Bachelorarbeit

Tobias Hutzler

Automatisierte Revisionsübernahme bei XML-basierten Wartungsdokumenten der Lufthansa AG

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Tobias Hutzler

Automatisierte Revisionsübernahme bei XML-basierten Wartungsdokumenten der Lufthansa AG

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. rer. nat. Michael Neitzke
Zweitgutachter : Dipl. Inform. Birgit Wendholt

Abgegeben am 25. Februar 2008

Tobias Hutzler

Automatisierte Revisionsübernahme bei XML-basierten Wartungsdokumenten der Lufthansa AG

Stichworte

XML, SGML, 3-Wege-Merge, Vergleich, Wartungsdokumenten, Revisionskontrolle, Arbeitskarte, iSpec 2200, Fluggesellschaften, Lufthansa AG, Lufthansa Systems

Kurzzusammenfassung

Gegenstand dieser Arbeit ist die Revisionsübernahme bei XML-basierten Wartungsdokumenten der Lufthansa AG. Die Lufthansa-interne Wartungsdokumentation basiert in der Regel auf Dokumenten der Hersteller. Durch eine ständige Revision der Herstellerdokumente ist eine komplexe Revisionskontrolle der internen Dokumentation erforderlich. Diese Arbeit verfolgt die Idee, durch Einsatz eines 3-Wege-Merge dem Anwender einen Vorschlag zur Aktualisierung eines internen Dokuments zu berechnen. Dazu bedarf es einer fachlichen Analyse der Revisionskontrolle und den durch die ATA iSpec 2200 spezifizierten Austausch und Inhalt der Wartungsdokumente, sowie einer technischen Analyse von 3-Wege-Merge- und Vergleichs-Verfahren unter Berücksichtigung des verwendeten XML-Formates. Die Ergebnisse beider Analysen dienen dem konzeptionellen Entwurf für eine anschließende prototypische Implementierung.

Tobias Hutzler

The automated adoption of revised XML-based maintenance documents of Lufthansa AG

Keywords

XML, SGML, 3-way-merge, xmldiff, comparison , maintenance documents, revision-control, job card, iSpec 2200, airlines, Lufthansa AG, Lufthansa Systems

Abstract

The subject of this thesis is the adoption of revised XML-based maintenance documents used by Lufthansa AG. The Lufthansa internal maintenance documentation is generally based on the documentation provided by the aircraft manufacturers. Due to the ongoing revision of the documentation a complex revision-control of the internal documentation is necessary. This thesis follows the idea, to provide the user a solution for the actualisation of an internal document, by means the 3-way-merge. This requires a functional analysis of the revision-control process and ATA iSpec 2200 standard for document exchange, as well as a technical analysis of xml-based 3-way-merging and diffing. The results of both analysis serve for the conceptual design of the then following implementation of the prototype.

Inhaltsverzeichnis

1. Einführung

Sicherheit geht vor: In der Luftfahrt ist das die oberste Maxime. Insbesondere bei der Reparatur und Wartung von Flugzeugen legen Fluggesellschaften besonderen Wert auf die lückenlose Dokumentation sowie Bereitstellung aller technischen Spezifikationen und Instandhaltungsdokumente. Techniker, die Wartungen oder Reparaturen an Flugzeugen durchführen, müssen ständig Einsicht in die Wartungs- und Instandhaltungsdokumente haben.

Abbildung 1.1.: Techniker bei Wartungarbeiten eines Triebwerks. Auf dem Monitor ist das entsprechende Wartungsdokument zu sehen. (Quelle: [LSY07])

Grundlage für die technische Dokumentation bei der Instandhaltung und Wartung sind die vom Hersteller (z.B. Airbus oder Boeing) bereitgestellten und von der Aircraft Transport Association (ATA) spezifizierten Dokumente, wie das Aircraft Maintenance Manual (AMM). Diese Herstellerdokumente erscheinen in Zyklen von drei bis sechs Monaten in einer neuen Revision.
Initiiert wird eine neue Revision der Herstellerdokumente durch Verbesserungen und Änderungen am Flugzeug oder dessen Komponenten, neuer Sicherheitsanforderungen oder neuer Gesetzesvorgaben. Die Herstellerdokumente unterliegen dadurch einer ständigen Nachbesserung und Optimierung, welche auch an die Betreiber, die Fluggesellschaften, weitergegeben werden müssen.
Die Revisionierung hat für die Fluggesellschaften zur Folge, ihre internen Dokumente, die

z.T. auf den Herstellerdokumenten basieren, in diesen Zyklen nachzubessern und inhaltlich zu aktualisieren. Dokumentation, die Lücken oder durch nicht aktualisierten Inhalt Fehlinformationen aufweist, auf deren Basis Wartungen am Flugzeug durchgeführt werden, stellen ein extrem hohes Sicherheitsrisiko dar. Der wirtschaftliche Schaden, der durch ein zu lang im Hangar stehendes Flugzeug entsteht, ist immens. Handelt es sich hierbei um Vorgaben durch die Flugsicherheitsbehörde, bleibt oft nur wenig Zeit die technische Dokumentation bis zum Stichtag auf den neusten Stand zu bringen.
Besonders betroffen sind in diesem Bereich, neben dem Maintenance Planning Document (MPD)[1], die Arbeitskarten (eng. job cards). Eine Arbeitskarte basiert auf einem AMM-Task und beschreibt, wie ein bestimmter Arbeitsablauf durchgeführt werden muss. Die Aufgabenbeschreibung und Illustrationen werden aus dem AMM übernommen und durch Ingenieure (die Autoren) an den internen Wartungsvorgang angepasst und optimiert.
Die Auslieferung der Herstellerhandbücher erfolgt meist in digitaler Form. Dabei werden Formate wie *Standard Generalized Markup Language*[2] (SGML) oder *Extensible Markup Language* (XML, siehe 2.2) genutzt. Die digitalen Informationen zur Flugzeugwartung und -instandhaltung werden bei der Lufthansa in einem Dokumenten-Management-System (DMS) gehalten und stehen dem gesamten Konzern elektronisch zum Abruf bereit. Im Folgenden wird das Problem der Revisionskontrolle der internen Dokumentation betrachtet und beschrieben.

1.1. Problembeschreibung

Ein AMM besteht in der Regel aus ca. 7000-10000 Tasks. Ca. 20 Prozent dieser Tasks sind revisioniert [3]. Der Aufwand für die Ingenieure lässt sich leicht abschätzen, wenn man sich die Flotte der Lufthansa verdeutlicht. Die Lufthansa besitzt ca. 20 verschiedene Flugzeugtypen[4]. Für jeden Flugzeugtyp existiert ein eigenes AMM.
Was bedeutet dies für die Aktualisierung der internen Dokumentation?
Die Ingenieure müssen bei jeder neuen Revision sämtliche, revisionierte Arbeitskarten pro Flugzeugtyp inhaltlich auf ihre Korrektheit und Vollständigkeit überprüfen und gegebenenfalls anpassen.
Im Detail betrachtet, muss einerseits ein Vergleich zwischen der Vorgängerrevision und der aktuellen Revision des entsprechenden AMMs stattfinden, um die Änderungen des Tasks zu

[1] Das MPD beinhaltet die zeitliche Wartungs- und Instandhaltungsintervalle, Angaben zum Arbeitsaufwand und Anzahl der benötigten Techniker - allgemein formuliert: wann, was zu tun ist.
[2] Standard Generalized Markup Language: Der ISO Standard 8879 entwickelt 1986 um den elektronischen Austausch und die Publikation von Textdokumenten zu unterstützen.[Sed02, S.543]
[3] Es werden Revisionsmarken wie Add, Delete, Revise, oder Changed gesetzt siehe dazu Kapitel 3.3.3
[4] Vergleich: `http://www.lufthansa-financials.de/servlet/PB/menu/1014422_l1/index.html` Stand: 15.10.2007

identifizieren. Andererseits dürfen aber die manuellen Änderungen der Arbeitskarte durch die Autoren nicht verloren gehen. Auf Basis dieser Informationen werden die Änderungen aus den beiden unterschiedlichen Quellen in eine neue Arbeitskarte überführt.
Während bei der herkömmlichen Änderungserkennung zwei Dokumente direkt miteinander verglichen werden, um die Unterschiede festzustellen und durch Revisionsmarken kenntlich zu machen, ist dies bei der prüfenden Wiederdurchsicht von Arbeitskarten nicht möglich. Der Unterschied ist klar ersichtlich, da Änderungen aus zwei verschieden Quellen gegenüber dem Ausgangsdokument berücksichtigt werden müssen. Die Konsequenz daraus ist, dass ein herkömmlicher Vergleich kein ausreichendes Ergebnis liefert und man Änderungen aus zwei Derivaten gegenüber dem Ursprungsdokument betrachten muss. Dieser Vorgang ist als „3-Wege-Merge“ bekannt.
Allgemein lässt sich der „3-Wege-Merge“ wie folgt beschreiben:
Ausgangssituation ist die Originalversion einer Datei (Base) und zwei verschiedener Versionen (Derivate) des Originals. Die Änderungen beider Derivate zum Original werden in eine neue Datei (Merge) zusammengeführt.[Lin01, S. 27] Überschneidungen von Änderungen (z.B.: in beiden Versionen wurde der Titel überarbeitet) werden als Konflikt bezeichnet, welcher vom Initiator aufgelöst werden muss. Anders formuliert: Konflikte sind Änderungen, welche vom Softwaresystem nicht aufgelöst werden können [CSFP07, S. 6]. Die folgende Abbildung verdeutlicht anhand eines einfachen Beispiels das Grundprinzip des „3-Wege-Merge“. In Abbildung 1.2 ist das Originaldokument mit den Zeilen 1 und 2 zu sehen. In

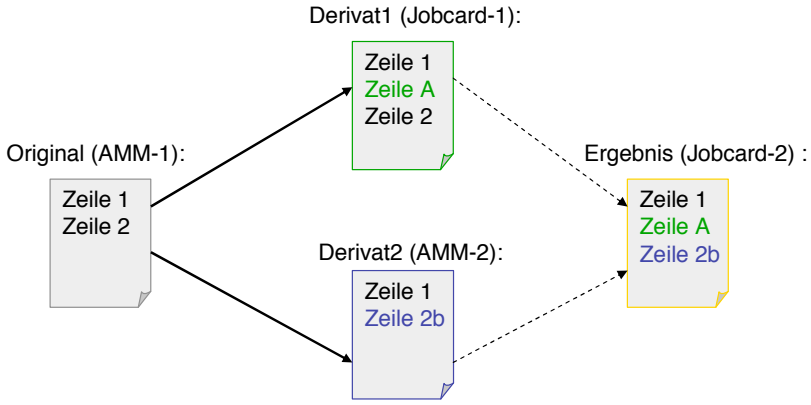


Abbildung 1.2.: Schematische Darstellung des 3-Wege-Merge.

Derivat1 wird die Zeile A nach Zeile 1 eingefügt (add). In Derivat2 wird die Zeile 2 in Zeile 2b geändert (change). Das Ergebnis liefert eine zusammengeführte Datei, die beide Änderungen beinhaltet.
Da an dieser Stelle das Problem beschrieben ist, wird im folgenden Abschnitt auf die Ziele dieser Arbeit eingegangen.

1.2. Zielsetzung

Ziel dieser Arbeit ist die Vision: „Die Revisionskontrolle der Arbeitskarten durch den 3-Wege-Merge zu unterstützen“ zu verfolgen und prototypisch umzusetzen.
Hierfür ist eine fachliche Analyse der Erstellung und Revisionskontrolle der Arbeitskarte, als auch eine Analyse der fachlichen Dokumente[5] und die Analyse der Spezifikation ATA iSpec2200 [ATA06], welcher den fachlichen Dokumenten zugrunde liegt erforderlich, um die fachlichen Anforderungen für eine software-gestützte Revisionskontrolle festzuhalten.
Darüber hinaus gelten für XML-Dokumenten aufgrund ihrer internen Struktur [W3C06a, 3 Logical Structures] besondere Anforderungen an Vergleichs- und 3-Wege-Merge-Programme. Die technischen Anforderungen gilt es zu bestimmen. und anhand dieser, die Programme *DeltaXML core/sync* [6] und *3DM Tool*[7] auf ihre Vergleichs- und 3-Wege-Merge-Fähigkeit hin zu analysieren.
Anhand der fachlichen als auch technischen Anforderungen kann dann ein Entwurf für die anschließende prototypische Implementierung erstellt werden.

Im nun folgenden Abschnitt wird der Leser mit dem inhaltlichem Aufbau dieser Arbeit vertraut gemacht, um gezielt in die für ihn interessanten Aspekte dieser Arbeit einzusteigen.

1.3. Inhaltlicher Aufbau dieser Arbeit

Kapitel 2 „Grundlagen“ legt erforderlichen Grundlagen für diese Arbeit dar.
Kapitel 3 „Analyse“ unterteilt sich in eine fachliche Analyse und technische Analyse. Die fachliche Analyse umfasst die Analyse des Erstellungs- und Revisionskontrollprozesses, der Dokumentarten und der Spezifikation iSpec 2200. Technisch werden die Besonderheiten von XML-Dokumenten, Vergleichs- und 3-Wege-Merge-Programmen analysiert. Aus den Ergebnissen der beiden Analysen werden die Anforderungen an den Entwurf und die prototypische Implementierung festgelegt.
Kapitel 4 „Design“ legt unter Berücksichtigung der gefundenen Anforderungen den Entwurf für eine prototypische Implementierung fest. Anschließenden werden in Kapitel 5 ausgewählte Aspekte der Realisierung des Prototypen vorgestellt.
Abschießend erfolgt eine Zusammenfassung der Arbeit. Dabei wird rückblickend der aktuelle Stand der Entwicklung dargelegt und Ausblicke gegeben. Abschließend wird die Arbeit bewertet.

[5] die Arbeitskarte, AMM, Temporary Revision und Service Bulletin (siehe 3.2)
[6] `http://www.deltaxml.com/dxml/products/index.html` aufgerufen: 21.02.2008
[7] `http://tdm.berlios.de/3dm/doc/index.html` aufgerufen: 21.02.2008

2. Grundlagen

In diesem Kapitel werden für das Verständnis und die Umsetzung dieses Themas notwendige Grundlagen aufbereitet. Der Abschnitt Bäume erläutert kurz die Eigenschaften eines Baums und dient hier als Grundlage für weitere Themen die in dieser Arbeit betrachtet werden. Die Extensible Markup Language, kurz XML, wird unter den für diese Arbeit interessanten Gesichtspunkten vorgestellt. Der restliche Teil dieses Kapitel widmet sich dem theoretischen Hintergrund für den Vergleich und dem 3-Wege-Merge von XML-Dokumenten. Es werden weiterhin Literaturangaben und Hinweise auf Arbeiten im angrenzenden Themenbereich gegeben.

2.1. Bäume

Ein Graph $T = (V, E)$ heißt *Baum* wenn er zyklenfrei und zusammenhängend ist. Meist wird ein bestimmter Knoten $w \in V$ als *Wurzel* bezeichnet. Da T zyklenfrei und zusammenhängend ist, gibt es für v_1 und v_2 genau einen Weg der diese Knoten verbindet. Für jeden Knoten $v \neq w$ gibt es genau eine Knoten v auf dem Weg zur *Wurzel* nächstliegenden Knoten $p(v)$ der als *Vater* von v bezeichnet wird. Alle anderen benachbarten Knoten v bezeichnet man als *Kinder* $c(v)$. Ein Knoten v der keine *Kinder* hat wird als *Blatt* l bezeichnet. Alle Knoten die weder *Blatt* noch *Wurzel* sind, werden als *innere Knoten* bezeichnet. Häufig werden *beschriftete* Bäume betrachtet: sei L eine Menge von Bezeichnern, dann gibt man zusätzlich zu der Menge von Knoten und Kanten ein Funktion $l : V \rightarrow L$ als *Beschriftung* an, welche jedem Knoten eine Bezeichner der Menge L zuordnet. Sei m ein Knoten eines Baumes B, dann bezeichnet $T(m)$ den *Teilbaum* von B, dessen *Wurzel* m ist. [Sed02, Lin01]

2.2. XML - Extenible Markup Language

Die Extensible Markup Language ist eine Auszeichnungssprache um hierarchisch strukturierte Daten darzustellen. Die vom World Wide Web Consortium (W3C) [W3C07] veröffentliche XML-Spezifikation, definiert eine Metasprache, welche es ermöglicht durch strukturelle und inhaltliche Einschränkungen anwendungsspezifische Sprachen zu definieren. Für die

Deklaration der strukturellen und inhaltlichen Einschränkungen werden entweder die Auszeichnungssprachen Document Type Definition[1] (DTD) oder XML-Schema verwendet. XML hat sich aus dem bekannten Standard SGML entwickelt und bildet eine Teilmenge dieses Standards. Ziel, der XML-Entwicklung war es die Komplexität des SGML Standards zu verringern, bei möglichst gleichbleibender Mächtigkeit, welche die SGML Sprache liefert. Die Anforderung einer internetfähigen Metasprache hat ebenfalls entscheidend die Entwicklung von XML forciert (SGML ist nicht internetfähig). Einsatz findet XML besonders im Austausch zwischen verschiedenen IT-Systemen. Ebenso eignet sich die Metasprache um XML-Anwendungen im Bereich der Technischen Dokumentation zu realisieren. Da sich durch XML klar der Inhalt vom Layout trennen lässt, erfährt XML zunehmender Beliebtheit bei der Erstellung von technischen Dokumentationen. [W3C06a]

2.2.1. Aufbau eines XML-Dokumentes

Die Hauptbestandteile eine XML-Dokumentes sind der Prolog [W3C06a, 2.8 Prolog] und das Wurzelelement. Ein XML-Element (auch das Wurzelelement) besteht aus einem Start-Tag und End-Tag oder einem Empty-Tag. Zwischen dem Start-Tag und End-Tag können Zeichenketten oder weitere XML-Elemente auftreten. Es existieren neben den XML-Elementen noch Kommentare und Processing-Instructions, welche aber hier keine Erwähnung finden. Jedes Element kann über eine Attributliste verfügen, die an ein Start-Tag oder Empty-Tag gebunden werden kann. Attribute bestehen aus Name-Wert-Paaren. [W3C06a, 3. Logical Structures]

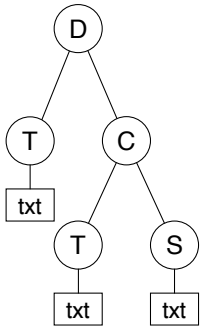
2.2.2. Baumrepräsentation eines XML-Dokumentes

Von besonderem Interesse für diese Arbeit ist, dass ein XML-Dokument in einem Baum überführt werden kann. Dies bildet die Grundlage auf denen bekannte Vergleichs- und 3-Wege-Merge-Algorithmen aufbauen. Dies soll an einem einfache XML-Dokument verdeutlicht werden.

```
1 <document autor="'Tobias Hutzler"' type="'BA"'>
2   <title>Revisionskontrolle</title>
3   <chapter nr="'1"'>
4     <title>Einführung</title>
5     <section>Problembeschreibung</section>
6   </chapter>
7 </document>
```

Listing 2.1: Einfaches XML-Dokument

[1] Document Type Definition definiert und deklariert in Form von Regeln die logische Struktur einer XML-Anwendung [Sed02, S.501]



Label	*Type*	*Name*	*Text*	*Attribute(Name=Wert)*
D	Element	Document		autor=Tobias Hutzler type = BA
T	Element	title		
txt	Text		Revisionskontrolle	
C	Element	chapter		nr=1
T	Element	title		
txt	Text		Einleitung	
S	Element	section		
txt	Text		Problembeschreibung	

Abbildung 2.1.: Der resultierende Baum und die Zuordnungstabelle des XML-Dokumentes

Die Abbildung 2.1 zeigt den resultierenden Baum und die Zuordnungstabelle der einzelnen Elemente.

2.2.3. Klassifikation von XML-Dokumenten

XML-Dokumenten lassen sich je nach eingesetztem Anwendungsfall und ihrer Struktur in dokumentzentrierte und datenzentrierte Dokumente unterteilen.

- dokumentzentriert: Dokumentzentrierte XML-Dokumente sind an Textdokumente angelehnt, die eher für den Mensch als Leser ausgelegt sind. Sie weisen oft tiefe hierarchische Strukturen auf (Document, Chapter, Section, Subsection, Subsubsection und Paragraph). Die Reihenfolge der Element ist in den meisten Fällen von Bedeutung.
- datenzentriert: Datenzentrierte XML-Dokumente sind eher für die Weiterverarbeitung von Maschinen gedacht. Die Daten können ebenfalls stark strukturiert sein, beschreiben aber in erster Linie die Beziehung zwischen einzelnen Entitäten. Bei datenzentrierten Dokumenten spielt die Typisierung der Entitäten eine wichtige Rolle.

Bei Wartungsdokumenten der Lufthansa handelt es sich um dokumentzentrierte Dokumente, bei denen die Reihenfolge der Elemente ein Rolle spielt.

2.3. Motivation der Grundlagen für den Vergleich- und 3-Wege-Merge-Algorithmen von XML-Dokumenten

Die nächsten Grundlagenabschnitte soll den Leser in die Kernthematik dieser Arbeit einführen. Es werden hier wichtige Begriffe und Konzepte veranschaulicht, die für den weiteren Teil

der Arbeit essenziell sind.
Die folgenden Abschnitte dienen dazu Problemklassen bei der jeweiligen zugrunde liegenden Dokumentstruktur als auch Problemklassen der eingesetzten Vergleichs- und 3-Wege-Merge-Algorithmen festzulegen. Durch diese Klassifizierung ist es möglich die Komplexität der jeweiligen Klassen einschätzen zu können.
Es werden die Algorithmen La-Diff[2] für den Vergleich von geordneten Bäumen und der 3-Wege-Merge Algorithmus des 3DM Tools für das Zusammenführen von geordneten Bäumen vorgestellt.

2.4. Berücksichtigungsgrad der jeweiligen Dokumentstrukturen

Programme, die allgemein Informationsbeständen[3] miteinander vergleichen oder zusammenführen, berücksichtigen die Struktur von Informationsbeständen. Ein Informationsbestand kann im Rahmen dieser Arbeit ein Textdokument oder auch ein XML-Dokument sein. Einer Klassifizierung nach diesem Kriterium liegt der Gedanke zugrunde, dass sich Informationsbestände einer Kategorie aus Elementen bestimmter Arten in bestimmter Weise zusammensetzen. Die Elemente, aus denen ein Textdokument besteht, werden aus Zeilen - also als Zeichenkette repräsentiert - die sich wiederum aus einzelnen Wörtern zusammensetzen. Hingegen werden Elemente einer hierarchischen Struktur als Knoten in einem Baum repräsentiert. Im Folgenden werden zwei Grade der Berücksichtigung der Dokumentstruktur beschrieben:

Zeilenorientiert: Programme, wie das *Unix Diff*, setzen beim Vergleich voraus, das es sich um Text-Dokumenten handelt, welche sich als Folge von Zeilen beschreiben lassen. Die Dokumente werden zeilenweise verglichen. Für Dokumente dieser Art, wie bspw. Programmcode erreichen diese Programme ein ganz aussagekräftiges Ergebnis.

Baumorientiert: Programme, wie *Deltaxml core/sync* und das *3DM Tool*, betrachten die zu vergleichenden Informationsstrukturen als Bäume, welche sich aus Knoten zusammensetzen.

[2] Diese Algorithmus wurde für den Vergleich von LaTeX-Dokumenten entwickelt
[3] Informationsbeständen, werden hier ganz allgemein betrachtet, es gibt

2.5. Differenziertheit in Bezug auf die Auswertung der Änderungsdarstellung

Der Grad der Differenziertheit der ermittelten Änderungen stellt ein weiteres Kriterium für Vergleichs- und Zusammenführungsprogramme dar.
Während einige Programme nur erkennen, dass Änderungen stattgefunden haben, sind andere in der Lage diese Änderungen nachzuvollziehen. Hierbei kann zwischen den nachvollziehbaren Änderungen differenziert werden. Im folgenden Abschnitt wird gezeigt, dass Änderungen als eine Abfolge von elementaren Operationen dargestellt werden, um einen Informationsstand in den anderen zu überführen. Diese Operationen werden auf die Elemente eines Informationsbestands angewandt. Betrachtet man ein Textdokument als Informationsbestand, so kann sich beispielsweise eine Operation auf einzelne Zeichen, Worte, Zeilen oder ähnliches beziehen, je nachdem, wie fein die *Granularität* der Betrachtung ist. Wenn Operationsfolgen (sog. *edit-script*) verwendet werden, lassen sich die Programme nach Umfang ihres Operations-Kataloges unterscheiden. Ein Operationskatalog ist die Menge von Änderungsoperationen, aus denen die durch ein edit-script beschriebene Folge von Änderungsoperationen bestehen kann. Bei *Deltaxml core/sync* werden die Operationen *insert*, *delete* und *updated* verwendet. Beim *3DM Tool* hingegen stehen die Operationen *insert*, *delete*, *updated*, *move*, *copy* und *glue* (*uncopy*) zur Verfügung. Die Funktionsweise der ersten fünf Operationen sollte intuitiv verständlich sein, während zur Operation *glue* an dieser Stelle nur folgendes erwähnt sei: Die Operation *glue* ist die Umkehrung zur *copy*-Operation. Die Ausführung einer *glue* Operation entfernt eines von zwei sehr ähnlichen Elementen, so dass man von einem *Einschmelzen* sprechen kann.
Mit dem Ziel eine Klassifizierung der Vergleichsprogramme vorzunehmen ist die Einteilung der Operationen in Gruppen sinnvoll. Dabei werden drei Gruppen gebildet, deren erste die Grundoperationen *insert*, *delete* und *update* enthält, die zweite um die *move*-Operation erweitert wird und schließlich die dritte Gruppe die zusätzlich zu den anderen Gruppen die Operationen *copy* und *glue* einhält. Darauf aufbauend können vier Gruppen von Änderungserkennungsprogrammen klassifiziert werden:

Programme, die nur Änderungen erkennen: Es handelt sich hierbei um Programme die Erkennen, dass sich ein Informationsbestand (Dokument) geändert hat, aber nicht wissen was sich geändert hat.

Programme, die Einfügen, Löschen und Ändern von Elementen erkennen: In diese Gruppe kann beispielsweise *Deltaxml core/sync* eingeordnet werden. Hier wird der Anwender durch Hervorhebung der entsprechenden Operation auf eine Änderung hingewiesen.

Programme, mit zusätzlichem Verschieben: Diese Programme erkennen neben den Grundoperationen noch das Verschieben von Elementen. Da diese Programme auch

auf Bäumen arbeiten, kann auch das Verschieben von Teilbäumen beschrieben werden.

Programme, mit den zusätzlichen Operationen Kopieren und Einschmelzen: Diese Programme sind in der Lage gesamte Teilbäume zu kopieren oder auch zu verschmelzen.

Im folgenden Abschnitt wird nun genauer auf die Funktionsweise der Algorithmen eingegangen und die einzelnen Änderungsoperationen vorgestellt. Dem Leser soll an dieser Stelle eine Einführung in die Theorie der Vergleichs- und Zusammenführungsoperationen gegeben werden. Da Deltaxml core/sync ein kommerzielles Produkt und der Sourcecode nicht vorliegt, können auf die Implementierung und die verwendeten Algorithmen nur Mutmaßungen angestellt werden. Deshalb wird am Beispiel von *La-Diff* und *MH-Diff* das Verständnis für die Vergleichsalgorithmen geschärft.

2.6. Vergleichs-Algorithmen

In vorherigen Abschnitt wurde eine Klassifizierung von Vergleichsprogrammen und Informationsbeständen vorgenommen. Diese bedienen sich unterschiedlicher Algorithmen um Änderungen zu erkennen. Einige werden hier genauer beschrieben, um die Funktionsweise zu verstehen.
Die in diesem Abschnitt vorgestellten Algorithmen unterscheiden sich im Umfang, in welchem sie auf die Struktur der Informationsbestände eingehen und welche Operationen sie hierfür nutzen. Grob kann man die Algorithmen in zwei Kategorien einordnen:

Sequenzielle Algorithmen: Solche, die die Informationsbestände als Sequenz betrachten. Darunter fallen Algorithmen, die im wesentlichen Sinne Listen von Elementen miteinander vergleichen(z.B. Zeilen in einem Textdokument). Dies wird in dieser Arbeit als linear bezeichnet.

Baumorientierte Algorithmen: Algorithmen, die die Informationsbestände als hierarchisch strukturiert betrachten, vergleichen zwei Bäume miteinander. Es muss bei baumorierntieren Algorithmen noch zwischen geordneten und ungeordneten Bäumen unterschieden werden. Sprich, ist die Reihenfolge der Geschwisterknoten beim Vergleich von Bedeutung oder nicht.

Im ersten Teil dieses Abschnittes wird auf die Repräsentation der Differenzen zweier Informationsbestände eingegangen. Im zweiten Teil wird der Vergleich von Bäumen näher betrachtet. Auf die sequenziellen Algorithmen wird nicht näher eingegangen.

2.6.1. Darstellung von Differenzen

Allgemein lässt sich die Erkennung von Änderungen durch die vier Teilschritte Analyse, Zuordnung, Repräsentation der Unterschiede und Bewertung der gefundenen Unterschiede unterteilen. Diese sind hier kurz beschrieben:

Analyse: Die zu vergleichenden Informationsbestände werden auf ihre Struktur hin analysiert.

Zuordnung: Bei der Zuordnung wird versucht möglichst große ähnliche Teile der jeweiligen Informationsbestände einander zuzuordnen. Bei Sequenzen sind es möglichst lange Teilsequenzen (*LCS*) und bei Bäumen sind es möglichst gleiche Teilbäume. Je größer die Zuordnung der Teile, desto kleiner sind in der Regel die Änderungen.

Repräsentation der Unterschiede: Nachdem die Gemeinsamkeiten bekannt sind, können Unterschiede herausgearbeitet werden und beispielsweise in Form eines *edit-script* (Folge von Änderungsoperationen) dargestellt werden.

Bewertung der gefundenen Unterschiede: Da oft mehrere Möglichkeiten existieren um Differenzen zu beschreiben, werden diese nach bekannten Kriterien (Anzahl und Kosten für die Operationen) bewertet. Diese Bewertung ist die Grundlage für die Auswahl einer Änderungsbeschreibung.

Im Folgenden wird anhand des *edit-script* näher auf die Repräsentation der Unterschiede eingegangen.

2.6.1.1. Edit-script

Die Differenz zweier ganzer Zahlen lässt sich durch eine dritte Zahl angeben. Für die Darstellung der Unterschiede zwischen strukturierten Informationsbeständen bedarf es hingegen komplexer Strukturen zur Angabe ihrer Differenz. Der Unterschied zweier Informationsbestände A und B wird häufig zur einer Sequenz von Operationen $es = (op_1, ...op_n)$ mit $op_i \in OPR$ beschrieben [ZS89, SZ90, CRGMW96] (die Menge *OPR* bildet dabei den Operationskatalog). Derartige Sequenzen werden als *edit-script* oder als Änderungsfolge bezeichnet.
Eine einzelne Operation op formt einen Informationsbestand A_0 in einen Informationsbestand A_1 um ($A_0 \rightarrow A_1$). Die Ausführung einer Folge von Operationen $es(op_1, op_2, op_3)$ auf einen Informationsbestand kann durch sequenzielles Anwenden der einzelnen Operationen auf den Informationsbestand verstanden werden. Diese Operationsfolge ist ein *edit-script* der A_0 in A_1 transformiert [CGM97].

Man kann sich vorstellen, dass es mehrere Möglichkeiten von Änderungsabfolgen gibt um einen Informationsbestand in eine anderen zu überführen. Nehmen wir die einfache Zeichenketten $A_1 =' abc'$ und $A_2 =' abb'$. Dann wäre es möglich über folgende Operationsfolgen die 1. Zeichenkette in die 2. Zeichenkette zu überführen.

- Ändere das 3. Zeichen von *„c“* in *„b“* (*update*).
- Lösche das 3. Zeichen *„b“* und füge das Zeichen *„c“* an 3. Stelle ein (*delete, insert*).
- Entferne alle Zeichen von A_1 und füge die Zeichen *„a“ „b“ „b“* ein (*3x delete, 3x insert*).

Es sind sogar beliebig viele *edit-scripts* denkbar, die einen Informationsbestand in eine anderen überführen. Die Auswirkung ist allerdings immer gleich. Durch das Operationspaar *delete* und *insert* steigt die Anzahl der möglichen Änderungsfolgen sogar in das Unendliche. Von Interesse sind allerdings nur solche, die mit einer möglichst geringen Zahl von Operationen auskommen, der *edit-distance*. Da es aber mehrere *edit-script* gibt, die die gleiche Länge (also Anzahl von Operationen) aufweisen können, deren Reihenfolge (chronologische Abfolge) ebenfalls entscheidend ist, liegt es nahe die Operationen mit Kosten zu bewerten, um *edit-scripts* gleicher Länge bewerten zu können.

2.6.1.2. Delta-Trees

Eine weitere Möglichkeit Änderungsabfolgen darzustellen ist der Einsatz von *Delta-Trees*. Diese Darstellungsart wird von *Deltaxml core* eingesetzt. Der wesentliche Unterschied gegenüber dem *edit-script* ist, dass das zu vergleichende Dokument um die Änderungsinformationen erweitert wird.

2.6.2. Vergleich von Sequenzen - Linearer Vergleich

Der Vergleich von Sequenzen erfolgt auf Zeichenketten oder Zeilen in einem Textdokument. Hierbei ist die zugrunde liegende Datenstruktur eine Liste von geordneten Elementen (z.B. Zeichen in Strings oder Zeilen in einem Textdokument). Die Vorstellung eines Algorithmus wird in dieser Arbeit ausgegrenzt kann aber beispielsweise bei [Mye86] nachgelesen werden.

2.6.3. Vergleich von Bäumen - strukturorientierter Vergleich

Weisen Informationsbestände, wie XML-Dokumente, hierarchisch strukturierte Daten auf, so lassen sie sich mit Hilfe von Bäumen darstellen (siehe 2.2.2).

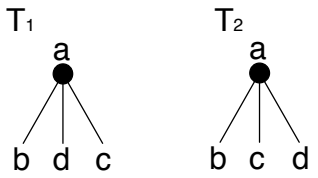


Abbildung 2.2.: ungeordneter Baum vs. geordneter Baum

Es ist nun weiterhin sinnvoll für die Betrachtung des Vergleichs von Bäumen zwischen geordneten und ungeordneten Bäumen zu unterscheiden. Spielt die Reihenfolge der Kinder von Knoten eine Rolle, spricht man von geordneten Bäumen, andernfalls von ungeordneten Bäumen. Die Abbildung 2.2 zeigt zwei Bäume, welche sich in der Reihenfolge der Kinder unterscheiden.

Betrachtet man diese Bäume als geordnet, so unterscheiden sich diese Bäume voneinander. Hingegen führt eine ungeordnete Betrachtungsweise zur Gleichheit der Bäume. Soll beispielsweise eine technische Dokumentation in Form eines XML-Dokumentes durch einen Baum repräsentiert werden spielt die Reihenfolge eine große Rolle (dokumentzentrierte XML-Dokumente).
Abhängig davon, ob man die zu vergleichenden Informationsbestände in geordneten oder ungeordneten Bäumen repräsentieren kann, fallen die Lösungen des Problems des Erstellens eines *edit-scripts* in unterschiedliche Problemklassen. Prinzipiell ist der Vergleich geordneter Bäume einfacher, als der Vergleich ungeordneter Bäume, da sich bei geordneten Bäumen Verfahren zur Auffindung längster gemeinsamer Untersequenzen (LCS) einsetzen lassen [CRGMW96]. Die folgenden Abschnitte beschreiben zwei Verfahren, die stellvertretend für die beiden Problemklassen gesehen werden können.

2.6.4. Vergleich von geordneten Bäumen

Dieser Abschnitt beschäftigt sich mit dem Vergleich geordneter Bäume. Aus einer Anzahl von Verfahren [ZS89, SZ90, CRGMW96] wird das in [CRGMW96] vorgestellte Verfahren *La-Diff* kurz vorgestellt. Diese Verfahren ist interessant, da neben den Operationen *insert*, *delete* und *update* noch die Operation *move* Verwendung findet.

2.6.4.1. Das La-Diff Verfahren

Bevor das eigentlich Verfahren vorgestellt wird, findet einleitend eine Beschreibung des Operations-Kataloges für das edit-script statt. Das eigentliche Verfahren, welches aus zwei

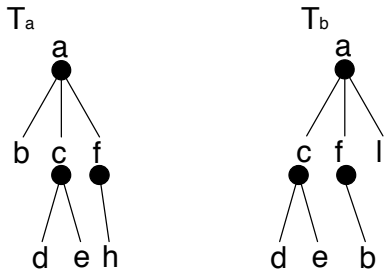


Abbildung 2.3.: Zu vergleichende Bäume T_a und T_b (Eigene Darstellung nach [CRGMW96])

Teilschritten besteht wird anschließend erläutert.

Der Operations-Katalog:

Das Verfahren untersucht zwei geordnete Bäume T_a und T_b mit dem Ziel, die Differenzen in Form eines edit-scripts zu ermitteln. Der dabei verwendete Operations-Katalog umfasst neben den Verfahren [ZS89, SZ90], welche die Operationen *insert*, *delete* und *update* verwenden, noch die Operation *move*, welche die Aussagekraft des edit-scripts erhöht.
Die Operationen wirken auf den Baum T_a und überführen diese schrittweise in den Baum T_b. Dabei bezeichnet n und p die Knoten des Baumes T_a, v eine Beschriftung eines Knotens und k die einzunehmende Position eines Knotens innerhalb der Kinder seines neuen Vaters:

***update*-Operation:** *upd(n, v)* Die *update*-Operation ändert die Beschriftung des Knoten n, so dass sie v entspricht.

***insert*-Operation:** *ins(v, p, k)* Die insert-Operation erzeugt einen neuen Knoten k, mit der Beschriftung v, des Kindes an der Position k des Knoten p.

***delete*-Operation:** *del(n)* Entfernt den Knoten n, Prämisse ist, dass n keine Kindknoten besitzt. Diese müssen unter Umständen vorher gelöscht oder verschoben worden sein.

***move*-Operation:** *mov(n, p, k)* Die *move*-Operation verschiebt den bei n beginnenden Teilbaum $t(n)$, so das p der Vater von n wird und an der Position k der Kinder erscheint.

Vorgehen des Verfahrens:

Das Verfahren besteht aus zwei Teilschritten die folgend genauer beschrieben werden.

Ermittlung der Zuordnung: Dieser Schritt beinhaltet die Zuordnung gleicher Elemente der Struktur. Es wird als „*the good matching problem*“ bezeichnet.

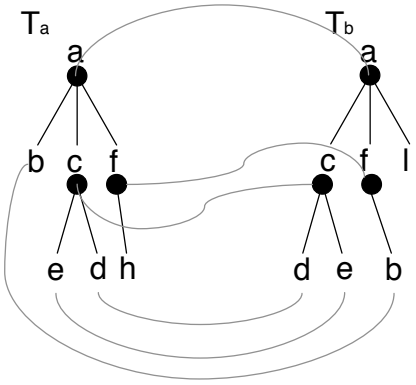


Abbildung 2.4.: Die gefundenen Zuordnungen der Bäume T_a und T_b (Eigene Darstellung nach [CRGMW96])

Erstellung eines edit-scripts aus der gefunden Zuordnung: In insgesamt fünf Phasen wird aus der ermittelten Zuordnung das edit-script erstellt.

Ermittlung der Zuordnung:

Das resultierende *edit-script* wird aus einer Folge von Operationen bestehen, die auf den Baum T_a ausgeführt einen zu Baum T_b äquivalenten Baum ergeben. Bei der Ermittlung der Zuordnung wird versucht für jeden Knoten $m_a \in T_a$ einen äquivalenten Knoten $m_b \in T_b$ zu finden. Sollte es eine Knoten $m_a \in T_a$ geben, dem kein Knoten $m_b \in T_b$ zugewiesen werden kann, so wird dies durch die *delete*-Operation im *edit-script* ausgedrückt. Sollte es genau umgekehrt sein, das es eine Knoten in $m_b \in T_b$, gibt der keine Partner in T_a besitzt, wird das durch eine *insert*-Operation im *edit-script* ausgedrückt.
Durch diese Zuordnung erhält jeder Knoten der Bäume einen Partner im jeweils anderen Baum, oder wird als zu entfernender oder hinzugefügter Knoten betrachtet. Die Abbildung 2.4 zeigt eine Zuordnung der Knoten.

Tragen die Knoten eine eindeutige Identifizierung, wird das Zuordnen merklich vereinfacht. Sind Knoten nicht eindeutig identifizierbar, wird die Zuordnung anhand der Knotenbeschriftung und Anzahl von denjenigen Zuordnungen ermittelt, welche die Kindknoten m_a mit den Kindknoten m_b verbinden.

Erstellen eines *edit-scripts*:

Das Erstellen eines *edit-scripts* läuft insgesamt in fünf Phasen ab, wobei in jeder Phase dem edit-script eine Gruppe von Operationen hinzugefügt werden:

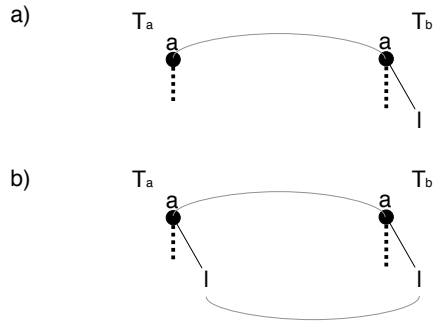


Abbildung 2.5.: a) Dem mit „l“ beschriftete Knoten in T_b ist kein Knoten in Ta zugeordnet. b) In T_a wurde ein neuer Knoten „l“ hinzugefügt und dem Knoten „l“ in T_b zugeordnet. (Eigene Darstellung nach [CRGMW96])

***update*-Phase:** In dieser Phase wird nach Paaren von $(m, n) \in K$ gesucht, bei denen $v(m) \neq v(n)$ gilt. Für alle diese Paaren wird dem *edit-script* die *update*-Operation hinzugefügt.

***align*-Phase:** Diese Phase beschäftigt sich mit den Kindern der Knoten m und n, die einander zugeordnet sind (wie zum Beispiel die beiden mit „c“ beschrifteten Knoten in Abbildung 2.3). In diesem Beispiel sind die „d“ und „e“ beschrifteten Knoten in den Bäumen T_a und T_b nicht in der gleichen Reihenfolge. Durch eine *move*-Operation einer der beiden Knoten in T_a lässt sich die Reihenfolge in T_b erreichen.

***insert*-Phase:** In dieser Phase wird nach Knoten $n \in T_b$ gesucht, die über keine Zuordnung verfügen, dessen Vater aber eine Zuordnung aufweist. Es wird dann ein neuer Knoten m in T_a mit der gleichen Beschriftung $v(n) = v(m)$ eingefügt und die Knoten n und m einander zugeordnet. Die Abbildung 2.5 gibt dafür ein Beispiel. Es wird der Konten *l* in T_a eingefügt und dem Knoten *l* in T_b zugeordnet.

***move*-Phase:** In dieser Phase wird nach einander zugeordneten Knoten (m, n) gesucht, deren Väter $p(m)$ und $p(N)$ nicht einander zugeordnet sind. Der Knoten b in Abbildung 2.3 ist das der Fall. Das Ziel einer solchen Verschiebung ist der Knoten $p(m)' \in T_a$ der dem Knoten $p(n)$ zugeordnet ist (siehe Abbildung 2.6).

***delete*-Phase:** Abschließend wird in der letzten Phase nach Knoten $m \in T_a$ gesucht, denen kein Knoten aus T_b zugeordnet ist und keine Kinder aufweisen. Diese Knoten werden aus dem Baum T_a entfernt und dem *edit-script* eine entsprechende *delete*-Operation hinzugefügt.

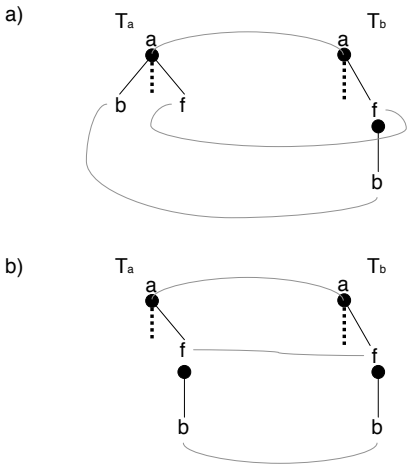


Abbildung 2.6.: a) Der mit „b“ beschriftet Knoten muss verschoben werden, da sein Vater (a) nicht dem Vater in T_b zugeordnet ist. b) Das Ziel der Verschiebung ist der Knoten „f“ da dieser dem Vater in T_b zugeordnet ist. (Eigene Darstellung nach [CRGMW96])

2.6.5. Vergleich von ungeordneten Bäumen

In [ZSS92] wird gezeigt und in [GC02] erwähnt, dass der Vergleich von ungeordneten, beschrifteten Bäumen NP-Vollständig ist. Aus diesem Grund wenden viele Verfahren für den Vergleich Heuristiken an. Da in dieser Arbeit der Vergleich von ungeordneten Bäumen eine untergeordnete Rolle spielt, wird nur kurz auf einen bekannten Vertreter dieser Kategorie eingegangen. Interessierten Lesern, empfehle ich den Algorithmus MH-Diff der im Artikel [CGM97] vorgestellt wird weiter zu verfolgen und die Projektseite „The C3 Project at Stanford“ [C397] zu besuchen.

2.6.5.1. Der Algorithmus MH-Diff

Der Algorithmus MH-Diff vergleicht zwei ungeordnete Baumstrukturen T_a und T_b mit dem Ziel, die Differenzen in Form eines *edit-scripts* darzustellen. Der dabei Verwendung findende Operations-Katalog umfasst gegenüber *Deltaxml core/sync* oder anderen Verfahren neben den Operationen *insert*, *delete* und *update* noch die Operationen *move*, *copy* und *glue*, welche die Aussagekraft des *edit-scripts* erhöhen.

Vorgehen des Verfahren:

Zur Erzeugung des *edit-scripts* sieht der Algorithmus drei Teilschritte vor:

Zuordnung: im ersten Schritt werden die einzelnen Knoten der beiden Bäume so miteinander in Beziehung gesetzt, dass jeder Knoten einen oder mehrere Partner im jeweils anderen Baum erhält. Eine solche Verbindung stellt die Gemeinsamkeiten der beiden Bäume zueinander dar.

Beschriftung der Verbindungen: Auf Grundlage dieser Zuordnung werden im zweiten Schritt die Verbindungen mit Beschriftungen versehen.

Erzeugung des edit-scripts: Im dritten Schritt wird mit Hilfe der Beschriftungen der Kanten das *edit-script* erzeugt.

Eine genauere Betrachtung diese Verfahrens würde an dieser Stelle zu weit führen.

2.7. 3-Wege-Merge-Algorithmus

Dieser Abschnitt beschreibt das Verfahren des 3-Wege-Merge von hierarchisch strukturierten Informationsbeständen. 3-Wege-Merge-Algorithmen auf unstrukturierten Informationsbeständen werden hier nicht betrachtet. Es wird hier das Verfahren von Lindholm vorgestellt, welches als 3DM Tool bekannt ist. Das Verfahren arbeitet auf geordneten, beschrifteten Bäumen.
Als *Base* wird das Original-Dokument (Baum) bezeichnet, die *Derivate1* und *2* sind Versionen gegenüber dem Original-Dokument. Das Ergebnis ist das zusammengeführte Dokument (merge), welches die Änderungen beider Derivate gegenüber dem Original beinhaltet.
Sollten sich Änderungen überschneiden, spricht man von einem Konflikt. Konflikte können in folgenden Fällen auftreten [Lin01, S55-56]:

1. *Update/Update-Konflikt*. Dieser Konflikt tritt auf, wenn in beiden Derivaten der gleiche Knoten geändert wurde.(Der Titel im Originaldokument (Base) „Aufgabenliste“ wird im Derivat1 auf „Aufgabenliste Stand 21.12.2007“ und im Derivat2 auf „TODO-Liste “ geändert.)

2. *Delete/Update, Delete/Move, Delete/Copy-Konflikt*. Dieser Konflikt kann immer dann auftreten, wenn ein Knoten in dem einen Derivat kopiert, verschoben oder geändert wurden und im anderen Derivat gelöscht wurde.

3. *Move/Move-Konflikt*. Dieser Konflikt tritt dann auf, wenn ein Knoten sowohl im einen als auch im anderen Derivat verschoben wurde. Wohin soll der Knoten nun verschoben werden?

4. *Append/Append-Konflikt* Existiert ein Knoten n, welche über Kinder $c(n)$ verfügt. In beiden Derivaten wird nun ein neuer Knoten an die Kinder von n angefügt (z.B. $T_B =$

$(R; a)$ $T_1 = (R; ab)$ and $T_2 = (R; ac)$). In manchen Fällen bspw. bei einer To-Do-Liste ist die Reihenfolge egal, in anderen Fällen wenn es sich um Abschnitte in einem Textdokument handelt, kann die Reihenfolge entscheidend sein und somit als Konflikt eingestuft werden.

2.7.1. Verfahren des 3-Wege-Merge

Im Folgenden wird nun das Verfahren des 3-Wege-Merges des 3DM Tools genauer betrachtet [Lin01]. Das Verfahren lässt sich grob in zwei Schritte unterteilen. Der erste Schritt ist ähnlich den oben erwähnten Vergleichs-Algorithmen und behandelt das Zuordnungsproblem von Bäumen, der zweite Schritt ist das eigentliche Zusammenführen von Bäumen.

Zuordnung: Lindholm bezeichnet diesen Teil des Verfahrens als *Tree-Matching*.

Zusammenführen der Bäume: Das Zusammenführen der Bäume läuft nach der vorher festgestellten Zuordnung der Bäume ab. Lindholm hat hier einen Algorithmus entwickelt, der ohne ein *edit-script* auskommt.

Im nächsten Abschnitt wird das Zuordnen oder *Tree-Matching* genauer betrachtet.

2.7.1.1. Zuordnung

Der Tree-Matching Algorithmus der bei Lindholm zum Einsatz kommt basiert auf einer einfachen Heuristik: „Finde so viele, wie mögliche gleiche Teilbäume zwischen T_a und T_b und ordne diese einander zu.“ [Lin01, S.59]
Diese Heuristik basiert auf der Heuristik des MH-Diff Verfahrens [CGM97]. Grund dafür ist die Vielzahl von Operationen, welche das MH-Diff Verfahren unterstützt (neben *insert*, *delete*, *update*, *move*, die Operationen *copy* und *glue*). Lindholm weist ausdrücklich darauf hin, dass es sich bei dem verwendeten Zuordnungsalgorithmus nur um einen Prototyp handelt. Den interessierten Lesern empfehle ich an dieser Stelle, das Kapitel 6 in der Masterthesis von Lindholm [Lin01, S.59-74].

2.7.1.2. Zusammenführung der Bäume

Der Merge-Algorithmus [Lin01, Kap.7, S. 75-97] arbeitet direkt auf den Zuordnungsbäumen M_1 und M_2, die im vorherigen Schritt „Zuordnung“ erstellt wurden. M_1 entspricht dem Zuordnungsbaum aus Base und Derivat1. M_2 entspricht dem Zuordnungsbaum aus Base und Derivat2.
Die Grundidee des Algorithmus basiert auf einer gleichzeitigen Traversierung der Bäume T_1

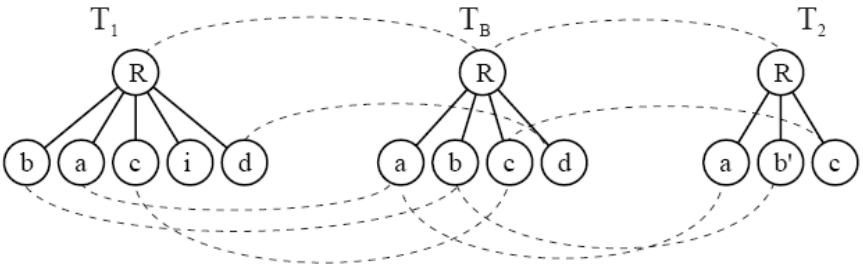


Abbildung 2.7.: Einfaches Merge Beispiel [Lin01, S.76]

und T_2, so dass Partnerknoten gleichzeitig betrachtet werden. Jeder Schritt in der Traversierung der Partnerknoten führt zu einem neuen Knoten im zusammengeführten Baum T_m. Der neue Knoten in T_m ist der aktuell betrachtete Knoten in T_1 und T_2. Zur Veranschaulichung wird das Tree-Cursor-Konzept eingeführt. Welches die aktuelle Positions in den beiden Bäumen wiedergibt. Es handelt sich hierbei genau genommen um ein Cursorpaar (C_1,C_2), wobei C_1 der Cursor des Baumes T_1 und C_2 der Cursor des Baumes T_2 ist. Für den zusammengeführten Baum T_m wird der Cursor C_m eingeführt. Der eigentlich Algorithmus gliedert sich in fünf Schritte:

Erstelle die Merge-Pair-List: In diesem Schritt werden Paare der Kinder der Knoten n(C_1) und n(C_2) entsprechend der Zuordnung erzeugt. Als nächstes wird die Sequenz dieser Paare unter Berücksichtigung von Verschiebungen in einem der beiden Bäume festgelegt. Gelöschte Knoten werden von dieser Liste entfernt. Im Beispiel aus der Abbildung 2.7 wurden aus den Listen *b a c i d* des Baumes T_1 und *a b' c* des Baumes T_2 Paare gebildet. Das Ergebnis ist wie folgt:

$$b\ a\ c\ i$$

$$b'\ a\ c\ \cdot$$

Die obere Reihe enthält die Knoten von T_1 die untere die Knoten von T_2. Der Reihenfolge resultiert aus der Verschiebung des Knoten *b* in T_1 gegenüber T_B. Der Knoten *d* welcher in T_2 entfernt wurde ist nicht mehr enthalten und das Einfügen von *i* als Ergebnis kein Paar hat.

Merge den Inhalt: Es kann jetzt der zusammengeführte Inhalt der Paare (u_i,v_j) bestimmt werden. Allgemein gesprochen wird immer der Inhalt der sich gegenüber der Base T_B verändert hat übernommen. In diesem Fall wäre der Inhalt: *b' a c i*.

Füge den Knoten *w* **in den Baum** T_M **ein:** Es wurden nun die Knoten u_i und v_j in den Knoten *w* zusammengeführt. Der zusammengeführte Knoten *w* wird in den Merge-Baum T_M eingefügt und die Cursor-Position wird so geändert, dass es möglich ist

neue Kinder an den neuen Knoten anzufügen. Nach dem Hinzufügen der Knoten *b' a c i* sieht der Merge-Baum wie folgt aus: T_M*(R; b' a c i)*.

Repositioniere die Cursor C_1 und C_2: Die Cursor-Positionen der Cursor C_1 und C_2 werden nun auf die Knoten im Merge-Paar gesetzt, deren zusammengeführter Inhalt in T_M eingefügt wurde. Im Normalfall zeigen die Cursor direkt auf u_i und v_j, allerdings gibt es einige Ausnahmen, ausgelöst dadurch wenn das Paar nur einen Knoten hat. Die Cursor-Änderungen für das Beispiel sehen wie folgt aus:

$$b, b' \Rightarrow C_1 = T_1(b) \wedge T_2(b')$$

$$a, a \Rightarrow C_1 = T_1(a) \wedge T_2(a)$$

$$c, c \Rightarrow C_1 = T_1(c) \wedge T_2(c)$$

$$i, \cdot \Rightarrow C_1 = T_1(i) \wedge T_2(0)$$

Rufe den Merge-Algorithmus rekursiv auf: Dies ist der rekursive Aufruf des Algorithmus. Es wurde jetzt erfolgreich ein neuer Merge-Knoten der Kinder der Knoten, auf welche die Cursor C_1 und C_2 gezeigt hatten in dem Merge-Baum T_M eingefügt. Die Cursor C_1 und C_2 wurden nun auf die neue Menge von Partnern zurückgesetzt, und der Cursor C_M steht auf einer neuen Einfügeposition im Merge-Baum T_M.

Die Abbildung zeigt für ein einfaches Merge-Beispiel den kompletten Ablauf des von Lindholm entwickelten 3-Wege-Merge.

2.8. Fazit

Dieses Kapitel eröffnete die Grundlagen mit Bäumen und XML. Es wurde weiter auf die Grundlagen der Vergleichs- und 3-Wege-Merge-Algorithmen eingegangen, mit dem Ziel den Leser in die komplexe Thematik einzuführen und wichtige Begrifflichkeiten wie das edit-script, edit-distance oder den Operations-Katalog vorzustellen. Dem Leser wurden einzelne Verfahren (La-Diff, Merge-Algorithmus des 3DM Tools) näher vorgestellt und dieser kann jetzt nachvollziehen, dass je nach Problemklasse[4] die Lösungen eher einfach oder komplex ausfallen.

[4] Die Problemklassen sind zum einen von der zugrundeliegenden Struktur, als auch von den verwendeten Operationen abhängig.

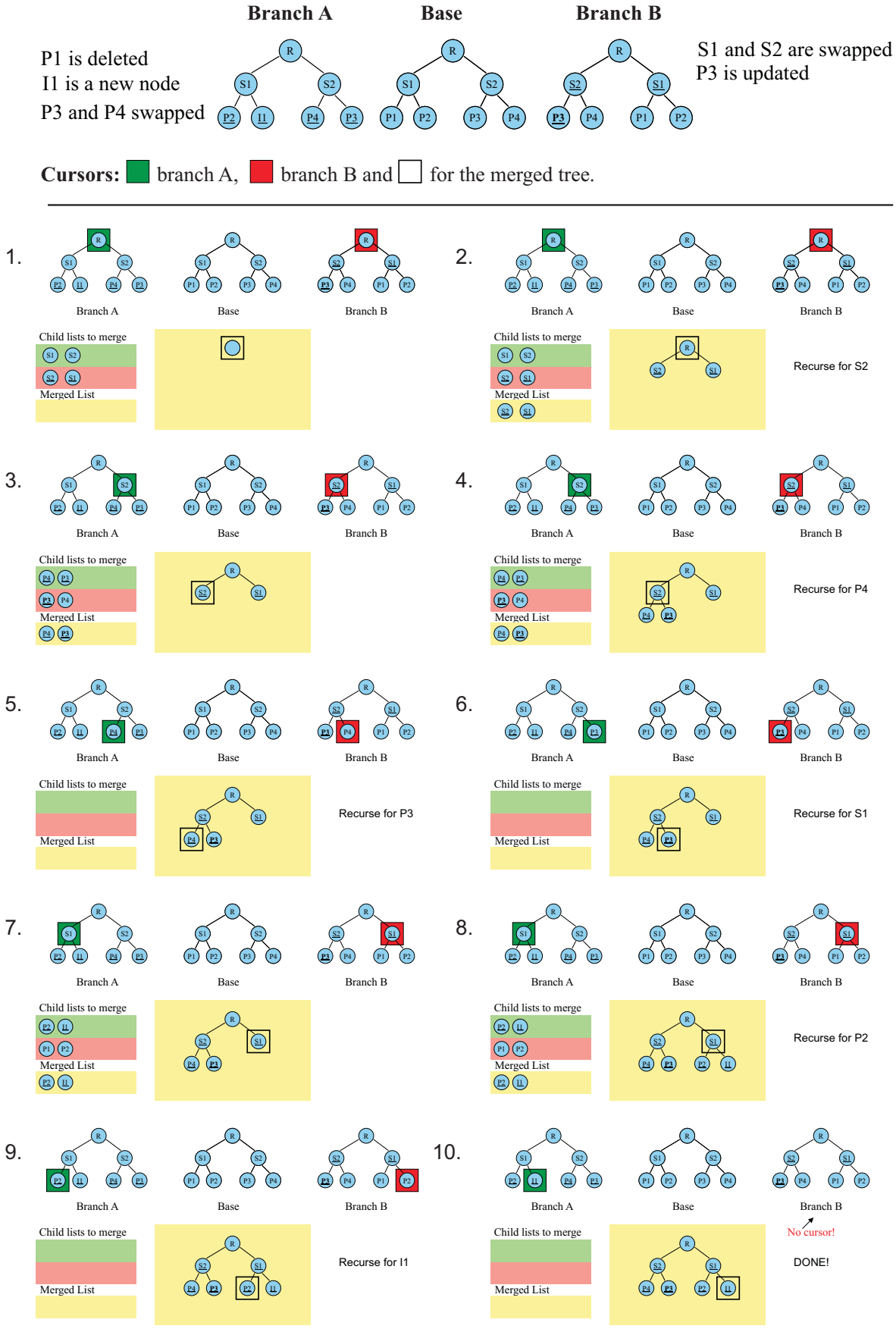


Abbildung 2.8.: Illustration des Merge-Algorithmus [Lin01, S.78]

3. Analyse

Die Analyse gliedert sich in eine fachliche und technische Analyse.
Fachlich wird der Erstellungs- und Revisionskonrtollprozess, die eingesetzten Dokumentarten und die Spezifikation ATA iSpec 2200 analysiert. Die Vision den 3-Wege-Merge für die Revisionskontrolle einzusetzen wird hier erarbeitet.
Die technische Analyse dient zum einen der Feststellung von Besonderheiten des XML-Formats. Zum anderen werden die beiden Programme *DeltaXML core/sync* [Del08] und *3DM Tool* [Lin06] auf ihre Vergleichs- und 3-Wege-Merge-Funktionalität bezüglich XML-Dokumenten untersucht. Eine Analyse weiterer Tools sehe ich für diese Arbeit nicht als notwendig, da diese in der Studienarbeit von Hottinger und Meyer [HM05] und in der Masterthesis von T. Lindholm [Lin01] ausreichend betrachtet wurden.
Das Ergebnis der beiden Analysen dient der Festlegung und Priorisierung für den konzeptionellen Entwurf und die anschließende prototypische Implementierung.

3.1. Erstellungsprozess und Revisionskontrolle bei Arbeitskarten

Um die fachlichen Anforderungen für die Unterstützung der Revisionskontrolle festzuhalten, ist es erforderlich den Prozess der Arbeitskartenerstellung und -revisionskontrolle zu beschreiben und zu analysieren. Ziel dieses Abschnittes ist die Findung der fachlichen Anforderungen des Prototyps.

3.1.1. Systemsicht auf den Prozess

Die Abbildung 3.1 zeigt den Workflow beim Arbeitskartenerstellungsprozess. Es werden im Folgenden nur die für diese Arbeit relevanten Aspekte betrachtet.
Die Herstellerdokumente werden als SGML-Dokumente angeliefert und im System DocSurf gespeichert.

Das System DocSurf dient dem Dokumentretrieval (Browser) und dem Export der Herstellerdokumente in ein XML-Schnittstellen-Format, welches dann in das SAP DVS (Dokumenten-Verwaltungs-System) importiert werden kann. Beim XML-Export wird das AMM auf Task-Ebene exportiert.
Jeder einzelne Task eines AMMs wird als Dokumentinfosatz und inhaltliche Taskbeschreibung im DVS gespeichert. Für jeden Dokumentinfosatz existieren Metadaten, wie bspw. Tasknummer, Datum, Revisionsnummer, Revisionsdatum. Diese Metadaten werden im MRO System (Maintenance, Repair and Overhaul) noch mit zusätzlichen Informationen bezüglich Arbeitseinteilung und Arbeitsplanung angereichert. Beide Systeme (DVS und MRO) sind SAP-Anwendungen.
Interessant ist für den 3-Wege-Merge allerdings nur die inhaltliche Beschreibung des Arbeitsschrittes, auf dessen Basis der Autor die Arbeitskarte erstellt. Erstellt werden die Arbeitskarten im DocCreate Client. Der DocCreate Client ist ein angepasster Arbortext Editor (XML-Editor), der um spezielle Funktionalitäten bezüglich des Ein- und Auscheckens aus dem DVS System erweitert ist. Ebenso wird das Layout der Arbeitskarte im Arbortext Editor durch FOSI-Stylesheets für den Autor anschaulich aufbereitet.

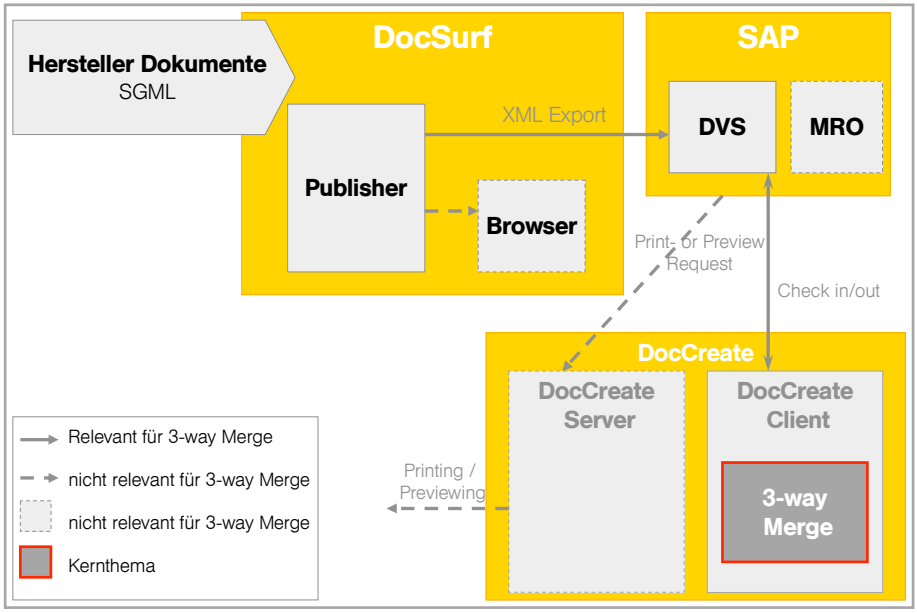


Abbildung 3.1.: Systemsicht des Arbeitskartenerstellungsprozesses

3.1.2. Rollen und Aufgaben

Es konnten folgende Rollen und deren Aufgaben identifiziert werden.

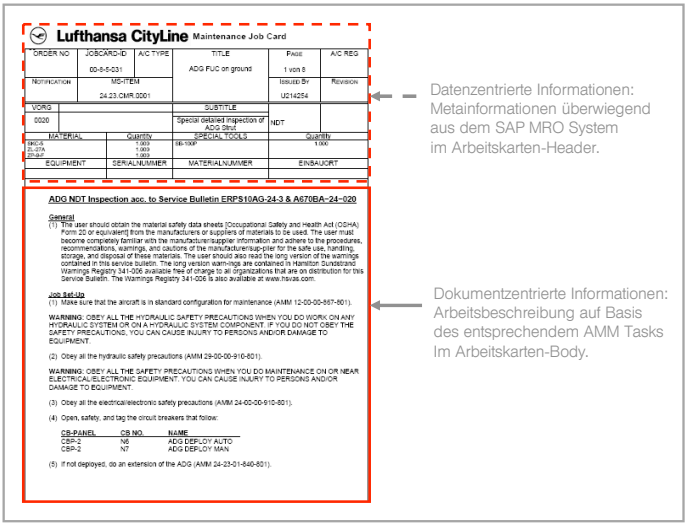


Abbildung 3.2.: Aufbau einer Arbeitskarte in Kopf- und Rumpfbereich ohne Deckblatt.

Hersteller: Die Hersteller sind für die Erstellung und Auslieferung der Handbücher verantwortlich. Sie beliefern die Betreiber mit den Handbüchern.

Autor: Die Rolle der Autoren kann den Wartungsingenieuren und Flugzeugbau- und Triebwerksingenieuren zugeschrieben werden. Zu ihren Aufgaben zählen die Erstellung und Revisionskontrolle der Arbeitskarten.

Arbeitsplaner: Arbeitsplaner beschäftigen sich mit der Koordination und Einteilung der Arbeitsabläufe. Diese nutzen das SAP MRO System.

Leser: Die Rolle der Leser nehmen die Mechaniker ein. Ihnen muss die Einsicht in die technische Wartungsdokumentation möglich sein.

Da die Autoren für die Erstellung und Revisionskontrolle der Arbeitskarten verantwortlich sind, kommt ihnen bei der Analyse ein besonderer Stellenwert zu, was eine genauere Betrachtung ihrer Aufgaben erfordert.

3.1.3. Die Arbeitskarte und die Rolle der Autoren

Die Arbeitskarte (engl. job card) dient der Beschreibung einer Wartungsarbeit. Im Folgenden wird der Aufbau einer Arbeitskarte beschrieben und näher auf die Rolle der Autoren eingegangen.

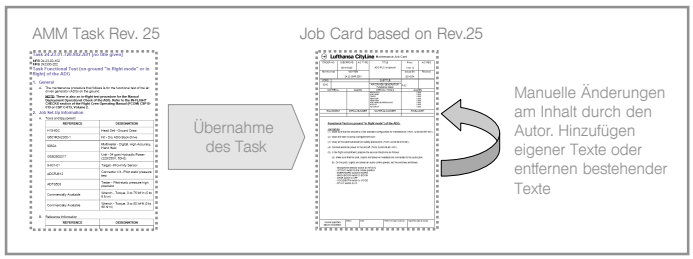


Abbildung 3.3.: Initialerstellung der Arbeitskarte und inhaltliche Übernahme aus dem AMM-Task

3.1.3.1. Aufbau der Arbeitskarte

In der Prozessbeschreibung wurde erkannt, dass die benötigten Daten einer Arbeitskarte aus unterschiedlichen Quellen bezogen werden:

1. Die eher datenzentrierten Informationen (siehe Grundlagen) hauptsächlich aus dem SAP MRO System (z.B. Mannstunden, Auftragsnummer und benötigte Materialien)
2. die dokumentzentrierten Informationen (siehe Grundlagen) aus der AMM-Task-Beschreibung (Durchführung bestimmter Wartungsvorgänge am Flugzeug)

Die datenzentrierten Informationen finden sich im Deckblatt, Kopf- und Fußzeile wieder, während die dokumenzentrierten Daten im Content/Body-Bereich erscheinen. Diese Aufteilung des Layouts in Kopf- und Fußzeile und in eine Content/Body-Bereich hat sich sehr bewährt und ist bei fast allen Dokumenten Standard. Abbildung 3.2 zeigt eine vereinfachte Darstellung einer Arbeitskarte ohne Deckblatt.

3.1.3.2. Initialerstellung einer Arbeitskarte

Der Autor einer Arbeitskarte ist in erster Linie für den inhaltlichen Bereich der Arbeitskarte verantwortlich. Er übernimmt den Inhalt eines AMM-Task und optimiert diesen für die internen Wartungsarbeiten am Flugzeug. Dabei löscht er bestimmte Textstellen oder fügt eigenen Inhalt hinzu. Die Abbildung 3.3 stellt die Übernahme des Inhaltes aus dem entsprechenden AMM Task dar. Im Arbortext Editor werden nun unnötige Textpassagen entfernt und zusätzliche wichtige Informationen hinzugefügt.

Beispiel für die Erstellung einer Arbeitskarte:
Ein Bespiel hierfür ist der AMM-Task 24-23-01-720-802-A01 aus dem AMM des CRJ700 AMM Rev. 25. Hierbei handelt es sich um eine Arbeitsbeschreibung für die Wartung des

„Air Driven Generator“ (ADG). Der ADG ist ein Stromgenerator, welcher bei einem Stromausfall unter der Flugzeugnase ausgeklappt werden kann und die wichtigsten Flugzeugkomponenten mit Strom versorgt. Ähnlich eines Windkraftwerkes wird der Generator durch den Flugwind angetrieben. Der Titel des AMMs lautet: “Task Functional Test (on ground „in flight mode“ or in flight) of the ADG“. Dieser Wartungscheck kann entweder im Hangar oder während des Fluges ausgeführt werden. Da dieser Wartungscheck bei der Lufthansa Cityline nur am Boden ausgeführt wird, löscht der Autor unnötige Informationen, die den in „flight mode check“ betreffen. So wird bspw. der Titel auf „ADG FUC on ground“ verkürzt. Darüber hinaus beinhaltet fast jeder Task allgemeine Vorbedingungen in einem Abschnitt „General“ und spezielle Vorbedingungen in einem Abschnitt „Job Setup“ mit einer Tabelle „Tools and Equipment“. Bezüglich „General“ sind die Informationen für die Arbeitsbeschreibung z. T. unnötig und werden entfernt. Die Tabelle „Tools and Equipment“ wird ebenfalls entfernt, da diese im MRO-System aufbereitet wird und im Kopfbereich der Arbeitskarte erscheint. Eine Redundanz in der inhaltlichen Beschreibung des Wartungsvorgangs ist nicht gewollt.

3.1.3.3. Revisionskontrolle einer Arbeitskarte

In zweiter Linie ist der Autor für die Revisionskontrolle der Arbeitskarten verantwortlich. Da das AMM einer ständigen Revision durch den Hersteller unterliegt, erscheinen in Zyklen von drei bis sechs Monaten neue Revisionen des AMMs. Diese Änderungen sind verbindlich und müssen in die Arbeitskarten mit eingepflegt werden. Hierbei vergleicht der Autor Änderungen des Herstellers mit daraus abgeleiteten Arbeitskarten und pflegt diese im Fall einer Änderung mit ein.

Abbildung 3.4 zeigt die prüfende Wiederdurchsicht durch den Autor. Der Autor muss sowohl seine manuell eingepflegten Änderungen als auch die Änderungen des neuen AMM-Tasks gegenüber dem alten AMM-Task in die neue Arbeitskarte übernehmen.

Man kann sich vorstellen, dass dieser Prozess sehr zeitintensiv und aufwendig ist. Erleichtern könnte man dem Autor die Revisionskontrolle dadurch, dass man eine neue zusammengeführte Arbeitskarte erstellt, die sowohl die Herstelleränderungen als auch die Änderungen des Autors berücksichtigt. Durch die Änderungsdarstellung innerhalb dieser zusammengeführten Arbeitskarte könnte der Autor auf einen Blick die Änderungen erkennen und diese akzeptieren oder verwerfen.

Diese Vision wird im nächsten Abschnitt ausführlich untersucht.

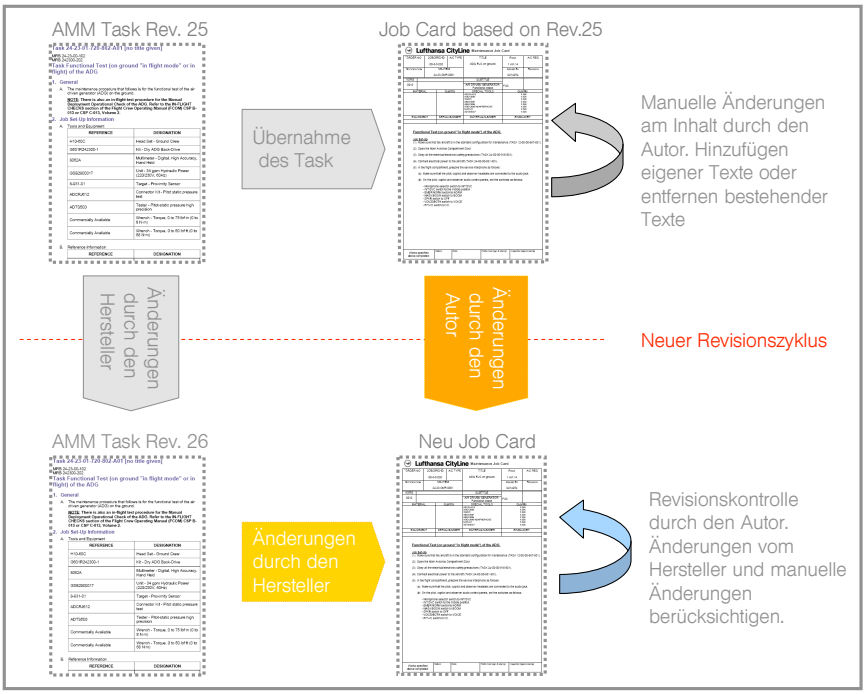


Abbildung 3.4.: Revisionskontrolle der Arbeitskarte

3.1.4. Vision: Unterstützung der Revisionskontrolle durch den 3-Wege-Merge

Die Änderungserkennung /-darstellung sowie das Zusammenführen von XML-Dokumenten wird heutzutage durch Tools unterstützt. Die folgende Abbildung 3.5 zeigt die Idee, die dahinter steckt. Durch einen 3-Wege-Merge soll sowohl die manuellen Änderungen des Autoren, als auch die Herstelleränderungen in einem neuen Arbeitskartenvorschlag zusammengeführt werden. Es handelt sich um keine endgültige Version der Arbeitskarte, sondern nur um einen Vorschlag aus folgenden Gründen:

1. Korrektheitsprüfung in diesem sicherheitskritischen Bereich ist durch eine menschliche Durchsicht erforderlich.
2. Konflikte können bei Überschneidungen von Änderungen auftreten. Diese können nur vom Menschen aufgelöst werden.
3. Für den Publikationsprozess der Arbeitskarte muss ein eigenes Revisionsmarkup gesetzt werden.
4. Noch keine praktische Erfahrung hinsichtlich des Einsatzes des 3-Wege-Merge bei Wartungsdokumenten der Lufthansa AG.

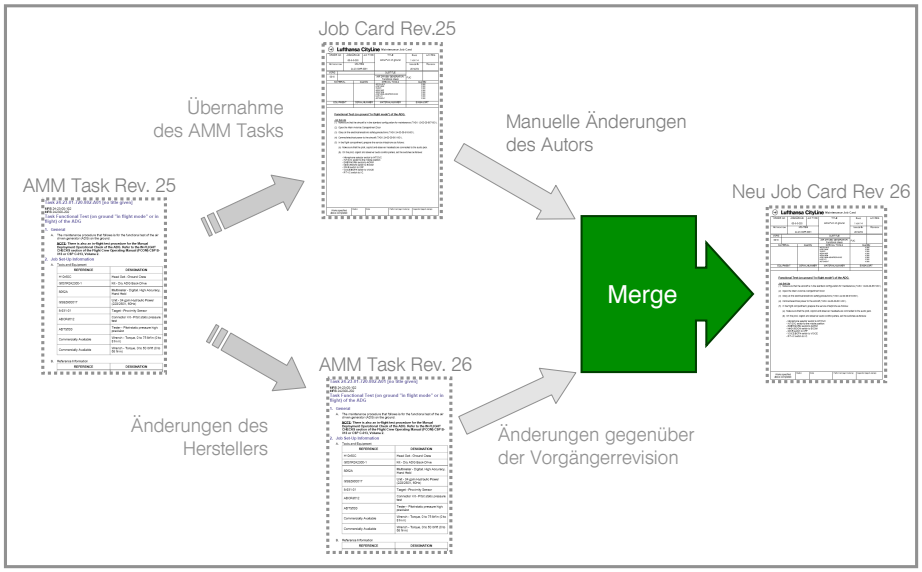


Abbildung 3.5.: Einsatz des 3-Wege-Merge.

Der Vorschlag dient allerdings den Autoren für eine einfachere Revisionskontrolle.

3.1.4.1. Einsatz des 3-Wege-Merge in Revisionszyklen

In den vorherigen Abschnitten wurde die Initialerstellung und die Revisionskontrolle durch den Autoren beschrieben. Aus der sich die Version der Verwendung des 3-Wege-Merges zur automatischen Erstellung der neuen Arbeitskarte abgeleitet hat. Um den Autoren durch den 3-Wege-Merge bei der Revisionskontrolle zu unterstützen, ist eine Betrachtung des Einsatzes des 3-Wege-Merge bei mehrerer Revisionszyklen erforderlich.

Die Abbildung 3.6 beschreibt die chronologische Abfolge der Revisionskontrolle und automatischer Erstellung der aktuellen (basiert auf der aktuellen AMM-Task-Revision) Arbeitskarte durch den 3-Wege-Merge. Erscheint durch den Hersteller eine neue Revision des AMM-Task, so ist es möglich, die aus dem 3-Wege-Merge gewonnene Arbeitskarte für die Erstellung einer neuen Arbeitskarte heranzuziehen. Eine vorherige Bereinigung der Revisionsmarkups ist allerdings erforderlich.

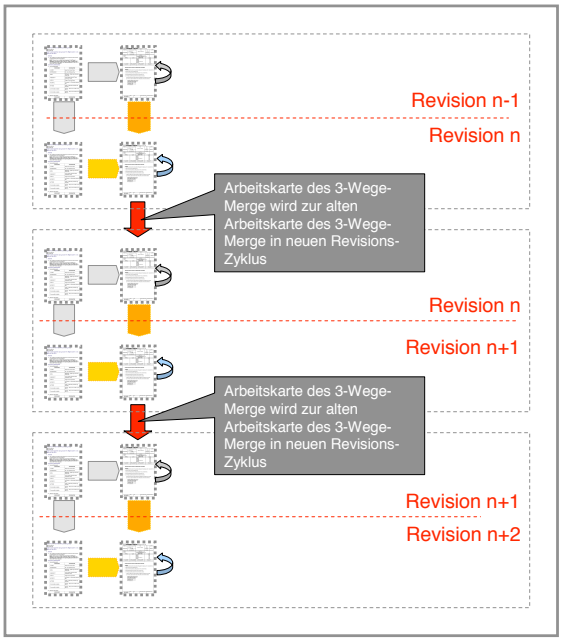


Abbildung 3.6.: Chronologische Revisionszyklen

3.1.5. Fazit

Der Einsatz für des 3-Wege-Merges zur Unterstützung des Autoren bei der Revisionskontrolle ist prinzipiell möglich. Es ergeben sich folgende fachlichen Anforderungen an das System:

1. Der Autor soll durch den 3-Wege-Merge für die aktuelle Revision eine Arbeitskarte erstellen können.
2. Der Autor muss Konflikte manuell lösen. Eine Automatisierung der Konfliktlösung ist nicht möglich.
3. Der Autor muss Konflikte erkennen. Diese sollen von anderen Elementen unterscheidbar sein.
4. Der Autor muss Änderungen zwischen den einzelnen Dokumentinstanzen nachvollziehen können. Dabei soll er möglichst flexibel in der Auswahl des Vergleiches sein (z.b. Vergleich alte/neue Arbeitskarte, alte/neue AMM-Task-Revision).
5. Änderungen müssen aussagekräftig dargestellt werden.
6. Abgrenzung: Das Revisionsmarkup für den Publikationsprozess aufzubereiten ist nicht Teil dieser Arbeit.

3.2. Dokumentarten

Wie einleitend erwähnt, sind für die Erstellung der Arbeitskarte die Herstellerdokumente die Grundlage. Das Wichtigste unter den Handbüchern ist das AMM, auf dessen Basis die Arbeitskarten erstellt werden. Daneben existiert noch eine Vielzahl anderer Dokumente, die für die Wartung am Flugzeug herangezogen werden. Aus den folgenden Dokumentarten lassen sich auf Task-Ebene die Arbeitskarten direkt ableiten und werden deshalb aufgeführt.

3.2.1. Aircraft Maintenance Manual - AMM

Das Aircraft Maintenance Manual ist das Wartungshandbuch eines Flugzeuges und dient als Grundlage für sämtliche Wartungsarbeiten und daraus resultierenden Arbeitskarten. Die entsprechenden Tasks (Arbeitsanweisungen) sind nach dem Unterstützungssystem Aircraft Maintenance Task Oriented Support System (AMTOSS) kategorisiert. Das AMTOSS nummeriert eindeutig und auf Basis von Chapter-Section-Subject das AMM. Die Sections beinhalten bestimmte Arbeitskategorien. So werden z.B. alle Tasks, die Arbeiten an der Tür, am Fahrwerk oder an der Klimaanlage betreffen, in einzelne Sections zusammengefasst. Das AMM weist im Durchschnitt pro Flugzeug 5000 bis 10000 Tasks auf. Diese Zahl variiert abhängig von der Größe, der Konfiguration und der Hersteller des Flugzeuges.

3.2.2. Temporary Revision - TR

Temporary Revisions beziehen sich auf Änderungen am AMM, die zwischen den Revisionszyklen erscheinen und deren Dringlichkeit es erfordert, die Änderungen den Betreibern noch vor Auslieferung der neuen Revision mitzuteilen. Die Temporary Revision beinhaltet meistens nur einen einzelnen Task des AMMs. TRs sind nur für einen Revisionszyklus gültig und müssen vom Hersteller bei der nächsten Revision in das AMM eingepflegt werden. Diese erfordern aber eine unverzügliche Anpassung der assoziierten Arbeitskarte.

3.2.3. Service Bulletin - SB

Das Service Bulletin ist eine technische Mitteilung über besondere Modifikationen am Flugzeug. In dieser Mitteilung wird bspw. darüber informiert, dass ein bestimmtes Bauteil am Flugzeug ausgetauscht werden muss. Klare Folge daraus ist die Erstellung einer Arbeitskarte, welche diesen Wartungsvorgang beschreibt. Meist erfordern die SBs nur eine einmalige Wartungsanweisung. Es kann aber auch vorkommen, dass diese SBs in den Wartungsturnus aufgenommen werden.

3.2.4. Fazit

Dadurch, dass die Arbeitskarte an sich aus den oben aufgeführten Dokumentarten resultiert, konnte bei der Analyse keine erwähnenswerten Besonderheiten festgestellt werden.
Alle Dokumentarten haben gemeinsam, dass sie der ATA iSpec 2200 entsprechen. An dieser Stelle ist es nun sinnvoll einen Blick in die Spezifikation zu werfen, um die einzelnen Modelle auf deren Relevanz für den 3-Wege-Merge und die Änderungserkennung zu analysieren. [1]

3.3. ATA iSpec 2200

Es sind eine Vielzahl von Flugzeugherstellern, Flugzeugzulieferern und Fluggesellschaften am Markt vertreten. Unter der Schirmherrschaft der ATA wurde die Spezifikation iSpec 2200 verabschiedet, welche die Wartungsdokumente und den Auslieferungsprozess der Wartungsdokumente beschreibt und spezifiziert. SGML und XML eignen sich besonders für diese Spezifikation, da der Austausch der Dokumente zwischen Anbieter und Betreiber sowie die Anpassung an ein individuelles Layout erleichtert werden.
Die iSpec 2200 standardisiert Wartungsdokumente wie das AMM, TRs oder SBs und deren Austausch. Die Spezifikation iSpec 2200 weist verschiedene Modelle auf, deren Relevanz für die Anforderungen essenziell sind. Darüber hinaus basieren alle DTDs der Herstellerdokumente auf diesem Standard, sämtliche Modelle die im Folgenden beschrieben werden, sind für die gefundenen Dokumentarten (siehe Kapitel 3.2) zulässig. Nachfolgend werden diese Modelle vorgestellt und zusammenfassend deren Relevanz bewertet. Grundlage für die Analyse der Modelle ist die ATA iSpec2200 [ATA06], welche der Lufthansa Systems in digitaler Form zur Verfügung steht. Die Dokumentation der Spezifizierung ist lizenziert und muss käuflich erworben werden.
Für die Analyse der ATA iSpec ist besonders das Kapitel 4 „Models and Schemas“ von Bedeutung. Dort finden sich die DTD-Beschreibungen als auch die Modellbeschreibung auf Element und Attribut hinsichtlich Einsatzzweckes und technische Umsetzung wieder.

3.3.1. ATA Dokument-Anker

Ein Dokument-Anker ist eine Einheit, über welche sich der Revisionsprozess steuern lässt, das Auffinden erleichtert oder die inhaltlich technische Diskussion zwischen zwei Personen erleichtern soll. Ein Dokument-Anker ist eine Struktur oder Teil einer Struktur um

[1] Die Dokumente: Arbeitskarte, AMM-Task, TR und SB werden im weiteren Verlauf der Arbeit als *fachliche Dokumente* betitelt

den Erstellungs- und Revisionsprozess zu steuern. Beispiele für Anker oder Strukturen sind chapter-section-topic, chapter-section-subject-task-subtask oder year-volume-article-heading. [ATA06, Kap.4-2-1 3.4]
Für das AMM wird die Struktur chapter-section-subject-task-subtask verwendet. Die Arbeitskarten werden auf task-subtask Ebene daraus abgeleitet.
Anker-Elemente dienen darüber hinaus der Redundanzvermeidung.
Elemente eindeutig zu identifizieren ist nicht trivial, besonders wenn Elementknoten eine Vielzahl von Attributen und Unterelementen aufweisen (siehe Grundlagen 2.7.1.1). Die einfachste Lösung für dieses Problem ist, die Identität über eine eindeutige ID festzustellen. Bei der Revisionskontrolle der Arbeitskarten ist deutlich zu erkennen, dass Dokumente aus unterschiedlichen Revisionen miteinander verglichen oder zusammengeführt werden. Die Konsequenz daraus ist, dass die Eindeutigkeit über die Dokumentinstanzen hinaus gewährleistet sein muss. Ich bezeichne diese Eindeutigkeit hier als revisionsfest. Eine Eindeutigkeit innerhalb der Dokumentinstanz reicht hier nicht aus. Für die in Tabelle 3.1 aufgeführten Elemente existiert ein Identifier in Form des Attributes „key“, der als revisionsfest gilt. Diese können für ein besseres Vergleichsergebnis herangezogen werden.

Elementname	**Attributname**
task	key
subtask	key
graphic	key
sheet	key
prcitem1,...,prcitem7	id
table	id
ftnote	ftnoteid

Tabelle 3.1.: Attribute vom Typ „ID“ in AMM-Tasks

Die Elemente „prcitem1,...,prcitem7“, „table“, „ftnote“ weisen innerhalb der Dokumentinstanz eine eindeutige ID auf, welche aber für ein besseres Vergleichs- und Zusammenführungsergebnis ebenfalls benutzt werden können. In der Realisierung wird die Verwendung von eindeutigen IDs bei bei DeltaXML core/sync vorgestellt, um ein effektiveres Ergebnis 3.5.4 zu erreichen.

3.3.2. Interner und externer Referenzmechanismus

Eine weitere Anforderung ist, ähnlich wie bei HTML-Dokumenten, Elemente untereinander zu verknüpfen. Der Leser kann über einen Hyperlink direkt in einen bestimmten Abschnitt des Dokumentes wechseln. Die iSpec 2200 sieht hierbei einen internen als auch ein externen Referenzmechanismus [ATA06, Kap.4-2-3 11] vor.

3.3.2.1. Internes Referenzmodell

Als interne Referenz wird eine Referenz bezeichnet, die sich auf ein Ziel innerhalb der gleichen Dokumentinstanz bezieht.

```
1 <!ELEMENT refint  - -  (#PCDATA)    >
2 <!ATTLIST refint
3           refid   IDREF   #IMPLIED  >
```

Listing 3.1: Auszug aus der ATA-SGML-DTD für das AMM

Über das Attribut „refid“ wird auf das entsprechende Ziel innerhalb des Dokumentes referenziert. Folgende Regeln sind für den internen Referenzmechanismus erforderlich:

- „refid“ ist immer vom Typ „IDREF“ und „IMPLIED“.
- „key“ ist vom Type „ID“ und „REQUIRED“ für den Anker.
- Alle Attribute vom Type „ID“ müssen als Attribut „key „bezeichnet werden außer „cals“ innerhalb von Tabellen (table und ftnote)
- Für alle Nicht-Anker-Elemente muss das Attribut „key“ eingefügt werden, wenn diese als Ziel einer Referenz dienen sollen.
- refint muss für alle internen Referenzen benutzt werden.

```
1  <doc>
2    <chapter key="cha01">
3      <title>Einleitung</title>
4      <para>Kapitel 1 beinhaltet die Einleitung.
5      In <refint refid = "cha02">Kapitel 2</refint> wird auf die Grundlagen eingegangen.
6       </para>
7    </chapter>
8    <chapter key="cha02">
9      <title>Grundlagen</title>
10     <para>Hier sollen Grundlagen geklärt werden </para>
11   </chapter>
12 </doc>
```

Listing 3.2: Interner Referenzmechanismus an einem einfachen Beispiel

Im Listing ist in Zeile 5 der Einsatz der internen Referenzierung verdeutlicht. Der Anker zeigt auf das Kapitel 2 in Zeile 8. Dort ist dem Chapter Element das Attribut „key“ mit dem Wert „cha02“ als Ziel eingetragen.

3.3.2.2. Externes Referenzmodell

Man spricht von externen Referenzen, wenn sie sich auf Ziele außerhalb der Dokumentinstanz beziehen. Beispiele hierfür sind Referenzen auf andere Tasks, Grafiken oder andere Dokumente.

3.3.2.3. Auswirkungen auf den Referenzmechanismus durch Änderungen und 3-Wege-Merge

Der 3-Wege-Merge wirkt sich insofern auf interne Referenzen aus, dass beim Zusammenführen Elemente mit eindeutigen Keys doppelt auftreten können. Ein interner Anker würde in diesem Fall zwei oder mehrere Ziele im Dokument finden. Die folgende Aufzählung fast die ID/IDREFS-Konflikte zusammen:

1. Das referenzierte Ziel wird gelöscht. Dies kann beim 3-Wege-Merge auftreten, wenn ein bestimmtes Element oder eine Textpassage vom Autor entfernt wird. Klare Folge daraus: Der Anker zeigt auf ein nicht mehr vorhandenes Ziel.
2. Durch das Zusammenführen von Dokumenten kann eine Verletzung der Eindeutigkeit der Keys entstehen. Dann zeigt der Anker auf zwei oder mehrere Ziele.

Manger befasst sich in seiner Diplomarbeit [Man00] mit der Erkennung und Auflösung von ID/IDREF-Konflikten. Interessierte Leser verweise ich auf dessen Arbeit.
Das Entstehen von ID/IDREF-Konflikten wird leider von keinem der beiden Tools erkannt oder unterstützt, sollte aber bei der Gültigkeitsprüfung eines Parsers gegenüber der DTD erkannt werden. Dieser Punkt muss vom Autor manuell überprüft werden.

3.3.3. ATA Revisionsmodell

Das Revisionsmodell [ATA06, Kap.4-2-3 1] dient der Kennzeichnung und Nachverfolgung von revidierten Dokumentpassagen. Dadurch lässt sich die Anforderung erfüllen, geänderte Inhalte gegenüber der Vorgängerrevision hervorzuheben. Das Modell sieht zum einen vor, Textinhalte im Textfluss zu kennzeichnen, zum anderen gesamte Strukturelemente als revidiert zu beschreiben.
Strukturelemente werden mit dem Konzept des „Document Anchor“ in der iSpec 2200 realisiert.

3.3.3.1. Revisionsmarken im Textfluss

Die verwendeten Elemente sind: „revst“ und „revend“. Diese kennzeichnen die revidierten Textpassagen. Diese Elemente treten alternierend direkt vor und nach dem revidierten Inhalt auf. Der Einsatz über Elementgrenzen hinweg ist nur auf unterster Anker-Ebene erlaubt, kann aber durch die DTD nicht abgesichert werden.
Folgendes Beispiel verdeutlicht den Einsatz der Revisionselemente „revst“ und „revend“:

```
1 <PARA><REVST>revised data<REVEND></PARA>
```

Listing 3.3: Revisionsmarken im Textfluss

3.3.3.2. Revisionsattribute bei Strukturelementen (Anker-Elemente)

Um Strukturelemente wie z. B. Chapter, Section, Subsection, Task als revidiert zu kennzeichnen, werden die Revisionsattribute „chg“ und „revdate“ verwendet. Diese sind wie folgt beschrieben:

- **chg** (Change Code) Attribute mit festen Werten „n“, „d“, „r“, „u“ (New, Deleted, Revised, Unchanged)
- **key** (Key Anchor Identifier) Eindeutig ID
- **revdate** (Revision Date) Datumsangabe der Änderung

Der Einsatz der Revisionsattribute hängt eng mit den Elementen zusammen. Das folgende einfache Beispiel soll den Einsatz des Revisionsmodells verdeutlichen und stammt aus der ATA iSpec 2200.

```
1  <DOC>
2    <CHAPTER CHG = "R">
3      <TITLE><REVST>revised title<REVEND></TITLE>
4      <SECTION CHG = "U">
5        <PARA>ffffffffffffffffffffffffff</PARA>
6        <SUBSEC CHG = "U">
7          <PARA>ddddddddddddddddddddddddddd</PARA>
8        </SUBSEC>
9      </SECTION>
10   </CHAPTER>
11 </DOC>
```

Listing 3.4: Revisionsmarken bei Anker-Elementen ATA iSpec 2200

In diesem Beispiel wurde der Titel geändert. Gut zu erkennen ist, dass sich das Revisionsattribut „CHG“ nur unmittelbar auf die geänderten Anker-Elemente auswirkt, nicht jedoch auf Unter- oder Oberstrukturelemente.

3.3.3.3. Besonderheit beim Löschen

Es muss weiterhin noch auf die Besonderheit beim Löschen eines Strukturelementes eingegangen werden. Dort gibt es zwei Fälle:

1. Das Anker-Element wird durch ein neues Element ersetzt. In diesem Fall wird der Wert des Attributs „chg“ auf „n“ gesetzt.

2. Das Anker-Element verschwindet ganz. Dies wird im Revisionsmodell über das Empty-Element „deleted“ ausgedrückt und kann nur nach „revst“ auftauchen.

Folgendes Beispiel verdeutlicht den Einsatz:

```
1 <INTRO key="INTRO01" chg="D" revdate="19941201">
2   <REVST><DELETED><REVEND>
3 </INTRO>
```

Listing 3.5: Einsatz des Elementes „deleted“ ATA iSpec 2200

Optional, aber doch erwähnenswert, ist das Element „chgdescr“, welches dem Verfasser die Möglichkeit bietet seine Änderungen in Form eines Kommentars zu begründen.

Relevant ist das Revisionsmodell aus zwei Gründen:

1. Beim 3-Wege-Merge und der Änderungsdarstellung dürfen die Revisionsangaben der Hersteller nicht in den Vergleich mit einbezogen werden. Diese sollen so wie vom Hersteller ausgeliefert erhalten bleiben.

2. Ein angenehmer Nebeneffekt ist, dass beim XML-Export aus DocSurf auf Basis der Change Codes „n“, “d“, „r“ oder „u“ die einzelnen Tasks des AMMs nach diesen Change Codes kategorisiert werden. Diese Kategorisierung erlaubt eine Einschränkung der Tasks auf revidierte Tasks, welche sich somit von den unveränderten Tasks unterscheiden. Liegt ein unveränderter Task vor, ist es nicht nötig einen 3-Wege-Merge auszuführen. Dadurch kann die Anzahl um ca. 80 Prozent reduziert werden.

3.3.4. Textgrafiken

Whitespaces[2] sind in sämtlichen Elementen der iSpec 2200 irrelevant und können dadurch normalisiert werden. Das Element „txtgrphc“ ist die einzige Ausnahme bei der Whitespace relevant ist und dient der Darstellung von Textgrafiken [ATA06, Kap. 2.1 Entity,Text Graphic]. Eine Textgrafik stellt ein Bild, Diagramm oder Tabelle mit Hilfe von reinem Text dar. Bei Textgrafiken sind Whitespaces signifikant. Zeilen einer Textgrafik werden über das Element „txtline“ realisiert. Normalerweise werden vom Parser unnötige Leerzeichen entfernt, um beim Versenden des XML-Dokumentes den Speicherbedarf zu reduzieren. Die Whitespace-Behandlung wird im Abschnitt 3.4 genauer betrachtet. Die XML-Spezifikation sieht aber durch das Attribut „xml:space“ mit dem Wert „preserve“ vor, einer Anwendung (in unserem Fall das Merge- bzw. Diff-Tool) die Signifikanz der Leerzeichen mitzuteilen und diese nicht zu entfernen.[W3C06a, 2.10 White Space Handling]

[2] Whitespaces sind Zeichen wie Leerzeichen, Tabulatorzeichen und Zeilenumbruchzeichen wie das Zeilenvorschubszeichen und das Wagenrücklaufzeichen

3.3.5. Fazit

In der iSpec 2200 Analyse wurden die Modelle beschrieben und aufgezeigt. Die besondere Behandlung der aufgeführten Modelle lassen sich nun in Form von weiteren fachlichen Anforderungen definieren.

- Die Anker-Elemente verfügen über eine eindeutig Identität in Form eines Identifiers, dies beeinflusst das Vergleichs- oder Merge-Ergebnis positiv.
- Für den Vergleich sind die Revisionsmarkups des Herstellers irrelevant. Diese werden durch eigene Markups ersetzt und können so für den Vergleich ignoriert werden.
- Beim Element „txtline“ ist Whitespaces relevant. Die Relevanz sollte einer XML-Anwendungen ausdrücklich mitgeteilt werden.
- Die Auflösung der ID/IDREF-Konflikte wird in dieser Arbeit nicht weiter verfolgt. Bei der Gültigkeitsprüfung gegenüber der DTD eines Parsers sollten diese Konflikte erkannt werden. Der Autor muss darauf hingewiesen werden, dass durch einen 3-Wege-Merge der Referenzmechanismus inkonsistent werden kann. Eine nähere Betrachtung wäre hier erforderlich.

3.4. Besonderheiten von XML-Dokumenten

Im Gegensatz zu unstrukturierten Textdokumenten, weisen XML-Dokumente eine wohldefinierte interne Struktur auf [W3C06a, 3. Logical Structures]. Diese Eigenschaft erfordert eine strukturorientierte Betrachtungsweise der XML-Dokumente. Dies wird ebenso in Arbeiten und Artikeln von [HM05, Lin01, Del08] festgestellt. Die Elemente eines XML-Dokumentes können und müssen als Einheit betrachtet werden. Die Repräsentation eines XML-Dokumentes als Textdatei spielt hier nur eine untergeordnete Rolle. Dass Whitespaces bei der Erstellung oder Transformation von XML-Dokumenten vorkommen, ist keine Seltenheit. Auch die Ausgabe der XML Datei in nur einer einzigen Zeile ist durchaus üblich.

Festzuhalten ist:

linear: als linear bezeichnet man Elemente in einer geordneten Liste, beispielsweise Zeilen in einem Textdokument

strukturorientiert: als strukturorientiert bezeichnet man Elemente, die in einer Baumstruktur abgebildet werden können, beispielsweise Elemente in einem XML-Dokument

Um die Unterschiede und die Konsequenz der beiden Ansätze linear und strukturorientiert zu verstehen, ist eine genauere Betrachtung der herkömmlichen Änderungserkennung bei Textdokumenten hilfreich.

3.4.1. Änderungserkennung bei Textdokumenten

Bei Textdokumenten erfolgt ein linearer Vergleich. Die Änderungserkennung in Textdokumenten gilt als gelöst [HM76]. Der Erkennungs-Algorithmus für den Vergleich arbeitet das Dokument zeilenweise ab und berechnet dabei die Longest Common Subsequence (LCS). Zeilen die nicht enthalten sind, werden entweder gelöscht oder hinzugefügt. Der LCS Algorithmus findet in bekannten Anwendungen wie das GNU Diff [3], CVS [4] oder Subversion [5] seine Anwendung.

3.4.2. Auswirkungen der linearen Betrachtungsweise von XML-Dokumenten

Doch welche Folgen hätte nun ein linearer Ansatz bei XML-Dokumenten?
Eine Änderungserkennung, die einen linearen Ansatz (zeilenbasierter Vergleich) verfolgt, kann nur ein unbefriedigendes Ergebnis liefern, da die interne Struktur und die Besonderheiten verschiedener Konzepte von XML nicht betrachtet wird. Der lineare Vergleich eines XML-Dokumentes würde zum einen Unterschiede erkennen wo keine sind, zum anderen könnte die Änderungsinformation nur ein sehr grobes Ergebnis liefern und keine Rückschlüsse auf die Änderungen zulassen. Das extremste Beispiel hierfür wäre, wenn die Dateirepräsentation eines XML-Dokumentes nur in einer einzigen Zeile steht.
Stattdessen kann ein strukturorientierter Vergleich die einzelnen XML-Elemente (beginnend mit einem Start-Tag - endend mit einem End-Tag) erkennen und diese als zu vergleichende Einheit betrachten. Es werden also nicht einzelne Zeilen miteinander verglichen, sondern Knoten und Teilbäume. Nachfolgend sind die Besonderheiten von XML namentlich erwähnt und mit Beispielen verdeutlicht.

3.4.2.1. Reihenfolge von Attributen

Die Reihenfolge von Attributen ist irrelevant. Es kann eine direkte Zuordnung aufgrund der Name-Wert-Paare erfolgen.

```
1  <element attribute1="a" attribute2="b" />

3  <element attribute2="b" attribute1="a" />
```

Listing 3.6: Identische Element, aber unterschiedliche Testrepräsentation

[3] http://www.gnu.org/software/diffutils/diffutils.html Stand: 21.02.2008
[4] http://www.nongnu.org/cvs/ Stand: 21.02.2008
[5] http://subversion.tigris.org/ Stand: 21.02.2008

3.4.2.2. Unterschiedliche Schreibweise von Empty-Elementen

Empty-Elemente können in ihrer textlichen Darstellung unterschiedlich ausgedrückt werden sind aber identisch. Empty-Elemente mit einem schließenden Start-Tag `<empty/>` oder mit leerem Start-Tag und End-Tag `<empty></empty>`.

3.4.2.3. Namespace

Namespaces[W3C06b] dienen der Kombination von verschiedenen XML-Anwendungen innerhalb einer Dokumentinstanz. Es gibt dabei verschiedene Deklarationsmöglichkeiten. Das Konzept der Namespaces wird von einem linearen Vergleich oder Merge nicht erkannt. Das folgende Beispiel verdeutlicht den Einsatz von Namensräumen:

```
1 <person xmlns="http://www.example.com/person">
2 <vorname>Tobias</vorname>
3 <nachname>Hutzler</nachname>
4 <geburtstag>05.01.1979</geburtstag>
5 </person>
```

Listing 3.7: XML-Dokument mit einem Default-Namespace

```
1 <simpleperson:person xmlns:simpleperson="http://www.example.com/person">
2 <simpleperson:vorname>Tobias</simpleperson:vorname>
3 <simpleperson:nachname>Hutzler</simpleperson:nachname>
4 <geburtstag>05.01.1979</simpleperson:geburtstag>
5 </simpleperson:person>
```

Listing 3.8: XML-Dokument mit dem Namespace „simpleperson“

Für eine XML-Applikation handelt es sich hierbei um ein identisches XML-Dokumenten, ein linearer Vergleich würde Unterschiede feststellen.

3.4.2.4. Umgang mit Whitespace

Ein besonderes Augenmerk muss auf den Umgang mit Whitespace gelegt werden. Whitespaces sind Zeichen wie Leerzeichen, Tabulatorzeichen und Zeilenumbruchzeichen wie das Zeilenvorschubszeichen und das Wagenrücklaufzeichen.
Hierbei muss zwischen Whitespace im Markup und Whitespace innerhalb von Elementen unterschieden werden.

Umgang mit Whitespace innerhalb des Markups:
Whitespace innerhalb des Markups ist für XML-Anwendungen völlig irrelevant und wird auf ein einziges Leerzeichen reduziert[Bra00, Kap.7 S. 98-100]. Das folgende Beispiel [Bra00, Kap.7 S. 98] ist für eine XML-Applikation genau gleich.

```
1 <book                                  <book issue="'3"' date="'15/3/97"'>
2 issue   ="'3"'
3 date    ="'15/3/97"'    >
```

Listing 3.9: Starttag eines „book“-Elementes mit und ohne Whitespace

Umgang mit Whitespace innerhalb von Elementen:
Wann soll Whitespace innerhalb eines Elementes ignoriert werden?
Dazu führt Neil Bradley ein ganzes Kapitel in seinem Buch [Bra00, Kap.7 S. 98-106] auf, das sich mit der Whitespace-Thematik beschäftigt. Er zieht hierbei einen Vergleich zu SGML und HTML, wie dort Whitespace behandelt wird und spricht abschließend eine Empfehlung für den Umgang mit Whitespace und Zeilenumbrüchen in Form von Regeln[Bra00, Kap.7 S. 106] aus, um eine möglichst hohe Kompatibilität unter XML-Anwendungen zu erreichen.
Die Konsequenz daraus ist, dass jeglicher Whitespace auf ein Zeichen reduziert wird, außer es wird explizit über das Attribut „xml:space“ mit dem Wert „preserve“ angeben [W3C06a, 2.10 White Space Handling]. Eine Strukturierung des XML-Dokumentes mit Hilfe von Whitespace und Zeilenumbrüchen ist somit nicht möglich. Ein linearer (zeilenweiser) Vergleich, welcher sich auf diese Strukturierung stützt, kann nur ein unbefriedigendes Ergebnis liefern.

3.4.2.5. Content Encoding

XML unterstützt verschiedene Encodings.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <text>&#196;&#xdc;Ö</text>
3 <?xml version="1.0" encoding="ISO-8895-1"?>
4 <text>&#xc4;&#220;&#xd6;</text>
```

Listing 3.10: Die beiden XML-Dokumente sind identisch, aber ihre Text-Repräsentation unterscheiden sich.

3.4.3. Fazit

Die Änderungserkennung und der 3-Wege-Merge eines XML-Dokumentes sollte nicht auf dessen lineare Repräsentation in einer Datei erfolgen, sondern die interne Struktur (siehe Grundlagen 2.2.2) berücksichtigen. Deshalb eignen sich Anwendungen, wie Unix Diff oder CVS, nicht für den Vergleich und das Zusammenführen von XML-Dokumenten. Der Ansatz der linearen Herangehensweise der Änderungserkennung und -darstellung sowie des 3-Wege-Merges wird hinsichtlich der zuvor aufgeführten Gründe in dieser Arbeit nicht weiter verfolgt.

3.5. Analyse der Vergleichsprogramme von XML-Dokumenten

In der Analyse der Vergleichsprogramme von XML-Dokumenten werden die Programme *DeltaXML core* und das *3DM Tool* auf ihre Funktionalität, Darstellung der Differenzen und Besonderheiten hin untersucht.

3.5.1. Methode

1. Zu Beginn werden die einheitliche Kriterien definiert, nach denen die Analyse durchgeführt wird. Diese Kriterien basieren auf einer Studienarbeit von Hottinger und Meyer [HM05, S. 18-20]. Die Kriterien gliedern sich in folgende zwei Kategorien:

   Einfache Änderungen: Diese Änderungen sind für die untersuchten Programme leicht handhabbar und z. T. vom eingesetzten XML-Parser abhängig.

   Verschiebungen und Kopien: Verschiebungen und Kopien in XML-Dokumenten zu erkennen, kann als schwierig betrachtet werden, da in den meisten Fällen die Komplexität der Algorithmen steigt. Betrachtet man XML-Dokumente als ungeordnete Bäume wird das Problem NP-Vollständig [CAW99, ZS89, S. 17].

2. Für jedes Kriterium wurde ein entsprechender Testfall angelegt. Jeder Testfall besteht aus zwei XML-Dokumenten, welche das entsprechende Kriterium abdecken(Die Testfälle befinden sich auf der beigefügten CD-Rom)

3. Das Ergebnis der Testdurchläufe wird in Form der Tabellen 3.2 und 3.4 festgehalten.

4. Zudem wurde anhand der Produktbeschreibung im Fall von *DeltaXML core* nach weiteren Besonderheiten gesucht.

5. Die Repräsentation der Änderungen ist im Hinblick auf die einfache Implementierung ein weiteres Kriterium für die anschließende Diskussion.

6. Abschließend sollen die Ergebnisse dieser Analyse diskutiert werden, dabei soll ein besonderes Augenmerk auf die prototypische Implementierung gelegt werden.

3.5.2. Kriterien

Die nun folgende Aufstellung zeigt die Kriterien für den Vergleich von XML Dokumenten auf. Dabei werden sämtliche Operationen die sich auf Änderungen auswirken in Betracht gezogen. Diese Kriterien wurden in Rahmen einer Studienarbeit von [HM05, S. 18-20] festgelegt, und können auch als solche für diese Arbeit übernommen werden. Die Kriterien sind in *einfache Änderungen* und in *Verschiebungen und Kopien* kategorisiert.

3.5.2.1. Einfache Änderungen

Für die aufgeführten Änderungen wurden die entsprechenden Testfälle erstellt. Die Testfälle und Ergebnisse finden sich auf der beiliegenden CDRom [6].

A01 Einfügen eines Elementes

A02 Einfügen von mehreren Tags unter dem selben Parent-Element

A03 Löschen eines Elementes

A04 Löschen von mehreren Tags unter dem selben Parent-Element

A05 Umbenennen eines Elementes

A06 Umbenennen von mehreren Tags unter dem selben Parent-Element

A07 Einfügen eines Attributs

A08 Einfügen von mehreren Attributen im selben Element

A09 Löschen eines Attributs

A10 Löschen von mehreren Attributen im selben Element

A11 Umbenennen eines Attributs

A12 Umbenennen von mehreren Attributen im selben Element

A13 Ändern der Reihenfolge von mehreren Attributen im selben Element (Da die Reihenfolge der Attribute nicht relevant ist, sollten solche Änderungen nicht angezeigt werden. Ein Häkchen bedeutet, dass die Änderung nicht angezeigt wird. Dieser Punkt zeigt deutlich die Überlegenheit der spezialisierten Algorithmen gegenüber dem zeilenbasierten Diff.)

A14 Ändern des Wertes eines Attributes

[6] Es wurde für jedes aufgeführtes Kriterium A01-A23, als auch M01-M03 entsprechende Testfälle erstellt. Ein Testfall besteht immer aus drei XML-Dateien. Original, Änderung (die das Kriterium abdeckt) und Ergebnis.

A15 Ändern des Wertes von mehreren Attributen im selben Element

A16 Ändern von Text

A17 Ändern von Text in Kommentaren

A18 Ändern von CDATA-Daten

A19 Normales Diff für nicht strukturierte Daten (Text, CDATA, Kommentare)

A20 Ändern des Zeichensatzes bei unveränderten Daten (dies sollte nicht als Änderung erkannt werden)

A21 Schema/DTD herbeigezogen (Whitespace, . . .)?

A22 Namespaces

A23 Verschiedene Schreibweisen für leere Tags (<abc></abc> sollte das Gleiche sein wie <abc/>, das wird oft vom Parser übernommen)

Die Kriterien A21 bis A23 sind mehr vom Paser abhängig als von den entsprechenden Algorithmen.

	A01 Element einfügen	A02 Elemente einfügen	A03 Element löschen	A04 Elemente löschen	A05 Element umbenennen	A06 Elemente umbenennen	A07 Attribut einfügen	A08 Attribute einfügen	A09 Attribut löschen	A10 Attribute löschen	A11 Attribut umbenennen	A12 Attribute umbenennen	A13 Reihenfolge Attribute	A14 Attributwert ändern	A15 Attributwerte ändern	A16 Text ändern	A17 Kommentar ändern	A18 CDATA	A20 Zeichensatz ändern	A21 DTD	A22 Namensräume	A23 leere Tags
DeltaXML core	✓	✓	✓	✓	±	±	✓	✓	✓	✓	±	±	–	✓	✓	✓	–	✓	–	–	✓	–
3DM Tool	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	✓	✓	✓	–	✓	–	–	✓	–

Tabelle 3.2.: Vergleichstabelle für einfache Änderungen

- ✓ Änderungen erkannt
- ± Änderungen nur als Folge von Delete und Insert erkannt
- – Änderungen nicht erkannt

Tabelle 3.3.: Legende für die Vergleichstabellen

Wie aus der Gegenüberstellung der beiden Tools zu erkennen ist, arbeiten beide Tools wie erwartet und können die Anforderung erfüllen.

3.5.2.2. Verschiebungen und Kopien

Für das Verschieben verwende ich in dieser Arbeit ebenfalls die Kriterien von Hottinger und Meyer [HM05, Seite 20]. Allerdings betrachte ich nicht die Kriterien M04 und M05, da diese für diesen Anwendungsfall nicht relevant sind. Die Testfälle sind ebenfalls dem Verzeichnis Sourcecode der CD zu entnehmen.

M01 Verschiebung eines unveränderten Teilbaumes

M02 Verschiebung eines veränderten Teilbaumes erkennen und Änderungen an diesem anzeigen

M03 Kopieren eines Teilbaumes

	M01 Verschiebungen	M02 Verschiebungen mit Änderungen	M03 Teilbaum kopieren
DeltaXML core	±	±	±
3DM Tool	✓	✓	✓

Tabelle 3.4.: Vergleichstabelle für Verschiebungen und Kopien

3.5.3. Änderungsdarstellung

In der Änderungsdarstellung unterscheiden sich die Tool erheblich. Das *3DM Tool* stellt die Änderungen in Form einer eigenen *Diff-Datei* dar, die auf der internen Daten-Repräsentation des Vergleichsalgorithmus basiert. Die Elemente werden durch ihre Position im Baum dargestellt:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2   <diff>
3     <diff:copy src="2" dst="1" />
4     <diff:insert dst="1">
5       <b />
6       <c />
7     </diff:insert>
8 </diff>
```

Listing 3.11: Das Änderungsergebnis des *3DM Tools* aus dem Kriterium **A02**

DeltaXML core dagegen erweitert das Basisdokument um spezielle Elemente und Attribute um die Änderungen darzustellen [Fon01, GC02]:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <root xmlns:deltaxml="http://www.deltaxml.com/ns/well-formed-delta-v1" deltaxml:delta="WFmodify">
3   <a deltaxml:delta="unchanged"/>
4   <b deltaxml:delta="add"/>
5   <c deltaxml:delta="add"/>
6 </root>
```

Listing 3.12: Das Änderungsergebnis von *DeltaXML core* aus dem Kriterium **A02**

Die Repräsentation eignet sich besonders für die Darstellung in andere Formate, wie beispielsweise HTML. Durch eine einfache XSLT-Transformation können die Änderungen aussagekräftig Darstellung als HTML-Dokument oder für die prototypische Implementierung in eine geeignete Darstellungsform für den Abortext Editor umgewandelt werden.

3.5.4. Besonderheit von DeltaXML

DeltaXML core/sync bieten einige interessante Besonderheiten, die für die prototypische Implementierung eingesetzt werden können.

DeltaXML key: DeltaXML bietet die Möglichkeit die Zuordnung (Tree-Matching) über ein weiteres Attribut „deltaxml:key“ festzulegen. Durch die eindeutige Identifizierung kann das Vergleichsergebnis optimiert werden. In der iSpec 2200-Analyse (siehe Kapitel 3.3.1) wurden Kandidaten (siehe Tabelle 3.1) für eine eindeutig Zuordnung identifiziert.

DeltaXML WordbyWordFilter: DeltaXML bietet darüber hinaus die Möglichkeit PCDATA-Abschnitte auf Wortebene zu vergleichen. Dadurch kann ein genaueres Vergleichsergebnis erreicht werden, welches zudem dem Revisionsmechanismus der ATA iSpec 2200 ähnelt.

3.5.5. Diskussion der Vergleichsergebnisse

Wie sind die Ergebnisse der Analyse zu interpretieren?

3.5.5.1. Einfache Änderungen

Die einfachen Änderungen können von beiden Tools erfolgreich erkannt werden und werden somit von beiden Tools unterstützt.

3.5.5.2. Verschiebungen und Kopien

Verschiebungen und Kopien werden von *DeltaXML core* nicht erkannt und können nur als eine Folge von *insert/delete*-Operationen dargestellt werden. Dies liegt am verwendeten Operations-Katalog von *DeltaXML core*. *DeltaXML core* verwendet nur die Operationen *insert*, *delete*, *change*. Die Ausweitung des Operationskataloges um die Operationen *move*, *copy* und *guel*(*uncopy* - Einschmelzen eines Teilbaums) führen zu einer höheren Komplexität des Vergleichsalgorithmus.
Kriterien für eine guten Vergleichsalgorithmus sind zum einen die Aussagekraft der Änderungsverfolgung (edit-script) und die benötigte Anzahl von Operationen, die ein XML-Dokument in das andere transformieren, der edit-distance (Der Betrag des |edit-scripts|). Während die Aussagekraft der Änderungsabfolge durch die Operationen *move* und *copy* gesteigert wird, ist dies bei der Operation *glue* sehr fraglich. Für einen Standard-Anwender wird die Änderungsabfolge dadurch oft sehr unverständlich. Dahingegen kann man mit einem größeren Operationskatalog das *edit-script* weiter minimieren. Bei gleichen Kosten für alle Operationen würden das Operationspaar *insert/delete* durch eine Operation *move* ersetzt werden. Damit steigt dann allerdings auch die Komplexität des Vergleichsalgorithmus.
Wohin kann ein Teilbaum verschoben werden? Nur auf Kindebene oder im gesamten Dokument? Dies ist ein Grund warum *DeltaXML core/sync* auf weitere Operation wie *move* und *copy* verzichten.

3.5.5.3. Darstellung der Änderungen

Die Darstellung der Änderungen des *3DM Tools* sind leider an die interne Datenrepräsentation des Tools gebunden. Der Aufwand diese Repräsentation wird in dieser Arbeit als zu hoch angesehen. Dagegen ist die Darstellung von DeltaXML für eine einfache Integration in den Arbortext besser geeignet und wird aus diesem Grund auch forciert.

3.5.5.4. Besonderheiten von DeltaXML

Die beiden Features von DeltaXML sind als sehr nützlich einzustufen und sollten für eine detailliertes und effektives Vergleichsergebnis eingesetzt werden.

3.5.6. Fazit

Dieser Abschnitt analysierte die Vergleichsprogramme *DeltaXML core* und das *3DM Tool*. Anhand einheitlicher Kriterien: einfache Änderungen, Verschiebungen und Kopien als auch die Änderungsdarstellung konnten die Programme einem Vergleich unterzogen werden. In der abschließenden Diskussion konnten die Nachteile durch die nicht unterstützten Operationen *move*, *copy* und *glue* (einschmelzen) bei *DeltaXML core* nicht klar erkannt werden. Der Vergleich auf Wort-Ebene bei *DeltaXML core* liefert bei diesem Anwendungsfall detaillierte Vergleichsergebnisse, die für den Autor sehr hilfreich[7] sind.

3.6. Analyse des 3-Wege-Merge von XML-Dokumenten

In diesem Abschnitt werden die Tools *DeltaXML sync* und das *3DM Tool* analysiert. Beim 3-Wege-Merge werden zwei unterschiedliche Versionen (Derivat1 und Derivat2) und deren Ausgangsdokument (Base) unter Berücksichtigung der Änderungen zum Ausgangsdokument zusammengeführt.
Der nächste Abschnitt beschreibt die Methode dieser Analyse.

3.6.1. Methode

1. Zuallererst werden die Anforderungen an eine 3-Wege-Merge festgelegt.
2. Da beim 3-Wege-Merge Konflikte durch Überschneidungen auftreten können, wird in diesem Schritt die Darstellung von Konflikten analysiert.
3. Es werden einheitliche Kriterien festgelegt um die beiden Tools miteinander zu vergleichen. Dazu bieten die Mergecases von Lindholm [Lin01, S. 163] eine gute Kriterien-Grundlage, werden aber noch um eigene Kriterien erweitert und unnötige gestrichen. Diese Mergecases lassen sich wie folgt kategorisieren:

   Insert I1-I3 In diesen Testfälle werden die Insert-Operationen in den Derivaten eingesetzt.

   Delete D1-D4 In diesen Testfälle werden die Delete-Operationen in den Derivaten eingesetzt.

   Copy C1-C5 In diesen Testfälle werden die Copy-Operationen in den Derivaten eingesetzt.

[7] Oft werden ganze Abschnitte eines Anker-Element (z.B. subtask, topic) im AMM-Task als revidiert gekennzeichnet und der Autor sucht bildlich gesprochen nach der Nadel im Heuhaufen

Move M1-M5 In diesen Testfälle werden die Move-Operationen in den Derivaten eingesetzt.

Update U1-U3 In diesen Testfälle werden die Update-Operationen in den Derivaten eingesetzt.

Konflikt X2-X4 Diese Testfälle zeigen Konfliktfälle durch unterschiedliche Operationen auf.

Kombinationen A1-A16 Diese Testfälle sind Kombinationen aus den Operationen.

Ergänzungen E1-E6 Diese Testfälle sind Ergänzungen zu den anderen Testfällen. Diese berücksichtigen die Änderungen auf Attribute.

Die Mergecases sind im Anhang A Mergecases zu finden.

4. Abschließend werden die gewonnenen Erkenntnisse diskutiert

Der nächste Abschnitt befasst sich mit den Anforderungen an den 3-Wege-Merge.

3.6.2. Anforderungen an den 3-Wege-Merge

1. Der 3-Wege-Merge muss alle Änderungen beider Derivate gegenüber dem Original erkennen.
2. Der 3-Wege-Merge muss Konflikte erkennen und diese auch als solche behandeln.
3. Konflikte sollen für den Anwendungsfall durch den Autoren aufgelöst werden. Ein „Silent-“Mode, der nach bestem Ermessen die Dokumente im Konfliktfall zusammenführt, darf nicht ohne den Hinweis auf diesen erfolgen.
4. Es müssen folgende Konflikttypen erkannt werden:

   update-update Konflikt: Dieser tritt immer dann auf wenn sich zwei Update-Operationen auf das gleiche Element auswirken.

   update-delete, (move-delete, copy-delete): Dieser Konflikt tritt immer dann auf, wenn im eine Derivat ein Elemente gelöscht wird, welches im anderen Derivat geändert wird und umgekehrt.

   move-move Konflikt: Ein Element wird an zwei unterschiedliche Ziele im Dokument verschoben.

   append-append Konflikt: Ein Sonderfall, der bei einer insert-insert Operation oder move-move Operation auftreten kann, was dann eine Reihenfolgeverletzung verursacht.

5. Die Konflikte sollen aussagekräftig und verständlich dargestellt werden.

3.6.3. Konfliktdarstellung

Bei der Konfliktdarstellung weichen die Darstellungsform ähnlich wie beim Vergleich der beiden Tools *DeltaXML sync* und *3DM Tool* stark voneinander ab.

3.6.3.1. Konfliktdarstellung bei *DeltaXML sync*

DeltaXML sync verwendet für die Darstellung von Konflikten drei Regeln und zeigt diese im zusammengeführten Dokument an. Dies hat den Vorteil, das eine Bearbeitung durch den Autor direkt im Dokument erfolgen kann [NWF04, WF04]. Eine Regel für Elemente, Attribute und PCDATA.

Es wird repräsentativ für jede Regeln ein Konflikt dargestellt. Ein Element-Konflikt ist in Abbildung 3.7 dargestellt, ein PCDATA-Konflikt in Abbildung 3.8 und ein Attribut-Konflikt in 3.9.

Konfliktregel	Base	Edit1	Edit2	Merge
Change in edit1, deletion in edit2	<e> <a/> </e>	<e> <a/> <b1/> </e>	<e> <a/> </e>	<e> <a/> <dxce:elementConflict dxce:type="modified-edit1-deleted-edit2"> <dxce:base> </dxce:base> <dxce:edit1> <b1/> </dxce:edit1> </dxce:elementConflict> </e>

Abbildung 3.7.: Zeigt die Darstellungform eines Element-Konflikt

Konfliktregel	Base	Edit1	Edit2	Merge
Change in edit1, deletion in edit2	<e>3</e>	<e>1</e>	<e></e>	<e> <dxce:pcdataConflict dxce:type="modified-edit1-deleted-edit2"> <dxce:base>3</dxce:base> <dxce:edit1>1</dxce:edit1> </dxce:pcdataConflict> </e>

Abbildung 3.8.: Zeigt die Darstellungsform eines PCDATA-Konflikt

Konfliktregel	Base	Edit1	Edit2	Merge
Values change both, different values	<e a="3"/>	<e a="1"/>	<e a="2"/>	<e> <dxce:attributeConflicts> <a dxce:type="three-way-conflict"> <dxce:base>3</dxce:base> <dxce:edit1>1</dxce:edit1> <dxce:edit2>2</dxce:edit2> </dxce:attributeConflicts> </e>

Abbildung 3.9.: Zeigt die Darstellungsform eines Attribut-Konfliktes

3.6.3.2. Konfliktdarstellung bei *3DM Tool*

Das 3DM Tool stellt die Konflikte nicht im zusammengeführten Dokument dar, sondern protokolliert die aufgetretenen Konflikte in einer Logdatei. Diese basiert, wie auch beim Vergleich auf die interne Datenrepräsentations des Merge-Algorithmus.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <conflictlist>
3  <warnings>
4    <delete>Modifications in deleted subtree.
5      <node tree="merged" path="/0" />
6      <node tree="base" path="/0/2" />
7      <node tree="branch1" path="/0/2" />
8    </delete>
9  </warnings>
10 </conflictlist>
```

Listing 3.13: Das Änderungsergebnis von *DeltaXML core* aus dem Kriterium **A02**

3.6.3.3. Der Append-Append Konflikt

Der *Append-Append Konflikt* kann durch folgende Änderungen auftreten. Siehe hierzu Abbildung 3.10 und Abbildung 3.11

Konfliktregel	Base	Edit1	Edit2	Merge
Append-Append Konflikt	<e> <a/> <b/> </e>	<e> <a/> <b/> <c1/> </e>	<e> <a/> <b/> <c2/> </e>	<e> <a/> <b/> <c1/> <c2/> </e>

Abbildung 3.10.: Zeigt eine Append Append Konflikt in beiden Derivaten wird ein neues Element an letzter Stelle des gleichen Vater-Element eingefügt

Beide Programme, sowohl *DeltaXML sync* als auch das *3DM Tool* stellen diese Art des Konfliktes nicht dar. Es wird von keinem die Reihenfolge berücksichtigt. Das *3DM Tool* führt

Konfliktregel	Base	Edit1	Edit2	Merge
Append- Append Konflikt	<e> <a/> <c/> </e>	<e> <a> <c/> </e>	<e> <a> <c/> </e>	<e> <a> <c/> <c/> </e>

Abbildung 3.11.: In beiden Derivaten werden unterschiedliche Elementen in ein anderes Element verschoben

die Änderungen zusammen und schreibt eine Warnung in die Logdatei.

DeltaXML arbeitet an einer Behebung dieses Problems. Die Konflikterkennung wird in einer der nächsten Versionen von *Deltaxml sync* eingepflegt. Für den Prototypen kann der Konflikt über einen Vergleich zur Base, also dem alten AMM-Task durch zwei „*adds*“ angezeigt werden. Dieser Work-Around ist für die prototypische Implementierung akzeptabel.

3.6.4. Diskussion der Ergebnisse aus den Mergecases

Auch hier stellt sich die Frage wie die Ergebnisse aus dem Anhang A zu interpretieren sind.

3.6.4.1. Verschiebungen und Kopien

Der erste Unterschied der sofort auffällt, ist der unterschiedliche Operations-Katalog. *DeltaXML sync* verzichtet auch hier auf die Operation *move* und *copy* [Fon02]. Das *3DM Tool* wendet die Operationen *copy* und *move* an, wodurch es in einigen Mergecases zu unterschiedlichen Ergebnissen kommt. Dies wird an den Mergecases **A5**, **A8**, **A13** und **A14** deutlich und exemplarisch am Mergecase **A5** 3.12 diskutiert:

Case	Base	Derivat1	Derivat2	3DM Merge	Delta Merge
A5	<R> <a/> </R>	<R> <a1/> </R>	<R> <a/> <a/> </R>	<R> <a1/> <a1/> </R>	<R> <a1/> <a/> </R>

Abbildung 3.12.: Mergecase A5: Update, Copy. Unterschiedliches Merge-Ergebnis.

Case	Base	Derivat1	Derivat2	3DM Merge	Delta Merge
D4	<R> <a> <c/> </R>	<R> <a> <c/> </R>	<R> <a> <c/> </R>	<R> <a> <c/> </R>	<R> <a> <c/> <c/> </R>

Abbildung 3.13.: Der Mergecas D4

Das Element „a“ wird im Derivat1 in „a1“ geändert. Im Derivat2 wird wird das Element „a“ nach „b“ kopiert. Das *3DM Tool* erkennt diese Kopie und führt die Änderung aus Derivat1 auch auf das kopierte Element unter b aus.
DeltaXML sync hingegen erkennt nicht, dass es sich dabei um eine Kopie handelt. Es wird zwar die Änderung beim Element a erkannt, aber nicht, das „a“ nach „b“ kopiert wurde. Die Gründe der unterschiedlichen Ergebnisse wurden bereits in 3.5.5.2 diskutiert. Doch stellt sich die Frage, welche Auswirkungen dies auf die Schlussfolgerungen des Autors hat? Anzunehmen ist, wenn der Autor eine Paragraphen in eine Liste kopiert und dieser Paragraph durch den Hersteller geändert wird, ob diese Änderung auch auf den Paragraphen in der Liste angewendet werden sollte. Die Abbildung 3.13 zeigt ebenfalls eine starken Umbau der Dokumentstruktur auf. Es werden dort Teilbäume eines Elementes gelöscht und verschoben. Diese beiden Fälle können nicht als Stärke von *DeltaXML sync* eingestuft werden. Fraglich ist allerdings, ob dieser Fall bei der Erstellung und Revisionskontrolle häufig oder überhaupt auftritt, es wird versucht Redundanzen zu vermeiden und den Inhalt der Arbeitskarte knapp und aussagekräftig für die Wartungsmechaniker zusammenzufassen. Bei der Suche nach geeigneten Testfällen der fachlichen Dokumente wurde deutlich, dass die Änderungen des Herstellers oft nur minimal sind und sich meist auf Textpassagen reduzieren. Komplexe Strukturänderungen eines AMM-Task sind sehr unwahrscheinlich.
Die Abbildung 3.14 zeigt zwei Verschiebungen.

DeltaXML sync führt das Element „a“ an beiden Positionen im zusammengeführten Dokument auf.
Bei allen anderen Mergecases zeigten beide Tools ein sehr zufriedenstellendes Ergebnis. Der Autor kann durch einen Vergleich zur Vorgängerversion die Unterschiede erkennen und je nach Belangen den Arbeitskartenvorschlag editieren.

Case	Base	Derivat1	Derivat2	3DM Merge	Delta Merge
M4	<R> <a/> <c/> </R>	<R> <a/> <c/> </R>	<R> <c/> <a/> </R>	<R> <a /> <c /> </R>	<R> <a/> <c/> <a/> </R>

Abbildung 3.14.: Mergecase M4

3.6.4.2. Darstellung der Konflikte

Konflikte werden bei DeltaXML Sync im zusammengeführten XML-Dokument dargestellt. Das 3DM Tool führt den 3-Wege-Merge immer aus und protokolliert die Warnungen über eine Konflikt in einer Logdatei. Nachteil daran ist das diese Logdatei auf der internen Datenrepräsentation beruht.

3.6.5. Fazit

Zu Beginn der Analyse des 3-Wege-Merge wurden die Anforderungen an den 3-Wege-Merge definiert. Anhand der einheitlichen Kriterien (Mergecases und Darstellung der Konflikte) konnten *DeltaXML sync* mit dem *3DM Tool* verglichen werden. Die Ergebnisse vielen sehr unterschiedlich aus. So liegen die Vorteile bei *DeltaXML sync* in der Darstellung der Konflikte, wohingegen die Unterstützung der Operationen *move* und *copy* des *3DM Tool* zum Teil zu besseren Zusammenführungsergebnissen führte.

3.7. Ergebnis der Analyse und Anforderungsanalyse

Basierend auf den Analysen aus 3.1 bis 3.6 werden die Anforderungen an den konzeptionellen Entwurf und die anschließenden prototypische Implementierung zur Unterstützung des Revisionsprozess der Arbeitskartenerstellung definiert.
Die Anforderungen lassen sich in fachliche, technische, funktionale und nicht funktionale Anforderungen kategorisieren.

Die Einstufung der Anforderungen wird in drei Prioritätsklassen vorgenommen. Anforderungen die der Prototyp unbedingt umsetzen muss, werden als „must“ klassifiziert, Anforderungen die der Prototyp unterstützen soll als „should“ und alle anderen Anforderungen, die wünschenswert sind, als „nice to have“.

3.7.1. Fachliche Anforderungen

Die fachliche Anforderungen ergeben sich aus den Analysen 3.1 bis 3.3.

must Die Revisionsangaben des Herstellers müssen entfernt werden.

must Beim Element „txtline“ darf kein Whitespace entfernt werden.

3.7.2. Technische Anforderungen

Die technischen Anforderungen ergeben sich aus den Analysen 3.5 und 3.6. *DeltaXML core/sync* unterstützt die Anforderungen XML-Dokumente zu vergleichen und zusammenzuführen. Eine aussagekräftige Darstellungsform der Änderungen und Konflikte ist gewährleistet.

should Der Vergleich für Textknoten soll auf Word-Ebene stattfinden, um ein besseres Vergleichsergebnis zu erreichen.

should Für eine bessere Zuordnung soll nach Möglichkeit die Unterstützung des DeltaXML-Keying eingesetzt werden.

should Das Markup soll nach dem 3-Wege-Merge und Vergleich von technisch hinzugefügtem Markup bereinigt werden.

3.7.3. Funktionale Anforderungen

Die funktionalen Anforderungen betreffen die Interaktion des Benutzers mit dem System.

must Der Autor kann über das System einen Arbeitskartenvorschlag für eine neuen Revisionszyklus aus der vorherigen Arbeitskarte, dem vorherigen AMM-Task und dem aktuellen AMM-Task automatisch erstellen lassen.

must Der Autor muss die fachlichen Dokumente (Arbeitskarte, AMM-Task, TR und SB) miteinander vergleichen können, um die Änderungen zu lokalisieren und die Revisionskontrolle zu vereinfachen.

nice to have Der Autor kann aus dem Arbeitskartenvorschlag eine für den Publikationsprozess automatisch erzeugtes Revisionsmarkup erzeugen.

must Die Änderungen müssen dem Benutzer in einer aussagekräftigen Darstellung innerhalb des fachlichen Dokumentes präsentiert werden.

must Der Benutzer muss Konflikte die beim Zusammenführen entstehen können manuell auflösen.

must Der Benutzer muss die Konflikte klar als solche erkennen können.

nice to have Der Benutzer kann die Konflikte software-gestützt auflösen.

3.7.4. nichtfunktionale Anforderungen

3.7.4.1. Korrektheit

Die Korrektheit ist eine wichtige nichtfunktionale Anforderungen bei Unterstützung der Revisionskontrolle. An den 3-Wege-Merge und Vergleich werden hohe Ansprüche gestellt. Diese müssen zum einen die Besonderheiten der XML-Dokumente als auch die Anforderungen aus den fachlichen Dokumenten unterstützen.

3.7.4.2. Vollständigkeit

Diese Anforderung erreicht neben der Korrektheit den gleichen Stellenwert. Es dürfen keine Änderungen verloren gehen und keine wesentlichen Informationen der fachlichen Dokumenten entfernt werden.

3.7.4.3. Weitere nichtfunktionalen Anforderungen

Auf die weiteren nichtfunktionalen Anforderungen wie Benutzbarkeit, Skalierbarkeit, Erweiterbarkeit, Robustheit und Performanz kann eigentlich kein vernünftiges Softwaresystem verzichten und werden in diesem Prototyp soweit wie möglich berücksichtigt. Diese beeinflussen den Systementwurf und die Umsetzung wesentlich.

4. Design

Dieses Kapitel beschreibt das beabsichtigte Design der prototypischen Implementierung. Das Design soll die festgelegten Anforderungen der Prioritätsklassen „must“ und „should“ (siehe 3.7) umsetzen. Die Bedienerfreundlichkeit und Integration in den Gesamtprozess standen nicht im Vordergrund. Trotzdem ist eine Integration des Prototypen als Plugin für den *Arbortext Editor* angestrebt. Ziel dieser Arbeit ist, die prinzipielle Machbarkeit der Revisionskontrolle durch den 3-Wege-Merge erfolgreich nachzuweisen. Das Plugin wird nicht produktiv eingesetzt.

4.1. Verwendete Entwicklungsumgebung und Sprachen

Bevor in den beiden nächsten Kapiteln Design und ausgewählte Aspekte der Realisierung vorgestellt werden, gibt dieser Abschnitt einen Überblick über die verwendete Entwicklungsumgebung.

4.1.1. Java 5.0

Bei der Frage nach der geeigneten Programmiersprache für die Entwicklung des Prototypen fiel die Wahl auf Java 5.0. Folgende Aspekte der Rahmenbedingungen sprechen für Java 5.0:

- Der Arbortext Editor unterstützt neben weiteren Sprachen Java 5.0. Die Integration ist dadurch möglich.
- Die Programme *DeltaXML core/sync* werden als Klassenbibliotheken für JAVA ausgeliefert.
- Die Sprache bietet eine Reihe von komfortablen Klassenbibliotheken für die Verarbeitung von XML-Dokumenten an.
- Java ist eine objektorientierte Programmiersprache.

4.1.2. Arbortext Command Language (ACL)

Arbortext Command Language (ACL) [PDC07b] ist eine flexible Scriptsprache. In Anlehnung an die Programmiersprache C ist es möglich, die Anwendung *Arbortext Editor* durch einfache und komplexe Anwendungen anzupassen. ACL unterstützt das Verwenden von Java-Bibliotheken durch Aufruf statischer Methoden.

4.1.3. UML

Die Unified Modelling Language (UML) ist eine Modellierungssprache. Es handelt sich um eine (überwiegend grafische) Notation, die dazu verwendet wird, die unterschiedlichen Aspekte eines Software-Entwurfs auszudrücken. Das Software-Design des entwickelten Prototyps lässt sich durch eine Reihe von Diagrammtypen unterstützen und beschreiben. Eingesetzt werden hier folgende Diagramme [OOS06]:

Komponenten-Diagramm um einen groben Überblick auf die Softwarearchitektur auf Komponentenebene darzustellen

Klassendiagramme für die statischen Darstellung von Klassen und ihrer Beziehung zueinander

Ablaufdiagramme für die dynamische Darstellung von Klassen und ihrer Beziehung zur Laufzeit

4.1.4. Entwurfsmuster

Entwurfsmuster [GHJV95, ABW97] bieten eine einfache und elegante Lösung für spezielle Probleme des objektorientierten Softwareentwurfs. Sie erfassen Problemlösungen, die im Laufe der Zeit gefunden wurden, und sich weiterentwickelt haben. Entwurfsmuster verfolgen neben dem Problemlösungsvorschlag noch das Ziel, die Software möglichst flexibel und erweiterbar zu entwerfen bzw. umzusetzen.

Im Rahmen dieser Arbeit werden die nachstehend aufgeführten Entwurfsmuster für das Design und Implementierung verwendet:

Fassade: Das Fassade-Muster bietet eine einheitliche Schnittstelle zu einer Vielzahl von Schnittstellen eines Subsystems. Ziel der Fassade ist es, die Benutzbarkeit eines Subsystems zu vereinfachen, um so die Komplexität des Subsystems zu reduzieren. Die Klienten müssen nicht mehr mit mehreren Objekten des Subsystems, sondern nur

mit der Fassade kommunizieren. Die Fassade kennt die notwendigen Komponenten des Subsystems und kann - wenn nötig - die einzelnen Ergebnisse zu einem Gesamtergebnis subsumieren. Die Fassade nimmt nicht notwendigerweise die Möglichkeit, dennoch auf Klassen des Subsystems zugreifen zu können, falls dies für bestimmte Klienten notwendig sein sollte. Für die meisten Klienten bietet sie aber die Möglichkeit, vereinfacht auf die Dienste eines komplexen Subsystems zugreifen zu können. Eine Fassade kann ebenfalls genutzt werden, wenn man Subsysteme in Schichten aufteilen möchte. Jede Fassade stellt dann einen Eintrittspunkt zu einer Subsystemschicht dar [GHJV95, ABW97].

Einsatz findet die Fassade in dieser Arbeit, um die Anwendungslogik von der Präsentationsschicht über eine Schnittstelle nutzen zu können, ohne die konkreten Klassen der Anwendungslogik kennen zu müssen.

Konkrete Fabrik: Eine konkrete Fabrik ist ein Teil des Musters *Abstrakte Fabrik* (abstract factory), welches eine Schnittstelle für die Erzeugung von Familien verwandter oder voneinander abhängiger Objekte bietet, ohne ihre konkrete Klasse zu benennen. Die konkrete Fabrik stellt Methoden zur Erzeugung von Objekten einer Klassenfamilie zur Verfügung [GHJV95, ABW97].

Für die konkreten XMLFilter wird eine konkrete Fabrik implementiert, welche die Erzeugung dieser Übernimmt. Es werden weitere konkrete Fabriken für die Erzeugung konkreter Objekte der Vergleichs- und 3-Wege-Merge-Klassen verwendet.

Singelton: Das Singelton-Muster (singelton-pattern) stellt sicher, dass es von einer Klasse nur eine Instanz gibt, welche meist global verfügbar ist [GHJV95, ABW97].

Die vorher erwähnten konkreten Fabrik-Klassen sind nach diesem Muster entworfen. Das garantiert, dass nur eine Fabrik für den entsprechenden Zweck existiert.

4.2. Prozessablauf

Die Anwendungsfälle: Erstellung einer Arbeitskarte und Vergleich der fachlichen Dokumente, wird hier unter Berücksichtigung der fachlichen und technischen Analyse aus 3.7 entworfen. Diese können als Prozessablaufketten dargestellt werden. Bevor die konkreten Prozessketten beschrieben werden, ist eine Einführung der Klassenbibliotheken *DeltaXML core/sync* erforderlich.

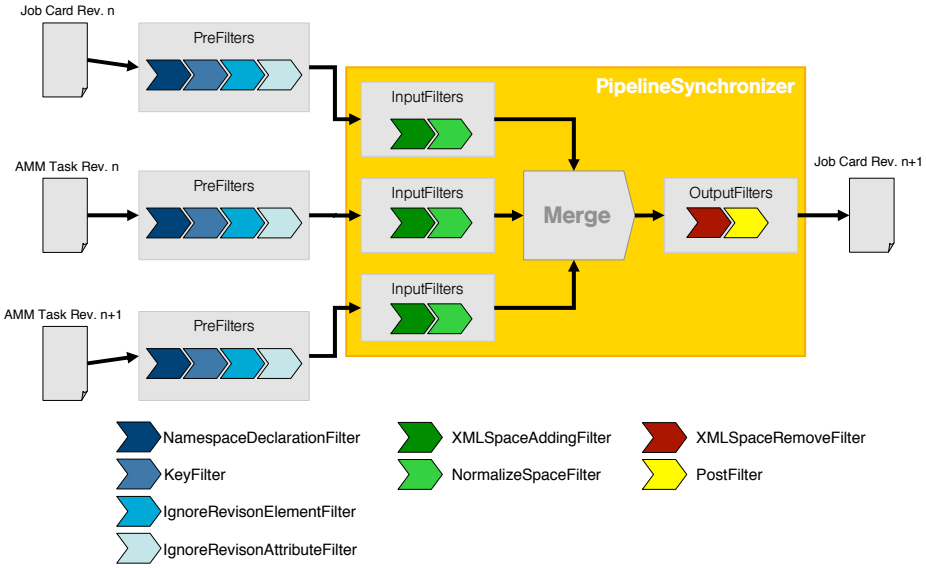


Abbildung 4.1.: Prozessablauf der Arbeitskartenerstellung durch den 3-Wege-Merge.

4.2.1. Klassenbibliotheken DeltaXML core/sync

DeltaXML core/sync bieten in den Klassenbibliotheken *deltaxml.jar* und *sync.jar* die Hauptfunktionen „Vergleichen“ durch die Klasse *PipelineComparator* und den „3-Wege-Merge“ in der Klasse *PipelineSynchronizer* an. Um eine Anpassung an den einzelnen Anwendungsfall zu ermöglichen, können die XML-Dokumente vor- bzw. nachbearbeitet werden. Dies erfolg durch spezielle Filter. Es wird ein Repertoire von Filtern für die Vor- bzw. Nach-Filterung bereitgestellt. Diese Filter können durch eigene Filter ergänzt werden. Nachteil an diesem Konzept ist allerdings, dass die parametrisierbaren Filter[1] nur eingeschränkt nutzbar sind, da nur einfache Typen (z.b. *Boolean*, *String*) übergeben werden können. Es soll allerdings möglich sein Strukturen wie *Listen* oder *Maps* zu übergeben. Um dieses Defizit auszugleichen, werden vor der Verarbeitung eigene Filterketten ausgeführt, die hier als *PreFilters* bezeichnet werden. Im folgenden werden die Prozessketten für den 3-Wege-Merge bzw. den Vergleich entworfen.

4.2.2. 3-Wege-Merge zur Erstellung eines neuen Arbeitskartenvorschlages

Die Abbildung 4.1 zeigt den Prozessablauf des 3-Wege-Merge, um aus dem alten AMM-Task, der daraus resultierenden Arbeitskarte und dem neuen AMM-Task die neue Arbeitskar-

[1] ein parametrisierbarer Filter ist ein Filter, der über Parameter gesteuert werden kann.

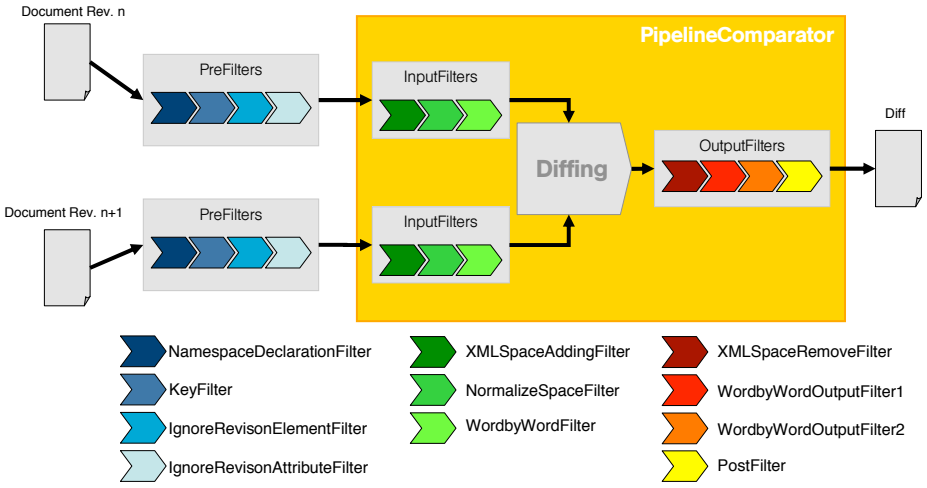


Abbildung 4.2.: Prozessablauf des Vergleiches

te zu erstellen. Dazu müssen die XML-Dokumente auf Grund der fachlichen und technischen Anforderungen mit zusätzlichen Informationen angereichert werden (oder unwesentliche Informationen entfernt werden), bevor diese zu einer neuen Arbeitskarte zusammengeführt werden. Die zusammengeführte Arbeitskarte muss von überflüssigem Markup bereinigt werden.
Das Design ist so gewählt, dass vor Ausführung des 3-Wege-Merges die übergebenen XML-Dokumente aufbereitet werden, und nach Ausführung bereinigt werden.
Die einzelnen Filter werden im Abschnitt 4.5.2 diese Kapitels beschrieben.

4.2.3. Vergleich der fachlichen Dokumenten um Änderungen für den Autor darzustellen

Die Abbildung 4.2 zeigt den Prozessablauf des Vergleichs von zwei XML-Dokumenten. Auch hier werden die XML-Dokumente vor dem Vergleich aufbereitet und nach dem Vergleich bereinigt. Die Aufbereitung durch die PreFilters entsprechen der Aufbereitung vor dem 3-Wege-Merge, dadurch ist eine Wiederverwendbarkeit gewährleistet und angestrebt. Die Input- und OutputFilter des *PipelineComparator* werden noch für den Vergleich auf Wort-Ebene durch die Filter *WordbyWordInputFilter*, *WordbyWordOuputFilter1* und *WordbyWordOutputFilter2* ergänzt.

4.2.4. Umsetzung der fachlichen und technischen Anforderungen in Filter

Die fachlichen und technischen Anforderungen sollen in Filter umgesetzt werden, welche die entsprechenden Anforderungen erfüllen.

KeyFilter: Dieser Filter ist für das Hinzufügen des Attributes „deltaxml:key“ zuständig. Über eine *Map<String,String>* werden die Elementnamen und Attribute für das „Keying“ übergeben.

NamespaceDeclarationFilter: Dieser Filter fügt dem Wurzelelement den Namespace für den Präfix: „deltaxml“ hinzu und ist für die Ausführung des *KeyFilters* erforderlich

RemoveElementFilter: Dieser Filter entfernt aus dem Dokument übergebene Elemente. Er wird genutzt, um die RevisionsMarkups des Herstellers zu entfernen.

RemoveAttributeFilter: Dieser Filter entfernt die übergebenen Attribute. Dies wird ebenfalls genutzt, um das RevisionsMarkup zu entfernen

XMLSpaceAddingFilter u. XMLSpaceRemoveFilter Diese Filter fügen vor dem Vergleich für Elemente, bei denen der Whitespace relevant ist das Attribute xml:space=“preserve“ ein, und teilt so XML-Anwendungen mit, dass kein Whitespace entfernt werden darf. Der Filter *XMLSpaceRemoveFilter* entfernt das Attribut xml:space

NormalizeSpace: Dieser Filter entfernt unnötigen Whitespace.

WordByWordInputFilter: Dieser Filter wird für den Vergleich auf Wort-Ebene verwendet.

WordByWordOutputFilter1 u. WordByWordOutputFilter2: Diese Filter sind für die Nachbereitung des .

PostFilter Dieser Filter bereinigt abschließend die fachlichen Dokumente von unnötigem Markup.

Nachfolgend wird die Architektur des Entwurfs vorgestellt.

4.3. Systemübersicht und 3-Schichten-Architektur

Die Abbildung 4.3 zeigt die abstrakte Sicht auf die Softwarearchitektur. Die technischen Komponenten, als auch die verwendeten Klassenbibliotheken sind dargestellt. Diese unterscheiden sich von den Klassen, welche später detailliert aufgezeigt werden.

Für das Design des Prototyps wurde eine 3-Schichten-Architektur gewählt. Die 3-Schichten-Architektur unterteilt die Softwarearchitektur in einzelne Aufgabenfelder, welche voneinander abhängig sind. Eine Schicht kapselt die Aufgaben und stellt sie der darüber liegenden Schicht zur Verfügung. Die Softwarearchitektur lässt sich somit in folgende Schichten unterteilen: **Präsentationsschicht**, **Anwendungslogik** und **Datenhaltung**.
Diese Schichten dürfen nicht übersprungen werden.

Von Vorteil bei dieser Architektur ist die Skalierbarkeit und Wartbarkeit. Die einzelnen

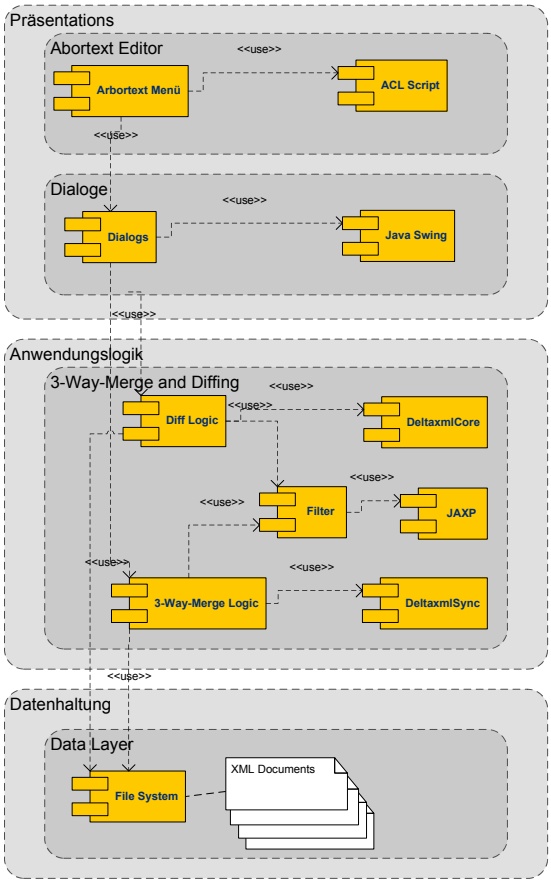


Abbildung 4.3.: Systemsicht

Schichten können so beispielsweise auf verschiedene Rechner verteilt werden. Die Aufteilung in diese Schichten bietet die Möglichkeit, den Prototypen in eine Client/Server Architektur umzubauen. So ist es möglich, die Datenhaltungsschicht durch einen zentralen Datenbankserver oder - noch naheliegender - direkt durch das DVS System auszutauschen. Die Anwendungslogik könnte auch auf eine Applikationsserver ausgelagert werden. In der Präsentationsschicht kommt der Arbortext Editor zum Einsatz, dieser könnte aus politischen oder sonstigen Gründen durch eine anderen XML-Editor ersetzt werden. Die Veränderungen

innerhalb einer Schicht haben meist nur kleine Auswirkungen auf die anderen Schichten, und vermeiden dadurch größere Manipulationen am System.

4.4. Präsentationsschicht

In der Präsentationsschicht findet die Interaktion zwischen Autor und System statt.

4.4.1. Arbortext Editor

Der Arbortext Editor [PDC07a] ist ein sehr mächtiger XML-Editor und dient den Autoren für die Erstellung der Arbeitskarten. Die Mächtigkeit wird durch die Einbindung eigener Java-Klassenbibliotheken, welche über die eigene Sprache ACL benutzt werden können, unterstrichen.
So lassen sich über ACL sowohl die Menüeinträge anpassen, als auch externe Aufrufe an die eingebundenen Klassenbibliotheken übergeben.
Des Weiteren bietet der Abortext Editor die Möglichkeit, Änderungen oder Konflikten für die Darstellung zu formatieren.[2]

Die Anwendungsfälle:

- Erstellen einer neuen Arbeitskarte durch den 3-Wege-Merge
- Vergleich der fachlichen Dokumente um Änderungen nachzuvollziehen

werden über Menüeinträge aufgerufen. In erscheinenden Dialogfenster *DialogMerge* oder *'DialogDiff*, kann der Benutzer die fachlichen Dokumente wählen und die Erstellung der Arbeitskarte oder den Vergleich der fachlichen Dokumente ausführen. Im Einzelnen wird das Design der Präsentationsschicht in einem Klassendiagramm und Ablaufdiagramm vorgestellt.

4.4.2. Klassendiagramm

Die Abbildung 4.4 zeigt die statische Sicht auf die Präsentationsschicht. Die Klassen *DialogDiff* und *DialogMerge* erben von der Klasse *JFrame*. Sie benutzen für die Dateiauswahl die

[2] Formatted Output Specification Instance(FOSI), ist eine Sprache die sich besonders für das Layout technischer Dokumentation eignet.

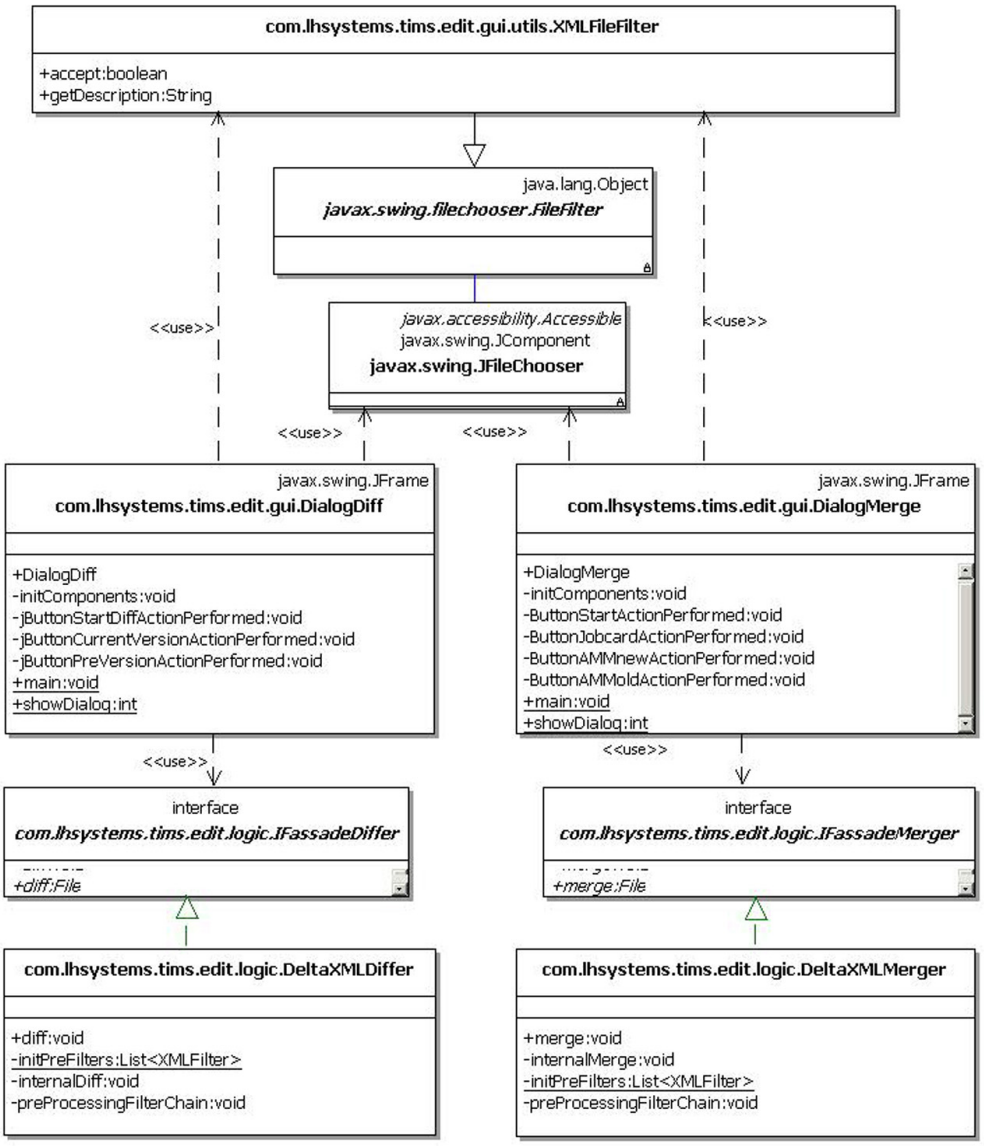


Abbildung 4.4.: Klassendiagramm der Präsentationsschicht

Klasse *JFileChosser* und übergeben dieser bei der Initialisierung die Klasse *XMLFileFilter*. Die *XMLFileFilter* Klasse erfüllt die Aufgabe der Vorselektion auf XML-Dokumenten und das Durchsuchen von Verzeichnissen.
Das Klassendiagramm 4.4 stellt darüber hinaus die Trennung zwischen der Präsentationsschicht und Anwendungslogik dar. Die Klassen *DialogDiff* und *DialogMerge* stellen in Form der Schnittstellen *IDifferFassade* und *IMergerFassade* die Funktionalität der Anwendungslogik bereit.

4.4.3. Ablaufdiagramm

Die Abbildung 4.5 stellt die dynamische Sicht auf einer möglichen Interaktion mit dem System dar. Um den Entwurf zu veranschaulichen wird das Beispiel des Vergleich eines Dokumentes gewählt. Interessant ist hier die Rückgabe des Dokumentes und Anzeige im Abortext Editor. Hierzu wird die Klassenbibliothek des Arbortext Editors *AOM.jar* verwendet. Der Anwender „Autor“ startet aus dem Menü des Arbortext Editors den Vergleich. Über ACL-Script wird die Klassenmethode „*showDialog*“ der Klasse „*DialogDiff*“ aufgerufen. Nach Auswahl der zu vergleichenden Dokumente kann der Vergleich über den Button „*vergleichen...*“ gestartet werden (aus Gründen der Übersicht nicht dargestellt). Der Dialog spricht die Vergleichsfunktion der Anwendungslogik an, welche nach Ausführung des Vergleichs das Ergebnis in Form eines Files zurückgibt. Die Klasse *Application* öffnet das Dokument, und kann dann der Klasse *ACL* die ID des Dokumentes übergeben. Die Klasse *ACL* veranlasst die Anzeige des Dokumentes im Arbortext-Editor über die Methode `Acl.func("doc_show", String.valueOf(((ADocument) diffDoc).getAclId()),"edit");.`

4.5. Anwendungslogik

Die Prozessketten zur Erstellung der Arbeitskarte durch den 3-Wege-Merge und der Vergleich der fachlichen Dokumente werden in dieser Schicht angesiedelt. Um diese Aufgabe für die Präsentationsschicht anzubieten, soll das Fassade-Muster verwendet werden. Diese führt die benötigten Teilaufgaben zusammen und bieten dem Klient einen Gesamtprozess aus den einzelnen Teilprozessen an. Im Folgenden wird das Design in Form von Klassendiagrammen dargestellt.

4.5.1. Klassendiagramm der Anwendungslogik

Die Abbildung 4.6 zeigt den Entwurf der Anwendungslogik. Die beiden Interfaces *IFassadeDiffer* und *IFassadeMerger* stellen die beiden Dienste *diff()* und *merge()* der Präsenta-

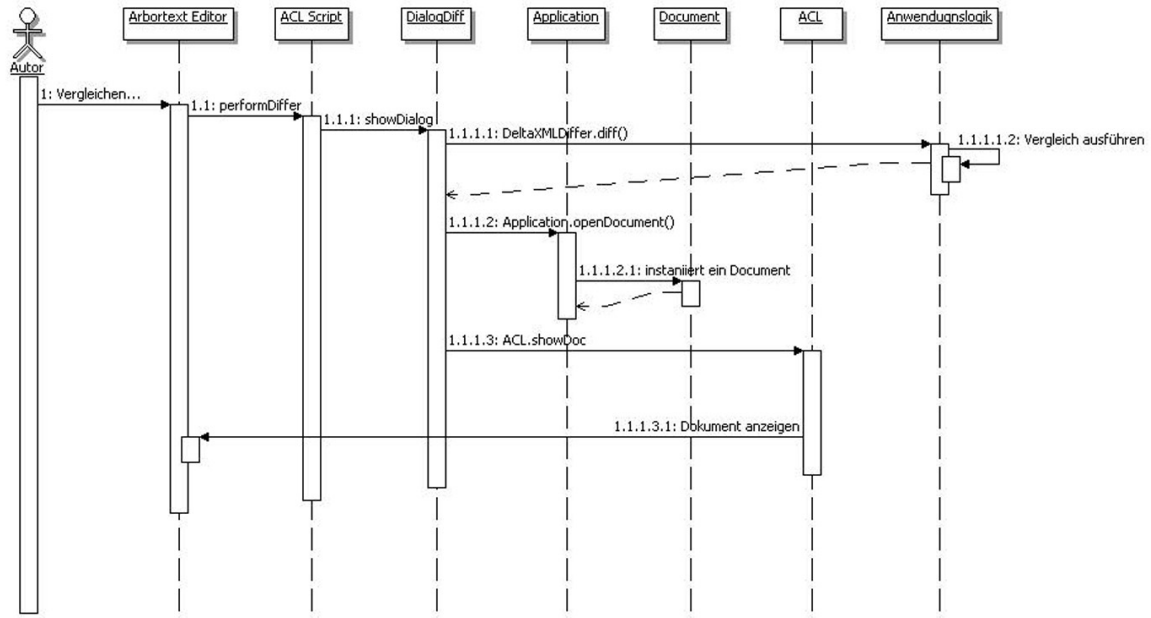


Abbildung 4.5.: Ablaufdiagramm: Der Autor startet einen Vergleich. Dargestellt ist der Ablauf und nach erfolgreicher Ausführung die Darstellung des Vergleichsdokumentes im Arbortext Editor

tionsschicht zur Verfügung. Die Klassen *DeltaXMLDiffer* und *DeltaXMLMerger* sind für die Instanziierung der benötigten Klassen zuständig, und kapseln so Ihre Aufgabendomänen „Vergleichen“ und „Zusammenführen/Erstellen“. Für diese Aufgaben sind wie in der Prozessbeschreibung dargestellt mehrere Teilschritte nötig. Die Aufbereitung der Dokumente soll in den *PreFilters* und *InputFilters* stattfinden. Nach Ausführung des Vergleichs bzw. Erstellung durch den 3-Wege-Merge sollen diese durch die *OutputFilters* bereinigt werden.
Um diese Aufgaben zu erfüllen, werden die *PreFilters* erzeugt (Fordert von der *XMLFilterFactory* die Objekte der konkreten Klassen an) und der Klasse *FilterChain* für die Verkettung der Filter und das Ausführen der Filter übergeben (eine genauere Beschreibung des Designs erfolgt später). Den eigentlichen Vergleich und das Erstellen der Arbeitskarte wird durch die Klassen *PipelineComparator* und *PipelineSynchronizer* der Klassenbibliothek *DeltaXML core/sync* angeboten. Diese lassen sich über Parameter je nach Anwendungsfall und Zweck konfigurieren. Für die prototypische Implementierung ist hier angestrebt diese Aufgabe an die konkrete Fabriken *PipelineComparatorFactory* und *PipelineSynchroniszierFactory* zu übergeben, die für das System die Objekte bereitstellen.

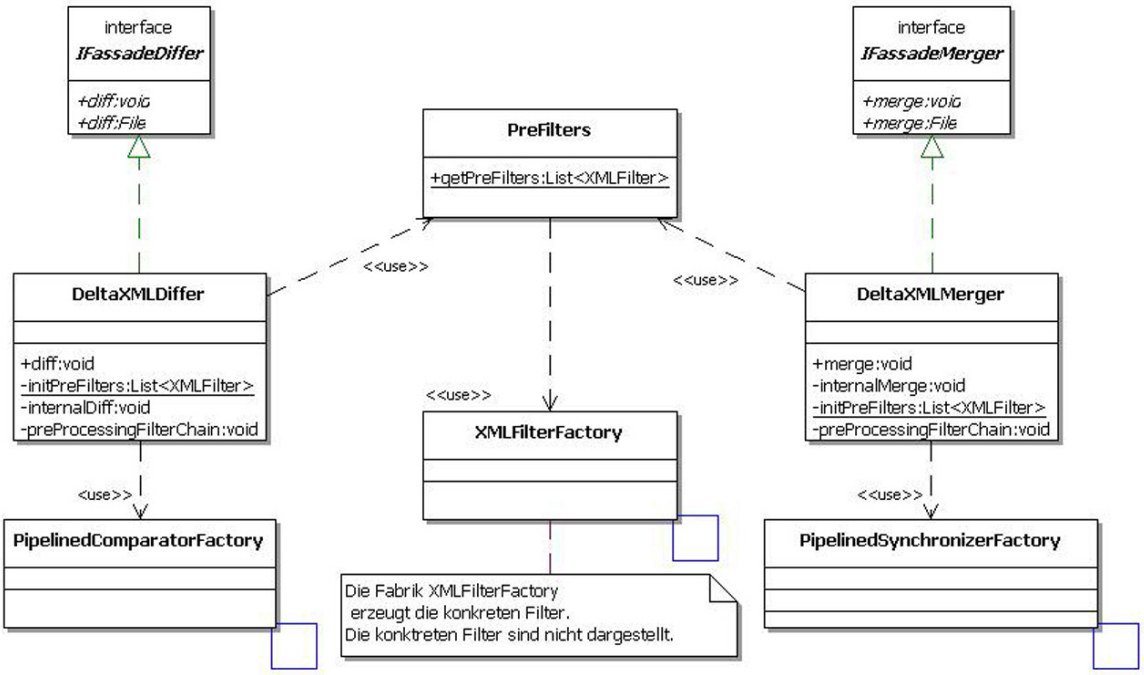


Abbildung 4.6.: Klassendiagramm der Anwendungslogik

4.5.2. Klassendiagramm der Filter-Komponente

Für die Aufbereitung der XML-Dokumente für den anschließenden Merge und Diff sollen *SAXFilter* verwendet werden. Motiviert wird diese Design-Entscheidung durch:

- Es werden nur kleine Modifikationen an den Dokumenten vorgenommen.
- Es handelt sich hierbei um einen sequenziellen Ablauf.
- Das Konzept der Filterketten ist sehr anpassbar und erweiterbar.
- Durch die Filter lassen sich einzelne fachliche und technische Anforderung in konkrete *SAXFilter* ausdrücken.
- Die Wiederverwendbarkeit durch den Einsatz der einzelnen konkreten Filter, die klar einen Aufgabenbereich erfüllen, steigt.
- Die interne Vor- und Nachfilterung, die von DeltaXML angeboten wird, weist Schwächen in der Erweiterbarkeit auf.

Die Abbildung zeigt das gewählte Design der Filter-Komponente.

An dieser Stelle erfolgt eine Beschreibung der einzelnen Schnittstellen und Klassen[3].

IFilterChain: *IFilterChain* ist die Schnittstelle der Filterkomponente. Klienten können darüber die Funktionalität der Filterkomponente nutzen.

FilterChain: Die Klasse *FilterChain* soll die Funktionalität der Filterkomponente anbieten. Die Klasse instanziiert Objekte der benötigten Klassen *XMLReader* und *Transformer*. An die Klasse wird eine *InputSource* (XML-Dokument) übergeben, auf welche die Filterkette angewendet werden soll. Durch die Methoden *AddFilter(XMLFilter filter)* und *AddFilters(List<XMLFilter> filters)* können einzelne Filter oder Listen von Filtern übergeben werden. Die Ausführung der Methode *executeFilterChain()* liefert als Ergebnis das durch die Filter modifizierte XML-Dokument.

XMLFilter: Das Interface *XMLFilter* bietet die Möglichkeit der Verkettung einzelner Filter über die Methode *setParent(XMLReader reader)*.

XMLFilterImpl: *XMLFilterImpl* implementiert das Interface *XMLFilter* und muss von den konkreten *XMLFilter* geerbt werden.

XMLFilterFactory: Auch hier soll für die Erzeugung der konkreten Filter, die entweder technische oder fachliche Anforderungen abdecken eine konkrete Fabrik verwendet werden.

[3] die konkreten Filter wurden unter Abschnitt subsec:fachtechfilter ausführlich beschrieben

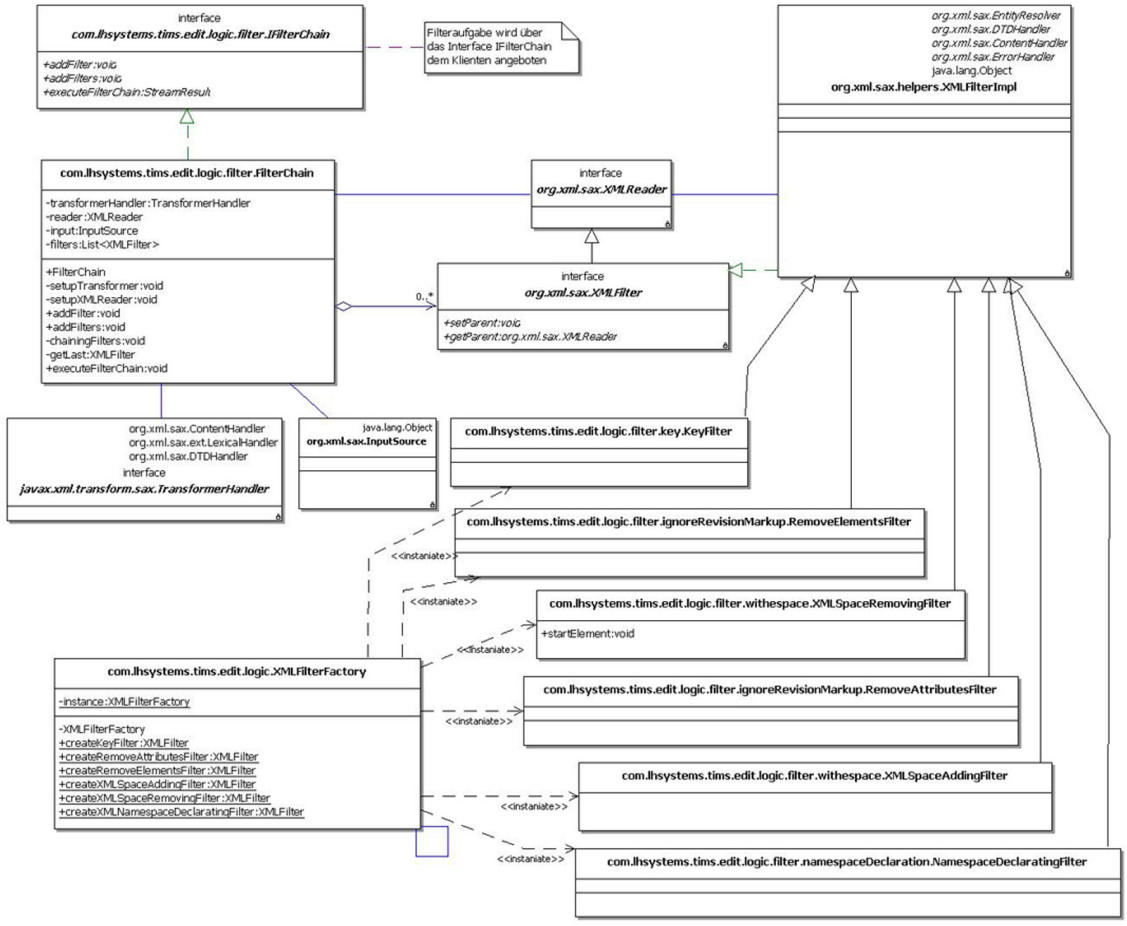


Abbildung 4.7.: Das Klassendiagramm der Filter-Komponente

4.6. Änderungs- und Konfliktdarstellung:

Die Änderungen und Konfliktdarstellung in Arbortext Editor wird über FOSI-Objekte realisiert und ist eine Anpassung der bisherigen Arbeitskarten-DTD-Lösung. Dabei wurde besonders auf die Wiederverwendbarkeit der definierten Elemente geachtet. Die FOSI-Definitionen der zu stylenden Elemente werden in Entitätsdateien gespeichert, und über Parameter-Entitäten in die *ACJ.FOS* integriert. Dies hat den Vorteil, dass diese Deklarationen durch andere Programme oder durch Style-Anpassungen einfach ausgetauscht werden können. In der Realisierung wird die Deklaration von FOSI-Objekte vorgestellt.

4.7. Prototyp

Der Prototyp soll nach dem hier vorgestellten Entwurf umgesetzt werden. Da durch Klassendiagramme die zu implementierenden Klassen benannt wurden, sowie statische und dynamische Beziehungen klar sind, ist die Realisierung des Prototyps sehr gut und effizient umzusetzen. Im Folgenden Kapitel werden einige Besonderheiten der Realisierung betrachtet.

5. Ausgewählte Aspekte der Realisierung

In diesem Abschnitt werden interessante Aspekte der Realisierung des Prototyps aufgegriffen. Die komplette Implementierung des Prototyps ist auf der beigefügten CD-Rom im Archiv source/Prototyp.zip zu finden.

5.1. Die Umsetzung des Singelton- und Factory-Pattern

Bei der Implementierung wurden die Entwurfsmuster des Designs umgesetzt. So wurden die konkreten Fabriken für *PipelineSynchronizerFactory*, *PipelineComparatorFactory* und *XMLFilterFactory* erstellt.

5.1.1. Einsatz des Singelton-Patterns

In jeder Fabrik wurde das Singelton-Pattern umgesetzt. Dazu wurde der Konstruktor auf *private* gesetzt und eine statische Methode *getInstance()* erstellt, welche beim ersten Aufruf eine Objekt der Klasse erzeugt, speichert und zurückgibt. Jeder weitere Aufruf der Methode *getInstance()* gibt die Klassenvaribale *instance* zurück. Somit kann sichergestellt werden, dass im gesamten System nur ein Objekt der Klasse zur Laufzeit existiert.

```
1  public class PipelinedSynchronizerFactory {
2
3    private static PipelinedSynchronizerFactory instance = null;
4
5    /**
6     *  private use getInstance()
7     */
8    private PipelinedSynchronizerFactory() {
9    }
10
11   /**
12    *
13    * @return Singelton der Factory
14    */
15   public PipelinedSynchronizerFactory getInstance() {
16     if (instance == null) {
```

```
17          instance = new PipelinedSynchronizerFactory();
18        }

20        return instance;

22      }
```

Listing 5.1: Singelton-Pattern am Beispiel der PipelinedSynchronizerFactory

5.1.2. Einsatz der konkreten Fabrik zu Erzeugung der Objekte einer Klassenfamilie

Die eigentliche Fabrik liefert die Objekte einer Klassenfamilie. Als Beispiel ist hier der Sourcecode der Klasse *XMLFilterFactory* dargestellt:

```
2       /**
3        * Instances a KeyFilter for the <code>Map<String,String></code> of
4        * Element and Attribute used for deltaxml:key
5        *
6        * @param keyAttributes
7        * @return Instance of KeyFilter
8        */
9       public static XMLFilter createKeyFilter(Map<String, String> keyAttributes) {
10        KeyFilter keyFilter = new KeyFilter(keyAttributes);
11        return keyFilter;
12      }

14      /**
15       *
16       * @return Instance of XMLSpaceAddingFilter
17       */
18      public static XMLFilter createXMLSpaceAddingFilter() {
19        XMLSpaceAddingFilter xmlSpaceAddingFilter = new XMLSpaceAddingFilter();
20        return xmlSpaceAddingFilter;
21      }

23      /**
24       *
25       * @return Instance of XMLSpaceRemovingFilter
26       */
27      public static XMLFilter createXMLSpaceRemovingFilter() {
28        XMLSpaceRemovingFilter xmlSpaceRemovingFilter = new XMLSpaceRemovingFilter();
29        return xmlSpaceRemovingFilter;

31      }
```

Listing 5.2: Auszug der Klasse XMLFilterFactory

5.2. Das Konzept der Filter-Verkettung und die Realisierung durch die Klasse FilterChain

Die Vorfilterung der XML-Dokumente durchläuft eine Filterkette nach dem Beispiel aus [ABB+05, S.311-318] (siehe auch [WN04]). Motiviert wird dieses Konzept der Konkatenation - die Aneinanderreihung von mehreren Filter - dadurch, dass das Ergebnis einer Transformation eines Filter an den nächsten Filter weitergegeben wird.
Im Entwurf für die Realisierung wurde eine bestimmte fachliche oder technische Anforderung durch einen Filter umgesetzt, um so die Wiederverwendbarkeit zu gewährleisten, und die Aufgabendomänen voneinander zu trennen. Grundlage für die Verkettung ist, das jeder Filter von der Klasse *XMLFilterImpl* (siehe Abbildung 5.1) erbt, welche die Schnittstelle *XMLFilter* implementiert. Die Schnittstelle *XMLFilter* ermöglicht die Verkettung von Filtern über die Methode *setParent(XMLReader reader)*.
Die Klasse *XMLFilterImpl* kann durch die Implementierung des Interfaces *ContentHandler* SAX Events [1] konsumieren als auch über das Interface XMLReader die Rolle als Reader einnehmen, um SAX Events zu erzeugen. Die Abbildung 5.1 stellt in Form eines Klassendiagramms die Zusammenhänge dar. Das Konzept, dass ein Filter durch die Implementierung beiden Interfaces *XMLReader* und *ContentHandler* die Erzeugung und das Empfangen von SAX Events einnimmt, ermöglicht es, dass die Ausgabe des einen, die Eingabe des andern Filters ist.

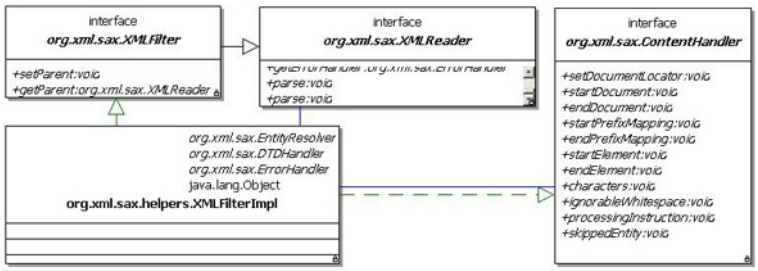


Abbildung 5.1.: Interfaces und Abstracte Klassen der SAXFilter

5.2.1. Ablauf der Filterkette

An den ersten Filter wird mit der Methode *setParent(XMLReader reader)* ein *XMLReader* übergeben, welcher das XML-Dokument als *InportSource* einliest. Dieser Filter ist für den

[1] Ein SAX Event, ist ein Ereignis, welches beim Parsen eines XML-Dokumentes auftritt. So wird beispielsweise durch ein Start-Tag, also der Beginn eines Elementes das Ereignis *startElement(...)* ausgelöst.

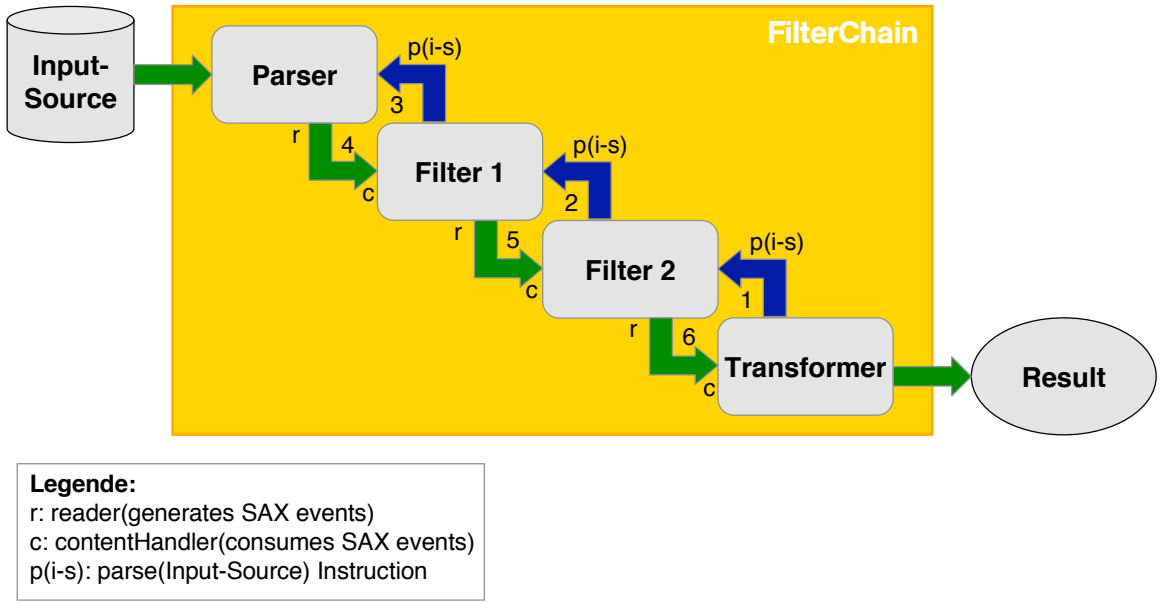


Abbildung 5.2.: Operation der verketteten Filter nach [ABB+05, S.315].

nächsten Filter wiederum der Parent. Dem Letzten Filter in der Kette wird der ContentHandler übergeben, der das Ergebnis in Form eines SteamResult[2] ausgibt. Die Verarbeitung der Filterkette wird über den letzten Filter durch die Methode *parse(InputSource i-s)* gestartet.

Die Abbildung 5.2 verdeutlicht die Ausführung der Filterkette.

Der Ablauf in der Abbildung 5.2 gezeigt und ist dort [ABB+05, S.315] wie folgt beschrieben:

1. The transformer sets up an internal object as the content handler for filter2 and tells it to parse the input source.
2. Filter2, in turn, sets itself up as the content handler for filter1 and tells it to parse the input source.
3. Filter1, in turn, tells the parser object to parse the input source.
4. The Parser does so, generating SAX events, which it passes to filter1.
5. Filter1, acting in its capacity as a content handler, processes the events and does its transformations. Then, acting in its capacity as a SAX reader (XMLReader), it sends SAX events to filter2.

[2] An eine StreamResult kann ein FileOutputStream übergeben werden, welcher das Ergebnis in eine Datei schreibt

6. Filter2 does the same, sending its events to the transformer's content handler, which generates the output stream.

5.2.2. Realisierung im Protoypen durch die Klasse FilterChain

Die Klasse FilterChain realisiert das vorgestellte Konzept der Verkettung von Filtern. Durch die privaten Methoden setupReader() und setupTransformer() werden ein Reader als auch ein Transformer erzeugt. Über die Methoden addFilter(XMLFilter filter) und addFilters(List<XMLFilter> filters) können die Filter übergeben werden. Durch die Methode executeFilterChain(InputSource input,SreamResult result) wird die Filterkette auf eine InputSource angewandt und in ein StreamResult ausgegeben. Diese Methode ruft die Private-Methode chainingFilter() vor Ausführung der Filter auf.
Die Funktionalität wird über das Interface *IFilterChain* bereitgestellt und bietet eine sehr flexible Handhabung.

5.2.3. Konkreter Filter am Beispiel von KeyFilter

Im folgenden wird beispielhaft für die Implementierung eines konkreten Filters die Klasse *KeyFilter* ausgewählt. Die Klasse *KeyFilter* hat die Aufgabe, für bestimmte Elemente das Attribut *deltaxml:key* einzufügen. Die Zuordnungspaare aus Elementnamen und Attributnamen werden als *Map<String,String>* übergeben.

```
1  public class KeyFilter extends XMLFilterImpl {
2
3    private String attNameKey;
4
5    private AttributesImpl attributes = new AttributesImpl();
6
7    private Map<String, String> keyAttributes;
8
9    /**
10    * Creates a KeyFilter with the given key attributes <code>Map</code>.
11    *
12    * @param keyAttributes
13    *            a <code>java.util.Map</code> object that contains mappings
14    *            of element names to attribute names
15    */
16   public KeyFilter(Map<String, String> keyAttributes) {
17     this.keyAttributes = keyAttributes;
18   }
19
20   /**
21    * @see org.xml.sax.helpers.XMLFilterImpl#startPrefixMapping(
22    *      java.lang.String, java.lang.String)
23    */
24   @Override
25   public void startPrefixMapping(String prefix, String uri)
26       throws SAXException {
```

```
27      if (uri.equals(DeltaXMLConstants.NAMESPACE_URI)) {
28        // store the DeltaXML prefix
29        attNameKey = prefix + ':' + DeltaXMLConstants.ATT_KEY;
30      }
31      super.startPrefixMapping(prefix, uri);
32    }

34    /**
35     * @see org.xml.sax.helpers.XMLFilterImpl#startElement(java.lang.String,
36     *      java.lang.String, java.lang.String, org.xml.sax.Attributes)
37     */
38    @Override
39    public void startElement(String uri, String localName, String qName,
40        Attributes atts) throws SAXException {
41      String keyAttributeName = keyAttributes.get(qName);
42      if (keyAttributeName != null) {
43        String key = atts.getValue(keyAttributeName);
44        if (key != null) {
45          // add the DeltaXML key attribute
46          attributes.setAttributes(atts);
47          attributes.addAttribute(DeltaXMLConstants.NAMESPACE_URI,
48              DeltaXMLConstants.ATT_KEY, attNameKey, "CDATA", key);

50          super.startElement(uri, localName, qName, attributes);

52          attributes.clear();
53        } else {
54          super.startElement(uri, localName, qName, atts);
55        }
56      } else {
57        super.startElement(uri, localName, qName, atts);
58      }
59    }
60  }
```

Listing 5.3: KeyFilter.java. Es werden die Methoden überschrieben, für die SAX Events, in welchen der Filter Modifikationen am Dokument vornehmen soll.

startPrefixMapping: Diese Methode speichert den Attributname mit Präfix (deltaxml:key).

startElement: In dieser Methode wird dem Elemente aus der *Map<String,String> keyAttributes* das Attribut „deltaxml:key“ hinzugefügt. Der Wert des Attributes „deltaxml:key“ ergibt sich aus dem Wert des zugeordneten Attributes. Vereinfacht gesprochen: Es wird der Attributwert eines bestimmten Attributes (z.B. <task key='wert') in das Attribut deltaxml:key kopiert.

5.3. Nicht realisierte wichtige Funktionen

Dieser Abschnitt geht auf die nicht realisierten aber wichtigen Funktionen ein.

5.3.1. Erzeugen einer endgültigen Arbeitskarte aus dem Vorschlag

Die Erzeugung einer endgültigen Arbeitskarte für die anschließende Publikation erfordert die Transformation der Änderungen in ein eigenes Revisionsmarkup, ähnlich dessen des Herstellers. Das Revisionsmarkup weist mit Hilfe von Revisionsbalken den Leser (in erster Linie den Wartungsmechaniker) auf neue Inhalte im Text hin. Diese Anforderung wurde als „nice to have“ eingestuft, und aus zeitlichen Gründen nicht umgesetzt.
Eine Umsetzung der Lösung wäre leicht möglich, da alle Änderungsinformationen vorhanden sind. Elemente und Attribute die als „gelöscht“ gekennzeichnet sind, werden entfernt. Die Elemente und Attribute, welche als „neu“ gekennzeichnet sind würden durch das Revisionsmarkup aufgewertet.

5.3.2. Konfigurierbarkeit des Plugins

Für den produktiven Einsatz ist es von Vorteil, das Plugin konfigurieren zu können. Besonders bei den parametrisierbaren Filter würde sich diese Funktionalität als äußerst nützlich erweisen. So könnte der Autor eigenständig die Elemente wählen, die für den Vergleich irrelevant sind, oder welche mit einer eindeutigen Key für ein optimiertes Vergleichs- und Zusammenführungsergebnis benutzt werden sollen. Die prinzipielle Machbarkeit wurde im Prototypen nachgewiesen nur auf eine komfortable Konfigurationsmaske wurde verzichtet.

5.4. Fazit

Dieses Kapitel hat ausgewählte Aspekte der Realisierung vorgestellt. Es wurde die Umsetzung der Entwurfsmuster konkrete Fabrik und Singelton eingegangen und an Auszügen des Java Quellcodes verdeutlicht.
Des Weiteren wurde auf das Konzept und die Realisierung der Verkettung von Filtern eingegangen und exemplarisch der Filter *KeyFilter* vorgestellt.
Natürlich eignet sich ein Prototyp nicht für den produktiven Einsatz, da neben den wichtigen nicht realisierten Funktionen(siehe Abschnitt 5.3) noch kleinere Schönheitsfehler behoben werden sollten.
In der Implementierung wurden die „must“ und „should“ Anforderungen aus der Analyse umgesetzt. Die Implementierung dient aber dem Erkenntnisgewinn und der Einschätzung von Problemen, welche bei deiner Produktentwicklung auftreten können.
Die Vision den 3-Wege-Merge für die Erstellung von Arbeitskarten einzusetzen, konnte konzeptionell umgesetzt werden.

6. Zusammenfassung

Um die Arbeit zusammenzufassen, wird auf den aktuellen Stand der Entwicklung und Ausblick eingegangen. Abschließend erfolgt eine Bewertung der Arbeit.

6.1. Stand der Entwicklung und Ausblick

Bis zum Abschluss dieser Arbeit sind sowohl die „must“ als auch die „should“ Anforderungen aus der Analyse 3.7 konzeptionell im Entwurf entwickelt und prototypisch implementiert worden.
Die Erstellung eines Arbeitskartenvorschlages ist erfüllt. Durch Auswahl der entsprechenden fachlichen Dokumente lässt sich eine Arbeitskarte erstellen, die der Autor nach kurzer Durchsicht direkt verwenden kann. Im seltenen Falle eines Konfliktes kann der Autor diesen auflösen. Unterstützt wird der Autor durch die uneingeschränkte Vergleichsmöglichkeit, die der Prototyp bietet. Über eine beliebige Kombination der fachlichen Dokumente beim Vergleich, lassen sich je nach Bedarf die Änderungen nachvollziehen. Die Erstellung einer endgültigen Arbeitskarte, welche das Erzeugen des Revisionsmarkups für den Publikationsprozess erfordert, wurde wie geplant nicht umgesetzt. Das Generieren des Revisionsmarkups ist aus den berechneten Änderungsinformationen durch den 3-Wege-Merge mittels einer einfachen Transformation möglich, lediglich müsste die DTD der Arbeitskarte angepasst werden.
Auch stand die Benutzbarkeit der Konfliktauflösung und Änderungsübernahme nicht im Vordergrund. Hier müsste ein mit den Autoren nahes Konzept für eine benutzerfreundliche Oberfläche erarbeitet werden.
Die Datenhaltungsschicht wurde im Prototyp stark vereinfacht. Die Anforderung, die fachlichen Dokumente in einer Datenbank zu speichern, wird auf jeden Fall an ein Produktivsystem herangetragen. Durch die klare Schichtenaufteilung der Architektur im Entwurf könnte dies ohne einen Komplettumbau des Systems erfolgen.

6.2. Bewertung der Arbeit

Diese Arbeit eröffnet für die Erstellung und Revisionskontrolle der internen Wartungsdokumentation neue Wege und zeigt, dass dieser komplexe Prozess durch Einhaltung der Konzepte umsetzbar ist.
Der Einsatz des 3-Wege-Merge zur Erstellung und Revisionskontrolle der Arbeitskarten konnte durch den Prototyp konzeptionell erfolgreich nachgewiesen werden.
Einige Fluggesellschaften als auch Hersteller weichen von der ATA iSpec 2200, welche den Austausch und Inhalt der Wartungsdokumente standardisiert, ab. Für diese könnte die hier erarbeitete Lösung ein weiterer Anreiz sein, den Standard konsequenter umzusetzen.
Fluggesellschaften, die den iSpec 2200 Standard bereits einsetzen, können durch die Ergebnisse dieser Arbeit weitere Vorteile für einen noch effektiveren Revisions-Prozess erzielen.

Tabellenverzeichnis

Abbildungsverzeichnis

Literaturverzeichnis

[ABB+05] ARMSTRONG, ERIC, JENNIFER BALL, STEPHANIE BODOFF, DEBBIE BODE CARSON, IAN EVANS, DALE GREEN, KIM HAASE und ERIC JENDROCK: *The J2EE™ 1.4 Tutorial*. Webseite, 2005. http://java.sun.com/j2ee/1.4/docs/tutorial/doc/J2EETutorial.pdf Stand: 12.01.2008.

[ABW97] ALPERT, SHERMAN R., KYLE BROWN und BOBBY WOOLF: *The Design Patterns Smalltalk Companion*. ADDISON-WESLEY, 1997.

[ATA06] ATA: *ATA ISpec2200*, 2006.

[Bra00] BRADLEY, NEIL: *The XML Companion*. Addison-Wesley, 2nd Auflage, 2000.

[C397] C3: *The C3 Project at Stanford*, 1997. http://infolab.stanford.edu/c3/c3.html Stand: 23.01.2008.

[CAW99] CHAWATHE, SUDARSHAN S., SERGE ABITEBOUL und JENNIFER WIDOM: *Managing Historical Semistructured Data*. Theory and Practice of Object Systems, 5(3):143–162, 1999.

[CGM97] CHAWATHE, SUDARSHAN S. und HECTOR GARCIA-MOLINA: *Meaningful change detection in structured data*. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Seiten 26–37, 1997. http://infolab.stanford.edu/c3/papers/ps/bbdiff.ps Stand: 15.12.2007.

[CRGMW96] CHAWATHE, SUDARSHAN S., ANAND RAJARAMAN, HECTOR GARCIA-MOLINA und JENNIFER WIDOM: *Change detection in hierarchically structured information*. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Seiten 493–504, Montréal, Québec, June 1996. http://infolab.stanford.edu/c3/papers/ps/tdiff3-8.ps Stand: 15.12.2007.

[CSFP07] COLLINS-SUSSMAN, BEN, BRIAN W. FITZPATRICK und C. MICHAEL PILATO: *Version Control with Subversion*, 2007 Stand: 8.11.2007. http://svnbook.redbean.com/en/1.4/svn-book.pdf Stand 12.02.2008.

[Del08] DELTAXML: *DeltaXML Products*, 2008. http://www.deltaxml.com/ Stand: 21.02.2008.

[Fon01] FONTAINE, ROBIN LA: *A Delta Format for XML: Identifying Changes in XML Files and Representing the Changes in XML*, 2001. http://www.gca.org/papers/xmleurope2001/papers/pdf/s29-2.pdf Stand: 12.10.2007.

[Fon02] FONTAINE, ROBIN LA: *Merging XML files: a new approach providing intelligent merge of XML data sets*, 2002. http://www.deltaxml.com/dxml/93/version/default/part/AttachmentData/data/merging-xml-files.pdf Stand: 12.10.2007.

[GC02] GRÉGORY COBÉNA, TALEL ABDESSALEM, YASSINE HINNACH: *A comparative study for XML change detection*, 2002. ftp://ftp.inria.fr/INRIA/Projects/verso/VersoReport-221.pdf Stand: 8.11.2007.

[GHJV95] GAMMA, ERICH, RICHARD HELM, RALPH JOHNSON und JOHN VLISSIDES: *Design Patterns: Elements of Reusable ObjectOriented Software*. ADDISON-WESLEY, 1995.

[HM76] HUNT, J. W. und M. D. MCILROY: *An Algorithm for Differential File Comparison*. Technischer Bericht CSTR 41, Bell Laboratories, Murray Hill, NJ, 1976. http://www.cs.dartmouth.edu/ doug/diff.ps Stand 15.12.2007.

[HM05] HOTTINGER, DANIEL und FRANZISKA MEYER: *XML-Diff-Algorithmen*. Technischer Bericht, ETH Zürich, 2005.

[Lin01] LINDHOLM, T.: *A 3-way Merging Algorithm for Synchronizing Ordered Trees — the 3DM merging and differencing tool for XML*. Diplomarbeit, Helsinki University of Technology, Dept. of Computer Science, September 2001. http://www.cs.hut.fi/ ctl/3dm/thesis.pdf Stand: 14.02.2008.

[Lin06] LINDHOLM, TANCRED: *The 3DM XML 3-way Merging and Differencing Tool*, 2006. http://www.cs.hut.fi/ ctl/3dm/index.html Stand: 17.02.2008.

[LSY07] LSY: *DocManage*, 2007. CD:<resources/LSY07/DocManage-engl.pdf>.

[Man00] MANGER, GERALD W.: *A Generic Algorithm for Merging SGML/XML-Instances*. Diplomarbeit, Johann Wolfgang Goethe-University, Frankfurt a. Main, Mai 2000. http://www.gca.org/papers/xmleurope2001/papers/html/s29-1.html Stand: 14.02.2008.

[Mye86] MYERS, EUGENE W.: *An O(ND) Difference Algorithm and Its Variations*. Algorithmica, 1(2):251–266, 1986.

[NWF04] NICHOLS, THOMAS, NIGEL WHITAKER und ROBIN LA FONTAINE: *Conflict Resolution in XML - Forms For All*, 2004. http://www.gca.org/papers/xmleurope2001/papers/pdf/s29-2.pdf Stand: 12.10.2007.

[OOS06] OOSE: *UML-Notationsübersicht*, 2006. http://www.oose.de/downloads/uml-2-Notationsuebersicht-oose.de.pdf Stand: 12.02.2008.

[PDC07a] PDC: *Arbortext Editor*, 2007. http://www.ptc.com/appserver/mkt/products/home.jsp?k=3591 Stand: 17.02.2008.

[PDC07b] PDC: *Arbortext Editor Online Hilfe*, 2007.

[Sed02] SEDGEWICK, ROBERT: *Algorithmen*. Addison-Wesley, Reading, 2nd Auflage, 2002.

[SZ90] SHASHA, DENNIS und KAIZHONG ZHANG: *Fast Algorithms for the Unit Cost Editing Distance Between Trees*. J. Algorithms, 11:581–621, 1990.

[W3C06a] W3C: *Extensible Markup Language (XML) 1.1 (Second Edition) W3C Recommendation 16 August 2006, edited in place 29 September*, 2006. http://www.w3.org/TR/2006/REC-xml11-20060816/ Stand: 12.10.2007.

[W3C06b] W3C: *Namespaces in XML 1.0 (Second Edition)*, 2006. http://www.w3.org/TR/REC-xml-names/ Stand: 21.02.2008.

[W3C07] W3C: *World Wide Web Consortium*. Webseite, 2007. http://www.w3.org Stand: 5.10.2007.

[WF04] WHITAKER, NIGEL und ROBIN LA FONTAINE: *A Generalized Grammar for Three-way XML Synchronization*, 2004. http://www.idealliance.org/proceedings/xml05/ship/102/Paper-v8.PDF Stand: 12.10.2007.

[WN04] WHITAKER, NIGEL und THOMAS NICHOLS: *Powering Pipelines with JAXP*, 2004. http://www.idealliance.org/proceedings/xml04/papers/120/pipe-paper.pdf Stand: 12.10.2007.

[ZS89] ZHANG, K. und D. SHASHA: *Simple fast algorithms for the editing distance between trees and related problems*. SIAM J. Comput., 18(6):1245–1262, 1989.

[ZSS92] ZHANG, KAIZHONG, RICHARD STATMAN und DENNIS SHASHA: *On the Editing Distance Between Unordered Labeled Trees*. 42:133–139, 1992.

A. Merge Cases

Auf der folgende Seite sind die Merge Cases für den Vergleich der Tools DeltaXML sync und 3DM Tool aufgeführt. Aus Gründen der Übersichtlichkeit wurden der Namespace aus dem Wurzelelement beim Ergebnis von DeltaXML sync entfernt. Das Wurzelelement würde korrekt:

```
<R xmlns:deltaxml="http://www.deltaxml.com/ns/well-formed-delta-v1"
  xmlns:dxu="http://www.deltaxml.com/ns/unified-delta-v1">
```

geschrieben.

Die Merge Cases sind aus [Lin01, S. 177ff.]. Die Merge Cases sind mit einem Buchstaben und einer Nummer bezeichnet. Der Buchstabe steht für die Operationen: I=insert, D=delete, M=move, C=copy, X=conflict und A=any (verschiedenen Kombinationen aus den anderen Operationen).

Für jeden Mergecase wurde ein entsprechender Testfall angelegt. Dieser Testfall besteht immer aus vier XML-Dokumenten, Base, Derivat1, Derivat2 und Merge. Beide Tools mussten alle Testfälle durchlaufen. Dazu wurde ein eigenes Test-Projekt angelegt.

Case	Base	Derivat1	Derivat2	3DM Merge	Delta Merge
I1	<R> <a> <c/> <d/> <e/> </R>	<R> <a> <i1/> <c/> <d/> <e/> </R>	<R> <a> <c/> <d/> <i2/> <e/> </R>	<R> <a> <i1/> <c/> <d/> <i2/> <e/> </R>	<R> <a> <i1/> <c/> <d/> <i2/> <e/> </R>
I2	<R> <a/> </R>	<R> <a/> <i1/> </R>	<R> <a/> <i2/> </R>	<R> <a/> <i1/> <i2/> </R>	<R> <a/> <i1/> <i2/> </R>
	Konfliktfall, der nicht erkannt wird. In welcher Reihenfolge sollen i1 und i2 auftreten?				
I3	<R> <a/> <c/> </R>	<R> <a/> <i1/> <c/> </R>	<R> <a/> <c/> <i2/> </R>	<R> <a/> <i1/> <c/> <i2/> </R>	<R> <a/> <i1/> <c/> <i2/> </R>
D1	<R> <s1> <p1/> <p2/> </s1> <s2> <p3/> <p4/> </s2> </R>	<R> <s1> <p1/> <p2/> </s1> </R>	<R> <s1> <p1/> </s1> <s2> <p3/> <p4/> </s2> </R>	<R> <s1> <p1/> </s1> </R>	<R> <s1> <p1/> </s1> </R>
D2	<R> <a/> </R>	<R> <a/> </R>	<R> <a/> </R>	<R> <a/> </R>	<R> <a/> </R>
D3	<R> <a/> <c/> <d/> </R>	<R> <a/> <d/> </R>	<R> <c/> <d/> </R>	<R> <d/> </R>	<R> <d/> </R>

D4	<R> <a> <c/> </R>	<R> <a> <c/> </R>	<R> <a> <c/> </R>	<R> <a> <c/> </R>	<R> <a> <c/> <c/> </R>
F1	<R> <a/> <c/> <d/> </R>	<R> <a/> <d/> </R>	<R> </R>	<R> </R>	<R> </R>
F2	<R> <a/> <c/> </R>	<R> <a/> <i1/> <i2/> <c/> <a/> <c/> </R>	<R> <a/> <i1/> <i2/> <c/> <a/> <c/> </R>	<R> <a/> <i1/> <i2/> <c/> <a/> <c/> </R>	<R> <a/> <i1/> <i2/> <c/> <a/> <c/> </R>
C1	<R> <a/> <c/> </R>	<R> <a> <c/> </R>	<R> <a/> <c> </c> </R>	<R> <a> <c> </c> </R>	<R> <a> <c> </c> </R>
C2	<R> <a/> <c/> <d/> </R>	<R> <a/> <a/> <c/> <d/> </R>	<R> <a/> <c/> <c/> </R>	<R> <a/> <a/> <c/> <c/> </R>	<R> <a/> <a/> <c/> <c/> </R>
C3	<R> <a/> 	<R> <a/> <a/>	<R> <a/> 	<R> <a/> <a/>	<R> <a/> <a/>

	<c/> </R>	 <c/> </R>	<c/> <c/> </R>	 <c/> <c/> </R>	 <c/> <c/> </R>
C4	<R> <a/> <c/> </R>	<R> <a> <c/> </R>	<R> <a> <c/> <c/> </R>	<R> <a> <c/> <c/> </R>	<R> <a> <c/> <c/> </R>
C5	<R> <a/> <c/> </R>	<R> <a/> <a/> <c/> </R>	<R> <a/> <c/> <a/> </R>	<R> <a/> <a/> <c/> <a/> </R>	<R> <a/> <a/> <c/> <a/> </R>
M1	<R> <a> <c/> <d/> <e> <f/> </e> </R>	<R> <a> <d/> <c/> <e> <f/> </e> </R>	<R> <a> <c/> <d/> <f> <e/> </f> </R>	<R> <a> <d/> <c/> <f> <e/> </f> </R>	<R> <a> <d/> <c/> <f> <e/> </f> </R>
M 2	<R> <p1/> <p2/> <p3/> <p4/> <p5/> </R>	<R> <p2/> <p1/> <p3/> <p4/> <p5/> </R>	<R> <p1/> <p2/> <p3/> <p5/> <p4/> </R>	<R> <p2/> <p1/> <p3/> <p5/> <p4/> </R>	<R> <p2/> <p1/> <p3/> <p5/> <p4/> </R>
M4	<R> <a/> <c/> </R>	<R> <a/> <c/> </R>	<R> <c/> <a/> </R>	<R> <a /> <c /> </R>	<R> <a/> <c/> <a/> </R>
M5	<R> <a/>	<R> 	<R> <a/>	<R> 	<R>

	<b/> <c/> <d/> <e/> </R>	<c/> <d/> <e/> <a/> </R>	<b/> <d/> <c/> <e/> </R>	<d/> <c/> <e/> <a/> </R>	<d/> <c/> <e/> <a/> </R>
U1	<R> <a/> <b/> </R>	<R> <a1/> <b/> </R>	<R> <a/> <b1/> </R>	<R> <a1/> <b1/> </R>	<R> <a1/> <b1/> </R>
U2	<R> <a/> <b/> </R>	<R> <a/> <b1/> </R>	<R> <a/> <b1/> </R>	<R> <a/> <b1/> </R>	<R> <a/> <b1/> </R>
U3	<R> <a/> <b/> </R>	<R> <a1/> <b/> </R>	<R> <a2/> <b/> </R>	<R> <a1/> <b/> </R>	<R> <a1/> <a2/> <b/> </R>
X1	<R> <a/> <b/> </R>	<R> <a/> <b2/> </R>	<R> <a/> <b1/> </R>	<R> <a/> <b2/> </R>	<R> <a/> <b2/> <b1/> </R>
X2	<R> <a/> </R>	<R> <a/> <a/> </R>	<R> </R>	<R> <a/> </R>	<R> <a/> </R>
	Source of copy is deleted				
X3	<R> <a/> <b/> </R>	<R> <a/> <a/> </R>	<R> <a/> </R>	<R> <a/> <a/> </R>	<R> <a/> <a/> </R>
	There is no conflict here. The case is mislabeled.				
X4	<R> <a/> <b/> <c/> </R>	<R> <b/> <a/> <c/> </R>	<R> <a/> <c/> <b/> </R>	<R> <b/> <a/> <c/> </R>	<R> <b/> <a/> <c/> <b/> </R>
A1	<R> <a/> <b/> </R>	<R> <a/> <b1/> </R>	<R> <a/> <b/> <c/> </R>	<R> <a/> <b1/> <c/> </R>	<R> <a/> <b1/> <c/> </R>
A2	<R> <a/>	<R> <a/>	<R> <a1/>	<R> <a1/>	<R> <a1/>

	 <c/> </R>	<c/> </R>	 <c/> </R>	<c/> </R>	<c/> </R>
A3	<R> <s1> <p1/> </s1> <s2> <p2/> </s2> <s3> <p3/> </s3> </R>	<R> <s1> <p1/> </s1> <s2> <p2/> </s2> <s3> <p3/> </s3> <s1> <p1/> </s1> </R>	<R> <s2> <p2/> </s2> <s1> <p1/> </s1> <s3> <p3/> </s3> </R>	<R> <s2> <p2 /> </s2> <s1> <p1 /> </s1> <s3> <p3 /> </s3> <s1> <p1 /> </s1> </R>	<R> <s2> <p2/> </s2> <s1> <p1/> </s1> <s3> <p3/> </s3> </R>
A4	<R> <a> <c /> <d> <e /> <f /> </d> </R>	<R> <a> <c/> <d> <e/> <f/> </d> </R>	<R> <a> <c/> <a> <d> <e/> <f/> </d> </R>	<R> <a> <c/> <a> <d> <e/> <f/> </d> </R>	<R> <a> <c/> <a> <d> <e/> <f/> </d> </R>
A5	<R> <a/> </R>	<R> <a1/> </R>	<R> <a/> <a/> </R>	<R> <a1/> <a1/> </R>	<R> <a1/> <a/> </R>
	Hier erkennt man die unterschiedliche Handhabung, wenn ein Update auf eine Konten erfolgt ist, der verschoben wird. (update a ->a' ,move a')				
A6	<R> <a> <c/> </R>	<R> <a> <c/> </R>	<R> <a> <c/> <a> <c/>	<R> <a> <c/> <a> <c/> 	<R> <a> <c/> <a> <c/>

			 </R>	 </R>	 </R>
A7	<R> <a/> <b/> <c/> </R>	<R> <a/> <a/> <c/> </R>	<R> <a/> <c/> <b/> </R>	<R> <a/> <a/> <c/> </R>	<R> <a/> <a/> <c/> </R>
A8	<R> <a> <b/> <c/> </R>	<R> <a> <b/> <c/> </R>	<R> <a> <b/> <c/> <a> <b/> <c/> </R>	<R> <a> <b/> <a> <b/> <c/> </R>	<R> <a> <b/> <c/> <a> <b/> <c/> </R>
	Wie A5 update -> move				
A9	<R> <a> <b/> <c/> </R>	<R> <a> <b/> <c/> <a> <c/> <b/> </R>	<R> <a> <b/> <c/> <d/> </R>	<R> <a> <b/> <c/> <d/> <a> <c/> <b/> </R>	<R> <a> <b/> <c/> <d/> <a> <c/> <b/> </R>
A10	<R> <l> <a/> </l> <l> <b/> </l> </R>	<R> <l> <b/> </l> <l> <a/> </l> </R>	<R> <l> <b/> </l> <l> <b/> </l> </R>	<R> <l> <b/> </l> <l> <b/> </l> </R>	<R> <l> <b/> </l> <l> <a/> </l> </R>
A11	<R> <a/> <b/> <c/> </R>	<R> <b/> <a/> <c/> </R>	<R> <a/> <b/> <c/> <i/> </R>	<R> <b/> <a/> <c/> <i/> </R>	<R> <b/> <a/> <c/> <i/> </R>
A12	<R> <a/> <b/>	<R> <a/> <a/>	<R> <a/> <b/>	<R> <a/> <a/>	<R> <a/> <a/>

	</R>	 </R>	<i/> </R>	 <i/> </R>	 <i/> </R>
A13	<R> <a/> </R>	<R> <aP/> <bP/> </R>	<R> <a/> </R>	<R> <aP/> <bP/> <bP/> </R>	<R> <aP/> <bP/> </R>
	Wie A5 update -> move				
A14	<R> <a> <d/> <e/> <f/> </R>	<R> <f/> <a> <d/> <e/> <i/> </R>	<R> <a> <e/> <d/> <bP/> </R>	<R> <bP/> <a> <e/> <d/> <i/> </R>	<R> <f/> <a> <e/> <d/> <i/> <bP/> </R>
	Wie A5 update -> move				
A15	<R> <p1/> <p2/> </R>	<R> <p1/> <p2P/> </R>	<R> <p1/> <a> <p2/> </R>	<R> <p1/> <a> <p2P/> </R>	<R> <p1/> <p2P/> <a> <p2/> </R>
A16	<R> <a> <m/> <c> <cc/> </c> </R>	<Rp></Rp>	<R> <c> <cc/> </c> <c> <cc/> </c> <m/> </R>	<Rp> <c> <cc /> </c> <c> <cc /> </c> <m> <i /> </m> </Rp>	
	Fehler in Deltaxml sync Wurzelelment muss gleich sein!				
E1	<R> <a a="1" /> </R>	<R> <a a="5" /> </R>	<R> <a a="1" /> </R>	<R> <a a="5" /> </R>	<R> <a a="5"/> </R>
E2	<R> <a a="1" />	<R> <a a="x" />	<R> <a a="x" />	<R> <a a="x" />	<R> <a a="x" />

	<b b="3" /> </R>	<b b="x" /> </R>	<b b="x" /> </R>	<b b="x" /> </R>	<b b="x" /> </R>
E3	<R> <a a="1" /> <b b="3" /> </R>	<R> <a a="5" / <b b="3" / </R>	<R> <a a="1" / <b /> </R>	<R> <a a="5" /> <b /> </R>	<R> <a a="5" /> <b /> </R>
E4	<R> <a a="1" /> <b b="3" /> </R>	<R> <a a="5" / <b /> </R>	<R> <a a="1" /> <b b="4" /> </R>	<R> <a a="5" /> <b /> </R>	Attribut Konflikt richtig erkannt
E5	<R> <a a="1" /> <b b="3" /> </R>	<R> <a a="5" / <b b="9" / </R>	<R> <a a="1" / <b /> </R>	<R> <a a="5" /> <b b="9" /> </R>	Attribut Konflikt richtig erkannt
E6	<R> <a a="1" /> <b b="3" /> </R>	<R> <a a="5" f="x" /> <b b="3" c="7"/> </R>	<R> <a a="1" f="x" /> <b b="4" c="7"/> </R>	<R> <a a="5" d="x" f="x"/> <b b="4" c="7"/> </R>	<R> <a a="5" d="x" f="x"/> <b b="4" c="7"/> </R>

B. CD-Inhalt

Dieser Bachelorarbeit ist eine CD beigelegt, auf welcher sich der Prototyp, die Testfälle in Form eines eigenen Eclipse-Projekt, der Sourcecode des Prototyps, alle genutzten Internetquellen und Artikel, sowie die Bachelorarbeit als PDF-Datei zu finden sind.

Die Quellcodes, die sich auf der CD befinden sind Eigentum der Lufthansa Systems und daher streng vertraulich zu behandeln. Sie dürfen in keinster Weise für den kommerziellen Einsatz reproduziert oder weitergegeben werden.

Die beigelegte CD hat folgenden Inhalt:

- `/bachelorthesis.pdf`: Die Bachelorarbeit als PDF-Datei
- `/Readme.txt`: Inhalt und Anmerkung zur CD-Rom
- Verzeichnis: `/code`: Sourcecode
  - `dev-3way-merge.zip`: der Sourcecode des Prototyps
  - `dev-testcases.zip`: der Sourcecode des Testprojekts und die Testfälle der Diff- und Mergecases
- Verzeichnis: `/resources`: Frei verfügbare oder für wissenschaftliche Zwecke veröffentlichbare Quellen aus dem Literaturverzeichnis, jeweils in einem eigenen Unterverzeichnis, benannt nach Kürzel der entsprechenden Quelle

Glossar

3-Wege-Merge Eine Originalversion einer Datei (Base) und zwei verschiedener Versionen (Derivate) des Originals werden zu einer neuen Datei zusammengeführt. Die Änderungen beider Derivate zum Original sind in der neue Datei (Merge) enthalten.

Air Transport Association - ATA Die Air Transport Association of America, Inc., kurz ATA ist ein Vereinigung von Fluggesellschaften. Die ATA beschäftigt sich neben Sicherheitsfragen auch mit der Standardisierung rund um die Luftfahrt. Es ist also auch zu klären, ob die Dokumentarten wie das AMM durch die ATA spezifiziert sind und welche Modelle diese Spezifikation für die Erstellung und den Austausch der Wartungsdokumente vorsieht.

Aircraft Maintenance Manual - AMM Das Aircraft Maintenance Manual ist das Wartungshandbuch eines Flugzeuges und dient als Grundlage für sämtliche Wartungsarbeiten und daraus resultierenden Arbeitskarten. Die entsprechenden Tasks (Arbeitsanweisungen) sind nach dem Unterstützungssystem Aircraft Maintenance Task Oriented Support System (AMTOSS) kategorisiert.

Arbeitskarte (engl. job card) Die Arbeitskarte (engl. job card) dient der Beschreibung einer Wartungsarbeit, Inspektion oder Instandsetzung.

ATA iSpec 2200 Die ATA Spezifikation ist eine Spezifikation, welche den Austausch und Inhalt von Wartungsdokumenten in der Luftfahrt standardisiert.

Dokumentanker Ein Dokumentanker ist ein Strukturelement nach ATA iSpec 2200.

DTD Document Type Definition, definiert die Grammatik einer SGML-/XML-Datei.

Entwurfsmuster Entwurfsmuster [GHJV95, ABW97] bieten eine einfache und elegante Lösung für spezielle Probleme des objektorientierten Softwareentwurfs. Sie erfassen Problemlösungen, die im Laufe der Zeit gefunden wurden, und sich weiterentwickelt haben. Entwurfsmuster verfolgen neben dem Problemlösungsvorschlag noch das Ziel, die Software möglichst flexibel und erweiterbar zu entwerfen bzw. umzusetzen.

fachliche Dokumente Als fachliche Dokumente werden AMM, AMM-Task, Arbeitskarte (engl. job card), Temporary Revision und Service Bulletin bezeichnet

Konflikt Ein Konflikt kann beim Zusammenführen von Dokumenten auftreten, wenn sich Änderungen überschneiden oder die Eindeutigkeit der Reihenfolge verloren geht.

konkrete Fabrik-Muster Eine konkrete Fabrik ist ein Teil des Musters *Abstrakte Fabrik* (abstract factory), welches eine Schnittstelle für die Erzeugung von Familien verwandter oder voneinander abhängiger Objekte bietet, ohne ihre konkrete Klasse zu benennen. Die konkrete Fabrik stellt Methoden zur Erzeugung von Objekten einer Klassenfamilie zur Verfügung [GHJV95, ABW97].

linear Als linear bezeichnet man Elemente in einer geordneten Liste, beispielsweise Zeilen in einem Textdokument.

LSY Lufthansa Systems AG

Namespaces[W3C06b] Namespaces[W3C06b] dienen der Kombination von verschiedenen XML-Anwendungen innerhalb einer Dokumentinstanz

Revisionsmarkup Als Revisionsmarkup werden Änderungsinformationen im Dokument bezeichnet

SAP DVS Das Dokumenten-Verwaltungs-System ist eine SAP-Anwendung, in welcher Dokumentinfosätze (Metadaten und Inhalt) der Dokumente verwaltet werden

SAP MRO Luftfahrzeug-Instandhaltung (Maintenance, Repair and Overhaul) umfasst die Wartung, Inspektion und Instandsetzung von Flugzeugen. Diese ist behördlich streng geregelt.

Service Bulletin - SB Das Service Bulletin ist eine technische Mitteilung über besondere Modifikationen am Flugzeug.

Simple API for XML - SAX SAX ist eine Klassenbibliothek die das Parsen von XML-Dokumenten anbietet. Die Verarbeitung findet sequenziell auf dem XML-Datenstrom statt. SAX ist zudem noch Event-basiert, was bedeutet, dass durch die Erzeugungen und das Empfangen von Events eine Verarbeitung des XML-Datenstroms möglich ist.

Singelton-Muster Das Singelton-Muster (singelton-pattern) stellt sicher, dass es von einer Klasse nur eine Instanz gibt, welche meist global verfügbar ist [GHJV95, ABW97].

strukturorientiert Als strukturorientiert bezeichnet man Elemente, die in einer Baumstruktur abgebildet werden können, beispielsweise Elemente in einem XML-Dokument.

Temporary Revision - TR Temporary Revisions beziehen sich auf Änderungen am AMM, die zwischen den Revisionszyklen erscheinen und deren Dringlichkeit es erfordert, die Änderungen den Betreibern noch vor Auslieferung der neuen Revision mitzuteilen.

Unified Modelling Language (UML) Die Unified Modelling Language (UML) ist eine Modellierungssprache. Es handelt sich um eine (überwiegend grafische) Notation, die dazu verwendet wird, die unterschiedlichen Aspekte eines Software-Entwurfs auszudrücken.

W3C World Wide Web Consortium (`http://www.w3.org`)

Whitespace Whitespaces sind Zeichen wie Leerzeichen, Tabulatorzeichen und Zeilenumbruchzeichen wie das Zeilenvorschubszeichen und das Wagenrücklaufzeichen.

XML Extensible Markup Language, Sprache zur Beschreibung strukturierter Dokumente

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 25. Februar 2008	
Ort, Datum	Unterschrift