar und stellt somit die Funktionalität der Wiederherstellung von Dokumenten nicht zur Verfügung. Die

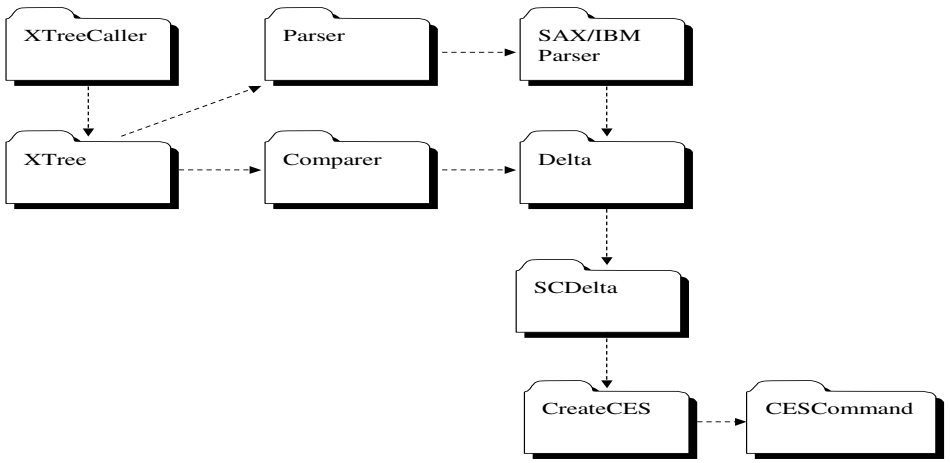


Abbildung 6.1: Grobarchitektur: Deltaerstellung

Erweiterung einiger Änderungsoperatoren (*copy*, *move*, *delete*, *glue*) war trivial. Allerdings war es notwendig sowohl für den *insert*, als auch für den *update* Befehl die betrachteten Inhalte mit ins Editskript zu integrieren. In Abbildung 6.2 ist die Grobarchitektur bezüglich des Operationenrepertoires dargestellt. Ebenso wie die einzelnen Befehlsmodule der Erweiterung bedurften, mußte auch die DTD für das *SCDelta* Editskript erweitert werden (vgl. Anhang B).

Eine weitere Kernfunktionalität liefern die Pakete *CreateCES* und *Delta*, die für die Erstellung eines Komplettdeltas zuständig sind. *Delta* ist verantwortlich für die Erstellung von Inhaltsdeltas. *CreateCES* erzeugt das Editskript, welches sich aus Strukturänderungsbefehlen und inhaltlichen Änderungsbeschreibungen zusammensetzt.

### 6.1.2 Recovery eines Dokuments

Ein Vergleichsdienst der für die Abspeicherung von Deltas eingesetzt werden soll, ist neben der Erstellung von Deltas die den Speicheraufwand reduzieren, auch für die Wiederherstellung von Versionen zuständig, die nur in Deltaform vorliegen. SCDelta ist so aufgebaut, daß die in XML-Form abgelegten Editskripte auf eine Version anwendbar sind und somit eine ältere Version erhalten zu können[1]. Die Ausführung der einzelnen Änderungsbeschreibungen wurde als Methode (*apply*) in die einzelnen Befehlsklassen integriert. Zuerst wird vom erweiterten *XTree* aus dem Editskript eine Operationsfolge generiert, die angewendet werden kann, um eine Version wiederherzustellen. Jede einzelne Änderungsoperation wird sukzessive durchgeführt. Die Klasse *Create* ist verantwortlich für die Wiederherstellung der Inhalte der inhaltlich veränderten Knoten. Liegt bei einem Knoten außer einer Strukturänderung auch eine Inhaltsänderung vor, so wird erst der Inhalt regeneriert und dann der Strukturänderungsbefehl ausgeführt.

[1]Diese Funktionalität wird von *XTree* nicht zur Verfügung gestellt, da die benötigten Informationen aus den erzeugten bipartiten Graphen generiert werden können und somit nicht alle notwendigen Informationen im Editskript ablegt werden.

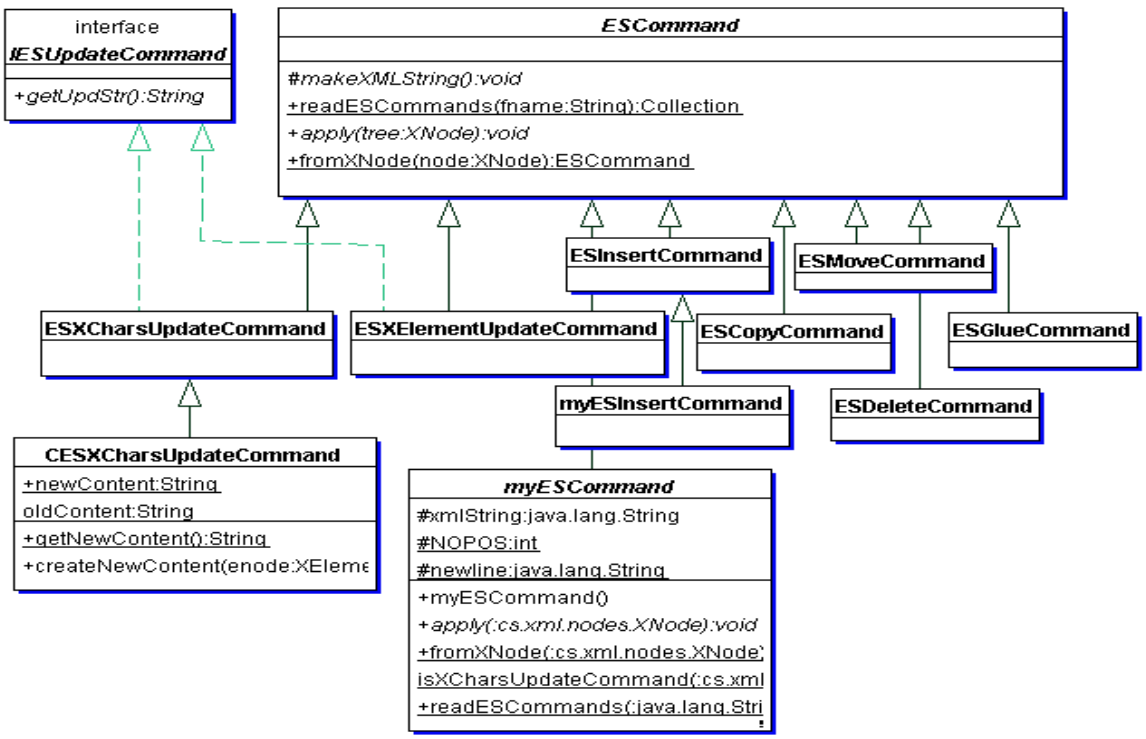


Abbildung 6.2: Architektur des Systems zur Erstellung eines Editskripts

Ein weiterer Punkt ist die Überführung der Baumstruktur in ein XML Dokument. *XTree* operiert auf einer selbst-geschaffenen Baumstruktur. XML Dokumente werden zuerst in die eigenen Objektklasse *XNode* überführt. Sämtliche Anwendungen geschehen auf der Klasse *XNode* und deren Unterklassen. Dies ist auch der Grund warum kein Standard-*XMLDocumentwriter* benutzt werden konnte, sondern ein *MyXMLDocumentwriter* implementiert wurde.

## 6.2 Verwendete Algorithmen

Die Deltabildung wird in zwei voneinander unabhängigen Teilschritten durchgeführt. Im ersten Schritt werden die Strukturen der Dokumente miteinander verglichen und die Differenzen durch Editieranweisungen beschrieben. Im zweiten Schritt werden ähnliche Knoten mit unterschiedlichen Inhalten einem Inhaltsvergleich unterzogen und die Inhaltsdifferenzen durch Editieranweisungen beschrieben. Diese Änderungsbeschreibungen werden schließlich in einem Editskript zusammengefaßt.

Für den Strukturvergleich wurde *XTree* ausgewählt. *XTree* ist ein Dienst der Unterschiede zwischen XML Dokumenten beschreibt. Der implementierte Inhaltsvergleich beruht auf einem Ansatz von Christoph Reichenberger [Rei94].

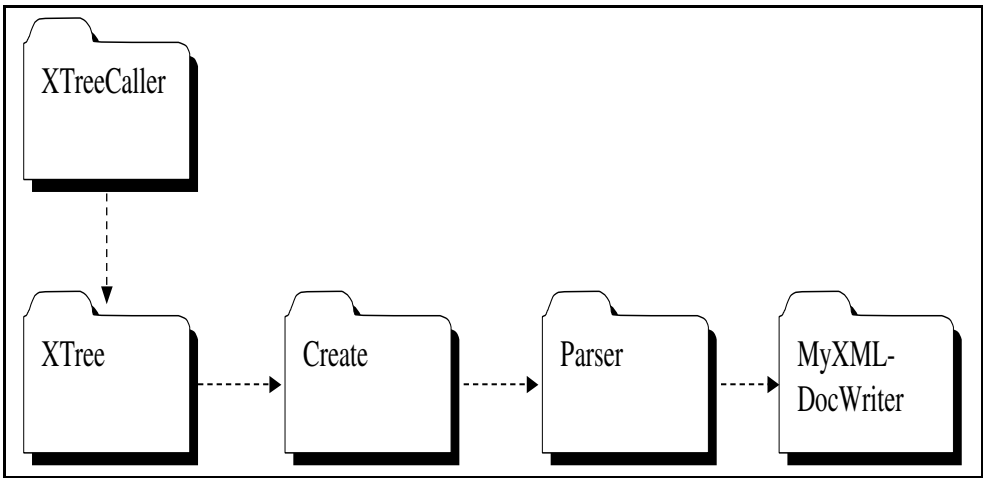


Abbildung 6.3: Grobarchitektur: Recovery

### 6.2.1 Strukturvergleich

Der von Christoph Spreen [Spr00] implementierte Dienst *XTree* wird für den Strukturvergleich ausgewählt, weil bei der Entwicklung des Verfahrens besonderes Augenmerk auf die Konfigurierbarkeit, die Erweiterbarkeit, die Wiederverwendbarkeit und die Nachvollziehbarkeit gelegt wurde. Außerdem läßt das Operationenrepertoire, welches sechs Änderungsbeschreibungen umfaßt, darauf hoffen, eine detailgetreue Evolutionsbeschreibung des betrachteten Dokuments zu erhalten. Der Fokus liegt nicht auf der Minimierung der Anzahl der Beschreibungselemente (und somit einer Minimierung des Aufwands), sondern in der Maximierung des Informationsgewinns für den Benutzer bezüglich der Änderung und deren Evolution.

Ein weiterer Hauptgrund für die Wahl von *XTree* ist die Operabilität mit XML Dokumenten. Viele in der Literatur angesprochenen Strukturvergleiche (siehe z.B. Cvs99) beschränken sich darauf mit einer limitierten Anzahl von Elementen zu arbeiten. *HTML* stellt ein Beispiel einer limitierten Auswahl von Elementen zur Strukturbeschreibung dar. Mit *XTree* dagegen können beliebige wohlgeformte XML Dokumente miteinander verglichen werden. Zusätzlich werden die Editierfolgen im XML-Format abgelegt, was es ermöglicht, Ergänzungen vorzunehmen und die Editierfolgen in Mehrwertdiensten weiterzuverwenden (z.B. Inhaltsvergleiche einzelner Knoten durchzuführen[2]).

#### *XTree*

Der generische Dienst für den Vergleich semistrukturierter Informationen *XTree* wird im folgenden näher beschrieben[3].

Zum Vergleich zweier XML-Dokumente werden die Dokumente mit Hilfe eines *SAX-Parsers*

[2]Somit können z.B. einzelne Kapitel herausgegriffen werden und auf ihre Entstehungsgeschichte hin analysiert werden.

[3]Für eine ausführliche Darstellung des Algorithmus, sowie deren informatische Grundlagen sei auf die ausführliche Dokumentation [Spr00] verwiesen.

*(Simple API for XML)* in einen Baum überführt. In Abbildung 6.4 ist die graphische Ober-

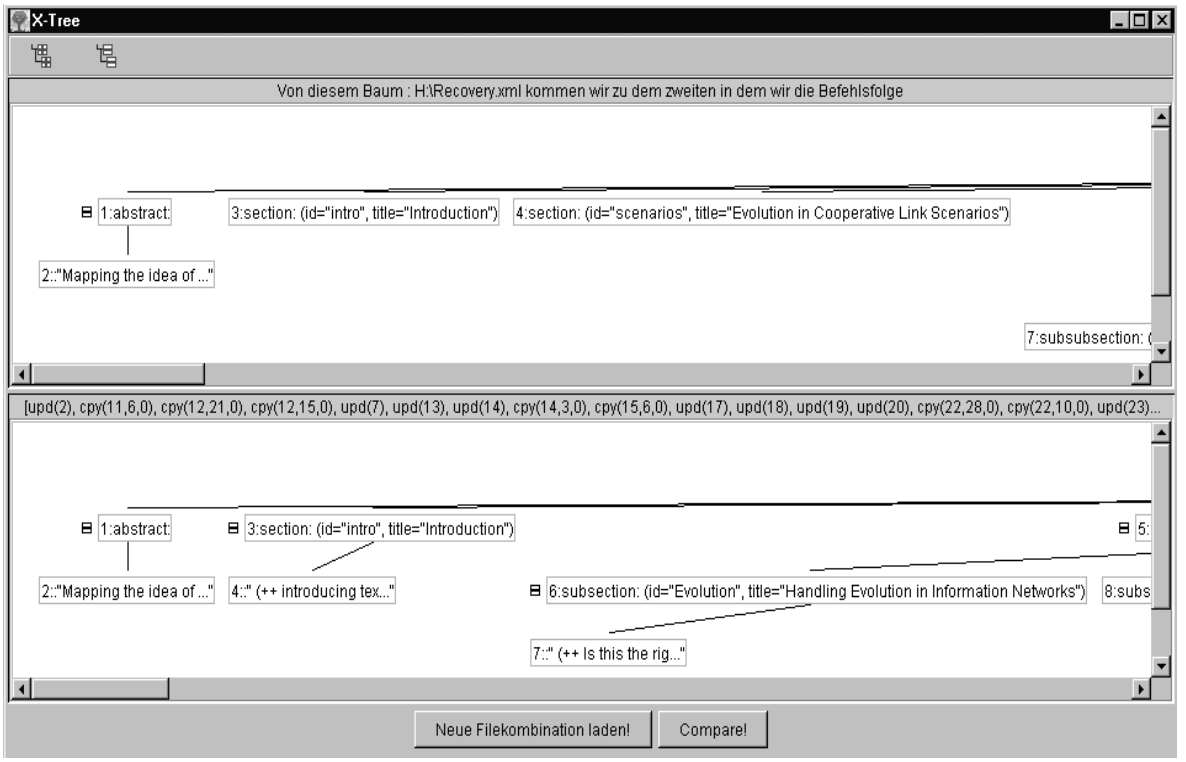


Abbildung 6.4: *XTree* Baumdarstellung

fläche von *XTree* dargestellt. Sie ist für kleinere Strukturen sehr anschaulich, wird aber bei der Verwendung von komplexeren Strukturen (z.B. größeren Textdokumenten) unübersichtlich. Aus dem Baum wird ein induzierter Graph erstellt, der durch Einfügen von Kanten die Knoten des einen Baumes mit den Knoten des anderen Baumes, verbindet. Die Kanten erhalten eine Wichtung, welche ein Maß für die Ähnlichkeit, der durch sie verbundenen Knoten, darstellt. Der Graph dient als Grundlage für die Erstellung eines gewichteten Korrespondenzproblems. Die Lösung dieses Korrespondenzproblems wird zur Erstellung des Editskripts herangezogen, welches im XML Format abgelegt wird. Die Wahl des XML Formats gewährleistet dabei eine hohe Kompatibilität. Aus demselben Grund werden auch in den getätigten Erweiterungen die bestehenden Strukturen weitestgehend beibehalten bzw. ergänzt. In Abbildung 6.5 ist ein Teil eines Editskripts dargestellt, welches aus der Evaluierung aus Kapitel 6.3 stammt.

### Das Operationenrepertoire

*XTree* nutzt einen Standard-Parser, um XML-Dokumente in Bäume zu überführen. Diese Bäume werden dann miteinander verglichen und die Differenz in Form eines Editskripts abgelegt. *XTree* zeichnet sich durch ein große Beschreibungstiefe aus. Insgesamt stehen sechs Typen von Operationen zur Verfügung, um die Unterschiede zwischen zwei XML Dokumenten zu beschreiben. Im Folgenden wird das Operationenrepertoire kurz beschrieben. Sie beziehen sich immer auf (Teil-) Bäume und nicht, wie in anderen Verfahren üblich, auf einzelne

```
<?xml version="1.0" ?>
<!DOCTYPE es (View Source for full doctype...)>
- <es>
  + <upd nodenr="9">
    <cpy nodenr="11" targetparent="19" childpos="0" />
    <cpy nodenr="16" targetparent="19" childpos="1" />
  + <upd nodenr="19">
  + <upd nodenr="24">
  + <upd nodenr="26">
  + <upd nodenr="31">
  + <upd nodenr="33">
  + <ins targetparent="0" childpos="2">
    <mov nodenr="29" targetparent="19" childpos="0" />
    <mov nodenr="34" targetparent="19" childpos="1" />
  </es>
```

Abbildung 6.5: Editskript von *XTree*

Knoten. Nach Ausführen der Operation werden die Knoten des Baums wieder in Präorder durchnumeriert; d.h. es findet eine Umnumerierung statt.

**update** Die Operation *upd* verändert den Inhalt eines einzelnen Knotens. Hierbei wird unterschieden, ob sich Knoteneigenschaften wie Name oder Attribut sich verändert haben oder der Inhalt. *XTree* ersetzt im Fall eines inhaltlichen Unterschieds den gesamten Inhalt. Beispiel:
*upd(1,neuerInhalt)* ersetzt den alten Inhalt in Knoten „1“ durch *neuerInhalt* (siehe Abbildung 6.6).

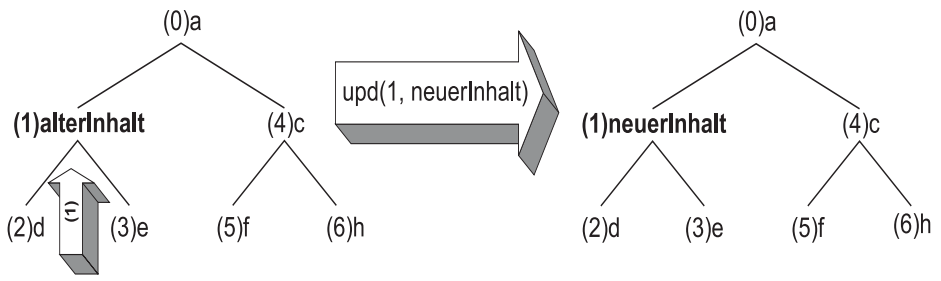


Abbildung 6.6: Der *Update*-Befehl

**insert** Die Operation *insert* fügt einen neuen Teilbaum in den Baum ein. Beispiel:
*ins(„m“(„n“,„o“),0,1)* fügt einen Teilbaum (Wurzel „m“ und Kindknoten „n“ und „o“) an

der Stelle „0“ (Wurzel) an der zweiten Kindposition (Numerierung beginnt bei 0) ein (siehe Abbildung 6.7).

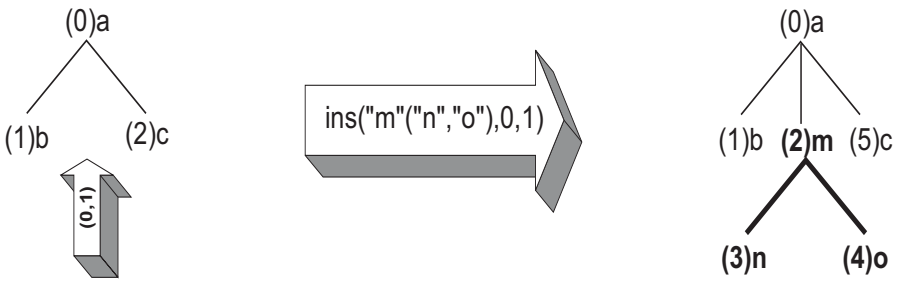


Abbildung 6.7: Der *Insert*-Befehl

**delete** Die Operation *del* löscht einen Teilbaum, dessen Wurzel im Editier-Befehl angegeben ist. Beispiel:
*del(4)* löscht den kompletten Teilbaum dessen Wurzel die Knotennummer „4“ trägt. In diesem Fall werden die Knoten „4“, „5“ und „6“ gelöscht (siehe Abbildung 6.8).

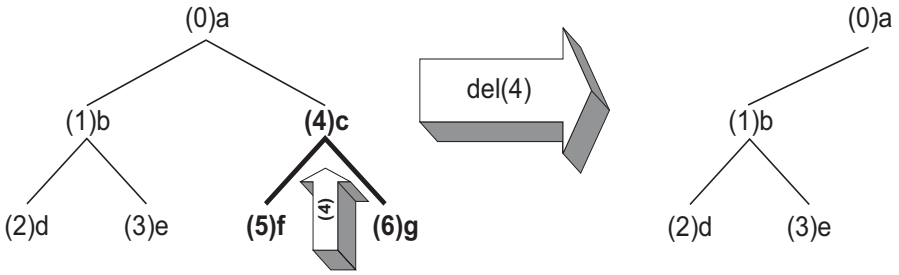


Abbildung 6.8: Der *Delete*-Befehl

**move** Die Operation *mov* verschiebt einen Teilbaum an eine neue Position im Baum. Beispiel:
*mov(1,4,1)* verschiebt den Teilbaum dessen Wurzel die Knotennummer „1“ trägt an den Knoten „4“, und zwar an die Kindposition „1“, also die zweite Stelle (siehe Abbildung 6.9).

**copy** Die Operation *cpy* kopiert einen Teilbaum an eine neue Position im Baum, so daß zwei identische Teilbäume entstehen. Beispiel:
*cpy(1,4,1)* kopiert den Teilbaum dessen Wurzel die Knotennummer „1“ trägt an den Knoten Nummer „4“ und zwar an die zweite Kindposition (siehe Abbildung 6.10).

**glue** Die Operation *glu* entfernt, ähnlich der *delete*-Operation, einen Teilbaum. Jedoch wird hier davon ausgegangen, daß der entfernte Teilbaum einem anderen Teilbaum hinrei-

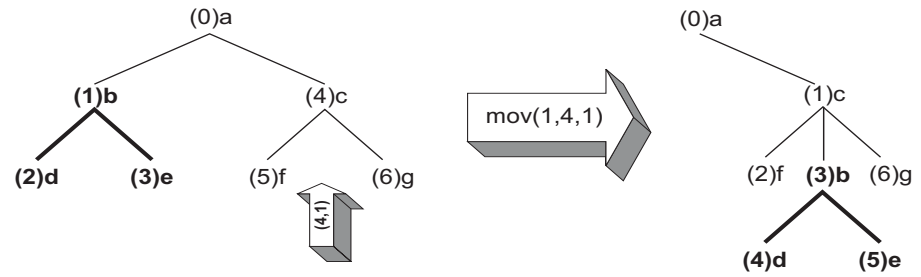


Abbildung 6.9: Der *Move*-Befehl

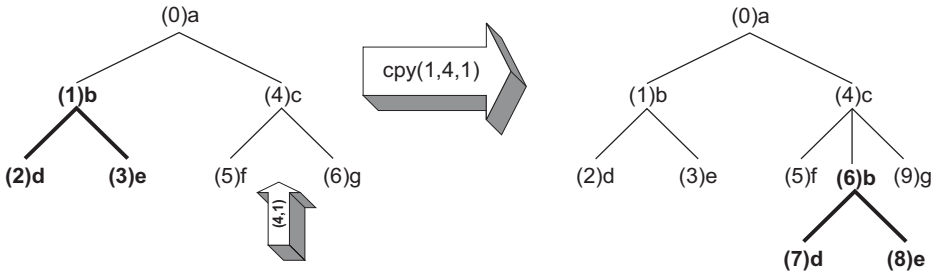


Abbildung 6.10: Der *Copy*-Befehl

chend ähnlich ist, um ihn „zu diesem verschmelzen zu lassen“, wobei der zweite Teilbaum unverändert bleibt. Diese Operation entspricht in etwa einer umgekehrten Ausführung der *copy*-Operation und versucht die Beschreibungstiefe zu erweitern, indem sie, obgleich gleichen Resultats wie die *delete*-Operation, mehr Information über die mögliche Entstehung enthält. Beispiel:
Die beiden Teilbäume mit den Wurzeln „1“ und „6“ seien hinreichend ähnlich. Der Befehl *glu(6,1)* entfernt nun den Teilbaum, dessen Wurzel der Nummer „6“ entspricht. Dieser Teilbaum wird aufgrund seiner Ähnlichkeit mit Teilbaum „1“ gewissermaßen *eingeschmolzen* (siehe Abbildung 6.11).

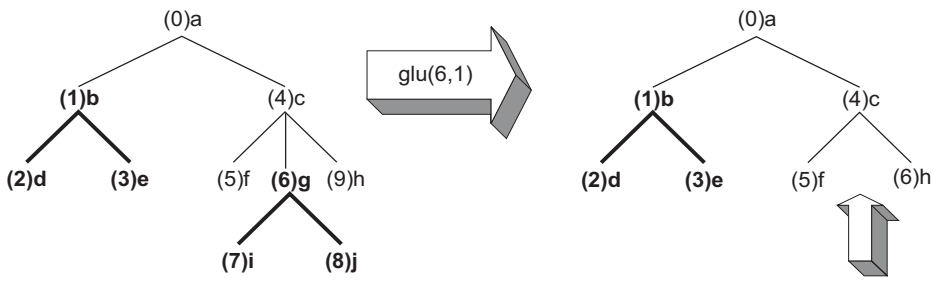


Abbildung 6.11: Der *Glue*-Befehl

#### Repräsentation der Editskripte

Die Editskripte im XML Format abzulegen hat einige Vorteile. Die einfache Darstellung und inhaltliche Strukturierung unterstützt die Weiterentwicklung von Zusatzdiensten, die Änderungsbeschreibungen nutzen. Die Erweiterbarkeit des Formats spielt eine große Rolle, da es zukünftigen Entwicklungen, wie z.B. *SCDelta*, ermöglicht werden soll, das Operationenrepertoire zu modifizieren. Schließlich soll die Nutzung des Formats nicht auf einzelne Systeme beschränkt, sondern möglichst vielen Anwendungen offen sein. Sowohl an Ausgangsdokumente, als auch an die Änderungsbeschreibungen werden ähnliche Ansprüche gestellt. Eine Konsistenz bezüglich der Formate liegt hier nahe. Die von *XTree* verwendete Original DTD ist in Anhang A zu finden. Sie wurde für die betrachtete Anwendung noch erweitert. Die erweiterte Fassung ist in Anhang B.

### 6.2.2 *SCDelta*

Der von Christoph Reichenberger [Rei94] beschriebene Algorithmus erlaubt die Erzeugung von Deltas zwischen beliebigen Dateien. Es werden keinerlei Annahmen über die Struktur der Dateien gemacht. „Besonders erfreulich ist die Tatsache, daß das neue Verfahren sich auch für Textdateien sehr gut eignet. [Rei94]“

Das Verfahren stellt eine Erweiterung des Algorithmus von Tichy [Tic84] zur Erzeugung von Deltas zwischen zwei Textdateien dar. Es werden zwei Befehlsarten verwendet: Kopier- und Anfügebefehle. Die Kopierbefehle sind für die Speicherung redundanter Daten zuständig, da sie den Verweis darstellen, daß gemeinsame Inhalte vorliegen und somit nicht zweimal abgespeichert werden müssen. Der Algorithmus arbeitet den gesamten Text durch und versucht möglichst lange gemeinsame Übereinstimmungen (*Longest Common Sequence (LCS)*) mit einem Vergleichstext zu finden, die durch Kopierbefehle dargestellt werden können[4]. Wird keine Übereinstimmung entdeckt, so wird der „neue“ Teil eingefügt.

#### Überblick

Im Foldenden wird die Vorgehensweise des implementierten Inhaltsvergleichs anhand eines Beispiels erläutert.

Um diese Suche nach den *LCS* in annehmbarer Zeit durchführen zu können wird der Inhalt vorstrukturiert. Reichenberger propagiert in seiner Arbeit die Nutzung von Teilfolgen einer bestimmten Länge, die er als *Präfixe* bezeichnet. Jedes *Präfix* wird mit einem eindeutigen Hashwert belegt. Die Teilfolgen werden zusammen mit den Dateipositionen an denen die *Präfixe* beginnen, in zwei Listen (*Hashtable* und *Linklist*) gespeichert. Der Eintrag im *Hashtable* enthält den Startindex des ersten Vorkommens des Präfixes. Gibt es noch weitere Präfixe mit demselben Hashwert, so zeigt der Eintrag des Hashwertes in der *Linklist* auf das nächste Vorkommen dieses *Präfixes*.

In Abbildung 6.12 ist ein Beispiel dargestellt. Der Anfang der Todesfuge [Cel99] soll mit einem anderen Text verglichen werden. Um das Verständnis zu erleichtern, wurden in dem Beispiel

[4] Um die Anzahl der Editierbefehle zu minimieren wird nach den längsten Übereinstimmungen gesucht.

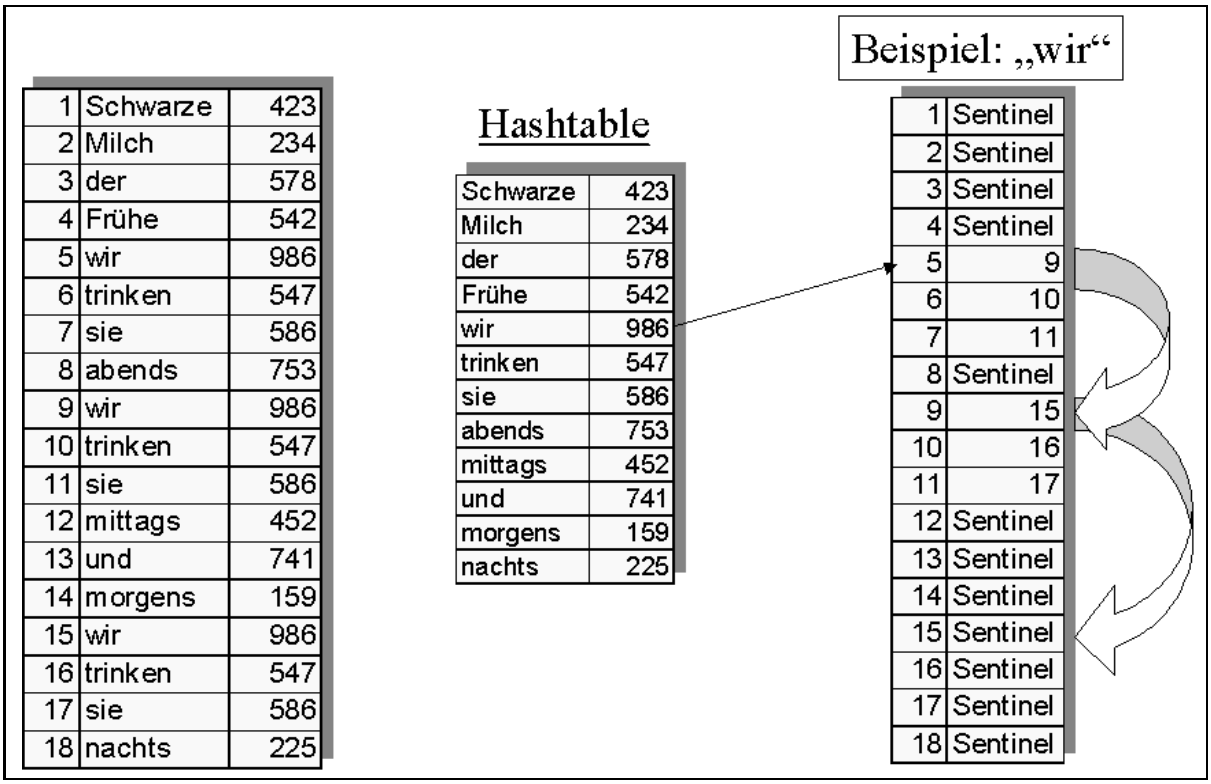


Abbildung 6.12: Hashtabelle und Linktabelle

ganze Worte als Entitäten ausgewählt. Jede Entität erhält einen eindeutigen Hashwert. Der Vergleichstext beginnt mit dem Wort „wir“. Zuerst wird für „wir“ der Hashwert berechnet und im *Hashtable* auf ein Vorkommen überprüft. Der *Hashtable* zeigt in der *Linklist* auf die Startposition des ersten Vorkommens. Hier beginnt der erste Vergleich auf der Suche nach der *LCS*. Differieren die betrachteten Texte, so wird die Länge der Übereinstimmung gespeichert und in der *Linklist* nachgesehen, ob der Beginn des Vergleichstextes („wir“) noch einmal vorkommt. An der nächsten Stelle (im Beispiel „9“) findet wiederum ein Textvergleich statt. Diese Prozedur wird solange fortgesetzt, bis kein Eintrag mehr in der *Linklist* vorhanden ist (hier durch den Platzhalter *Sentinel* symbolisiert). Aus den getätigten Vergleichen wird die *LCS* ausgewählt und als Editieranweisung gespeichert.

In Abbildung 6.13 ist der Ausschnitt eines Editskripts von *SCDelta* dargestellt. Betrachten man zunächst die erste Editieranweisung, so erkennt man, daß der zweite Knoten betrachtet wird. Dieser weist inhaltliche Unterschiede zum Vergleichsdokument auf:

1. Kopiere sechs Entitäten (in diesem Fall Sätze) vom Beginn des Dokuments.
2. Hänge die beschriebenen Inhalte an.
3. Kopiere weitere vier Entitäten von der Position sieben an.

Dieses Beispiel stellt eine klassische Anwendung für die Suche nach der *LCS* dar. Im Vergleichsdokument wurde nämlich inmitten des Inhaltsknotens eine Textpassage eingeführt. Die

Suche nach der *LCS* hat sämtliche Übereinstimmungen entdeckt und somit die gleichgebliebenen, oder übereinstimmenden Daten durch Kopierbefehle übernommen.

```
<?xml version="1.0" ?>
<!DOCTYPE es (View Source for full doctype...)>
- <es>
  - <upd nodenr="2">
      <copy copystart="0" copylength="6" delimiter="." />
    - <add delimiter=".">
        <content>This mechanism is based on the metaphor of "aging": In the same way a
          mechanical link ages by time and use gradually loosing its functionality, a digital
          link ages effected by the evolution of the link target gradually loosing its
          validity.</content>
      </add>
      <copy copystart="7" copylength="4" delimiter="." />
    </upd>
    <cpy nodenr="11" targetparent="6" childpos="0" />
    <cpy nodenr="12" targetparent="21" childpos="0" />
    <cpy nodenr="12" targetparent="15" childpos="0" />
  - <upd nodenr="7">
    - <add delimiter=".">
        <content>The general process of link aging may be summarized as follows: Each link is
          associated with a value that reflects its age. This value is set to zero upon creation
          of the link. Whenever a new version of the link target is created or another
          evolution-indicating event occurs the link age changes to reflect the new situation.
          %As the unit of aging, evolution units are used, enabling a flexible adaptation of
```

Abbildung 6.13: Editskript von *SCDelta*

Die Vorstrukturierung der Texte ist flexibel und eröffnet dem Anwender die Möglichkeit, die Granularität der Inhaltsteile (Größe oder Art der betrachteten Entitäten) selbst zu bestimmen. Hierauf wird in Abschnitt 6.2.2 noch eingegangen. Der betrachtete Algorithmus kann sowohl Dateien die aus Zeichenketten bestehen, als auch binäre Dateien miteinander vergleichen. Dies ist besonders wichtig in Bezug auf die Rolle von XML als Verwalter verschiedener Inhaltsformen. *SCDelta* wurde in Java implementiert und ist somit offen und erweiterbar. Dies erleichterte die Einbindung in den erweiterten *XTree*-Kontext.

### Hashfunktionen

Die Wahl der Hashfunktion ist wesentlich für die Effizienz des betrachteten Algorithmus. Hier geht es vornehmlich um die Vermeidung von Kollisionen. Da bei der Suche nach der längsten Gemeinsamkeit die Elemente einer Kette in der Linkliste durchlaufen werden, ist das Verfahren quadratisch in Abhängigkeit von der Länge der verknüpften Entitäten. Hieraus wird klar, daß eine möglichst gleichverteilte Hashfunktion angestrebt wird.

Weiterhin muß beachtet werden, daß die Gleichverteilung sowohl von der Textstruktur (z.B. Verhältnis semantische Texte/Zeichenketten (z.B. Programmiercodes)), als auch von der Wahl

der (Größe der) Entitäten abhängt. Es ist also nicht möglich eine bestmögliche Hashfunktion für alle Anwendungen anzugeben. Dies macht die Möglichkeit der Austauschbarkeit der Hashfunktion unabdinglich.

In [Rei94] wurden verschiedene Hashfunktionen auf ihre Tauglichkeit für das beschriebene Delta-Erstellungsverfahren untersucht. Hierbei zeigt sich, daß sogenannte additive Verfahren starke Performanzeinbußen im Bereich kleiner Entitätengrössen oder auch bei Vorkommen von Anagrammen haben. Die in dieser Arbeit implementierte Hashfunktion wird im folgenden kurz beschrieben. Der Algorithmus nach Rabin und Karp [Rei94] sieht folgendermaßen aus:

```
public long hashValue (char[] zkette){

long h = 0;
int hashlength = 256;
int hlength = 16381;

String len = new String(zkette);
long prelength = (long) len.length();

for(int i = 1; i < prelength; i++ ){
long a = (long) (h*hashlength + zkette[i]);
h = a;
}
return h;
}
```

- $h$ ergibt den errechneten Hashwert
- Die Konstante *prelength* symbolisiert die Präfixlänge
- *hashlength* stellt die Maximalgröße der Hashwerte dar
- *zkette[i]* ist die Teilentität, wobei *b[1..prelength]*
- *hlength* ist die Länge der Hashliste, d.h. die Hashwerte sollen im Intervall *1..prelength* verteilt sein.

Der beschriebene Algorithmus ist in seinem Hashwertebereich allerdings eingeschränkt. Dies schränkt die Möglichkeiten der Gleichverteilung und der Eindeutigkeit der betrachteten Entitäten ein[5]. Um den Bereich erweitern zu können, wird noch ein sogenannter *Extension*-Algorithmus nachgeschaltet, der den Wertebereich erweitert und somit die Gleichverteilung fördert. Kollisionen können somit minimiert werden. Die Methode nutzt einen Ansatz der den Bereich ausweitet. Andere Methoden operieren mit Zufallszahlen, die einen zusätzlichen Aspekt der Gleichverteilung einbringen. Im betrachteten Fall ist die vorliegende Erweiterung ausreichend, da die betrachteten Textstellen in ihrer Länge begrenzt sind.

[5] Für genauere Darstellungen sei auf [Rei94] und die vorgeschlagene Literatur verwiesen.

```
public long hashExtended(){
char[] zkette = (char[]) zkette.clone();
long h1 = hashValue(zkette);
zkette[0] = (char) (zkette[0] + 1);
long h2 = hashValue(zkette);
long hashextended = (h1*256+h2);
return hashextended;
}
```

Zum Vergleich wurde auch die von Java zu Verfügung gestellte interne Hashfunktion genutzt. In der Performanz liessen sich keine großen Unterschiede feststellen. Allerdings ist *SCDelta* mit einer doppelten Kollisionsvermeidung ausgestattet. Zusätzlich zu den Hashwerten werden auch noch die einzelnen textuellen Entitäten auf ihre Gleichheit überprüft. Dies verlangsamt den Prozeß der Deltaerstellung, verhindert aber Kollisionen gänzlich. Um die Effizienz in diesem Bereich zu erhöhen müßte eine Kollisionsvermeidung verifiziert werden. Die ist jedoch nicht Thema dieser Arbeit. Der Fokus lag vordringlich auf der Eindeutigkeit der erstellten Deltas.

### Größe der Entitäten

Reichenberger [Rei94] propagiert in seiner Arbeit Zeichenketten konstanter Länge als Entitäten. Allerdings lagen die Gründe für diese Wahl in der maximalen Performanz des Algorithmus hinsichtlich Speicher- und Laufzeiteffizienz. Bei Betrachtung der Deltas wird deutlich, daß eine Lesbarkeit ohne weitere Vorbehandlung nicht unterstützt wird. Die Entitätengröße soll flexibel sein, aber auch auf semantische Strukturen Rücksicht nehmen. Im Zuge der Evaluierung (siehe Kap. 6.3) wurden sowohl Worte als auch Sätze als Entitäten getestet. Für beide Fälle gibt es klare Anwendungsbeispiele. Im Falle eines editoralen Ansatzes zur Überprüfung der Rechtschreibung ist eine Vergleichsbetrachtung auf Wortebene sinnvoll. Steht allerdings inhaltlicher Kontext im Vordergrund so birgt die Änderungsbeschreibung mit Hilfe von Satzstrukturen Vorteile. In jedem Fall sollte die Art und Gröse der betrachteten Entitäten flexibel sein, um an die jeweiligen Anforderungen angepaßt werden zu können.

## 6.3 Bewertung des Verfahrens

Zur Durchführung einer Bewertung des Verfahrens wird der entworfene Dienst an einer realen Anwendung getestet. Als Evaluierungsumgebung wird die Entstehung eines wissenschaftlichen Artikels gewählt. Die Bewertung ist in zwei Teile aufgeteilt.

Zuerst werden quantitative Aspekte untersucht. Hierzu wird als Vergleichsmaßstab Unix-*diff* gewählt. Für den zweiten Teil wird auf den in Kapitel 2.4.2 erstellten Kriterienkatalog (Tabelle 2.1) zurückgegriffen, der sich an einer Studie für die Versionenunterstützung für strukturierte Dokumente orientiert und mehrheitlich Fragen der kooperativen Dokumentenerstellung qualtitativ betrachtet.

### 6.3.1 Evaluierungsumgebung

Da das Hauptaugenmerk dieser Arbeit der kooperativen Erstellung semistrukturierter Dokumente gilt, wird als Fallbeispiel die Entstehung eines wissenschaftlichten Artikels betrachtet. Der Artikel „Aging Links“ [NSSM00] wurde im Jahre 2000 von wissenschaftlichen Mitarbeitern am Arbeitsbereich Softwaresysteme der Technischen Universität Hamburg Harburg erstellt. Ursprünglich war die Arbeit in LaTeX verfaßt worden, wurde aber für diesen Kontext in XML umgesetzt. Es existieren Werkzeuge für die automatische Konversion der Formate, aber in diesem Fall ist eine manuelle Umsetzung vorgezogen worden. Automatische Werkzeuge setzen immer eine „Nachbearbeitung“ voraus. Da die Anzahl der getesteten Dokumente eher klein war, schien die Verwendung eines XML-Editors mit gleichzeitiger Validierung des Dokuments sinnvoller.

Von dem betrachteten Artikel standen verschiedene Entwicklungsstufen bis zur Veröffentlichungsreife zur Verfügung. Von jeweils sukzessiven Versionen wurden Deltas gebildet. Dabei wurde überprüft, daß eine Wiederherstellung einer Version aus Delta und Folgeversion (*Recovery*) informationsverlustfrei möglich ist. Dies soll den realen Anwendungsfall einer Dokumentevolution simulieren und einen Anhaltspunkt für die Anwendbarkeit des implementierten Dienstes geben.

### 6.3.2 Auswertung

Die im Kapitel 2.4.1 vorgestellten quantitativen und qualitativen Kriterien werden im folgenden getrennt angewendet und einer Bewertung unterzogen. Hierbei wird nochmal auf die Probleme der Allgemeingültigkeit eingegangen und abschliessend ein Fazit gezogen.

#### Quantitativer Maßstab

Die Laufzeit von SCDelta steigt mit wachsender Anzahl der Knoten quadratisch. Die Laufzeit von Unix-*diff* dagegen wird aufgrund der Zeilenorientierung Proportional zur Zeilenzahl länger. Dieses Verhalten konnte in der durchgeführten Verifizierung nachgewiesen werden. Die Komplexität des Baumvergleichsalgorithmus wurde von Christoph Spreen [Spr00] in seiner Arbeit ausführlich beschrieben. Ein quantitativer Vergleich ist aufgrund der verschie-

| Anzahl der Knoten | Anzahl der Zeichen | | |
|---|---|---|---|
| | Unix-*diff* | SCDelta | Differenz |
| 26/38 Knoten | 20941 | 20234 | 707 |
| 38/47 Knoten | 31898 | 19696 | 12202 |
| 47/50 Knoten | 15768 | 9459 | 6309 |
| 50/54 Knoten | 52367 | 26315 | 26052 |
| 54/62 Knoten | 54179 | 20749 | 33430 |
| 54/58 Knoten | 59032 | 19944 | 39088 |
| 58/64 Knoten | 38877 | 10814 | 28063 |
| 62/66 Knoten | 55678 | 16013 | 39665 |
| 64/64 Knoten | 41028 | 41028 | 32480 |
| 64/66 Knoten | 34450 | 12508 | 21942 |

Tabelle 6.1: Vergleich der Größe der resultierenden Editskripte

denen Komplexitäten wenig sinnvoll. Nach Hunt et al [HVT98] gehört Unix-*diff* aufgrund seiner Zeilenorientierung zu den schnellsten auf dem Markt befindlichen Algorithmen. Auch hinsichtlich der Größe der Deltas schneidet Unix-*diff* im Vergleich mit neueren Diensten zufriedenstellend ab: „[For] text files *diff* [...] reaches the effectiveness of *vdelta* and *bdiff* when it is combined with *gzip*.“.

Im Bezug auf die Größe der Deltas wird ein direkter Vergleich der Ergebnisse von SCDelta und Unix-*diff* (genaue Befehlsdarstellung in Anhang 6.14) vorgenommen. Die Ergebnisse sind in Tabelle 6.1 dargestellt. In der ersten Spalte ist die Anzahl der Knoten der zu vergleichenden Dokumente wiedergegeben (Knotenzahl 1. Dokument/ Knotenzahl 2. Dokument). Die folgenden beiden Spalten stellen die Größe der Deltas anhand der Anzahl der Zeichen dar. In der letzten Spalte ist die Differenz der Deltas in Zeichen aufgetragen. Hierbei wurde auf ein Vorzeichen verzichtet, da *SCDelta* in allen betrachteten Vergleichen kleinere Deltas aufweist. Beispiel:
In Zeile eins wurden zwei Dokumente miteinander verglichen die aus 26 bzw. 38 Strukturknoten aufgebaut sind. Für Unix-*diff* ergibt sich eine Deltagröße von 20941 Zeichen, während *SCDelta* mit 20234 Zeichen auskommt.

Mit steigender Anzahl von Knoten läßt sich einen Trend zu einer reduzierten Deltagröße erkennen. Bei Bestätigung kann somit der Speicheraufwand niedrig gehalten werden. Gleichzeitig wird eine strukturierte Lesbarkeit gewährleistet (siehe Kapitel 2.4.2). Abbildung 6.14 zeigt die Größenunterschiede der Deltas in Abhängigkeit von der Größe der betrachteten Dokumente. Hier ist zu sehen, daß die von *SCDelta* erstellten Deltas nicht notwendigerweise mit der Dokumentengröße mitwachsen. Diese Evaluierung ist nicht repräsentativ (oder gar) als Vergleich mit Unix-*diff* zu sehen, sondern soll den Dienst einen realistischem Test unterziehen. *SCDelta* kann noch optimiert werden und auch für Unix-*diff* sind sicherlich noch kleinere Deltas denkbar. Ebenso können beide Deltaformen zusätzlich durch Komprimierungsdienste verkleinert werden.

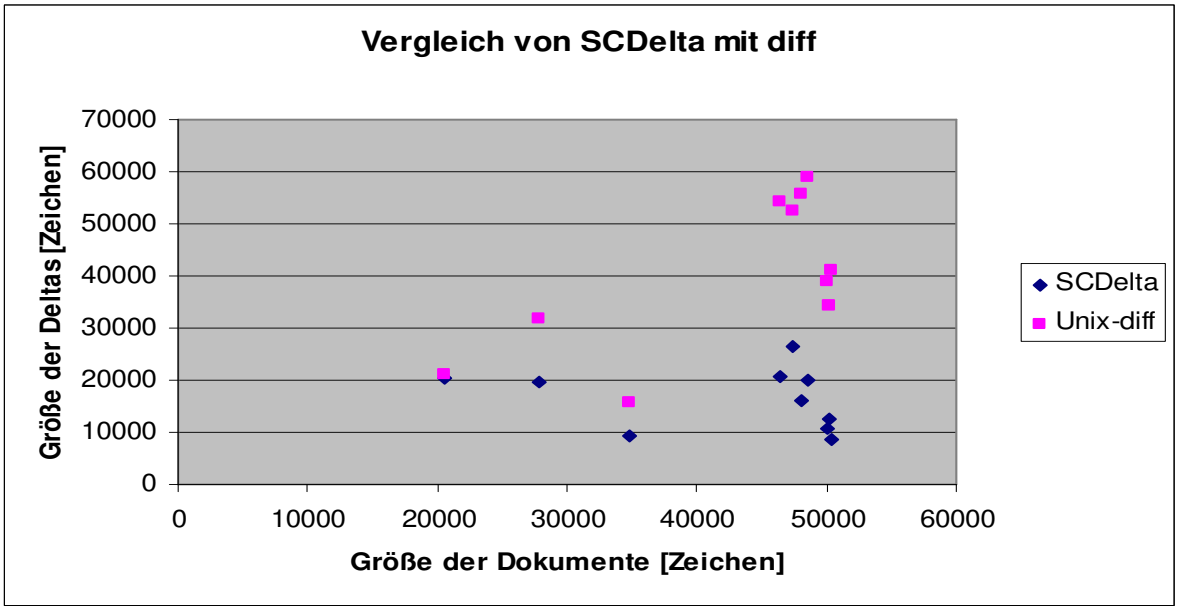


Abbildung 6.14: Vergleich *SCDelta*: Unix-*diff*

### Beobachtungen

Ein Punkt wurde im Laufe der Evaluierung deutlich: XTree kann, obwohl mit einer großen Beschreibungstiefe ausgestattet, die „reale“ Evolutionshistorie nicht getreu wiedergeben. Ausgehend von den Überlegungen in Kapitel 6.2.1 könnte es sinnvoll sein den deterministischen Ansatz durch einen probalistischen zu ersetzen, der heuristisch an die jeweilige Anwendung angepaßt werden kann. Dieser würde nicht nach der minimalen Änderungsbeschreibung suchen, sondern mit Wahrscheinlichkeiten operieren, die zwar kein optimales Resultat versprechen, aber eine verkürzte Laufzeit. Außerdem erschließt dies die Möglichkeit weniger Beschreibungselemente zu verwenden (gerade bei einem probalistischen Ansatz sinnvoll) und somit weitere Performanzsteigerungen zu erhalten.

Ebenso wie beim Strukturvergleich werden auch für den Vergleich der Inhalte die Einzelkomponenten modular programmiert, um eine Optimierung zu ermöglichen. Optimierungen sind in zahlreichen Modulen denkbar. Hier spielen zum größten Teil heuristische Überlegungen eine Rolle. In den Versuchsreihen wurde deutlich, daß auch in der eigentlichen Erstellung der Editskripte noch Optimierungen möglich sind.

Die optimale Wahl der Hashfunktion verspricht deutliche Schnelligkeitsgewinne für den Inhaltsvergleichsalgorithmus, da eine möglichst gute Gleichverteilung des Textes die Schnelligkeit des Algorithmus begünstigt. Auch dieser läßt bei Einbeziehung eines „gewichteten“ Verfahrens (siehe HCS in Kapitel 3.2.3) durchaus noch Potentiale erwarten.

Interessant ist, daß ohne eine Optimierung der Algorithmusparameter eine deutliche Reduzierung der Größe der Deltas mit wachsender Knotenanzahl erreicht wird. Dies ist sowohl relevant für die Ersparnis in der Abspeicherung, als auch für die Reduzierung der Wiederherstellungszeit.

Der Schnelligkeitsnachteil in der Delta-Erstellung muß für die betrachtete Anwendung keine allzu große Rolle spielen, da die vorgestellte Architektur ohnehin mit einer offline Delta-Erstellung arbeitet. Die zuletzt einge-checkten Versionen werden als Vollversionen gespeichert. Die Deltaerstellung findet im Hintergrund für eine ältere Version statt (siehe Kapitel 5.1.1).

### Qualitativer Maßstab

Im Folgenden wird die qualitative Bewertung anhand der vorgegebenen Kriterien (Tabelle 2.1) durchgeführt. Jeder Punkt wird einzeln erläutert und analysiert, bevor eine Abschlußbewertung erfolgt. Die von Anja Haake benutzten Termini mögen nicht immer den im Sprachgebrauch üblichen entsprechen. Für die genaue Darstellung der Begriffe sei auf [Haa96] verwiesen.

**Abwärtskompatibilität** Abwärtskompatibilität heißt, daß das Werkzeug in eine bereits bestehende Umgebung eingebunden werden kann, ohne bestehende Daten zu gefährden. Dies ist erfüllt, da keine Deltabildung notwendig, aber möglich ist, solange es sich um XML Dokumente handelt. Eine Metadatenebene der einzubindenden Dokumente ist wünschenswert, aber nicht zwingend. Versionierte und nicht-versionierte Dokumente können in der vorgestellten Architektur nebeneinander verwaltet werden.

**Aufwärtskompatibilität** Aufwärtskompatibilität meint in diesem Fall, daß bereits bestehende Dokumente in die Versionierungsumgebung eingebunden werden können. Nur wenige bisher bekannte Dienste erlauben die Konvertierung nicht versionierter Objekte in versionierte Objekte. Dies ist mit der vorgestellten Architektur kein Problem. SCDelta kann jederzeit auf einen Pool von Versionen angewandt werden und die erwünschten Deltas erstellen. Nach [Haa96] sollte der „Übergang zur Versionierung der Daten [...] schrittweise erfolgen.“. Dies entspricht auch dem Ergebnis aus Kapitel 2.4.2, nach welchem ein Großteil der Delta-Erstellung offline geschieht. SCDelta arbeitet mit einem Dokumentenstandard (XML) und kann somit in jeden Dienst integriert werden, der mit diesem Standard arbeitet. Somit kann SCDelta aufgrund seiner Schnittstellengestaltung auch in ein Merge-Tool integriert werden.

**Versionen aller Dokumentbestandteile** Um die Flexibilität zu unterstützen, sollen sowohl ganze Dokumente, als auch einzelne Dokumententeile versionierbar sein. Zum Beispiel kann sich die Frage nach früheren Fassungen einzelner Abschnitte stellen. Im Falle der betrachteten XML Dokumente ist dies möglich, da sie hierarchisch aufgebaut sind und Untermengen auch wieder leicht als XML Strukturen zu erkennen sind. SCDelta kann sowohl zur Erstellung von Deltas oder Rekonstruktion von Versionen einzelner Unterdokumente oder kompletter Dokumentstrukturen verwendet werden. Hierzu muß die Verwaltung von einzelnen Teilen von zusätzlichen Diensten zur Verfügung gestellt werden.

Interessant ist in diesem Kontext auch die Möglichkeit, die Entwicklung eines einzelnen Knoten über mehrere Versionen zu verfolgen. Ist ein Knoten eindeutig identifizierbar, so können sukzessive Deltas benutzt werden, um die „Unter“ -Historie eines Knotens zu verfolgen. Dies

kann sowohl für den laufenden Prozeß, als auch für die spätere Gesamtevaluierung einer Entstehungshistorie von Nutzen sein.

**Zugriff auf Versionen über Attribute und Beziehungen** „Bei der Suche nach früheren Zuständen einzelner Dokumentbestandteile müssen zumindest die gleichen Zugriffsmöglichkeiten zur Verfügung stehen, die für nur in einem Zustand vorliegenede Komponenten verfügbar sind." [Haa96]. Aufgrund der XML Struktur der entstehenden Deltas können sowohl die in XML abgelegten Metadaten zur Suche herangezogen werden, als auch die inhaltlichen Änderungen. Es ist sogar möglich, nach Auffinden einer Version alle weiteren Versionen zu identifizieren, in denen ein bestimmter Knoten verändert worden ist. Desweiteren sind Darstellungen einer Änderungsevolution einzelner Teile einer Version möglich (siehe auch 6.3.2). Zusätzlich können Attribute, die aus Versionierungsaktionen (wie z.B. Extraktion/Archivierung) resultieren oder die Bestandteil eines versionierten Dokuments sind, zur Navigation oder Evaluierung herangezogen werden. Hier können also zwei Metaebenen unterschieden werden, die aber in einer Dokumentstruktur abgelegt sind und somit mit einem *Retrieval*-Mechanismus angesprochen werden können.

**Zugriff auf Versionen über Entstehungszeitpunkt und Autoreninformation** Die Ablage der Metadaten in XML erlaubt sowohl den Zugriff auf die automatisch generierten Daten, die beim Ein-/Auschecken gespeichert werden, als auch auf übergeordnete Metadaten, die eine Entwicklung eines Dokuments beschreiben. Die Protokolle können durch einfaches Ändern der DTD an die jeweiligen Bedürfnisse angepaßt werden.

**Wiedernutzung von Inhalten** XML Dokumente sind hierarchisch aufgebaut. Somit ist es möglich sowohl einzelne Teile, als auch einen gesamten Kontext weiterzuverarbeiten. Insbesondere im Bereich der Deltas gibt es die Möglichkeit komplette Evolutionshistorien einzelner Knoten oder auch Knotenmengen zu extrahieren. Desweiteren können die Metadaten mit Hilfe von Standard XML Werkzeugen analysiert und als Arbeitsnachweis oder Studie verwendet werden.

SCDelta speichert sowohl den Zusammenhang (Knotenbezeichnung inklusive der Attribute), als auch die Änderung. Andere Werkzeuge wie z.B. *xdiff* leisten ähnliches, allerdings sind diese Informationen aus dem Delta selbst nicht erschließbar, sondern werden erst wieder aufbereitet. SCDelta kann dagegen aufgrund seiner XML Struktur mit Standardwerkzeugen weiterverarbeitet werden. Dies schließt insbesondere die in Abschnitt 2.3.4 ausführlich behandelten Mehrwertdienste ein.

### Beobachtung

Aus der qualitativen Bewertung ergeben sich einige Vorteile aufgrund des betrachteten XML Formats als Standard. Hierbei ist hervorzuheben, daß es sich bei XML um einen offenen Standard handelt, der die Weiterverarbeitung und Integration fördert. Außerdem erscheint es sinnvoll, eine Dokumentenerstellungsumgebung mit einem Inhaltsmanagementsformat zu gestalten.

### 6.3.3 Fazit

Besonders aus der qualitativen Auswertung geht hervor, daß SCDelta viele Möglichkeiten bietet, die von herkömmlichen Versionierungsumgebungen bis dato noch nicht geleistet werden. Obwohl verschiedene Dienste auf dem Markt existieren, erfüllen diese doch nicht die nach [Haa96] gestellten Anforderungen. Viele dieser Anforderungen können durch den betrachteten Ansatz erfüllt werden. Es wird sowohl die Aufwärts-/Abwärts-Kompatibilität, als auch die partielle Versionierung unterstützt. Außerdem stehen signifikant stärkere Such- und Bewertungsmöglichkeiten zur Verfügung die sich aus der Nutzung eines Inhaltsmanagementsformats ergeben. Somit können die gleichen Analysewerkzeuge sowohl für die betrachteten Ausgangsdokumente, als auch für die gespeicherten Deltas verwendet werden. Eine Konsequenz ist die Möglichkeit, Deltas miteinander zu vergleichen und somit das Evolutionsmanagement weiter zu unterstützen. Desweiteren wird durch die erhaltene Struktur der Inhalte die Visualisierung erleichtert.

Nachteilig ist der erhöhte Aufwand auf Seiten des Strukturvergleiches. Die Performanzschwächen treten in der vorgestellten Versionenverwaltung nicht zu Tage, da die Generierung der Deltas offline geschieht und somit nur indirekt Zeitbeschränkungen unterliegt. Dennoch scheint eine Kosten-Nutzen-Rechnung angebracht. Der in dieser Arbeit verwendete Baumvergleichsalgorithmus verspricht eine möglichst getreue Wiedergabe der Evolutionshistorie. Dies wird einerseits durch ein erweitertes Beschreibungsrepertoire[6], andererseits durch einen deterministischen Ansatz gewährleistet werden. Da die kooperative Unterstützung unter autonomen Aspekten im Mittelpunkt dieser Arbeit steht, ist eine *realistische* Evolutionshistorie ein wichtiger Qualitätsfaktor. Allerdings konnte die „realistische“ Beschreibung der Enstehungshistorie in der Evaluierung nicht nachgewiesen werden. Als Beispiel kann genannt werden, daß verschobene Textteile teilweise nicht mit einem *mov*-Befehl, sondern vielfach durch *copy* und *update* beschrieben wurden. Hier könnte eine Optimierung des Algorithmus eine Lösung sein. Allerdings wurde in der qualitativen Auswertung deutlich, daß eine ausreichende Beschreibung der Evolutionshistorie auch mit einem reduzierten Operationenrepertoire möglich ist.

Insofern scheint in einer praktikablen Versionierungsumgebung die Ersetzung des Strukturvergleichs durch einen probalistischen Ansatz mit einem reduzierten Beschreibungsrepertoire sinnvoll. Auf der einen Seite sind Effizienzgewinne in der Schnelligkeit zu erwarten, auf der anderen Seite kann durch Einsparung von Beschreibungselementen, sowohl die Erstellung, als auch die Darstellung der Deltas vereinfacht werden. Die genauere Unterstützung der Entstehungshistorie wird in diesem Fall durch die erstellten Extraktions- und Archivierungsprotokolle geleistet und kann somit auch anwendungsspezifisch leicht modifiziert werden.

Zusammenfassend läßt sich feststellen, daß der vorgestellte Ansatz zur flexiblen Versionierung semistrukturierter Dokumente im Kontext von Autonomie und Kooperation, die in der Literatur gestellten Forderungen inhaltlich weitgehend erfüllt. Die Auswahl der Dienste zur Erstellung der Änderungsbeschreibungen sollte allerdings noch an die Anforderungen einer anwenderfreundlichen Umgebung angepaßt werden. Im Kontext der vorgestellten Architektur ist eine Implementierung mit den genannten nötigen Änderungen durchaus vorstellbar.

[6] Sechs Beschreibungselemente. Viele andere Vergleichsalgorithmen kommen mit vier Operatoren aus.

# Kapitel 7

# Zusammenfassung und Ausblick

Dieses abschließende Kapitel liefert eine Zusammenfassung der im Rahmen dieser Arbeit vorgenommenen Entwürfe und Implementierungen, sowie einen Ausblick auf mögliche Erweiterungen und Verbesserungen der prototypischen Implementierung.

Versionsverwaltungen haben es mit immer komplexeren Problemstellungen zu tun. Ausgehend von der industriellen Fertigung, über die Softwareentwicklung bis hin zur Dokumentenerstellung werden Versionsverwaltungen eingesetzt. Sie helfen Informationen zu erhalten und zu verwalten und somit verfügbar zu machen.

In der Dokumentenerstellung arbeiten heutzutage oft viele verschiedene Autoren zusammen an demselben Dokument, wobei sie sich nicht notwendigerweise am selben Ort befinden. Die Koordination dieser Arbeiten ist komplex und wird von verschiedenen Diensten übernommen. Die Versionsverwaltung übernimmt ihren Teil, den Informationsfluß koordinierbar und übersichtlich zu erhalten (vgl. Kapitel 2). In diesem Zusammenhang ergeben sich durch die Fokussierung auf semistrukturierte Dokumente sowohl Einschränkungen, als auch Möglichkeiten.

Auf dem Gebiet der Versionierung existieren bereits einige Dienste. Sie gehen von verschiedenen Warten an das Thema der Versionsverwaltung heran. Eine Klassifizierung dieser Dienste ist nach verschiedenen Gesichtspunkten möglich, die in vielerlei Weise ihre Eignung begründen. Eine Auswahl und Darstellung dieser Dienste findet sich in Kapitel 3. Hier wurde auch näher auf die Verwendung der Deltaspeicherung eingegangen, welche im Fokus dieser Arbeit steht.

Das Ziel einer Versionsverwaltung, die Kooperation und Autonomie möglichst weit zu unterstützen, wurde in Kapitel 5 erläutert. Hier sind sowohl allgemeine Anforderungen beschrieben, als auch die dafür notwendigen Basisdienste. Aus diesen Überlegungen entstand der Entwurf eines Deltaerstellungsdienstes, der den oben genannten Anforderungen gerecht wird.

Bei der Implementierung des entwickelten Verfahrens wurde besonderer Wert auf die Konfigurierbarkeit, die Erweiterbarkeit, die Wiederverwendbarkeit und die Nachvollziehbarkeit gelegt. Auf der Grundlage eines generischen Vergleichswerkzeuges, welches XML Dokumente miteinander vergleicht und die Änderungen beschreibt, wurde der Versionierungsdienst *SCDelta*

entwickelt. *SCDelta* ist in der Lage Unterschiede zwischen XML Dokumenten in wählbarer Granularität zu beschreiben. Die Unterschiede werden in sogenannten Editskripten abgelegt, die ebenfalls durch ein XML Dokument beschrieben werden. Die Editskripte sind als Deltas nutzbar, welche angewendet das alte Dokument wiederherstellen können. Weiterhin stellen sie die Evolutionsinformationen des Dokuments im XML Format dar und sind somit für weitere Mehrwertdienste in Zusammenhang mit der Kooperation und der Änderungsverwaltung nutzbar.

## 7.1 Bewertung des entwickelten Systems

Der entwickelte Dienst stellt einen Basisdienst der Versionsverwaltung dar. Er ist in der Lage zwei XML Dokumente miteinander zu vergleichen und eine Folge von Änderungsoperationen zu ermitteln, die die Unterschiede beschreiben. Diese Unterschiede werden in einem Editskript zusammengefaßt, welches ebenfalls im XML Format abgelegt wird.

Die Tatsache, daß es sich bei den Daten um semistrukturierte Daten handelt, spielt eine wichtige Rolle. So wird z.B. zwischen Strukturänderungen (Einfügen von Elementen, Entfernen von Elementen, Verschieben von Elementen etc.) und inhaltlichen Änderungen unterschieden. Für den Vergleich der Strukturen findet der Dienst *XTree* von Christop Spreen [Spr00] Verwendung. Dieser wurde angepaßt und durch einen inhaltlichen Vergleich ergänzt. Die entstehenden Deltas können zur Wiederherstellung von Versionen herangezogen werden, was *SCDelta* als Deltamodul in einer Versionsverwaltung einsatzfähig macht.

*SCDelta* ist, je nach Kontext, flexibel einstellbar. Es kann gewählt werden, ob die Editskripte die Unterschiede auf Zeichenkettenbasis (mit flexibler Länge) oder auf semantischer Basis (Worte, Sätze, Absätze etc.) beschreiben. Im Hinblick auf das Augenmerk „semantischer Kontext“(autonomes Arbeiten in kollaborativen Projekten) erscheint der semantische Ansatz sinnvoll. Allerdings sind, z.B. auf editoraler Ebene, auch zeichenorientierte Beschreibungen denkbar.

Da *SCDelta* als Basisdienst einer Versionsverwaltung betrachtet wird, wird für Ein- und Ausgabedateien des Dienstes ein standardisiertes Format (XML) verwendet. Durch die Verwendung von XML für den Austausch von Informationen, sowohl für die miteinander zu vergleichenden Dokumente als auch für die erstellten Editskripte, wird weiteren Modulen aus dem Versionsverwaltungskontext die Integration erleichtert.

Auf der Grundlagen der in der Einleitung angegebenen Zielsetzungen in Verbindung mit dem in Abschnitt 2.4.1 vorgestellten Kriterienkatalog wurde *SCDelta* in Abschnitt 6.3 einer Bewertung unterzogen. Der Dienst ist aufgrund seiner deterministischen Ausrichtung und seiner großen Beschreibungstiefe nur begrenzt effizient in Bezug auf die Laufzeit. Dies ist nicht primär relevant, da nach der in Abschnitt 6.3.2 entworfenen Versionsverwaltung keine Online-Deltaerstellung notwendig ist, sollte aber dennoch bedacht werden. Im Bereich der Größe der entstehenden Deltas ergab die Evaluierung, daß die generierten Deltas im Vergleich mit auf dem Markt befindlichen Diensten (in diesem Fall Unix-*diff*) positiv abschneiden.

Der Fokus der Arbeit lag auf dem Aspekt der Dokumentenerstellung im Bezug auf Kooperation und Autonomie. Hier wurde zur Evaluierung ein Auszug eines Kriterienkatalogs von Anja Haake [Haa96] (siehe Abschnitt 2.4.2) zu Rate gezogen. Die getesten Punkte bezie-

hen sich zuweilen auf vollständige Versionierungsumgebungen, wurden aber trotzdem unter dem Aspekt der Möglichkeiten, die sich durch Integration von *SCDelta* ergeben, untersucht. Als Fazit läßt sich ziehen, daß der Ansatz von *SCDelta* als Basis einer Versionsverwaltung viele Möglichkeiten eröffnet, die bis dato noch nicht geleistet wurden. Für die partielle Versionierung und stärkere Such- und Evaluationsmöglichkeiten wurden die Grundlagen gelegt. Die Formatkonsistenz eröffnet die Möglichkeit im Deltabereich mit den gleichen Diensten zu operieren, die auch im Dokumentenbereich verwendet werden. Visualisierung und damit Übersicht werden stark vereinfacht. Der vorgestellte Ansatz schafft die Voraussetzungen die in der Literatur gestellten Anforderungen zu erfüllen (siehe Abschnitt 2.4.2).

## 7.2 Verbesserungsmöglichkeiten

Trotz umfangreicher Implementierung hat das entwickelte System Prototypcharakter. Es sind noch einige Validierungszyklen notwendig, um Produktreife zu erzielen. Bei der Auswahl der Algorithmen wurde zwar auf Performanz geachtet, aber wichtiger war die Austauschbarkeit der Komponenten, so daß Verbesserungen leicht integriert werden können.

Ein erwarteter Vorteil von *XTree* eine möglichst genaue Evolutionsbeschreibung des Dokuments erhalten zu können, konnte in der durchgeführten Evaluierung nicht bestätigt werden. Da dies ein Argument für die große Zahl von Beschreibungsoperatoren ist, können mit einem reduzierten Operationenrepertoire sicherlich kleinere Laufzeiten erzielt werden. Ebenso ist Ansatz als Kontrast zu dem momentan verwendeten deterministischen Ansatz denkbar, welcher die Fuzzy-Methoden zum Vergleich der Knoten benutzt und somit die Laufzeit weiter verkürzen könnte. Weiterhin schreibt Spreen [Spr00]: „...daß Elemente mit unterschiedlichen Elementnamen als unterschiedlich betrachtet werden.“Auch dies sollte in einem neuerlichen Ansatz berücksichtigt werden, um die Grundlage zu legen, ganze Knotenevolutionshistorien zu erstellen.

Der implementierte Inhaltsvergleich nach Reichenberger [Rei94] ist noch zu optimieren und unter Umständen durch weitere Ansätze (z.B. *Heaviest Common Sequence* siehe Kapitel 3.2.3) zu ergänzen. Insgesamt sollte das Augenmerk stärker (auch im Sinne der Versionierung) auf der Ergänzung, als auf dem Austausch einzelner (Dienst-) Komponenten oder Module liegen, da jede Änderung einer langwierigen Validierung bedarf. Unter diesem Aspekt ist es weiterhin wichtig ein Archiv von Hashfunktionen anzulegen, welche für die optimale Strukturierung der textuellen Inhalte elementar sind. Ziel muß es sein, von jedem Modul möglichst viele Alternativen zur Verfügung zu haben, um in einem Zyklus geeignete Kombinationen kontextabhängig zu bestimmen.

Ein wesentlicher Punkt der während der Evaluierung zu Tage trat, ist die eindeutige Identifizierung einzelner Elemente über verschiedene Versionsstränge hinweg. Um die Vorteile der semistrukturierten Dokumente selber, als auch der strukturierten Änderungsinformationen nutzen zu können, muß es möglich sein, einzelne Elementhistorien nachverfolgen zu können. Dies wird aber durch eine kollisionsbehaftete Knotenidentifizierung eingeschränkt. Desweiteren würde durch eindeutige Identifizierung der Knoten die Optimierung des Strukturvergleichs erleichtert.

*SCDelta* wurde in dieser Arbeit für textuelle Dateien entworfen und evaluiert. Die benutzten

Algorithmen können aber auch für binäre Dateien genutzt werden. Dies ist gerade bei der Verwendung von heterogenen Formaten in XML Dokumenten notwendig (siehe auch Abschnitt 5.4.2). Eine ausführliche Evaluierung für diesen Bereich steht noch aus.

*SCDelta* ist auch als isoliertes Modul von Interesse, da Deltas in vielen Webapplikationen Anwendung finden. Hunt [HVT98] schreibt: „The recent explosive expansion of the World Wide Web has accentuated another use of deltas: they can be used for distributing updates for software and other data. Besides saving time, space, and network bandwidth, distributing deltas has another interesting benefit: a delta ist effectively an encrypted form of the new version. It can only be decoded if the original is available. Thus software updates in the form of deltas could be made available while strongly reducing the threat of piracy. Of course, revision control is also needed for simplifying web maintenance and providing access to old web pages.“ Da XML im Internet immer größere Bedeutung als Datenaustauschformat zukommt, sind also mögliche Anwendungen auch außerhalb der Versionsverwaltung zu suchen.

# Anhang A

# DTD von XTree

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!ELEMENT es (date?, time?, fnames?, comment? (upd|ins|del|mov|cpy|glu)*)>

<!ELEMENT date EMPTY>
<!ATTLIST date
    year  CDATA #REQUIRED
    month CDATA #REQUIRED
    day   CDATA #REQUIRED>

<!ELEMENT time EMPTY>
<!ATTLIST time
    hour   CDATA #REQUIRED
    minute CDATA #REQUIRED
    second CDATA #REQUIRED>

<!ELEMENT fnames EMPTY>
<!ATTLIST fnames
    fname1 CDATA #REQUIRED
    fname2 CDATA #REQUIRED>

<!ELEMENT comment (#PCDATA)>

<!ELEMENT upd (charcontent|attribute*)>
<!ELEMENT charcontent (#PCDATA)>
<!ATTLIST upd
    nodenr CDATA #REQUIRED>

<!ELEMENT ins (node)>
<!ATTLIST ins targetparent CDATA #REQUIRED
              childpos     CDATA #IMPLIED>
```

```
<!ELEMENT del EMPTY>
<!ATTLIST del
    nodenr CDATA #REQUIRED>

<!ELEMENT mov EMPTY>
<!ATTLIST mov
    nodenr       CDATA #REQUIRED
    targetparent CDATA #REQUIRED
    childpos     CDATA #IMPLIED>

<!ELEMENT cpy EMPTY>
<!ATTLIST cpy
    nodenr       CDATA #REQUIRED
    targetparent CDATA #REQUIRED
    childpos     CDATA #IMPLIED>

<!ELEMENT glu EMPTY>
<!ATTLIST glu
    nodenr     CDATA #REQUIRED
    originalnr CDATA #REQUIRED>

<!ELEMENT node (elnode|data)>

<!ELEMENT elnode (attribute*, (node*))>
<!ATTLIST elnode
    name CDATA #REQUIRED>

<!ELEMENT data (#PCDATA)>

<!ELEMENT attribute EMPTY>
<!ATTLIST attribute
    attname  CDATA #REQUIRED
    attvalue CDATA #REQUIRED
    atttype  CDATA #REQUIRED>
```

# Anhang B

# DTD von SCDelta

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!ELEMENT es (date?, time?, fnames?, comment? (upd|ins|del|mov|cpy|glu)*)>

<!ELEMENT date EMPTY>
<!ATTLIST date
    year  CDATA #REQUIRED
    month CDATA #REQUIRED
    day   CDATA #REQUIRED>

<!ELEMENT time EMPTY>
<!ATTLIST time
    hour   CDATA #REQUIRED
    minute CDATA #REQUIRED
    second CDATA #REQUIRED>

<!ELEMENT fnames EMPTY>
<!ATTLIST fnames
    fname1 CDATA #REQUIRED
    fname2 CDATA #REQUIRED>

<!ELEMENT comment (#PCDATA)>

<!ELEMENT es (upd|ins|del|mov|cpy|glu)*>

<!ELEMENT upd (add*|copy*)*>
<!ATTLIST upd nodenr CDATA #REQUIRED>

<!ELEMENT ins (node)>
<!ATTLIST ins targetparent CDATA #REQUIRED
              childpos     CDATA #IMPLIED>
```

```
<!ELEMENT del EMPTY>
<!ATTLIST del nodenr CDATA #REQUIRED>

<!ELEMENT mov EMPTY>
<!ATTLIST mov nodenr       CDATA #REQUIRED
              targetparent CDATA #REQUIRED
              childpos     CDATA #IMPLIED>

<!ELEMENT cpy EMPTY>
<!ATTLIST cpy nodenr       CDATA #REQUIRED
              targetparent CDATA #REQUIRED
              childpos     CDATA #IMPLIED>

<!ELEMENT glu EMPTY>
<!ATTLIST glu nodenr     CDATA #REQUIRED
              originalnr CDATA #REQUIRED>

<!ELEMENT copy EMPTY>
<!ATTLIST copy copystart    CDATA #REQUIRED
               copylength   CDATA #REQUIRED
               delimiter    CDATA #REQUIRED>

<!ELEMENT content (#PCDATA)>

<!ELEMENT add (content)>
<!ATTLIST add delimiter CDATA #REQUIRED>

<!ELEMENT node (elnode|data)>

<!ELEMENT elnode (attribute*, (node*))>
<!ATTLIST elnode name CDATA #REQUIRED>

<!ELEMENT data (#PCDATA)>

<!ELEMENT attribute EMPTY>
<!ATTLIST attribute attname  CDATA #REQUIRED
                    attvalue CDATA #REQUIRED
                    atttype  CDATA #REQUIRED>
```

# Anhang C

# DTD zur Versionsarchivbeschreibung

Eine alternative DTD (Dokument Typ Definition) zur Beschreibung der Evolutionshistorie ohne Verweisstrukturen könnte folgendermaßen aussehen:

*<?xml version="1.0"encoding="ISO-8859-1"?>*
*<!ELEMENT DOKUMENT (Version|Revision|Variante)>*
*<!ATTLIST DOKUMENT Identifizierer ID #REQUIRED>*

*<!ELEMENT Version(Vorgängerliste, Nachfolgerliste, Attributliste, Data)>*
*<!ATTLIST Version Nummer CDATA #REQUIRED*
*...*
*>*

*<!ELEMENT Revision (Vorgängerliste, Nachfolgerliste, Attributliste, Data)>*
*<!ATTLIST Revision Nummer CDATA #REQUIRED*
*...*
*>*

*<!ELEMENT Variante (Vorgängerliste, Nachfolgerliste, Attributliste, Data)>*
*<!ATTLIST Variante Nummer CDATA #REQUIRED*
*...*
*>*

*<!ELEMENT Attributliste EMPTY>*
*<!ATTLIST Attributliste Name CDATA #REQUIRED*
*Datum CDATA #REQUIRED*
*Autor CDATA #REQUIRED*
*...*
*>*

*<!ELEMENT Vorgängerliste (Vorgänger*)>*

```
<!ELEMENT Nachfolgerliste (Nachfolger*)>

<!ELEMENT Vorgänger EMPTY>
<!ATTLIST Vorgänger Identifizierer ID #REQUIRED>

<!ELEMENT Nachfolger EMPTY>
<!ATTLIST Nachfolger Identifizierer ID #REQUIRED>

<!ELEMENT Data (#PCDATA)>
```

# Anhang D

# Quantitative Evaluierung

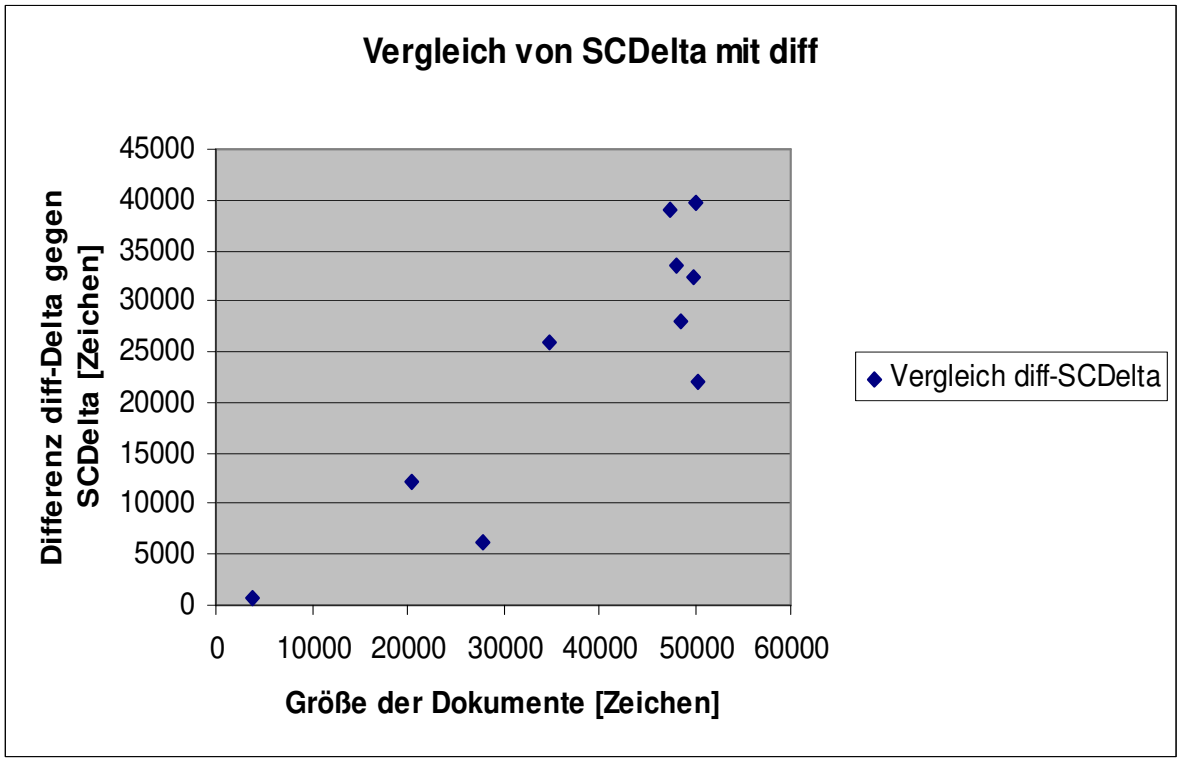


Abbildung D.1: Vorteil von *SCDelta* gegenüber Unix-*diff* in Zeichen

Für die quantitative Bewertung wurde *SCDelta* mit *diff* verglichen. Laufzeiten wurden nicht verglichen, da *SCDelta* von einer Java-Umgebung gestartet wurde, während die der *diff*-Vergleich auf einer *Sun-Workstation* durchgeführt wurde. Für die Differenzenerstellung unter Unix wurde mit folgendem Befehl operiert:

$$diff - b - t - w - c\ version_alt.xml\ version_neu.xml > alt_neu.diff$$

- b ignoriert alle überfüssigen Leerzeichen an Textanfängen oder Schlüssen, wie es auch

XML-Parser tun.

- t berücksichtigt Tabulatorbefehle in der Ausgabe, um die Lesbarkeit zu verbessern.
- c sorgt dafür, daß der inhaltliche Kontext in der unmittelbaren Umgebung ergänzt wird. Da Einordnung in den Kontext ein Aspekt in der betrachteten Arbeit ist, wurde c gesetzt.

In Abbildung D.1 ist dargestellt um wieviel kleiner die Deltas von *SCDelta* gegenüber den Deltas von Unix-*diff* mit wachsender Zahl der Zeichen in den Ausgangsdokumenten sind.

# Anhang E

# Zustand- und Änderungsorientierte Versionierung

Die Einteilung hinsichtlich zustands- oder änderungsorientierter Versionierung ist nicht immer ganz eindeutig. Anja Haake [Haa96] gibt zwei Definitionen an. Für eine noch feinere Einteilung sei auf [CW98] verwiesen.

## E.1 Zustandorientierte Versionenunterstützung

Eine zustandsorientierte Versionenunterstützung ermöglicht die explizite Erzeugung von Versionen komplexer Objekte durch die Auswahl von Versionen der Komponentenobjekte. Eine Version eines komplexen Objekts wird praktisch aus den Versionen der Komponentenobjekte konstruiert (bottom-up Verfahren der Versionenerzeugung). Unterschiede zwischen zwei Versionen können als Änderungen aufgefaßt werden die durchzuführen sind, um die eine Version in die andere zu überführen.

## E.2 Änderungsorientierte Versionenunterstützung

Eine änderungsorientierte Versionenunterstützung ermöglicht die Erzeugung von Versionen komplexer Objekte durch Änderung einer Ausgangsversion (initial die 'leere' oder 'initiale Version' des Objekts). Durch diese Änderungen können Versionen der Komponentenobjekte entstehen (top-down Verfahren der Versionenerzeugung).

# Literaturverzeichnis


[AG00] Software AG. http://www.softwareag.com/corporat/default.htm, 2000.

[Alt99] Josef Altmann. *Kooperative Softwareentwicklung: Rechnerunterstützte Koordination und Kooperation in Softwareprojekten.* Number 25. Universit"atsverlag Rudolf Trauner, Linz, 1999. Reihe C - Technik und Naturwissenschaften, Band 25.

[AN91] R. Ahmed and S.B. Navathe. Version management of composite objects in cad databases. *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, pages 218–227, may 1991.

[BG81] D.G. Bobrow and I.P. Goldstein. Layered networks as a tool for software development. *Proceedings of the 7th International Joint Vonference on AI*, pages 913–919, august 1981.

[BNLINLSP97] Lars Bendix, Per Nygaard Larsen, Anders Ingemann Nielsen, and Jesper Lai Sndergaard Petersen. Coed a tool for cooperative development of hierarchical documents. *Technical Report R-97-5012*, september 1997.

[BSF99] D. Birsan, H. Sluiman, and Stacey-Anne Fernz. Xml diff and merge tool. http://www.alphaworks.ibm.com/aw.nsf/techmain/xmldiffandmerge, 1999.

[Cel99] Paul Celan. *Todesfuge.* Rimbaud, Aachen, 1999.

[Cha00] Sudarshan S. Chawathe, 2000.

[Che00] Y. Chen. Topblend: An efficient implementation of htmldiff in java, 2000.

[Con87] J. Conklin. Hypertext: An introduction and survey. *IEEE Computing*, (20), September 1987.

[Con00] World Wide Web Consortium, august 2000.

[CW98] Reidar Conradi and Bernhard Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2), june 1998.

[Emp99] Empolis. Topicmaps. http://www.topicmaps.com/index.html, 1999.

[FRK89] D. Fafchamps, D. Reynold, and A. Kuchinsky. Proceedings of the first european conference on computer supported collaborative work. In *The dynamics of small group decision-making over the e-mail channel.* Computer Science House, Slough, september 1989.

[Haa96] Anja Haake. Versionenunterstützung für strukturierte hyperdokumente im elektronischen publizieren., july 1996.

[Hop92] J. Hoppe. Integrated management of technical dokumentation - the system sprite. *Research Reports ESPRIT, Project 2001, SPRITE*, 1, 1992.

[HVT98] James J. Hunt, Kiem-Phong Vo, and Walter F. Tichy. Delta algorithms: An empirical analysis. *ACM Transactions on Software Engineering and Methodology*, 7(2):192–214, april 1998.

[Inc99] Applian Technologies Inc., august 1999.

[Jac01] Cornelia Jacobsen. Merging: Generische dienste zum zusammenführen semistrukturierter dokumente: Strategien, visualisierung, anwendungen, february 2001.

[Kat90] R. H. Katz. Towards a unified framework for version modeling in engineering databases. *ACM Computing Surveys*, 22(4):375–408, december 1990.

[K"o92] Andrea Köhler. Entwicklung einer komfortablen, graphischen benutzungsoberfläche fr das versionen-verwaltungssystem rcs. 1992.

[KPM99] Cheryl Kirk and Natanya Pitts-Moultis. *XML black book.* Coriolis Group Books, 1999.

[Mac00] Joshua P. MacDonald. xdelta. http://www.xcf.berkeley.edu/ jmacd/xdelta.html, 2000.

[MD94] David G. Munson and Prasun Dewan. A flexible object merging framework. Technical report, University of North Carolina-Chapel Hill, 1994.

[MDFK97] Jeffrey C. Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Potential benefits of delta encoding and data compression for http. Research Report 97/4(97.4), July 1997.

[MNW98] Ethan V. Munson, Charles Nicholas, and Derick Wood. *Principles of Digital Document Processing.* Springer-Verlag Berlin Heidelberg, 1998. 4th international workshop: proceedings / PODDP 1998, Saint Malo, France, March 29-30, 1998.

[NCK+92] C.M. Neuwirth, R. Chandok, D.S. Kaufer, P. Erion, J. Morris, and D. Miller. Flexible diff-ing in a collaborative writing system. In *Proceedings of the Fourth Conference on Computer Supported Cooperative Work*, pages 147–154. ACM, ACM Press, 1992.

[NSSM00] C. Niederée, U. Steffens, J. W. Schmidt, and F. Matthes. Aging links. august 2000.

[Pre90] Vassili Prevelakis. *Object Management: Versioning Issues for Hypertext Systems.* Centre Universitaire d'Informatique, University of Geneva, D. Tsichritzis (Ed.), July 1990.

[Pre00] Pressemitteilung. Xml-delta modul. http://www.schema.de, 2000.

[Rei94] Christoph Reichenberger. *Konzepte und Verfahren für die Software-Versionsverwaltung.*, volume Reihe C - Reihe Technik und Naturwissenschaften. Universitätsverlag Rudolf Trauner, 1994.

[Sha93] Mike Sharples. *Computer Supported Collaborative Writing.* Springer Verlag, 1993.

[Spr00] Christoph Spreen. Change management für metadaten in digitalen bibliotheken, february 2000.

[Sto96] Elmar Storath. Kontextsensitive wissensbereitstellung in der konstruktion, december 1996.

[Sun00] Sun. Java dokumentation, august 2000.

[TCHN99] Naoyuki Tokuda, Liang Chen, Liang Huang, and Akira Nagai. A template-based online intelligent tutoring system for language translation. *International Symposium on Intelligent Multimedia and Distance Education*, August 1999.

[Thu00] Karsten Thurow. Ein generisches notifikations-framework, september 2000.

[Tic84] Walter F. Tichy. The string-to-string correction problem with block moves. *ACM Transactions on Computer Systems*, 2(4), november 1984.

[Tic85] Walter F. Tichy. Rcs-a system for version control. *Software-Practice and Experience*, 15(7):637–654, july 1985.

[vHP00] Arthur van Hoff and Jonathan Payne. The generic diff format. http://www.w3.org/TR/NOTE-gdiff-19970901.html, 2000.

[W3C00] W3C, august 2000.

[WL92] U.K. Wiil and J.J. Leggett. Hyperform: Using extensibility to develop dynamic, open and distributed hypertext systems. *Proceedings of the 4th ACM Conference on Hypertext*, pages 251–261, 1992.

[WL93] U.K. Wiil and J.J. Leggett. Concurrency control in collaborative hypertext systems. *Proceedings of the 5th ACM Conference on Hypertext*, pages 14–24, november 1993.