Versionierungskonzepte mit Unterstützung für Differenz- und Mischwerkzeuge

Vom Fachbereich Elektrotechnik und Informatik der Universität Siegen

zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften (Dr. rer. nat.)

genehmigte Dissertation

von

Dipl.-Ing. Dirk Ohst

1. Gutachter: **Prof. Dr. Udo Kelter**
2. Gutachter: **Prof. Dr. Andreas Henrich**
Vorsitzender: **Prof. Dr. Rainer Brück**

Tag der mündlichen Prüfung: 07.09.2004

URN:

urn:nbn:de:hbz:467-831

Kurzfassung

Softwarekonfigurations-Management (SKM) ist ein wichtiger Bestandteil moderner Softwareentwicklungsprozesse. Ein Vorteil des Einsatzes von SKM-Werkzeugen ist die Möglichkeit Versionen zu erzeugen, Unterschiede festzustellen und Produkte zu veröffentlichen, die eine Konfiguration von bestimmten Dokumenten darstellen. Es ist eine große Anzahl von SKM-Systemen und Konzepten verfügbar, jedoch arbeiten die meisten von ihnen (einschl. CVS, RCS oder SCCS) fast ausschließlich auf Textdateien. Deshalb ist der Einsatz von SCM-Werkzeugen in den späten Phasen der Softwareentwicklung, insbesondere bei der Programmierung und Wartung, etabliert und gängige Praxis. SKM hat in den frühen Phasen (z. B. Analyse und Entwurf) noch nicht diese Bedeutung erlangt. Existierende SKM-Systeme sind weniger gut geeignet, um Dokumente der frühen Phasen zu versionieren, Unterschiede zu bestimmen oder um diese zu visualisieren. Der Grund hierfür liegt darin, daß es sich bei diesen Dokumenten üblicherweise nicht um Text, sondern um Diagramme handelt, die Bestandteil von Modellierungssprachen sind, wie z. B. UML.

Der erste Teil dieser Arbeit beschäftigt sich mit der Aufgabenstellung der Versionierung von Diagrammen. Das vorgeschlagene Versionierungskonzept basiert auf dem Einsatz eines Objektmanagement-Systems (OMS) und der Nutzung einer OMS-orientierten Werkzeugarchitektur. Die Werkzeugtransaktionen des OMS sind die Basis für das vorgestellte Versionierungskonzept. Alle modifizierten Objekte werden automatisch innerhalb der Werkzeugtransaktionen versioniert. Einzelne Objektversionen faßt eine Konfiguration zusammen, die eine Dokumentversion repräsentiert. Entwurfstransaktionen dienen zur Verwaltung der Dokumentversionen.

Dieses Versionierungskonzept erlaubt eine andere Form der Gruppenarbeit im Vergleich zu der Nutzung von z. B. CVS oder RCS. Alle Entwickler arbeiten auf denselben Daten. Änderungen sind sofort für alle Entwickler sichtbar, die an der selben Version arbeiten. Die Konsistenz wird durch Transaktionssperren auf Objektversionen sichergestellt.

Der zweite Teil dieser Arbeit beschäftigt sich mit der Visualisierung der Differenzen zwischen zwei Diagrammen im Fall der kooperativen Arbeit. Die Berechnung der Unterschiede zwischen den Versionen stützt sich auf die eindeutigen Objektidentifizierer ab, die durch das OMS vergeben werden. Das vorgeschlagene Visualisierungskonzept ist nur abhängig von der verwendeten Modellierungssprache, jedoch nicht vom verwendeten Versionsverwaltungs-System. Zur Visualisierung wird ein so genanntes *Vereinigungsdiagramm* verwendet, welches die gemeinsamen Elemente der beiden zu vergleichenden Diagramme und die spezifischen Diagrammelemente enthält. Diese werden hierin farbig hervorgehoben.

Das Versionierungskonzept und die Visualisierung sind prototypisch im OMS H-PCTE und in der Werkzeugsammlung PISET implementiert. PISET bietet Editoren für UML Kollaborations-, Objekt-, Anwendungsfall-, Klassen- und Zustandsdiagramme. Die Visualisierung für Klassendiagramme ist ebenfalls im CASE-Werkzeug FUJABA implementiert.

Abstract

Software configuration management (SCM) is an indispensable part of high-quality software development processes. An advantage of using SCM systems is that one can create versions of a document, detect the differences between them and release systems as configuration of certain document versions. A large number of SCM systems and concepts are available, however most of them (incl. systems such as RCS, CVS and SCCS) only work with text files. Therefore SCM is a well established and common practise in the later phases of software development processes, notably during programming and maintenance. SCM is a less common practise during the early phases, i.e. analysis and design. Existing SCM systems are not well suited for the versioning, detection and visualisation of differences between documents in the early phases, because those documents are not text, but diagrams usually part of modelling languages such as the UML.

The first part of this thesis addresses the problem of versioning diagrams. The proposed versioning system is based an object management system (OMS) to store all diagrams and on an OMS-oriented tool architecture. The tool transactions of the OMS are the basis of the proposed versioning concept. All modified objects are automatically versioned inside the tool transactions. Single object versions are combined in a configuration, which forms one version of an entire document in the OMS. Design transactions provide the required versions of the documents to the tool transactions.

This versioning concept offers a different kind of team cooperation compared to the use of RCS, CVS, etc. All developers work on the same data. Thus modifications are visible to all developers working on the same document version and do not affect others. The consistency is ensured by transaction locks on object versions.

The second part of this thesis addresses the visualisation of diagram differences when working cooperatively. The computation of the differences is based on the object identifiers offered by the OMS. The proposed visualisation of the differences is independent from the used version management system but depend on the modelling language and its syntax elements. The differences are visualised by using an so called *unified diagrams*. This kind of diagram includes the common and the specific parts of the compared diagrams. The specific parts are highlighted using different colours.

The versioning concept and the visualisation has been prototypically implemented on the OMS H-PCTE and the tool set PISET. PISET supports Editors for UML collaboration, object, use case, class and statechart diagrams . The visualisation of UML class diagrams have also been implemented in the Fujaba CASE tool.

Vorwort

Die vorliegende Dissertation ist in Rahmen meiner Tätigkeit als wissenschaftlicher Mitarbeiter der Fachgruppe für Praktische Informatik an der Universität Siegen entstanden.

An dieser Stelle möchte ich mich bei all den Personen bedanken, die durch ihre Unterstützung zum Gelingen dieser Arbeit mit beigetragen haben. Zuerst gilt mein Dank Herrn Prof Dr. Udo Kelter, der als Erstgutachter und Leiter der Fachgruppe die Betreuung dieser Arbeit übernommen hat. Die angenehme Arbeitsatmosphäre sowie die guten Rahmenbedingungen haben einen nennenswerten Beitrag zu dieser Arbeit geleistet. Weiterhin gilt mein Dank Herrn Prof. Dr. Andreas Henrich für die Übernahme des Zweitgutachtens.

Allen meinen (ehemaligen) Kollegen möchte ich an dieser Stelle für die vielen anregenden Diskussionen danken, die mit ihren Anregungen ihrerseits einen Beitrag geleistet haben. Insbesondere gilt mein Dank Herrn Dr. Dirk Platz und Herrn Dr. Marc Monecke, deren Arbeiten als Ausgangspunkt und als Testumgebung für die Realisierung der in dieser Arbeit entwickelten Konzepte dienten. Weiterhin gilt mein Dank Herrn Dipl.-Ing. Michael Welle, der durch die Implementierung und Validierung des beschriebenen Konzepts für Differenz- und Mischwerkzeuge einen Beitrag geleistet hat.

Nicht zuletzt geht mein herzlichster Dank an meine Eltern, die mich auf dem gesamten Weg bis zum Abschluß dieser Arbeit auf vielfältige Weise unterstützt haben. Auch möchte ich meinen Freunden Anja, Susanne und Bruno danken, die mich motiviert haben, diese Arbeit abzuschließen.

Hennef, im Oktober 2004

Dirk Ohst

Die anderen kennen ist Weisheit.
Den anderen seinen Willen aufzwingen ist Stärke.
Ihn sich selbst aufzwingen ist größere Stärke.
Laotse

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung und Motivation

1.1 Einleitung

Unter dem Begriff Software versteht man nicht nur die ausführbaren Programme oder den dazugehörigen Quelltext, sondern alle Dokumente, die während der gesamten Softwareentwicklung entstanden sind, wie z. B. Analyse- oder Entwurfsdokumente [34]. Der Werkzeugeinsatz, ohne den die moderne Softwareentwicklung nicht mehr denkbar ist, erleichtert einerseits die Erstellung der diversen Arten von Software-Dokumenten, andererseits führt er jedoch auch zu einem erhöhten Verwaltungsaufwand durch die steigende Anzahl an Dokumenten und Versionen dieser Dokumente. Diese Tendenz zu mehr Dokumenten und Versionen wird noch verstärkt durch die Einführung von Softwareentwicklungsprozessen [9, 90, 184] wie z. B. den Rational-Unified-Process [102, 208], die ISO 9001 [49, 108, 113] oder das Capability Maturity Model [10, 177, 185]. Diese definieren einen festgelegten Ablauf bei der Softwareentwicklung und Phasen, die die Dokumente zu durchlaufen haben. Das Ziel ist ein hohes Qualitätsniveau für das fertige Produkt. Desweiteren wird Software nicht mehr von einem einzelnen Entwickler geschrieben, sondern in Gruppen, die nicht selten 30 oder mehr Entwickler umfassen [186, 77]. Diese parallele Entwicklung von Software erhöht die Anzahl an erstellten Dokumentversionen, die z.T. gleichzeitig existieren. Mögliche Gründe für die Existenz von parallelen Versionen, die man als *Varianten* bezeichnet, sind:

1. die Wartung einer Version eines Software-Produktes und die parallele Neuentwicklung der Nachfolger-Version

2. die Anpassung eines Produktes an unterschiedliche Kundenanforderungen

3. konkurrierende Arbeit an einer Version eines Dokuments durch mehrere Entwickler zur Vermeidung von Wartezeiten bei notwendigen Änderungen[1]

Besonders bei der konkurrierenden Arbeit an einer Dokumentversion, müssen die Varianten wieder zusammengeführt werden. Aus dem beschriebenen Szenario folgt die Aufgabe, nicht nur die erstellten Dokumente und deren Versionen zu archivieren, sondern auch noch derart zu verwalten, daß zusammengehörende Versionen unterschiedlicher Dokumente rekonstruiert und bearbeitet werden können, einschließlich dem Anzeigen von Differenzen und dem Mischen von Dokumenten.

[1] Das gilt unabhängig von der Art, wie das Versionsmanagement-System die Zugriffe synchronisiert: beim pessimistischen Verfahren werden die Versionen explizit durch die Anwender angelegt und beim optimistischen Verfahren implizit durch das System selbst.

Es wurden bereits viele Lösungen für textuelle Dokumente entwickelt oder neue Ansätze vorgeschlagen, einen Überblick gibt [58]. Die einfachen Versionsmanagement-Systeme, wie z. B. SCCS [195], RCS [221] oder CVS [24,36] dienen primär der Verwaltung von Versionen einzelner Dateien und bieten nur wenige Möglichkeiten, zusammengehörige Versionen, die auch als *Konfigurationen* bezeichnet werden, zu verwalten. Neben diesen einfachen Versionsmanagement-Systemen wurden *Software-Konfigurationsmanagement-Systeme* (SKM-Systeme) für unterschiedliche Aufgaben und Anwendungsschwerpunkte entwickelt [58]. Beispiele sind Adele [78, 79], welches die Versionsverwaltung und den Softwareentwicklungs-Prozeß zusammenführt, oder ClearCase [143, 236], welches eine Unterstützung für unterschiedliche Dokumenttypen und Konfigurationen bietet.

Neben den rein textuellen Dokumenten, wie Quelltext, Spezifikationen oder textuell beschriebene Anwendungsfälle, gibt es auch zunehmend graphische Dokumente, die z. B. in der *Unified Modelling Language* (UML) [181] notiert sind. Beispiele hierfür sind Klassendiagramme, Aktivitätsdiagramme oder Interaktionsdiagramme. Diese Dokumenttypen bezeichnen wir im weiteren als Diagramme. Die Versionierung der Diagramme wird durch die vorhandenen *Versionsmanagement-Systeme* (VM-Systeme) nur unzureichend unterstützt [175].

Der entscheidende Grund hierfür ist, daß sich die (persistente) textuelle Repräsentation von UML-Diagrammen erheblich von deren (logischer & visueller) Darstellung in den CASE-Werkzeugen unterscheidet [172]. Im Gegensatz hierzu sind die einzelnen Repräsentationen von Textdokumenten sehr ähnlich, unabhängig davon, ob sie durch Editoren angezeigt werden, oder in einer Datei gespeichert sind. Die Darstellung in den Editoren unterscheidet sich nur durch Hervorhebungen (z. B. Schlüsselworte einer Programmiersprache) oder Einrückungen von der physischen Darstellung in einer Datei. Aufgrund der unterschiedlichen Repräsentationen von UML-Diagrammen versagen auch konventionelle VM-Systeme bei der Anzeige von Differenzen oder dem Mischen von Diagrammversionen.

Viele CASE-Werkzeuge bieten keine oder nur eine unzureichende Unterstützung bei der Versionierung oder für kooperative Arbeit [205,175] an. Diese besteht in den wenigen existierenden Fällen darin, daß die Repräsentation des Diagramms auf dem persistenten Speicher durch ein konventionelles VM-System versioniert wird. Den Zeitpunkt, wann z. B. eine neue Version angelegt wird, kann der Entwickler im CASE-Werkzeug bestimmen. Die Unterstützung der kooperativen Arbeit beschränkt sich in diesen Fällen auf die durch das VM-System angebotenen Funktionen.

Häufig sind die CASE-Werkzeuge in Werkzeugsammlungen, den *Softwareentwicklungsumgebungen* (SE-Umgebungen) [120], integriert. Der Einsatz von mehrbenutzerfähigen SE-Umgebungen kann die kooperative Arbeit unterstützen. Ein Konzept hierzu stellt Platz [187] vor, welches jedoch keine Versionierung berücksichtigt. Das Konzept basiert direkt auf dem Ansatz des Integrationsrahmens (Frameworks) [120]. Die Idee des Integrationsrahmens ist die Bereitstellung von zentralen Diensten, wie z. B. der Datenhaltung. Diese Dienste können durch die Werkzeuge genutzt werden und so die Werkzeugentwicklung vereinfachen. Monecke [167] beschreibt einen Werkzeugkonstruktionsansatz, der darauf basiert.

Die dieser Arbeit zugrundeliegende Idee ist die Entwicklung einer Architektur für eine SE-Umgebung zur Versionierung der gespeicherten Daten mit Unterstützung für die Differenzberechnung, -anzeige und das Mischen von Versionen. Die Ziele sind eine auf die Dokumenttypen optimierte Versionierung, die sich einerseits leicht in CASE-Werkzeugen einsetzen läßt, aber auch für die Anwender der CASE-Werkzeuge leicht zu bedienen ist.

1.2 Versionsverwaltung

Die Entwicklung eines Versionsmanagement-Systems für Software-Dokumente der frühen Phasen setzt ein Verständnis der besonderen Eigenschaften dieser Dokumente voraus. Um die existierenden Konzepte zur Versionsverwaltung besser beurteilen zu können, analysieren wir zu Beginn unterschiedliche Dokumenttypen und Versionsverwaltungskonzepte.

1.2.1 Dokumenttypen und deren Versionierung

Versionsmanagement-Systeme (VM-Systeme) gibt es für unterschiedliche Dokumenttypen. Jeder dieser Typen hat spezifische Eigenschaften, die bei dessen Versionierung und später beim Mischen von Versionen oder der Anzeige von Differenzen berücksichtigt werden sollten, um ein möglichst „gutes“ Ergebnis zu erhalten. Man kann folgende Dokumenttypen unterscheiden:

- Text-Dokumente: z. B. Texte ohne Strukturinformationen
- strukturierte Text-Dokumente: z. B. LaTeX, XML, HTML oder Quelltexte einer Programmiersprache
- strukturierte graphische Dokumente: z. B. Diagramme oder CAD-Zeichnungen
- komplexe zusammengesetzte Dokumente: z. B. Dokumente des Component Based Development

Die unstrukturierten Text-Dokumente unterscheiden sich von den strukturierten Texten darin, daß letztere explizite Angaben zur logischen Struktur beinhalten, z. B. in Form von bestimmten Schlüsselwörtern. Erstere dagegen können zwar auch eine Struktur besitzen, die jedoch nicht explizit gekennzeichnet ist und die nur der Leser anhand der Semantik oder des Layouts bestimmen kann. Diese beiden Dokumenttypen sind die traditionellen Gebiete der VM-Systeme, wofür es eine große Anzahl von Systemen gibt: z. B. SCCS [195, 194], RCS [219] oder CVS [36, 24]. Eine wichtige Eigenschaft dieser Dokumente ist, daß ein Entwickler sie manuell erstellt und sie daher selten vollständig umstrukturiert werden. Auf dieser Eigenschaft basieren die VM-Systeme, um die einzelnen Versionen der Dokumente kompakt, d.h. nur die einzelnen Differenzen zwischen den Versionen, die man als Delta[2] bezeichnet, zu speichern, aber auch um die Differenzen anzuzeigen oder um sie zu mischen. Diese Systeme können auch andere Dokumenttypen verwalten, jedoch mit der Einschränkung, daß die Versionen als Ganzes gespeichert werden und die Anzeige von Differenzen oder das automatische Mischen von Versionen nicht möglich ist. Ein weiterer Nachteil dieser einfachen Versionsmanagement-Systeme ist, daß sie nicht oder nur mit Einschränkungen die Konsistenz von einzelnen Versionen unterschiedlicher Dokumente sicherstellen können. Dies ist jedoch notwendig, da in einem größeren Projekt mehrere Dokumente existieren, die zueinander konsistent sein müssen. Dabei kann es sich z. B. um Quelltext oder um Spezifikationen handeln.
Softwarekonfigurations-Managementsysteme (SKM-Systeme) wie z. B. ClearCase [143] beherrschen diese Disziplin, und mit der Unterstützung weiterer Werkzeuge beherrschen sie auch die Anzeige von Differenzen mit anschließendem Mischen von Dokumentversionen, die in proprietären Text- oder binären Formaten (z. B. RTF oder UML-Modelle aus Rational Rose) gespeichert sind [236]. Die VM-Systeme und SKM-Systeme sind nicht auf bestimmte Typen von strukturierten Texten festgelegt, so daß sie deren Struktur nicht berücksichtigen. Dieser Thematik wird von speziellen Werkzeugen Rechnung getragen, wie z. B. von CoEd [19, 20],

[2] Eine Definition der Begriffe Differenz und Delta folgt in Abschnitt 1.2.4

spezialisiert auf die kooperative Arbeit an LaTeX-Dokumenten, Historian [3], eine Erweiterung des XEmacs zur Versionierung von C, Lisp, HTML, und UNIX Troff oder den verschiedenen Vorschlägen zur Versionierung von Hypertextdokumenten [222, 28, 37]. Desweiteren gibt es spezielle Lösungen zur Differenzanzeige und zum Mischen von Versionen von XML- oder LaTeX-Dokumenten [227, 50, 41, 148, 158, 40]. Die Grundannahme, die diesen speziellen Werkzeugen zu Grunde liegt ist, daß die Qualität der Mischergebnisse mit der Menge der berücksichtigten Dokument-Eigenschaften steigt.

Ein weiterer Dokumenttyp sind graphische Dokumente, z. B. Diagramme, wie sie durch die UML definiert sind. Diese diskutieren wir in Abschnitt 1.2.2.

Des weiteren wurden VM-Systeme für bestimmte Anwendungsgebiete vorgestellt, wie z. B. für das Component Based Development (CBD) [160], das CSCW, z. B. [145, 60] oder CAD [117, 77] und Engineering Data Management (EDM) [4, 235]. Der Schwerpunkt bei der Versionierung von Dokumenten des CDB liegt nicht so sehr auf einzelnen Dokumenten, sondern mehr auf deren Zusammenstellung. Diese Konzepte zur Versionierung betrachten wir im folgendem nicht weiter, da für diese Anwendungsgebiete andere Voraussetzungen und Annahmen gelten, z. B. sollen im CBD Produkte aus Komponenten gebaut werden oder beim CAD können in einem Dokument mehrere Versionen eines komplexen Objektes enthalten sein.

1.2.2 Modellierung und Versionierung von UML-Dokumenten

UML-Diagramme besitzen andere Eigenschaften als Textdokumente. Im folgenden betrachten wir UML-Diagramme, um die Anforderungen an ein VM-System für diesen Dokumenttyp angeben und existierende VM-Systeme dahingehend bewerten zu können.

1.2.2.1 Modellierung von UML-Diagrammen

Meta–Metamodell	Die Infrastruktur für ein Metamodell. Definiert die Sprache, um Metamodell zu spezifizieren.	Meta Object Facility (MOF)
Metamodell	Eine Instanz eines Meta–Metamodells. Definiert die Sprache, um ein Modell zu spezifizieren.	z.B. UML mit Klassen, Methoden oder Attributen
Modell	Eine Instanz eines Metamodells. Definiert eine Sprache zur Beschreibung von Informationen.	z.B. DB–Schmea zur Verwaltung von Konten
Nutzdaten	Eine Instanz eines Modells. Definiert Daten eines bestimmen Anwendungsfalles.	z.B. die verwalteten Konten

Abbildung 1.1: Metadaten-Architektur der UML

Die Spezifikation der UML ist in eine 4 Schichten Metadaten-Architektur eingebettet (siehe Abbildung 1.1). Die unterste Schicht, die *Meta Object Facility* (MOF) [178], definiert die Sprache, in der die UML definiert ist. Die MOF bezeichnet man als ein Meta-Metamodell. Neben der UML können mit Hilfe der MOF weitere Modellierungssprachen oder auch Datentyp-Abbildungen zwischen der IDL [112] und einzelnen Programmiersprachen beschrieben werden. Die Modellelemente [181] der UML sind in der zweiten Schicht definiert. Diese ist unterteilt in die Beschreibung der Semantik der einzelnen Modellelemente und in deren externe Darstellung, insbesondere die Form der Modellelemente (Notation) der einzelnen Diagrammtypen. Diese Schicht dient als Referenz für die Modellierung von UML-Diagrammen in CASE-Werkzeugen.

Die erstellten UML-Diagramme – also die Modelle – sind der dritten Schicht zuzuordnen, da sie Instanzen des Metamodells sind. Instanzen der Modelle sind, sofern vorhanden, der vierten Schicht zuzuordnen. Hierzu gehören z. B. der zu Klassendiagrammen korrespondierende Quelltext oder die in einer Datenbank gespeicherten Daten im Falle eines Datenbankschemas.

Relevant für die Modellierung von UML-Diagrammen ist die Modell- und die Metamodell-Schicht. Das Metamodell zur Beschreibung von UML-Diagrammen in einem CASE-Werkzeug kann sich von der UML-Spezifikation in einigen Details unterscheiden. Ein (stark) vereinfachtes Metamodell für Klassendiagramme ist in Abbildung 1.2 dargestellt.

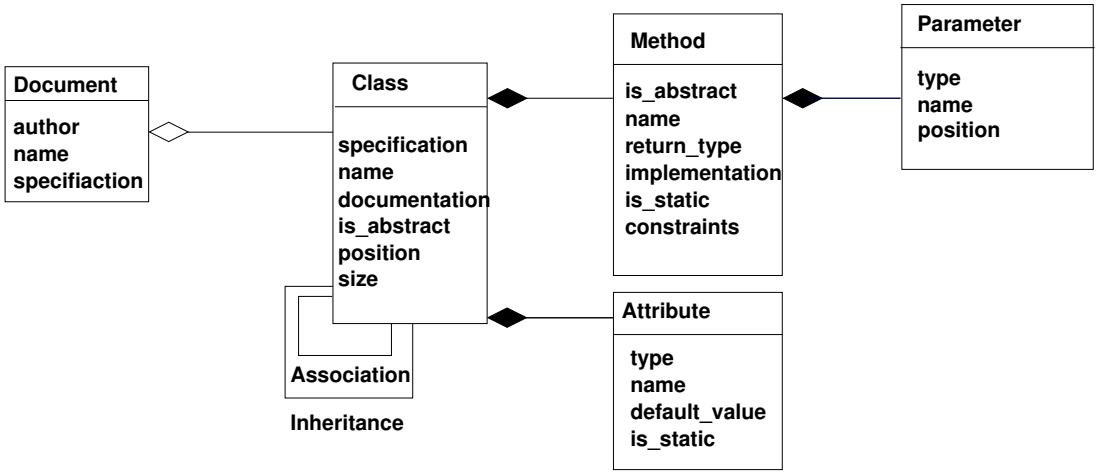


Abbildung 1.2: Vereinfachtes Metamodell eines Klassendiagramms

Hierin ist jede Komponente einer Klasse (Methoden und Attribute) als eigenständiger Objekttyp realisiert. Diese Objekttypen stehen über Assoziationen mit Komponenteneigenschaft miteinander in Beziehung, d.h. das Löschen einer Klasse impliziert das Löschen aller Komponenten der Klasse. Die Beziehungen im Klassendiagramm sind als einfache Assoziationen zwischen den Objekttypen, die die Klasse selbst modellieren, realisiert; die Eigenschaften der Klasse oder deren Komponenten als Attribute der Objekttypen.

Neben dem Metamodell muß ein CASE-Werkzeug auch die Notation, also das Aussehen der Diagrammelemente, definieren. Die Notation ist durch die Implementierung des Werkzeugs vorgegeben. Die Implementierung sagt jedoch noch nichts über die Größe und die Plazierung der Diagrammelemente auf der Zeichenfläche aus, also über das Layout. Dieses ist diagrammspezifisch, und daher Teil des Modells. Das Modell eines Diagramms ist während der Laufzeit eines Werkzeugs i.d.R. als *Syntaxbaum* im Hauptspeicher abgelegt, der beim Beenden in eine Datei auf die Festplatte geschrieben oder in einer Datenbank abgelegt wird. Die vollständige graphische Repräsentation der Diagramme ergibt sich durch die Kombination von Notation und Layout.

1.2.2.2 Versionierung von UML-Diagrammen

Aus obigen Überlegungen folgt, daß die mit Werkzeugen erstellten UML-Diagramme in drei unterschiedlichen Repräsentationen vorliegen (siehe auch Abbildung 1.3):

- graphische Darstellung im Werkzeug: das „Bild“ des Diagramms
- physische Repräsentation: z. B. eine oder mehrere Binär-Dateien oder Text-Dateien z. B. im XMI-/XML-Format

- Repräsentation ähnlich einem Syntaxbaum: Instanzen des werkzeugspezifischen Metamodells der Diagramme, die im Hauptspeicher oder in einem Repository gespeichert sein können. Diese Repräsentation ist werkzeugspezifisch und unterscheidet sich i.d.R. vom abstrakten Syntaxbaum. Das werkzeugspezifische Metamodell enthält z. B. nur eine Teilmenge, der in der UML definierten Elemente und Beziehungen. Die Editiermöglichkeiten der Diagramme sind somit direkt von den auf dem Metamodell durch eine Grammatik definierten Operationen abhängig. Alle Änderungen an den Diagrammen erfolgen auf einer Instanz des Metamodells, von daher bezeichnen wir diese Repräsentation im folgenden als *Editiermodell* [129]. Die Struktur des Editiermodells ist oft graphartig. Das Metamodell des Editiermodells bezeichnen wir als *Editier-Metamodell*. Wenn wir im folgenden den Begriff *Syntaxbaum* verwenden, verstehen wir darunter den Spannbaum des Editiermodells, der durch die Komponentenbeziehungen aufgespannt wird und *nicht* den abstrakten Syntaxbaum.

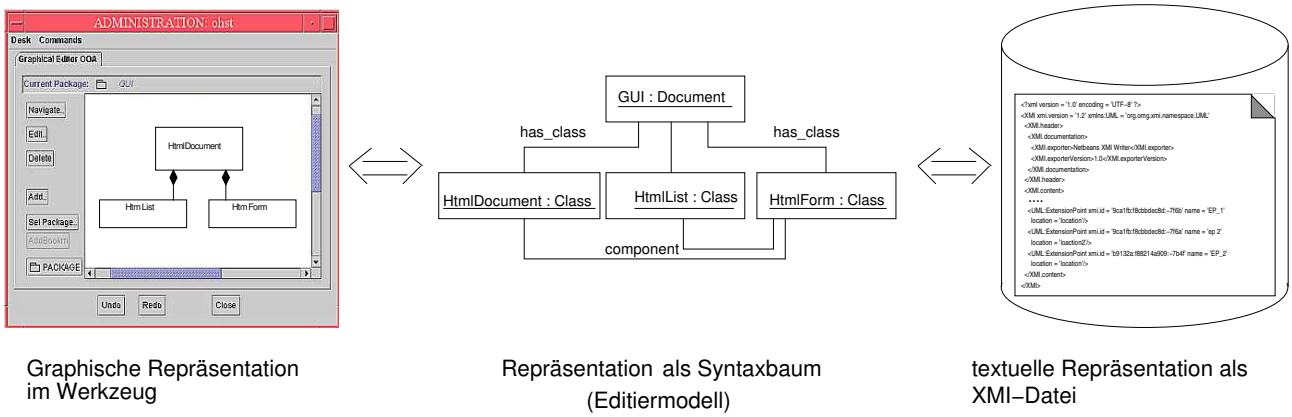


Abbildung 1.3: Darstellung der einzelnen Repräsentationen von Diagrammen in Werkzeugen

Versionierung von Diagrammen. Die graphische Darstellung ist für die Versionierung der Diagramme nicht geeignet, da es sich „nur“ um eine Grafik handelt, die auf dem Bildschirm angezeigt oder ggf. noch als Bild gespeichert werden kann. In beiden Fällen handelt es sich um binäre Daten. Diese ließen sich zwar versionieren, indem von jeder Version eine Kopie angelegt wird, jedoch sind dann keine Änderungen mehr möglich, ebensowenig wie eine Differenzbestimmung zwischen zwei Versionen.

Eine vergleichbare Aussage gilt auch für die physische Repräsentation als Binär-Datei. Diese läßt sich versionieren. Konventionelle VM-Systeme legen jedoch intern Kopien an, da sie die interne Struktur nicht kennen und somit die Versionen auch nicht kompakt speichern können. Üblicherweise will man nicht nur Versionen verwalten, sondern auch Unterschiede zwischen ihnen feststellen und diese mischen. Hierzu verwendet man Werkzeuge zur Anzeige und zum Mischen von Differenzen. Diese Werkzeuge arbeiten jedoch i.d.R. auf textuellen Dokumenten, so daß sie bei der Verwendung eines binären Dateiformates versagen. Daher können konventionelle VM-Systeme für Textdateien nicht zur Versionierung von UML-Diagrammen, die in Binär-Dateien gespeichert sind, verwendet werden.
Für eine Versionierung kommt somit also nur die physische Repräsentation als Textdatei oder das Editiermodell in Betracht.

Die UML-Diagramme können unter Verwendung eines textuellen Formates, wie z. B. XML [38]/XMI [179], in einer Datei gespeichert sein. Jedoch unterscheiden sie sich von den rein textuellen Dokumenten darin, daß sie nicht von einem Entwickler geschrieben wurden,

sondern durch ein Werkzeug generiert worden sind. Diese Dateien lassen sich mit Hilfe der VM-Systeme zur Versionierung von Textdateien versionieren.
Die textuellen Repräsentationen der UML-Dokumente unterscheiden sich jedoch von reinen Textdokumenten darin, daß die CASE-Werkzeuge üblicherweise nicht nur ein Diagramm pro Datei speichern, sondern ein vollständiges Projekt mit einer großen Anzahl an Diagrammen (z. B. ArgoUML [193]) und ggf. weitere Dokumente. Das führt dazu, daß auch von unveränderten Diagrammen eine neue Version angelegt wird, wenn die Datei als Ganzes versioniert wird. Das widerspricht z.T. dem Sinn der Versionierung.

Differenzen zwischen Diagrammen. Ein weiterer Unterschied zwischen Textdokumenten und als Text gespeicherten UML-Dokumenten besteht darin, daß die logische Struktur des Dokuments nicht durch die Position der logischen Blöcke im Dokument, wie bei Texten, die durch einen Menschen erstellt wurden, bestimmt ist, sondern durch eindeutige Identifizierer, die durch das Werkzeug vergeben werden. Daher kann dieses auch die Ordnung dieser logischen Blöcke nach jedem Laden-Ändern-Speichern-Zyklus ändern, ohne daß sich die Anzeige des Dokuments ändert. Die VM-Systeme interpretieren das als eine große Anzahl von Änderungen und somit als Differenzen zwischen den Dokumenten. Diese Art von falsch erkannten Differenzen sind ein Beispiel für *Phantom-Differenzen*.

Beim *Mischen* von Versionen, das Systeme wie CVS oder ClearCase für Textdateien bzw. eine kleine Anzahl weiterer Datei-Typen unter Verwendung externer Werkzeuge beherrschen, rufen Phantom-Differenzen eine Vielzahl an *Konflikten* hervor.
Bei Vorliegen eines Konfliktes wurden in beiden Dokumentversionen die selben Dokumentteile verändert, so daß das Mischwerkzeug nicht mehr automatisch anhand einer gemeinsamen Vorgängerversion der Dokumente entscheiden kann, welche Änderung in die Mischversion übernommen werden soll. In diesem Fall müssen die Entwickler die Konflikte manuell auflösen, was jedoch eine fehlerbehaftete und aufwendige Arbeit ist. Das Ziel muß also sein, die Anzahl an Differenzen und insbesondere an Phantom-Differenzen so weit wie möglich zu reduzieren.

Die Phantom-Differenzen sind z. B. durch die Permutierung des Inhalts in der physischen Repräsentation aufgetreten. Durch die Vermeidung der Permutierung, also ggf. durch Änderungen an den Werkzeugen, kann man nicht alle Phantom-Differenzen beseitigen. Unabhängige Änderungen in der graphischen Repräsentationen von Dokumentversionen führen zu Änderungen in der physischen Repräsentation, die nicht notwendigerweise an unterschiedlichen Stellen auftreten und somit wieder zu Phantom-Differenzen führen. Der Hauptgrund hierfür ist, daß die Dokumentstruktur bei der Versionierung, bei der Differenzbestimmung und beim Mischen nicht berücksichtigt wird [244, 175][3].

Unabhängig von den Phantom-Differenzen existiert ein weiterer Nachteil bei der textuellen Differenzbestimmung oder beim Mischen von Versionen. Das Differenz- und Misch-Werkzeug des VM-Systems zeigt nur textuelle Differenzen und nicht die Differenzen zwischen den Diagrammen, die einen Entwickler interessieren. Er müßte aus den Differenzen in der textuellen Repräsentation die Differenzen im Diagramm ableiten, was bei großen Diagrammen sehr kompliziert werden kann. Wenn in der Datei dann neben den Modelldaten auch noch Layoutdaten gespeichert sind, wie in den XMI-Dateien, die z. B. Poseidon generiert, dann wird ein Vergleich von zwei Versionen nahezu unmöglich, insbesondere dann, wenn das Layout eines Diagramms verändert wurde. Sinnvoll ist daher die Anzeige und das Mischen der Differenzen in den CASE-Werkzeugen.

[3] Selbst bei der Versionierung von Quelltexten, die ebenfalls strukturiert sind, wird deren Struktur durch konventionelle Versionsmanagement-Systeme nicht berücksichtigt [147], was einen Bruch darstellt zwischen der Denkwelt der Entwickler, die auf Klassen, Methoden usw. basiert und der Denkwelt des SKM-Systems, welche nur Dateien kennt.

Zusammenfassend läßt sich sagen, daß die textuelle Repräsentation von UML-Diagrammen nicht zur Versionierung und Differenzbestimmung geeignet ist, da sie viele Nachteile besitzt, die darauf zurückzuführen sind, daß das Editiermodell unberücksichtigt bleibt. Daraus folgt, daß das Editiermodell zur Versionierung verwendet werden sollte [244, 175].

1.2.3 Modellierungs- und Versionierungsarten

Viele Werkzeuge speichern die Daten in einer oder mehreren Dateien auf dem persistenten Speicher. Diese Art der Datenspeicherung, in der die gesamten Informationen als ein großer zusammenhängender Datenblock abgelegt werden, bezeichnen wir als *grobkörnige Speicherung*. Wenn die Werkzeuge keine eigenen Versionierungsmechanismen anbieten, können externe VM-Systeme nur auf den Dateien auf dem persistenten Speicher aufsetzen. Das besitzt einige Nachteile:

- Der Grad der konkurrierenden Arbeit an einem Dokument ist gering, wenn das gesamte Dokument gegen Änderungen durch das VM-System gesperrt ist.
- Es ist schwierig abschätzbar, wieviel Aufwand die spätere Integration einer konkurrierenden Änderung bedeutet und ob es daher besser ist, zu warten, bis die gesperrte Version wieder freigegeben wird.
- Die Versionsgeschichte eines Teildokumentes ist nur schlecht rekonstruierbar [46].
- Bei strukturierten Dokumenten versagen die externen Differenz- und Mischwerkzeuge (siehe Abschnitt 1.2.4).

Eine andere Methode persistenter Datenspeicherung ist der Einsatz eines *Repositorys* bzw. *Objektmanagement-Systems* (OMS). In diesem Fall ist der Einsatz von externen VM-Systemen nicht möglich, da diese keinen Zugang zu den Daten im OMS haben. Die einzige Möglichkeit, Daten zu versionieren, besteht darin, daß das verwendete OMS eigene Versionierungsmechanismen anbietet, die genutzt werden können.
Bei der Speicherung von Daten in einem OMS kann man zwischen feinkörniger Speicherung und grobkörniger Speicherung unterscheiden. Bei letzterer Methode werden die Daten wie in einer Datei als Ganzes gespeichert. Beispiele für OMS zur grobkörnigen Speicherung sind PCTE [225, 109] oder DAMOKLES [1].
Bei der feinkörnigen Speicherung verwendet man ein Datenbankmodell zur Speicherung der Daten im OMS, welches dem Editier-Metamodell der Dokumente entspricht. Beispiele für OMS mit Versionsunterstützung zur feinkörnigen Speicherung sind [233, 197, 6, 154, 20, 118, 224, 175]. In IPSEN [233], GOODSTEP [197] oder PCTE [225, 109] wird beim Anlegen einer neuen Version eines Dokumentes das gesamte Dokument versioniert. Hier sprechen wir von *grobkörniger Versionierung*. Das hat den Nachteil, daß aus Benutzersicht mehr versioniert wird als notwendig und daß man die Information verliert, welche Dokumentteile in welcher Version geändert wurden. Der Vorteil ist jedoch, daß die Konsistenz von unterschiedlichen Dokumenten durch Beziehungen sichergestellt wird.
Andere Vorschläge zur Versionierung wie das Unified Extensional Versioning Model [8] basieren auf dem Prinzip der *Versionspropagierung*. Das Erzeugen einer neuen Version wird hierbei auf dem Pfad vom geänderten Knoten bis zum Wurzelknoten des Editiermodells propagiert. VM-Systeme, die hierauf basieren, versionieren weniger Daten als IPSEN und GOODSTEP. Jedoch führt auch hierbei die Versionierung eines einzelnen Knotens im Editiermodell zur Versionierung des gesamten Dokumentes, da der Wurzelknoten ebenfalls versioniert wird. Eine andere

Methode zur Versionierung von Programmtexten und Hypermedia Dokumenten verwendet Ensemble [223, 224]. Diese bezeichnen wir als *feinkörnige Versionierung*, da hierin jeder Knoten des Editiermodells unabhängig versioniert wird. Die Konsistenz der einzelnen Versionen der Knoten stellt ein dokumentweiter Versionsbaum sicher. Jedoch wird nicht die Konsistenz von unterschiedlichen Dokumenten sichergestellt.

Alle bisher vorgestellten Ansätze zu Versionierung von Daten basieren auf der *zustandsbasierten Versionierung*[4]. Aus Benutzersicht wird hierbei von dem versionierten Element[5] eine neue Version, d.h. eine Kopie mit den neuen Eigenschaften angelegt[6], die i.d.R. über eine Versionsnummer referenziert werden kann.
Neben der zustandsbasierten Versionierung gibt es noch die änderungsbasierte Versionierung [82] und die operationsbasierte Versionierung [136,191,6]. Bei diesen Verfahren wählt man die Versionen nicht anhand von Versionsnummern aus, sondern anhand der durchgeführten Änderungen. Die zustandsbasierte Versionierung basiert also auf den Zuständen der Daten zu diskreten Zeitpunkten und die änderungsbasierte Versionierung auf den Zustandsübergängen. Der Unterschied zwischen der änderungsbasierten Versionierung und der operationsbasierten Versionierung ist die Art der Änderungen. Die operationsbasierte Versionierung speichert die Änderungen in Form der ausgeführten Operationen auf den Daten und die änderungsbasierte Versionierung verwendet die Unterschiede zwischen den Daten. Beispiele hierfür sind NUCM [104], EPOS [97, 96], Aide-de-Camp [64] und DaSC [152].
Diese Verfahren haben im Vergleich zu der zustandsbasierten Versionierung den Vorteil, daß die Unterschiede zwischen den einzelnen Versionen schnell bestimmbar sind und die Versionen auf Basis der Änderungsgeschichte gemischt werden können. Durch die Speicherung auf Basis der Unterschiede eignen sich diese Verfahren prinzipiell auch für strukturierte Dokumente. Die entscheidenden Nachteile sind jedoch, daß die Änderungs- oder Operationslisten durch die Editoren geführt werden müssen, was den Werkzeugbau erschwert, und daß das Wiederherstellen einer älteren Version einen erhöhten Aufwand erfordert, da diese erst berechnet werden muß.

1.2.4 Differenzbestimmung und Mischen von Versionen

Eine wichtige Funktionalität bei VM-Systemen ist die Möglichkeit, Differenzen zwischen einzelnen Versionen zu bestimmen und Varianten zu einer gemeinsamen Nachfolger-Version zu mischen [161]. Die hierbei auftretenden Begriffe wie Differenz, Phantom-Differenz, Delta und Edit-Skript wurden bisher nur unpräzise beschrieben. Eine Definition ist jedoch zum Verständnis der Problematik der aDifferenz. B.rechnung, -Anzeige und beim Mischen von Dokumenten relevant. Daher sollen die Begriffsdefinitionen jetzt nachgeholt werden.

Begriffsdefinitionen. Von zentraler Bedeutung für die Bestimmung und Anzeige von Differenzen ist das Identifizieren von *korrespondierenden Dokumentteilen* der zu vergleichenden oder zu mischenden Dokumente. Der Begriff *Dokumentteil* meint hierbei einen Teil eines Dokumentes (oder anders formuliert: eine Komponente des Editiermodells) und ist abhängig von den zu vergleichenden Dokumenttypen und der Repräsentation, die zur Differenzberechnung verwendet wird. Bei Textdokumenten sind Dokumentteile z. B. Folgen von Zeilen, und bei UML-Diagrammen in der Repräsentation eines Editiermodells können einzelne Objekte oder Teilbäume einen Dokumentteil bilden. Eine vorläufige Definition des Begriffs *korrespondierend*,

[4] engl.: state based versioning
[5] Das kann ein Knoten eines Editiermodells, das gesamte Editiermodell, aber auch eine Datei sein.
[6] Die VM-Systeme optimieren i.d.R. intern die Speicherung von Versionen, indem sie nur die Unterschiede zwischen den einzelnen Versionen speichern.

die als Ausgangsbasis für weitere Überlegungen dient, lautet folgendermaßen: *Je ein Dokumentteil eines Dokumentes A und eines Dokumentes B korrespondieren, wenn sie durch Änderungen an einem Dokumentteil eines Dokumentes C aus diesem hervorgegangen sind und das Dokument C ein gemeinsamer Vorgänger von A und B ist*[7]. Für die folgenden Betrachtungen legen wir eine zustandsbasierte Versionierung zugrunde. Diese Definition basiert implizit auf zwei Annahmen:

1. Jeder Dokumentteil läßt sich eindeutig identifizieren, z. B. durch einen eindeutigen Identifizierer.

   Das ist jedoch nicht in jedem Fall gegeben. Einfluß auf die Bestimmung der korrespondierenden Dokumentteile haben u.a. der Dokumenttyp und der verwendete Algorithmus zur Differenzberechnung, selbst wenn das zugrundeliegende Editiermodell eindeutige Identifizierer enthält.

   Die über einen Identifizierer als korrespondierend identifizierten Dokumentteile können sich teilweise oder auch vollständig unterscheiden. Ein Beispiel hierfür sind die Versionen eines Objektes, deren Attributwerte sich (vollständig) unterscheiden. Beide Objektversionen werden jedoch anhand ihrer Identifizierer als korrespondierend betrachtet.

2. Beide Dokumente sind aus einem Ausgangsdokument entstanden.

   Diese Aussage trifft nicht immer zu. Einerseits kann Dokument B auch aus Dokument A entstanden sein[8], andererseits können auch Dokumente verglichen oder gemischt werden, die keine gemeinsame Versionshistorie besitzen. Der zweite Fall widerspricht auch der ersten Annahme, daß sich jeder Dokumentteil eindeutig identifizieren und somit der korrespondierende Dokumentteil einfach bestimmen läßt.

Bei der Definition des Begriffs korrespondierende Dokumentteile sollte, nach den vorangegangen Überlegungen, auch die (teilweise) Ähnlichkeit der Dokumentteile selbst berücksichtigt werden, da anhand von Identifizierern bestimmte korrespondierende Dokumentteile auch durch Änderungen an ihnen vollständig unterschiedlich sein können. Genau genommen ist die Existenz von Identifizierern auch ein Sonderfall, der die Differenzbestimmung erleichtert. In vielen Fällen gibt es für die Differenzberechnung zwischen Editiermodellen, die keine eindeutigen Identifizierer besitzen, keine andere Möglichkeit als die Ähnlichkeit der Dokumentteile zu berücksichtigen. Daher gilt, daß korrespondierende Dokumentteile nicht in jedem Fall dieselben Eigenschaften, Komponenten bzw. Attributwerte besitzen müssen und die Unterscheidung zwischen korrespondierenden und *unveränderten* Dokumentteilen erforderlich ist.

Eine weitere Betrachtung der oben genannten Definition führt zu der Feststellung, daß keine Aussage darüber getroffen wird, an welcher „Position“ die korrespondierenden Dokumentteile abgelegt sind. Bei Textdokumenten geht man i.d.R. davon aus, daß die Reihenfolge der korrespondierenden Dokumentteile – also auch deren Position – in beiden Dokumenten identisch ist. Diese Annahme hat jedoch den Nachteil, daß Verschiebungen innerhalb eines Dokumentes unberücksichtigt bleiben und somit zu Phantom-Differenzen (s.u.) führen. Besitzen die Dokumentteile Identifizierer, so lassen sich auch Verschiebungen innerhalb eines Dokumentes nachvollziehen. Bei graphischen Dokumenten, wie z. B. UML-Diagrammen, kann man zwei Positionsbegriffe unterscheiden:

[7] Sind die Dokumente in einem VM-System gespeichert, welches auf der änderungsbasierten Versionierung basiert, so treten die korrespondierenden Dokumentteile in den Hintergrund, da die Differenzen zwischen den Dokumenten die Grundlage der Versionierung bilden. Korrespondierende Dokumentteile sind nur dann relevant, wenn man unversionierte Dokumente oder Dokumente, die in einem zustandsbasierten VM-System gespeichert sind, vergleichen will.

[8] Hierbei handelt es sich bei den Dokumenten A und C aus der vorläufigen Definition um dasselbe Dokument.

- Position in der graphischen Repräsentation: In diesem Fall sagt die Position etwas darüber aus, wo ein Dokumentteil in Bezug zu anderen Dokumenten angeordnet ist. Der Begriff bezieht sich auf das Layout.

- Unter Position kann man aber auch die Position im Editiermodell meinen, also z. B. welchem Knoten des Editiermodells ist der Knoten, der das Dokumentteil repräsentiert, als Komponente zugeordnet, welche Knoten besitzt er als Komponenten und welche Geschwisterknoten gibt es.

Der Begriff *korrespondierend* läßt sich nach diesem Überlegungen allgemein nur vage definieren: *Dokumentteile zweier Dokumente sind* korrespondierend *wenn sie hinreichend viele übereinstimmende Merkmale besitzen.* Wann sie hinreichend viele übereinstimmende Merkmale besitzen, wie z. B. unveränderte Attributwerte an Objekten, unveränderte Zeilen eines Absatzes in einem Text oder denselben Objekt-Identifizierer, ist abhängig vom Dokumenttyp und den Anforderungen des Anwenders.

Differenz vs. Delta. Nach diesen Überlegungen, können wie den Begriff *Differenz* wie folgt definieren: *Unter Differenz versteht man einen Dokumentteil eines Dokumentes A, das keinen korrespondierenden Dokumentteil in einem Dokument B besitzt oder den inhaltlichen Unterschied zwischen zwei korrespondierenden Dokumentteilen.*
Aus dieser Definition folgt, daß ein Vergleich von zwei Dokumenten viele Differenzen als Ergebnis liefern kann. Alle Differenzen, in denen sich zwei Dokumente unterscheiden, bezeichnen wir als *Delta*, in Anlehnung an die Definition von Conradi und Westfechtel [58]. Wenn wir im folgenden von Differenzen sprechen, meinen wir eine Teilmenge der durch ein Delta beschriebenen Differenzen zwischen zwei Dokumenten.

Algorithmen zur Differenzberechnung bestimmen das Delta von zwei Dokumenten und Werkzeuge zur Differenzanzeige visualisieren es. Ein Delta kann

- mengenwertig oder

- operational repräsentiert werden.

Die Darstellung einer Differenz des mengenwertig repräsentierten Deltas enthält die Position und den inhaltlichen Unterschied der korrespondierenden Dokumentteile. Unterscheiden sich zwei korrespondierende Dokumentteile an derselben Position[9], so beschreibt die Darstellung der Differenz die korrespondierenden Dokumentteile und den inhaltlichen Unterschied zwischen ihnen. Daher enthält die Darstellung einer Differenz Informationen über den Inhalt *beider* korrespondierender Dokumentteile und entspricht somit einer zustandsbasierten Beschreibung der Unterschiede. Ein mengenwertig repräsentiertes Delta bezeichnet man auch als *symmetrisches Delta*.
Die operationale Repräsentation eines Deltas definiert in einer Folge von Änderungsanweisungen wie das Dokument A in das Dokument B zu überführen ist. Eine Änderungsanweisung ist somit die Darstellungsform einer Differenz eines operational repräsentierten Deltas. Da die Folge der Änderungsanweisungen immer nur ein Dokument in das jeweils andere überführen kann, aber nicht umgekehrt, bezeichnet man diese Repräsentation eines Deltas als *gerichtetes Delta*. Eine spezielle Folge von Änderungsanweisungen, die ein konkretes Dokument in ein anderes Dokument überführen, heißt *Edit-Skript*. Wird ein Delta auf ein Dokument *angewendet*, so werden alle Änderungsanweisungen auf diesem Dokument ausgeführt. Daraus folgt, wendet man ein gerichtetes Delta auf das Dokument A an, so erhält man das Dokument B.

[9]Beispiel: Ein Attribut von zwei korrespondierenden Objekten enthält unterschiedliche Attributwerte.

I.d.R. existieren mehrere äquivalente[10] Edit-Skripte, die ein gerichtetes Delta zwischen zwei Dokumenten beschreiben. Ziel von vielen Differenzalgorithmen ist die Berechnung eines *minimalen Edit-Skriptes*. Wann ein Edit-Skript minimal ist, hängt von den Änderungsanweisungen ab, die für einen konkretes Edit-Skript erlaubt sind, und von der Gewichtung der Änderungsanweisungen. Komplexere Änderungsanweisungen führen dazu, daß ein gerichtetes Delta mit weniger Änderungsanweisungen beschrieben werden kann. Die unterschiedliche Gewichtung von Änderungsanweisungen beeinflußt die Auswahl einer (Folge von) Änderungsanweisung(-en), die einen Dokumentteil in den korrespondierenden Dokumentteil überführen, und somit beeinflusst die Gewichtung auch die Länge des Edit-Skriptes. Mit der Einführung von gewichteten Änderungsanweisungen ist ein minimales Edit-Skript nicht mehr mit dem Edit-Skript gleichzusetzen, welches die geringste Anzahl an Änderungsanweisungen beinhaltet. Die Bezeichnung „minimal“ bezieht sich in diesem Fall auf die Summe der Gewichte der Änderungsanweisungen.

Phantom-Differenzen. Eine spezielle Form von Differenzen sind *Phantom-Differenzen*. Das sind Differenzen, die auf nicht oder falsch erkannte korrespondierende Dokumentteile zurückzuführen sind. Diese treten aufgrund fehlender Informationen über die Syntax der zu vergleichenden Dokumente auf. Ein Vergleich der Versionen in einer Repräsentation auf einer anderen Abstraktionsebene, die die Syntax berücksichtigt, könnte die korrespondierenden Dokumentteile beider Versionen korrekt zuordnen und würde daher nicht zu den Phantom-Differenzen führen.

Gründe für falsch oder nicht zugeordnete korrespondierende Dokumentteile sind z. B. Permutierungen (vgl. Abschnitt 1.2.2.2) oder auch geänderte Identifizierer von Dokumentteilen.
Phantom-Differenzen können nicht nur in der physischen Repräsentation der UML-Diagramme auftreten, sondern auch in der Repräsentation als Editiermodell. Wenn man die Differenzen zwischen zwei Instanzen eines Editiermodells unter Verwendung von Knotenidentifizierern berechnet und z. B. eine Klasse in einem UML-Diagramm löscht und anschließend exakt wieder anlegt, so wird diese Klasse als unterschiedlich betrachtet, da beide Versionen der Klasse unterschiedliche Identifizierer besitzen. Würde die Differenzberechnung jedoch in der Repräsentation als abstrakter Syntaxbaum ohne Knotenidentifizierer unter Berücksichtigung der Syntax und der Definition eines Ähnlichkeitsmaßes durchgeführt, so könnten beide Versionen der Klasse korrekt zugeordnet werden.[11]
Phantom-Differenzen treten nach diesen Überlegungen immer dann auf, *wenn für zwei unterschiedliche Repräsentationen eines Dokuments auf einer Abstraktionsebene, genau eine Repräsentation des Dokuments auf der nächsthöheren Abstraktionsebene existiert.* Interpretiert man die Transformation der Repräsentation eines Dokumentes auf einer Abstraktionsebene in eine Repräsentation auf einer anderen Abstraktionsebene als Funktion, so treten Phantom-Differenzen immer dann auf, wenn diese Funktion nicht injektiv ist.

Mischen und Konflikte. Die Berechnung eines Deltas zwischen zwei Dokumenten, den Basisdokumenten bzw. Basisversionen[12], ist der erste Schritt zum Mischen von diesen Dokumenten. Das Ergebnis des Mischens ist das *Mischdokument* bzw. die *Mischversion*. Dieses enthält alle Dokumentteile, die beide Basisdokumente gemeinsam besitzen und eine Menge von Dokumentteilen oder Differenzen, die spezifisch für ein Basisdokument sind.

[10] „Äquivalent“ bedeutet hier, daß unterschiedliche Folgen von Änderungsanweisungen möglich sind, um das Dokument A in das Dokument B zu überführen.

[11] Das würde jedoch wahrscheinlich dazu führen, daß andere Klassen nicht korrekt zugeordnet werden könnten. Die besten Ergebnisse erzielt man wahrscheinlich mit einer Kombination beider Ansätze.

[12] Abhängig davon, ob die Dokumente versioniert sind oder nicht, sprechen wir von Basisversion oder resp. Basisdokument.

Man unterscheidet beim Mischen zwischen dem *2-Wege-Mischen* und dem *3-Wege-Mischen*. Beim 2-Wege-Mischen führt man zwei unabhängige Dokumente zu einem Mischdokument zusammen. Dabei müssen die beiden Basisdokumente nicht in einem gemeinsamen Versionsbaum liegen. Beim 3-Wege-Mischen führt man zwei Basisdokumente zusammen, die in einem gemeinsamen Versionsbaum liegen und somit eine gemeinsame Vorgängerversion besitzen.
Ein wesentlicher Unterschied dieser beiden Verfahren besteht in der Erstellung der Mischversion. Beim 2-Wege-Mischen muß man für *jede* Differenz zwischen den beiden Basisdokumenten entscheiden, ob und wie sie in das Mischdokument übernommen werden soll. Eine automatische Entscheidung ist nicht möglich.
Im Gegensatz hierzu ist beim 3-Wege-Mischen eine automatische Entscheidung in einem bestimmten Fall möglich. Dieser liegt vor, wenn für korrespondierende Dokumentteile nur eine Differenz zwischen einer Basisversion und der Vorgängerversion und eine Differenz zwischen beiden Basisversionen existiert. Gibt es Differenzen zwischen beiden Basisversion jeweils zur Vorgängerversion, ist auch beim 3-Wege-Mischen eine manuelle Entscheidung notwendig.

Unabhängig von dem Mischverfahren, bezeichnet man die Fälle, in denen eine manuelle Entscheidung notwendig ist, als *Konflikte*. Allgemein formuliert ist ein Konflikt eine Differenz eines gerichteten Deltas, für die nicht automatisch entschieden werden kann, ob der Zustand vor oder nach Anwenden der Differenz in das Mischdokument übernommen werden soll. Diese Aussage gilt sowohl für 2-Wege als auch für 3-Wege-Mischverfahren. Für 3-Wege-Mischverfahren kann man die Aussage einschränken: Bei einem Konflikt handelt es sich um konkurrierende Änderungen an korrespondierenden Dokumentteilen, die nicht beide gleichzeitig in dem Editiermodell der Mischversion enthalten sein können, da sie das Editiermodell an derselben Position in unterschiedlicher Art verändern.

Werkzeuge zur Differenzberechnung und zum Mischen. Die verwendete Methode zur Differenzbestimmung und zum Mischen hängt von der Art der Versionierung ab. Bei der zustandsbasierten Versionierung müssen die Differenzen zwischen den einzelnen Dokumentversionen bestimmt werden. Das kann durch eine Differenzberechnung realisiert sein, die auf dem Zustand der Dokumente aufsetzt und diese vergleicht. Das Ergebnis ist ein Delta, wie z. B. das Ergebnis vom UNIX-diff [115]. Diese Methode läßt sich auch für unversionierte Dokumente nutzen, mit der Einschränkung, daß alle Differenzen manuell zu mischen sind (2-Wege-Mischen). Sind die Dokumente versioniert, so kann man einen 3-Wege-Mischalgorithmus nutzen, wodurch weniger manuelle Mischentscheidungen notwendig sind.
Andere Ansätze (z. B. [196, 244]) berechnen auf Basis der Versionen eine Liste von durchgeführten Änderungen, vergleichbar mit der änderungsbasierten Versionierung, die als Grundlage zur Differenzanzeige und primär zum Mischen dient. Anhand der berechneten oder gespeicherten Änderungslisten können dann die Versionen automatisch gemischt und evtl. Konflikte bestimmt werden [150]. Die Anwender müssen die aufgetretenen Konflikte lösen.
Bei den meisten VM-Systemen muß der Anwender, der das Mischen der Versionen initiiert, die aufgetretenen Konflikte alleine lösen. Waren an der Erstellung der Versionen mehrere Entwickler beteiligt, kann die Konfliktlösung eine langwierige Aufgabe sein, an der alle beteiligten Entwickler mitwirken sollten [21, 30]. Die direkte Kooperation von Entwicklern unterstützen jedoch die wenigsten Systeme.

Die meisten Mischwerkzeuge arbeiten interaktiv. Sie laden die zu mischenden Versionen, lösen die Konflikte so weit wie möglich und fragen bei nicht automatisch lösbaren Konflikten den Anwender (Beispiele sind Teamware/Filemerge [211] für Textdokumente oder ModelIntegrator/IBM-Rational Rose [107] für UML-Diagramme). Bei einer geringen Anzahl an manuell zu lösenden Konflikten ist dieses Vorgehen praktikabel, da die Konfliktlösung innerhalb einer kurzen Zeitspanne durchführbar ist. Sind die zu mischenden Dokumente umfangreich,

so können viele Konflikte zu lösen sein. Bei einer datei-basierten Datenspeicherung ist eine Kooperation von Entwicklern nahezu unmöglich. Die Werkzeuge müßten die Änderungen und die Datenspeicherung synchronisieren.

Ein Nachteil der meisten Mischwerkzeuge ist der Bruch in den Denkwelten von Entwickler und Mischwerkzeug [77]. Die Entwickler denken i.d.R. in Aufgaben, wohingegen viele VM-Systeme auf Dateien oder Dokumentversionen arbeiten, denen beim Mischen ein Konflikt zugeordnet wird. Eine Zuordnung von an einem Konflikt beteiligter Änderung zu der Aufgabe oder sogar die Konfliktlösung auf Basis der Aufgaben ist nicht möglich [176].

1.2.5 Kooperation in VM-Systemen

Viele Entwicklungsumgebungen und SKM-Systeme bieten den Entwicklern keine ausreichenden Möglichkeiten zur Kooperation [46], die jedoch aufgrund der Größe der Entwicklerteams [186, 77] und der daraus folgenden hohen Wahrscheinlichkeit konkurrierender Arbeit an denselben Dokumentteilen sinnvoll wäre.
Die Kooperationsfunktionalität beschränkt sich in den meisten Fällen darauf, Versionen von Dokumenten zentral zu verwalten und den Entwicklern temporär Kopien zur Bearbeitung in privaten oder öffentlichen *Arbeitsbereichen* (engl.: *Workspace*) zur Verfügung zu stellen. Nach Abschluß der Bearbeitung wird die zentrale Verwaltung mit der neuen Version der Dokumente aktualisiert. Dieses Vorgehen bezeichnet man als *Check-Out/Check-In-Modell*. Die Unterstützung der kooperativen Arbeit besteht darin, während der Bearbeitung die Dokumente zu sperren (pessimistischer Ansatz, Annahme: Konflikte sind häufig; z. B. [195, 194, 219]) und so Konflikte zu vermeiden oder die konkurrierende Bearbeitung zu erlauben (optimistischer Ansatz, Annahme: Konflikte treten eher selten auf; z. B. [1, 2, 36]) und die Versionen später zu mischen. Das Check-Out/Check-In-Modell ist weitgehend akzeptiert, jedoch stellt es einen Flaschenhals hinsichtlich der Produktivität dar [30]. Beim pessimistischen Ansatz ist ein Dokument u.U. lange gesperrt und beim optimistischen Ansatz kann das Mischen zeitaufwendig, nicht trivial und fehleranfällig sein [141].

Diese rudimentären Möglichkeiten zur Kooperation wurden in verschiedenen Ansätzen erweitert. Die meisten Vorschläge basieren auf Transaktionen oder Arbeitsbereichen mit veränderten Eigenschaften. Viele Vorschläge kombinieren kooperative Transaktionen und Versionsmanagement (z. B. EPOS [53]). Die Transaktionen können hierbei vor dem Commit Daten austauschen [59]. Der Datenaustausch und die auszuführenden Handlungen bestimmt ein Protokoll [173]. Der Abbruch einer Transaktion kann zu kaskadierenden Rollbacks von kooperierenden oder untergeordneten Transaktionen führen. Um den Abbruch von Transaktionen zu verhindern, sieht ein weiterer Vorschlag das Verändern der Ausführungsreihenfolge von Transaktionen vor [141]. Das ist jedoch für interaktive Arbeit kompliziert oder sogar unmöglich realisierbar, da im voraus alle zu verändernden Daten bekannt sein müssen. Andere Kooperationsformen sehen die kooperative Arbeit von Transaktionen auf einem Arbeitsbereich vor [80]. Das VM-System, in dessen Rahmen die Transaktionen ausgeführt werden, erzeugt in diesem Fällen automatisch von veränderten Objekten eine neue Version und stellt so die Konsistenz der Daten für jede Transaktion sicher. Der Datenaustausch findet in anderen Vorschlägen zwischen den Arbeitsbereichen und nicht zwischen Transaktionen statt (z. B. Adele [73]), wobei Regeln die Synchronisation der Daten in den einzelnen Arbeitsbereichen, die dynamisch in einer Hierarchie angelegt werden können, festlegen. Transaktionen können auf unterschiedliche Arten Daten austauschen. In CONCORD bzw. dessen Nachfolger SERUM [99] gibt es drei Arten, wie Transaktionen Daten austauschen können. Der kooperierenden Transaktion kann der Zugriff auf Daten mittels geänderten Zugriffsrechten gewährt werden, die Transaktion kann der neue Eigentümer der Daten werden oder die Kooperation geschieht implizit durch Sperren.

Eine andere Art der Kooperation wird in COACT [136] verfolgt. Dieses basiert nicht auf einem Datenaustausch, sondern auf dem Austausch der auf den Daten ausgeführten Operationen und ist somit vergleichbar mit der änderungsbasierten Versionierung, die die Autoren als *History Merging* bezeichnen.
Es gibt primär zwei Gründe für die rudimentären oder sehr komplexen Methoden der Kooperation:

1. Dokumente sind grobkörnig gespeichert und versioniert.

2. Entwickler arbeiten auf einer eigenen Kopie der grobkörnig modellierten Dokumente, wie z. B. Quelltext, um deren Konsistenz sicherzustellen.

Die Notwendigkeit von allen Dokumenten, die bearbeitet werden sollen, eine Kopie pro Entwickler zu haben, resultiert primär daher, daß die Dokumente zueinander konsistent sein müssen. Nur wenn sie konsistent sind, läßt sich Quelltext übersetzen und somit die Änderung auf Funktion prüfen. Dieser Grund ist jedoch bei der Bearbeitung von Diagrammen nicht in dieser Ausprägung gegeben. Es ist z. B. ausreichend wenn Analyseklassendiagramme am Ende der Bearbeitung konsistent sind, während der Entwicklung dürfen sie inkonsistente Zwischenzustände annehmen. Vergleichbares gilt für Interaktions-, Anwendungsfall-, Implementierungs-, Zustands- und Aktivitätsdiagramme. Bei Entwurfsklassendiagrammen muß man ihren Entwicklungsstand und die eingesetzten Werkzeuge berücksichtigen. Werden Werkzeuge eingesetzt, die ein Entwurfsdiagramm und Quelltext parallel verwalten, so sind die Konsistenzanforderungen an die Entwurfsklassendiagramme und an den Quelltext nahezu identisch. Vergleichbares gilt auch für andere Diagrammtypen, wenn sich Änderungen an ihnen direkt auf Quelltext auswirken. Ansonsten kann die Konsistenz mit Hilfe eines Analyse-Werkzeugs am Ende der Bearbeitung sichergestellt werden. Diese Lockerung der Konsistenzbedingung erleichtert die konkurrierende Bearbeitung der Dokumente durch mehrere Entwickler und verringert somit die Anzahl der Varianten, die wieder gemischt werden müssen.
Die grobkörnige Speicherung und Versionierung erschweren die kooperative Arbeit an Dokumenten, da ein Dokument nur vollständig in ein Werkzeug geladen werden kann und somit die parallele Bearbeitung verhindert wird. Im Gegensatz hierzu bietet die feinkörnige Speicherung nicht nur Vorteile bei der Versionierung [175], sondern auch bei der kooperativen Arbeit in großen Projekten [47], da die Dokumente durch mehrere Werkzeuge gleichzeitig bearbeitet werden können.

Der Bereich *Computer Supported Cooperative Work* (CSCW) [29] beschäftigt sich primär mit der Kooperation, meistens jedoch ohne die Versionierung zu berücksichtigen. Die Entwickler von TOGA [204] sagen sogar, daß der Einsatz von Versionierung bei synchroner Kooperation nicht notwendig ist. Das mag evtl. für den Einsatzbereich der kooperativen Arbeit an Dokumenten gelten, jedoch nicht, wenn die Versionsgeschichte der Dokumente wichtig ist. Im Gegensatz hierzu schlägt Platz [187] ein Transaktionskonzept für Softwareentwicklungsumgebungen vor, welches die kooperative Arbeit unterstützt. Er sieht Bedarf, das Konzept um Versionsmanagement zu erweitern. Desweiteren sieht Conradi [59] Forschungsbedarf bei der Integration von VM-Systemen und Groupware-Anwendungen.

1.3 Grundlagen und Anforderungen

In Abschnitt 1.2 haben wir verschiedene Aspekte der Versionierung von Dokumenten der Softwareentwicklung insbesondere von Diagrammen betrachtet. Das Fazit, das wir aus diesen Betrachtungen ziehen können, ist, daß die Diagramme feinkörnig modelliert und versioniert werden

sollten. Die feinkörnige Versionierung erfordert entweder die Integration des VM-Systems in die CASE-Werkzeuge, wodurch diese sehr komplex würden, oder eine *OMS-orientierte Werkzeugarchitektur* [132]. In diesem Fall kann die Versionierungsfunktionalität durch das OMS realisiert werden, was den Bau der Werkzeuge vereinfacht, da diese auf Funktionen des OMS zurückgreifen können [167, 187, 65]. Im folgenden gehen wir von einer OMS-orientierten Werkzeugarchitektur aus.

1.3.1 OMS-orientierte Werkzeugarchitektur

In der „klassischen“ Werkzeugarchitektur laden die Werkzeuge die Daten vom persistenten Speicher und erzeugen daraus eine Kopie im Hauptspeicher, die i.d.R. bei CASE-Werkzeugen das Editiermodell ist. Alle Änderungen an den Daten werden zuerst auf der Kopie ausgeführt, die auf Anforderung des Werkzeuganwenders wieder auf den persistenten Speicher geschrieben wird [130], d.h. die Daten auf dem persistenten Speicher und die Kopie im Hauptspeicher divergieren während der Bearbeitung durch den Werkzeuganwender. Das ist auch der Grund für die Probleme bei der Versionierung von feinkörnig modellierten Daten, die grobkörnig gespeichert werden (siehe Abschnitt 1.2.3). Die Notwendigkeit, Varianten bei paralleler Bearbeitung anzulegen, rührt auch daher.
Die OMS-orientierte Werkzeugarchitektur unterscheidet sich von der klassischen Werkzeugarchitektur in einigen entscheidenden Punkten:

- Die Daten sind feinkörnig modelliert und im OMS auch feinkörnig gespeichert. Das hat den Vorteil, daß die Struktur der Daten in den Werkzeugen und im OMS identisch ist und somit keine Konvertierung der Daten notwendig macht.
- Die Anwendung lädt nur die Daten aus dem OMS, die für die Anzeige und Bearbeitung benötigt werden. Das ermöglicht eine größere Parallelität, da weniger Daten infolge der Bearbeitung gesperrt werden müssen.
- Alle Änderungen schreibt das Werkzeug direkt in das OMS zurück. Dadurch enthält das OMS stets die aktuellen Daten und kann andere Werkzeuge über diese Aktualisierung benachrichtigen [188].
- Die Dokumente sind durch eine Multiple-View-Integration [162, 95] redundanzfrei modelliert. Die einzelnen Dokumente bilden ein großes zusammenhängendes Dokument. Die Definition von *Sichten* filtert aus dem gesamten Datenbestand die für die jeweiligen Werkzeuge benötigten Daten heraus.

Der Vorteil der OMS-orientierten Werkzeugarchitektur ist einerseits eine Vereinfachung beim Bau von Werkzeugen, Monecke [167] beschreibt dies ausführlich in seiner Arbeit, andererseits erhält das OMS alle Informationen, die es für die Versionierung der feinkörnig modellierten Daten benötigt. Hierzu zählen Informationen wie z. B., welcher Anwender welche Daten gerade ändert. Durch Kenntnis der in den Werkzeugen ausgeführten Transaktionen kann das OMS neue Versionen automatisch anlegen.

1.3.2 Repository

Die in dieser Arbeit vorgestellten Konzepte sind in dem OMS H-PCTE [124, 121] und in der Werkzeugsammlung PI-SET [167] evaluiert worden. H-PCTE ist eine partielle Implementierung des ISO und ECMA Standards PCTE [109]. Auf H-PCTE wird über eine API [110] zugegriffen.

Das Datenbankmodell von PCTE basiert auf einem erweiterten ER-Modell und besteht aus attributierten Objekttypen, die durch Beziehungen miteinander verbunden sind. Die Beziehungen bezeichnet man in PCTE als Links, die ebenfalls Attribute besitzen können. Die Attribute und Links sind getypt, jedoch liegen sie nicht in einer Typhierarchie wie die Objekttypen. Ein Datenbankschema definiert die Objekt-, Link- und Attributtypen.
Mittels der Datenbankschemata ist es möglich, Sichten auf die H-PCTE-Datenbank, die man auch als *Objektbank* bezeichnet, zu definieren. Die Sichten beinhalten eine Teilmenge aller Typdefinitionen, wobei diese in andere Sichten importiert werden können. Auf Basis dieses Sichtenkonzepts ist die Multiple-View-Integration realisierbar.

Die Dienste des Repositorys. H-PCTE bietet den Werkzeugen mehrere Basisdienste an. Die wichtigste Dienstleistung ist die persistente Verwaltung der Objekte und Links mittels Zugriffs- und Modifikationsoperationen. Neben der reinen Datenverwaltung bietet H-PCTE weitere Dienste an, auf die die OMS-orientierte Werkzeugarchitektur aufsetzt. Zu den Diensten zählen:

- Benutzer- und Benutzergruppen-Verwaltung
- Zugriffsrechte auf Instanz- und Typ-Ebene
- Prozeßkonzept
- Werkzeug-Transaktionskonzept
- Benachrichtigungsmechanismus

Die Benutzer- und Benutzergruppen-Verwaltung ist die Grundlage für die Verwaltung der Zugriffsrechte für die Objekte. Die Rechte können mittels Access Control Lists (ACL) feinkörnig für einzelne Benutzer oder Gruppen vergeben werden.
Das Prozeßkonzept ermöglicht es den Werkzeugen, mehrere Aufgaben quasi-parallel auf unterschiedlichen Teildokumenten auszuführen. Zur Ausführung benötigt jedes Werkzeug eine eigene Sicht auf die Objektbank. Unter Verwendung des Prozeßkonzepts kann jedes Werkzeug für einzelne Aufgaben einen Sub-Prozeß mit einer eigenen Sicht starten.
Zur Synchronisation der Zugriffe der einzelnen Prozesse dienen die Werkzeug-Transaktionen (WTA). Die Besonderheit der WTA ist ein feinkörniges Sperrmodell. Dieses ist in der Lage, nicht nur Objekt-Mengen oder einzelne Objekte zu sperren, sondern auch einzelne Attribute und Links der Objekte. Das bietet den Vorteil, daß andere Datenbank-Prozesse nicht auf die Freigabe von Sperren warten müssen, sofern sie keine Änderungen an denselben Attributen oder Links durchführen wollen. Insbesondere bei der Multiple-View-Integration ist das sinnvoll, da verschiedene Werkzeuge auf dieselben Objekte in unterschiedlichen Sichten zugreifen können. Im Extremfall enthalten beide Sichten keine gemeinsamen Attribut- oder Link-Typen, so daß beide Werkzeuge gleichzeitig auf dasselbe Objekt ohne Konflikte zugreifen können.
Desweiteren wird auf Basis der WTA ein Undo- und Redo-Mechanismus angeboten, der sich auf die OMS-internen Recovery-Mechanismen abstützt. Das entlastet die darauf aufsetzenden Werkzeuge von der Notwendigkeit entsprechende Funktionen zu implementieren.
Durch den erhöhten Grad der Parallelität können mehrere Werkzeuge auf denselben Daten gleichzeitig arbeiten. Das macht allerdings ein anderes Konzept zur Aktualisierung des angezeigten Dokumentes notwendig, da konkurrierende Transaktionen die gerade angezeigten Daten geändert haben könnten. Mit Hilfe des Benachrichtigungsmechanismus informiert das OMS Prozesse über geänderte Objekte, einschließlich der neuen Werte. Die Werkzeuge brauchen somit nur noch die gelieferten Informationen aus der Nachricht auslesen und ihre Anzeige anpassen. Ein Zugriff auf die Objektbank ist in den meisten Fällen nicht mehr notwendig.

1.3.3 Die Anforderungen

An ein VM-System für Diagramme aus den frühen Phasen der Softwareentwicklung stellen sich andere Anforderungen als an andere VM-Systeme. Diese wollen wir i.f. näher beleuchten. Die Anforderungen beruhen einerseits auf den Eigenschaften der zu versionierenden Dokumente, andererseits auf der Art, wie diese Diagramme bearbeitet und weiterentwickelt werden. Ein VM-System sollte beides berücksichtigen.
Die Anforderungen können wir in drei Problembereiche einteilen:

1. Versionierung der feinkörnig modellierten Diagramme
2. Kooperative und isolierte Arbeit an den Diagrammen
3. Differenzbestimmung und Mischen von Diagrammversionen

Anforderung an die Versionierung. Wenn wir diese Problembereiche näher betrachten, fällt auf, daß die letzten beiden Bereiche davon abhängig sind, wie die Dokumente versioniert werden. D.h. wählt man eine andere Art der Versionierung für die Dokumente, beeinflußt das die Art der kooperativen Arbeit, aber auch die Differenzbestimmung und das Mischen von Diagrammversionen. Je mehr Informationen das VM-System berücksichtigen kann, um so umfassender kann die Unterstützung des Anwenders bei der Arbeit sein. Daher ist die wichtigste Forderung: *Feinkörnig modellierte Diagramme sollten feinkörnig versioniert werden.* Das bedeutet für die Diagramme, daß jeder Knoten im Editiermodell unabhängig versioniert werden kann und als eigenständiges Objekt im OMS realisiert ist. Im folgenden sprechen wir daher von Objekten.

Die feinkörnige Versionierung führt zu einer hohen Anzahl an versionierten Objekten [47]. Wenn wir jede Version eines Objektes mit jeder Version aller anderen Objekte kombinieren könnten, würde das zu einer exponentiellen Anzahl an möglichen Kombinationen führen, von denen nur ein kleiner Teil ein sinnvolles, d.h. konsistentes Diagramm ergeben würde. Erschwert würde die Arbeit zusätzlich dadurch, daß die Entwickler die Versionen manuell auswählen müßten. Daher muß das VM-System *konsistente Kombinationen von Objektversionen verwalten.* Die Kombination von konsistenten Objektversionen bezeichnen wir als Konfiguration, die dem Anwender den Zugriff auf die Versionen eines Diagramms erleichtern, da er nicht die gewünschte Diagrammversion aus der Vielzahl von Objektversionen zusammenstellen muß.

Während der Entwicklung tritt öfters der Fall auf, daß ein älterer Zwischenstand des Dokuments benötigt wird, von dem jedoch keine Version angelegt wurde, um keine später nicht benötigten Zwischenversionen speichern zu müssen [15]. Daher *sollte das VM-System Zwischenversionen automatisch anlegen und getrennt verwalten.* Das erhöht die Wahrscheinlichkeit, daß benötigte Zwischenstände als eigene Version vorliegen und verwendet werden können. *Die Vorgabe eines konsistenten Endzustands eines Dokuments muß den Entwicklern überlassen bleiben.*

Bei der (Weiter-)Entwicklung denken die Entwickler i.d.R. in Aufgaben und nicht in Versionen [77]. Die Notwendigkeit, darüber nachzudenken, welche Version als Ausgangspunkt verwendet werden muß, ist für die Erfüllung der eigentlichen Aufgabe eher hinderlich. Verstärkt wird dies noch dadurch, daß der Workflow sich in der Versionierung widerspiegeln sollte [34]. Daraus resultiert die Anforderung, daß *die Versionierung an den Aufgaben orientiert sein sollte*, d.h. einzelne Versionen sollten bestimmten Aufgaben zugeordnet sein. Daraus folgt direkt *die Entwickler sollten nur die Aufgabe wählen müssen, anhand derer das VM-System die hierfür gerade aktuelle Version bestimmt und zur Bearbeitung bereitstellt.*

Da die Konstruktion von Werkzeugen für sich allein betrachtet schon eine komplexe Aufgabe ist, sollte diese nicht durch komplizierte Schnittstellen zum VM-System erschwert werden.

Daraus folgt, daß *die Funktionen des VM-Systems ohne großen Aufwand in die Werkzeuge integrierbar sein sollten.*

Anforderung an die kooperative Arbeit. Softwareentwicklung ist heute nicht mehr eine Aufgabe, an der zwei oder drei Entwickler beteiligt sind, sondern eine Aufgabe für eine größere Gruppe von Entwicklern. Da es zur Lösung bestimmter Aufgaben notwendig sein kann, mehrere Dokumente zu verändern, steigt die Wahrscheinlichkeit, daß ein Dokument durch mehrere Entwickler parallel bearbeitet werden muß. Daraus folgt, daß *ein VM-System kooperative Arbeit unterstützen muß.* Die kooperative Arbeit gibt es in verschiedenen Ausprägungen. Üblich ist die isolierte Arbeit pro Entwickler, deren Änderungen nach Fertigstellung der Teilaufgabe zusammengeführt werden. Dieses Vorgehen ist für Dokumenttypen, wie z. B. Quelltext gut geeignet, da in diesem Fall parallele Arbeit an der selben Dokumentversion zu Konflikten und somit zu Verzögerungen führen würde. Als einfachste Form der kooperativen Arbeit *sollte ein VM-System die isolierte Arbeit an einem Dokument unterstützen.*
In der objekt-orientierten Entwicklung gibt es zentrale Konzepte wie Klassen und Interfaces. Diese sind, speziell in den frühen Phasen, häufigen und z.T. umfassenden Änderungen unterworfen, so daß bei isolierter Arbeit die Entwickler ihre Änderungen frühzeitig wieder allen anderen Entwicklern zugänglich machen müssen [48]. Mit entsprechenden Maßnahmen zur Konsistenzsicherung, wie z. B. dem Einsatz von kooperativen Transaktionen [187], kann eine synchrone kooperative Arbeit an den Dokumenten ermöglicht werden. Das vermeidet Fehler beim Mischen von Varianten der Dokumente und ermöglicht allen Entwicklern, an der jeweils aktuellen Version der Dokumente zu arbeiten. *Daher sollte ein VM-System die synchrone kooperative Arbeit unterstützen.*
Jedoch läßt sich nicht in allen Fällen auf Varianten verzichten, insbesondere dann nicht, wenn die Änderungen exploratorisch sind oder weitreichende Änderungen an vielen Teilen des Dokuments erfordern. Sind die Varianten noch durch mehrere Entwickler erstellt worden, so ist das Mischen eine komplexe Aufgabe, die nicht von einem Entwickler allein durchgeführt werden kann. Daher sollte *ein kooperatives Mischen von Versionen durch alle beteiligten Entwickler möglich sein.* Das beschleunigt einerseits das Mischen und vermeidet andererseits Fehler.

Anforderung an die Differenzdarstellung- und das Mischwerkzeuge. Bei größeren Projekten, an denen mehrere Entwickler beteiligt sind, entstehen viele Versionen von Dokumenten. Diese unterscheiden sich teilweise in vielen direkt offensichtlichen Differenzen oder nur in einigen Details. Das VM-System für Dokumente der frühen Phasen sollte daher Werkzeuge anbieten, die die Differenzen zwischen zwei Versionen eines (UML-)Diagramms anzeigen und bei Bedarf auch mischen können. In der Differenzdarstellung dieser Werkzeuge *sollten die Differenzen deutlich gekennzeichnet sein*, wobei *mindestens ein beteiligtes Basisdiagramm*[13] *wiedererkennbar sein sollte.* Wenn die Entwickler die Diagramme nicht wiedererkennen, können sie die Differenzen nur schlecht oder auch gar nicht interpretieren, so daß die Differenzdarstellung keinen Nutzen für die Entwickler hat.
Enthält die Differenzdarstellung viele Differenzen, so erschwert das deren Lesbarkeit, insbesondere wenn ein Entwickler nur an Differenzen an einzelnen Diagrammteilen oder an Differenzen, die auf bestimmte Änderungen zurückzuführen sind, interessiert ist. Daher *sollte das Werkzeug zur Differenzanzeige die Option bieten, bestimmte Differenzen hervorzuheben und andere auszublenden.* Damit kann die Aufmerksamkeit des Entwicklers auf bestimmte Differenzen gelenkt werden, wodurch dieser einen größeren Nutzen aus der Differenzdarstellung ziehen kann. Das

[13] Unter Basisdiagrammen verstehen wir im folgenden die Diagramme, zwischen denen die Differenzen bestimmt und angezeigt oder die gemischt werden sollen.

ist insbesondere dann der Fall, wenn *der Entwickler selber die Differenzen bestimmen kann, die ihn interessieren.*

Neben der Darstellung der Differenzen muß das Mischen von Versionen unterstützt werden. Da dies eine z.T. aufwendige und fehleranfällige Aufgabe ist, *sollte ein Mischwerkzeug für (UML-)Diagramme so viele Differenzen wie möglich automatisch mischen können.* Das ist durch die Verwendung des 3-Wege-Mischverfahrens möglich. Die automatisch gewählte Lösung beim Mischen ist nicht in allen Fällen die richtige Entscheidung. Wenn beispielsweise ein Diagrammteil automatisch gelöscht wurde, ist es nur unter großem Aufwand möglich, wieder neu anzulegen, falls er weiterhin benötigt wird. Daher *sollten die automatisch getroffenen Entscheidungen vom Entwickler rückgängig gemacht werden können.*
Die Mischfunktion kann nicht alle Differenzen automatisch mischen. Bei einem Mischkonflikt kann keine automatische Entscheidung getroffen werden, so daß der Entwickler manuell entscheiden muß. Treten beim Mischen viele Konflikte auf, so ist das manuelle Mischen eine langwierige und fehleranfällige Aufgabe. Daher *sollten so viele Konflikte wie möglich mit so wenigen Entscheidungen wie nötig gelöst werden können.* Die Entwickler *sollten zum besseren Verständnis des aktuellen Zustands der Mischversion das aktuelle Zwischenergebnis jederzeit sehen können.* Damit die Aufmerksamkeit der Entwickler auf die Konflikte gelenkt wird, an denen sie interessiert sind, *sollte es möglich sein nur die Teilmenge der Konflikte zu markieren, die gerade durch die Entwickler gelöst werden sollen.*

1.4 Das Versionierungskonzept für Analyse- und Entwurfsdiagramme

Während der Softwareentwicklung durchlaufen die erstellten Dokumente unterschiedliche Entwicklungsstadien. Einerseits kann man die Stadien anhand der gängigen Vorgehensmodelle in Phasen einteilen, z. B. in Analyse und Entwurf, andererseits kann man diese Phasen weiter unterteilen in einzelne Aufgaben, die in einer Phase durchzuführen sind. Zur Umsetzung einer Aufgabe setzt man i.d.R. mehrere Werkzeuge ein. Abhängig von der Aufgabe kann deren Bearbeitung einen längeren Zeitraum in Anspruch nehmen und die Nutzung verschiedener Werkzeuge notwendig machen. Wir können also zusammenfassend sagen, daß die Dokumente in einer Phase verschiedene Zustände durchlaufen, in denen eine Anzahl an Dokumentversionen angelegt wird. Die Versionsverwaltung sollte dieser Problematik Rechnung tragen. Zu berücksichtigen sind dabei die Aspekte der kooperativen Arbeit und damit zusammenhängend die Konsistenz einzelner Dokumente, aber auch die Konsistenz der Dokumente untereinander. Für diese Aufgaben eignen sich Transaktionen, wie sie aus dem Bereich des Software-Konfigurationsmanagements (SKM) bekannt sind.
In diesem Abschnitt stellen wir das im Rahmen dieser Arbeit entwickelte Versionsverwaltungskonzept für feinkörnig modellierte Dokumente vor, die in einem OMS gespeichert sind.

1.4.1 Entwurfstransaktionen und Arbeitsbereiche

Die Transaktionen im Bereich des SKM besitzen andere Eigenschaften als die Transaktionen, die aus dem Bereich der relationalen Datenbanken bekannt sind. Insbesondere besitzt diese Art von Transaktionen, die man auch als *Entwurfstransaktionen* (ETA) (z. B. [135, 138, 69, 123, 12]) bezeichnet, eine deutlich längere Laufzeit. Die zugrundeliegende Idee ist im Bereich des SKM als Konzept der *langen Transaktion* [82][14] bekannt. Dabei stellen diese Transaktionen die Basis

[14] Neben diesem Konzept gibt es noch drei weitere grundlegende Konzepte, die wir in Abschnitt 2.1.3 vorstellen.

zur Kooperation zwischen (Gruppen von) Entwicklern dar. Die ETA dienen dabei weniger der Synchronisation von gleichzeitigen Zugriffen auf dieselbe Dokumentversion als vielmehr deren Bereitstellung einschließlich von Zugriffskontrollen. Daraus folgt, daß eine ETA mehrere Werkzeugsitzungen umfaßt, in denen die Dokumente bearbeitet werden.

Bearbeitung von Aufgaben in Entwurfstransaktionen. Die ETA bilden die Grundlage des Versionierungskonzepts. Sie müssen jedoch den Besonderheiten der OMS-orientierten Werkzeugarchitektur angepaßt werden.
Die Grundidee ist folgende: Für eine oder eine Gruppe von Aufgaben wird eine ETA angelegt. Diese kann in weitere Sub-ETA unterteilt werden, so daß dem Aufgaben-Baum ein ETA-Baum gegenüber steht. Die Wurzel entspricht dabei z. B. einem Projekt, welches in einzelne Entwicklungsphasen, Aufgaben und Teilaufgaben unterteilt sein kann. Die ETA gliedern somit die Dokumentversionen entsprechend den Aufgaben oder Projektphasen. Die Struktur der ETA ist flexibel und ist somit an unterschiedliche Prozesse anpaßbar. Die Anzahl der ETA und die Schachtelungstiefe sind frei wählbar.

Pro Aufgabe sind i.d.R. ein oder mehrere Dokumente zu bearbeiten, die durch die korrespondierende ETA verwaltet und bereitgestellt werden. Während der Bearbeitung der Aufgabe können neue Dokumente in der ETA angelegt oder existierende Dokumente aus der übergeordneten ETA oder aus einer Geschwister-ETA importiert werden.
Falls eine isolierte Arbeit an den Dokumenten einer ETA notwendig sein sollte, so kann ein Entwickler exklusiven Zugriff auf eine ETA und damit auf die von ihr verwalteten Dokumente erhalten. Andernfalls ist es durch die Nutzung von *Werkzeugtransaktionen* (siehe Abschnitt 1.4.2) möglich, gleichzeitig an den durch eine ETA verwalteten Dokumenten zu arbeiten. Die Werkzeugtransaktionen stellen die Konsistenz mittels eines Sperr-Protokolls sicher.
Vor dem Abschluß der Aufgabe müssen die Dokumente mit der jeweiligen Vorgängerversion in der übergeordneten ETA gemischt werden. Das Mischen der Versionen kann nicht vollständig durch das VM-System realisiert werden, insbesondere ist das Lösen von möglichen Konflikten eine interaktive Aufgabe, die wir in Abschnitt 1.5 skizzieren und in Abschnitt 5.4 ausführlich diskutieren.

Die virtuellen Arbeitsbereiche. Die Dokumente einer Entwurfstransaktion liegen im ihr zugeordneten Arbeitsbereich. Die Arbeitsbereiche dienen vor allem der *Isolation* von Änderungen einzelner Entwickler. Das Konzept des Arbeitsbereichs, wie es in den meisten Entwurfstransaktionskonzepten verwendet wird, ist in der konventionellen Form nur solange anwendbar, wie die Dokumente grobkörnig modelliert und gespeichert sind, also in einzelnen Dateien vorliegen. Liegt eine feinkörnige Speicherung vor, kann man die Arbeitsbereiche im klassischen Sinn nicht mehr verwenden, da nicht nur einzelne Dateien zwischen den Arbeitsbereichen transferiert (engl. check-out/check-in) werden müssen, sondern eine große Anzahl von Objekten und Links[15]. Der Grund hierfür liegt darin, daß ein Dokument aufgrund der feinkörnigen Modellierung aus einer Vielzahl an Objekten und Links besteht.
Der Einsatz von Arbeitsbereichen im klassischen Sinn birgt bei feinkörnig modellierten Dokumenten einen Nachteil: Entwickler haben nur Zugriff auf Daten ihres Arbeitsbereichs, benötigen sie weitere Daten, müssen diese in ihren privaten Arbeitsbereich transferiert werden. Besteht ein Dokument aus einer Vielzahl an Objekten, müssen vor dem Transfer alle benötigten Objekte bestimmt werden. Aufgrund unterschiedlicher Sichten bei der Multiple-View-Integration sind jedoch nicht alle Objekte eines Dokumentes für jedes Werkzeug sichtbar. Daher müßten alle

[15] Wir sprechen hier von Links, um die Beziehungen im OMS deutlicher von den Beziehungen in den Diagrammen unterscheiden zu können.

Objekte eines Dokumentes durch das OMS zwischen den Arbeitsbereichen transferiert werden, was einen erheblichen zusätzlichen Aufwand darstellt. Die Beschränkung auf die Objekte in der Sicht eines Werkzeugs ist auch keine gangbare Lösung, da eine enge Kooperation zwischen den Werkzeugen und dem VM-System erforderlich wäre und somit den Bau der Werkzeuge weiter komplizieren würde. Der Einsatz von konventionellen Arbeitsbereichen ist somit nicht praktikabel. Die Arbeitsbereiche müssen aus den genannten Gründen transparent für die Werkzeuge sein, so daß die Dokumente nicht vollständig transferiert werden müssen.

Durch die Integration des VM-Systems in das OMS liegen die Arbeitsbereiche ausschließlich im OMS vor. Daher kann man die Isolationseigenschaft der Arbeitsbereiche aufweichen, so daß Objekte nur noch vor dem Durchführen einer Änderung in einen Arbeitsbereich übertragen werden müssen. Für reine Lesezugriffe sind die Arbeitsbereichsgrenzen transparent. Automatisiert man auch das Übetragen der Objekte vor Schreibzugriffen, so sind die Arbeitsbereiche für die Werkzeuge vollkommen transparent, wenn man von der Verwaltung der Dokumente im Rahmen der ETA absieht.
Bei dieser Realisierung eines virtuellen Arbeitsbereichs kann der Fall auftreten, daß ein Link zwischen zwei Objekten die Grenze zwischen zwei Arbeitsbereichen überbrückt. Das ist der Fall, wenn eines der an einer Beziehung beteiligten Objekte bereits verändert wurde, das andere jedoch nicht. Das OMS muß diesen Fall unterstützen.

Durch das implizite Check-Out einzelner Objekte in einen virtuellen Arbeitsbereich ist es nicht mehr sinnvoll möglich, die übertragenen Dokumente im übergeordneten Arbeitsbereich gegen konkurrierende Änderungen zu sperren. Die Gründe hierfür sind, daß nicht vollständige Dokumente transferiert werden, sondern nur Teile von Dokumenten und daß den Anwendern der Transfer der Objekte zwischen den Arbeitsbereichen nicht bewußt ist. Sperren würden in dieser Situation bewirken, daß einzelne Teile eines Dokumentes, welches durch ein Werkzeug angezeigt wird, durch einen Anwender nicht änderbar sind, wohingegen andere Teile desselben Dokumentes geändert werden können. Dieses Verhalten wäre für einen Werkzeuganwender nicht nachvollziehbar. Gleiches gilt auch für Benachrichtigungsmechanismen zwischen einzelnen Arbeitsbereichen.

Integration von Entwurfstransaktionen und Arbeitsbereichen. Der Hauptgrund für die Existenz der Arbeitsbereiche im vorgestellten Konzept ist die Verwaltung der Dokumente einer ETA. Das OMS ist dadurch in der Lage, die Dokumente und deren Versionen den einzelnen Aufgaben zuzuordnen und den Werkzeugen zur Verfügung zu stellen.
Voraussetzung hierfür ist der Import der Dokumente oder die Erzeugung in den Arbeitsbereichen. Da die Arbeitsbereiche ein Konzept im OMS sind und die Dokumente feinkörnig modelliert vorliegen, ist es nicht notwendig, die gesamten Dokumente zwischen den Arbeitsbereichen zu transferieren. Beim Import eines Dokumentes wird lediglich eine Referenz auf das betreffende Dokument, genauer gesagt auf dessen Wurzel-Objekt, angelegt.

Einbettung der ETA in das OMS. Aufgrund der teilweise langen Bearbeitungsdauer der Aufgaben (im Bereich von Stunden für kleine Aufgaben bis hin zu Monaten für Projektphasen), können die ETA nicht an Betriebssystem-Prozesse gebunden sein, d.h. sie müssen persistent durch das OMS verwaltet werden. Hierzu bietet sich die persistente Repräsentation als Objekt im OMS an. Das bietet einige Vorteile:

- selbstreferentieller Zugriff auf die ETA
- nur wenige neue Schnittstellen des OMS notwendig
- Erweiterung der an einer ETA gespeicherten Daten möglich

Zur Bearbeitung einer Aufgabe muß eine ETA ausgewählt werden. Durch die Repräsentation als Objekt im OMS kann man dafür die regulären Schnittstellen des OMS verwenden. Lediglich zur Verwaltung der ETA und der durch sie verwalteten Dokumente sind neue Schnittstellen notwendig. Diese sind in das Konzept der OMS-orientierten Werkzeugarchitektur integrierbar, da die Abstraktionsebene nicht verändert wird: Dokumente und ETA sind beide als Objekte zugreifbar und somit für die Werkzeuge vergleichbar zu handhaben.

Durch den Sichten-Mechanismus von H-PCTE kann ein Werkzeugentwickler ein ETA-Objekt um weitere benötigte Informationen ergänzen. Hierzu könnten z. B. die Aufgabenbeschreibung zählen oder weitere Informationen aus dem Bereich des Änderungsmanagement (engl.: Change-Management) [18,62,63], welche sich mit der Strukturierung und Steuerung von Änderungen an einem Software-Projekt beschäftigen. Änderungsmanagement und dessen Beziehung zum SKM [163] sind jedoch ein eigenes Forschungsgebiet und werden daher in dieser Arbeit nicht weiter betrachtet.

1.4.2 Versionierung in erweiterten Werkzeugtransaktionen

Die feinkörnig modellierten Dokumente erfordern ein anderes Vorgehen bei deren Versionierung als grobkörnig modellierte Dokumente. Es ist bei der OMS-orientierten Werkzeugarchitektur nicht mehr möglich und auch nicht sinnvoll, von allen Objekten eines Dokumentes eine neue Version anzulegen, wenn das Dokument bearbeitet wird. Es würden einerseits zu viele Objekte versioniert, andererseits würde auch von Objekten eine neue Version angelegt, die gar nicht in dem verwendeten Werkzeug aufgrund dessen Sicht zugreifbar sind. Daher ist es sinnvoll, jedes Objekt einzeln zu versionieren.

Durch die feinkörnige Modellierung besteht ein Dokument aus einer Vielzahl an Objekten. Ein Werkzeug greift auf mehr als ein Objekt zu, so daß mehrere Objekte bei Änderungen versioniert werden müssen. Das manuelle Anlegen von Versionen durch die Werkzeuganwender ist genausowenig praktikabel wie das Anlegen der Versionen durch die Werkzeuge. In beiden Fällen müßte für jedes Objekt geprüft werden, ob eine neue Version angelegt werden muß. Diese Aufgabe würde einerseits den Bau der Werkzeuge immens erschweren und andererseits die Werkzeug-Entwickler von der eigentlichen Aufgabe – dem Bau der Werkzeuge – ablenken, da neben der Anwendungsfunktionalität auch noch ein Versionierungskonzept umgesetzt werden müßte. Die Lösung besteht darin, das Anlegen von Objektversionen dem OMS zu überlassen, da es die Objekte, deren Versionen und Zugriffe auf diese verwaltet. D.h. das OMS ist zu jedem Zeitpunkt darüber informiert, welche Werkzeug-Transaktionen (WTA) auf welche Objekte und Links zugreifen. Der Vorteil hiervon ist die einfache Integration der Versionsverwaltung in existierende oder neu zu bauende Werkzeuge.

Anlegen und Zugriff auf Versionen. Die grundlegende Idee des vorgestellten Versionierungskonzepts besteht darin, daß automatisch durch das OMS ausschließlich von Objekten und Links eine neue Version angelegt wird, auf die schreibend zugegriffen werden soll. Betrachtet man die durch das OMS angebotenen Dienste, so eignen sich die WTA für diese Aufgabe, da sie alle Zugriffe auf die Objekte synchronisieren[16].

Bei jedem schreibenden Zugriff auf ein Objekt (einen Link) in einer WTA, legt das OMS von diesen eine neue Version an, jedoch nur wenn in derselben WTA von diesem Objekt (diesem Link) noch keine neue Version angelegt wurde. Eine mehrfache Versionierung eines Objektes

[16]Die Art der Synchronisation ist abhängig von der Art des Zugriffs und der Art der WTA. Reine Lesezugriffe können auch nicht synchronisiert durchgeführt werden, was dann jedoch die Gefahr von nicht-wiederholbarem Lesen in sich birgt.

(eines Links) im Rahmen einer WTA ist nicht notwendig, da diese nur den Sinn einer Undo-Funktion hätte, die jedoch bereits durch die WTA selbst angeboten wird. Daraus resultiert eine unabhängige Versionierung aller Objekte und Links.

Die feinkörnige Modellierung führt bei diesem Versionierungskonzept zu einer sehr großen Anzahl an Versionen und möglichen Kombinationen daraus. Nicht jede Kombination ergibt ein konsistentes Dokument. Aus diesem Grund legt jede WTA eine *Konfiguration* an, die alle in der WTA erzeugten Versionen von Objekten und Links zusammenfaßt. Während der Laufzeit einer WTA bezeichnen wir deren aktuelle Konfiguration als *Arbeitskonfiguration* .

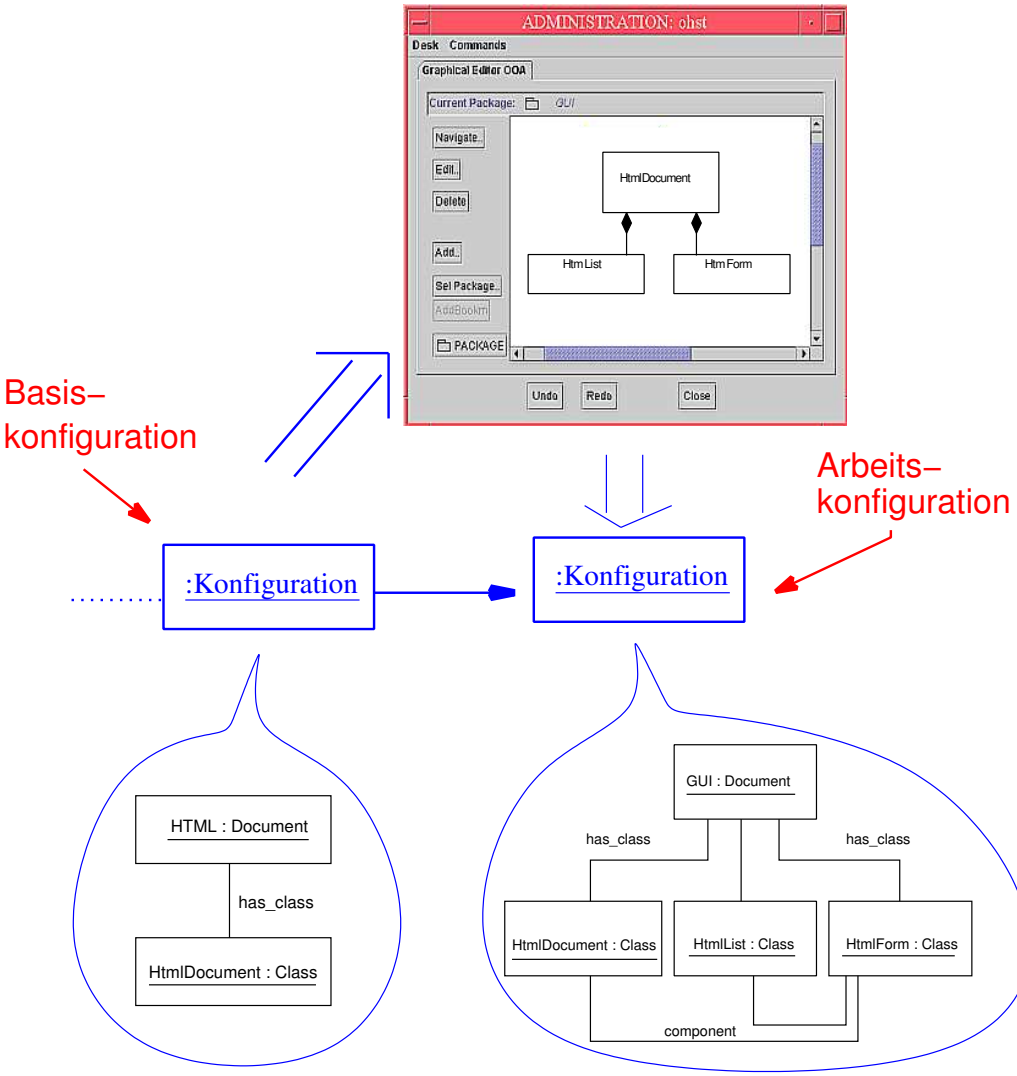


Abbildung 1.4: Zusammenhang von Werkzeug, Konfigurationen und Repository

Die Konfigurationen dienen zur Festlegung aller in einer WTA zugreifbaren Versionen der Objekte und Links. Beim Start einer WTA wird eine Konfiguration als *Basiskonfiguration* ausgewählt. Die Basiskonfiguration ist daher der direkte Vorgänger der Arbeitskonfiguration, siehe Abbildung 1.4. Zur Kennzeichnung besitzt jede Konfiguration einen eindeutigen Identifizierer. Eine WTA kann nur auf die Objekt- und Linkversionen zugreifen, die in der Basiskonfiguration oder deren Vorgängern enthalten sind. Gibt es mehrere Versionen, so ist nur die jüngste Version zugreifbar.

Integration von Versionsverwaltung und Entwurfstransaktionen. Für sich allein betrachtet, stellen die WTA mit den durch sie angelegten Konfigurationen einen Versionierungs-

mechanismus für feinkörnig modellierte Dokumente zur Verfügung. Jedoch wird durch jede WTA eine neue Konfiguration angelegt, so daß eine große Anzahl von Konfigurationen nach einer längeren Entwicklungszeit vorhanden ist. Diese Konfigurationen lassen sich durch die ETA bzw. deren Arbeitsbereiche einzelnen Projekten und deren Aufgaben zuordnen.

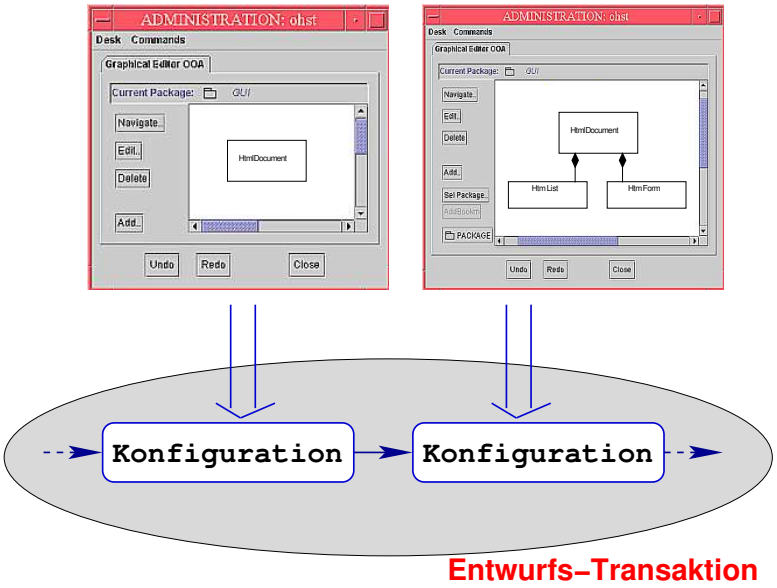


Abbildung 1.5: Integration von WTA und ETA

Die Konfigurationen kann man als Zwischenschritte zur Lösung eines Projektes oder einer von dessen (Teil-)Aufgaben interpretieren. Sie realisieren daher ein werkzeugsitzungsübergreifendes Undo, wobei jeweils die gesamten Änderungen einer Werkzeugsitzung zurückgenommen werden. Die Zuordnung der Zwischenschritte zu einzelnen ETA strukturiert die Änderungen anhand der Aufgaben. Das hat den Vorteil, daß übergeordneten ETA nicht „zu viele“ Konfigurationen zugeordnet sind, die die Übersicht deutlich einschränken würden. Abbildung 1.5 zeigt die Ausführung von WTA in Rahmen einer ETA.

Unterstützung kooperativer Arbeit. Die Kern-Aufgabe der WTA vor der Erweiterung um das Versionierungskonzept bestand in der Unterstützung der kooperativen Arbeit, u.a. durch ein feinkörniges Sperrmodell und den Benachrichtigungsmechanismus. Die Anpassung dieser Mechanismen an das Versionierungskonzept ist sinnvoll, da durch die Kooperation von Entwicklern die Anzahl an Varianten, die wieder gemischt werden müssen, reduziert werden kann. Mit Einführung der Versionen ergibt sich aber auch die Möglichkeit isoliert, an einer Aufgabe zu arbeiten, sofern dies für bestimmte Fälle sinnvoll ist. Die WTA sollten daher beide Arbeitsformen: (a) kooperativ und (b) isoliert unterstützen. In welchem Modus eine WTA arbeiten soll, wird bei deren Start festgelegt.
Bei der isolierten Arbeit muß lediglich an der Konfiguration vermerkt werden, daß keine Kooperation gewünscht wird. Sollten weitere WTA dieselbe Basiskonfiguration wählen, wie die isoliert arbeitende, so wird eine parallele Arbeitskonfiguration angelegt, und die WTA können unabhängig auf Varianten arbeiten. Neue Versionen von Objekten und Links ordnet das OMS den entsprechenden Arbeitskonfigurationen zu.
Die Unterstützung der kooperativen Arbeit erfordert, daß die WTA auf derselben Basiskonfiguration aufsetzen und eine gemeinsame Arbeitskonfiguration besitzen. Dadurch greifen beide WTA auf dieselben Versionen der Objekte und Links zu, die entsprechend dem Sperr-Protokoll gesperrt werden.

Einbettung im OMS und deren Auswirkungen. Die Konfigurationen sind wie die ETA persistent als Objekte im OMS gespeichert. Die Werkzeuge können daher in gleicher Weise auf sie zugreifen wie auf die ETA-Objekte. Mittels des Sichten-Mechanismus des OMS ist es möglich, an den Konfigurationsobjekten zusätzliche Daten zu speichern, wie z. B. Änderungskommentare.

Die Konfigurationsobjekte sind über Links von den ETA-Objekten erreichbar. Hierdurch läßt sich der Zusammenhang von ETA und Konfigurationen im OMS nachbilden, so daß diese Informationen durch die Werkzeuge ausgelesen werden können, ohne daß dafür besondere Schnittstellen erforderlich sind.

Neben den erweiterten Funktionen der WTA und den neuen Objekt-Typen hat das Versionierungskonzept auch Auswirkungen auf den Benachrichtigungsmechanismus. Dieser muß jetzt zusätzlich die Versionen der Objekte und Links berücksichtigen und in Relation zu der Arbeitskonfiguration der jeweiligen WTA setzen. Anhand dieser Informationen muß dann entschieden werden, ob der Prozeß, in dem eine WTA ausgeführt wird, eine Nachricht erhält oder nicht. Eine weitergehende Diskussion dieser Problematik gibt es in Abschnitt 4.1.6.

Das Anlegen von Versionen betrifft auch den Sperr-Mechanismus. Bisher war eine Sperre für das gesamte Objekt gültig. Durch die Einführung von Versionen kann ein Objekt gleichzeitig in unterschiedlichen Modi gesperrt sein, wenn die einzelnen WTA auf unterschiedliche Versionen zugreifen. Beim Anlegen einer neuen Version aufgrund eines Schreibzugriffs muß die existierende Sperre an die neue Version vererbt werden und es muß weiterhin geprüft werden, ob die neue Sperre zuteilbar ist. In Abschnitt 4.1.4 diskutieren wir diese Thematik ausführlicher.

1.5 Das Konzept für Differenz- und Mischwerkzeuge

In den frühen Phasen der Softwareentwicklung ist es notwendig, Unterschiede zwischen einzelnen Versionen von Dokumenten, insbesondere Diagrammen zu bestimmen und die Versionen bei Bedarf zu mischen. Dieser Abschnitt stellt das im Rahmen dieser Arbeit entwickelte Konzept für Differenz- und Mischwerkzeuge für UML-Diagramme vor. Werkzeuge, die diesem Konzept entsprechen, nutzen die durch das im vorangegangen Abschnitt vorgestellte VM-System gelieferten Informationen, um die Differenzen und Konflikte einzelnen Konfigurationen zuzuordnen und somit dem Werkzeuganwender eine bessere Übersicht zu ermöglichen. Das Konzept umfaßt die Anzeige der Differenzen und Konflikte in erweiterten Diagrammtypen, die Interaktion der Werkzeuge mit den Anwendern und die Kooperation des VM-Systems mit den Differenz-/Mischwerkzeugen. Die Identifizierung von korrespondierenden Dokumentteilen und somit auch die Bestimmung der Differenzen basiert auf den Objekt- und Versionsidentifizierern des Editiermodells. Werkzeuge, die dieses Konzept umsetzen sind in der Werkzeugsammlung PISET integriert worden [231, 232].

1.5.1 Das Vereinigungsdokument

Die zentrale Problematik bei der Visualisierung der Differenzen liegt darin, unveränderte Diagrammelemente aus beiden Basisdiagrammen und Differenzen zwischen ihnen so darzustellen, daß die Entwickler diese leicht erkennen können. Der bei textuellen Dokumenten gebräuchliche Ansatz einer zweispaltigen Anzeige kann aufgrund der Dokument-Eigenschaften (siehe Abschnitt 1.2.4 und 5.1) nicht verwendet werden. Aus diesem Grund werden beide Diagramme überlagert dargestellt, was man als ein *Vereinigungsdokument* interpretieren kann.

Das Vereinigungsdokument enthält alle Diagrammelemente[17] beider Basisdiagramme. Korrespondierende Diagrammelemente, also Diagrammelemente, die beide Basisdiagramme gemeinsam besitzen, werden nur einmal gezeichnet und die basisdiagrammspezifischen Diagrammelemente werden unterschiedlich eingefärbt. Die Wahl der Farbe richtet sich dabei nach dem ursprünglichen Basisdiagramm dieses Elementes. Die gemeinsamen Elemente werden nur einmal gezeichnet. Abbildung 1.6 zeigt zwei Klassendiagramme und das dazugehörende Vereinigungsdiagramm.

Die Farben im Vereinigungsdiagramm kennzeichnen somit nicht, ob ein Diagrammelement erzeugt oder gelöscht wurde, sondern nur, zu welcher Diagrammversion es gehört. Die Aussage, ob ein Diagrammelement erzeugt oder gelöscht wurde, kann man nur dann treffen, wenn beide Basisdiagramme eine gemeinsame Vorgänger-Version besitzen und die bei der Differenz. B.rechnung mit berücksichtigt wurde. Diese Unterscheidung würde jedoch auch die Anzahl der benötigten Farben im Vereinigungsdiagramm von 3 auf 5 erhöhen, was die Lesbarkeit des Diagramms verschlechtern würde. Dabei stellt sich die Frage, wie groß der Nutzen dieser zusätzlichen Informationen wäre, da i.d.R. die Unterschiede zwischen den beiden Versionen von Interesse sind und nicht die Unterschiede beider Versionen gegenüber dem gemeinsamen Vorgänger.

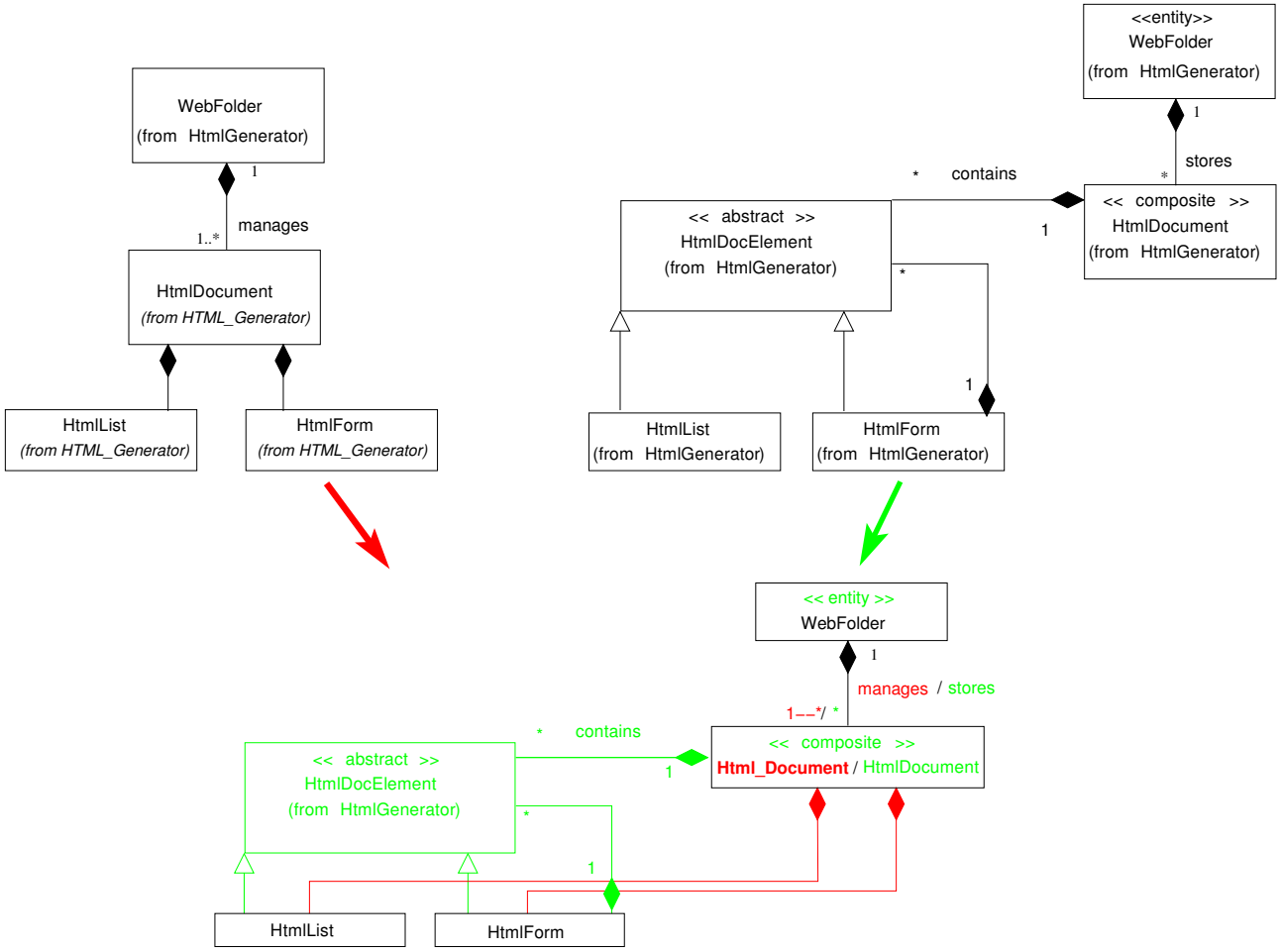


Abbildung 1.6: Vereinigungsdiagramm mit beiden Basisversionen

[17] Unter Diagrammelement verstehen wir jedes Element eines Diagramms aus dem Editiermodell, welches in dessen graphischer Darstellung gezeichnet wird.

Intra-Knoten Differenzen. Die feinkörnige Modellierung der Diagramme ermöglicht eine „feinkörnige Differenzberechnung“. D.h. man kann für jedes Element des Editiermodells bestimmen, in welcher Version des Diagramms es enthalten ist und ob einzelne Attribut-Werte eines Objekts oder eines Links unterschiedlich sind. Daher ist die Anzeige der Differenzen im Vereinigungsdiagramm nicht auf Knoten und Beziehungen beschränkt, sondern umfaßt auch Differenzen innerhalb der Knoten (*Intra-Knoten Differenzen*), oder Eigenschaften der Beziehungen zwischen den Knoten. Mögliche Arten von Intra-Knoten Differenzen sind geänderte Bezeichner (siehe Klasse `HtmlDocument` in Abbildung 1.6), neue/gelöschte Komponenten oder zwischen Knoten verschobene Komponenten. Ein Beispiel hierfür sind zwischen Paketen verschobene Klassen oder zwischen Klassen verschobene Attribute oder Methoden. Die Anzeige der Differenzen und eine Auflistung der möglichen Differenzen wird für die einzelnen UML-Diagrammtypen in Abschnitt 5.2 diskutiert.

Layout. Das Vereinigungsdiagramm enthält i.d.R. mehr Elemente als die beiden Basisdiagramme, so daß das Vereinigungsdiagramm ein anderes Layout besitzt als diese. I.d.R. sollte das jedoch kein Problem darstellen, da Entwickler meist an den Differenzen zwischen den Modellen und weniger an Differenzen im Layout interessiert sind. Sollten die Differenzen im Layout relevant sein, so kann die Technik des Vereinigungsdiagramms nicht verwendet werden.

Das Layout des Vereinigungsdiagramms sollte ähnlich zu dem Layout (mindestens) eines Basisdiagramms sein, da andernfalls die Entwickler keines der Basisdiagramme mehr wiedererkennen können. Das würde die Interpretation des Vereinigungsdiagramms erheblich erschweren.

Das Layout des Vereinigungsdiagramms kann man von dem Layout eines Basisdiagramms ableiten und um die zusätzlichen Elemente ergänzen. Die Berechnung eines optimalen Layouts ist ein eigenes Forschungsgebiet (z. B. [202]) und eine weitergehende Betrachtung würde den Rahmen dieser Arbeit übersteigen.

1.5.2 Gruppierung der angezeigten Differenzen

Der Nutzen des Vereinigungsdiagramms hängt entscheidend von dessen Übersichtlichkeit ab. Sind darin „zu viele“[18] Differenzen markiert, reduziert das die Übersichtlichkeit und somit den Nutzen deutlich[19].

Aufbauend auf das Editiermodell und die Informationen, die die Versionsverwaltung des OMS liefert, ist es möglich, die angezeigten Differenzen zu gruppieren. Eine Gruppierung auf Basis des Editiermodells faßt Differenzen, die einzelne Typen von Diagrammelementen betreffen, zusammen, wie z. B. Klassen, Methoden oder Beziehungen. Die Differenzen können aber auch anhand der Konfigurationen gebildet werden, zu denen die Objekt- oder Linkversionen gehören.

Zur Verbesserung der Übersichtlichkeit kann die Markierung einzelner Gruppen von Differenzen aufgehoben werden, indem sie nicht mehr farbig, sondern grau gezeichnet werden. Dadurch bleiben sie weiterhin erkennbar, lenken jedoch nicht mehr die Aufmerksamkeit der Entwickler auf sich. Abbildung 1.7 zeigt ein Werkzeug mit grau gefärbten Differenzen in einem Klassendiagramm.

[18] Ab wann ein Vereinigungsdiagramm zu viele Differenzen anzeigt und damit unbrauchbar wird, müßte gezielt in der Praxis untersucht werden. Der exakte Wert ist wahrscheinlich abhängig von den einzelnen Entwicklern, die ein Diagramm betrachten und deren Anforderungen an die Differenz-Anzeige. Diese Evaluation würde den Rahmen dieser Arbeit übersteigen.

[19] Ein vergleichbares Phänomen kennt man von Differenz-Werkzeugen für Texte, wenn die zu vergleichenden Dokumente nur noch wenige gemeinsame Textstellen aufweisen

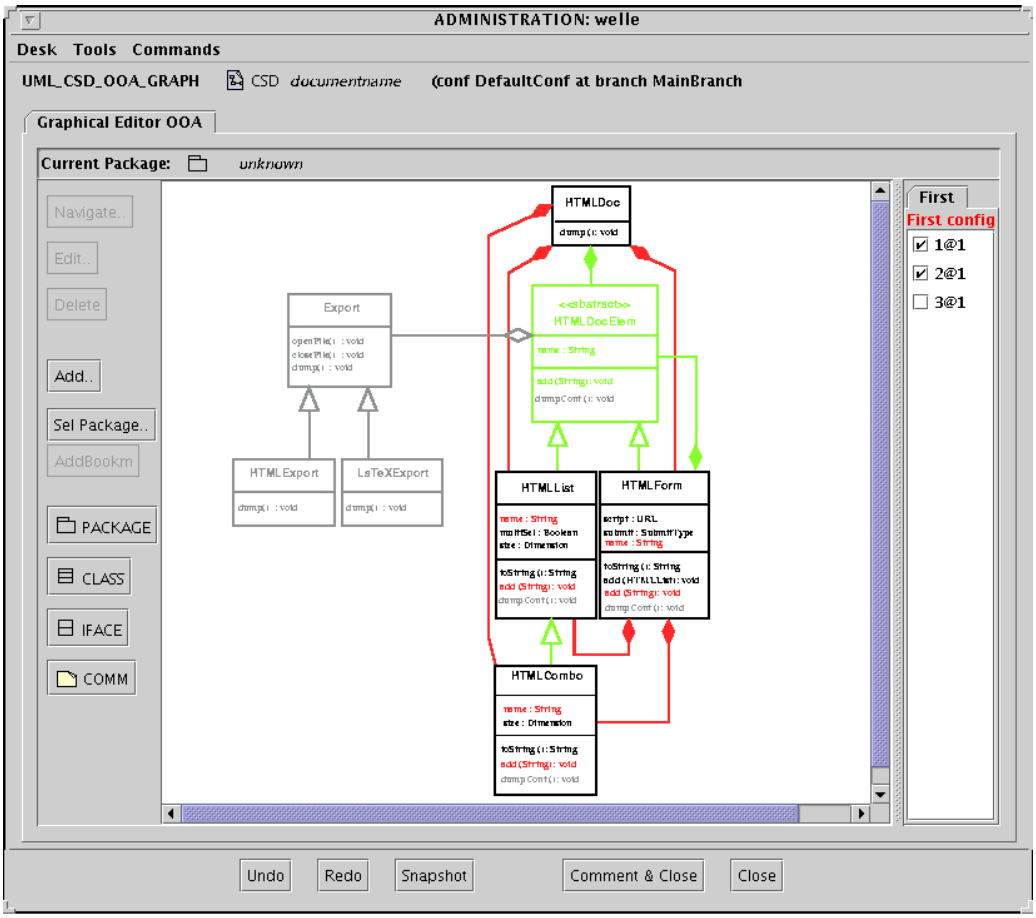


Abbildung 1.7: Werkzeug mit ausgeblendeten Differenzen

1.5.3 Berechnung der Differenzen

Die Editiermodelle der zu vergleichenden Diagrammversionen sind die Grundlage der Differenz. B.rechnung, die in drei Stufen verläuft:

1. Traversieren der Editiermodelle beider Diagramme
2. Zuordnung korrespondierender Objektversionen
3. Bestimmung der Unterschiede zwischen den Versionen

Zur Berechnung der Differenzen müssen die Editiermodelle abgeglichen und korrespondierende Objekte gefunden werden. Bei korrespondierenden Objekten handelt es sich um Versionen eines Objektes, die nicht notwendigerweise unterschiedlich sein müssen. Hierzu traversiert man parallel beide Syntaxbäume mittels der Breitensuche und vergleicht die Objekt-Identifizierer. Sind diese gleich, so hat man zwei korrespondierende Objektversionen gefunden. Die Differenzen[20] lassen sich dann durch einen einfachen Vergleich der Attribut-Werte und der ausgehenden Links bestimmen[21]. Zur Optimierung vergleicht man vorher die Versionsnummern. Wenn diese identisch sind, so sind es auch die Objekte, und somit ist der Vergleich der Objekte nicht notwendig.

[20] Abhängig von den durchgeführten Änderungen an einem Basisdiagramm, kann aus Benutzersicht ein anhand der Identifizierer als korrespondierend erkanntes Diagrammelement vollkommen unterschiedlich sein.

[21] Ohne die Identifizierer müßten die Objektversionen anhand einer gemeinsamen Menge von Attribut-Werten und Links bestimmt werden. Eine optimale Lösung ist unter diesen Voraussetzungen nicht möglich, da sich zwei Versionen eines Objekts erheblich unterscheiden können, so daß eine Zuordnung unmöglich werden kann.

Unter der Annahme, daß Objekte eher selten im Editiermodell verschoben werden, kann man die Suche nach korrespondierenden Objekten auf jeweils eine Ebene in den Teilbäumen des Spannbaums beschränken. Diese Annahme ist u.a. dadurch begründet, daß die Objekte nicht wahlfrei verschoben werden können. Das wird durch das Editier-Metamodell beschränkt. Klassen können zwar zwischen Paketen verschoben werden, jedoch kann eine Methode niemals direkte Komponente eines Pakets werden.
Verschobene Objekte lassen sich finden, indem man bei der Traversierung alle Objekte, für die kein korrespondierendes Objekt gefunden wurde, zwischenspeichert und nach der Traversierung untereinander noch einmal abgleicht. Wenn anschließend kein korrespondierendes Objekt gefunden wurde, so existiert dieses Objekt nur in einem Editiermodell.

Persistentes Vereinigungsdiagramm. Wie die so bestimmten Differenzen weiterverarbeitet werden, hängt von dem Werkzeugkonstruktionsansatz ab. I.d.R. müßte ein eigenständiges Werkzeug zur Anzeige des Vereinigungsdiagramms konstruiert werden, welches eng mit der Differenz. B.rechnung kooperiert, wenn diese nicht sogar vollständig in das Werkzeug integriert sein müßte.
Der Grund hierfür liegt darin, daß beide Diagramme geladen, die Differenzen zwischen ihnen bestimmt und als ein Vereinigungsdiagramm angezeigt werden müssen. Erzeugt man hingegen ein persistentes Vereinigungsdiagramm, in dem die Differenzen ebenfalls gespeichert sind, so läßt sich die Konstruktion des Werkzeugs deutlich vereinfachen. Die existierenden Werkzeuge müssen dann lediglich die zusätzlichen Daten laden, interpretieren und passend anzeigen. Technisch gesehen, sind die Werkzeuge zur Anzeige und zum Mischen der Differenzen Spezialisierungen der konventionellen Editoren, die um die zusätzlichen Funktionen erweitert sind.
Bei der Differenz. B.rechnung erzeugt man dann ein neues Diagramm, das persistente Vereinigungsdiagramm, basierend auf einem erweiterten Editier-Metamodell. In diesem sind zusätzlich die Differenzen gespeichert.

1.5.4 Mischen von Diagrammversionen

Im Gegensatz zur Differenz-Anzeige ist das Mischen von Versionen eine z.T. interaktive Tätigkeit, insbesondere um aufgetretene Mischkonflikte zu lösen.

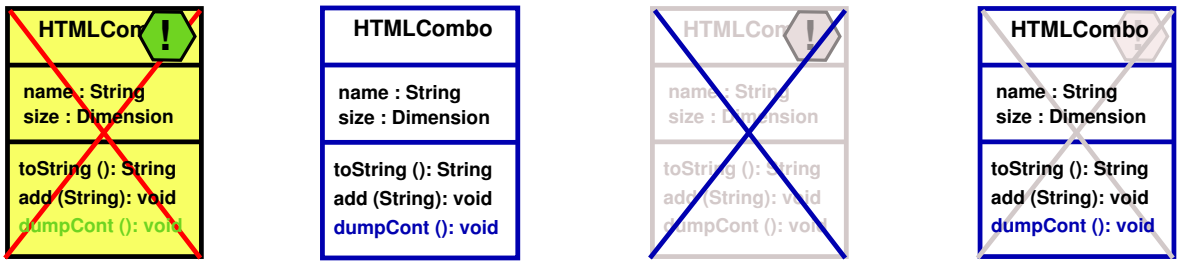
Die Mischstrategie. Sinnvoll ist es, die Benutzerinteraktion auf Basis der graphischen Darstellung und das Mischen auf den Editiermodellen der Diagramme zu realisieren. Das erfordert spezielle Mischwerkzeuge, die nach folgender Mischstrategie arbeiten:

1. vorläufige Mischversion erstellen
2. Konflikte manuell lösen
3. endgültige Mischversion erstellen

Die vorläufige Mischversion eines Diagramms (im weiteren als *Pre-Mischversion* bezeichnet) ist vergleichbar mit dem Vereinigungsdiagramm. Sie unterscheidet sich jedoch in zwei wesentlichen Aspekten: 1. sie basiert auf den beiden zu mischenden Versionen einschließlich der gemeinsamen Vorgängerversion (*3-Wege-Mischen*) und 2. für die Differenzen, die nur auf Änderungen zwischen der Vorgängerversion und *einer* Basisversion zurückzuführen sind, wurde bereits automatisch eine (vorläufige) Mischentscheidung getroffen. Die Pre-Mischversion dient weiterhin als Ausgangspunkt für die Entwickler, die darauf aufbauend die Konflikte lösen und (automatisch oder manuell) getroffene Mischentscheidungen ändern können.

Mögliche Arten von Konflikten sind einerseits konkurrierende Änderungen eines Objekt- oder Link-Attributs, andererseits das Löschen eines Teilbaums des Syntaxbaums in einer Version und eine beliebige Änderung am Teilbaum der anderen Diagrammversion.
Alle getroffenen Entscheidungen zum Lösen von Konflikten werden nicht sofort im Editiermodell nachvollzogen, sondern erst im Werkzeug angezeigt und im Editiermodell als Anweisung notiert. Nachdem alle Konflikte gelöst wurden, wird die endgültige Mischversion erzeugt. Das bietet die Möglichkeit, einmal getroffene Entscheidungen einfach wieder rückgängig zu machen und den Konflikt durch Wahl der anderen Änderung zu lösen.

Die Pre-Mischversion. Die Pre-Mischversion ist eine durch Anwendung des 3-Wege-Mischverfahrens entstandene Version. Alle darin enthaltenen automatisch getroffenen Mischentscheidungen sind durch den Anwender änderbar. Die Pre-Mischversion wird als Diagramm angezeigt, in dem die vorläufigen Mischentscheidungen und die noch zu lösenden Konflikte markiert sind.
Alle automatisch gemischten Differenzen sind blau gezeichnet, Konflikte werden ähnlich wie Differenzen in Vereinigungsdiagrammen markiert. Im Unterschied zum Vereinigungsdiagramm ist es hier wichtig, zwischen erzeugten und gelöschten Diagrammelementen zu unterscheiden. Daher werden gelöschte Diagrammelemente durchgestrichen gezeichnet. Liegt ein Konflikt für ein in einer Version gelöschtes Diagrammelement vor, wird dieses mit einem zusätzlichen Hinweis-Symbol dargestellt, welches auf die in Konflikt stehende Änderung hinweist. In Abbildung 1.8(a) wurde die Klasse in einer Diagrammversion gelöscht und in der anderen Version wurde eine neue Methode hinzugefügt. Die Farbe der Linien, mit denen das Diagrammelement durchgestrichen ist, gibt die Diagrammversion an, in der das Element gelöscht wurde. Die Darstellung von manuell gelösten Konflikten unterscheidet sich nicht von der Darstellung automatisch gemischter Differenzen (siehe Abbildung 1.8(b): die neue Methode wurde gewählt und somit die Klasse nicht gelöscht).



(a) ungelöster Konflikt (b) gelöster Konflikt (c) Anzeige des früheren Konflikts nachdem Löschen (d) Anzeige des früheren Konflikts nach Wahl der Änderung

Abbildung 1.8: Darstellungsarten eines Konflikts

In einigen Fällen ist es notwendig, bereits getroffene Misch-Entscheidungen, sei es die automatisch oder die manuell getroffenen, wieder abzuändern. Ein Beispiel könnte das Lösen eines Konflikts sein, der es erfordert, eine bereits vorher gelöschte Klasse doch nicht zu löschen, sondern die hinzugefügte Methode zu verwenden. Um das zu ermöglichen, ist die Anzeige der gelösten Konflikte notwendig. Hierfür sollte ein Mischwerkzeug eine Funktion anbieten, um zwischen der Ansicht, in der nur die gewählte Konfliktlösung zu sehen ist und der Anzeige einschließlich der alternativen Änderung zu wechseln. Die jeweils nicht gewählte Änderung wird

grau gezeichnet. Die Abbildungen 1.8(c) (Lösung des Konflikt durch Löschen der Klasse) und 1.8(d) (Konfliktlösung durch Auswahl der Änderung und damit Erhalt der Klasse) zeigen das beispielhaft.

Interaktives Lösen von Konflikten. Die Pre-Mischversion dient nicht ausschließlich zur Anzeige der Konflikte, sondern auch um diese zu lösen und evtl. getroffene Entscheidungen wieder abzuändern. Eine Problematik, die sich beim manuellen Lösen von Konflikten ergibt, ist, daß die Anzahl der Konflikte stark davon abhängt, in welchem Ausmaß sich die Versionen unterscheiden. Je stärker die zu mischenden Versionen voneinander abweichen, um so mehr Konflikte treten i.d.R. auf. Folgende Möglichkeiten gibt es unter Verwendung der durch das VM-System gelieferten Daten:

- Konflikte einzeln lösen
- eine Diagrammversion priorisieren und Konflikte zusammen lösen
- Konflikte gruppieren und gruppenweise lösen

Die aus konventionellen Mischwerkzeugen bekannte Methode, jeden Konflikt einzeln zu lösen, funktioniert bei einer kleinen Anzahl an Konflikten. Bei einer großen Anzahl ist das eine langwierige und fehleranfällige Aufgabe.
Die Konfliktlösung, die eine Diagrammversion priorisiert und somit alle Konflikte zu Lasten der anderen Version löst, bietet sich an, wenn alle Änderungen der priorisierten Version in der Mischversion enthalten sein müssen. In allen anderen Fällen würden zu viele Konflikte auf die falsche Art gelöst.
Durch die Nutzung weiterer Informationen des VM-Systems ist die Gruppierung der Konflikte anhand der angelegten Konfigurationen möglich, zu der die an einem Konflikt beteiligten Versionen von Objekten und Links gehören. Das Lösen der Konflikte, indem die Versionen, die zu einer Konfiguration gehören, gewählt werden, ist unter der Annahme sinnvoll, daß in einer Konfiguration zum größten Teil nur Versionen gespeichert sind, die im Kontext einer (Teil-)Aufgabe angelegt wurden.

1.6 Zusammenfassung

In dieser Arbeit wird ein Konzept zur Versionierung von Dokumenten aus den frühen Phasen der Softwareentwicklung vorgestellt, insb. zur Versionierung von UML-Diagrammen. Besonderes Augenmerk wird dabei auf die Struktur der Dokumente, die Einbindung der Versionierung in die CASE-Werkzeuge und auf die Unterstützung kooperativer Arbeit gelegt. Ergänzend wird ein Konzept für Differenz- und Mischwerkzeuge für Diagramme vorgeschlagen, welches eng mit dem VM-System zusammenarbeitet und somit mehr Informationen nutzen kann, um einerseits dem Anwender eine bessere Übersicht zu ermöglichen, anderseits um Mischwerkzeuge komfortabler zu gestalten.
Eine wichtige Erkenntnis, die im Rahmen dieser Arbeit gewonnen wurde, ist, daß die Dokumente in unterschiedlichen Repräsentationen vorliegen, die jedoch nicht alle in gleicher Weise zur Versionierung geeignet sind. Bei strukturierten Dokumenten eignet sich das Editiermodell am besten, da die Syntax bekannt ist und sich somit korrespondierende Modellelemente leichter zuordnen lassen.

Die Grundlage der Arbeit bildet das feinkörnige Versionsmodell, welches die Dokumente in deren Repräsentation als Editiermodell versioniert und die OMS-orientierte Werkzeugarchitektur mit der Multiple-View-Integration berücksichtigt. Eine zentrale Frage im Zusammenhang

mit der Multiple-View-Integration betrifft die Sichtbarkeit der einzelnen Objekte im OMS. Obwohl nicht alle Objekte in einem Werkzeug in Abhängigkeit von dessen Sicht zugreifbar sind, muß die Konsistenz des bearbeiteten Dokumentes sichergestellt werden. Die Lösung besteht darin, nur von veränderten Objekten eine neue Version anzulegen und die Konsistenz durch die Einführung von Konfigurationen, also einer Menge von Objektversionen, sicherzustellen. Diese ist unabhängig von der Sicht des Werkzeugs.
Integriert man bei der technischen Umsetzung das VM-System und den Transaktionsmanager, bietet das folgende Vorteile:

1. *Einfache Integration des VM-Systems in die CASE-Werkzeuge:* Zur Synchronisation konkurrierender Zugriffe verwendet man in der OMS-orientierten Werkzeugarchitektur Transaktionen. Durch deren Erweiterung um die Versionierungsfunktionalität müssen die Werkzeuge lediglich zusätzlich die Auswahl der Basiskonfiguration unterstützen. Es sind sonst keine anderen Erweiterungen notwendig.

2. *Weitgehende Automatisierung beim Anlegen von Versionen und Konfigurationen*: Die manuelle Versionierung wäre bei der feinkörnigen Modellierung zu aufwendig. Bei jedem Schreibzugriff im Rahmen einer Transaktion wird gleichzeitig eine neue Version angelegt. Durch das Anlegen einer Konfiguration durch jede Transaktion erhält man zusätzlich einen Undo-Mechanismus, der transaktionsübergreifend arbeitet. Zum Rücksetzen einer abgeschlossenen Transaktion setzt man auf eine ältere Konfiguration auf.

3. *Direkte Kooperation von Anwendern*: Zwei Transaktionen können auf die selbe Konfiguration aufsetzen und somit eine direkte Kooperation von mehreren Entwicklern ermöglichen. Die Konsistenz wird durch das feinkörnige Sperrmodell sichergestellt. Die Kooperation verringert die Anzahl an Varianten, die wieder gemischt werden müssen.

Erweitert man das Konzept um Entwurfstransaktionen, so lassen sich die Konfigurationen besser strukturieren, indem sie den Entwurfstransaktionen zugeordnet werden. Wenn die Entwurfstransaktionen genutzt werden, um einzelne Aufgaben im Rahmen des Softwareentwicklungsprozesses zu bearbeiten, so erreicht man auch eine Verknüpfung der Konfigurationen zu der korrespondierenden Aufgabe. Die Auswahl der Dokumentversion ist dann durch Wahl der Aufgabe, also durch Wahl einer Entwurfstransaktion möglich. Die Berücksichtigung der Aufgaben und der darauf basierenden Versionsauswahl entspricht eher der Denkweise von Entwicklern als die Versionsauswahl anhand eines Versionsidentifizierers ohne Aussagekraft.

Im Gegensatz zur Versionierung, die sich nur auf die Syntax-Bäume der Diagramme beschränkt, wird bei der Differenzberechnung und -Anzeige sowie beim Mischen von Differenzen auf zwei Abstraktionsebenen der Diagramme gearbeitet. Zur Berechnung wird weiterhin das Editiermodell verwendet. Entwickler interessieren sich jedoch nicht für die Differenzen zwischen zwei Editiermodellen, sondern zwischen den Diagrammen, so daß zur Anzeige und zum Mischen die graphische Darstellung genutzt wird. Das ist ein entscheidender Unterschied zu existierenden Werkzeugen, die sich ausschließlich auf eine Abstraktionsebene beschränken.
Ein weiterer Unterschied zu konventionellen Differenz- und Mischwerkzeugen ist die Möglichkeit, die angezeigten Differenzen und Konflikte anhand von Versionsinformationen zu gruppieren und die Anzeige auf ausgewählte Gruppen zu beschränken. Damit können die Entwickler selbst entscheiden, an welchen Differenzen und Konflikten sie interessiert sind.
Durch die persistente Realisierung der Differenz- und Pre-Misch-Diagramme kann die OMS-orientierte Werkzeugarchitektur auch für die Differenz- und Mischwerkzeuge genutzt werden, so daß beim Mischen auch mehrere Entwickler kooperieren können.

1.7 Gliederung der Arbeit

In Kapitel 2 diskutieren wir existierende Konzepte der Versionsverwaltung, der Differenzanzeige und des Mischens unter Berücksichtigung der Eigenschaften von Softwaredokumenten. Desweiteren geben wir einen Überblick der Funktionen und Konzepte von H-PCTE. Das in diesem Kapitel überlicksartig vorgestellte Versionsverwaltungskonzept wird in Kapitel 3 detailliert behandelt unter Berücksichtigung der kooperativen Arbeit an Dokumenten. Aspekte, die die Realisierung und Umsetzung des Konzepts am Beispiel H-PCTE betreffen, diskutiert Kapitel 4. Kapitel 5 vertieft das Konzept für Differenz- und Mischwerkzeuge für UML-Diagramme, beginnend mit einer Diskussion von Darstellungsmöglichkeiten von Differenzen zwischen Versionen eines UML-Diagramms. Fragen, die die Realisierung des Differenz- und Mischkonzepts betreffen, diskutieren wir in Kapitel 6. Abschließend gibt es in Kapitel 7 eine Zusammenfassung der Arbeit und einen Ausblick auf Fragen, die einer weiteren Forschung bedürfen.

Kapitel 2

Hintergrund

Dieses Kapitel gibt eine Übersicht der dieser Arbeit zugrunde liegenden Konzepte und Techniken. Die Betrachtung ist untergliedert in die Teilgebiete des Software-Konfigurationsmanagements (Abschnitt 2.1) und in die Bestimmung und das Mischen von Differenzen zwischen Dokumenten (Abschnitt 2.2). Abschließend wird eine Übersicht über PCTE, H-PCTE und PI-SET gegeben (Abschnitt 2.3).

Das Gebiet des Software-Konfigurationsmanagements umfaßt viele Teilgebiete, wie z. B. Versionsverwaltung, Änderungsmanagement, Team-Unterstützung oder Prozeß-Unterstützung. Nicht alle diese Teilgebiete sind für diese Arbeit relevant. Daher beschränken wir die Übersicht auf folgende Teilgebiete. In Abschnitt 2.1.1 geben wir eine Übersicht der wichtigsten Konzepte zur Versionsverwaltung. Die Konsistenz von zusammengehörenden Versionen wird durch Konfigurationen sichergestellt, die wir in Abschnitt 2.1.2 vorstellen. Benutzungskonzepte stellen wir in Abschnitt 2.1.3 vor und Kooperationskonzepte in Abschnitt 2.1.4. Abschließend stellen wir Konzepte zur Realisierung von Versionsverwaltungs-Systemen in Abschnitt 2.1.5 vor.

Die Anzeige und das Mischen von Differenzen teilt sich auf in die Anzeige der Differenzen (Abschnitt 2.2.1), deren Berechnung (Abschnitt 2.2.2) und das anschließende Mischen (Abschnitt 2.2.3).

Die Übersicht der Konzepte und Techniken beenden wir mit der Vorstellung der grundlegenden Konzepte von PCTE, die Erweiterungen, die durch H-PCTE eingeführt wurden, und eine Übersicht der Werkzeugsammlung PI-SET, die die Dienste von H-PCTE verwendet und das als Grundlage zur Evaluation der in dieser Arbeit vorgestellten Konzepte zur Versionierung von UML-Dokumenten, der Differenzberechnung und -Anzeige sowie dem Mischen dient.

2.1 Software-Konfigurationsmanagement

Software-Konfigurationsmanagement (SKM) beschäftigt sich mit der Verwaltung von Software-Komponenten und mit den Änderungen an ihnen [90]. Es gibt viele unterschiedliche Konzepte und Systeme, die die Aufgaben im Bereich des SKM auf die verschiedensten Arten angehen. Einige Konzepte sind noch im Stadium der Erprobung und Erforschung, andere haben eine hohe Praxisrelevanz erlangt [89]. Die praxisrelevanten Konzepte werden an Hochschulen gelehrt [17, 70, 128, 127, 126]. Die Konzepte und Techniken des SKM lassen sich im Stil von Entwurfsmustern [91] klassifizieren, wie Tichy und Hunt erläutern [105]. Einen guten Überblick der Konzepte findet man in [27, 58, 233].
Dart [68] hat ein Schema entwickelt, nachdem sich die Aufgaben und Anforderungen an das SKM einteilen lassen (siehe Abbildung 2.1). Sie unterscheidet dabei primär zwei Anwendungsgebiete:

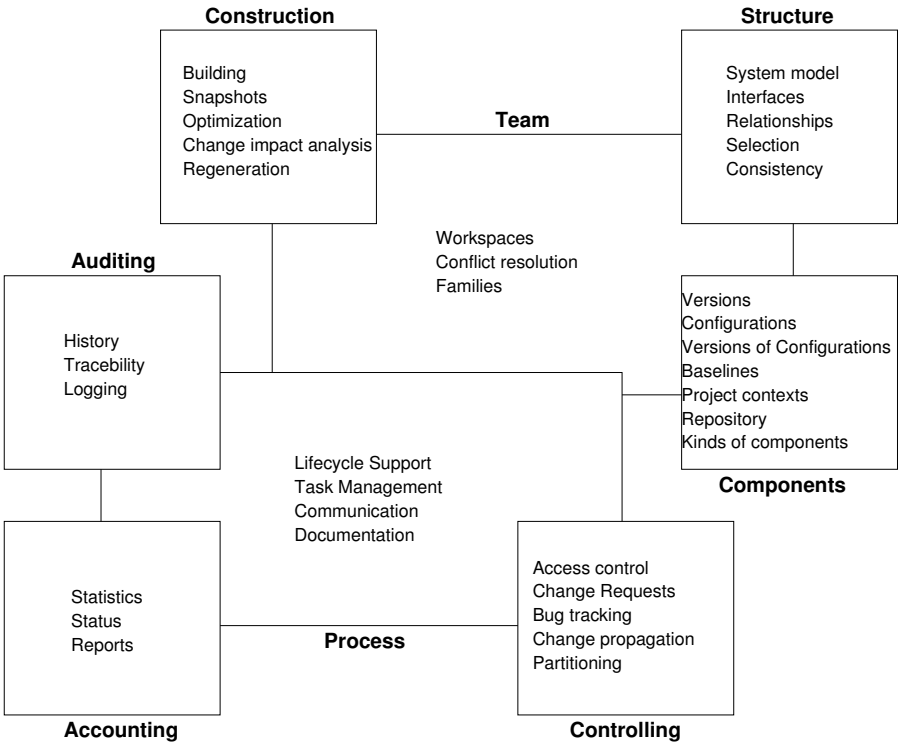


Abbildung 2.1: SKM Funktionalitätsanforderungen (aus [68])

1. **Prozeß-Unterstützung:** Dieser Bereich deckt Management-Aspekte ab, weshalb Westfechtel und Conradi [58] ihn auch als Management-Support bezeichnen. Insbesondere fallen folgende Aufgaben in diesen Bereich, die z.T. auch im Bereich des Projekt-Managements (z. B. [90, 101]) oder der Prozeßmodellierung (z. B. [208, 81]) anzusiedeln sind:

   - Durchführung von Änderungen anhand eines strikten Vorgehensmodells: Es wird exakt festgelegt welcher Projektteilnehmer auf welche Dokumente in welcher Form (lesen, ändern oder neu erstellen) Zugriff hat. Hierzu zählen auch Fragen des Workflow, also welcher Projektteilnehmer ein Dokument als nächstes bearbeiten muß, z. B. muß ein geändertes Dokumente erst zur Qualitätssicherung, bevor es freigegeben wird.
   - Änderungsmanagement (engl. change management): Verwaltung von zusätzlichen Anforderungen an ein Produkt, von Fehlerbeschreibungen und den dazu gehörigen Änderungsaufgaben einschließlich der Zuordnung an die Entwickler(-gruppen).
   - Qualitätssicherung: Die Einhaltung interner und externer Qualitätsanforderungen an die Dokumente, z. B. Vorgaben an Form und Inhalt der Spezifikationen oder Testfälle für einzelne Klassen oder das gesamte Produkt.

2. **Team-Unterstützung:** Die kooperative Softwareentwicklung innerhalb von Entwicklergruppen erfordert die Koordination und Konsistenzsicherung der zu bearbeitenden Dokumente. Die Komplexität dieser Aufgaben steigt mit zunehmender Gruppen- und Projektgröße, so daß eine Werkzeugunterstützung notwendig ist. Westfechtel und Conradi [58] bezeichnen diesen Bereich daher auch als Development-Support. Man findet u.a. folgende Funktionen:

   - Versionsverwaltung: Hierzu zählen alle Eigenschaften und Funktionen zum Anlegen, Löschen und Ändern von Versionen, einschließlich des Zugriffs auf diese.

- Konfigurationsverwaltung:
  - Sicherung existierender Zusammenstellungen von Versionen eines Produktes
  - Wiederherstellung älterer Konfigurationen
  - Zusammenstellung neuer Konfigurationen anhand von Beschreibungen in Form von Regeln, die die Eigenschaften der gewünschten Konfiguration angeben.
- Buildmanagement: Erstellen automatisch erzeugbarer Dokumente (z. B. Objektdateien) aus Quell-Elementen (z. B. Quelltext)

Das in dieser Arbeit vorgestellte Versionierungskonzept konzentriert sich auf Aspekte der Team-Unterstützung, so daß wir uns im folgenden auf eine Diskussion von Konzepten, die diesem Bereich zuzuordnen sind, beschränken. Für Konzepte, die die Prozeß-Unterstützung adressieren, sei z. B. auf [11, 14, 78] verwiesen.

2.1.1 Versionsverwaltung

Es gibt unterschiedliche Sichten auf die Struktur versionierter Software, also die Menge der in einem Softwareentwicklungsprozeß erstellten Dokumente, Teil-Dokumente oder Objekte. Im folgenden sprechen wir allgemein von Elementen. Conradi und Westfechtel [58] unterscheiden zwei orthogonale Sichten:

1. Der *Produkt-Raum* beschreibt die Beziehungen zwischen *allen* (unversionierten) Elementen. Hierzu zählen Abhängigkeits- und Kompositionsbeziehungen zwischen einzelnen Klassen und/oder zwischen Diagrammen aus der Analyse- und Entwurfsphase. Die Existenz von Versionen bleibt hierin unberücksichtigt. Jedes Element kann eindeutig identifiziert werden, z. B. durch eine Objekt-ID die das OMS vergibt.

2. Der *Versionsraum* beschreibt hingegen die Beziehungen zwischen den Versionen *eines* Elementes. Die einzelnen Versionen unterscheidet man anhand eines Versionsidentifizierers. Verwaltet ein Repository mit Versionsierungsfunktionalität die Elemente, so sind Produkt- und Versionsraum darin integriert.

   Anhand der Struktur des Versionsraums kann man zwei Arten von Versionen unterscheiden:

   (a) Eine Folge von Versionen dokumentiert die Änderungsgeschichte eines Elementes. Eine einzelne Version hiervon bezeichnet man als *Revision*.

   (b) Eine *Variante* bezeichnet zwei oder mehr parallel existierende Zustände eines Elementes zu einem Zeitpunkt. Diese können infolge einer konkurrierenden Bearbeitung auftreten oder durch Anpassen an unterschiedliche Anforderungen, wie zum Beispiel durch die Optimierung eines Algorithmus/Schaltkreises auf Speicher-/Flächenbedarf oder Laufzeitverhalten. Jede Variante kann dabei in mehreren Revisionen vorliegen, wie in Abbildung 2.2 beispielhaft dargestellt.

   Mahler [156] unterscheidet folgende Arten von Varianten:

   - zeitlich begrenzt existierende vs. permanente Varianten : Einige Varianten existieren nur einen begrenzten Zeitraum, innerhalb dessen sie wieder mit anderen Varianten gemischt werden. Diese Varianten existieren primär, um Wartezeiten bei konkurrierender Arbeit zu vermeiden.
   Die permanenten Varianten existieren über einen längeren Zeitraum und modellieren unterschiedliche Versionen eines Produktes, z. B. zwei ausgelieferte Versionen einer Bürosoftware.

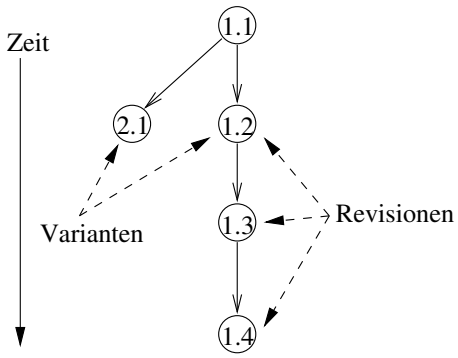


Abbildung 2.2: Beispiel eines Versionsgraphen mit Revisionen und Varianten

- **Aggregat-Varianten** : Aggregate können aus unterschiedlichen Komponenten bestehen, in Abhängigkeit von den Anforderungen. Beispiele sind Programmpakete wie Bürosoftware, die in verschiedenen Varianten vorliegen.
- **abgeleitete Varianten** : Von einem gemeinsamen Quelltext auf unterschiedlichen Rechnerplattformen erzeugte Binärdateien. Beispielsweise kann eine Version einer Bürosoftware für unterschiedliche Betriebssysteme übersetzt werden.
- **mehrfache Varianz** : Versionierte Elemente variieren i.d.R. nicht nur in einem Aspekt, sondern in mehreren, die nicht voneinander abhängig sein müssen. Beispielsweise wird eine Bürosoftware in unterschiedlichen Zusammenstellungen für unterschiedliche Betriebssysteme ausgeliefert. Einerseits variieren die einzelnen Komponenten dahingehend, daß diese an die besonderen Eigenschaften der einzelnen Betriebssysteme angepaßt werden müssen und andererseits variieren Komponenten der graphischen Bedienschnittstelle dahingehen, daß sie unterschiedliche Funktionalitäten anbieten müssen, die dann entsprechend in der Software integriert ist.

Durch die Erweiterung des Repositories H-PCTE um Versionierungsfunktionalität, sind darin Produkt- und Versionsraum integriert. Das muß beim Zugriff auf die Dokumente, die als Objektgraphen modelliert sind, und somit bei der Erweiterung der Programmierschnittstelle berücksichtigt werden. Durch den Schwerpunkt der Team-Unterstützung, sind insbesondere zeitlich begrenzt existierende und permanente Varianten sowie Revisionen, also die Struktur des Versionsraums, von Interesse.

2.1.1.1 Zustandsbasierte vs. änderungsbasierte Versionierung

Die Struktur des Versionsraums ist von der Art der Versionierung abhängig. Liegt eine zustandsbasierte Versionierung vor, so läßt sich der Versionsraum durch Versionsgraphen (siehe Abbildung 2.2) darstellen. Jeder Knoten beschreibt eine einzelne Version eines versionierten Elementes, also dessen Zustand zu einem bestimmten Zeitpunkt. Die Versionen können beschreibende Attribute besitzen, wie z. B. den Namen des Entwicklers, der die Version angelegt hat, oder einen abstrakten Bezeichner. Die abstrakten Bezeichner können zusammengehörige Versionen mehrerer Elemente kennzeichnen.
Zustandsbasierte Versionsverwaltungs-Systeme gibt für für verschiedene Anwendungsbereiche. Viele SKM-Systeme sind spezialisiert auf die Versionierung von Textdateien (z. B. SCCS [195, 194], RCS [219], CVS [36, 24]) oder beliebigen Dateien wie n-DFS [88], wobei teilweise die Verzeichnisstruktur versioniert wird. Beispiele hierfür sind ClearCase [143, 236] und NSE [61, 164]. n-DFS und ClearCase kann man auch als virtuelle Dateisysteme bezeichnen,

da sie ein Dateisystem simulieren, auf dem die Werkzeuge arbeiten und die Daten speichern. Anderen zustandsbasierten Versionsverwaltungs-Systemen liegt eine objektorientierte Datenbank zugrunde, in der sie alle Daten abgelegen, wie z. B. ADELE [79], DAMOKLES [71, 2, 1], GOBE [16], GRAS [233], MS-Repository [23] PACT [182] oder PCTE [109, 110]. Die Einstatzbereiche dieser Systeme liegen z. B. in der Versionierung von Diagrammen aus der Softwareentwicklung [191] oder im Bereich des CAD [190, 117]. Neben objektorientierten Datenbanken wurden auch deduktive Datenbanken [134] und funktionale Objekt-Systeme (CLOS, Common LISP Object System) [168, 118] um Versionierungsfunktionalität erweitert.

Der Versionsraum von änderungs- und operationsbasierten SKM-Systemen läßt sich ebenfalls als Graph darstellen. Im Unterschied zur zustandsbasierten Versionierung verwaltet das VM-System nicht die Versionen der Elemente, sondern die Beziehungen zwischen den Knoten, die den durchgeführten Änderungen entsprechen. Eine Ausnahme stellt der Wurzelknoten dar, der die Ausgangsversion eines Elementes repräsentiert, auf die sich die Änderungen beziehen. Bei dateibasierten SKM-Systemen beschreiben die Änderungen die geänderten Zeilen in den einzelnen Dateien. Eine Menge von zusammengehörenden Änderungen in allen Dateien (eine Kante im Graphen) wird auch als *Change-Set* [229] bezeichnet. Um eine Version eines Elementes zu erhalten, kombiniert man alle gewünschten Change-Sets mit der Basisversion.

Die Change-Sets sind eine Methode, um logisch zusammengehörende Änderungen einzelner Zeilen von verschiedenen Dateien zusammenzufassen. Eine andere Art der änderungsbasierten Versionierung sind *Change-Packages* [229]. Ein Change-Package verwaltet, im Gegensatz zu Change-Sets, die Versionen der geänderten Dateien und faßt diese zu einer logischen Änderung zusammen. Man könnte diese Art der Versionierung als eine erweiterte Form der zustandsbasierten Versionierung unter Verwendung von abstrakten Bezeichnern verstehen, da die Versionen und nicht die Änderungen verwaltet werden. Jedoch ordnet Weber [229] sie der änderungsbasierten Versionsverwaltung mit der Begründung zu, daß die VM-Systeme nicht die Versionen, sondern nur die Unterschiede zwischen den Versionen als Delta speichern. Berücksichtigt man die internen Deltas[1] der VM-Systeme, so gruppiert ein Change-Package eine Menge von Deltas, und gehört daher zu den änderungsbasierten VM-Systemen. Die änderungsbasierte Versionierung ist nicht auf dateibasierte VM-Systeme beschränkt, sie läßt sich auch in Datenbanken mit Versionierungsfunktionalität realisieren, wie z. B. in [146] und [169].

Beispiele für änderungsbasierte VM-Systeme die Change-Sets verwenden sind forschungsorientierte Systeme wie EPOS [97, 96], DaSC [152] oder PIE [94]. Abbildung 2.3 zeigt ein Beispiel für Änderungsebenen in PIE, die einer Change-Set entsprechen. Die resultierende Version ergibt sich aus der Kombination der Änderungsebenen. Ein Beispiel für ein kommerzielles System ist Aide-de-Camp (heute ADC/Pro) [64].

Der Vergleich von Change-Sets und Change-Packages macht deutlich, daß die Versionierungs-Strategie von der Speicherung der Versionen unabhängig ist. NUCM [104] trennt die Versionsspeicherung von der Versionierungs-Strategie. Auf einer einheitlichen Speicherungsmethode können in NUCM verschiedene Strategien implementiert werden, wie z. B. das Check-Out/Check-In-Modell oder auch Change-Sets.

Ein Vergleich der zustands- und änderungsbasierten VM-Systeme zeigt die unterschiedlichen Vor- und Nachteile beider Versionierungs-Strategien [56]. Bei der zustandsbasierten Versionierung läßt sich der Versionsgraph der einzelnen versionierten Elemente leicht bestimmen, jedoch muß die Konsistenz mit erhöhtem Aufwand sichergestellt werden, z. B. durch Change-Packages. Im Gegensatz hierzu ist bei der änderungsbasierten Versionierung der Versionsgraph nur mit erhöhtem Aufwand zu bestimmen. Die Konsistenz der Versionen innerhalb eines Change-Sets

[1] Delta wird hier im Plural verwendet, da nicht eine einzelne Datei gemeint ist, sondern mehrere zusammengehörende Dateien.

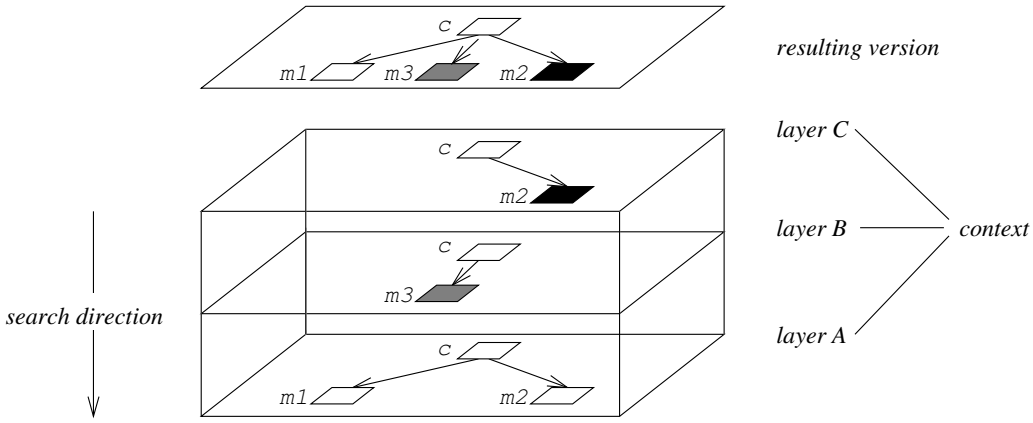


Abbildung 2.3: Änderungsbasierte Versionierung in PIE (aus [55])

wird durch die Strategie selbst sichergestellt, die Konsistenz zwischen verschiedenen Change-Sets muß man explizit sicherstellen.

Durch das Zusammenfassen von zusammengehörigen Änderungen entspricht die änderungsbasierte Versionierung eher der Denkweise von Entwicklern. Dies erschwert aber auch die Bestimmung von Änderungen zwischen zwei Versionen *eines* Elementes, da alle Change-Sets nach Änderungen an dem betrachteten Element durchsucht und anschließend zusammengefaßt werden müssen.

Ein weiterer Unterschied betrifft die Flexibilität bei der Zusammenstellung von Versionen. Bei der zustandsbasierten Versionierung kombiniert man die Versionen der einzelnen Elemente zu einem gemeinsamen Produkt, man muß jedoch auf eine konsistente Auswahl der Element-Versionen achten. Insgesamt ergeben sich v^m Kombinationsmöglichkeiten, mit $v :=$ Anzahl der Versionen und $m :=$ Anzahl der versionierten Elemente. Bei der änderungsbasierten Versionierung kombiniert man die Änderungen zu einem Produkt, wobei es zu berücksichtigen gilt, daß nicht alle Kombinationen ein konsistentes Produkt ergeben. Hierbei gibt es 2^v Kombinationsmöglichkeiten.

Zusammenfassend läßt sich sagen, daß beide Versionierungs-Strategien ihre spezifischen Vor- und Nachteile besitzen und die Wahl der Methode abhängig vom Anwendungsbereich ist. Bei feinkörnig modellierten Dokumenten benötigt man Unterstützung bei der Konsistenzsicherung der einzelnen Versionen. Der Einsatz eines änderungsbasierten VM-Systems wäre aus dieser Sicht sinnvoll. Nachteilig wirkt sich jedoch die benötigte Zeit zur Rekonstruktion der Versionen aus, da bei deren Zugriff, die einzelnen Change-Sets auf die Basisversion angewendet werden müssen. Bei interaktiven Werkzeugen, worunter UML-Diagrammeditoren fallen, würde das zu nicht vernachlässigbaren Verzögerungen führen. Da die Reaktionszeit eines interaktiven Werkzeugs aus Anwendersicht ein entscheidender Faktor ist, ist eine zustandsbasierte VM-Strategie vorzuziehen. Einen Mittelweg stellen die Change-Packages dar, die einerseits die Konsistenz sicherstellen, andererseits auch für feinkörnig modellierte Dokumente anwendbar sind. Bei genauer Betrachtung entspricht ein Change-Package einer expliziten Konfiguration, die wir in Abschnitt 2.1.2.2 vorstellen.

2.1.1.2 Identifizierung von Versionen

Bei den bisherigen Betrachtungen haben wir zur Unterscheidung von einzelnen Versionen nur allgemein von Versionsidentifizierern gesprochen. Die konkrete Ausprägung eines Versionsidentifizierers ist von der Definition des Versionsraums V abhängig. Diese legt auch fest, ob der

Identifizierer eine oder mehrere Versionen kennzeichnet und wie auf die Version(en) zugegriffen wird. Man unterscheidet zwei Definitionen für V:

- **Extensional Versioning**: $V = \{v_1, v_2, \ldots, v_n\}$
- **Intensional Versioning**: $V = \{v|c(v)\}$, c die Regeln zur Konstruktion von Versionen

Beim Extensional Versioning fordert der Anwender eine Version v_i aus dem VM-System an, verändert diese und übergibt sie wieder der Kontrolle des VM-Systems. Dieses vergibt eine neue Versionsnummer v_{i+1}, unter der die Version dann zugreifbar ist. Im Gegensatz hierzu verwalten VM-Systeme, welche auf dem Prinzip des Intensional Versioning basiert, keine Versionsnummern. Diese VM-Systeme verwalten die Versionen anhand von Beschreibungsmerkmalen. Liegt ein änderungsbasiertes Versionsmodell zugrunde, so können bisher noch nicht erstellte Versionen durch die Kombinationen von Beschreibungsmerkmalen erstellt werden. Hierbei ist zu beachten, daß die Auswahl einer Version anhand von Regeln ein NP-vollständiges Problem ist [57].

Üblicherweise basiert die änderungsbasierte Versionierung, wie z. B. in COV (VM-System von EPOS), auf dem Intensional Versioning. Der Umkehrschluß, daß dem Intensional Versioning ausschließlich änderungsbasierte VM-Systeme zugrunde liegen, gilt jedoch nicht. Ein Beispiel hierfür ist Adele [79], welches die Versionen zustandsbasiert verwaltet und mittels Regeln auf diese zugreift.

Eine Kombination von Extensional und Intensional Versioning ist möglich, beispielsweise durch die Verwendung von bedingter Kompilierung, wie sie aus den Programmiersprachen C und C++ bekannt ist und der Verwendung von CVS als VM-System [58]. In diesem Fall sind jedoch beide Arten der Versionierung nicht in einem System integriert. CVS wäre bei dieser Kombination für temporäre Varianten und vor allem für die Revisionen verantwortlich, wohingegen permanente Varianten durch die bedingte Kompilierung verwaltet werden.

Intensional Versioning ist bei feinkörnig modellierten Dokumenten prinzipiell einsetzbar, benötigt dann jedoch besondere Mechanismen zur Sicherstellung der Konsistenz. Die Beschreibungsmerkmale des Intensional Versioning beziehen sich auf Eigenschaften der Dokumente, das Editiermodell bleibt jedoch unberücksichtigt. Ohne die Berücksichtigung von Abhängigkeiten einzelner Änderungen des Editiermodells kann jedoch die Konstruktion einer Version fehlschlagen, da z. B. ein Attribut eines Objekts gesetzt werden soll, welches aufgrund der Auswahl der Beschreibungsmerkmale (noch) gar nicht existiert. Daraus folgt, daß das VM-System weitere benötigte Merkmale bei Bedarf automatisch auswählen muß. Die Auswahl ist laufzeitintensiv und das Ergebnis ist schlecht vorhersagbar, insbesondere dann wenn eine Kombination von Beschreibungsmerkmalen verwendet wird, die vorher noch nicht genutzt wurde.

Ein weiteres Problem betrifft die Erstellung/Bearbeitung eines Dokumentes. Ein Beschreibungsmerkmal beschreibt eine Eigenschaft, die während der Erstellung eines Dokumentes z. B. als Aufgabe an einen Entwickler gegeben wird. Die Aufgabe kann u.U. nicht mit einem Werkzeug oder in einer Werkzeugsitzung erledigt werden, ggf. ist sogar eine Kooperation mit anderen Entwicklern notwendig. Um diesen Anwendungsfall mit Intensional Versioning realisieren zu können, wird pro Werkzeugsitzung ein eigenes Beschreibungsmerkmal benötigt. Das führt zu einer großen Anzahl an voneinander abhängigen Beschreibungsmerkmalen, die alle in einer bestimmten Reihenfolge kombiniert werden müssen. Verwendet man stattdessen Extensional Versioning und beschreibt eine Dokumentversion durch ein Change-Package, so ist die Konsistenz eines Dokumentes sichergestellt. Durch die Realisierung der Change-Packages als eigenständige Objekte in einem Repository, läßt sich die Beziehung zu der bearbeiteten Aufgabe modellieren.

Versionsidentifizierer. In der Praxis und in Forschungsprototypen werden folgende Identifizierungsmethoden eingesetzt, um Versionen zu referenzieren und eindeutig zu identifizieren:

1. Automatische Generierung einer Nummer
2. Angabe eines symbolischen Namens
3. Beschreibung der Eigenschaften

Viele einfache Versionsverwaltungen vergeben eine automatisch erzeugte eindeutige Nummer, Beispiele sind SCCS, RCS oder CVS. Ein Nachteil ist, daß zueinander konsistente Versionen von unterschiedlichen Objekten verschiedene Nummern erhalten, wodurch die Rekonstruktion erschwert wird.
Dieser Nachteil tritt bei der Verwendung eines symbolischen Namens nicht auf. Hierbei vergibt der Benutzer einen symbolischen Namen, den die Versionsverwaltung an allen durch den Benutzer ausgewählten Versionen setzt.

Beide Methoden eignen sich in ihrer ursprünglichen Form nicht für feinkörnig modellierte Dokumente, da ein Dokument aus eine Vielzahl an Objekten besteht, die zueinander konsistent sein müssen. Wünschenswert wäre es, wenn das VM-System automatisch allen zueinander konsistenten Objekten einen Identifizierer zuordnet, da die manuelle Angabe zu aufwendig ist.
Die in den beiden bisher genannten Verfahren verwendeten Versionsidentifizierer sagen nichts über die Eigenschaft einer bestimmten Version aus. Der Entwickler muß hierbei wissen, wie die Nummer oder der Name einer bestimmten Version lautet. Hier setzt das dritte Verfahren an. Die Version erhält eine Beschreibung, anhand derer sie später wieder gefunden werden kann. Die Beschreibung besteht i.d.R. aus Booleschen Werten, die angeben, ob eine bestimmte Version eine Eigenschaft erfüllt oder nicht. Zur Suche der Version setzt man deduktive Verfahren oder Datenbanken [134] ein, wie z. B. in EPOS [97], Adele II [14] oder NORA [207,85]. Diese Technik wird beim Intensional Versioning eingesetzt, die jedoch für feinkörnig modellierte Dokumente nicht geeignet ist.

2.1.1.3 Zustände und Sichten

Ein Aspekt, den einige SKM-Systeme berücksichtigen, ist, daß sich einzelne Versionen in verschiedenen Zuständen befinden können. Im einfachsten Fall unterscheidet man zwei Zustände [199, 51], wovon ein Zustand ausdrückt, daß sich die Version in Bearbeitung befindet, dieser Zustand wird als *unstable* bezeichnet. Der andere Zustand besagt, daß die Version fertig bearbeitet wurde und den Anforderungen entspricht. Hier spricht man von *stable* oder eingefroren (engl. frozen). Andere Systeme, z. B. Adele [79] unterscheiden mehrere Zustände von Versionen. Eine Version kann z. B. bearbeitet werden, sie kann gerade getestet werden, sie kann getestet sein, usw. Zu berücksichtigen ist bei allen Systemen, wann und wie Zustandsübergänge stattfinden. In Adele steuert eine integrierte Prozeßmaschine die Zustandsübergänge. Für eine ISO 9001 [108] Zertifizierung ist u.a. sicherzustellen, daß nur validierte Software-Module ausgeliefert werden. Das wird lt. Frühauf et al. [90] erreicht, indem man alle Software-Module, die sich in einem gemeinsamen Zustand befinden, in einem Arbeitsbereich (siehe Abschnitt 2.1.3.1) ablegt. Den Übergang in einen anderen Zustand und damit die Verlagerung in einen anderen Arbeitsbereich steuert eine Prozeßmaschine anhand des Prozeßmodells. Neben den Zuständen von Versionen ist die Information noch wichtig, wer welche Änderung aus welchem Grund zu welchem Zeitpunkt gemacht hat. Magnusson [154] fordert daher, daß das SKM-System eine Möglichkeit bieten sollte, diese Informationen zu verwalten und sie dem Anwender zugänglich zu machen.

Die Anzahl und Art der Zustände, in denen sich eine Version befinden kann ist stark abhängig vom zugrundegelegten Prozeßmodell und vom Anwendungskontext. Daher kann es keine allgemeingültige Lösung geben. Das VM-System sollte daher die Möglichkeit bieten, eigene Zustände zu definieren und sie den Versionen zuzuordnen, einschließlich ergänzender Informationen.

Neben den Zuständen, in denen sich die Versionen befinden (können), sind Sichten ein weiteres Kriterium, welches Auswirkungen auf die Versionen besitzen kann. Kent [133] stellt die Frage, ob ein versioniertes Objekt in einer Datenbank, die einen Sichtenmechanismus besitzt (vgl. Abschnitt 2.3.1), unter verschiedenen Sichten gleich versioniert wird. Also ob die Versionierung eines Elementes in der Sicht A auch die Versionierung desselben Elementes in einer Sicht B impliziert. Kent betrachtet dabei Datenbanken für E-CAD-Anwendungen. Diese Fragestellung ist nicht auf diese Anwendungen beschränkt, da einige Datenbanken für andere Anwendungsbereiche auch Sichtenmechanismen besitzen.

Eine allgemeingültige Antwort gibt es nicht. Diese hängt vom konkreten Anwendungskontext ab. Jedoch gilt es zu berücksichtigen, daß es sich bei dem Element, welches in beiden Sichten betrachtet wird, immer noch um *ein* Element handelt. Wenn sich also ein Element hinsichtlich eines Aspektes in einer Sicht ändert und daher eine neue Version angelegt wird, so hat dies oft auch Auswirkungen auf andere Aspekte, die ausschließlich in anderen Sichten zugreifbar sind. Weiter gilt es zu berücksichtigen, daß die Sichten nicht ausschließlich disjunkte Aspekte umfassen. Eine dritte Sicht kann somit veränderte und unveränderte Aspekte zusammenfassen. Es müßten also zwei unterschiedliche Versionen eines Elementes in dieser Sicht präsentiert werden. Daher erscheint es sinnvoll, ein Objekt unter verschiedenen Sichten gleich zu versionieren. Existierende VM-Systeme vernachlässigen diesen Gesichtspunkt.

2.1.2 Konfigurationen

Der Begriff Version bezieht sich auf ein einzelnes versioniertes Element. Die Elemente gibt es auf unterschiedlichen Ebenen der Komponenten-Hierarchie eines Software-Produktes (vgl.: Produktraum in Abschnitt 2.1.1). Abbildung 2.4 zeigt ein vereinfachtes Beispiel[2] für eine Komponenten-Hierarchie. Hieraus ist ersichtlich, daß eine Version eines Software-Produktes aus mehreren versionierten Komponenten besteht, die wiederum aus versionierten Komponenten bestehen. Jede Komponente kann in unterschiedlichen Versionen vorliegen (vgl.: Versionsraum in Abschnitt 2.1.1).

Nicht jede Kombination von Komponenten und Versionen der Komponenten ergibt ein konsistentes Software-Produkt[3]. Das VM-System sollte deswegen die Auswahl von zueinander konsistenten Versionen unterstützen [149]. Eine konsistente Zusammenstellung bezeichnet man als *Konfiguration*. Konfigurationen kann man als Bindeglied zwischen dem Produkt- und dem Versionsraum verstehen.

Nicht alle SKM-Systeme unterstützen Konfigurationen, Beispiele hierfür sind die einfachen VM-Systeme wie RCS, SCCS oder CVS. Konfigurationen sollten nicht nur als Menge von zueinander konsistenten Versionen verstanden werden. Sie sollten auch die Struktur der Komponenten und

[2]Diese Hierarchie erhebt nicht den Anspruch der Vollständigkeit, sie soll nur einen Eindruck über die unterschiedlichen Arten von Komponenten eines Software-Produktes vermitteln. Des weiteren wurden die Abhängigkeiten zwischen einzelnen Komponenten (wie z. B. die Abhängigkeit des Quelltextes von den UML-Diagrammen) nicht berücksichtigt.

[3]Diese Problematik gibt es speziell bei der zustandsbasierten Versionierung, siehe Abschnitt 2.1.1.1. Bei der änderungsbasierten Versionierung besteht die Problematik nicht in der Wahl der Komponenten-Versionen, sondern in der Wahl der zueinander konsistenten Änderungen.

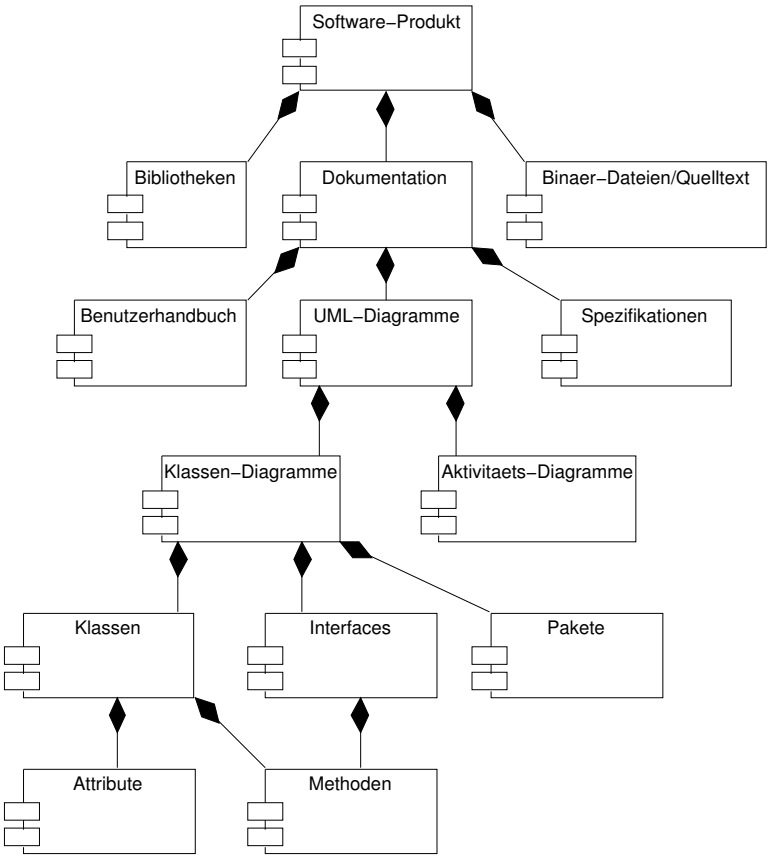


Abbildung 2.4: Beispiel für eine Komponenten-Hierarchie eines Software-Produktes

die Beziehungen zwischen ihnen berücksichtigen [44]. Die Auswahl der Komponenten und deren Versionen ist bei der Erstellung einer Konfiguration die wesentliche Aufgabe.

2.1.2.1 Auswahl von Versionen und Komponenten

Bei der Auswahl von Versionen und Komponenten gilt es zu unterscheiden zwischen der Komponentenhierarchie einerseits, der Auswahlordnung von inkludierten Komponenten und deren Versionen andererseits. Man unterscheidet drei Auswahlordnungen:

1. Product First
2. Version First
3. intertwined

Dabei bedeutet Product First, daß als erstes das Produkt bzw. Dokument gewählt wird. Damit ist automatisch die Struktur des gesamten Produktes und auch alle Komponenten im voraus festgelegt. Es besteht dann nur noch Freiheit bei der Versionsauswahl der Komponenten, Beispiele hierfür sind SCCS, RCS oder CVS. Bei deren Einsatz wechselt man als erstes in ein Verzeichnis – in den Arbeitsbereich – und wählt damit das Produkt und dessen Struktur aus, anschließend kann man bei Bedarf andere Versionen einzelner Komponenten auswählen oder neue Versionen anlegen.
Bei der Auswahlmethode Version First wird als erstes die Version gewählt, an die die Produkt- bzw. Dokumentstruktur gebunden ist. D.h. mit der Wahl einer anderen Version kann das Produkt bzw. Dokument eine andere Struktur besitzen. Diese Methode findet in PCTE Verwendung.

Eine gemischte Auswahl ist ebenfalls möglich, diese wird als intertwined bezeichnet. Hierbei wählt man abwechselnd Versionen und Komponenten aus, ein Beispiel hierfür ist ClearCase. Diese Methode ist für feinkörnig modellierte Dokumente nicht praktikabel, da alle Komponenten ein konsistentes Dokument ergeben müssen. Wenn bei jeder Komponente Freiheit bezüglich der Versionsauswahl besteht, ist die Konsistenz nicht sichergestellt.
AND/OR-Graphen visualisieren die Auswahlordnung, wobei die Struktur von Produkt- und Versionsraum nicht betrachtet wird. Hier wird auf die AND/OR-Graphen nicht weiter eingegangen, da sie für diese Arbeit nicht relevant sind. Für Details sei auf [218] verwiesen.

Bei der bisherigen Betrachtung der Auswahlordnung blieben Abhängigkeiten in der Komponentenhierarchie weitgehend unberücksichtigt. Diese Abhängigkeiten sind jedoch bei feinkörnig modellierten Dokumenten relevant. Allgemein gibt es drei Arten, die Auswahlordnungen auf die Komponentenhierarchie anzuwenden:

1. Komponenten-Versionierung (engl. component versioning)
2. Vollständige Versionierung (engl. total versioning)
3. Produkt-Versionierung (engl. product versioning)

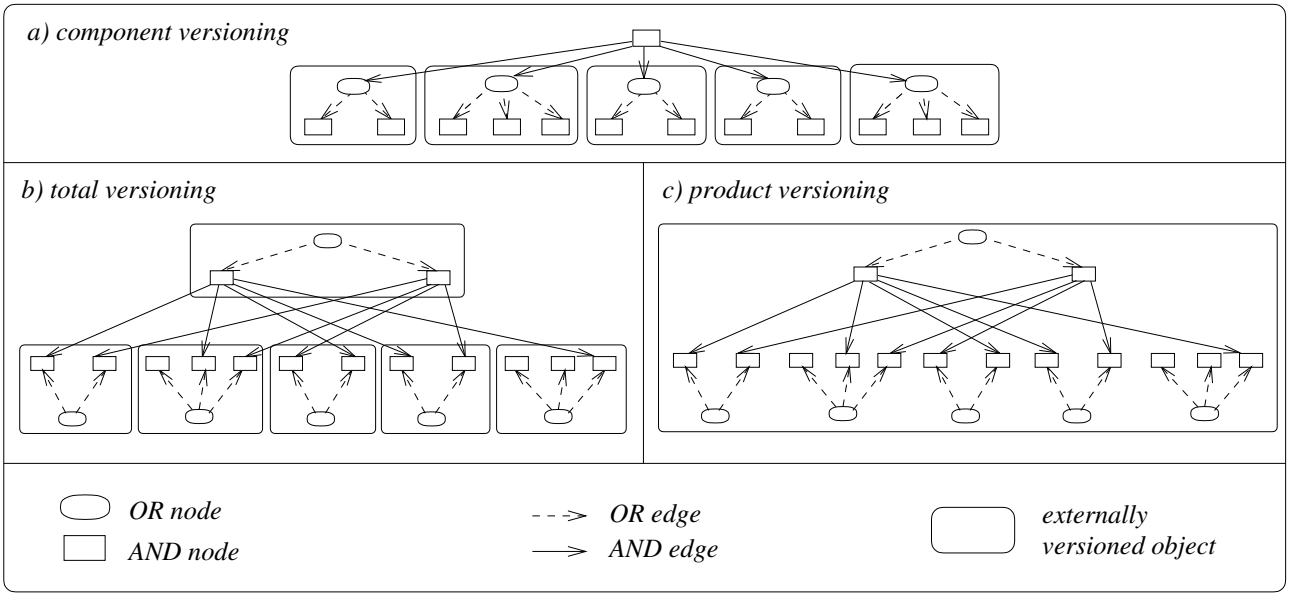


Abbildung 2.5: Versionsauswahl von Komponenten (aus [55])

Bei der Komponenten-Versionierung werden ausschließlich die Blätter der Hierarchie versioniert (siehe Abbildung 2.5(a)), so daß die Auswahlordnung Product First vorliegt (Bsp.: SCCS oder CVS). Problematisch sind bei dieser Art der Versionierung Umbenennungen oder Verschiebungen von Komponenten innerhalb der Hierarchie, z. B. verschieben einer Datei von einen Verzeichnis in ein anderes. Das entscheidende Problem bei dieser Versionierungsart ist die fehlende Versionierung des gesamten Produktes. Zueinander konsistente Versionen der Komponenten müssen mit Hilfsmitteln, wie z. B. abstrakten Bezeichnern oder Zeitmarken bestimmt werden. Das VM-System bietet hierfür keine Unterstützung. Somit ist diese Technik nicht für feinkörnig modellierte Dokumente geeignet.
Hier setzt die Vollständige Versionierung an. Hierin wird die gesamte Komponentenhierarchie versioniert (siehe Abbildung 2.5(b)), so daß man die Versionierungsart mit den Auswahlordnungen Version First oder intertwined kombinieren kann. PCTE und ClearCase arbeiten nach

diesem Konzept. In PCTE legt die Version eines Elementes auch die Komponenten und deren Versionen fest, so daß PCTE nach der Auswahlordnung Version First arbeitet. Im Gegensatz hierzu legt die Version eines Verzeichnisses in ClearCase dessen Komponenten, aber nicht deren Versionen fest, es liegt also eine intertwined Auswahlordnung vor. Bei der Vollständigen Versionierung besitzt jedes versionierte Element seinen eigenen Versionsraum und wird auch unabhängig von anderen Elementen versioniert. Die Versionsidentifizierer besitzen daher keine Aussagekraft darüber, ob zwei versionierte Elemente mit demselben Versionsidentifizierer zur selben Dokumentversion gehören, sofern es sich bei den beiden Elementen um Komponenten eines Dokumentes handelt.
Die Produkt-Versionierung betrachtet nicht alle Objekte für sich, sondern das gesamte Produkt als ein zu versionierendes Ganzes in einem zusammenhängenden Versionsraum (siehe Abbildung 2.5(c)). Diese Methode verfolgt die Idee, daß durch eine Änderung oftmals nicht nur eine atomare Komponente betroffen ist, sondern mehrere. Der globale Versionsraum erleichtert die Wahl von konsistenten Versionen der Komponenten. Vor allem änderungsbasierte SKM-Systeme unterstützen diesen Ansatz. Ein Nachteil der Produkt-Versionierung ist das Fehlen von Modularität, da alle Versionen global sind. Dieser Nachteil ist für feinkörnig modellierte Dokumente nicht gravierend, da die Komponenten eines Dokumentes nicht für sich allein betrachtet werden. Von Interesse ist eher das gesamte Produkt und dessen Versionen. Unter Berücksichtigung dieser Tatsache ist die fehlende Modularität eher ein Vorteil, da durch den gemeinsamen Versionsraum die Konsistenz des Dokumentes sichergestellt ist, insbesondere gilt das für UML-Diagramme, da man diese als eine „Sicht“ auch das eigentliche UML-Modell interpretieren kann.

2.1.2.2 Binden von Konfigurationen

Die Zusammenstellung einer Konfiguration bezeichnet man als *binden*. Man unterscheidet:

1. dynamisches Binden
2. statisches Binden

Beim dynamischen Binden werden die Versionen der Komponenten beim Zugriff auf eine Konfiguration ausgewählt. Somit ist es z. B. möglich, immer die aktuellen Entwicklerversionen der Komponenten einer Konfiguration zu erhalten. Dynamisch gebundene Konfigurationen realisiert man zum Beispiel mit Hilfe von Regeln wie bei Make [84] oder DSEE [144]. Der Nachteil ist jedoch einerseits ein größerer Aufwand beim Zugriff auf eine Konfiguration, durch die Notwendigkeit diese dynamisch zusammenzustellen, andererseits die Komplexität der Regeln.

Im Gegensatz zum dynamischen Binden legt man beim statischen Binden die Version von allen Komponenten des Produktes im voraus fest. Diese bestehen i.d.R. aus Versionen, die zu einem Zeitpunkt im Arbeitsbereich eines Entwicklers vorlagen. Bei einem späteren Zugriff werden dann wieder dieselben Versionen der Komponenten ausgewählt. Man bezeichnet das auch als eine *gebundene Konfiguration* . Ein Anwendungsfall von gebundenen Konfigurationen ist die Markierung von Meilensteinen während der Entwicklung.

Eine einfache Möglichkeit der Realisierung von gebundenen Konfigurationen in dateibasierten VM-Systemen ist, jede Version mit einem Bezeichner zu versehen, der die Konfiguration angibt, in der diese Version verwendet wird. CVS nutzt zum Beispiel diese auch als *Tagging* bezeichnete Methode. Sie ist einfach zu realisieren, besitzt jedoch auch einige Nachteile. Da die Konfigurationen nicht explizit verwaltet werden, ist es nicht möglich, weitere Informationen an einer Konfiguration zu speichern oder Beziehungen, wie zum Beispiel eine Nachfolgerbeziehung, zwischen zwei Konfigurationen zu verwalten.

Werden nicht Dateien sondern Objekte in einem Repository verwaltet, so sind zusätzlich die Beziehungen der Objekte untereinander zu berücksichtigen, so daß die Methode des Tagging nicht anwendbar ist. In diesem Bereich gibt es zwei Methoden, konsistente Objektversionen zu verwalten:

- Implizite Konfigurationen: Jedes Objekt ist eine Konfiguration für seine Komponenten und referenziert somit direkt bestimmte Versionen.

- Explizite Konfigurationen: Jedes Objekt des gesamten Dokumentes ist unabhängig und stellt *keine* Konfiguration für seine Komponenten dar. Unabhängig von den Objekten, die das Dokument modellieren, werden die Konfigurationen als sogenannte *first class* Objekte [34] verwaltet.

Ein Beispiel für ein Repository mit impliziten Konfigurationen ist das VM-System von PCTE (siehe Abschnitt 2.3.1). Die einzelnen Versionen eines atomaren Objektes sind durch Vorgänger- und Nachfolger-Beziehungen miteinander verknüpft.
Bei dieser Art der Realisierung sind laut Kent [133] einige Fragen zu klären. Diese betreffen die ausgehenden Beziehungen der Objekte. Gehören diese zur Version des Ausgangsobjektes oder werden diese unabhängig versioniert? Wenn die Beziehungen zur Version des Ausgangsobjektes gehören, wie wird dann auf Versionsänderungen des Zielobjektes reagiert? Die Beziehung referenziert schließlich eine konkrete Version des Zielobjektes. Weiterhin ist zu unterscheiden, ob die Beziehungen innerhalb eines Dokumentes (z. B. innerhalb eines komplexen Objektes) verlaufen oder ob sie verschiedene Dokumente verknüpfen [116]. Bei dokumentübergreifenden Beziehungen sind dieselben Fragen wie bei dokumentinternen Beziehungen zu klären, nur daß auf Versionsänderungen ggf. unterschiedlich zu reagieren ist.

Versionspropagierung. Eine Möglichkeit, auf Versionsänderungen zu reagieren, ist die Versionspropagierung, d.h. wenn von einer Komponente eine neue Version angelegt wird, so wird auch von dem kapselnden Objekt eine neue Version angelegt. Die Versionspropagierung dient der Konsistenzerhaltung zwischen Teildokumenten und ggf. vollständigen Dokumenten. Die Versionspropagierung ist jedoch nicht in jedem Fall erwünscht, da dies in einer großen Anzahl von nicht benötigten Versionen von Dokumenten resultieren kann [157].

Der Vorteil der Versionspropagierung liegt darin, daß keine expliziten Konfigurationen angelegt oder verwaltet werden müssen. Nachteilig ist jedoch, daß selbst unveränderte Objekte versioniert werden und man somit nicht aufgrund unterschiedlicher Versionsidentifizierer auf Änderungen am Knoten selbst schließen kann.
Einsetzen läßt sich diese Technik bei Dokumenten, die eine baumartige Struktur besitzen. Problematisch sind Graph-Strukturen, in denen ein Knoten auf mehreren Pfaden erreicht werden kann. Entlang welchen Pfades soll die Versionierung propagiert werden? Was ist, wenn es Beziehungen mit unterschiedlichen Semantiken (Link-Kategorien, siehe Abschnitt 2.3.1) gibt, wie z. B. in PCTE? Welchen Einfluß hat die Multiple-View-Integration?
Betrachtet man das Metamodell der UML, so stellt man fest, daß darin zwischen den Modelldaten und einzelnen Diagrammen unterschieden wird. Ein Element der Modelldaten kann in mehreren Diagrammen dargestellt werden. Die Diagramme und die Modelldaten müssen daher konsistent gehalten werden. Hierfür ist die Versionspropagierung ungeeignet. Ebenso, wenn mit einer Konfiguration zusätzliche Informationen gespeichert werden sollen, wie z. B. der Name des Entwicklers, der die Version angelegt hat, der Grund für die Änderung usw.

Explizite Konfigurationen. Die explizite Modellierung von Konfigurationen als „first-class“ Objekte [34] in dem SKM-System löst diese Probleme, da die Beziehungen zwischen einzelnen Objekten nicht eine bestimmte Version des Ziel-Objektes referenzieren und somit alle Objektversionen unabhängig voneinander versioniert werden. Deshalb ist auch die Versionierung von graphartigen Dokumenten möglich. Die Objektversionen, die zusammen ein konsistentes Dokument bilden und zueinander konsistente Dokumente werden durch die Konfigurationen bestimmt. Diese Technik läßt sich daher auch mit der Multiple-View-Integration und UML-Diagrammen einsetzen. Weiterhin lassen sich die expliziten Konfigurationen nutzen, um zusätzliche Informationen an einer Konfiguration zu speichern, wie z. B. den Namen des Entwicklers, eine Änderungsbeschreibung o.ä. Der Nachteil liegt in einem erhöhten Aufwand zur Realisierung und Verwaltung der Konfigurationen.
Explizite Konfigurationen können auf zwei Arten realisiert werden, zum einen können die Konfigurationsobjekte ihre Komponenten referenzieren, wie z. B. in COACT [137] oder aber die Komponenten referenzieren die Konfiguration zu der sie gehören, wie in Ensemble [223, 224]. Welche dieser beiden Ansätze sinnvoll ist, hängt von der Art des Zugriffs auf die Objekte und der geforderten Funktionalität ab. Müssen sich alle Objekte einer Konfiguration leicht bestimmen lassen, ist der erste Ansatz sinnvoll. Das hat jedoch den Nachteil, daß die Objektidentifizierer an den Konfigurationen verwaltet werden müssen und bei einem navigierendem Zugriff auf die Objekte muß vor jedem Zugriff aus der Konfiguration die passende Objektversion ausgelesen werden. Das stellt einen nicht unerheblichen Aufwand dar. Der zweite Ansatz ist in diesem Fall vorzuziehen, da u.U. gar nicht alle Komponenten benötigt werden und bei jedem Objekt lokal entschieden werden kann, welche Version zu der geforderten Konfiguration gehört.

Das Versionierungskonzept von Ensemble ist nach diesen Überlegungen prinzipiell auch zur Versionierung von UML-Diagrammen geeignet. Durch die Ausrichtung von Ensemble auf inkrementelle Algorithmen, wie z. B. inkrementelle Parser, ergeben sich einige Eigenschaften, die sich für die Versionierung von UML-Diagrammen jedoch als nachteilig erweisen. Beim Anlegen einer Version wird nicht ausschließlich das geänderte Objekt versioniert, sondern auch an dessen Vaterknoten ein Hinweis gespeichert, daß eine Komponente geändert wurde. Diese Hinweise werden wie bei der Versionspropagierung an allen Objekten bis zum Wurzelobjekt gesetzt. Das ist für interaktive Werkzeuge nicht geeignet, da u.U. viele Objekte betroffen sein können und sich somit die Antwortzeit merklich verlängert. Ein weiterer Nachteil besteht darin, daß Ensemble keine Mischwerkzeuge besitzt. Der Einsatz in kooperativen Arbeitsumgebungen wird hierdurch erschwert. Ein weiterer Nachteil für kooperative Arbeit ist, daß aufeinander folgende Konfigurationen zu einer Konfiguration zusammengefaßt werden, so daß die Informationen verloren gehen, welcher Entwickler für welche Änderungen verantwortlich ist, sofern diese Daten an der Konfigurationen gespeichert sind. Durch Zusammenfassen entstandene Konfigurationen beinhalten auch viele einzelne Änderungen, so daß weniger Rücksetzpunkte existieren, sofern bestimmte Änderungen zurückgenommen werden sollen.

2.1.2.3 Buildmanagement

Das Buildmanagement ist ein Teilaspekt des Konfigurationsmanagements. Das Ziel ist hierbei, die Konsistenz von Quell-Versionen und generierten Dokumenten sicherzustellen. Eine Quell-Version kann ein Programm-Quelltext, aber auch ein Klassendiagramm sein, wobei die generierten Dokumente zum einen das ausführbare Programm und zum anderen eine Menge von automatisch erzeugten Klassenrümpfen sind. Änderungen an den Quell-Versionen sollen dabei zu einer erneuten Generierung der abgeleiteten Dokumente führen. Das primäre Ziel ist die Minimierung des Aufwands für die Generierung. Man strebt dabei an, nur die Dokumente neu zu erzeugen, deren Quell-Versionen tatsächlich geändert wurden. Ein Werkzeug, welches hierfür

eingesetzt wird, ist z. B. Make [84]. Diese Thematik ist im Rahmen dieser Arbeit nicht von Interesse und wird daher nicht weiter betrachtet,

2.1.2.4 Laufzeitumgebungen

Die Konfiguration von Laufzeitumgebungen ist ein weiterer Bereich, wo Konfigurationen eingesetzt werden. Hierbei liegt der Schwerpunkt auf der Zusammenstellung von Bibliotheken und Werkzeugen, die zusammen eingesetzt werden sollen. Larsson und Crnkovic [142] diskutieren diesen Bereich. Berghoff et al. [22] betrachten Konfigurationen in verteilten Systemen mit Hilfe von Agenten. Diese Art von Konfigurationen werden hier nicht weiter betrachtet, da sich die Ziele unterscheiden.

2.1.3 Benutzungsmodelle

Die einzelnen SKM-Systeme besitzen unterschiedliche Benutzungsmodelle. Diese bestimmen in welcher Art und Weise der Anwender mit dem SKM-System interagiert. Abhängig vom Anwendungsbereich besitzt jedes Modell seine eigenen Vor- und Nachteile. Es gibt die folgenden vier Modelle [82, 67]:

- Check-Out/Check-In
- Kompositionsmodell
- Lange Transaktion[4]
- Change Set Model

Check-Out/Check-In-Modell: Das Check-Out/Check-In-Modell basiert auf dem logischen oder physischen Kopieren von Elementen zwischen *Arbeitsbereichen* (engl. Workspaces, siehe Abschnitt 2.1.3.1). Arbeitsbereiche sind logische oder physische Gruppierungen von versionierten Elementen[5], die i.a. hierarchisch angeordnet sind. Ein Check-Out kopiert ein oder mehrere Elemente aus einem übergeordneten Arbeitsbereich oder aus einem Repository in einen (untergeordneten) Arbeitsbereich. Die Check-In-Funktion kopiert die Elemente wieder zurück und legt i.d.R. von den Elementen dabei eine neue Version an. Beispielsweise nutzen RCS und SCCS dieses Modell.
In Kombination mit diesem Modell werden pessimistische oder optimistische Verfahren des Concurrency Control eingesetzt (vgl. Abschnitt 2.1.4). Asklund [6] schlägt eine feinkörnige Modellierung vor, um die Wahrscheinlichkeit des konkurrierenden schreibenden Zugriffs auf eine Elementversion zu reduzieren. Das Check-Out/Check-In-Modell berücksichtigt nicht die Dokument-Struktur, so daß die Anwender entscheiden müssen, welche Elemente sie in ihrem Arbeitsbereich benötigen.

[4] Das Modell der langen Transaktion ist vergleichbar mit den langen Transaktionen (Entwurfstransaktionen), die aus dem Kontext der Datenbanken bekannt sind. Darauf ist der Name zurückzuführen.

[5] Unter Elementen sind hier bis auf weiteres Dateien im Dateisystem oder Objekte aus einer OODB zu verstehen.

Kompositionsmodell: Das Kompositionsmodell ist eine Erweiterung des Check-Out/Check-In-Modells, welches eine bessere Konfigurationsverwaltung besitzt. Hierin wird durch ein System-Modell beschrieben, welche Elemente zu einem Dokument/Produkt gehören und mittels Check-Out in einen Arbeitsbereich kopiert werden sollen. Durch Auswahlregeln wird die Version der einzelnen Elemente festgelegt. Es basiert auf der Methode der Vollständigen Versionierung.

Ein wesentlicher Nachteil des Kompositionsmodells und somit auch des Check-Out/Check-In-Modells besteht darin, daß das Übertragen der einzelnen Elemente nicht atomar ist. Es besteht die Möglichkeit, einzelne Elemente vorzeitig in den übergeordneten Arbeitsbereich zu übertragen. Das hat den Nachteil, daß logisch zusammengehörige Änderungen einzeln übertragen werden können und somit der übergeordnete Arbeitsbereich inkonsistente Dokumente enthalten würde. Diese Modelle sind daher nicht für feinkörnig modellierte Dokumente geeignet.

Lange Transaktion: Das Modell der langen Transaktion konzentriert sich auf Konfigurationen (siehe auch Abschnitt 2.1.2), logischen Änderungen und auf die kooperative Arbeit von Entwicklergruppen. Die Idee ist die Entwicklung des Gesamtsystems, die in größere Arbeitseinheiten aufgeteilt ist und die in Form von logischen Änderungen integriert werden.
Zur Bearbeitung einer Aufgabe legt man einen neuen Arbeitsbereich an und überführt alle benötigten Element-Versionen aus dem übergeordneten Arbeitsbereich. Das bezeichnet man als *Bringover* (siehe Abbildung 2.6). Die Entwickler führen die Änderungen anschließend lokal im Arbeitsbereich durch. Die Elemente können ergänzend im Arbeitsbereich versioniert werden, ohne daß das Auswirkungen auf andere Arbeitsbereiche hat. Die Element-Versionen stellen einen temporären Versionszweig dar. Alle Element-Versionen in einem Arbeitsbereich kann man als eine gebundene Konfiguration interpretieren, die eine Version des Gesamt-Systems darstellt. Bearbeitet ein anderer Entwickler dieselben Element-Versionen in einem eigenen Arbeitsbereich, so stellen diese Element-Versionen eine Konfigurationsvariante dar. Somit ist dieses Benutzungsmodell optimistisch, da kein Entwickler durch Sperren gehindert wird, ein Element zu verändern.

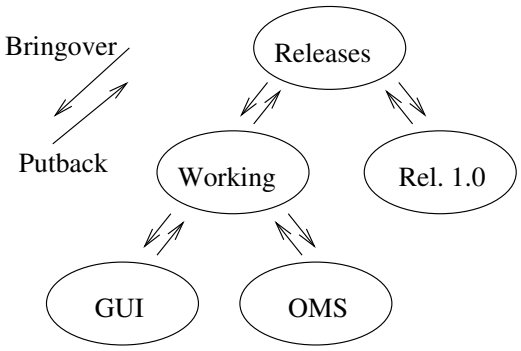


Abbildung 2.6: Hierarchie von Arbeitsbereichen im Rahmen von langen Transaktionen (aus [7])

Nach Fertigstellung der Arbeitseinheit werden alle geänderten Elemente gemeinsam mittels *Putback* in den übergeordneten Arbeitsbereich übertragen. Andere Entwickler haben anschließend die Möglichkeit, die Elemente in ihren Arbeitsbereichen mit der *Update-Operation* zu aktualisieren.
Wenn vor dem Putback die im lokalen Arbeitsbereich geänderten Elemente ebenfalls im übergeordneten Arbeitsbereich verändert wurden, so muß der Entwickler ein Update durchführen und die Änderungen mischen, siehe Abbildung 2.7.
Alle Operationen sind atomar, d.h. es werden entweder alle Elemente übertragen oder keins. Somit bietet es sich für feinkörnig modellierte Dokumente an, da sichergestellt ist, daß nur

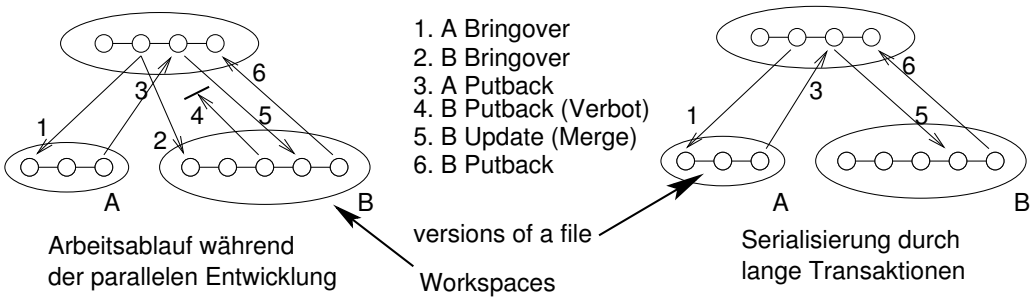


Abbildung 2.7: Parallele Entwicklung und Serialisierung durch lange Transaktionen (aus [7])

die Dokumente als Ganzes übertragen werden und nicht einzelne Teile. Nachteilig ist, daß die zusammengehörigen Versionen nicht explizit verwaltet werden, was für feinkörnig modellierte Dokumente sinnvoll ist.
Dieses Verhalten serialisiert alle Änderungen im übergeordneten Arbeitsbereich (siehe Abbildung 2.7) und ist deswegen vergleichbar mit der Serialisierung, wie sie von Transaktionen in Datenbankmanagement-Systemen bekannt ist. Aufgrund dieser Eigenschaft hat das Modell seinen Namen erhalten.
Die Atomarität der Putback-Operation ermöglicht auch die Interpretation von Änderungen an allen Elementen als eine einzige logische Änderung, die durch die Operationen veröffentlicht[6] (integriert) werden. Die Trennung von Veröffentlichung und Integration bietet eine *konservative Update-Strategie*, in der jeder Entwickler selbst entscheiden kann, zu welchem Zeitpunkt er die Änderungen seiner Kollegen mit den eigenen Änderungen integriert.

Change-Set-Modell: Dieses Modell basiert auf Change-Sets, wie sie bereits in Abschnitt 2.1.1.1 vorgestellt wurden. Die Change-Sets können als Anknüpfungspunkte für das Änderungsmanagement dienen, um die durchzuführenden Arbeitseinheiten zu verwalten.
Ein Change-Set kann man als das Ergebnis einer langen Transaktion interpretieren [7]. Der Unterschied zwischen den beiden Benutzungsmodellen ist, daß in dem Change-Set-Modell die logischen Änderungen explizit verwaltet werden, und vor allem, daß die Änderungen erst später integriert werden. Im Gegensatz hierzu müssen im Modell der langen Transaktionen die Änderungen sofort integriert werden. Die späte Integration im Change-Set-Modell ermöglicht die Kombination von Änderungen, die bisher noch nicht existierte. Diese Kombination muß jedoch auf Konsistenz getestet werden, da diese nicht automatisch sichergestellt werden kann. Für dieses Modell gelten die in Abschnitt 2.1.1.1 genannten Vor- und Nachteile.

Kombinierte Modelle: Die einzelnen Benutzungsmodelle können in einem konkreten VM-System kombiniert werden. Beispiele hierfür sind VM-Systeme, die Change-Packages verwenden. Für feinkörnig modellierte Dokumente bietet sich ein Benutzungsmodell auf Basis der Langen Transaktion an, welches alle mit der Putback-Operation übertragenen Elemente in einem Change-Package zusammenfaßt.

Neben der Benutzer-Interaktion ist auch das Software-Prozeßmodell zu berücksichtigen. Zeller kritisiert, daß viele SKM-Systeme nach einem eigenen Prozeßmodell arbeiten, an das die Benutzer gebunden sind [242]. Seiner Ansicht nach sollte das SKM-System an unterschiedliche Prozeßmodelle anpaßbar sein. Als Beispiel hierfür nennt er das System ICE, daß auf Feature-Logik basiert [240].

[6]Das Übertragen von geänderten Elementen in den übergeordneten Arbeitsbereich wird auch als veröffentlichen bezeichnet, da anderen Entwickler dadurch Zugriff auf diese neuen Versionen erhalten. Sie werden sozusagen öffentlich gemacht.

2.1.3.1 Arbeitsbereiche

Während der Softwareentwicklung sind inkonsistente Zwischenzustände eher die Regel als die Ausnahme. Diese können andere Entwickler bei ihrer eigenen Tätigkeit behindern. Daher sollten diese Zwischenzustände nur für die jeweiligen Entwickler sichtbar sein. Die Änderungen sollen also isoliert werden. Nach Abschluß einer Aufgabe durch einen Entwickler müssen dessen Änderungen mit den Änderungen der anderen Entwickler integriert werden, um das gesamte Produkt zu erstellen und auch um allen anderen Entwicklern den Zugriff auf die Änderungen zu ermöglichen.
Die Lösung der Problemstellung, einerseits eine Isolation, andererseits eine Kooperation zu ermöglichen, ist das Konzept der Arbeitsbereiche. Die Kooperation beschränkt sich jedoch in den meisten Fällen auf den Austausch von veröffentlichten Änderungen. Estublier [79] stellt verschiedene Modelle zur Verwaltung von Arbeitsbereichen vor.

Modelle. Im einfachsten Fall ist ein Arbeitsbereich ein Verzeichnis im Dateisystem, in das Dateien einzeln aus einem zentralen Repository kopiert werden, vgl. Abbildung 2.8. Die Benutzerinteraktion basiert in diesem Fall auf dem Check-Out/Check-In-Modell (CO/CI).
Die Daten im Arbeitsbereich unterliegen nicht der Kontrolle des VM-Systems. Daher kann dieses nicht sicherstellen, daß alle geänderten Dateien wieder in das Repository übertragen werden und es kann auch keine weitergehende Funktionalität wie z. B. eine lokale Versionierung innerhalb eines Arbeitsbereichs anbieten. Diese wäre während der Entwicklung jedoch wünschenswert, um Zwischenzustände temporär zu verwalten [15] und bei Bedarf wieder rekonstruieren zu können. Der Entwickler muß, um Zwischenergebnisse zu sichern, diese in das Repository kopieren, was darin zu einer großen Anzahl an Versionen führt, die er evtl. später und andere Entwickler gar nicht benötigen.

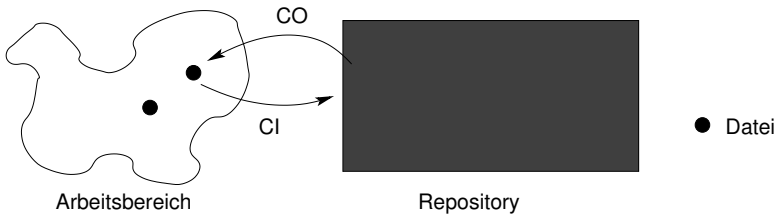


Abbildung 2.8: Check-Out/Check-In Paradigma (aus [79])

In erweiterten VM-Modellen verwaltet das VM-System explizit den Arbeitsbereich. Dieses kopiert eine *Menge* von konsistenten Dateien mittels CO/CI zwischen Repository und Arbeitsbereich, der im Dateisystem liegt. Der Zugriff auf die Arbeitsbereiche kann durch das VM-System kontrolliert werden, sofern alle Dateizugriffe durch das VM-System gekapselt sind (siehe Abbildung 2.9). Die meisten SKM-Systeme gehören in dieser Gruppe, auch wenn sich diese im Detail unterscheiden. Ein Beispiel für diese Systeme ist NSE [61, 164].

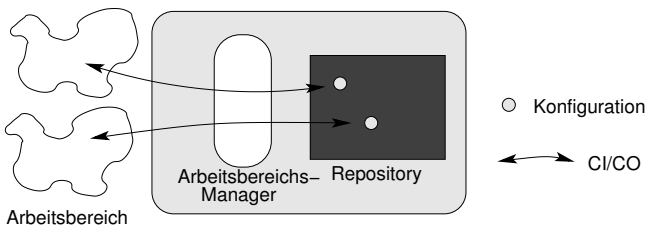


Abbildung 2.9: Explizite Arbeitsbereichsverwaltung (aus [79])

Das Problem dieser beiden Modelle ist die Synchronisierung von Arbeitsbereich und Repository, ähnlich wie bei den einfachen Modellen. Andere VM-Systeme kombinieren daher Arbeitsbereich und Repository. Hierfür gibt es zwei Möglichkeiten:

1. Anpassung des Dateisystems, welches den Arbeitsbereich bereitstellt
2. Kapselung der Werkzeuge

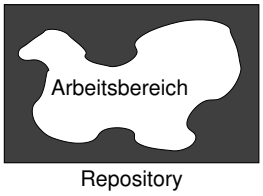


Abbildung 2.10: Arbeitsbereich liegt im Repository (aus [79])

Man kann das Dateisystem anpassen, indem der Arbeitsbereich in einem virtuellen Dateisystem liegt, welches mittels NFS in das reguläre Dateisystem eingebunden wird. Andere Möglichkeiten sind spezielle Dateisystem-Treiber, wie sie ClearCase verwendet, oder der Austausch von Bibliotheksfunktionen, die die Dateihandhabung betreffen. Bei dieser Lösung können existierende Werkzeuge beibehalten werden, jedoch sind Eingriffe in die Betriebssystem-Umgebung notwendig.

Die bisher vorgestellten Arbeitsbereichsmodelle besitzen für feinkörnig modellierte Dokumente den Nachteil, daß sie ausschließlich auf der physischen Repräsentation der Dokumente arbeiten. Im Fall vom UML-Editoren sind das i.d.R. Dateien, die alle Dokumente eines Projektes beinhalten. Nutzt man diese Modelle zur Versionierung, so wird die Projektdatei und somit alle Dokumente gemeinsam versioniert. Änderungen an einzelnen Teilen der UML-Diagramme lassen sich mit dieser Technik nicht nachvollziehen.

Einen anderen Weg verfolgt die Kapselung der Werkzeuge. Hierbei sind keine Änderungen am Betriebssystem notwendig, jedoch müssen die Werkzeuge mit dem VM-System integriert werden, so daß existierende Werkzeuge nicht ohne Anpassungen weiterverwendbar sind. Die Werkzeuge müssen auf die Schnittstellen des VM-Systems aufsetzen. Jedoch können sie dann auch weitere Dienste nutzen, wodurch die Entwicklung von Werkzeugen vereinfacht werden kann [132]. PCTE gehört in diese Kategorie.

Im Bereich der Datenbanken wurde das Konzept der Sub-Datenbasis entwickelt. Beispiele sind Coven [46] und DAMOKLES [2]. Eine Sub-Datanbasis beinhaltet eine Teilmenge der Daten der zentralen Datenbank. Die Sub-Datenbasen können als Datenspeicher für lange Transaktionen angesehen werden. Änderungen an den Daten in einer Sub-Datenbasis sind nur in dieser selbst sichtbar. Erst beim Commit werden die Änderungen in die zentrale Datenbank propagiert (siehe Abbildung 2.11).

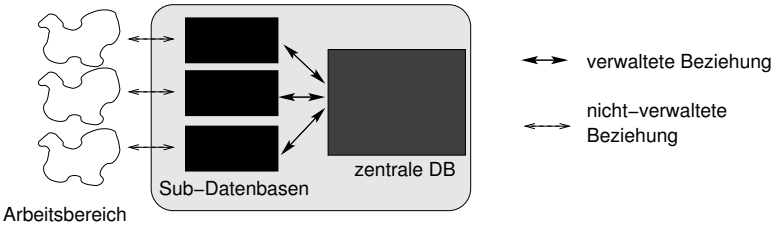


Abbildung 2.11: Sub-Datenbasen als Arbeitsbereiche (aus [79])

Das Vorgehen ist vergleichbar mit konventionellen Arbeitsbereichen, nur daß hier nicht Dateien, sondern Objekte verwaltet werden. Daher ist das Datenmodell flexibler und es können die üblichen Dienste einer Datenbank genutzt werden, wie z. B. Abfragen, Verteilung oder Zugriffsschutz. Des weiteren ist eine erweiterte Funktionalität möglich, wie z. B. transparente Versionierung oder konkurrierende Änderungen an denselben Objekten.

Kombiniert man Arbeitsbereich, Sub-Datenbasis und Repository, so entfallen die meisten Nachteile [79]. Alle Dienste des Repositories sind nutzbar, wie z. B. lokale Versionierung in den Arbeitsbereichen, Abfragen oder Zugriffsschutz. Das VM-System hat die volle Kontrolle über die Arbeitsbereiche und kann den Datenaustausch zwischen den einzelnen Arbeitsbereichen steuern. Des weiteren können die einzelnen Arbeitsbereiche eine unterschiedliche Sicht auf die Daten besitzen, die sowohl die Struktur als auch die Semantik betrifft.
Dieser Ansatz eignet sich prinzipiell zur Versionierung von feinkörnig modellierten Dokumenten. Die Dokumente werden mit Hilfe der Arbeitsbereiche verwaltet. Alle Objekte, die zusammen ein Dokument modellieren sind in einer Sub-Datenbasis gespeichert, die einem Arbeitsbereich zugeordnet ist. Die Versionierung innerhalb der Sub-Datenbasis ist möglich, so daß evtl. benötigte Zwischenversionen existieren. Für die Versionierung der Dokumente ist es jedoch nicht von entscheidender Bedeutung, ob eine Sub-Datenbasis für einen Anwender explizit „sichtbar“ ist oder ob sie nur ein virtuelles Konzept ist, das die modifizierten Objekte kapselt.

Strukturen von Arbeitsbereichen. In der Literatur finden sich unterschiedliche Strukturen von Arbeitsbereichen. Santoyridis et al. [199] beschreiben ein hierarchisches Arbeitsbereichsmodell (VSSCD), welches aus drei Ebenen besteht (Projekt, Subprojekt und Privat), in denen sich Objekte in fünf unterschiedlichen Zuständen befinden können. Abhängig vom Zustand einer Version kann diese in den übergeordneten Arbeitsbereich transferiert werden. In einem Experiment von Santoyridis et al. [199] stellten sich die fünf Zustände als kontraproduktiv heraus, so daß sie diese auf drei reduziert haben.

In dem System Adele [73] bilden die Arbeitsbereiche eine Baumstruktur. Die Arbeitsbereiche kann man einzeln anlegen. Regeln legen die Art der Kooperation zwischen den einzelnen Arbeitsbereichen fest. Sie bestimmen, wie und ob Änderungen zwischen einzelnen Arbeitsbereichen propagiert werden. Eine ähnliche Struktur bilden die Arbeitsbereiche in EPOS [173]. Hier arbeiten die Werkzeuge in langen geschachtelten Transaktionen, denen je ein einzelner Arbeitsbereich zugeordnet ist. Die Änderungen innerhalb eines Arbeitsbereichs werden i.d.R. beim Commit der langen Transaktion in den übergeordneten Arbeitsbereich übernommen. Es besteht jedoch die Möglichkeit, Änderungen vor dem Commit der Transaktion in den übergeordneten Arbeitsbereich zu übertragen. Bei einem Abbruch der Transaktion müssen diese vorzeitig übertragenen Änderungen rückgängig gemacht werden, was zu einem kaskadierenden Roll-Back führen kann, wenn weitere Transaktionen auf diese Daten zugegriffen haben.

Das Coo-Projekt [93] ist vergleichbar mit EPOS. Darin arbeitet jeder Prozeß innerhalb einer Transaktion, die jeweils einem Arbeitsbereich zugeordnet ist. In jedem Prozeß wird die Struktur des Dokuments einschließlich dessen Integritätsbedingungen beschrieben. Die Prozesse kann man in Subprozesse aufteilen, um Aufgaben parallel bearbeiten zu können. Durch die Bindung der Transaktionen an die Prozesse entsteht eine offen geschachtelte Transaktionsstruktur. Die Kooperation von Prozessen wird durch den Transfer von Zwischenergebnissen in den übergeordneten Arbeitsbereich vor Ende der Transaktion ermöglicht. Die Einhaltung der Integritätsbedingungen wird nur bei den Endergebnissen überprüft und nicht bei Zwischenergebnissen.

Problematisch sind bei den komplexen Arbeitsbereichsmodellen, die Abhängigkeiten zwischen einzelnen Arbeitsbereichen sofern Daten vor Abschluß der Bearbeitung ausgetauscht wurden. Die Möglichkeit, daß Änderungen durch ein erzwungenes Roll-Back verloren gehen, ist für

interaktive Werkzeuge nicht tolerierbar. Ein größeres Problem ist, daß diese Arbeitsbereichsmodelle nicht zwischen Dokumenten und den Objekten, die ein Dokument modellieren unterscheiden. Entweder werden Dokumente ausgetauscht oder die Objekte. Komponentenstrukturen berücksichtigt kein genanntes Modell.

Isolation von Änderungen. Ein Aspekt beim Einsatz von Arbeitsbereichen ist die Isolation der Arbeit mehrerer Entwickler. Da es für die kooperative Entwicklung nicht sinnvoll ist, die Arbeitsergebnisse mehrerer Entwickler vollständig gegeneinander abzuschirmen (vgl. Abschnitt 1.2.5), sollte das SKM-System, unterschiedliche Grade der Isolation anbieten. EPOS [53] bietet z. B. vier unterschiedliche Grade an:

- **global:** keine Isolation; alle Entwickler arbeiten auf einem gemeinsamen Datenbestand.
- **local:** Vollständige Isolation; jeder Entwickler arbeitet auf einer eigenen Kopie der Daten.
- **copy-on-write:** Kopie bei Schreibzugriff; die Entwickler arbeiten auf einer Kopie der Daten, die in den privaten Arbeitsbereich kopiert wird, wenn diese verändert werden soll.
- **copy-on-read:** Kopie bei Lesezugriff; im Unterschied zur vollständig isolierten Arbeit greift ein Entwickler bei dem ersten Zugriff auf ein Objekt/Datei auf die aktuelle Version zu, die seit Beginn der Arbeit aus einen anderen untergeordneten Arbeitsbereich veröffentlicht worden sein kann.

Zusätzlich zu dem Grad der Isolation ist der Umfang der zu propagierenden Daten interessant. Der Datenbestand, auf dem ein Entwickler aufsetzt, muß konsistent sein. Zu beachten sind dabei Abhängigkeiten zwischen den Objekten oder Dateien. Beim Check-Out/Check-In ist sicherzustellen, daß die Konsistenz nicht verletzt wird. Ein Ansatz, die Konsistenz sicherzustellen, ist die Berechnung der transitiven Hülle [44].

Setzt man die Kombination von Arbeitsbereich und Repository voraus, so lassen sich alle vier genannten Isolationsgrade realisieren. Im Fall von feinkörnig modellierten Dokumenten muß man jedoch unterscheiden, ob man die Isolation der Dokumente oder der sie modellierenden Objekte betrachtet, i.d.R. beschränken sich existierende VM-Systeme auf eine Abstraktionsebene. Des weiteren ist zu berücksichtigen, ob mehrere Entwickler kooperativ arbeiten können. Für die synchrone Kooperation von Entwicklern an einem Dokument wäre eine vollständige Isolation ungeeignet. Hierfür bietet sich eine angepaßte Form an, in der mehrere Entwickler in einem Arbeitsbereich tätig sein können. Jedoch sollte das nicht für alle Entwickler gelten. Entwickler, die an anderen Aufgaben arbeiten, sollten nicht von Änderungen betroffen sein, die nicht im Zusammenhang mit ihrer Aufgabe stehen. Von daher sollte ein VM-System die Grade *local* und *global* für Dokumente anbieten.
Betrachtet man die Isolation auf die Objekte bezogen, so ist es sinnvoll, den Isolationsgrad *copy-on-write* zu nutzen. Das hat den Vorteil, daß für Arbeitsbereiche, die für eine isolierte Entwicklung von Dokumenten vorgesehen sind, nicht alle Objekte eines Dokumentes kopiert werden müssen. Es ist in diesem Fall ausreichend, nur das Wurzelobjekt direkt in den Arbeitsbereich zu kopieren und die verbleibenden Objekte wenn sie verändert werden sollen. Für diesen Fall muß jedoch gelten, daß alle Objekte, die unverändert sind, den Isolationsgrad *global* in Bezug auf alle untergeordneten Arbeitsbereiche besitzen, damit sie von diesen zugreifbar sind. Diese Realisierung entkoppelt die Dokumente von den sie modellierenden Objekten. Somit sind weniger Objekte zu kopieren und die Nutzung der Multiple-View-Integration ist unproblematisch, da ausschließlich auf „sichtbare“ Objekte zugegriffen wird.

2.1.3.2 Entwurfstransaktionen

Die von relationalen Datenbanken bekannten Transaktionskonzepte [139, 122] haben sich beim Einsatz in Entwicklungsumgebungen als nicht brauchbar erwiesen, da sie die Arbeit und Kooperation der Entwickler durch die ACID-Eigenschaften und die längere Bearbeitungsdauer zu sehr einschränken [123, 135]. Aus diesem Grund wurden diverse Vorschläge zu neuen Transaktionskonzepten gemacht, die einzelne Eigenschaften der konventionellen Transaktionen abschwächen [12, 69]. Diese Art von Transaktionen bezeichnet man als *lange Transaktionen* oder auch als lang laufende Transaktionen. Die Einsatzbereiche sind z. B. CAD- [138] oder CASE-Anwendungen [53] für den stationären aber auch für den verteilten und mobilen Einsatz [198, 226].

Werkzeuge in Softwareentwicklungsumgebungen (SEU) können auch mehrere Arten von Transaktionen unterstützen. Platz [187] unterscheidet hierbei zwischen Entwurfstransaktionen und Werkzeugtransaktionen. Die Werkzeugtransaktionen dienen der Bearbeitung der Daten durch CASE-Werkzeuge und sind Einheiten des Recovery und des Concurrency-Control. I.d.R. wird in ihrem Rahmen ein Werkzeug ausgeführt. Die Entwurfstransaktionen dienen zur Modellierung von Entwurfsabläufen. Sie sind länger andauernd als Werkzeugtransaktionen und umfassen mehrere Werkzeugausführungen. Sie stellen die von den CASE-Werkzeugen benötigten Daten zur Verfügung.

Einige Konzepte unterstützen das Austauschen von (Teil-)Dokumenten zwischen Entwurfstransaktionen, z. B. durch Weitergabe oder Verleihen [135, 159]. Einige der Entwurfstransaktionskonzepte stellen die Daten in einer Hierarchie von Arbeitsbereichen zur Verfügung, wobei die Daten mittels Check-Out/Check-In zwischen den Arbeitsbereichen übertragen und versioniert werden [135]. Einen anderen Weg gehen Kobialka und Meyke [137]. In ihrem Konzept gibt es drei Arten von Entwurfstransaktionen, jedoch keine Arbeitsbereiche. Die Entwurfstransaktionen arbeiten direkt auf dem Repository. Dieses Konzept sieht eine Hierarchie von drei Transaktionstypen vor: Tasks, Tasks Sessions und Tool Transactions. Die Entwurfstransaktionen werden als persistente Objekte im Repository gespeichert und besitzen Beziehungen zu den Entwurfsaufgaben und den zugeordneten Entwicklern. Die Tasks dienen der Modellierung einer komplexen Aufgabe, die Task-Sessions zur Modellierung einer aktuellen Entwurfstätigkeit eines Entwicklers und die Tool Transactions koordinieren die Werkzeugausführungen. Der vollständige Verzicht auf Arbeitsbereiche hat zur Folge, daß alle Entwickler auf alle Versionen zugreifen können; eine isolierte Bearbeitung ist so nicht möglich.

Beispiele für SKM-Systeme, die Entwurfstransaktionen unterstützen sind Coo [93] mit einer offen geschachtelten Transaktionshierarchie oder EPOS [54]. Darin gibt es lange geschachtelte Transaktionen, denen jeweils ein Änderungsauftrag zugeordnet ist, mit der Möglichkeit, Änderungen vor dem Commit der Transaktion in den Arbeitsbereich einer kooperierenden Transaktion zu transferieren.

Das Entwurfstransaktionskonzept von Platz kann genutzt werden, um Dokumente, die für bestimmte Entwurfsaufgaben benötigt werden, den entsprechenden Werkzeugen zur Verfügung zu stellen. Nicht berücksichtigt ist in der vorgeschlagenen Form, eine Möglichkeit die Entwurfstransaktionen mit zusätzlichen Informationen wie z. B. den beteiligten Entwicklern oder den zugrunde liegenden Aufgaben zu verknüpfen. Hier bietet sich die Idee von Kobialka und Meyke an, die Entwurfstransaktionen als persistente Objekte in Repository zu modellieren. Diese Objekte können dann für diese Aufgabe genutzt werden. Der Verzicht auf Arbeitsbereiche, wie von Kobialka und Meyke vorgeschlagen, verhindert eine isolierte Arbeit mehrerer Entwickler und ist somit keine geeignet Lösung.

2.1.4 Kooperation

Viele VM-Systeme ermöglichen neben der Versionierung der durch sie verwalteten Daten auch die Kooperation von Entwicklern. Bei den meisten VM- oder SKM-Systemen ist die Kooperation jedoch beschränkt auf die Möglichkeiten, die durch die Arbeitsbereiche (vgl. Abschnitt 2.1.3.1), vorgegeben sind. Die einfachen VM-Systeme wie z. B. SCCS oder RCS bieten nur einen einfachen Sperr-Mechanismus an, in dem eine Datei im Repository gegen weitere Änderungen gesperrt wird, solange ein Entwickler diese zur Bearbeitung markiert hat. Erst wenn dieser Entwickler die Datei wieder freigibt, sei es durch Transferieren einer neuen Version in das Repository oder durch den Abbruch der eigenen Bearbeitung, können andere Entwickler diese Datei modifizieren [82]. Dieser pessimistische Ansatz des Concurrency-Control führt zu Verzögerungen, wenn eine Datei oder ein versioniertes Element durch einen anderen Entwickler zur selben Zeit modifiziert werden muß [200].
Das VM-System CVS setzt einen optimistischen Ansatz ein und sperrt die Dateien im Repository nicht, sondern erzwingt und unterstützt das Mischen von parallel durchgeführten Änderungen. Die Entwickler müssen lediglich Änderungen, die dieselben Zeilen in den Dateien betreffen – die Konflikte – manuell mischen. Stehen zwei Änderungen an einer Datei nicht in Konflikt, so mischt CVS die Änderungen automatisch. Zur Vermeidung von Konflikten, bietet CVS die Möglichkeit, Entwickler über bestimmte Operationen (z. B. Check-In, Check-Out für eine Änderungen) zu informieren. Das optimistische Check-Out und die Möglichkeit der (grobkörnigen) Benachrichtigung stellen eine Verbesserung gegenüber dem Sperren der Daten dar, jedoch bietet auch CVS sonst keine weitergehenden Funktionen, die die kooperative Arbeit unterstützen, die synchrone Kooperation von mehreren Entwicklern an einem Dokument ist nicht möglich.

Arbeitet eine größere Gruppe von Entwicklern an einem Projekt, welches evtl. noch in mehrere Versionen (z. B. aktuelle Produktversion, noch gewartete Vorgängerversion und die Nachfolgeversion) unterteilt ist, reicht das halbautomatische Mischen nicht aus. Hierarchische Arbeitsbereiche, das Verbessern des Team-Bewußtseins (engl.: collaborative awareness/team awareness) [7] und die Unterstützung von synchroner und asynchroner Kooperation [145] sind weitere Anforderungen, die sich aus agilen Entwicklungsmethoden [183] und auch aus den unterschiedlichen Arten von Entwickler-Gruppen ableiten lassen. Frühauf et al. [90] unterscheiden dabei folgende Arten von Teams:

- **anarchisches Team**: In dieser Art eines Teams organisiert jeder Entwickler seine Aufgaben unabhängig, es gibt evtl. Absprachen zwischen einzelnen Entwicklern, jedoch gibt es keine gemeinsame Koordination der anstehenden Aufgaben.

- **demokratisches Team**: Auch in diesem Team gibt es keine Führungsebene. Die Entscheidungen trifft jedoch nicht jeder Entwickler für sich, sondern diese werden demokratisch getroffen.

- **hierarchisches Team**: Teams dieses Typs besitzen eine Führungsebene, die aus dem Projektleiter und ggf. weiteren Experten für bestimmte Aufgabengebiete besteht. Des weiteren können die Entwickler in Gruppen eingeteilt sein, die wiederum einen Gruppenleiter besitzen. Jede Gruppe arbeitet an einer Menge von Aufgaben, die z.T. Auswirkungen auf die Arbeit anderer Gruppen haben können. Innerhalb der Gruppen kann die Arbeit wieder auf einzelne Entwickler verteilt sein, die aber auch kooperativ an einer Aufgabe arbeiten können.

- **Chief-Programmer Team**: Hierbei handelt es sich um eine spezielle Ausprägung des hierarchischen Teams, in dem der Projektleiter ebenfalls aktiv am Projekt tätig ist.

Insbesondere hierarchische Teams, in denen kleine Gruppen von Entwicklern eng miteinander kooperieren und die zu anderen Gruppen eine eher lose Kopplung besitzen, erfordern Kooperationsmöglichkeiten, die über die Kooperation durch Isolation, wie sie durch die klassischen Arbeitsbereiche angeboten wird, hinausgehen. Die Erweiterung der flachen Arbeitsbereichsstrukturen hin zu hierarchischen Strukturen mit der Möglichkeit, Daten vor Beendigung der Bearbeitung auszutauschen, ist eine Verbesserung (siehe Abschnitt 2.1.3.1), die sich speziell auf die asynchrone Kooperation konzentriert. Die synchrone Kooperation von Entwicklern wird jedoch nicht unterstützt. Auch erweiterte Konzepte von Arbeitsbereichen, die sich auf die Kooperation von Entwicklern konzentrieren und flexible Mischstrategien anbieten, z. B. [74] mit Gruppen von Areitsbereichen, unterstützen keine synchrone Kooperation.

Die durch die Arbeitsbereiche eingeführte Isolation von Entwicklern ist gut und schlecht zugleich [200]. Der Vorteil ist, daß einzelne Entwickler nicht durch die Änderungen anderer Entwickler unterbrochen und gestört werden. Der Nachteil ist jedoch die Problematik, daß die Entwickler nicht wissen, welche Änderungen ihre Kollegen durchführen. Das führt häufig bei der Integration von deren Änderungen zu Problemen, die durch ein stärkeres Team-Bewußtsein gemildert werden könnten [200].

Auch SKM-Systeme, wie z. B. Adele oder EPOS bieten keine Funktionen, die das Team-Bewußtsein steigern können, sie bieten eher fortgeschrittene Techniken zur Propagierung von Änderungen zwischen den einzelnen Arbeitsbereichen an. Ein stärker ausgeprägtes Team-Bewußtsein, worunter die Benachrichtigung über Änderungen anderer Entwickler fällt, hilft, nicht notwendige Arbeit und letztendlich auch die Entstehung von Chaos während des Projektes zu vermeiden [33].

SKM-Systeme sollten daher auch die synchrone Kooperation von Entwicklern besser unterstützen [30]. Das reduziert die Notwendigkeit, (temporäre) Varianten anzulegen, die später wieder gemischt werden müssen. Arbeiteten mehrere Entwickler an den zu mischenden Varianten, so sollten alle beteiligten Entwickler auch beim Mischen kooperieren können [21].

Die Unterstützung synchroner Kooperation von Entwicklern erfordert die Nutzung weiterer Dienste, wie sie z. B. auch im Bereich des CSCW [29] eingesetzt werden. Insbesondere sind weitergehende Maßnahmen zur Konsistenzsicherung zwischen den Werkzeugen zu berücksichtigen. Im Bereich des CASE finden sich entsprechende Konzepte, z. B. [187]. Dieses Konzept bietet einen feinkörnigen Sperr- und einen Benachrichtigungsmechanismus an. Diese sind auf feinkörnig modellierte Dokumente optimiert, jedoch ohne Versionierungsfunktionalität zu berücksichtigen.

2.1.4.1 Einsatz von Sperren

Eine in Datenbanken häufig eingesetzte Technik zur Konsistenzsicherung von konkurrierender Arbeit sind Sperren, die einem Subjekt das Recht einräumen, die gesperrten Daten zu ändern und die andere Subjekte am Lesen oder Modifizieren dieser Daten hindern. Diese Art von Sperren koordinieren die Aktionen auf *einem* Datensatz. Die in VM-Systemen eingesetzten Sperren beeinflussen oft die Aktionen auf einer Version eines Datensatzes nur indirekt. Diese Sperren kontrollieren das Anlegen oder Lesen von Versionen; nicht das Modifizieren oder Lesen der Daten. Zusammenfassend kann man sagen, daß es drei Arten von Sperren gibt, Sperren, die das Lesen, das Schreiben oder das Anlegen von Versionen verbieten. Ist keine Sperren gesetzt, so sind alle Aktionen erlaubt.

Die bekanntesten VM-Systeme, die Sperren einsetzen, sind RCS oder SCCS (s.o.). Die Sperren in DAMOKLES [2] besitzen eine vergleichbare Semantik, nur daß nicht Versionen von einzelnen Dateien, sondern Objektversionen gesperrt werden. Die zu sperrenden Objekte werden anhand

der transitiven Hülle bestimmt, so daß sich Sperren nicht ausschließlich auf ein Objekt auswirken können, sondern auf mehrere.

Erweiterte Sperr-Protokolle wie das *Multiversion Two Phase Locking* [151] oder das *C3-Sperrprotokoll* in CONCORD bzw. SERUM [99] bieten bessere Möglichkeiten, den Zugriff auf Versionen zu kontrollieren, wie z. B. Vermeidung von Deadlocks oder erweiterte Sperr-Modi, die zwischen dem Lesezugriff, dem Schreibzugriff und dem Ableiten einer Version unterscheiden. Diese Ansätze basieren auf der Idee, daß jeder Entwickler isoliert in seinem eigenen Arbeitsbereich tätig ist und sich die Kooperation auf die Integration von Änderungen aus dem übergeordneten Arbeitsbereich beschränkt oder explizit Daten zwischen Arbeitsbereichen durch Vergabe entsprechender Zugriffsrechte ausgetauscht werden können [165].

Die in Raleigh [118] verwendete Sperr-Semantik soll nicht den Zugriff auf Versionen arbeitsbereichübergreifend synchronisieren, sondern sicherstellen, daß von bestimmten Versionen keine Varianten angelegt werden, um zeitunabhängige Inkonsistenzen zu vermeiden. Aus diesem Grund werden einmal gesetzte Sperren auch nicht wieder freigegeben. Diese verbleiben auf unbestimmte Zeit an den Versionen, was eine synchrone Kooperation vollständig verhindert.

ADDD [137] nutzt Sperren zur Isolation von konkurrierenden Änderungen, da es vollständig auf Arbeitsbereiche verzichtet. Der Verzicht soll einen ungehinderten Datenfluß zwischen einzelnen Werkzeugen und Entwicklern ermöglichen. Die Sperren synchronisieren Zugriffe zwischen konkurrierenden Transaktionen. Kind-Transaktionen werden hingegen durch Sperren einer übergeordneten Transaktion nicht am Zugriff gehindert.

Zusammenfassend läßt sich sagen, daß die in VM-Systemen eingesetzten Sperren meistens zur Koordination von arbeitsbereichübergreifender Versionierung dienen, nicht aber der Koordinierung der Zugriffe innerhalb *eines* Arbeitsbereichs. Auf diesem Gebiet besteht noch Forschungsbedarf [76]. Eine Ausnahme stellt ADDD dar, welches keine Arbeitsbereiche besitzt und somit alle Versionen für alle Werkzeuge zugreifbar sind. Hierdurch ist die Isolation der einzelnen Entwickler aufgehoben. Die Konsistenz wird nur durch die Sperren sichergestellt. Die Möglichkeit, isoliert alternative Lösungen zu testen oder bei Bedarf kooperativ größere Erweiterungen durchzuführen, besteht in ADDD durch die fehlenden Arbeitsbereiche nicht.

2.1.4.2 Benachrichtigung über Änderungen

Coven [46] verwendet einen Sperr-Mechanismus, dessen Aufgabe nicht die Synchronisierung von Zugriffen ist, sondern andere Entwickler über konkurrierende Änderung zu informieren. Die Sperren werden daher auch als *Soft-Locks* bezeichnet und besitzen eher die Semantik einer Markierung als die Semantik einer Sperre. Bei der Modifikation einer Objektversion wird an dieser ein Soft-Lock gesetzt. Andere Entwickler erhalten, wenn sie diese Version ändern wollen, den Hinweis, daß diese Version bereits bearbeitet wird. Entscheidet sich der zweite Entwickler, die Version ebenfalls zu bearbeiten, erhält der erste Entwickler eine Benachrichtigung darüber. Coven unterstützt weiterhin hierarchische Entwicklergruppen, indem es eine Hierarchie von (Sub-)Repositories verwaltet, zwischen denen die benötigten Daten transferiert werden. Eine Menge von Objekten kann daher in mehreren Sub-Repositories gespeichert sein. Bei Beendigung der Bearbeitung einer Aufgabe in einem Sub-Repository wird ein Check-in der Objekte in das übergeordnete Repository durchgeführt. Sind diese Objekte in einem anderen Sub-Repository gespeichert, so erhalten die Entwickler, die in diesem Sub-Repository arbeiten, eine Nachricht über die neue Version im übergeordneten Sub-Repository. Diese können dann selbständig entscheiden, wann sie die neue Version in ihr Sub-Repository integrieren.

Andere VM-Systeme bieten keine so weitreichende Unterstützung bei der Benachrichtigung über Änderungen. CVS bietet z. B. nur wenige einfache Möglichkeiten, um Informationen darüber auszutauschen, was welcher Entwickler modifizieren will. Dieser Datenaustausch

kann jedoch von den Entwicklern durch Umgehung der regulären Schnittstellen verhindert werden. OReilly et al. [183] machen einige Vorschläge, wie sie besser Informationen darüber austauschen, welcher Entwickler welche Datei gerade modifiziert. In EPOS [173] können Informationen über Änderungen an Objekten zwischen Transaktionen ausgetauscht werden und n-DFS [88] verschickt bei Auftreten eines Ereignisses Nachrichten an einen externen Server, der dann bestimmte Aktionen ausführen kann.

Die bisher vorgestellten Systeme informieren über Änderungen an einzelnen Versionen. Die Nachrichten können aber auch eingesetzt werden, um die Entwickler darauf hinzuweisen, daß von einem Objekt referenzierte Versionen verändert wurden [43]. Die Entwickler sollten dann dieses Objekt und die referenzierten Objekte auf Konsistenz überprüfen. Diese Nachrichten können entweder automatisch durch das System zugestellt oder aber an den Objekten gespeichert werden, so daß sie explizit abgefragt werden müssen.

Die hier vorgestellten Benachrichtigungstechniken sind primär darauf ausgerichtet, über Änderungen in unterschiedlichen Arbeitsbereichen zu informieren oder die Konsistenz von Versionen sicherzustellen. Konkurrierende Änderungen innerhalb eines Arbeitsbereichs bleiben darin unberücksichtigt, so daß es keine Unterstützung der synchronen Kooperation mehrerer Entwickler in einem Arbeitsbereich gibt. Existierende VM-Systeme bieten keine Funktionalität, die mit dem Benachrichtigungsmechanismus von Platz [187] vergleichbar wäre. Für die synchrone Kooperation von Entwicklern in einem Arbeitsbereich ist es daher sinnvoll, daß von Platz vorgestellte Konzept dahingehend zu erweitern, daß es Versionen berücksichtigt.

2.1.5 Realisierungskonzepte

Bisher haben wir bei den Betrachtungen nur Konzepte berücksichtigt, die für Anwender der VM-Systeme direkt erkennbar sind. Conradi und Westfechtel [58] bezeichnen diese Konzepte auch als *externe Versionierung*. Hingegen wird die Realisierung eines VM-Systems als *interne Versionierung* bezeichnet, die wir in diesem Abschnitt kurz beleuchten. Als Schnittstelle zwischen interner und externer Versionierung kann die Integration des VM-Systems in die Betriebssystem- und Anwendungsumgebung interpretiert werden. Diese bestimmt, wie die Anwender/Anwendungen auf das VM-System zugreifen können und wie bzw. wo die versionierten Daten gespeichert sind. Leblang [143] unterscheidet drei verschiedene Typen:

1. Vaults (engl., Kellergewölbe oder Tresor)
2. virtuelle Dateisysteme
3. Standard-Repositories

Als Vault bezeichnet man einen bestimmten Bereich im Dateisystem, indem das VM-System die Versionen ablegt. Aus diesem Bereich kopiert das VM-System auf Anforderung durch die Anwender die Versionen in die jeweiligen Arbeitsbereiche der Anwender. Diese bearbeiten dann die Versionen in ihren Arbeitsbereichen. Ein Nachteil der Vaults ist, daß sie i.d.R. durch keine besonderen Vorkehrungen gegen Änderungen durch die Anwender geschützt sind. Diese können daher das VM-System umgehen und die versionierten Dateien manipulieren, verschieben oder sogar löschen. Neben Aspekten der Konsistenz der Versionen untereinander ruft das auch Inkonsistenzen zwischen den Versionen im Vault und den Versionen in den Arbeitsbereichen der Anwender hervor. Die einfachen dateibasierten VM-Systeme wie RCS, SCCS oder CVS gehören zu dieser Gruppe.

Der zweite Typ der VM-Systeme verwaltet die Versionen auf einem separaten persistenten Speicher. Die Versionen der Elemente sind über ein virtuelles Dateisystem zugreifbar, so daß

Änderungen an einzelnen versionierten Elementen nur durch das VM-System durchführbar sind. Die Verwaltung der Versionen ist für die Anwender transparent, da der Zugriff auf versionierte Daten im virtuellen Dateisystem und den unversionierten Daten im konventionellen Dateisystem identisch ist. Ein Beispiel hierfür ist das VM-System ClearCase.

Ein Vorteil dieser beiden Typen von VM-Systemen ist, daß existierende Werkzeuge auf die versionierten Daten zugreifen können, ohne daß sie von dem VM-System etwas „wissen“. Es sind somit keine speziellen Werkzeuge notwendig. Das kann aber auch ein Nachteil sein, da die Anwender neben den Werkzeugen auch mit dem VM-System interagieren müssen und somit auch Aufgaben durchführen müssen, die mit der Bearbeitung der ursprünglichen Entwicklungsaufgabe nicht in Beziehung stehen. Weiterhin kann das VM-System insbesondere bei der Differenzbestimmung zwischen verschiedenen Versionen keine Unterstüztung bieten, da die Struktur der in den Dateien gespeicherten Dokumente vollkommen unberücksichtigt bleibt. Das muß vollständig durch externe Werkzeuge realisiert werden.

Zu dem dritten Typ von VM-Systemen gehören Repositories mit Versionierungsfunktionalität. Im Gegensatz zu den beiden anderen Typen bieten diese Systeme eine Programmierschnittstelle (API) zum Datenzugriff, ein Beispiel hierfür ist PCTE. Diese Systeme kapseln die Daten und deren Versionen vollständig. Die Anwendungen können ausschließlich über die API auf die Versionen zugreifen, so daß existierende Werkzeuge angepaßt oder neue Werkzeuge erstellt werden müssen. Der Vorteil ist jedoch, daß Werkzeuge und VM-System besser integriert sind und somit die Werkzeuge einerseits den Versionierungsprozeß teilweise automatisieren können, andererseits mehr Informationen über die Versionsgeschichte vom VM-System erhalten. Das gilt insbesondere bei einer feinkörnigen Datenmodellierung.

2.1.5.1 Interne Versionierung

Die interne Versionierung betrachtet Konzepte der internen Speicherung von Versionen. Dabei gilt es unterschiedliche Aspekte zu berücksichtigen. Einerseits erfordert die Versionierung von Daten einen erhöhten Speicherbedarf, andererseits einen erhöhten Aufwand beim Zugriff auf die Daten, da z. B. nicht nur eine spezielle Datei bestimmt werden muß, sondern auch noch eine spezielle Version dieser Datei. Beide Aspekte beeinflussen sich gegenseitig:

- die kompakte Speicherung der Daten benötigt Zeit, um die gewünschte Version zu speichern und wieder zu rekonstruieren
- die vollständige Speicherung der Versionen für einen schnellen Zugriff benötigt mehr Speicherplatz

Ein VM-System kann nicht beide Aspekte „optimal“ lösen. Daher muß ein Kompromiß gefunden werden, der die speziellen Anforderungen hinsichtlich des Speicherbedarfs und der Laufzeit für den Anwendungsbereich am besten erfüllt. In existierenden VM-Systemen haben sich folgende fünf Ansätze etabliert:

1. Kopie
2. Delta
3. Operations-Logs
4. Path-Copy
5. Fat-Node

Die einfachste Art, die Versionen zu speichern, besteht darin, eine Kopie anzulegen. Das kann z. B. eine Kopie einer Datei im Dateisystem oder eine Kopie des Objektes im Repository sein, deren Dateiname bzw. dessen Objektidentifizierer um einen Versionsidentifizierer erweitert wird. Diese Methode bietet den Vorteil eines schnellen Zugriffs auf alle Versionen, da die vorhergehenden Versionen nicht rekonstruiert werden müssen. Der Speicherbedarf ist hierbei linear zur Anzahl der angelegten Versionen.

Um den benötigten Speicher zu reduzieren, speichern viele Versionsverwaltungen die Versionen eines Objektes nicht vollständig, sondern als eine Basisversion mit einer Menge von Deltas, die die einzelnen Versionen beschreiben. Bei der zustandsbasierten Versionierung muß ein Delta durch das VM-System berechnet werden. Nutzt das VM-System jedoch eine operationsbasierte Versionierung, so ergibt sich das Delta automatisch aus der Folge der durchgeführten Operationen.
Vor dem Zugriff auf eine bestimmte Version berechnet die Versionsverwaltung diese, indem sie die Deltas, die die gewünschte Version beschreiben, auf die Basisversion anwendet.

Zur internen Versionsspeicherung, die auf Deltas basiert, setzt man i.d.R. gerichtete Deltas (vgl. Abschnitt 1.2.4) ein. Bei diesen unterscheidet man zwischen Vorwärts- und Rückwärtsdeltas [58]. Vorwärtsdeltas enthalten Informationen, wie die Nachfolgeversion berechnet wird. Im Gegensatz hierzu beschreibt ein Rückwärtsdelta die Berechnung der Vorgängerversion. Daher ist die Definition der Basisversion abhängig von der Art der Deltas. Bei Vorwärtsdeltas ist die älteste Version die Basisversion, bei Rückwärtsdeltas die jüngste Version. Der Einsatz von Rückwärtsdeltas ist immer dann vorteilhaft, wenn öfters auf jüngere Versionen zugegriffen wird als auf die älteren Versionen, da der Berechnungsaufwand reduziert wird.
Eine spezielle Form der Deltas stellen Operations-Logs [136, 150] dar. Diese enthalten die Deltas nicht als berechnete Differenzen zwischen zwei Versionen, sondern als Folge von Operationen, die auf die Ausgangsversion angewendet wurden. Diese Technik läßt sich nur einsetzen, wenn das VM-System und die Werkzeuge kooperieren oder VM-System und Datenhaltung integriert sind wie in Repositories mit Versionierungsfunktionalität. Andernfalls hat das VM-System keine Möglichkeit, die ausgeführten Operationen zu bestimmen. Der Vorteil ist, daß ein Operations-Log nicht auf ein versioniertes Element beschränkt ist, sondern die Unterschiede zwischen mehreren Elementen beschreiben kann. Somit eignet sich dieses Verfahren auch zur Versionierung von Objekt-Strukturen in Repositories, im Gegensatz zu den beiden erstgenannten Techniken. Ein weiterer Vorteil besteht in der Unterstützung beim Mischen von Versionen, siehe Abschnitt 2.2.3. Der Nachteil aller Speicherungsmethoden, die auf Deltas basieren, ist der Berechnungsaufwand zur Rekonstruktion einer Version. Die benötigte Laufzeit kann für interaktive Werkzeuge nicht tolerierbar sein.

Ähnlich wie die Operations-Logs ist das Einsatzgebiet der Fat-Node- und der Path-Copy-Methode die Versionierung von Objekt-Bäumen in Repositories [42]. Diese Verfahren eignen sich daher zur Versionierung feinkörnig modellierter Dokumente.
Bei der Fat-Node-Methode speichert man alle Versionen im Objekt selbst. Die Objekt-Struktur wird beim Anlegen einer neuen Version nicht verändert (siehe Abbildung 2.12). Die einzelnen Versionen können innerhalb des Objekts als Kopie, als Delta oder als eine Kombination von beiden realisiert sein [223, 224]. Ein Beispiel für die Kombination ist, daß Revisionen als Delta und Varianten als Kopie verwaltet werden. Durch diese Kombination wird der Zugriff auf Varianten optimiert, da diese nicht durch die Deltas berechnet werden müssen. Ein Nachteil der Fat-Node-Methode ist die Sicherstellung der Konsistenz der Komponenten eines Knotens, die durch zusätzliche Mechanismen, wie z. B. Change-Packages, erreicht werden muß.
Bei der Path-Copy-Methode, die Asklund [8] als Versionspropagierung bezeichnet, legt man von allen Objekten auf dem Pfad vom veränderten Objekt bis zur Dokument-Wurzel eine neue Version an. In Abbildung 2.13 wird der Knoten `G` verändert, was zur Folge hat, daß

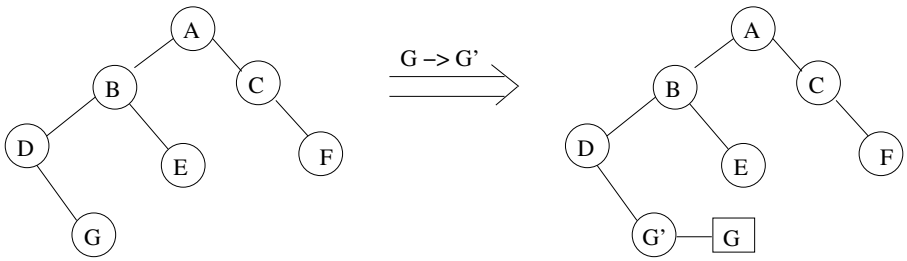


Abbildung 2.12: Beispiel für die Fat-Node Methode (aus [42])

von den Knoten G, D, B und A eine neue Version angelegt wird. Dadurch, daß vom Wurzel-Knoten eine neue Version angelegt wird, erhält man vom gesamten Dokument eine neue Version, da jede Änderung auch die Versionierung des Wurzel-Knotens bedeutet. Der Vorteil ist die Sicherstellung der Konsistenz aller referenzierten Komponenten eines Knotens [8], da ein Knoten immer exakt eine Version seiner Komponenten referenziert.

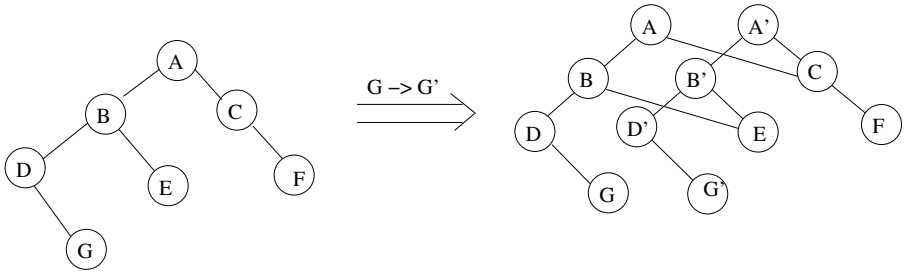


Abbildung 2.13: Beispiel für die Path-Copy Methode (aus [42])

Welches der genannten Konzepte eingesetzt wird, hängt von mehreren Faktoren ab wie z. B. Speicher- und Laufzeitanforderungen, aber auch von der Modellierung der verwalteten Dokumente. Versionen von grobkörnig modellierten Dokumenten unterscheiden sich wahrscheinlich nur in geringem Umfang. Durch Anlegen einer Kopie würde daher sehr viel Redundanz erzeugt. Bei feinkörnig modellierten Dokumenten speichert ein versioniertes Element deutlich weniger Daten als Elemente grobkörnig modellierter Dokumente. Die Wahrscheinlichkeit, daß zwei Versionen mehr Unterschiede aufweisen als Gemeinsamkeiten, ist demzufolge deutlich größer, so daß u.U. der Speicherbedarf von Kopie und Delta vergleichbar ist.

Ein weiterer Aspekt, der beim Anlegen von Version berücksichtigt werden muß, betrifft den konkurrierenden Zugriff auf einen Versionsgraphen. Sollen von einer Version zwei Nachfolgeversionen angelegt werden, kann das zu Parallelitätsanomalien führen. Sperren können das verhindern, wodurch jedoch die Nutzbarkeit auf kleine Entwicklergruppen beschränkt wird, da die Wahrscheinlichkeit des gleichzeitigen Zugriffs auf das VM-System mit der Gruppengröße ansteigt und sich die Entwickler somit gegenseitig blockieren. Daher schlagen Kelter [119] und Ritter [192] erweiterte Sperrmodelle für Versionsgraphen vor, die eine größere Parallelität erlauben.

2.1.5.2 Speicherung von Varianten

Die fünf vorgestellten Konzepte zur Speicherung von Versionen unterscheiden nicht, ob es sich bei den Versionen um Revisionen oder Varianten handelt. Letztere betrachtet Mahler [156] und stellt zwei Realisierungskonzepte vor, die weniger die interne Realisierung eines VM-System berücksichtigen als vielmehr die konzeptuelle Ebene:

Eine Methode besteht darin, alle Unterschiede in einem Element zu speichern, wobei Steueranweisungen dem VM-System mitteilen, was zu welcher Variante gehört. Er bezeichnet das als *Single Source Variant*. Diese Technik ist vergleichbar mit der Technik, die Choi und Kwon [42] als Fat-Node-Methode bezeichnen. Eine mögliche Umsetzung ohne spezielles VM-System stellt die bedingte Kompilierung von C-Quelltext dar. Bei diesem Vorgehen der Verwaltung von Varianten reduziert sich die Übersicht mit der Anzahl der Varianten. Diese Problematik adressiert Zeller [241] mit der Einführung der *Feature Logic*, die die Steueranweisungen durch logische Ausdrücke kapselt.
Die andere Methode der Verwaltung von Varianten bezeichnet Mahler als *Variant Segregation*. Jede Variante liegt als eigene Kopie vor, diese kann explizit im Dateisystem angelegt werden und rein logisch im VM-System als eigener Zweig im Versionsgraphen. Das zu lösende Problem hierbei ist, alle zusammengehörenden Varianten verschiedener Komponenten eines Aggregats konsistent zu kennzeichnen. Existierende VM-Systeme adressieren diese Problematik durch die Vergabe von abstrakten Bezeichnern, die die Varianten kennzeichnen. Diese zusätzlichen Bezeichner erhöhen jedoch die Komplexität des Namensraums der Versionsidentifizierer.

Beide Ansätze betrachten ausschließlich Varianten. Die Revisionen der einzelnen Varianten können nach einem anderem Konzept verwaltet werden. Bei Verwendung der Variant Segregation können die Revisionen z. B. als Delta im VM-System gespeichert sein.

Für feinkörnig modellierte und in einem Repository gespeicherte Dokumente erscheint lediglich die Methode der Variant Segregation geeignet. Diese Annahme ist so zu begründen, daß für die Nutzung der Methode der Single Source Variant, an jedem Attribut eines Objektes gespeichert werden müßte, welcher Attributwert für welche Variante gilt. In diesem Fall wären mehr Metadaten zu speichern als Nutzdaten, da es sich bei den Attributwerten häufig um Zahlen oder kurze Texte handelt. Die Varianten sollten daher, ebenso wie die Revisionen, unter Verwendung der Fat-Node-Methode verwaltet werden.

2.1.6 Verwandte Gebiete

Das Gebiet des Software-Konfigurationsmanagements, welches eine Disziplin des Computer Aided Software Engineering darstellt, grenzt an andere Bereiche in der Informatik an. Vergleichbare Aufgabenstellungen finden sich im Computer Aided Design (CAD). Katz [117] gibt einen Überblick der Entwicklung des Konfigurationsmanagements in CAD Datenbanken. Ein anderes Gebiet ist die Versionierung von Hypertextdokumenten [28, 237].
Problembereiche des SKM finden sich in weiteren Forschungsbereichen, die hier nur kurz erwähnt werden sollen, für eine ausführlichere Darstellung sei auf [58] verwiesen.

Temporal Databases erfassen die Veränderung von Daten über der Zeit, wobei teilweise zwischen Real- und Transaktionszeit unterschieden wird. Diese Systeme berücksichtigen keine Aspekte des Projektmanagements und bieten daher auch keine Möglichkeit, die Versionierung anhand von Aufgaben durchzuführen.

Schema-Versionierung: Viele SKM-Systeme versionieren lediglich Daten. Bei der Schema-Versionierung wird neben den Daten auch das Schema der Daten versioniert. Das ist sinnvoll, wenn sich das Schema während der Entwicklung ändert und die Daten, die unter einem anderen Schema erstellt wurden, weiterhin zugreifbar bleiben müssen. Es gibt hierfür zwei Methoden, die Daten an das neue Schema anzupassen. Man unterscheidet dabei zwischen einem lazy und eager Ansatz. Eine Anpassung nach dem lazy Ansatz konvertiert die Daten erst bei einem Zugriff. Der eager Ansatz konvertiert die Daten

sofort nach der Schemaänderung. In vielen Fällen wird nur eine einfache Form der Schema-Versionierung angewendet, die *Schema-Evolution*, dabei ist immer nur die letzte Version des Schemas gültig. Ein Zurücksetzen auf vorhergehende Versionen ist nicht möglich.

Die Versionierung des Datenbankschemas ist sinnvoll, wenn sich dieses häufig ändert. Im Bereich von UML-Editoren treten Schemaänderungen eher selten auf, da diese i.d.R. eine Überarbeitung des Werkzeugs nachsichziehen[7]. Die Schema-Versionierung ist daher für die Anwender der UML-Editoren vernachlässigbar.

Deduktive Datenbanken basieren auf einem regelbasierten Datenmodell, vergleichbar mit PROLOG. Vergleichbare Funktionalität wird bei der Intensional Versionierung eingesetzt. In vielen Fällen sind die SKM-Systeme um deduktive Komponenten erweitert worden.

Software Prozeß Management überschneidet sich in der Funktionalität mit SKM. Ein Beispiel hierfür ist der Transfer von Objekten zwischen Arbeitsbereichen. Jeder Arbeitsbereich sollte in einem SKM Konzept nicht nur als reiner Datenspeicher verstanden werden, sondern auch als Container für Dokumente, die sich innerhalb des Softwareentwicklungsprozesses im selben Zustand befinden. Um das sicherzustellen, ist der Einsatz einer Prozeßmaschine sinnvoll, die den Transfer von Daten zwischen Arbeitsbereichen anhand von Regeln steuert. Softwareentwicklungsumgebungen mit Prozeßunterstützung (z. B. [11,14,78]) können beide Funktionalitäten anbieten.

Die Integration von SKM-System und Prozeßmaschine ist eine interessante Forschungsfrage, die den Rahmen dieser Arbeit jedoch übersteigen würde.

2.2 Differenzbildung und Mischen

Ein wichtiger Bestandteil der SKM- und VM-Systeme sind Werkzeuge, die Unterschiede zwischen einzelnen Versionen von Dokumenten bestimmen und anzeigen oder Versionen mischen. Die Algorithmen zur Differenzberechnung finden nicht nur Anwendung in Werkzeugen, die für die Interaktion mit dem Benutzer gedacht sind, sondern sie werden auch von einigen VM-Systemen intern zur kompakten Speicherung der Versionen verwendet.

Die Berechnung und Anzeige der Differenzen ist abhängig von den zugrunde liegenden Dokumenttypen. Die bekanntesten und am meisten verwendeten Algorithmen und Werkzeuge arbeiten auf textuellen Dokumenten. Diese Werkzeuge sind jedoch nicht zur Differenzberechnung und Anzeige von strukturierten Dokumenten, wie z. B. UML-Diagrammen, XML-Dokumenten oder Syntax-Bäumen, geeignet [244], da deren Struktur unberücksichtigt bleibt. Für strukturierte Dokumenttypen wurden spezielle Algorithmen und Werkzeuge entwickelt.

Abhängig vom Dokumenttyp muß man zwischen der physischen Repräsentation der Dokumente und der Anzeige in den Werkzeugen unterscheiden. Zur Anzeige der Differenzen sollte die Repräsentation der Dokumente verwendet werden, in der sie auch erstellt wurden. Differenzen zwischen Diagrammen sollten daher auch als Diagramm dargestellt werden [176].

2.2.1 Anzeige von Differenzen

Abhängig von den Dokumenttypen gibt es unterschiedliche Methoden Differenzen zwischen zwei Versionen eines Dokumentes anzuzeigen. Bei textuellen Dokumenten gibt es zwei verschiedene Arten zur Anzeige der Differenzen:

[7] Die Änderung der UML-Spezifikation ist ein Beispiel hierfür.

1. Beide Versionen werden nebeneinander in zwei Spalten angezeigt. Korrespondierende Zeilen sind nebeneinander durch einfügen zusätzlicher Leerzeilen ausgerichtet; Gemeinsamkeiten und Unterschiede sind durch Nutzung verschiedener Farben markiert, ein Beispiel hierfür zeigt Abbildung 2.14. Viele Werkzeuge zur Anzeige textueller Differenzen verwenden diese Technik. Die einzelnen Werkzeuge unterscheiden sich i.d.R. nur durch die Art, wie die Unterschiede markiert werden. Einige Werkzeuge markieren die Differenzen zeichenweise (z. B. Filemerge von SUN [210]). Andere Werkzeuge kennzeichnen die Differenzen zeilenweise (z. B. ältere Versionen von tkdiff [189]). Letztere Methode hat den Nachteil, daß die Differenzen zwischen Zeilen, die sich nur in wenigen Zeichen unterscheiden, schlecht erkennbar sind.

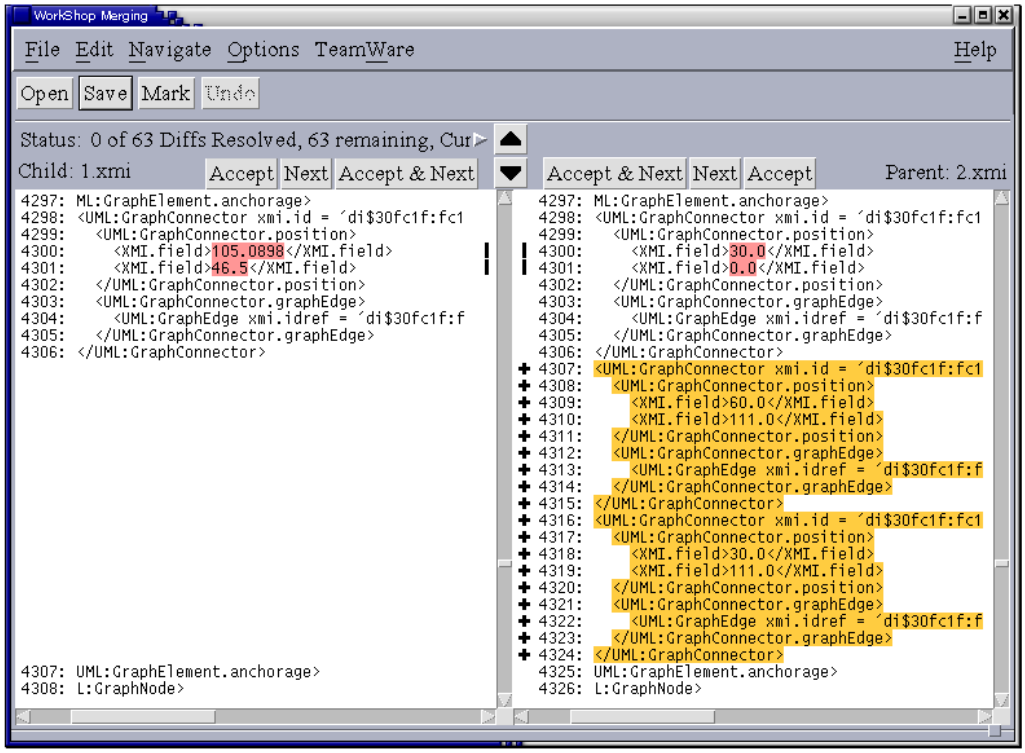


Abbildung 2.14: Werkzeug zur Anzeige textueller Differenzen

2. Bei Verwendung der zweiten Methode zur Visualisierung von Differenzen werden beide Versionen überlagert dargestellt. Die gemeinsamen Textabschnitte stellen diese Werkzeuge nur einmal dar, im Gegensatz zur ersten Methode. Die Differenzen sind farblich markiert, wie in Abbildung 2.15 dargestellt. Diese Darstellungsform kann man als Vorstufe eines Mischdokumentes interpretieren, da sie beide Dokumente einschließlich der Differenzen zusammenfaßt.

Gemeinsam ist beiden Methoden, daß unterschiedliche Farben die Differenzen zwischen beiden Dokumenten kennzeichnen. Die einzelnen Werkzeuge unterscheiden sich in einigen Details, wie z. B. der Anzeige von Zeilennummern oder der Darstellung korrespondierender Textabschnitte. Diese können nebeneinander durch Einfügen von Leerzeilen oder durch die Kombination von Hilfslinien und unabhängiges Blättern beider Dokumentversionen dargestellt werden. Beim Einsatz von Hilfslinien wird teilweise auf die farbliche Kennzeichnung verzichtet. Ein Beispiel hierfür ist Eclipse [86], siehe Abbildung 2.16.

Im Gegensatz zu textuellen Dokumenten gibt es nur wenige Werkzeuge, die Differenzen zwischen Versionen anderer Dokumenttypen anzeigen. Die meisten Veröffentlichungen diskutieren

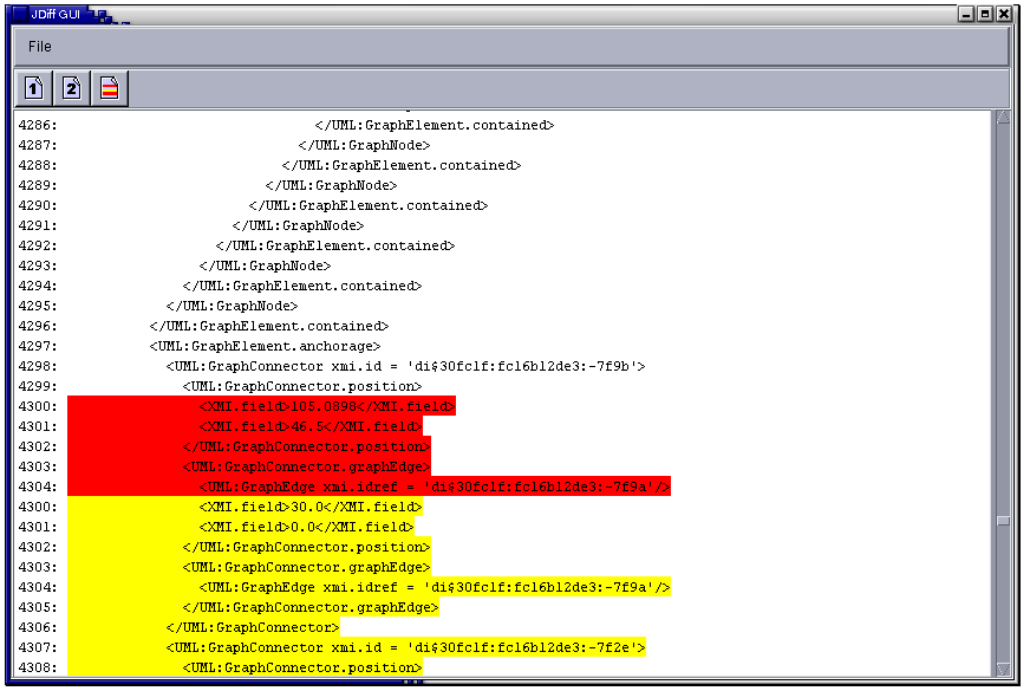


Abbildung 2.15: Werkzeug zur Anzeige textueller Differenzen

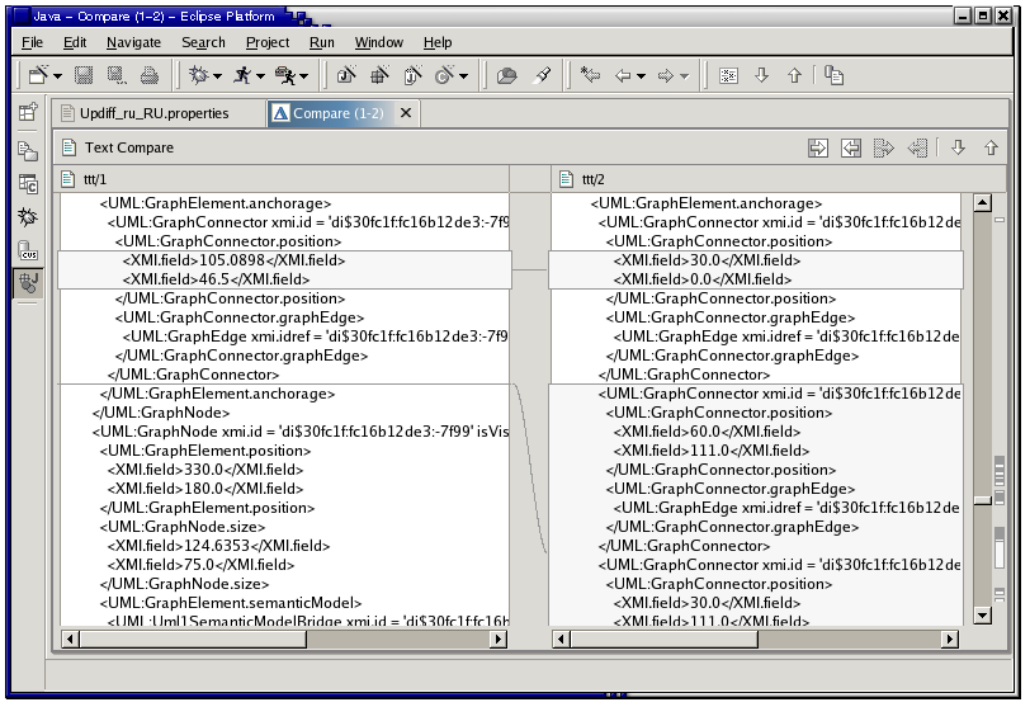


Abbildung 2.16: Werkzeug zur Anzeige textueller Differenzen

die Versionierung der Dokumente oder die Differenzberechnung und das Mischen von Versionen, jedoch nicht die Anzeige der Differenzen.

Eines der wenigen Werkzeuge, mit dem man Differenzen zwischen UML-Diagrammen anzeigen kann, ist der ModelIntegrator von IBM-Rational Rose [107]. Dieser stellt die Dokumente als Editiermodell dar. Die Differenzen in der Baumstruktur sind an den Knoten durch Symbole gekennzeichnet, Differenzen zwischen Knoten-Attributen sind tabellarisch dargestellt. Ein Bei-

spiel hierfür zeigt Abbildung 2.17. Im Gegensatz zur Werkzeugen zur Anzeige von Differenzen zwischen Texten kann der ModelIntegrator bis zu sieben Versionen vergleichen und mischen.

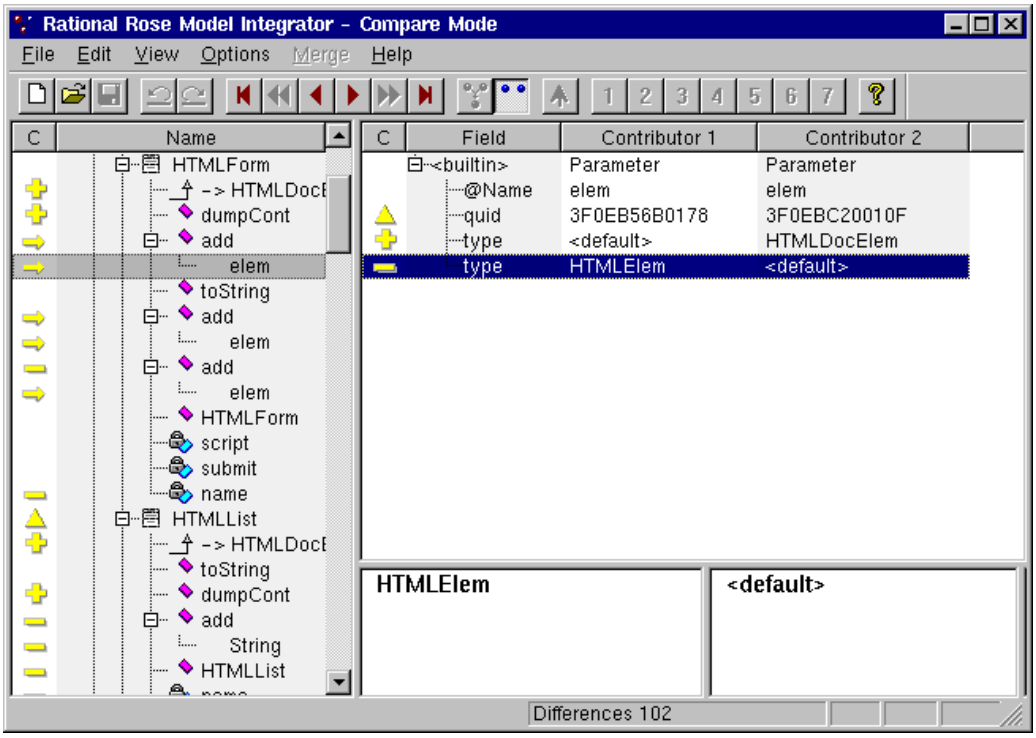


Abbildung 2.17: ModelIntegrator: Werkzeug zur Anzeige von Differenzen zwischen UML-Diagrammen

Die Nachteile der Darstellung als Editiermodell liegen in der abweichenden Repräsentation der Dokumente bei deren Erstellung und der Anzeige der Differenzen. Die Entwickler müssen von der Darstellung als Editiermodell abstrahieren, um die Differenzen im Diagramm erkennen zu können. Des weiteren enthält diese Darstellung mehr Informationen, als Diagramme üblicherweise anzeigen. Das Editiermodell enthält alle Informationen, die zur Darstellung der Diagramme notwendig sind, so auch Layoutdaten oder Knotenidentifizierer, die zur Differenzanzeige nicht relevant sind. Diese sollte nicht mehr Informationen enthalten als notwendig, um die Übersicht zu verbessern [176]. Diese Nachteile des ModelIntegrator sind wahrscheinlich auf dessen primäres Einsatzgebiet, das Mischen von Modellen, zurückzuführen.

2.2.2 Differenzberechnung

Die Berechnung der Differenzen zwischen zwei Versionen eines Dokumentes ist von dessen Typ abhängig. Je mehr Informationen über die Struktur und die speziellen Eigenschaften eines Dokumenttyps bei der Differenzberechnung berücksichtigt werden können, um so höher ist die Qualität der Differenz-Informationen. Aus diesem Grund wurden für einzelne Dokumenttypen angepaßte Differenz-Algorithmen entwickelt. Algorithmen zur Berechnung von Differenzen zwischen zwei Versionen eines Dokumentes gibt es z. B. für Textdokumente (z. B. [171]), für strukturierte Dokumente wie z. B. Bäume (z. B. [203]) oder Graphen (z. B. [155]). Diese Algorithmen sind auf bestimmte Dokumenttypen angepaßt worden, so auch für XML-Dokumente (z. B. [158]) oder UML-Modelle (z. B. [244]).
Ausgangspunkte für die Entwicklung der Differenz-Algorithmen liegen in Anwendungsbereichen, in denen Dokumente in mehreren Versionen vorliegen können, wie z. B. in der Versionsverwal-

tung von Dokumenten, im CSCW (z. B. [170]) oder in der Verwaltung und Erstellung von Metamodellen wie z. B. UML-Modelle [140, 205, 5].

2.2.2.1 Algorithmen für textuelle Dokumente

Die Einsatzgebiete von Algorithmen zur Differenzberechnung zwischen Texten sind traditionell die komprimierte Speicherung von Versionen in VM-Systemen und die Visualisierung von Differenzen in Werkzeugen. Daher sind zwei wichtige Kriterien zur Optimierung dieser Algorithmen (z. B. diff [171], Bdiff [220] oder Vdelta [87]) kurze Laufzeiten und kompakte Ergebnisse. Anhand dessen kann man die einzelnen Algorithmen miteinander vergleichen und beurteilen. Von den genannten Differenz-Algorithmen erzeugt Vdelta die kompaktesten Ergebnisse in der kürzesten Zeit [106]. Allgemein bezeichnet man die Problemstellung als *String-to-String Correction Problem*.
Diese Algorithmen basieren auf der Suche nach den längsten gemeinsamen Zeichenfolgen (engl.: longest common subsequence, LCS) in den Texten. Die Ergebnisse können als Edit-Skript ausgegeben werden, welches die notwendigen Operationen enthält, um den einen Text in den anderen Text zu transformieren. Das Optimierungsziel ist daher, ein möglichst kurzes Skript in kurzer Zeit zu berechnen.
Allen Algorithmen gemein ist die Grundannahme, daß die Dokumente aus einer Folge von Zeilen bestehen, deren Reihenfolge auch die Semantik der Texte definiert. Verschieben von Textblöcken verändert daher die Aussage des Textes und wirkt sich auf das Ergebnis der Differenzberechnung aus. Der Algorithmus von Myers [171] basiert auf einen gewichteten Änderungsgraphen, in dem der kürzeste Pfad gesucht wird (single-source shortest path).

2.2.2.2 Algorithmen für strukturierte Dokumente

Im Gegensatz zu reinen Text-Dokumenten ist die Semantik von strukturierten Dokumenten nicht durch die Folge von Text-Zeilen definiert, sondern durch das Metamodell. Daher müssen die Algorithmen zur Differenzberechnung auf der Struktur und nicht auf der externen Repräsentationen der Struktur arbeiten, da diese nach einen Laden-Speichern-Zyklus verändert worden sein kann, obwohl die Struktur unverändert ist [244].
Zur Berechnung der Differenzen zwischen strukturierten Dokumenten, insbesondere Bäumen, gibt es zwei Ansätze:

1. Berechnung von *Edit-Skripten*, die die notwendigen Operationen beschreiben, um einen Baum in den anderen zu überführen.

   Die Berechnung basiert auf der Bestimmung von Edit-Skripten, in denen jeder Operation ein Kostenmaß zugeordnet wurde. Das Skript, dessen Operationsfolge die geringsten Kosten verursacht, ist die Lösung. Diese Aufgabenstellung bezeichnet man als *Tree-to-Tree Correction Problem*. Es stellt eine Erweiterung des String-to-String Correction Problem dar.

2. Suche von korrespondierenden Knoten mit anschließendem Vergleich. Die so gefundenen Differenzen lassen sich als Edit-Skript oder Delta-Baum darstellen.

   Die Suche nach korrespondierenden Knoten in den zu vergleichenden Dokumenten ist einfach, sofern die Knoten eindeutige Identifizierer besitzen, die über die gesamte Lebensdauer eines Knotens unverändert bleiben. Üblicherweise vergeben VM-Systeme für strukturierte Daten Identifizierer bzw. können einzelne versionierte Elemente eindeutig identifizieren. Man kann sich in diesem Fall auf den Vergleich der Elemente konzentrieren. Diese Algorithmen für versionierte Daten diskutieren wir in Abschnitt 2.2.3.

Existieren keine Knoten-Identifizierer, so müssen andere Techniken zur Suche nach korrespondierenden Knoten angewendet werden [238]:

(a) (Familien von) Schlüsselattribute: Eine Menge von Attributen der Knoten wird als Schlüssel zur Identifizierung der Knoten definiert. Im Gegensatz zu den (künstlichen) Knoten-Identifizierern enthalten die Schlüsselattribute Nutzdaten.

(b) Subgraph-Isomorphismus: Die Äquivalenz von komplexen Objekten basiert auf dem Vergleich deren Struktur.

(c) Look-up-Table: Es gibt eine externe Tabelle, die die Äquivalenz von Knoten in den zu vergleichenden Dokumenten beschreibt.

(d) Fuzzy Keys: Es gibt keine Attribute, die exakt übereinstimmen. Zur Identifizierung von äquivalenten Knoten müssen die Attribute nur zu einem bestimmten Grad übereinstimmen.

(e) Negative Schlüssel: Attribute, die festlegen, daß zwei Knoten nicht äquivalent sind.

Nachdem die korrespondierenden Knoten identifiziert wurden, kann man die Differenzen zwischen den Dokumenten bestimmen. Die Dokumente können sich in der Struktur und in den Attribut-Werten unterscheiden. Unterschiedliche Attribut-Werte lassen sich durch einen einfachen Vergleich der Werte bestimmen. Bei Textattributen kann man auf Algorithmen zur Differenzberechnung zwischen Texten zurückgreifen. Differenzen, die die Dokumentstruktur betreffen, hängen von den Dokumenttypen ab.

Differenzberechnung bei Bäumen. Bei der Differenzberechnung zwischen zwei Bäumen gilt es zu berücksichtigen, ob es sich um *ordered* oder um *unordered trees* handelt. Bei ersteren ist eine Ordnung zwischen den Teilbäumen eines Knotens definiert, bei letzteren nicht.
Beide Arten unterscheiden sich in der Menge der möglichen Editier-Operationen. Durch die Existenz einer Ordnung zwischen den Teilbäumen eines Knotens in ordered trees ist deren Reihenfolge relevant für die Semantik, d.h. es muß eine Operation existieren, um die Reihenfolge der Teilbäume zu modifizieren. In unordered trees gibt es keine entsprechende Operation. Auf beiden Baum-Typen sind Operationen zum Einfügen, Löschen und Modifizieren von Knoten definiert. Bei ordered trees muß beim Einfügen eines Knotens zusätzlich dessen Position innerhalb der Teilbaum-Sequenz angegeben werden.

Die Differenz-Algorithmen für Bäume unterscheiden sich hinsichtlich der auf den Bäumen definierten Operationen. Beim Algorithmus von Selkow [203] können Knoten ausschließlich als Blätter eingefügt oder gelöscht werden. Der Algorithmus arbeitet rekursiv und betrachtet den Baum ebenenweise. Das resultierende Edit-Skript basiert auf einer Kosten-Berechnung der möglichen Edit-Skripte. Abhängig von der Vorgabe der Kosten für die einzelnen Operationen kann das Ändern oder die Kombination Einfügen und Löschen eines Knotens geringere Kosten verursachen.
Im Gegensatz zu Selkows Algorithmus nutzt Tais Algorithmus [212] einen Pre-Order-Durchlauf des Baums und basiert auf einer vergleichbaren Kosten-Berechnung. Teilweise ist Backtracking notwendig, um das Edit-Skript zu berechnen. Zhang und Shashas Algorithmus [243] traversiert die Bäume in Post-Order und benötigt kein Backtracking. Beide Algorithmen nutzen dieselben Operationen wie Selkows Algorithmus.

Die drei betrachteten Algorithmen arbeiten auf unordered trees; Barnards Algorithmus [13] basiert auf ordered trees. Für diesen Algorithmus sind die Einfüge-, Lösch- und Änderungsoperationen anders definiert, und es kommen weitere Operationen hinzu: einfügen und löschen eines Teilbaums, einfügen und löschen von inneren Knoten, vertauschen von zwei Teilbäumen,

und editieren von Teilbäumen. Das Löschen eines inneren Knotens ist so definiert, daß die Teilbäume des gelöschten Knotens zu Teilbäumen des Vater-Knotens werden. Der Algorithmus ist ansonsten vergleichbar mit dem Algorithmus von Zhang und Shasha und gehört daher zur Gruppe von Algorithmen, die das Tree-to-Tree Correction Problem lösen.

MH-Diff [39] arbeitet auf unordered trees, unterstützt jedoch mehr Operationen als die drei erstgenannten Algorithmen. Neben dem Einfügen, Löschen und Aktualisieren von Knoten können Teilbäume innerhalb des Baums verschoben werden, also einen anderen Vater-Knoten erhalten, Teilbäume können kopiert und kopierte Teilbäume wieder zusammengeführt werden. Das Löschen eines inneren Knotens resultiert darin, daß dessen Teilbäume an den Vater gehängt werden. Zur Suche von korrespondierenden Knoten erzeugt dieser Algorithmus Beziehungen zwischen allen Knoten beider Bäume und entfernt anschließend offensichtlich falsche Beziehungen. Die verbleibenden Knoten und Korrespondenz-Kanten werden als bipartiter Graph mit gewichteten Kanten interpretiert. Die Lösung besteht im Graph mit minimalen Gewichten. Die so gefundenen Knoten können anschließend verglichen werden.

Die Berechnung eines Edit-Skriptes für unordered trees ist als NP-hart nachgewiesen [39]. Kann man Details der zu vergleichenden Dokumente berücksichtigen, verringert sich die Komplexität der Algorithmen. Beispielsweise bestimmt LaDiff [40] Differenzen zwischen LaTeX-Dokumenten. Diese werden als ordered trees mit getypten Knoten interpretiert. LaDiff arbeitet in zwei Phasen, erst werden korrespondierende Knoten gesucht und anschließend die Unterschiede bestimmt. Diese werden in Form eines Delta-Baums dargestellt, um den Anwender die Identifizierung der Differenzen zu erleichtern.
LaDiff setzt keine Knoten-Identifizierer voraus, sofern welche existieren, können sie jedoch genutzt werden. Korrespondierende Knoten werden bottom-up gesucht. „Zu unterschiedliche“ Knoten werden nicht zugeordnet; es wird von einer eindeutigen Abbildung ausgegangen, so daß keine Kopien berücksichtigt werden. Diese Voraussetzungen führen zu einer besseren Laufzeit als beim Algorithmus von Zhang und Shasha.

Differenzberechnung bei XML-Dokumenten. Die oben betrachteten Algorithmen zur Differenzberechnung zwischen Bäumen sind unabhängig vom Anwendungsbereich (mit Ausnahme von LaDiff) und haben keine besonderen Anforderungen an die Dokumente. Zur Verbesserung der Qualität und der Laufzeit sollten vorhandene Informationen verwendet werden. Ein Beispiel ist die Information über existierende Knoten-Identifizierer, wie sie z. B. in XML definiert sind. Zu berücksichtigen ist, daß die Identifizierer nicht in jedem Fall genutzt oder über mehrere Werkzeug-Sitzungen hinweg konstant sein müssen.

Die Differenz-Algorithmen für XML-Dokumente basieren z.T. auf den Tree-to-Tree Correction Algorithmen, wobei sie die Eigenschaften der XML-Dokumente berücksichtigen und z. B. bei deren Versionierung nutzen. Ein Beispiel ist das XML-Warehouse Xyleme [158, 50], welches die ID-Attribute zur Identifizierung von Versionen eines Knotens nutzt.
Die einzelnen Algorithmen unterscheiden sich in deren Interpretation der XML-Dokumente als ordered [148, 158, 50] oder als unordered trees [227] und darin, ob Teilbäume innerhalb eines Dokumentes verschoben werden können [148, 158, 50] oder nicht [41, 227]. Die Begründung für oder gegen diese Operation basiert auf den Kosten, die diese Operation verursacht. Vernachlässigt man den Aufwand für einzelne Operationen, so kann man das Verschieben eines Teilbaums auf Einfügungen und Löschungen zurückführen; die Operation zum Verschieben ist also nicht notwendig. Setzt man jedoch für das Verschieben geringere Kosten als für die Einfügen-Löschen-Kombination an, so ist die Existenz einer Operation zum Verschieben von Knoten sinnvoll.

Xyleme [158, 50] setzt eine kombinierte bottom-up und top-down Strategie zur Bestimmung korrespondierender Knoten ein. Zu Beginn werden für ein Dokument eindeutige Signaturen erstellt, die durch die bottom-up und top-down Kombination im Baum propagiert und in einem

Verzeichnis verwaltet werden. Anschließend berechnet der Algorithmus für das zweite Dokument die Signaturen und gleicht diese mit den im Verzeichnis gespeicherten Signaturen des ersten Dokumentes ab. So werden identische Teilbäume identifiziert. Von den so gefundenen Teilbäumen betrachtet man als nächstes die Kinder der Vater-Knoten, da diese gute Kandidaten für weitere korrespondierende Knoten sind. Knoten, die dann noch nicht zugeordnet werden konnten, können innerhalb eines Dokumentes verschoben worden sein, was im letzten Schritt geprüft wird.
Das Ziel der Differenzberechnung in Xyleme sind möglichst kurze Laufzeiten, um das Einfügen von neuen Versionen eines XML-Dokumentes in das XML-Warehouse zu optimieren. Die einzelnen Versionen der Knoten erhalten für alle Versionen eines Knotens denselben Identifizierer.

X-Diff [227] nutzt zum Abgleich der Syntax-Bäume ein ähnliches Verfahren wie Xyleme. Für die Teilbäume werden Signaturen berechnet, anhand derer korrespondierende Teilbäume in den zu vergleichenden Dokumenten identifiziert werden. Der Abgleich der verbleibenden Teilbäume basiert auf der Berechnung der *Editing Distance*, die die Kosten für die Transformation eines Teilbaums in einen anderen Teilbaum angibt. Die Tupel von Teilbäumen mit der minimalen Editing Distance werden dann als korrespondierend angesehen. Der letzter Schritt kann durch die Einführung von Schwellwerten, die die Äquivalenz von Teilbäumen angeben, vereinfacht werden, was die Qualität des Ergebnisses verschlechtert.

Diagramme der Softwareentwicklung. Für Dokumente aus der Softwareentwicklung gibt es einige Ansätze [191, 244, 205, 5], die sich mit der Bestimmung von Differenzen zwischen den Dokumenten und dem anschließendem Mischen dieser Dokumente beschäftigen. Alle Vorschläge setzen feinkörnig modellierte Dokumente voraus. Zur Bestimmung der Differenzen suchen alle Vorschläge zuerst korrespondierende Modell-Elemente, um anschließend die Differenzen zu bestimmen. Die Unterschiede liegen im Detail.
Einige Ansätze basieren auf der Existenz von eindeutigen Identifizierern [244, 5], um die korrespondierenden Modell-Elemente zuzuordnen, oder auf eindeutigen Bezeichnungskonventionen [205]. Die so gefundenen Paare von Modell-Elementen werden anschließend verglichen und die Differenzen bestimmt. Hierbei unterscheidet [5] zwischen ausgehenden Beziehungen mit und ohne Ordnung. Die Darstellung der Differenzen unterscheidet sich in allen Ansätzen. Diese werden entweder als Delta [244], als Liste von Operationen [5] oder als ein neues Modell [205] repräsentiert. Letzterer Vorschlag unterscheidet sich von den bisher betrachteten Differenzdarstellungen darin, daß er nicht nur die Differenzen enthält, sondern ein vollständig neues Modell, einschließlich aller benötigten Vater Knoten, jedoch ohne die gemeinsamen Teilbäume. Dieser Vorschlag basiert auf der Mengen-Interpretation der Modell-Elemente. Kein Vorschlag beschäftigt sich mit der visuellen Darstellung der gefundenen Differenzen.

2.2.3 Mischen von Dokumenten

Durch die Komplexität heutiger Software-Produkte sind an deren Entwicklung nicht einige wenige Entwickler beteiligt, sondern eine größere Gruppe von Entwicklern [186, 77]. Das hat zur Folge, daß einzelne Dokumente eines Produktes parallel durch mehrere Entwickler bearbeitet werden und somit Varianten entstehen. Diese müssen i.d.R. wieder gemischt werden.

Bei den folgenden Betrachtungen über Mischalgorithmen, Begriffe und Werkzeuge beschränken wir uns auf Vorschläge für strukturierte Dokumente. Weitergehende Informationen finden sich in der Übersicht von Mens [161].

Beim Mischen treten Konflikte auf, wenn dieselben Teildokumente parallel verändert wurden. Die Anzahl der Konflikte sollte jedoch so gering wie möglich gehalten werden. Die Notwendigkeit, die Anzahl der Konflikte gering zu halten, wird z. B. durch durch die Studie [186] belegt,

woraus hervorgeht, daß große Änderungen eine Vielzahl an Konflikten hervorrufen, die nur mit großem Aufwand wieder zu lösen sind. Es gibt mehrere Möglichkeiten, die Anzahl der Konflikte zu reduzieren [161]:

- Graph-Partitionierung z. B. [150]: Die versionierten Elemente und deren Abhängigkeiten lassen sich als Graph darstellen. Die Knoten des Graphen kann man anhand der Anzahl der Beziehungen partitionieren. Innerhalb der Sub-Graphen lassen sich Konflikte finden und lösen.
- feinkörniges Daten- und Versionsmodell, z. B. [6]: Je kleiner die versionierten Elemente sind, desto weniger Konflikte können auftreten, da die Wahrscheinlichkeit für konkurrierende Änderungen an einem versionierten Element sinkt.
- Team-Bewußtsein: Durch den Austausch von Informationen wer an welchem Dokument was ändert, können die Entwickler ihre eigenen Änderungen mit den anderen Entwicklern koordinieren, um so die Anzahl der Konflikte gering zu halten, siehe auch Abschnitt 2.1.4.
- Heuristiken z. B. [31]: Speziell in Situationen, wo es einen Haupt- und mehrere Nebenentwicklungszweige gibt, sollte man die beiden folgenden Empfehlungen berücksichtigen:
  1. Im Hauptentwicklungszweig sollten nur Bug-Fixes erfolgen.
  2. Die Anzahl der Nebenentwicklungszweige sollte so klein wie möglich gehalten werden. Das Anlegen von Zweigen ist eine signifikante Änderung, die durch das Management geplant und freigegeben werden sollte.

Konflikte lassen sich nicht in jedem Fall vermeiden. Daher gibt es Techniken und Werkzeuge zum Mischen von Dokumentversionen. Etabliert haben sich 3 Vorgehensweisen (siehe Abbildung 2.18):

1. Raw Merging
2. 2-Wege-Mischen
3. 3-Wege-Mischen

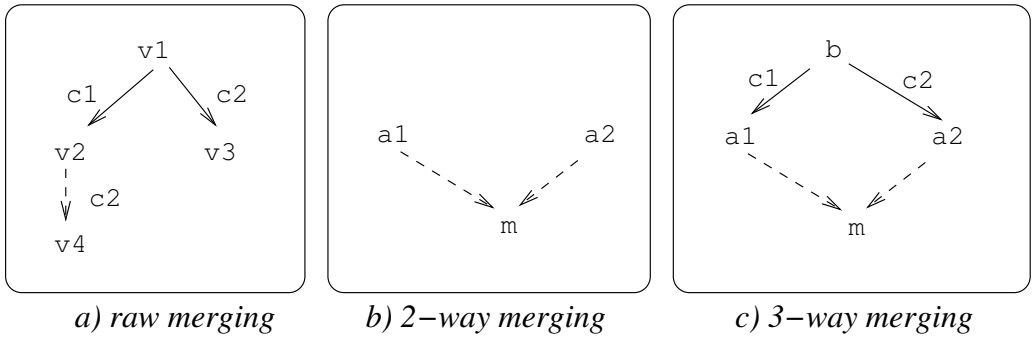


Abbildung 2.18: Vorgehensweisen zum Mischen (aus [58])

Werkzeuge, die nach dem Prinzip des Raw Merging arbeiten, wenden Änderungsinformationen an, ohne vorherige Änderungen zu berücksichtigen. SCCS nutzt z. B. diese Vorgehensweise. Im Gegensatz dazu vergleicht ein 2-Wege-Mischwerkzeug zwei Versionen miteinander und visualisiert die Differenzen. Die Auswahl muß dann der Anwender treffen. Um ihm die Arbeit

zu erleichtern, berücksichtigt ein 3-Wege-Mischwerkzeug die gemeinsame Vorgängerversion. Dabei bestimmt es nicht nur die Differenzen zwischen den beiden Versionen, sondern auch die zur ersten gemeinsamen Vorgängerversion. Diese nutzt es, um zu bestimmen, ob die Differenz ausschließlich auf die Änderung in einer Dokumentversion zurückzuführen ist, die dann in die Mischversion übernommen wird. Falls beide Dokumentversionen gegenüber der gemeinsamen Vorgängerversion geändert wurden, ist eine automatische Konflikt-Lösung nicht möglich. In diesem Fall muß der Anwender entscheiden.

Ein weiteres Unterscheidungskriterium für Mischverfahren ist, ob sie symmetrisch oder asymmetrisch arbeiten. Bei einem symmetrischen 3-Wege-Mischwerkzeug gewichtet man in Konflikt stehende Änderungen an beiden Versionen gleich. Der Algorithmus des Verfahrens kann in diesem Fall nicht selbsttätig entscheiden, welche Änderung in die gemeinsame Version übernommen werden soll. In diesem Fall muß der Anwender entscheiden. Bei einem asymmetrischen 3-Wege-Merge Verfahren wird eine Version höher gewichtet als die andere. Dadurch ist es für den Algorithmus möglich, bei einem Konflikt eine der beiden Änderungen zu wählen. Er wählt dann die Änderung der höher gewichteten Version. Das birgt jedoch das Risiko, daß die falsche Änderung in der Mischversion enthalten ist.

Benutzer-Interaktion und Layout. Beim Mischen von zwei Dokumenten müssen dem Benutzer drei Dokumente gezeigt werden, die beiden zu mischenden Dokumentversionen und die Mischversion. Bei Textdokumenten hat sich die Darstellung bewährt, die beiden zu mischenden Versionen nebeneinander darzustellen, wobei man korrespondierende Zeilen nebeneinander darstellt. Die Mischversion ist oft unterhalb dargestellt. Bei Diagrammen sind neben den inhaltlichen Änderungen auch Änderungen des Layouts zu berücksichtigen. Es gibt Ansätze, die die Layoutdaten beim Mischen nicht berücksichtigen [191]. In einem Editor, der auf EPOS basiert, werden die Layoutdaten inkrementell berechnet und nicht in der Datenbank gespeichert [96]. Hier finden Verfahren zum automatischen Layout-Anwendung, wie zum Beispiel eine Erweiterung des Sugiyama-Algorithmus [202].

Änderungen am Layout haben jedoch den Nachteil, daß die Benutzer die Ausgangsversionen nicht oder nur schlecht wiedererkennen können. Das ist jedoch nicht gewünscht. Ein Mischwerkzeug für Quelltexte [239] verfolgt daher eine andere Lösung. Die zu mischenden Dokumentversionen werden überlagert dargestellt und die Konflikte werden farblich markiert. Die Farbanzahl ist dabei auf drei Farben beschränkt, da eine Studie gezeigt hat, daß zu viele Farben die Übersicht einschränken. Vergleichbares gilt für die Nutzung von unterschiedlichen Schriftarten anstelle von Farben. Damit man die Ausgangsversionen wieder erkennen kann, werden alle Whitespaces (Leerzeichen, Tabulatoren, usw.) in die Mischversion übernommen.

Mischen und Dokumenttypen. Viele Werkzeuge und Algorithmen zum Mischen betrachten ausschließlich Textdateien, zum Beispiel SCCS, RCS und CVS. Da diese einfachen Mischwerkzeuge viele Nachteile besitzen, die darauf zurückzuführen sind, daß sie die Struktur der zu mischenden Dokumente nicht berücksichtigen, wurden verschiedene Vorschläge entwickelt, die die Syntax und einige sogar die Semantik der Dokumente beim Mischen berücksichtigen [26]. Mischwerkzeuge, die die Syntax berücksichtigen, sind jedoch auf die unterstützten Dokumenttypen, z. B. bestimmte Programmiersprachen, beschränkt. Beispiele hierfür sind [239] oder [32]. In den einzelnen Sprachen enthaltene Makros, wie z. B. das Konstrukt `#define` in der Sprache `C`, erschweren derartige Lösungen, wenn sie sie nicht sogar vollständig verhindern. Soll sogar die Semantik der Dokumente beim Mischen betrachtet werden, so stößt man schnell an Grenzen, da semantische Konflikte definiert werden müssen, die weder zu streng noch zu schwach sind [161].

Weitere Vorschläge und Werkzeuge zum Mischen von Dokumenten gibt es im Bereich von VM-Systemen, CSCW-Systemen aber auch im Bereich der Modellierung von UML-Dokumenten. Diese Bereiche betrachten wir im folgendem.

2.2.3.1 Mischen von strukturierten Dokumenten in VM-Systemen

Das Editiermodell strukturierter Dokumente bestehen i.d.R. aus (attributierten) Knoten, im folgendem als Objekte bezeichnet und Beziehungen zwischen diesen. Das Mischen von zwei Versionen eines Objektes besteht lediglich darin, die Attribut-Werte zu vergleichen (ggf. mit der gemeinsamen Vorgänger-Version) und dafür ein 3-Wege-Mischwerkzeug zu realisieren.
Gesondert zu betrachten sind die Beziehungen in diesen Dokumenten, insbesondere ob die Beziehungen getypt sind, ob sie selber Attribute oder ob sie eine Komponenten-Semantik für die beteiligten Objekte besitzen. Weiterhin kann die Identität einer Beziehung durch einen Schlüssel bestimmbar sein oder durch die beteiligten Objekte.
Bei getypten Beziehungen können nur Beziehungen gleichen Typs gemischt werden. Die Attribute einer Beziehung sind vergleichbar mit Objekt-Attributen und sind entsprechend zu mischen. Eine evtl. vorhandene Komponenten-Eigenschaft oder Identitätseigenschaft von Beziehungen kann Einfluß auf die Definition der Differenz und somit auch Einfluß darauf haben, welche Änderungen zu einem Konflikt führen und welche nicht. Diese Fragen sind jedoch abhängig von Dokumenttyp sowie der Definition der Differenz und müssen für jeden Dokumenttyp einzeln diskutiert werden. Wir beschränken uns auf eine Definition von Differenzen, die wir in Kapitel 6 geben.

Durch die Existenz von Beziehungen ergeben sich weitere Arten von Entscheidungen, die beim Mischen zu treffen sind. Einerseits, was geschieht mit Beziehungen, die in einer Version gelöscht wurden, und was ist, wenn eine Beziehung[8] in einer Version gelöscht und in der anderen Version verändert wurde, z. B. neues Zielobjekt oder Änderung von Attributen. Konflikte mit erzeugten Beziehungen können nicht auftreten, da sie nur in einer Version vorhanden sind und somit nicht in der anderen Dokumentversion verändert worden sein können, mit der Ausnahme, daß die Identität einer Beziehung durch die Zielobjekte bestimmt wird. In diesem Fall könnte in beiden Dokumentversionen eine Beziehung zwischen denselben Objekten angelegt worden sein, jedoch mit unterschiedlichen Attribut-Werten oder bei Existenz einer Ordnung zwischen den Beziehungen an einer anderen Position in der Liste der Beziehungen. Der erste Fall ist trivial: Mischen der Attribute. Im zweiten Fall, stellt sich die Frage, ob die Position Einfluß auf die Identität der Beziehung besitzt. Das muß aber wieder im Einzelfall geklärt werden.

Es gibt mehrere Möglichkeiten, eine unterschiedliche Menge von ausgehenden Beziehungen an zwei Objektversionen zu mischen:

- Mengenvereinigung, z. B. [191, 44]
- Änderung hat Vorrang (symmetrisch), z. B. [234]
- Version mit höherer Priorität entscheidet (asymmetrisch), z. B. [6]

Mischverfahren, die die Beziehungen als Mengen betrachten, wenden eine Mengenvereinigung auf die Mengen der Beziehungen von beiden Versionen an. Beziehungen zum selben Zielobjekt fassen sie zu einer Beziehung zusammen. Die grundlegende Idee dieser Verfahren ist die, daß das Löschen von nicht benötigten Beziehungen einfacher ist als das erneute Anlegen. Bei

[8] Sofern man zwei Beziehungen anhand der beteiligten Objekte oder anhand eines Identifizierers zuordnen kann.

feinkörnig modellierten Dokumenten führt das jedoch dazu, daß u.U. eine sehr große Anzahl an Beziehungen überprüft werden muß, ob sie evtl. gelöscht werden müssen.
Beim symmetrischen Mischen von Beziehungen hat die Änderung einer Beziehung Vorrang. Wenn also zum Beispiel eine Beziehung in einer Version gelöscht wurde und in der anderen Version unverändert geblieben ist[9], ist diese Beziehung in der Mischversion nicht mehr enthalten. Das steht im Gegensatz zur Mengenvereinigung, wo die in der einen Version gelöschte Beziehung wieder in der Mischversion enthalten ist. Falls eine Beziehung in beiden zu mischenden Versionen geändert wurde, so liegt ein Konflikt vor, den der Benutzer auflösen muß. Dieser Konflikt tritt beim asymmetrischen Mischen von Beziehungen nicht auf. Wenn zum Beispiel in der höher gewichteten Version die Beziehung gelöscht worden ist, so wird die Änderung der anderen Version in der Mischversion verworfen und umgekehrt, falls die Beziehung in der höher gewichteten Version verändert und in der anderen Version gelöscht worden ist.

Mischen bei zustandsbasierter Versionierung. Das in IPSEN [234] verwendete Mischverfahren basiert auf einem symmetrischen 3-Wege-Mischen für abstrakte Syntax-Graphen. Aufgrund der Ausrichtung auf Programm-Quelltexte gibt es drei Arten von Regeln:

1. Identifizierer-Regeln
2. Struktur-Regeln
3. Listen-Regeln

Die Regeln gelten dabei für die drei entsprechenden Arten von Knotentypen im Syntaxbaum, der die Struktur von Programm-Quelltexten modelliert. Um verschiedene Versionen eines Knotens identifizieren zu können, besitzt jeder Knoten im Baum einen eindeutigen Identifizierer, der auch beim Anlegen einer neuen Version nicht verändert wird. Die erste Version des Mischverfahrens arbeitete kontextfrei. Das führte beim Mischen von Dokumenten bei der Bindung von Variablenbezeichnern an die Variablen-Deklarationen zu Problemen. Für diesen Fall wurde eine kontextsensitive Erweiterung realisiert, die die Bezeichner-Bindungen gesondert betrachtet. Diese Erweiterung ist für jeden Dokumenttyp einzeln zu implementieren.

COOP/Orm [6] verwendet im Gegensatz zu IPSEN ein asymmetrisches 3-Wege-Mischverfahren für Baum-Strukturen. Zwei Versionen eines Dokumentes werden anhand von elf Regeln gemischt, von denen zwei asymmetrisch sind. Diese betreffen den Fall, daß in der ersten Version der Inhalt des Knotens geändert und in der zweiten Version der Knoten gelöscht wurde. In diesem Fall wählt der Algorithmus die Änderung der höher priorisierten Version. Im Fall, daß in beiden Versionen der Inhalt des Knotens geändert wurde, liegt ein Konflikt vor, den der Anwender lösen muß. Die Mischversion stellt in COOP/Orm lediglich einen Vorschlag dar, den der Anwender ändern kann. Die Versionen des Baumes werden ebenenweise gemischt, d.h. der Algorithmus beginnt bei der Wurzel und arbeitet eine Ebene nach der anderen ab. Der Anwender muß dabei jeden Konflikt zum Zeitpunkt des Auftretens lösen. Bei einer großen Anzahl von Konflikten stellt das einen nicht unerheblichen Aufwand dar. Abhängig vom Dokumenttyp kann der Anwender sogar nicht eindeutig entscheiden welche Lösung die Richtige ist, da diese evtl. von Dokumentteilen abhängt, die noch nicht bearbeitet wurden.
Die Inhalte der Knoten sind durch die Applikation zu mischen, der Server von COOP/Orm besitzt keinerlei Kenntnis über die innere Struktur der Knoten. Dieser verwaltet lediglich Deltas zwischen den Versionen, die durch die Applikationen bereitgestellt werden müssen. Durch diese Realisierung ist der größte Teil des Mischverfahrens in den Werkzeugen zu realisieren. Daher

[9] Diese Entscheidung kann man nur bei einem 3-Wege-Mischverfahren treffen.

muß für jeden Dokumenttyp und für jedes Werkzeug das Mischen neu oder reimplementiert werden.

Ragnarok [44] verwendet wie COOP/Orm ein asymmetrisches 3-Wege-Merge-Verfahren zum Mischen von Versionen. Jedoch wird beim Mischen von Beziehungen eine Mengenvereinigung verwendet.

Mischen bei änderungsbasierter Versionierung. Die bisher vorgestellten Mischverfahren stützen sich auf die zu mischenden Versionen ab. Dabei müssen die beiden Versionen zueinander und zur ersten gemeinsamen Vorgängerversion verglichen werden. Eine andere Klasse von Mischverfahren, speziell bei SKM-Systemen, die eine änderungsbasierte Versionierung auf Basis von Änderungsoperationen unterstützen, mischen nicht die Zustände, sondern Folgen von Änderungsoperationen. Dazu gehören das History Merging aus COACT [136, 228] und das operation-based Merging in CAMERA [149, 150]. Diese Verfahren berechnen Abhängigkeiten der Operationen voneinander und bestimmen daraus evtl. Konflikte, die der Benutzer auflösen muß. Nachteilig auf die Laufzeit wirken sich hierbei lange Operationsfolgen oder feinkörnig modellierte Dokumente[10] aus [161].

Durch die Versionierung auf Basis von Änderungsoperationen in COACT [136] haben sich die Autoren für ein Mischverfahren auf Basis der Sequenz der Änderungsoperationen einschließlich deren Parameter entschieden. Sie bezeichnen das Verfahren als History Merging. Die Anwender arbeiten in Arbeitsbereichen. Zwischen den Arbeitsbereichen werden nicht die Daten, sondern die im Arbeitsbereich ausgeführten Operationsfolgen ausgetauscht. Durch die Zuordnung eines eindeutigen Identifizierers zu den ausgeführten Operationen ist es möglich, eine Operation, deren Änderung sich in zwei Arbeitsbereichen befindet, eindeutig zu identifizieren. Beim Mischen bestimmt der Algorithmus nicht die Unterschiede zwischen zwei Versionen, sondern Abhängigkeiten zwischen den ausgeführten Operationen. Die Liste der Abhängigkeiten wird anschließend verwendet, um festzustellen, ob zwei Operationsfolgen in Konflikt stehen. Falls ein Konflikt festgestellt wurde, werden dem Benutzer zwei alternative Operationsfolgen angeboten, woraus er eine auswählen muß. Eine Operation, die in den beiden zu mischenden Operationsfolgen vorhanden ist, verursacht dabei keinen Konflikt, da sie anhand des Identifizierers erkannt wird. Wenn kein Konflikt auftrat, wird die Operationsfolge auf der anderen Version wiederholt. Diese Realisierung des Mischens durch Operationsfolgen bietet sich für diese System an, hat jedoch den Nachteil, daß der Benutzer anhand von evtl. komplexen Operationsfolgen entscheiden muß, wie die Dokumente zu mischen sind. Der Benutzer muß also von den Operationsfolgen abstrahieren und deren Auswirkungen auf die Dokumente nachvollziehen. Werkzeugunterstützung zur Umsetzung der Operationsfolge in die resultierenden Dokumente ist hierbei unerläßlich.
Um Speicherplatz zu sparen, werden Operationen, die sich gegenseitig aufheben, aus der Operationsfolge gelöscht. Wenn die beiden inversen Operationen in zwei unterschiedlichen Werkzeugläufen aufgerufen werden, kann durch das Löschen der Operationen der Zustand zwischen den beiden Werkzeugläufen nicht wiederhergestellt werden, ein vollständiges Undo ist daher mit diesem Verfahren nicht realisierbar.

Ein ähnlicher Ansatz wie in COACT wird in CAMERA [149, 150] verwendet. Die Autoren bezeichnen dieses Verfahren als operation-based Merging. Der wesentliche Unterschied zum History Merging besteht darin, daß der Benutzer mehr Möglichkeiten bei der Konflikt-Behandlung besitzt. Er kann die Reihenfolge der Operationen verändern und die Operationsfolge editieren, einschließlich des Löschens einer Teilfolge. Hier gelten i.w. dieselben Einschränkungen wie für das Mischverfahren von COACT.

[10] Das führt wieder zu langen Operationsfolgen.

2.2.3.2 Mischen von Dokumenten in CSCW-Systemen

Die bisher vorgestellten Verfahren zum Mischen von Versionen sind auf eine Methode zum Mischen beschränkt, sie sind z. B. entweder symmetrisch oder asymmetrisch und arbeiten nach festgelegten Regeln. Sie sind somit nicht konfigurierbar. Munson und Dewan [170] stellen ein Framework zum Mischen von strukturierten Dokumenten vor. Dieses Framework bietet die Möglichkeit, unterschiedliche Objekttypen auf verschiedene Arten zu mischen. Das vorgestellte Konzept basiert jedoch nicht auf einem SKM-System und ist daher lediglich ein 2-Wege-Mischverfahren, wodurch mehr Konflikte auftreten, die der Anwender auflösen muß. Mit dem Konzept kann man strukturierte, attributierte Objekte unterschiedlichen Typs mischen.
Für jeden Objekt-Typ definiert man in einer Matrix die möglichen Konflikte und wie diese aufzulösen sind. Die Spalten und Zeilen der Matrix entsprechen den Änderungsoperationen (Attribut geändert, Sub-Objekte eingefügt, gelöscht usw.), die Einträge legen die Mischsemantik fest. Folgende Einträge sind möglich:

- Wähle bestimmte Änderung eines der beiden Ursprungsdokumente
- Frage Benutzer
- Wende Merge mit Matrix vom Sub-Objekttyp an
- Wende Funktion an, um Wert für Mischversion zu berechnen oder zur Auswahl, welche Änderung der beiden zu mischenden Dokumente in die Mischversion übernommen werden soll. Mögl. Funktionen sind:
  - Vordefiniert: MIN, MAX, Summe, ...
  - Selbstgeschrieben: z. B. Validierung der Ergebnisse

Für einige bekannte Mischverfahren bieten die Autoren vordefinierte Matrizen an.

Das CSCW-System CoNus [21] besitzt eine Komponente, die das Mischen von Dokumenten einschließlich des Lösens von Konflikten unterstützt. Das zugrundeliegende Datenmodell besteht aus Objekten, die über Beziehungen verknüpft sind. Es wird von einer feinkörnigen Modellierung ausgegangen, da das beim Lösen von Konflikten mehr Flexibilität bietet, als es bei einer grobkörnigen Modellierung möglich wäre.
Das Mischen basiert auf dem Prinzip des History Merging. Alle beteiligten Anwender werden beim Mischen nach der Konfliktlösung gefragt. Das Werkzeug bietet Möglichkeiten zur direkten Kommunikation zwischen den Anwendern. Ansonsten gilt das für das History Merging Gesagte.

2.2.3.3 Mischen von UML-Dokumenten

Je mehr Informationen beim Mischen von Dokumenten berücksichtigt werden können, desto höher ist die Qualität des gemischten Dokumentes. Das gilt insbesondere bei UML-Dokumenten, da diese in unterschiedlichen Repräsentationen vorliegen (siehe Abschnitt 1.2.2.2) und einige Informationen ausschließlich für einige dieser Repräsentationen notwendig sind. Ein Beispiel hierfür sind die Layoutdaten von Diagrammen.
Die Layoutdaten sind beim Mischen von UML-Dokumenten nur insofern von Interesse, daß das resultierende UML-Dokument ein ähnliches Layout aufweisen sollte wie eines des beiden Ausgangsdokumente. Beim Mischen sollten die Layout-Informationen unberücksichtigt bleiben, da kleine Änderungen am Layout eine Vielzahl an Mischkonflikten hervorrufen können.

Beim Mischen von UML-Dokumenten können unterschiedliche Arten von Konflikten auftreten. Die am leichtesten zu identifizierende Art eines Konfliktes sind konkurrierende Änderungen am selben Modell-Element. Diese Konflikte können durch einen einfachen 3-Wege-Misch-Algorithmus identifiziert werden. Neben diesen Konflikten gibt es noch syntaktische Mischkonflikte. Ein Beispiel hierfür ist das Hinzufügen einer Klasse in zwei Versionen eines Klassendiagramms, die später gemischt werden. Die beiden Klassen haben denselben Bezeichner, besitzen jedoch eine unterschiedliche Schnittstelle. Nach dem Mischen der beiden Versionen enthält das Mischdokument zwei Klassen mit demselben Bezeichner. Das widerspricht der Konsistenzanforderung, daß die Klassenbezeichner eindeutig sein müssen. Die Frage ist, ob syntaktische Konflikte auch beim Mischen erkannt und gelöst werden müssen oder ob das Aufgabe des Anwenders und spezieller Analysatoren ist.
Syntaktische Konflikte treten nicht ausschließlich beim Mischen von Dokumenten auf, sondern können auch im Rahmen der „normalen“ Entwicklung auftreten, so daß diese keine spezielle Problematik im Kontext des Mischens von Dokumentversionen sind. Deren Definition und die Analyse von Dokumenten in Bezug auf diese Konflikte ist eine komplexe Aufgabenstellung, die den Rahmen dieser Arbeit übersteigen würde, daher wird sie nicht weiter betrachtet.

Das Mischen von UML-Dokumenten ist nicht ausschließlich eine Aufgabenstellung von VM-Systemen. Mit dieser Problematik beschäftigt sich auch die Modell-Transformation. Das Ziel hierbei ist die Transformation von vorgegebenen Modellen in andere Modelle oder Modell-Typen. Ein Beispiel hierfür ist das Zusammenführen von mehreren Zustandsübergangsdiagrammen in ein gemeinsames Diagramm [92]. Dieser Vorschlag bezieht sich nur auf einen Diagrammtyp. Eine andere Art der Transformation beschäftigt sich mit der Konvertierung von bestimmten Diagrammtypen in andere Typen, wie z. B. die Konvertierung von Szenario-Modellen in Verhaltensdiagramme [201, 206]. Tritt in mehreren Modellen dasselbe Element auf, so können diese mit Correspondence-Beziehungen [48] verknüpft werden. Darauf aufbauend können die Diagramme dann zusammengeführt werden. Die Autoren sehen darin eine Unterstützung von konkurrierender Arbeit durch mehrere Entwickler.

Der Vorschlag von Selonen [205] führt Mengen-Operationen für UML-Modelle ein. Eine von diesen Mengen-Operationen ist die Vereinigung, die mehrere Modelle gleichen Typs zusammenführt. Die Mengen-Operation setzt eindeutige Bezeichner voraus, um die einzelnen Modell-Elemente identifizieren zu können. Der Unterschied zwischen der Vereinigung und dem Mischen von UML-Modellen liegt darin, daß eine Vereinigung die Löschoperation ignoriert. Das Vereinigungsmodell enthält daher alle Modell-Elemente der Ursprungsmodelle. Ein weiterer Grund hierfür liegt darin, daß die Modelle unversioniert sind und somit nicht entschieden werden kann, ob ein Element in einem Modell neu erzeugt oder im anderen Modell gelöscht wurde.

Das Versions Modell für Software Diagramme von Rho und Wu [191] verwendet ein symmetrisches 3-Wege-Mischverfahren. Es nutzt Regeln zum Mischen der Struktur, den Attributen an Knoten und Kanten sowie zum Mischen von mengenwertigen Attributen. Die Mischversion enthält alle Objekte der beiden zu mischenden Versionen, auch wenn ein in der gemeinsamen Vorgängerversion enthaltenes Objekt in einer Version gelöscht worden ist. Die Autoren begründen diese Realisierung damit, daß es einfacher ist, ein Objekt zu löschen, als es wieder anzulegen. Der Anwender wird beim Mischen über das Vorhandensein solcher möglicherweise „nutzloser“ Objekte informiert. Durch die Trennung in Entwurfs-, Quasi-Entwurfs- und Layoutdaten ist es möglich, die Layoutdaten beim Mischen zu ignorieren.

Neben der Anzeige von Differenzen ist das Mischen von UML-Modellen die Hauptaufgabe des ModelIntegrators von IBM-Rational Rose [107]. Differenzen und Konflikte werden in einer Kombination aus Baumdarstellung und tabellarischer Anzeige dem Anwender präsentiert. Die Anzeige umfaßt alle Daten, die Rose intern zur Anzeige und Verwaltung der Dokumente benötigt, einschließlich von Meta-Daten wie Layout oder Knoten-Identifizierern. Der Anwender

muß von diesen Daten abstrahieren und den Bezug zu den Ausgangsdokumenten herstellen, was bei großen Dokumenten eine komplexe Aufgabe ist. Es besteht nicht die Möglichkeit, die angezeigten Konflikte zu gruppieren oder bestimmte Arten von Konflikten, wie z. B. Konflikte im Layout, auszublenden.

Ein weiterer Vorschlag zum Mischen von UML-Modellen [5] basiert auf einer mit dem History Merging vergleichbaren Technik. Die Differenzen zwischen den Modellen werden als Listen von Operationen gespeichert. Beim Mischen von zwei Modellen werden die Listen traversiert und eine weitere Liste erstellt, die die Mischanweisungen enthält. In dieser Liste müssen Konflikte gelöst und doppelte Operationen entfernt werden. Das Lösen der Konflikte geschieht teilweise automatisch unter Berücksichtigung der Syntax. Ein von den Autoren genanntes Beispiel ist das Layout, jedoch ohne das Beispiel zu vertiefen. Ein wesentlicher Unterschied zu anderen Mischverfahren liegt darin, daß hier zwischen geordneten und ungeordneten Mengen von Beziehungen unterschieden wird.
Die Bestimmung der Differenzen basiert auf eindeutigen Identifizieren an den Modell-Elementen. Durch Anwenden der Misch-Operationsliste werden zwei Modelle zusammengeführt.

Es gibt nur wenige Werkzeuge, die auf das Mischen von UML-Diagrammen spezialisiert sind und Fragen das Layout betreffend berücksichtigen. Funktionen, um nur bestimmte Arten von Konflikten anzuzeigen, die Konflikte zu gruppieren, z. B. anhand der Aufgaben, in deren Kontext die korrespondierenden Änderungen durchgeführt wurden, sind nicht vorhanden.

2.3 PCTE, H-PCTE und PI-SET

PCTE (Portable Common Tool Environment) ist ein ECMA und ISO Standard für Integrationsrahmen von Softwareentwicklungsumgebungen (SEU) [109, 225, 35, 213]. Es stellt Basisdienste zum Bau von SEU zur Verfügung. Hierzu zählen u.a.:

- Datenverwaltung: Objektmanagement und Versionsmanagement
- Transaktionsverwaltung
- Prozeßmanagement einschl. Interprozeß-Kommunikation
- Zugriffskontroll-Mechanismen
- Verteilung von Prozessen und Objekten
- Benutzer- und Benutzergruppen-Verwaltung

Alle angebotenen Dienste sind über definierte Schnittstellen (API) [110, 111] zugreifbar. Die Kernidee von PCTE ist, alle Dienste zentral zur Verfügung zu stellen und so den Werkzeugen einer SEU eine homogene Arbeitsumgebung zu bieten, die unabhängig von der eingesetzten Rechnerarchitektur ist. Das soll einerseits die Portierung von Werkzeugen zwischen unterschiedlichen Rechnerarchitekturen erleichtern, andererseits den einfachen Datenaustausch zwischen den Werkzeugen ohne redundante Datenspeicherung ermöglichen [215].

Die in dieser Arbeit vorgeschlagenen Konzepte sind in und auf H-PCTE [124, 121] realisiert. H-PCTE (*h*ochperformantes oder *h*auptspeicherbasiertes PCTE) ist eine partielle Implementierung von PCTE, die unter besonderer Berücksichtigung der Anforderungen [125] zur Speicherung und zum konkurrierenden Zugriff auf feinkörnig modellierte Daten entworfen und implementiert wurde. Eine der Anforderungen, die auf interaktive Werkzeuge zurückzuführen sind, ist der hochperformante Zugriff auf die Daten. Dieser wird durch die hauptspeicherbasierte Datenhaltung von H-PCTE erreicht. Ein Nachteil der hauptspeicherbasierten Datenhaltung

ist der begrenzte Adreßraum, in dem maximal 4GByte an Daten adressierbar sind. Die Realisierung eines 64-Bit Speicherkonzeptes [174] löst diesen Nachteil auf absehbare Zeit. Auf dieser Basis wurden diverse Werkzeuge [66,131,166,25] und zwei Abfragesprachen [100,98] entwickelt.

Aufgrund der besonderen Anforderungen einer OMS-orientierten Werkzeugarchitektur weichen insbesondere die Transaktionsverwaltung und das Prozeßmanagement von H-PCTE von den Definitionen des PCTE Standards ab. Im weiteren betrachten wir die Dienste von H-PCTE, die für die Realisierung des in dieser Arbeit vorgeschlagenen Konzepts relevant sind.

2.3.1 Das Datenbankmodell

Das Datenbankmodell von PCTE ist operational objektorientiert und basiert auf einem erweiterten ER-Modell mit attributierten Objekt- und Link-Typen. Die Objekte sind über gerichtete Links miteinander verbunden, die i.d.R. paarweise existieren. Über die API greift man *navigierend* auf die Objekte zu, d.h. von einem *Referenz-Objekt* des OMS (es gibt mehrere Referenz-Objekte: das Wurzel-Objekt des OMS: *Common-Root* als „_“ dargestellt und das *home*-Objekt des Benutzers, dargestellt als „∼“) ausgehend verfolgt man einzelne Links und erhält so Zugriff auf weitere Objekte, von denen weiter navigiert werden kann. Diese Objekte sind mittels *Objekt-Referenzen* identifizierbar. Da die Links paarweise auftreten, kann man auch rückwärts navigieren.
Alle Objekte, Links und Attribute sind in PCTE durch ihren Typ definiert:

Objekt-Typ: legt die Menge der Attribute, die Menge der zulässigen ausgehenden Link-Typen und die Menge der Eltern-Typen fest. Die Gesamtheit der Objekt-Typen bildet eine Typhierarchie, deren Wurzel der Objekt-Typ `object` ist.

Link-Typ: legt die folgenden Eigenschaften fest:

- Minimale und maximale Anzahl von Instanzen eines Link-Typs pro Objekt
- Liste von Schlüsselattribute (Wertebereiche: `Natural` und `String`)
- Menge von Nicht-Schlüsselattribute
- Typ des reversen Links
- Menge von Zielobjekt-Typen
- Kategorie: definiert semantischen Eigenschaften
- ...[11]

Von einem Objekt kann eine definierte Anzahl an Links eines Typs ausgehen, die anhand ihrer Schlüsselattribute unterschieden werden. Einzelne Links referenziert man anhand des *Linknamens*, der sich aus der Liste der Schlüsselattribute und dem Link-Typnamen zusammensetzt.

Zusätzlich zu den Schlüsselattributen können Links auch Nicht-Schlüsselattribute besitzen, die zur Speicherung der Daten, die die Beziehung betreffen, dienen. Der Link-Typ legt die Kategorie fest, diese bestimmt folgende semantische Eigenschaften der Links:

1. *Komponenteneigenschaft*: Das Zielobjekt ist eine Komponente des Ausgangsobjektes. Das Ausgangsobjekt einschließlich aller Zielobjekte von Links mit dieser Eigenschaft bezeichnet man als *komplexes Objekt*. Einige Operationen (`object_copy`,

[11] Einige weitere Eigenschaften, die im Rahmen dieser Arbeit irrelevant sind.

`object_move, object_delete, object_list_links`) der API von PCTE arbeiten auf dem gesamten komplexen Objekt.

Wird ein Objekt oder eine Objektstruktur von zwei oder mehr Links mit Komponenteneigenschaft referenziert, so handelt es sich bei diesem Objekt(-Struktur) um eine *gemeinsame Komponente*. Gemeinsame Komponenten gehören semantisch zu beiden komplexen Objekten. Daher hat das Löschen eines komplexen Objektes keine Auswirkungen auf gemeinsame Komponenten. Diese werden nicht mit gelöscht.

2. *Existenzeigenschaft*: Ein Link dieser Kategorie sichert die Existenz des Ziel-Objekts. Ein Objekt muß mindestens durch einen Link dieser Kategorie referenziert werden, damit es existieren kann. Löscht man den letzten Link dieser Kategorie, wird das Zielobjekt ebenfalls gelöscht.
3. *Referentielle Integrität*: Die Existenz des Zielobjekts ist sichergestellt.
4. *Relevanz für das Ausgangsobjekt*: Das Anlegen oder Löschen eines Links führt zu einer Änderung des Ausgangsobjektes.

Tabelle 2.1 gibt einen Überblick dieser Eigenschaften in Abhängigkeit von der Kategorie.

Kategorie	Semantische Eigenschaft			
	Komponenten-Eigenschaft	Existenz-Eigenschaft	Referentielle-Integrität	Relevanz für das Ausgangsobjekt
Composition	+	+	+	+
Existence	-	+	+	+
Reference	-	-	+	+
Implicit	-	-	+	-
Designation	-	-	-	+

Tabelle 2.1: Kategorien der Link-Typen

Attributtyp: Die Attribute sind in PCTE atomar und durch den Attribut-Namen, den Wertebereich sowie den initialen Wert beschrieben. Eine Übersicht der Wertebereiche und Standard-Initialwerte gibt Tabelle 2.2.

Wertebereich	Initialer Wert
Integer	0
Natural	0
Boolean	false
Time	1980-01-01T00:00:00Z
Float	0.0
String	leerer String
Enumeration	1. Eintrag

Tabelle 2.2: Wertebereiche der Attributtypen

Sichten und Typdefinitionen. Die einzelnen Objekt-, Link- und Attributtypen werden in *Schema-Definition-Sets* (SDS) definiert und *selbstreferentiell* durch Objekte und Links im OMS gespeichert. Ein SDS kann Typen aus anderen SDS importieren und diese erweitern. Daraus

folgt, daß die Definition *eines* Typs vollständig in einem SDS enthalten oder aber über mehrere SDS verteilt sein kann und daß weitere SDS neue Eigenschaften ergänzen können.

Jedes Werkzeug greift auf das OMS unter Verwendung einer Sicht zu. Die Sichten bezeichnet man als *Arbeitsschema*. Ein Arbeitsschema definiert durch eine Folge von SDS eine Teilmenge aller Typen des OMS, die in dem Werkzeug sichtbar und somit zugreifbar sind. Existieren in mehreren SDS Typen mit den selben Bezeichnern, kann man sie durch ihren *langen Typnamen*: `SDS-Name-Typname` unterscheiden.
Ein Arbeitsschema kann man als einen Filter interpretieren (siehe Abbildung 2.19), der aus der Gesamtheit aller Objekte nur diejenigen filtert, die in einem Werkzeug benötigt werden.

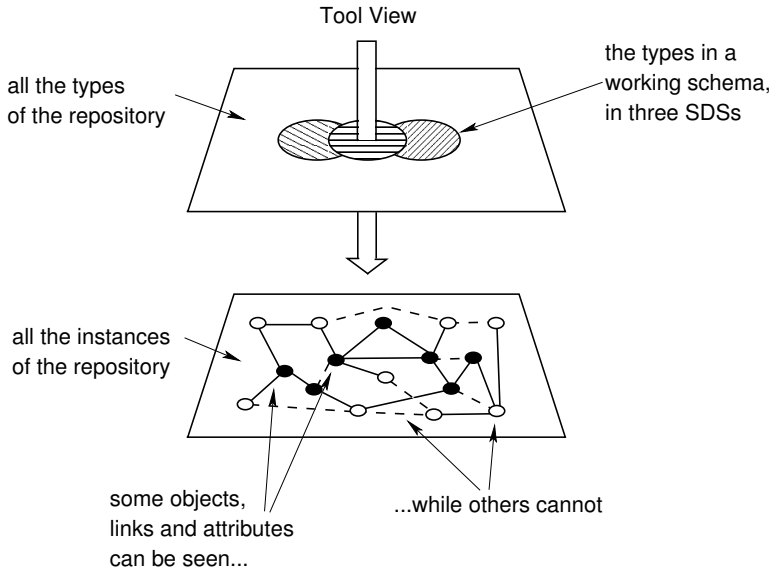


Abbildung 2.19: Das Arbeitsschema als Filter [225]

2.3.2 Zugriffskontrollen

In PCTE sind verschiedene Arten von Zugriffskontrollen definiert, jedoch unterstützt H-PCTE nur Typrechte und gruppenorientierte diskretionäre Zugriffskontrollen. PCTE kennt darüber hinaus noch Informationsflußkontrollen und die Protokollierung sicherheitsrelevanter Ereignisse.

Die Typrechte gelten für alle Instanzen eines Typs und sind in den SDS festgelegt. Sie definierten, in welcher Art und Weise auf die einzelnen Typen zugegriffen werden kann, im wesentlichen unterscheidet man zwischen dem lesenden, schreibenden und navigierenden (ausschließlich bei Links) Zugriff.

PCTE kennt eine hierarchische Benutzergruppenstruktur, die einzelne Benutzer und Gruppen zu übergeordneten Gruppen zusammenfaßt. Die Benutzer und Benutzergruppen sind selbstreferentiell in H-PCTE gespeichert. Des weiteren unterscheidet PCTE 23 verschiedene Zugriffsmodi für die Objekte und Links. In *Access Control Lists* (ACL) können die Zugriffsrechte für jedes Objekt in Abhängigkeit von der Benutzer(-Gruppe) festgelegt werden. Damit ist ein sehr genaue Festlegung der Zugriffsrechte jedes einzelnen Benutzers möglich.

2.3.3 Verteilung und Segmentierung

In PCTE können Prozesse und Teile der Objektbank, die auf *Volumes* gespeichert sind, zwischen gleichberechtigten Workstations verteilt werden. Im Gegensatz hierzu basiert H-PCTE auf einer Client/Server-Struktur, in der Datenbankprozesse Teile der Objektbank, die hier als

Segmente bezeichnet werden, laden können. Eine Verteilung der Datenbank-Prozesse selbst ist nicht möglich.
Ein Segment kann explizit durch Anforderung eines Datenbank-Prozesses in den Server oder in einen Klienten geladen werden, sofern es noch nicht geladen ist. Auf Segmente, die im Server geladen sind, kann jeder Klient zugreifen. Ist ein Segment stattdessen in einem anderen Klienten geladen, so hat ein Klient keinen Zugriff auf die dort gespeicherten Daten.

Verteilung in JH-PCTE. Der Standard von PCTE sieht nur eine C-API und eine ADA-API vor. In H-PCTE wurde zusätzlich die API in Java nachgebildet. Hierbei wurde soweit wie möglich auf eine direkte Umsetzung der C-API Wert gelegt, so daß die Definitionen der C-API auch für die Java-API gelten, die als JH-PCTE bezeichnet wird.
JH-PCTE ist zweigeteilt: (1) ein Java-Paket, welches die Zugriffsmethoden auf das OMS bereitstellt, und (2) den *JH-PCTE-Server*. Java-seitig sind Methoden implementiert, die die C-API nachbilden und alle Daten über eine Netzwerkverbindung an den JH-PCTE-Server verschicken. Dieser nimmt die Daten entgegen und reicht diese an die C-API weiter, so daß er aus Sicht des H-PCTE-Servers ein Klient ist. Diese Architektur hat Auswirkungen auf die Segment-Verwaltung. Die Segmente können nicht in die Java-Klienten geladen werden, sondern nur in den JH-PCTE-Server, daher kann man die Klienten des JH-PCTE-Servers als *Thin-Clients* [114] ansehen. Abbildung 2.20 verdeutlicht diese Architektur.

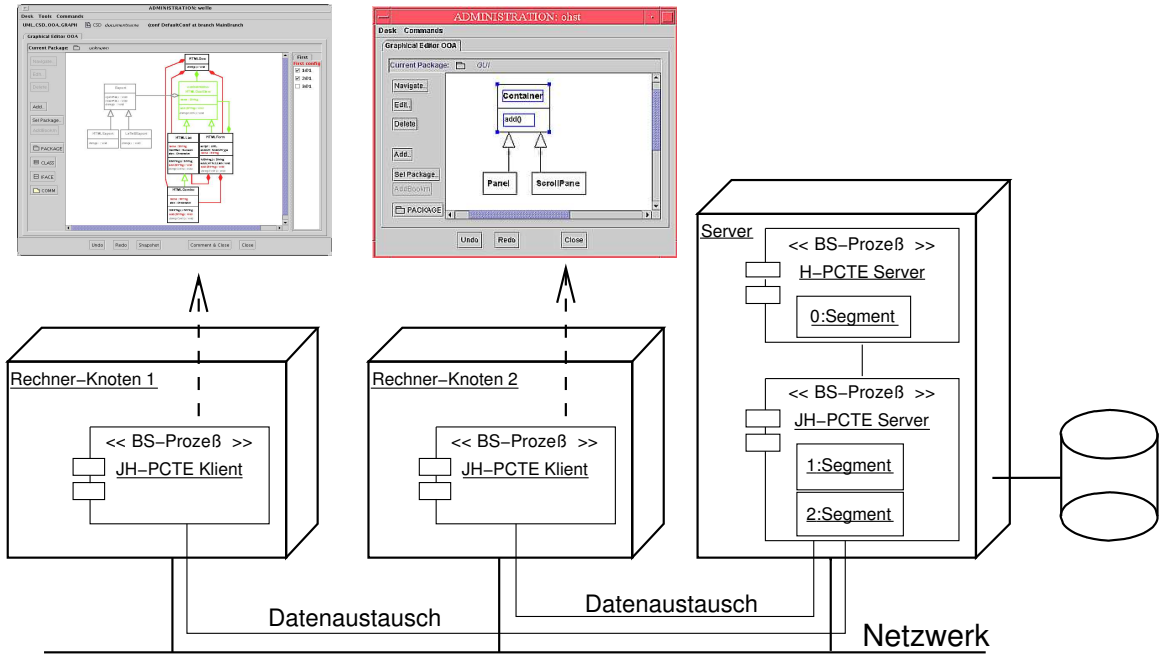


Abbildung 2.20: Verteilungsmodell von H-PCTE

2.3.4 Versionsverwaltung

Die Versionierungsfunktionalität, wie sie in PCTE definiert ist, bietet nur sehr rudimentäre Möglichkeiten, um die Daten zu versionieren. Der Hauptgrund hierfür ist, daß das Einsatzgebiet von PCTE während der Standardisierung nicht auf bestimmte Anwendungsfelder beschränkt wurde und somit die Funktionalität so allgemein wie möglich gehalten werden mußte. Für bestimmte Einsatzbereiche muß dann eine weitere Schicht die zusätzliche Funktionalität realisieren [214, 182]. Jede Annahme über den Anwendungsbereich hätte die Funktionalität für diesen Bereich verbessert, jedoch für die anderen Bereiche wahrscheinlich eingeschränkt.

Die Versionierungsfunktionalität beschränkt sich daher auf die grundlegenden Funktionen, um Versionen von Objekten anzulegen (`version_add_predecessor, version_revise`), zu löschen (`version_remove, version_remove_predecessor, version_snapshot`) oder um Eigenschaften von Versionen abzufragen (`version_is_changed, version_test_ancestry, version_test_descent`). Links werden nur im Zusammenhang mit ihrem Ausgangsobjekt versioniert.
Aufbauend auf diesen grundlegenden Funktionen muß der gewünschte Versionierungsprozeß implementiert werden. Bei feinkörnig modellierten Daten, wo eine Vielzahl an Objekt- und Linkversionen betroffen ist, muß man Aspekte wie Konsistenzsicherung, Versionserzeugung oder Versionsauswahl berücksichtigen. Diese Funktionalität muß entweder als eigener Dienst des OMS oder durch die Werkzeuge selbst realisiert werden.

2.3.5 H-PCTE-Prozesse und Transaktionen

Das Prozeß-Konzept und das Werkzeugtransaktionskonzept (WTA-Konzept) von H-PCTE unterstützen die Realisierung von SEU, indem sie viele wiederkehrende Aufgaben übernehmen und so den Implementierungsaufwand der Werkzeuge deutlich reduzieren können [167].

Prozeß-Modell. Der Zugriff auf die Objekte und Links des OMS ist ausschließlich in einem H-PCTE-Prozeß möglich. Das Prozeß-Konzept von H-PCTE unterscheidet drei Arten von Prozessen:

1. Ein (schwergewichtiger) *Betriebssystem-Prozeß* führt die gesamte SEU aus und kapselt die H-PCTE-Prozesse.

2. Eine Menge von (leichtgewichtigen) *H-PCTE-Hauptprozessen* führen je eine Gruppe von Werkzeugen eines Funktionsbereichs (z. B. Editoren und Werkzeuge für Administration, Analyse oder Entwurf) der SEU aus.

3. Eine Hierarchie von (leichtgewichtigen) *H-PCTE-Sub-Prozessen* eines H-PCTE-Hauptprozesses führen jeweils Werkzeuge oder Teile eines Werkzeugs aus (z. B. Editoren oder Analysatoren, siehe Abbildung 2.21).

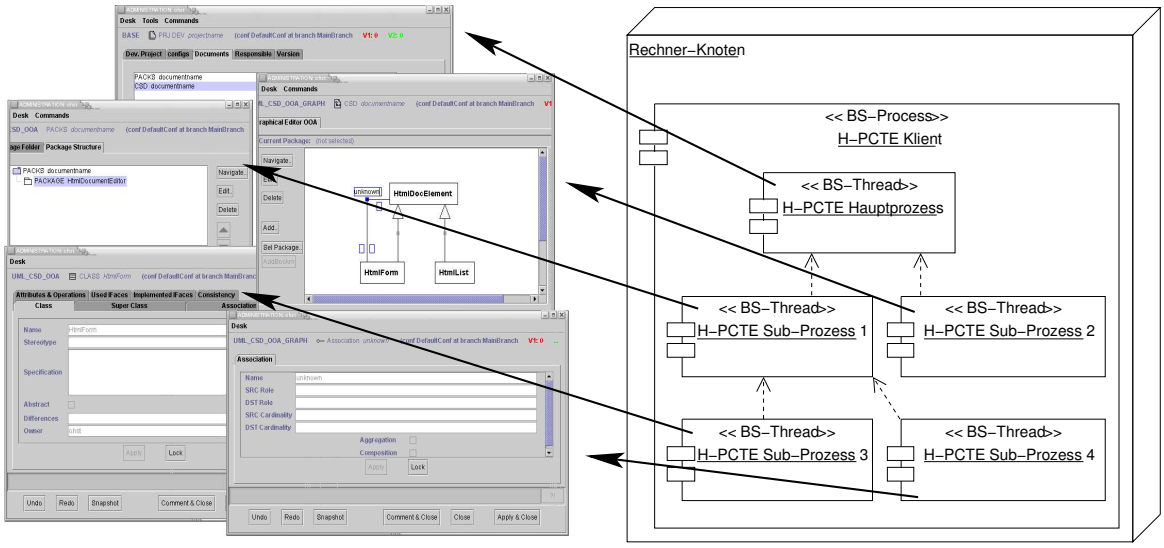


Abbildung 2.21: Prozeßmodell von H-PCTE

Zum Starten eines H-PCTE-Hauptprozesses ist eine explizite Anmeldung eines Benutzers erforderlich. Der Hauptprozeß läuft dann im Kontext dieses Benutzers ebenso wie alle gestarteten Sub-Prozesse. Jeder H-PCTE-Prozeß besitzt zusätzlich ein eigenes Arbeitsschema und eine *Default ACL*, die ein Objekt bei dessen Erzeugung erhält und die voreingestellten Zugriffsrechte bestimmt. Um konkurrierende Zugriffe auf dieselben Ressourcen durch parallele H-PCTE-Prozesse zu synchronisieren, kann jeder H-PCTE-Prozeß eine Werkzeugtransaktion starten. Die Werkzeugtransaktionen bilden dann eine geschachtelte Transaktionshierarchie.

Werkzeugtransaktionskonzept. Das Werkzeugtransaktionskonzept (WTA) von H-PCTE [187] ist die Grundlage der kooperativen oder isolierten Arbeit der Werkzeuge einer SEU. Das Konzept definiert sieben Arten von WTA mit unterschiedlichen Eigenschaften, die das Anlegen von Sperren und das Verhalten bei einem Transaktionsabbruch betreffen. Ein Transaktionsabbruch resultiert bei konventionellen Transaktionen darin, daß alle Änderungen zurückgenommen werden. Bei interaktiven Werkzeugen ist das aufgrund der langen Laufzeit nicht praktikabel, da u.U. viele Änderungen verloren gehen würden; bei nicht-interaktiven Werkzeugen hingegen ist das vollständige Rücksetzen sinnvoll. Das WTA-Konzept unterstützt beide Anwendungsfälle. Das gewünschte Verhalten wird beim Start einer WTA festgelegt. Die Transaktionen für den Einsatz in interaktiven Werkzeugen heißen *Editor-Transaktionen*, bei einem Abbruch werden nur die Änderungen bis zu dem letzten *Sicherungspunkt* zurückgenommen. Die Sicherungspunkte müssen explizit durch die Werkzeuge gesetzt werden.
In Abhängigkeit vom Werkzeugtyp und dem gewünschten Grad der Kooperation sind unterschiedliche Grade der Isolation der Transaktionen gegeneinander sinnvoll, die verschiedene Sperr-Strategien erfordern. H-PCTE kennt vier Sperr-Strategien, die auch als Editor-Transaktion vorliegen, so daß es insgesamt sieben Transaktionstypen gibt:

- `UNPROTECTED`: Es ist keine Transaktion gestartet. Jeder Prozeß läuft in diesem Modus, bis er explizit eine Transaktion startet. Es werden keine Sperren gesetzt.
- `READ_UNPROTECTED_WRITE_TRANSACTION`: Lesezugriffe laufen ohne Sperren ab, wohingegen alle Schreibzugriffe durch exklusive Sperren geschützt werden.
- `READ_UNPROTECTED_WRITE_EDITOR_TRANSACTION`: Editor-Transaktion, Sperr-Strategie wie READ_UNPROTECTED_WRITE_TRANSACTION
- `TRANSACTION`: Konventionelles Sperrprotokoll; setzen von Lese- bzw. Schreib-Sperren abhängig von der Art des Zugriffs.
- `EDITOR_TRANSACTION`: Editor-Transaktion, Sperr-Strategie wie TRANSACTION
- `WRITE_TRANSACTION`: Alle Zugriffe sind durch exklusive Sperren geschützt, unabhängig von der Art des Zugriffs.
- `WRITE_EDITOR_TRANSACTION`: Editor-Transaktion, besitzt die selbe Sperr-Strategie wie eine WRITE_TRANSACTION

Sperrmodell. Die meisten Datenbanken verwenden ein grobkörniges Sperrmodell, welches ganze Speicherseiten oder Tabellen gegen konkurrierende Zugriffe sperrt, in anderen Fällen betreffen die Sperren einzelne Datensätze oder Objekte. Diese Sperren, die konkurrierende Zugriffe auf eine große Datenmenge verhindern, sind bei einer feinkörnigen Modellierung nicht praktikabel und verhindern eine größere Parallelität [187]. H-PCTE bietet ein *feinkörniges Sperrmodell*, welches unterschiedliche Sperreinheiten kennt (siehe Tabelle 2.3).

Sperr-Name	gesperrte Einheit
O	Gesamtes Objekt
OA	Einzelne Objekt-Attribute
LTO	Menge der Linknamen eines Link-Typs von ausgehenden Links eines Objektes
KAL	Schlüsselattribute eines Links
NAL	Einzelne Nicht-Schlüsselattribute eines Links

Tabelle 2.3: Sperr-Granulate

Diese Sperrmodell hat den Vorteil, daß nur die Ressourcen gesperrt werden, die von einer Änderung direkt betroffen sind. Beispielsweise betrifft die Änderung eines einzelnen Objekt-Attributes in keinster Weise die ausgehenden Links desselben Objektes. Daher ist es möglich, beides unabhängig voneinander zu sperren und den konkurrierenden Zugriff auf dasselbe Objekt zu erlauben. Insbesondere ist das für Werkzeuge mit unterschiedlichen Sichten von Vorteil, da diese sich nicht gegenseitig blockieren, obwohl beide Werkzeuge gleichzeitig dieselben Objekte oder Links modifizieren.

Undo- und Redo-Funktionalität. Die WTA bieten einen Undo- und Redo-Mechanismus an, der über das Verhalten konventioneller Transaktionen bei Auftreten von Systemfehlern hinausgeht. Während der Laufzeit einer konventionellen Transaktion protokolliert diese alle durchgeführten Änderungen in einem Undo/Redo-Log, um bei Systemfehlern nur partiell ausgeführte Transaktionen, zurücksetzen zu können und um Transaktionen zu wiederholen, die zwar abgeschlossen, deren Ergebnisse aber noch nicht vollständig auf dem persistenten Speicher gesichert waren. Auf dieser Funktionalität setzt der Undo- und Redo-Mechanismus von H-PCTE auf, der es Werkzeugen gestattet, auf Benutzeranforderung Änderungen zurückzunehmen (*Undo*) und auch wieder herzustellen (*Redo*), sofern zwischenzeitlich keine weiteren Änderungen durchgeführt wurden.
Sicherungspunkte bestimmen die zurückzusetzenden oder wiederherzustellenden Operationen. Diese sind durch zwei Sicherungspunkte eingeschlossen. Die Sicherungspunkte müssen durch die Werkzeuge oder auf explizit Benutzeranforderung gesetzt werden. Hierfür stellt H-PCTE eine Schnittstelle zur Verfügung, ebenso wie für das Undo und das Redo.

2.3.6 Benachrichtigungsmechanismus

Das Werkzeugtransaktionskonzept ermöglicht durch die verschiedenen Sperrmodi einen hohen Grad an konkurrierender Arbeit. Verschiedene Werkzeuge können dieselben Objekt- und Linkversionen und unterschiedlichen Fenstern anzeigen. Änderungen an einem Objekt oder Link haben somit Auswirkungen auf die Anzeige mehrerer Werkzeuge, die keine Informationen über die Existenz des jeweils anderen Werkzeugs besitzen. Als Folge kann der Fall auftreten, daß die Werkzeuge veraltete Informationen anzeigen, da das andere Werkzeug Attribute verändert sowie Objekte und Links erzeugt oder gelöscht hat.
Der verteilte Benachrichtigungsmechanismus von H-PCTE adressiert diese Problematik. Die Werkzeuge können dem OMS mitteilen, daß sie über bestimmte Arten von Änderungen an bestimmten Objekten oder Links interessiert sind. Hierzu setzen die H-PCTE-Prozesse, in denen die Werkzeuge ausgeführt werden, sog. *Notifizierer*. Tritt die Änderung ein, so informiert das OMS alle interessierten H-PCTE-Prozesse über diese Änderung. Dabei ist es unerheblich, ob die Änderungen durch den H-PCTE-Prozeß selbst, einen anderen H-PCTE-Prozeß derselben

Prozeß-Hierarchie, einen Prozeß in einer anderen Prozeß-Hierarchie desselben Betriebssystem-Prozesses oder sogar durch einen H-PCTE-Prozeß auf einem anderen Rechner durchgeführt wurden. Tabelle 2.4 gibt einen Überblick der möglichen Typen von Benachrichtigungen. Die Nachrichten (*Notifizierungsnachrichten*) enthalten i.d.R. alle benötigten Informationen, um die Anzeige des korrespondierenden Werkzeugs zu inkrementell aktualisieren.

Typ der Notifizierungsnachrichten	Art der Änderung
Änderung an einem bestimmten Objekt	
SET_LOCK_ON_OBJECT_EVENT	Sperre am Objekt gesetzt
UNSET_LOCK_ON_OBJECT_EVENT	Sperre am Objekt zurückgezogen
OBJECT_DELETE_EVENT	Objekt gelöscht
OBJECT_MOVE_EVENT	Objekt zwischen Segmenten verschoben
OBJECT_APPEND_ANY_LINK_EVENT	Link angelegt
OBJECT_APPEND_VISIBLE_LINK_EVENT	angelegter Link im Arbeitsschema
OBJECT_APPEND_LINK_OF_TYPE_EVENT	Link eines best. Typs angelegt
OBJECT_DELETE_ANY_LINK_EVENT	Link gelöscht
OBJECT_DELETE_VISIBLE_LINK_EVENT	gelöschter Link im Arbeitsschema
OBJECT_DELETE_LINK_OF_TYPE_EVENT	Link eines best. Typs gelöscht
OBJECT_CONVERT_EVENT	Typänderung des Objekts
OBJECT_MODIFY_ACL_EVENT	ACL des Objekts geändert
OBJECT_MODIFY_EVENT	Objektattribute geändert
Änderung an einem bestimmten Link	
LINK_DELETE_EVENT	Link gelöscht
SET_LOCK_ON_LINK_EVENT	Sperre am Link gesetzt
UNSET_LOCK_ON_LINK_EVENT	Sperre am Link zurückgezogen
LINK_MODIFY_EVENT	Linkattribute geändert
Erzeugung eines bestimmten Objekt- oder Link-Typs	
OBJECT_OF_TYPE_CREATE_EVENT	Objekt eines best. Typs angelegt
LINK_OF_TYPE_CREATE_EVENT	Link eines best. Typs angelegt

Tabelle 2.4: Benachrichtigungstypen

2.3.7 PI-SET

PI-SET ist ein Prototyp einer SEU, dem eine OMS-orientierte Werkzeugarchitektur zugrunde liegt. PI-SET basiert auf dem Framework *genform* [167], welches die Dienste von H-PCTE verwendet. Das Ziel von PI-SET und genform ist es, einen Werkzeugkonstruktionsansatz zu validieren, der es ermöglicht, existierende Werkzeuge an spezielle Bedürfnisse in einem Projekt anzupassen und weitere zusätzlich benötigte Werkzeuge ohne großen Aufwand zu bauen. Diese Flexibilität wird durch einen komponentenbasierten Konstruktionsansatz erreicht. Hierin können neue Werkzeuge aus einer existierenden Menge an Komponenten zusammengestellt und durch Parameter konfiguriert werden.

Jedes Werkzeug (z. B. graphische Diagramm-Editoren oder textuelle Editoren für einzelne Diagrammelemente) startet PI-SET in einem eigenem H-PCTE-Prozeß, der Werkzeugtransaktionen vom Typ READ_UNPROTECTED_WRITE_EDITOR_TRANSACTION startet. Dadurch wird eine hohe Parallelität erreicht. Die Transaktions-Struktur ist daher identisch mit der Prozeß-Struktur, wie sie in Abbildung 2.21 auf Seite 85 dargestellt ist. Diese Struktur bedingt, daß mehrere

Werkzeuge konkurrierend auf dieselben Objekt- und Linkversionen zugreifen müssen (vgl. Anforderungen an kooperative Arbeit in Abschnitt 1.3.3). Beispielsweise wird in einem graphischen Klassendiagramm-Editor an einem Diagramm gearbeitet. Eine darin enthaltene Klasse soll um deren Spezifikation erweitert werden. Daher startet der Entwickler einen tabellarischen Editor, um die Eigenschaften der Klasse zu bearbeiten. In diesem Beispiel müssen beide Editoren auf der selben Objektversion arbeiten.

Kapitel 3

Das Versionsverwaltungskonzept für Software-Dokumente

In diesem Kapitel beschreiben wir das in dieser Arbeit vorgeschlagene Versionsverwaltungskonzept für feinkörnig modellierte Dokumente aus den frühen Phasen der Softwareentwicklung, insbesondere für UML-Diagramme. Wir unterstellen hierbei, daß die Dokumente in H-PCTE gespeichert und als komplexe Objekte modelliert sind. Das Konzept ist jedoch nicht auf H-PCTE beschränkt, sondern kann auch in anderen OMS oder Softwareentwicklungsumgebungen eingesetzt werden, sofern diese auf einem feinkörnigen Editier-Metamodell basieren.

Der Aufbau dieses Kapitels ist wie folgt. Zu Beginn geben wir in Abschnitt 3.1 eine Übersicht des Versionierungskonzepts einschließlich Hinweisen, wie dieses in Werkzeuge integriert werden kann. Die Übersicht wird in den folgenden Abschnitten vertieft. Die Grundlage sind Entwurfstransaktionen, die einzelne Aufgaben repräsentieren und in deren Kontext die Dokumente verwaltet und versioniert werden. Diese stellen wir in Abschnitt 3.2 vor. Die Bearbeitung und auch die Versionierung der Dokumente erfolgt in den Werkzeugtransaktionen, die um entsprechende Funktionen erweitert werden. Aufgrund der feinkörnigen Modellierung der Dokumente können die Dokumente nicht manuell versioniert werden; es ist eine automatische Versionierung zu bevorzugen (Abschnitt 3.3). Konsistente Versionen werden durch Konfigurationen zusammengefaßt (Abschnitt 3.3.1), die auch als Grundlage für die kooperative Entwicklung dienen (Abschnitt 3.3.2). In Abschnitt 3.3.3 beschreiben wir abschließend die Versionierung der Objekte und Links.

3.1 Übersicht des Versionierungskonzepts

Viele Werkzeuge bieten keine oder nur rudimentäre Unterstützung bei der Versionierung der mit ihrer Hilfe erstellten Dokumente. Diese Unterstützung besteht oft nur darin, von der persistenten Darstellung, also den Dateien auf der Festplatte, eine neue Version in einem externen Versionsmanagement-System (VM-System) anzulegen. Das ist ein Bruch zwischen den Denkwelten, einerseits die Denkwelt der Dokumente, bei Quellcode besteht diese z. B. aus Klassen, Methoden oder Funktionen, andererseits die Denkwelt der VM-Systeme, die oft aus Dateien und Verzeichnissen besteht [147]. Entwickler denken jedoch nicht in Dateien, in Versionen oder in Klassen sondern eher in den durchzuführenden Aufgaben [77], die sie dann auf die unterliegende Struktur abbilden. Oft muß das Anlegen und der Zugriff auf Versionen der Dokumente mit Hilfe externer Werkzeuge durchgeführt werden. Einzelne Versionen kann man oft nur über nichts sagende Versionsnummern bestimmen, wobei die Versionsnummern zusammengehörender Dokumentversionen oft nicht identisch sind. Ein VM-System sollte das berücksichtigen.

Teilweise ist es auch notwendig und sinnvoll, Zwischenversionen anzulegen. Für diese Tätigkeiten muß der Entwickler die Werkzeuge wechseln und auf den Abschluß des Check-In-Vorgangs warten. Das stellt eine Unterbrechung der eigentlichen Entwicklungstätigkeit dar und unterbricht den Entwickler in seiner Konzentration. Christensen [45] nennt das *Disruptive Delays* und *Change of Mental Focus*. Um diese Unterbrechungen zu minimieren, sollten Versionen automatisch angelegt werden [157]. Das erleichtert auch die Sicherstellung der Konsistenz von abhängigen Dokumenten.

Wenn man diese Kritikpunkte an existierenden VM-Systemen berücksichtigt, folgt daraus, daß das VM-System besser in die Werkzeuge integriert sein und daß die Versionierung nicht auf nichts sagenden Versionsnummern aufbauen sollte, sondern auf den zugrundeliegenden Änderungsaufgaben, die durchzuführen sind. Die einzelnen Änderungsaufgaben können einzelnen Entwicklern oder Entwicklergruppen zugeordnet sein und ein Projekt, eine Erweiterung eines bestehenden Systems oder nur die Behebung eines einzelnen Fehlers umfassen.

Aus diesen Überlegungen folgt, daß die Versionierung sich an den Aufgaben orientieren sollte. Das bedingt auch die Integration des VM-Systems in die Werkzeuge, die zur Verbesserung der kooperativen Entwicklung mehrbenutzerfähig sein sollten. Die Entwickler müssen sich daher vor Beginn der Tätigkeit an der Softwareentwicklungsumgebung (SEU) anmelden und bekommen eine Liste von Aufgaben präsentiert, die dem einzelnem Entwickler oder der Entwicklergruppe zugeordnet sind, s. Abbildung 3.1. Die Liste kann bei Bedarf von der Entwicklergruppe noch weiter verfeinert und einzelne Aufgaben hiervon den Gruppenmitgliedern zugeordnet werden. Die teilweise langen Bearbeitungszeiten der Aufgaben erfordern oft mehrere Werkzeugsitzungen, um die Aufgabe vollständig zu bearbeiten. Daher verbietet es sich, die Aufgaben an eine Werkzeugsitzung zu binden. Sinnvoll erscheint der Einsatz von Entwurfstransaktionen (ETA) (siehe Abschnitt 2.1.3.2), die jeweils eine Aufgabe repräsentieren und die erforderlichen Dokumente verwalten. Liegt eine Hierarchie von Aufgaben vor, kann diese in einer ETA-Hierarchie abgebildet werden.

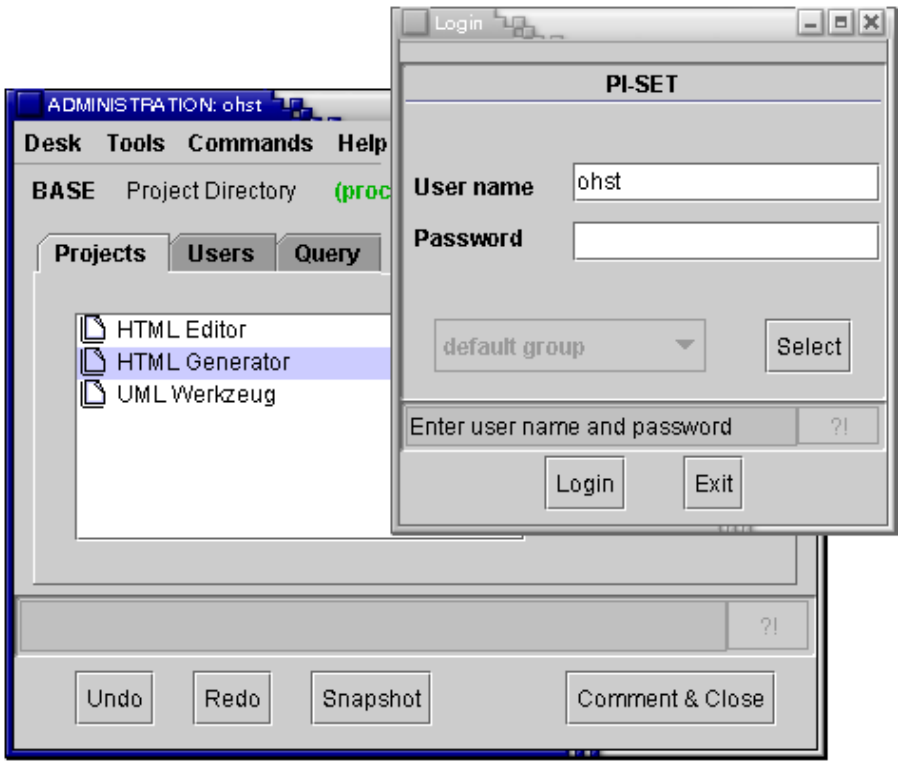


Abbildung 3.1: Beispiel eines Werkzeugs mit Anmeldefenster und Projektliste

Die eigentliche Bearbeitung wird in Werkzeugsitzungen durchgeführt, die im Rahmen einer ETA ausgeführt werden. Die Aufgabe der ETA ist es hierbei, die Dokumente in der benötigten Version bereitzustellen, sie ist jedoch nicht für das Recovery oder den Zugriffsschutz zuständig. Alle Änderungen einer Werkzeugsitzung, die im Anlegen von neuen Objekt- und Linkversionen resultiert, faßt eine Konfiguration zusammen, die der entsprechenden ETA zugeordnet ist und automatisch durch das VM-System angelegt wird. Die Konfigurationen spiegeln die Entwicklungsgeschichte aller Dokumente einer ETA wider. Aufeinanderfolgende Werkzeugsitzungen legen auch aufeinanderfolgende Konfigurationen an. Das bietet einige Vorteile. Die Entwickler müssen sich nicht mehr um das Anlegen neuer Versionen kümmern und können sich somit vollständig auf die eigentliche Aufgabe konzentrieren. Durch das Sichern der Ergebnisse jeder einzelnen Werkzeugsitzung werden Zwischenversionen angelegt, auf die ein Entwickler bei Bedarf wieder zurücksetzen kann. Der Entwickler muß in diesem Fall eine Konfiguration auswählen, auf die er aufsetzen will. In diesem Fall wird eine parallele Konfiguration angelegt. Abbildung 3.2 zeigt ein Beispiel für ein Werkzeug zur Auswahl einer Konfiguration.

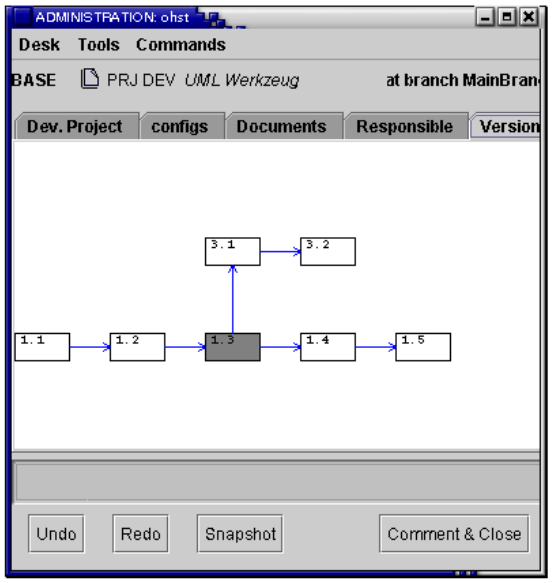


Abbildung 3.2: Beispielanzeige eines Konfigurationsgraphen im Werkzeug

Eine Werkzeugsitzung meint hier nicht ausschließlich die Verwendung eines einzelnen Werkzeuges, sondern kann auch die Benutzung mehrerer Werkzeuge einer SE-Umgebungsausführung umfassen. Die Änderungen aller Werkzeuge einer einzelnen SE-Umgebungsausführung werden dann in einer Konfiguration gespeichert.

Ein Nachteil vieler VM-Systeme ist, daß Änderungskommentare immer nur zu bestimmten Zeitpunkten eingegeben werden können [45], entweder beim Anlegen einer Änderungsaufgabe *vor* der eigentlichen Änderung, was dem Entwickler eine starre Struktur vorgibt, oder beim Check-In *nach* dem Editieren der Dokumente. Dann hat er speziell nach längeren Änderungssitzungen die konkreten Änderungen wieder vergessen und der Kommentar wird ungenau, was den Nutzen reduziert. Daher ist es sinnvoll, die Eingabe von Änderungskommentaren zu einem beliebigen Zeitpunkt zu ermöglichen [45].
Die Werkzeuge besitzen daher ein Eingabefeld für Kommentare, so daß die Entwickler parallel zu den eigentlichen Änderungen die Kommentare eingeben können. Ein Kommentar wird dann jeweils der entsprechenden Konfiguration zugeordnet. Zum besseren Überblick über die bereits in der ETA durchgeführten Änderungen zeigt das Werkzeug eine Liste mit allen Änderungskommentaren der vorhergehenden Werkzeugsitzungen an.

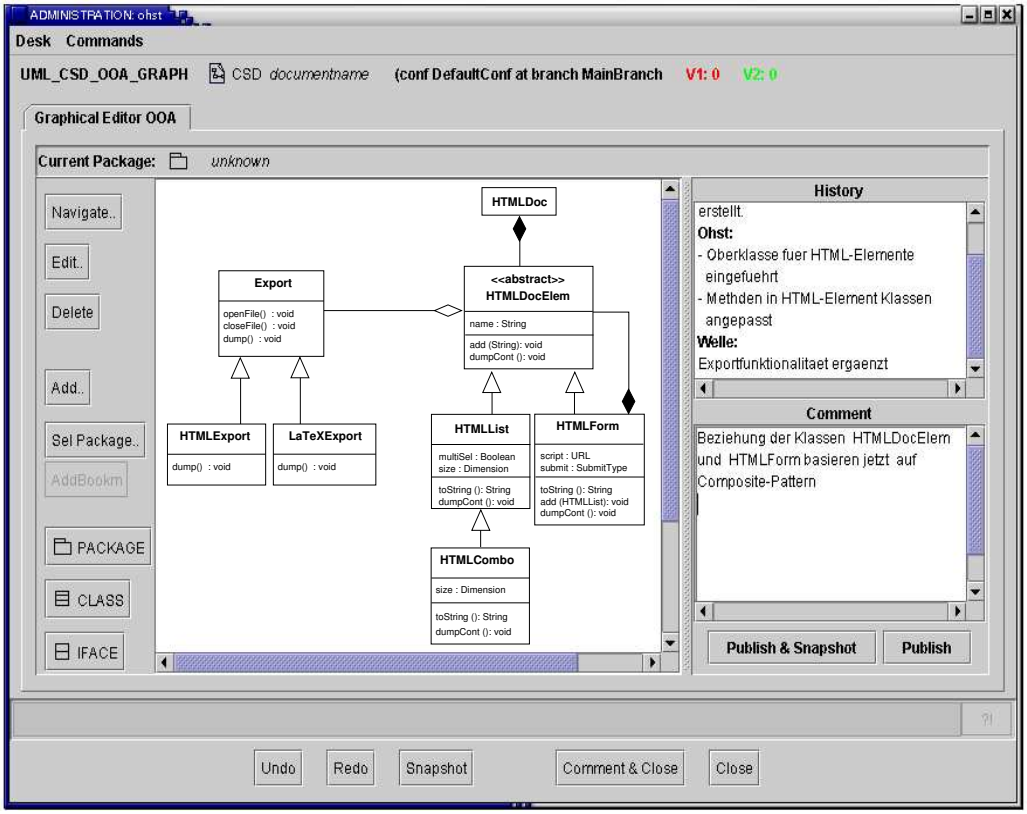


Abbildung 3.3: Beispiel eines Klassendiagramm-Editors mit der Möglichkeit direkt Änderungskommentare einzugeben

Die kooperative Arbeit an UML-Dokumenten ist im Gegensatz zu Quelltexten möglich, da hier der Zeitpunkt, in denen die Dokumente bestimmten Konsistenzkriterien entsprechen müssen, i.d.R. der Abschluß der Aufgabe ist. Im Laufe der Entwicklung sind inkonsistente Zwischenzustände erlaubt, was bei Quelltexten nicht der Fall ist. Diese werden öfters compiliert, um die durchgeführten Änderungen zu testen. Zu diesen Zeitpunkten muß der Quelltext konsistent sein, da andernfalls die Compilierung mit einem Fehler abbrechen würde.

Die kooperative Entwicklung bietet einige Vorteile. Sie hilft, die Anzahl der angelegten Varianten zu reduzieren, und ermöglicht bei einem feinkörnigen Sperrmodell, wie es z. B. H-PCTE anbietet, die Zeiten, in denen Entwickler auf die Freigabe von gesperrten (Teil-)Dokumenten warten, zu reduzieren. Jedoch bietet die Kooperation nicht nur während der Bearbeitung einer Aufgabe Vorteile, sondern auch beim Mischen der Ergebnisse, die im Rahmen von verschiedenen Aufgaben durch unterschiedliche Entwicklergruppen entstanden sind. Hier können alle beteiligten Entwickler gemeinsam die Änderungen zusammenführen und bei Auftreten eines Konfliktes diesen lösen.

Die Basis der Kooperation stellen die Konfigurationen dar. Meldet sich ein Entwickler bei der SEU an und wählt eine Aufgabe, die bereits durch einen anderen Entwickler bearbeitet wird, können die Entwickler kooperativ arbeiten. Die Änderungen beider Entwickler dann werden in der Konfiguration zusammengefaßt.

Die Koordination der Entwickler ist in H-PCTE durch den Einsatz des Benachrichtigungsmechanismusses möglich, indem alle beteiligten Entwickler über die Änderungen der jeweils anderen informiert werden. Die Benachrichtigungen erfolgen auf zwei Arten. Einerseits werden die durchgeführten Änderungen direkt in den Werkzeugen aller kooperierenden Entwickler sichtbar. Andererseits kann der Benachrichtigungsmechanismus auch auf die Änderungskommenta-

re angewendet werden. Alle Kommentare, die die Entwickler zu einer Konfiguration machen, werden in der Liste der Änderungskommentare angezeigt. Beides steigert zusätzlich das Team-Bewußtsein, da die einzelnen Entwickler nicht isoliert voneinander arbeiten und immer über den Zustand der Arbeiten ihrer Kollegen informiert sind. Bei Bedarf können sie sich dann mit den Kollegen abstimmen, um Inkonsistenzen zu vermeiden.

Nach Abschluß einer (Teil-)Aufgabe, also einer Sub-ETA, müssen alle Änderungen an die übergeordnete ETA übertragen werden. Das entspricht einem Check-In, wie es von den meisten VM-Systemen bekannt ist. Anschließend kann die ETA beendet und nicht mehr benötigte Konfigurationen können gelöscht werden, sofern das gewünscht wird.

3.2 Das Entwurfstransaktionskonzept

Die Entwurfstransaktionen (ETA) dienen, wie im vorigen Abschnitt bereits erwähnt, der hierarchischen Strukturierung von Projekten anhand von Aufgaben und der Bereitstellung der zur Bearbeitung der Aufgaben benötigten Dokumentversionen. Das bietet den Vorteil, daß die Entwicklung eines Projektes und der dazugehörigen Dokumente anhand der Aufgabenstruktur dokumentiert werden kann. Weiterhin haben die vergebenen Versionsnummern nur eine untergeordnete Bedeutung und die Anzahl der für das Projekt verwalteten Versionen wird reduziert, da ausschließlich die Endergebnisse der einzelnen ETA an die übergeordnete ETA weitergereicht werden.
Die Struktur der ETA ist nicht von einem Prozeßmodell oder Vorgehensmodell abhängig. Daher können die ETA manuell durch die Entwickler mit Hilfe eines Werkzeugs oder automatisch z. B. durch eine Prozeßmaschine angelegt werden. Die Struktur der ETA ist somit auch an verschiedene Projekte anpaßbar.
Das einzige, was vorgegeben ist, ist der „Lebenszyklus“ einer ETA, siehe Abbildung 3.4. Zu Beginn eines Projektes muß man die Wurzel-ETA, die das gesamte Projekt repräsentiert, anlegen und initialisieren. Anschließend ist sie vorbereitet, um in ihrem Kontext Dokumente anzulegen oder aber das Projekt weiter zu strukturieren, indem man Sub-ETA anlegt und initialisiert. In den Sub-ETA können auch Dokumente angelegt oder aber von einer übergeordneten ETA (Super-ETA) importiert werden. Vor Beendigung einer Sub-ETA muß diese mit der Super-ETA synchronisiert worden sein. Hierbei mischt man die, in der Sub-ETA erstellten oder weiterentwickelten Dokumente, mit den (älteren) Versionen in der Super-ETA. Vor Beendigung der Super-ETA müssen alle ihre Sub-ETA beendet worden sein. Die einzelnen Schritte wollen wir im folgenden detaillierter betrachten.

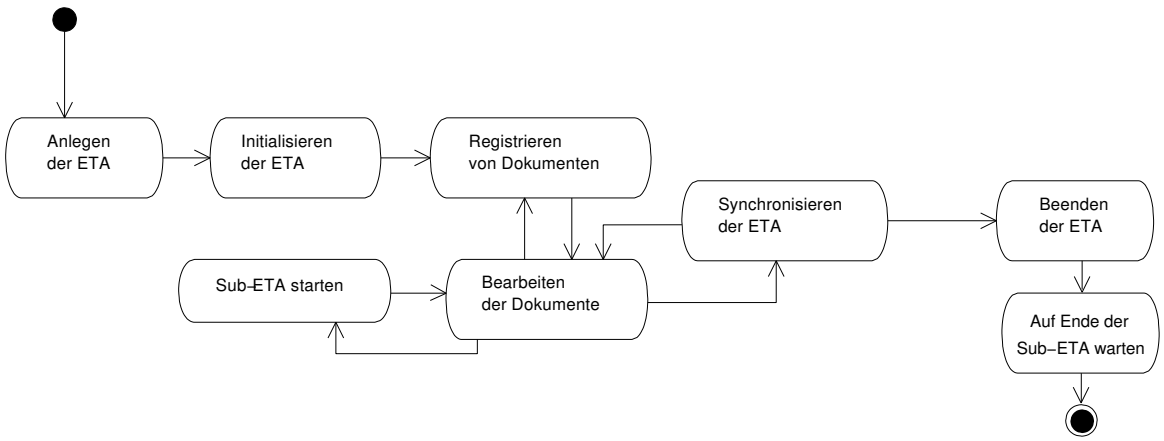


Abbildung 3.4: Lebenszyklus einer ETA

3.2.1 Anlegen und Initialisierung einer Entwurfstransaktion

In einer Instanz von H-PCTE können Dokumente für verschiedene Projekte gespeichert werden. Für jedes im OMS gespeicherte Projekt existiert eine eigene ETA-Struktur, die explizit anzulegen ist. Ausschließlich Werkzeuge, die im Rahmen einer ETA ausgeführt werden, können die Versionierungsfunktionalität des OMS nutzen und auf die versionierten Dokumente zugreifen. Vor Beginn der eigentlichen Bearbeitung einer Aufgabe muß man daher die korrespondierende ETA einmalig angelegen.
Aufgrund der langen Dauer einer ETA und der in H-PCTE üblichen Methode, Meta-Daten selbstreferentiell zu speichern, bietet sich das auch für die ETA an. Diese werden daher als eigenständige Objekte vom Typ `design_transaction` (siehe Anhang A und Abbildung 3.5) im OMS angelegt. Das Anlegen der ETA-Hierarchie ist unterteilt in das Erzeugen der ETA-Objekte und in die Initialisierung der ETA. Zum Anlegen der ETA-Objekte kann man die reguläre API (`Pcte_object_create`) nutzen.

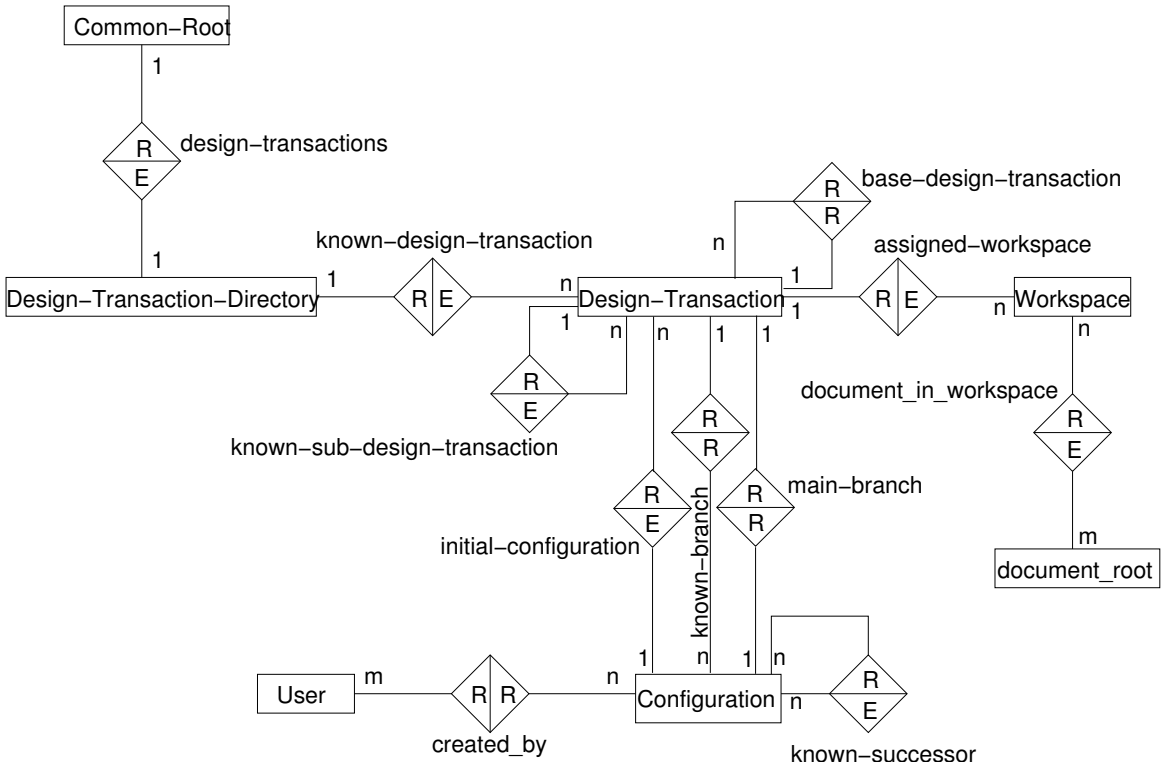


Abbildung 3.5: Schema für die selbstreferentielle Verwaltung der ETA und Konfigurationen

Bei der Initialisierung einer ETA (`HPcte_dta_initialize`) wird das ETA-Objekt in die Struktur der bereits initialisierten ETA-Objekte eingebunden. Hierbei sind zwei Fälle zu unterscheiden. Wenn die ETA ein Projekt darstellt und somit die Wurzel einer neuen ETA-Hierarchie ist, muß diese in die OMS-weite Verwaltung der ETA eingehängt werden. Handelt es sich bei der ETA um eine Teilaufgabe einer existierenden ETA, so wird diese als Sub-ETA der entsprechenden ETA eingetragen. Desweiteren wird im OMS ein Arbeitsbereich für die ETA angelegt, der alle in dieser ETA zu bearbeitenden Dokumente verwaltet, s. Abschnitt 3.2.2. Der Arbeitsbereich ist in H-PCTE als eigener Objekt-Typ realisiert und steht mit dem ETA-Objekt über einen Link in Beziehung.

Durch die explizite Speicherung der ETA als Objekte bietet sich auch die Möglichkeit an, diese als Anknüpfungspunkte für andere Funktionsbereiche eines Software-Konfigurationsmanagement-Systems (SKM-System) zu nutzen, z. B. für das Änderungsmanage-

ment. Unterstützt wird das noch durch die Trennung der Erzeugung von ETA-Objekten und der Initialisierung der ETA. So ist es möglich, Subtypen vom ETA-Objekttyp abzuleiten und diese zur Datenspeicherung für andere Funktionsbereiche zu verwenden. Dieser Aspekt wird jedoch im Rahmen dieser Arbeit nicht weiter verfolgt, da es sich um einen eigenen Forschungsbereich handelt.

3.2.2 Verwalten der Dokumente

Man kann in einer ETA nur auf die versionierten Dokumente zugreifen, die im Arbeitsbereich der ETA abgelegt sind. Dieser enthält nach der Initialisierung zunächst keine Dokumente. Die Dokumente müssen entweder von einer anderen ETA importiert oder neu angelegt werden.

Das navigierende Datenmodell, die feinkörnige Modellierung und die Versionierung der Dokumente (siehe Abschnitt 3.3) legen nahe, daß die Dokument nicht vollständig in einen Arbeitsbereich kopiert werden. Beim Import (`HPcte_dta_register_document`) wird nur ein Link auf das Wurzel-Objekt des zu importierenden Dokuments angelegt bzw. beim Anlegen eines neuen Dokuments (`HPcte_dta_create_document`) wird nur dessen Wurzel-Objekt vom Arbeitsbereichsobjekt ausgehend erzeugt (vgl. Abbildung 3.5). Wenn ein Dokument in der ETA nicht mehr benötigt wird, kann es gelöscht werden (`HPcte_dta_delete_document`). Hierbei wird jedoch nicht das gesamte Dokument mit allen Versionen gelöscht, sondern es wird nur in dieser ETA als gelöscht markiert.
Zum Import von Dokumenten aus anderen ETA müssen die gewünschten Dokumente und deren Versionen, die eine Konfiguration beschreibt, bestimmt werden. Eine einfache Lösung wäre, den Import von Versionen entlang der ETA-Hierarchie zu realisieren. Das hätte jedoch den Nachteil, daß die Dokumentweitergabe zwischen Geschwister-ETA nicht möglich wäre. Die Dokumente müßten immer zuvor in der gemeinsamen Super-ETA eingemischt werden, bevor diese importierbar sind. Um diese Problematik zu umgehen, gibt man nach der Initialisierung der ETA die *indexEntwurfstransaktion!Basis-Basis-ETA* an. Diese bestimmt die ETA, aus der die Dokumente zu importieren sind. Als Basis-ETA kann man die Super-ETA oder eine Geschwister-ETA auswählen. Zusätzlich muß man noch die Dokumentversionen anhand einer Konfiguration der Basis-ETA bestimmen (`HPcte_dta_set_initial_configuration`), die dann die *Initial-Konfiguration* der neuen ETA ist (siehe Abschnitt 3.3.1).

In einer ETA werden i.d.R. nicht alle Dokumente der Basis-ETA benötigt. Daher ist es notwendig, eine Liste aller Dokumente der Basis-ETA erhalten zu können, aus der dann die zu importierenden Dokumente gewählt werden können. Diese kann man mit der Schnittstelle zum Lesen der ausgehenden Links eines Objektes, in diesem Fall des Arbeitsbereichsobjektes, erhalten. Ergänzend, zur Arbeitserleichterung der Werkzeugentwickler, gibt es eine Schnittstelle (`Pcte_dta_list_documents`), die diese Aufgabe übernimmt. Als Eingabe erhält die Funktion die ETA und als Ausgabe liefert sie eine Liste aller Dokumente, die diese ETA verwaltet. Ergänzend kann man angeben, ob man die Dokumente der ETA oder die Dokumente der ETA einschließlich der Dokumente der Basis-ETA erhalten möchte.

Diese Realisierung eines Arbeitsbereichs, der nur das Wurzelobjekt der in ihm verwalteten Dokumente referenziert, hat den Nachteil, daß man vom Arbeitsbereichsobjekt keinen direkten Zugriff auf alle in dieser ETA neu angelegten *Objekt*-Versionen[1] der Dokumente hat. Man könnte eine spezielle Schnittstelle anbieten, sofern ein Zugriff auf diese Objektversionen notwendig

[1] Die Versionierung der Dokumente einer ETA betrachten wir im nächsten Abschnitt, im Detail. Hier sei nur gesagt, daß durch die feinkörnige Modellierung der Dokumente, diese aus einer Vielzahl an Objekten bestehen, die einzeln versioniert werden.

sein sollte[2]. Durch die Konzentration auf die Dokumente und nicht auf die Objekte besteht im Rahmen dieses Konzeptes keine Notwendigkeit hierfür.

3.2.3 Bearbeitung und Beenden einer Entwurfstransktion

Die Bearbeitung der von einer ETA verwalteten Dokumente findet in den Werkzeugen der SEU statt, die in Datenbank-Prozessen ausgeführt werden. Vor Beginn der Bearbeitung müssen sich die Datenbank-Prozesse an der ETA anmelden, um Zugriff auf die Dokumente der ETA zu erhalten. Zur Anmeldung wird in H-PCTE die Funktion `HPcte_process_set_dta` verwendet. Beim Anmeldevorgang kann man angeben, ob die Werkzeuge einen exklusiven Zugriff auf alle Dokumente der ETA benötigen. Im Laufe der normalen Entwicklungstätigkeit an den Dokumenten ist i.d.R. kein exklusiver Zugriff erforderlich. Die Dokumente werden durch die Transaktionssperren der Werkzeugtransaktionen, die in den einzelnen Werkzeugen gestartet werden, gegen konkurrierende Zugriffe geschützt und bei Bedarf wird eine neue Version angelegt, s. Abschnitt 3.3.
Der exklusive Zugriff auf die Dokumente einer ETA ist sinnvoll, um weitere Änderungen an diesen Dokumenten zu verhindern. Einsatzbereiche sind z. B. Konsistenztests oder die Synchronisierung von zwei ETA. Hierbei werden alle Änderungen an den Dokumenten von einer Sub-ETA an deren übergeordnete ETA übertragen. Das Synchronisieren von ETA diskutieren wir in Abschnitt 3.2.4.
Der exklusive Zugriff auf eine ETA kann entweder explizit aufgehoben werden, indem sich der Datenbank-Prozeß bei der ETA abmeldet (`HPcte_process_unset_dta`), oder implizit beim Beenden des Prozesses.

Um eine ETA zu beenden, muß diese zuvor synchronisiert worden sein, andernfalls bricht die Funktion `HPcte_dta_commit` mit einer Fehlermeldung ab. Ob eine ETA synchronisiert wurde, läßt sich anhand eines Objekt-Attributes des ETA-Objektes überprüfen. Bei der Synchronisierung setzt die entsprechende Operation das Attribut `state` auf den Wert `SYNCHRONIZED`. Meldet sich nach der Synchronisierung ein weiterer Prozeß an der ETA an, so wird das Attribut wieder auf `RUNNING` gesetzt. Nur wenn das Attribut den Wert `SYNCHRONIZED` besitzt, läßt sich die ETA beenden. Nachdem die ETA beendet wurde, ist keine weitere Bearbeitung der durch sie verwalteten Dokumente in ihrem Kontext mehr möglich.
Nach dem Beenden einer ETA ist der lesende Zugriff auf alle Versionen der Dokumente, die im Rahmen der ETA entstanden sind, weiterhin möglich. Die meisten Versionen stellen jedoch Zwischenergebnisse dar, die in den meisten Fällen automatisch und in einigen Fällen durch die Entwickler explizit angelegt wurden. Diese Zwischenversionen sind i.d.R. nach Abschluß der ETA nicht mehr von Interesse. Dann sind vor allem die Endversionen der Dokumente relevant. Daher ist es möglich, diese nicht mehr benötigten Zwischenversionen beim Beenden der ETA zu löschen. Das kann man bei der Commit-Funktion für der ETA angeben, die dann alle Dokumentversionen (Konfigurationen) löscht, die nicht mit übergeordneten ETA synchronisiert wurden.

3.2.4 Synchronisieren von Entwurfstransaktionen

Beim Synchronisieren (`HPcte_dta_synchronize`) von zwei ETA werden die von den ETA verwalteten Dokumente verglichen und Differenzen gemischt (siehe Kapitel 5). Beteiligt sind i.d.R. eine ETA und deren übergeordnete ETA. Das beruht auf der Idee, daß eine Sub-ETA für die Bearbeitung einer Teilaufgabe zuständig ist und nach deren Abschluß die Ergebnisse in die

[2]Die techn. Realisierung kann auf Designation-Links basieren, ähnlich, wie ein Segment in H-PCTE alle auf ihm abgelegten Objekte referenziert.

übergeordnete Aufgabe einfließen. Die beteiligten ETA sind jedoch vom unterliegenden Softwareentwicklungsprozeß abhängig und nicht statisch vorgegeben. Die ETA kann man durch die ETA bestimmen, an der die Synchronisierungsfunktion aufrufende H-PCTE-Prozeß angemeldet ist und welche ETA als Ziel-ETA für die Synchronisierung angegeben wurde.
Beim Synchronisieren werden alle am Arbeitsbereichsobjekt verwalteten Dokumente in die Dokumente der Ziel-ETA eingemischt[3], so daß dort eine neue Konfiguration angelegt wird, die der Mischversion entspricht; aufgetretene Konflikte sind manuell durch die Werkzeuganwender zu lösen (siehe Abschnitt 5.4).

Die Synchronisierung auf Basis der Dokumente ist aus mehreren Gründen der Synchronisierung auf Basis der versionierten Objekte vorzuziehen:

- *Abstraktionsebene*: Das primäre Ziel der Synchronisierung ist der Abgleich der Dokumente, an denen die Werkzeuganwender interessiert ist. Die Objekte sind hierbei nur ein Hilfsmittel.

- *fehlende Semantik*: Ein Beispiel hierfür sind unterschiedliche Vorgehensweisen bei der Modellierung von Dokumenten. Die Linkkeys können semantisch bedeutsam sein oder aber auch nicht. Diese Eigenschaft wird ausschließlich von dem Werkzeugentwickler bestimmt, das unterliegende VM-System kann daher keine Annahmen darüber treffen. Daraus folgt direkt, daß beim Mischen die Modellierung der Dokumente berücksichtigt werden muß, also eine andere Art der Modellierung auch Auswirkungen auf die Synchronisierung hat.

- *fehlender Kontext*: Objekte treten in H-PCTE nicht für sich alleine auf, sie stehen immer über Links mit anderen Objekten in Beziehung; neue Objekte können nur im Zusammenhang mit Links bestimmter Kategorien angelegt werden. Man muß also auch immer das Ausgangsobjekt und den Link mitberücksichtigen. Diese lassen sich beim Synchronisieren auf Basis der Objekte nur mit höherem Aufwand bestimmen. Verstärkt wird diese Problematik noch durch *gemeinsame Komponenten*, also Teile eines komplexen Objektes, die auch Teil eines weiteren komplexen Objektes sind.

- *größerer Aufwand*: Man müßte alle Objekte bestimmen, die in der zu synchronisierenden ETA verändert wurden. Hierfür böten sich zwei Methoden an, entweder man betrachtet jedes Objekt in H-PCTE oder man speichert Referenzen auf die modifizierten Objekte. Beide Vorgehensweisen sind mit einem erhöhtem Aufwand verbunden: Entweder durch eine längere Laufzeit für die Suche nach den Objekten oder durch einen erhöhten Speicherbedarf zur Verwaltung der Objekte.

Die starke Abhängigkeit der Synchronisierung von der Art, wie die Dokumente modelliert sind, stellt jedoch einen Nachteil dar. Insbesondere bei der Multiple-View-Integration muß die Sicht passend gewählt werden, da andernfalls die Dokumente nicht vollständig sichtbar sind und das Ergebnis unvollständig oder sogar falsch sein kann. Eine generische Lösung unabhängig von der Modellierung und somit unabhängig von den Werkzeugen ist somit nicht möglich.
Durch die Multiple-View-Integration stellt sich auch die Frage nach der Reihenfolge, in der die Dokumente synchronisiert werden müssen. In diesem Fall handelt es sich bei den Dokumenten nicht unbedingt um vollständig disjunkte komplexe Objekte. Die komplexen Objekte können überlappen oder die Dokumente können sogar nur andere Sichten auf ein großes komplexes Objekt sein. Diese Eigenschaft ist auch wieder abhängig von der Modellierung, so daß es keine

[3]Hier ist auch die Lösung denkbar, daß die Dokumente in der Sub-ETA gemischt und anschließend in die Ziel-ETA übertragen werden. Das hätte jedoch die Nachteile, daß in beiden ETA die gemischten Dokumente redundant vorliegen würden und daß in der Ziel-ETA weitere Änderungen durchgeführt worden sein könnten.

allgemeingültige Lösung hierfür geben kann. Um diese Aufgabe zu lösen, kann man der Funktion `HPcte_dta_synchronize` eine Liste mit den Wurzel-Objekten der Dokumente übergeben, die diese dann synchronisiert.

Standardmäßig synchronisiert die Funktion nur die Version der Dokumente, die durch die jüngste Konfiguration der Hauptentwicklungszweige beider ETA spezifiziert ist. Wenn weitere Versionen der Dokumente als neue Varianten in der Ziel-ETA angelegt werden sollen, so muß dies explizit durch den Werkzeuganwender unter Angabe der Konfiguration, die die Dokumentversion bestimmt, veranlaßt werden.

Um sicherstellen zu können, daß während des Synchronisierens keine weiteren Änderungen an der ETA durchgeführt werden, kann der H-PCTE-Prozeß bei der Anmeldung den exklusiven Zugriff auf die zu synchronisierende ETA wählen. Der exklusive Zugriff auf die Ziel-ETA ist nicht notwendig. Hier ist es ausreichend sicherzustellen, daß während der Synchronisierung keine konkurrierenden Änderungen an dem Konfigurationszweig stattfinden (siehe Abschnitt 3.3.2). Nach Abschluß der Synchronisierung wird die ETA als synchronisiert markiert.

3.3 Die Versionierung der Dokumente

Die Entwurfstransaktionen dienen der Verwaltung der Dokumente; bearbeitet werden sie in den Werkzeugtransaktionen (WTA) von H-PCTE, siehe Abbildung 3.6. Das bietet den Vorteil, daß Dienste wie Recovery und Concurrency-Control nutzbar sind. Im Rahmen des Versionierungskonzepts wurden die WTA jedoch um die Versionierungsfunktionalität erweitert, bieten ansonsten den vollen Funktionsumfang, einschließlich der unterschiedlichen Transaktionstypen, die in Abschnitt 2.3.5 diskutiert wurden.

Während der Bearbeitung eines Dokumentes greifen die Werkzeuge lesend und schreibend auf die Objekte, Links und Attribute zu, die ein Dokument modellieren. Die explizite Versionsverwaltung ist bei feinkörnig modellierten Dokumenten nicht praktikabel, weder durch die Werkzeuganwender, noch durch die Werkzeuge. Erstere müßten das Metamodell der Dokumente berücksichtigen, das sie nicht kennen und für ihre Arbeit auch nicht relevant ist. Sollten die Werkzeuge die Versionierung übernehmen, müßte das von den Werkzeugentwicklern berücksichtigt werden. Da bereits allein der Werkzeugbau eine komplexe Aufgabe ist, sollte diese nicht noch durch Fragen der Versionierung erschwert werden. Insbesondere durch die feinkörnige Modellierung muß eine große Anzahl an Objekten, Links und Versionen hiervon verwaltet werden, wobei auch deren Konsistenz sichergestellt werden muß. Allein eine Klasse mit fünf Attributen und zehn Methoden besteht aus mehr als 15 Objekten. Bei zehn Klassen können es schon mehr als 150 Objekte sein, die zusätzlich in verschiedenen Versionen vorliegen können. Aus diesen Überlegungen folgt, daß die Versionierung als ein weiterer Dienst des OMS zu realisieren ist und daß konsistente Versionen zusammen verwaltet werden sollten.

Setzt man voraus, daß die Dokumente ausschließlich in WTA bearbeitet werden, so kann die Versionierungsfunktionalität in diese integriert werden. Das Ändern eines Objektes oder Links innerhalb einer WTA impliziert die Erzeugung einer neuen Version.

Berücksichtigt man weiterhin die Eigenschaft von Transaktionen, daß nach Abschluß einer Transaktion sich die Datenbank in einem konsistenten Zustand befindet, sofern sie auf einem konsistenten Zustand gestartet wurde, so können die WTA auch für die konsistente Verwaltung von Versionen genutzt werden. Die Grundlage hierfür sind Konfigurationen, die wir im folgenden vorstellen.

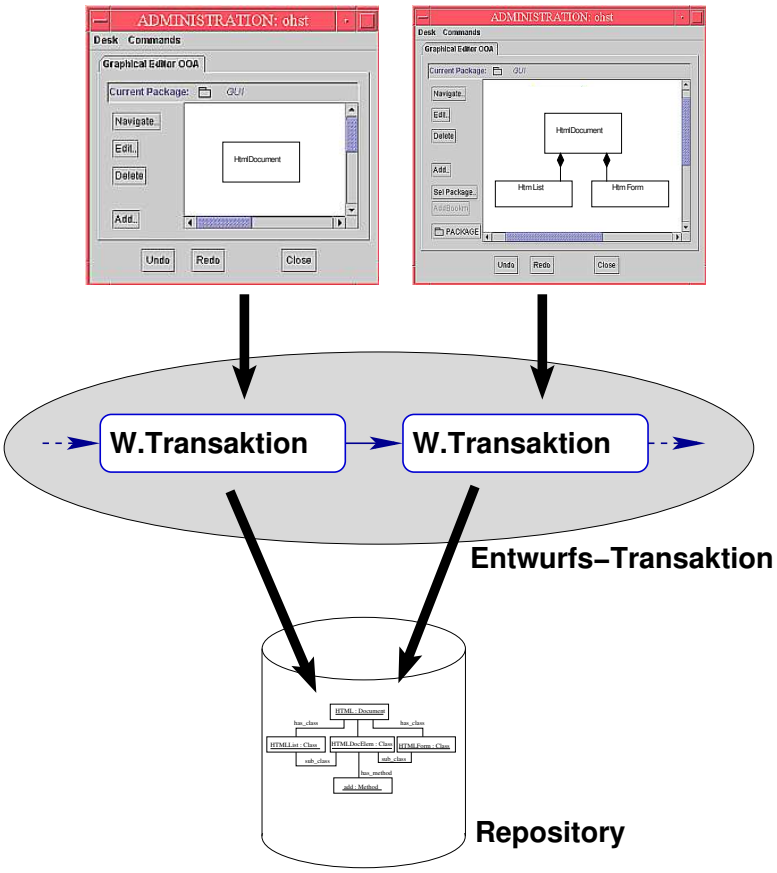


Abbildung 3.6: Bearbeitung der Dokumente durch Werkzeugtransaktionen im Rahmen einer Entwurfstransaktion

3.3.1 Konfigurationsverwaltung durch Werkzeugtransaktionen

Konfigurationen dienen der konsistenten Verwaltung von Objekt- und Linkversionen und stellen einzelne Teilschritte zur Lösung der durch die ETA definierten Aufgabe dar. Alle Objekt- und Linkversionen, die eine WTA anlegt, faßt eine Konfiguration zusammen. Die Konfigurationen kann man daher mit den in Abschnitt 2.1.1.1 vorgestellten Konzept der Change-Packages vergleichen. Auch wenn diese Versionen von Dateien zu einer Konfiguration zusammenfassen und nicht einzelne Objekt- oder Linkversionen, ist das grundlegende Prinzip vergleichbar. Im Unterschied zu den Change-Packages entsprechen die Konfigurationen nicht einer logischen Änderungen, sondern Teilschritten zur Lösung einer Aufgabe.

3.3.1.1 Anlegen von Konfigurationen

Beim Start einer neuen WTA (`HPcte_tta_start`) legt diese auch eine neue Konfiguration – die *Arbeitskonfiguration* – an, die beim Beenden (`HPcte_tta_end`) eingefroren wird. Eine neu gestartete WTA muß eine eingefrorene Konfiguration als *Basiskonfiguration* angeben, die die in der WTA zugreifbaren Versionen festlegt (siehe Abschnitt 3.3.1.2). Die Arbeitskonfiguration steht über eine Nachfolgerbeziehung mit der Basiskonfiguration in Beziehung. Wird eine Konfiguration als Basiskonfiguration gewählt, die bereits eine nachfolgende Konfiguration besitzt, so wird ein neuer *Konfigurationszweig* eröffnet.

Der Konfigurationszweig, der die Hauptentwicklungslinie darstellt, ist der *Hauptentwicklungszweig*. Sollte sich während der Arbeit an den Dokumenten herausstellen, daß z. B. ein Seitenzweig eine elegantere Lösung der Aufgabe bietet oder daß einer alternativen Lösung

der Vorzug gegeben wird, so kann man diesen Seitenzweig zum Hauptentwicklungszweig (`HPcte_dta_set_main_branch`) definieren.

Unter Berücksichtigung dieser Informationen können wir den Anmeldevorgang eines H-PCTE-Prozesses an eine ETA konkretisieren. Bei der Anmeldung an eine ETA erhält der Prozeß den Hauptentwicklungszweig zugewiesen, sofern bei der Anmeldefunktion `HPcte_process_set_dta` nichts anderes angegeben wird. Dadurch vereinfacht sich auch das Starten einer WTA, da man nicht explizit eine Basiskonfiguration angeben muß. Erhält die Funktion `HPcte_tta_start` keine weiteren Angaben zu einer speziellen Konfiguration, so verwendet sie automatisch die jüngste Konfiguration des bei der Anmeldung gewählten Konfigurationszweiges als Basiskonfiguration.
Wenn durch den Start einer WTA ein neuer Konfigurationszweig angelegt wird, so wechselt der H-PCTE-Prozeß in den neuen Konfigurationszweig. Der Prozeß verbleibt auch nach dem Beenden der WTA in diesem Zweig, so daß WTA, die im selben Prozeß zu einem späteren Zeitpunkt gestartet werden, ebenfalls auf diesem Zweig aufsetzen. Der Wechsel des Zweiges ist nur durch ein Abmelden und erneutes Anmelden des Prozesses bei der ETA möglich. Das sollte jedoch kein Problem darstellen, da i.d.R. mit dem Beenden des Werkzeugs durch die SEU auch der korrespondierende H-PCTE-Prozeß beendet wird. Sollte die SEU die H-PCTE-Prozesse nicht beenden, sondern für neu gestartete Werkzeuge wiederverwenden, um die Startzeit zu verkürzen, so müssen die Laufzeitdaten der Prozesse, wie z. B. Objekt-Referenzen oder auch die Anmeldung an eine ETA, neu initialisiert werden.

3.3.1.2 Zugriff auf Versionen

Im Unterschied zu VM-Systemen, die das gesamte Dokument versionieren oder die nach dem Prinzip der Versionspropagierung arbeiten, bildet ein Objekt in H-PCTE *keine* Konfiguration für seine Komponenten. Ein Link referenziert daher nicht eine einzelne Version eines Objekts, sondern alle möglichen Versionen, die jedoch nicht alle gleichzeitig zugreifbar sind. Aufgrund der feinkörnigen Modellierung und der daraus folgenden großen Anzahl an Objekten ist die manuelle Versionsauswahl für jedes Objekt nicht praktikabel. Stattdessen legt die Basiskonfiguration einer WTA die zugreifbaren Versionen fest. Die Idee ist vergleichbar mit den Änderungsschichten in PIE (vgl. Abschnitt 2.1.1.1 auf Seite 39).
Die Basiskonfiguration einschließlich ihrer Vorgängerkonfigurationen enthalten alle Versionen – genauer gesagt alle Revisionen eines Zweiges –, von allen in der WTA zugreifbaren Objekten und Links. Das resultiert daher, daß die jeweilige Arbeitskonfiguration einer WTA die Nachfolgerkonfiguration der Basiskonfiguration ist und alle erzeugten Versionen zusammenfaßt.
Da ein Objekt oder Link mehrfach in unterschiedlichen WTA verändert worden sein kann, gibt es von diesen auch mehrere Versionen in der Folge der Konfigurationen. In diesem Fall ist nur die jüngste Version, also die Version, die zuletzt angelegt wurde, für die WTA „sichtbar“. Die Auswahl einer Version richtet sich demzufolge ausschließlich nach der gewählten Basiskonfiguration. Das bietet den Vorteil, daß ein Werkzeug nur einmal eine Konfiguration bestimmen muß und anschließend alle Zugriffe automatisch auf die „richtige“ Version angewendet werden. Hat eine WTA z. B. eine Konfiguration als Basiskonfiguration (K_B) festgelegt, die bereits eine Nachfolgerkonfiguration (K_n) besitzt, so liefert das Auslesen aller ausgehenden Links eines Objektes nur die Links, die in K_B und deren Vorgängern angelegt wurden, nicht jedoch die Links in K_n und deren Nachfolgern. Dasselbe gilt auch für gelöschte Objekte oder Links. Wurde ein Link in K_B gelöscht, so ist er nicht mehr sichtbar. Wäre jedoch die Vorgänger-Konfiguration von K_B als Basiskonfiguration gewählt worden, so hätte die Löschung in K_B keine Auswirkung auf dessen Sichtbarkeit. Das OMS liefert demzufolge nur die Versionen, die durch die Basiskonfiguration definiert sind. Der Werkzeugentwickler braucht die Versionsauswahl nicht zu berücksichtigen.

Komplexe Objekte. Besondere Beachtung benötigen die komplexen Objekte in H-PCTE und die Operationen darauf. Diese beeinflussen nicht einzelne Eigenschaften eines Objektes, sondern ein oder mehrere atomare Objekte als Ganzes. Da diese Operationen bisher nur auf unversionierten Objekten definiert sind, muß deren Semantik durch die Existenz der Versionen erweitert werden. Betroffen sind hiervon die Schnittstellen, die komplexe Objekte löschen (`Pcte_object_delete`), kopieren (`Pcte_object_copy`) oder zwischen Segmenten verschieben (`Pcte_object_move`).
Die Operation `Pcte_object_delete` darf sich nur auf die aktuelle Version des Objektes auswirken, da der Werkzeuganwender nur diese im Werkzeug sieht und löschen will. In diesem Fall bleibt die Semantik der Operation nahezu unverändert. Alle Tests hinsichtlich von gemeinsamen Komponenten oder Links, die das Löschen verhindern, bleiben erhalten. Im Unterschied zu der unversioniert arbeitenden Operation werden die Objekte jedoch nicht gelöscht, sondern es wird eine neue Version der Objekte angelegt, die eine Löschmarkierung enthält.
Ein atomares oder komplexes Objekt einschließlich aller Versionen zu löschen ist nicht erlaubt, da die Versionen der Objekte eines Projektes bzw. einer Entwurfstransaktion zueinander konsistent sein müssen. Das Löschen eines einzelnen Objektes mit seinen Versionen würde durch das Löschen von Linkversionen die eingefrorenen Versionen von anderen Objekten verändern, was dem Sinn der Versionierung widerspricht. Ein weiteres Problem wäre, daß für alle Versionen der zu löschenden atomaren Objekte eines komplexen Objektes überprüft werden müßte, ob die Objekte gelöscht werden dürfen, da noch Links auf einzelne Versionen existieren können, die das Löschen verhindern. Sofern nur eine Objektversion einen Link besitzt, der das Löschen verhindert, darf das gesamte komplexe Objekt einschließlich aller Versionen nicht gelöscht werden. Durch die Multiple-View-Integration besteht jedoch eine hohe Wahrscheinlichkeit, daß Links dieser Art existieren. Aus diesen Überlegungen folgt, daß nur das Löschen des gesamten Projektes sinnvoll ist. Hierfür ist eine spezielle Operation notwendig, die eine ETA inklusive aller Sub-ETA und allen Dokumenten löscht. Dabei ist zu beachten, daß die zu löschenden Dokumente in keiner übergeordneten ETA mehr enthalten sein dürfen.

Beim Kopieren eines Objektes in einer WTA greift diese nur auf die gültige Version zu, so daß auch nur diese Version kopiert wird. Das folgt auch aus der Überlegung, daß ein Objekt nur auf Anforderung eines Werkzeuganwenders kopiert wird. Er ist i.d.R. nur an der Version interessiert, die er gerade im Werkzeug sieht. Aufgrund der Implementierung des Zugriffs auf die Versionen der Objekte und Links (siehe Abschnitt 4.1.3) sind keine Änderungen an der Operation notwendig.

Im Gegensatz hierzu kann es wünschenswert sein, daß die Operation `Pcte_object_move` das gesamte Objekt einschließlich aller Versionen auf ein anderes Segment verschiebt. Benötigt wird diese Operation, wenn ein Projekt abgeschlossen ist und somit die korrespondierende ETA mit allen Dokumenten zur Archivierung auf ein anderes Segment verschoben werden kann. Die Operation ist dann den administrativen Operationen für ETA zuzuordnen.
Das Verschieben der aktuellen Version eines Objektes ist beispielsweise zur Verteilung von Dokumenten sinnvoll, um diese dezentral zu bearbeiten. Das spätere Zusammenführen von Versionen eines Objektes kann mit dieser Operation nicht realisiert werden, da die Objekte modifiziert wurden. Das Zusammenführen entspricht dem Mischen von Versionen, so daß das auf das Mischen von Dokumenten abgestützt werden muß.

Die Funktionen zum Löschen oder Verschieben von ETA mit allen Dokumenten oder einzelnen Versionen sind der Projektverwaltung zuzuordnen, die in dieser Arbeit nicht das zentrale Thema ist. Daher werden diese Fälle nicht weiter berücksichtigt.

Einfluß auf die Dokumente und deren Verwaltung. Üblicherweise sind alle von einer ETA verwalteten Dokumente als komplexe Objekte modelliert. Die automatische Versionierung

der atomaren Objekte und die Zusammenfassung deren Versionen durch eine Konfiguration führen zu einem gemeinsamen Versionsbaum aller Dokumente der ETA, der unabhängig von den komplexen Objekten ist. Das bietet zwei Vorteile. Sofern die Dokumente unabhängig modelliert sind, wird durch die gemeinsame Versionierung sichergestellt, daß alle Dokumentversionen zueinander konsistent sind[4]. Die manuelle Auswahl von konsistenten Objektversionen entfällt. Bei der Multiple-View-Integration hingegen sind die Dokumente nur unterschiedliche Sichten auf ein zusammenhängendes Modell. Ein atomares Objekt kann in diesem Fall in verschiedenen Sichten enthalten sein. Die in den Sichten enthaltenen Attribute müssen jedoch nicht identisch sein, d.h. das Objekt stellt sich in den einzelnen Sichten anders dar. Die unabhängige Versionierung der Dokumente wäre in diesem Fall nur durch eine sichtenabhängige Versionierung möglich, da die Dokumente ausschließlich durch die Sichten definiert sind. Das Problem hierbei ist, daß die Sichten nicht fest vorgegeben sind, sondern dynamisch zur Laufzeit der Werkzeuge neu zusammengestellt werden können. Für jede erstmals gewählte Sicht müßten dann die zueinander konsistenten Versionen der atomaren Objekte definiert werden, was manuell aufgrund der feinkörnigen Modellierung eine sehr aufwendige und fehleranfällige Aufgabe wäre.

Die automatische Versionierung der Objekte und die Zuordnung von Objektversionen zu Konfigurationen hat auch Einfluß auf die Dokumentverwaltung der ETA. Ob ein Dokument (oder besser die Dokument-Wurzel) nur in einzelnen Zweigen einer ETA enthalten ist oder in allen Konfigurationen, hängt davon ab, ob die Operationen zum Import (`HPcte_dta_register_document`) bzw. zum Anlegen (`HPcte_dta_create_document`) eines Dokumentes in einer WTA oder außerhalb ausgeführt wurden. Beide Möglichkeiten haben ihre Vor- und Nachteile, die im Einzelfall vom Werkzeugentwickler abgewogen werden müssen.
Werden die Dokumente außerhalb einer WTA angelegt, so sind sie in allen Konfigurationen zugreifbar. Der Werkzeuganwender muß sich keine Gedanken darüber machen, in welcher Konfiguration ein Dokument angelegt wurde, ein Anwendungsfall ist z. B. der Import in eine weitere ETA. Die Existenz der Dokument-Wurzel ist jedoch nicht mit der Existenz der aktuellen Version des Dokumentes gleichzusetzen. Diese erhält man nur durch Auswahl der entsprechenden Konfiguration, da die das Dokument modellierenden Objekte durch die Konfiguration bestimmt sind. Legt man die Dokumente in einer WTA an, so daß sie nur in der Basiskonfiguration der WTA und deren Nachfolger-Konfigurationen sichtbar sind, kann man beim Import nur auf die Dokumente, die in der gewählten Konfiguration enthalten sind, zugreifen. Die Auswahl einer Konfiguration ist in beiden Fällen notwendig.

3.3.1.3 Selbstreferentielle Verwaltung der Konfigurationen

Die Konfigurationen sind, wie die ETA, durch Objekte im OMS repräsentiert und stehen über Links miteinander in Beziehung. Ein ETA-Objekt referenziert jeweils die jüngsten Konfigurationen aus ihren Zweigen, siehe Abbildung 3.7. Der Vorteil dieser selbstreferentiellen Speicherung besteht darin, daß die Werkzeuge auf die Metadaten der Versionsverwaltung in der Weise zugreifen können, wie sie auch für die zu verwaltenden Dokumente verwendet wird. Ein Beispiel hierfür ist das Auslesen des Konfigurationsgraphen (siehe Abbildung 3.2), der in den Werkzeugen dargestellt wird, damit der Werkzeuganwender eine Dokumentversion auswählen kann, die er bearbeiten möchte.

Zur Unterscheidung einzelner Konfigurationen und der Konfigurationszweige gibt es *Konfigurationsidentifizierer* (K-ID), die an den Konfigurationsobjekten gesetzt werden. Der K-ID besteht aus zwei numerischen Angaben. Die erste Zahl spezifiziert den Konfigurationszweig (Zweig-ID) und die zweite Zahl die Konfiguration innerhalb eines Zweiges. Diese beginnt in

[4]Damit wird jedoch nicht sichergestellt, daß die Dokumente semantisch konsistent sind. Das kann nur manuell und ggf. mit Hilfe von Analysatoren sichergestellt werden.

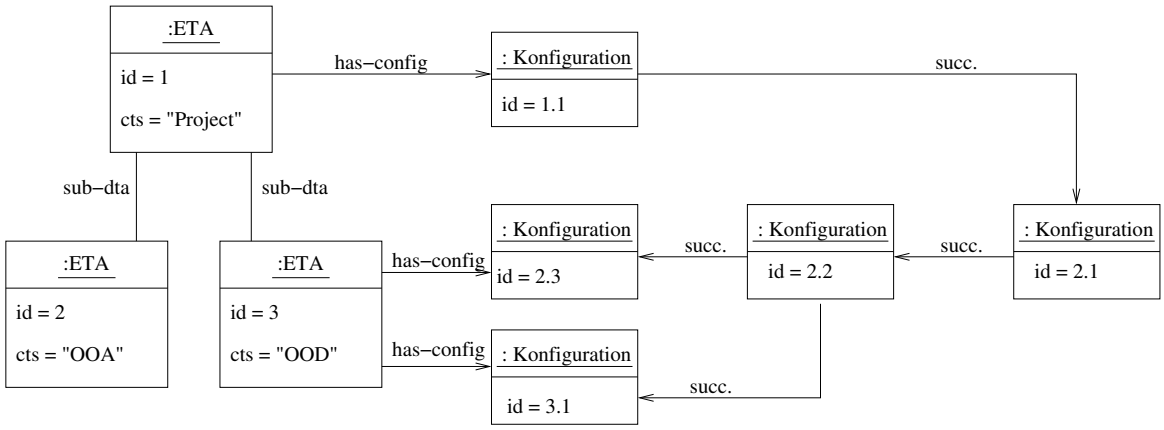


Abbildung 3.7: Zusammenhang von Entwurfstransaktionen und Konfigurationen

jedem Zweig bei eins und wird fortlaufend inkrementiert. Die Zweig-ID ist im gesamten OMS eindeutig. Sie wird ebenfalls fortlaufend mit dem Anlegen eines neuen Konfigurationszweigs erhöht. Die Zweig-ID ermöglicht keine Rückschlüsse auf den Zusammenhang von zwei Zweigen. Die Struktur der Zweige läßt sich ausschließlich anhand der Konfigurationsobjekte bestimmen. Das stellt jedoch keine Einschränkung dar, da die Konfigurationen lediglich Zwischenschritte zur Lösung einer Aufgabe darstellen, die der korrespondierenden ETA zugeordnet ist. Varianten eines Produktes sind daher als parallele ETA modelliert und nicht als Konfigurationszweige. Konfigurationszweige sollten nur angelegt werden, damit einzelne Entwickler isoliert an einer Teilaufgabe arbeiten können, deren Ergebnisse anschließend wieder in der Hauptentwicklungszweig eingemischt werden.

Zusätzlich zum K-ID speichert ein Konfigurationsobjekt weitere Informationen, zu nennen sind hier der Zustand der Konfiguration, also ob es noch eine aktive WTA gibt oder ob die Konfiguration eingefroren wurde. Durch die Mehrbenutzerfähigkeit von H-PCTE und daher auch von dem Versionierungskonzept ist es notwendig, daß an der Konfiguration auch vermerkt wird, welcher Entwickler die Konfiguration anlegte. Da die Benutzer auch selbstreferentiell in H-PCTE gespeichert sind und sich diese zum Starten eines Prozesses anmelden müssen, kann eine WTA exakt einem Benutzer zugeordnet werden. Diese Zuordnung wird durch einen Link zwischen dem Konfigurationsobjekt und dem Benutzerobjekt ausgedrückt.

Desweiteren können die Änderungskommentare am Konfigurationsobjekt gespeichert werden. In Abschnitt 4.2 diskutieren wir die verschiedenen Ansätze zu deren Speicherung.

3.3.1.4 Konfigurationen als Sicherungspunkte

Während der Entwicklung durchlaufen die Dokumente verschiedene Phasen, in denen sie mehr oder weniger konsistent sind. Auch während der Arbeit in einer WTA kann ein Dokument einen Zustand annehmen, den der Entwickler als teilweise konsistentes Zwischenergebnis ansieht und ihn deshalb als eigene Konfiguration sichern möchte. Für diesen Fall gibt es die Möglichkeit, explizit eine Konfiguration anzulegen (`HPcte_tta_create_savepoint`), die einem Sicherungspunkt entspricht. Die alte Konfiguration wird dadurch eingefroren und alle weiteren Änderungen dann der neuen Konfiguration zugeordnet.

Sollte nach dem Anlegen einer Sicherungspunkt-Konfiguration die WTA abgebrochen werden, entweder durch einen Systemfehler oder explizit durch den Benutzer (`HPcte_tta_abort`) werden nur die Änderungen der letzten aktiven Konfiguration zurückgenommen. Alle Änderungen bis zum Anlegen des Sicherungspunktes bleiben erhalten. Wurde in einer WTA kein Sicherungs-

punkt angelegt, so werden bei einem Abbruch der WTA alle Änderungen bis zur Basiskonfiguration zurückgenommen.

3.3.1.5 Read-Only-Werkzeugtransaktionen

Einige Anwendungsfälle erfordern einen reinen Lesezugriff; Änderungen sind nicht vorgesehen. Ein Beispiel hierfür sind Analysatoren, die nur Abfragen, z. B. mit P-OQL [100], auf den Dokumenten durchführen. Ein weiteres Beispiel sind Werkzeuge, die ausschließlich zur Anzeige der Dokumente dienen und keine Änderungen an den Dokumenten erlauben.
Für diese Anwendungsfälle würde der Einsatz der beschriebenen WTA aufgrund der selbstreferentiellen Verwaltung der Konfigurationen, einen zu großen Aufwand bedeuten. WTA, die einen reinen Lesezugriff (ro-WTA, `HPcte_read_only_tta_start`) erlauben, bieten hier einige Vorteile. Aufgrund ihrer „Nur-Lesezugriff"-Eigenschaft verhindern sie Änderungen an den Dokumenten. Das erleichtert die Werkzeugkonstruktion dahingehend, daß nicht zwei verschiedene Werkzeuge, eins zum Ändern und eins zur Anzeige[5] der Dokumente, realisiert werden müssen. Die Konstruktion eines Werkzeugs, welches in Abhängigkeit von der gewünschten Verhaltensweise eine WTA oder eine ro-WTA startet, ist ausreichend. Die Werkzeuge sollten in jedem Fall prüfen, ob sie Schreibrechte auf den Objekten, Links und Attributen, die die Dokumente modellieren, besitzen. Wenn das Werkzeug keine Schreibrechte besitzt, werden die Änderungsfunktionen des Werkzeugs gesperrt und somit unerlaubte Änderungen verhindert, andernfalls würden die entsprechenden Änderungsfunktionen von H-PCTE (z. B. `Pcte_object_set_attribute`) mit einem Fehler abgebrochen.
Ein Werkzeug kann aus unterschiedlichen Gründen kein Schreibrechte besitzen:

- Das Werkzeug hat für diesen speziellen Objekt-, Link- oder Attributtyp kein Schreibrecht. Das bezeichnet man als Typrechte, die im SDS angegeben werden.
- Der Werkzeuganwender hat an der speziellen Instanz keine Schreibrechte. Das wird durch die ACL des Objekts bestimmt.
- Die Transaktion hat keine Schreibrechte, da die notwendige Schreibsperre aufgrund in Konflikt stehender Sperren nicht zugeteilt werden kann.

Durch die Einführung der ro-WTA können die Schreibrechte auch daher fehlen, daß sich das Werkzeug in einer ro-WTA befindet. Diesen Sachverhalt kann man als einen Spezialfall, der nichtzuteilbaren Schreibsperre interpretieren, so daß die Werkzeuge diesen Fall nicht explizit testen müssen. Die entsprechenden Funktionen zur Zuteilbarkeit einer Sperre (`HPcte_test_object_lock, HPcte_test_link_lock`) liefern in diesem Fall diese Information.

Durch die Beschränkung auf Lesezugriffe kann auch der Zugriff auf die Dokumente durch eine vereinfachte Sperranforderung beschleunigt werden. Hier sind zwei Fälle zu unterscheiden. Die ro-WTA nutzt eine eingefrorene Konfiguration als Basiskonfiguration. In diesem Fall sind keine Sperren notwendig, da Änderungen nicht auftreten können, da alle möglichen Zugriffe reine Lesezugriffe sind, die per Definition kompatibel sind. Ist die Basiskonfiguration hingegen aktiv, müssen nur bei der Sperr-Strategie `TRANSACTION` (siehe Abschnitt 2.3.5) Lesesperren gesetzt werden. Die Sperr-Strategie `WRITE_*_TRANSACTION` ist generell nicht möglich, da diese das Setzen von Schreibsperren fordert, was im Widerspruch zur Semantik der ro-WTA steht. Bei der Auswahl einer `READ_UNPROTECTED_*` Sperr-Strategie sind keine Sperren notwendig, da Lesezugriffe generell erlaubt und alle Schreibzugriffe verboten sind.

[5] Das verhindert, daß die Dokumente versehentlich modifiziert werden.

Würde man nur das Starten einer WTA beschleunigen wollen, könnte man auf die ro-WTA verzichten und das Konfigurationsobjekt beim ersten Schreibzugriff anlegen. Das hätte jedoch den Nachteil, daß die Konfigurationsobjekte nicht mehr als Grundlage für die kooperative Arbeit genutzt werden können, wie sie im nächsten Abschnitt beschrieben wird. Ein weiterer Grund gegen diese mögliche Realisierung ist, daß die Semantik der Funktionen von H-PCTE, die Daten verändern, davon abhängig wäre, welche anderen Funktionen zuvor aufgerufen wurden.

3.3.2 Kooperation in einer Konfiguration

Die synchrone kooperative Arbeit an Software-Dokumenten wie z. B. UML-Diagrammen bietet einige Vorteile, wie in den Abschnitten 1.2.5 und 2.1.4 diskutiert. Die technischen Beschränkungen, die diese Form der kooperativen Arbeit bei dateibasierten VM-Systemen nahezu verhindern, existieren im vorgestellten Versionierungskonzept (siehe auch Abschnitt 3.3.3) nicht, da die Dokumente einerseits feinkörnig modelliert, andererseits nicht in Dateien, sondern in einem OMS gespeichert sind. Das ermöglicht einen konkurrierenden Zugriff auf dieselbe Dokumentversion. Konventionelle VM-Systeme bieten häufig nur die isolierte Arbeit in Arbeitsbereichen oder sperren eine Version gegen parallele Arbeit.

Die Version von H-PCTE *ohne* die Versionierungsfunktionalität bietet Mechanismen (die feinkörnige Sperrverwaltung basierend auf den Werkzeugtransaktionen und den Benachrichtigungsmechanismus), die es erlauben, daß mehrere Entwickler konkurrierend auf dieselben Dokumente zugreifen. Aufgrund der Erweiterung um die Versionierungsfunktionalität müssen diese Dienste so modifiziert werden, daß sie auf unterschiedlichen Versionen arbeiten können.
Die Kombination der Versionierungsfunktionalität, des Benachrichtigungsmechanismus und der Sperrverwaltung erlauben konkurrierende Zugriffe auf dieselben Dokumentversionen, ohne daß Parallelitätsanomalien auftreten. Die konkurrierenden Zugriffe sind hierbei durch die Sperren gegeneinander geschützt[6] und die Werkzeuge werden über konkurrierende Änderungen benachrichtigt, so daß alle Werkzeuge den aktuellen Zustand der Dokumente anzeigen, einschließlich der Änderungen aller kooperierenden Entwickler.
Zu berücksichtigen ist hier, daß die Sperren auf den Versionen und nicht mehr auf der Instanz angefordert werden. Vergleichbares gilt für den Benachrichtigungsmechanismus. Die sich aus der Einführung der Versionierungsfunktionalität ergebenden Änderungen stellen wir in Abschnitt 4.1.4 bzw. in Abschnitt 4.1.6 vor. Hier beschränken wir uns auf die konzeptuellen Erweiterungen an den Werkzeugtransaktionen.

3.3.2.1 Erweiterung des Werkzeugtransaktionskonzeptes

Möchte man die aus der unversionierten Version von H-PCTE bekannten Kooperationsmöglichkeiten auch in Kombination mit den Versionierungsfunktionalitäten nutzen, muß man die Semantik der Konfigurationen und der Funktionen `HPcte_tta_start`, `HPcte_tta_create_savepoint` und `HPcte_tta_abort` anpassen.

An der Bearbeitung einer Konfiguration ist im kooperativen Fall nicht mehr eine einzelne WTA beteiligt, sondern alle WTA, die kooperativ an einer Dokumentversion arbeiten. Die aktive Arbeitskonfiguration wird durch die erste gestartete WTA angelegt. Alle weiteren kooperativen WTA nutzen dieselbe Konfiguration als Arbeitskonfiguration. Beim Starten einer WTA kann anhand des Zustands der gewählten Basiskonfiguration entschieden werden, ob eine neue Arbeitskonfiguration angelegt werden muß oder nicht. Ist die gewählte Basiskonfiguration noch als aktiv markiert, so wird diese als Arbeitskonfiguration gewählt, sofern die Kooperation

[6] Auf den verwendeten Sperr-Modus gehen wir hier nicht im Detail ein, da die Sperrmodi unverändert bleiben. Die Abhängigkeiten und Kompatibilitäten zwischen den Sperren kann in [187] nachgelesen werden.

gewünscht ist, andernfalls wird eine neue Konfiguration erzeugt. Die Arbeitskonfiguration wird im Fall der Kooperation erst eingefroren, wenn die letzte WTA beendet wird.
Nicht in jedem Fall ist eine synchrone Kooperation gewünscht; die isolierte Arbeit ist teilweise zu bevorzugen. Dieser Fall wird berücksichtigt, indem zwei Modi der WTA unterschieden werden:

- `HPCTE_COOPERATIVE`: Die WTA arbeiten kooperativ und nutzen dieselbe Arbeitskonfiguration.
- `HPCTE_ISOLATED`: Die WTA arbeiten isoliert; jede WTA nutzt ihre eigene Arbeitskonfiguration.

Um entscheiden zu können, ob eine als Basiskonfiguration gewählte aktive Konfiguration als Arbeitskonfiguration nutzbar ist, muß der Kooperationsmodus der WTA an der Arbeitskonfiguration vermerkt sein und beim Start ausgewertet werden. In welchen Fällen eine neue Konfiguration als Arbeitskonfiguration angelegt und wann eine existierende Konfiguration verwendet wird, ist in Tabelle 3.1 aufgelistet.

Typ der existierenden Konfiguration	Typ der zu startenden WTA	
	isoliert	kooperativ
keine aktive	isolierte Konf. anlegen	kooperative Konf. anlegen
isoliert	Variante (isoliert) anlegen	Variante (kooperativ) anlegen
kooperativ	Variante (isoliert) anlegen	als Arbeitskonfiguration wählen

Tabelle 3.1: Verhalten beim Starten einer WTA

Neben dem Verhalten beim Starten der WTA gibt es auch Modifikationen beim Beenden der WTA. Werden alle kooperierenden WTA erfolgreich beendet (*commit*), so wird, nachdem die letzte WTA beendet wurde, die Konfiguration eingefroren. Falls eine Teilmenge der WTA abgebrochen wird, können die verbleibenden WTA weiterhin kooperativ arbeiten und nach dem Commit die Konfiguration einfrieren[7]. Ausschließlich in dem Fall, daß alle WTA die Funktion `HPcte_tta_abort` aufrufen, wird auch die Konfiguration wieder entfernt, einschließlich aller Änderungen.

Der Kooperationsmodus der Konfiguration ist in einem Attribut des Konfigurationsobjektes gespeichert. Da an der Kooperation nicht ausschließlich verschiedene Werkzeuge eines Entwicklers beteiligt sein können, sondern wahrscheinlich auch mehrere Entwickler beteiligt sind, referenziert das Konfigurationsobjekt nicht mehr nur ein Benutzerobjekt, sondern es existieren Links auf alle Benutzerobjekte der beteiligten Entwickler.

3.3.2.2 Sicherungspunkte bei kooperativer Arbeit

Besondere Beachtung benötigen die Konfigurationen, die als Sicherungspunkte angelegt wurden. Diese Konfigurationen sollen den Zustand des Dokumentes sichern, so daß es zu einem späteren Zeitpunkt wiederhergestellt werden kann, falls die folgenden Änderungen sich als unzureichend erweisen sollten. Zu berücksichtigen gilt es hierbei, daß mehrere Entwickler kooperativ arbeiten, der Sicherungspunkt jedoch nicht von allen Entwicklern benötigt oder explizit gewünscht wird. D.h. der Entwickler, der den Sicherungspunkt anlegt, möchte, daß seine Änderungen bis zu dem Sicherungspunkt bei einem Abbruch der WTA erhalten bleiben, für die anderen Entwickler wäre

[7]Das gilt auch, wenn eine WTA ein Commit durchführt und alle anderen WTA anschließend abgebrochen werden.

das eine nicht gewünschte Verhaltensweise. Jedoch sollten alle Entwickler nach Anlegen eines Sicherungspunktes weiterhin kooperativ arbeiten können.
Die Lösung sieht so aus, daß, wenn ein Entwickler einen Sicherungspunkt anlegt, eine neue Arbeitskonfiguration, die alle weiteren Änderungen zusammenfaßt, erzeugt wird, wie im nicht kooperativen Fall. Diese neue Konfiguration wird dann automatisch zur Arbeitskonfiguration der kooperierenden Entwickler. Jedoch stellt diese Konfiguration keinen Sicherungspunkt für diese Entwickler dar. Ein Abbruch ihrer WTA würde alle Änderungen, die sie seit dem Start der WTA durchführten, zurücknehmen. Das Abbrechen aller WTA nach dem Anlegen eines Sicherungspunktes resultiert darin, daß alle Änderungen zurückgenommen werden, mit Ausnahme der Änderungen bis zum Anlegen des Sicherungspunktes, die in der WTA gemacht wurden, die den Sicherungspunkt anlegte. Diese Änderungen werden durch die ursprüngliche Arbeitskonfiguration zusammengefaßt. Daher kann die letzte Konfiguration ebenfalls wieder gelöscht werden.
Zu beachten ist, daß die neue Konfiguration nur einen Sicherungspunkt für die Änderungen des Entwicklers darstellt, der diesen angelegte. Wird die WTA eines anderen Entwicklers abgebrochen, ändert sich der Zustand des Dokumentes auch für den Zeitpunkt, an dem der Sicherungspunkt angelegt wurde! Dieses Verhalten tritt insb. dann auf, wenn die Werkzeuge eine WTA mit der Sperr-Strategie `READ_UNPROTECTED_*` verwenden. In diesem Fall greifen die kooperierenden WTA auf Änderungen der anderen WTA zu, die noch nicht durch deren Commit persistent gemacht wurden. Daher sind auch noch nicht alle Versionen, die durch die ursprüngliche Arbeitskonfiguration zusammengefaßt werden, persistent, mit Ausnahme der Versionen, die durch die WTA angelegt wurden, die den Sicherungspunkt anlegte.
Würden die WTA eine Sperr-Strategie nutzen, die neben den Schreib- auch Lese-Sperren anfordert, so würden die kooperierenden Werkzeuge die Änderungen entweder nicht durchführen resp. nicht anzeigen können, abhängig davon, welches Werkzeug zuerst auf das Dokument zugreift. Diese restriktivere Sperr-Strategie verhindert jedoch nahezu vollständig die Kooperation, da alle Zugriffe serialisiert werden. Hier muß sich der Werkzeugentwickler zwischen diesen Alternativen entscheiden.
Dieser Fall tritt aber nur auf, wenn alle beteiligten WTA am selben Dokument arbeiten. Da eine ETA jedoch mehrere Dokumente verwalten kann und somit auch kooperierende WTA an verschiedenen Dokumenten arbeiten können[8], entschärft sich diese Problematik.

Die selbstreferentielle Verwaltung der Konfigurationen ermöglicht es, daß kooperierende Entwickler über neu angelegte Sicherungspunkt-Konfigurationen mit Hilfe des Benachrichtigungsmechanismus informiert werden und sich mit diesen vor einem Abbruch der eigenen WTA absprechen können.

3.3.3 Versionierung der Objekte, Links und Attribute

Bei den bisherigen Betrachtungen sind wir nicht im Detail auf die Versionierung der Objekte und Links eingegangen, sondern haben die Betrachtung auf die abstrakte Ebene der ETA und der Konfigurationen beschränkt. Bei der Versionierung ist zu beachten, daß die Dokumente feinkörnig modelliert sind und daß die Werkzeuge nach dem Konzept der Multiple-View-Integration arbeiten. Aufgrund der Modellierung bestehen die Dokumente i.d.R. aus einer großen Anzahl von Objekten und Links. Daher ist es nicht praktikabel, bei einer Änderungen am Dokument von allen Objekten des Dokumentes eine neue Version anzulegen, da das in einer großen Anzahl an Versionen resultiert, die sich nicht von ihrer Vorgängerversion unterscheiden. Weiterhin muß nicht in allen Werkzeugen das gesamte Dokument aufgrund unterschiedlicher

[8] Bsp.: In einem Werkzeug wird ein Klassendiagramm erweitert und in einem anderem Werkzeug ein zugehöriges Aktivitätsdiagramm. Diese Änderungen beeinflussen sich i.d.R. nur in geringem Maß.

Sichten zugreifbar sein. Einzelne Teildokumente könnten im Werkzeug gar nicht sichtbar sein. Würde man bei einer Änderung von allen Objekten des gesamten Dokuments neue Versionen anlegen, so würde das auch Objekte betreffen, die gar nicht änderbar sind. Eine vergleichbare Problematik ergibt sich auch bei der Versionspropagierung. In diesem Fall kann der Fall auftreten, daß die Dokumentwurzel im Werkzeug nicht sichtbar ist. Von dieser müßte aber auch eine neue Version angelegt werden, da bei dieser Technik eine Objektversion eine gebundene Konfiguration für ihre Komponenten darstellt.
Die Lösung ist folgende. Durch die explizite Modellierung von Konfigurationen kann jedes Objekt einzeln und unabhängig von seinen Komponenten und unabhängig vom komplexen Objekt, zu dem es gehört, versioniert werden. Das bietet den Vorteil, daß nicht mehr Versionen von Objekten oder Links erzeugt werden als notwendig. Die Versionen werden ausschließlich in WTA erzeugt, deren Arbeitskonfiguration die neuen Versionen zusammenfaßt. Die Zuordnung von neu angelegter Version zu Konfiguration basiert auf den Konfigurationsidentifizierern. Jede Version erhält den K-ID, der Arbeitskonfiguration der WTA, in der die Version erzeugt wurde, als Versionsidentifizierer (V-ID). Man kann daher anhand des V-ID erkennen, in welcher Konfiguration eine Version erzeugt wurde. Daraus folgt aber auch, daß die Versionen keine fortlaufenden Identifizierer besitzen. Das wäre nur dann der Fall, wenn ein Objekt in jeder Konfiguration verändert würde.

Feinkörnige Versionierung Die Beschränkung auf Objekte als kleinste versionierte Einheit hat den Vorteil, daß relativ wenig Metadaten zur Verwaltung der Objektversionen benötigt werden. Der Nachteil ist jedoch, daß man keine Möglichkeit hat, die Unterschiede zwischen einzelnen Versionen ohne einen Vergleich der Attribute und Links zu erkennen. Erweitert man jedoch die Versionierung auf alle Datenmodell-Elemente, also versioniert man neben den Objekten auch die Links und Attribute, so ist es möglich, allein anhand des V-ID zu erkennen, in welcher Konfiguration ein Attribut oder Link verändert wurde. Das bietet einige Vorteile:

- Die Änderung eines Link-Attributs führt nicht zu einer neuen Version des Ausgangsobjekts, es wird nur das Attribut selbst versioniert.
- Durch die Multiple-View-Integration kann ein Objekt-Typ mehr Attribute und Links besitzen, als im aktuellen Arbeitsschema eines Werkzeugs sichtbar sind. Änderungen an einem Attribut führen durch die unabhängige Versionierung (konzeptionell) nicht zu neuen Versionen der nicht sichtbaren Attribute.
- Unterschiede zwischen einzelnen Versionen können besser visualisiert werden, indem man die Differenzen gruppiert anzeigt (siehe Abschnitt 5.3).
- Die Wahrscheinlichkeit eines Konflikts beim Mischen von Versionen reduziert sich, da man die Änderungen an unterschiedlichen Attributen eines Objekts oder Links und Änderungen an einem Attribut anhand der separaten V-ID unterscheiden kann.
- Das Sperrmodell und das Versionierungsmodell sind besser aufeinander abgestimmt, da sie auf denselben Datenmodell-Elementen arbeiten, d.h. eine Änderung eines Attributs führt nur zu dessen Versionierung und auch nur zu dessen Sperrung, andere Attribute oder sogar Links werden nicht beeinflußt.

Diese feinkörnige Modellierung hat jedoch auch den Nachteil, daß der Anteil der Metadaten deutlich ansteigt und evtl. mehr Metadaten als Nutzdaten zu verwalten sind. Das gilt insbesondere dann, wenn ein Objekt überwiegend numerische Attribute besitzt. Vergleichbares gilt, wenn in Textattributen nur einzelne Wörter gespeichert sind (siehe Abschnitt 4.1.3).

3.4 Zusammenfassung

In diesem Kapitel haben wir das Versionierungskonzept für feinkörnig modellierte Dokumente aus der Softwareentwicklung vorgestellt. Ein Unterscheidungskriterium zu den meisten Versionsverwaltungs-Systemen besteht darin, daß das Editiermodell auf dem die Dokumente basieren, die Grundlage der Versionierung ist. Eine notwendige Voraussetzung hierfür besteht darin, daß das Versionsverwaltungs-System Zugriff auf das Editiermodell hat. Das erfordert eine größere Integration von CASE-Werkzeugen und Versionsverwaltungs-System. Jedoch darf hierdurch nicht die Entwicklung der Werkzeuge in zu großem Maß verkompliziert werden, so daß eine weitgehend automatische Versionierung der Knoten des Editiermodells notwendig ist. Hierdurch hat man das Problem, konsistente Versionen von Knoten des Editiermodells auswählen zu müssen. Die Lösung besteht darin, nicht die einzelnen Versionen eines Knotens zur Auswahl anzubieten, sondern Zusammenstellungen hiervon, die Konfigurationen. Um den Entwicklern den Zugriff auf bestimmte Versionen zu erleichtern, sind die Konfigurationen anhand der Aufgaben, in deren Kontext die Änderungen durchgeführt wurden, strukturiert. Das entlastet die Entwickler bei der Versionsauswahl zwischen einer Vielzahl an Konfigurationen wählen zu müssen; die Auswahl der durchzuführenden Aufgabe ist ausreichend.
Mit dieser Lösung erreicht man eine stärkere Orientierung am Projektablauf anstatt an der zeitlichen Reihenfolge, wie die Versionen angelegt wurden, wie z. B. in Historian [3]. Diese Lösung basiert auf einer zustandsbasierten Versionierung der Dokumente, bietet jedoch auch einige Vorteile von änderungsbasierten Versionierungskonzepten, die auf Change-Packages beruhen.
Im Vergleich zu Coven [46] repräsentieren die Konfigurationen im vorgestellten Konzept nur eine Menge von Versionen, die im Rahmen derselben Werkzeugsitzung angelegt wurden. Die Compound Artifacts in Coven dienen zusätzlich der Strukturierung der Dokumente in einzelne Teildokumente. Die gemeinsame Versionierung von zusammengehörigen Dokumenten ist somit in Coven nicht möglich. Das ist jedoch notwendig, wenn eine Multiple-View-Integration vorliegt, in der die Dokumente durch unterschiedliche Sichten auf ein gemeinsames Modell realisiert sein können.

Im Unterschied zu allen vorgestellten Versionierungskonzepten beschränkt sich die Versionierung nicht auf vollständige Dateien oder einzelne Objekte, sondern sie berücksichtigt einzelne Attribute und Links. Dieser Informationsgewinn, in welcher Werkzeugsitzung ein Attribut oder Link verändert wurde, läßt sich bei der Berechnung und Anzeige sowie dem späteren Mischen von Differenzen ausnutzen.

Kapitel 4

Realisierungs-Aspekte des Versionsverwaltungskonzeptes

Im vorherigen Kapitel haben wir das Versionsverwaltungskonzept vorgestellt. Bei der Umsetzung sind jedoch einige Aspekte zu berücksichtigen, die wir in diesem Kapitel näher beleuchten. Diese sind nicht vollständig an die Realisierung in H-PCTE gebunden, sondern können in vergleichbarer Form auch bei der Realisierung in anderen Systemen auftreten. Wir beginnen die Betrachtung mit der Realisierung des Versionsverwaltungskonzeptes in H-PCTE (Abschnitt 4.1). Anschließend skizzieren wir die notwendigen Änderungen in PI-SET, die notwendig waren, um PI-SET um die Versionierung zu erweitern (Abschnitt 4.2).

4.1 Realisierung in H-PCTE

Die Realisierung in H-PCTE ist von einigen Entscheidungen abhängig, die teilweise von der System-Architektur von H-PCTE bestimmt werden. Daher beginnen wir die Betrachtung mit einer kurzen Übersicht der Datenstrukturen von H-PCTE (Abschnitt 4.1.1). Die Erweiterung von H-PCTE um das Versionsverwaltungskonzept macht Änderungen an unterschiedlichen Teilsystemen notwendig. Die geringsten Auswirkungen auf die bestehende Architektur hat die selbstreferentielle Verwaltung der Metadaten (Abschnitt 4.1.2), da diese i.w. nur eine Erweiterung der bestehenden Schnittstelle darstellt. Die Versionsverwaltung selbst bedarf weitreichender Änderungen an der vorhandenen Realisierung (Abschnitt 4.1.3), die auch Auswirkungen auf andere Funktionsbereiche hat. Zu nennen sind hier die Sperrverwaltung (Abschnitt 4.1.4), Änderungen am Recovery (Abschnitt 4.1.5) und am Benachrichtigungsmechanismus (Abschnitt 4.1.6). Diese Änderungen und Erweiterungen sind nicht ausschließlich spezifisch für die Implementierung, sondern sind auch konzeptueller Natur.

4.1.1 Übersicht der Datenstrukturen

Bei dieser Übersicht beschränken wir uns auf die internen Datenstrukturen von H-PCTE, die für die Realisierung der Versionsverwaltung relevant sind. Für Fragen, die die Abbildung der H-PCTE Prozesse auf Betriebssystem-Prozesse, die Umsetzung der Sperrverwaltung, des Benachrichtigungsmechanismus oder der Transaktionsverwaltung betreffen, sei auf [187] verwiesen, die persistente Speicherung der Daten wird in [174] beschrieben.

Abbildung 4.1 zeigt die Datenhaltung von H-PCTE, die sich in persistente und transiente Daten unterteilen läßt. Transiente Daten sind als solche in der Abbildung gekennzeichnet. Aus Gründen der Übersichtlichkeit ist nur die Struktur dargestellt, auf die Darstellung von Details wurde verzichtet.

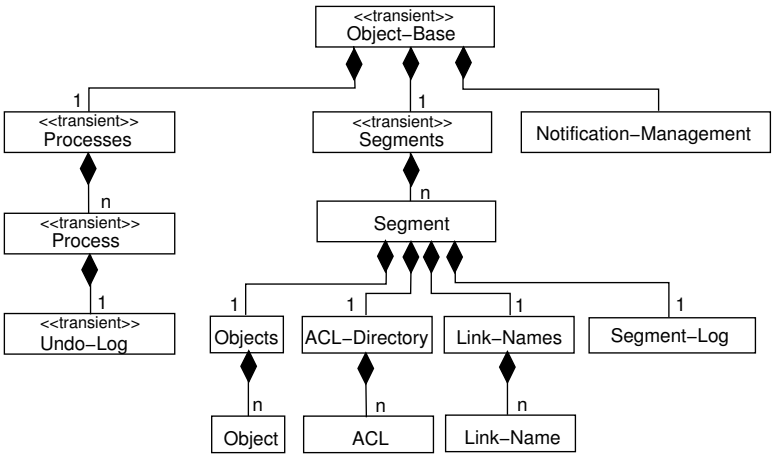


Abbildung 4.1: Interne Datenhaltung von H-PCTE

Die persistenten Daten sind auf das Verteilungsmodell von H-PCTE ausgerichtet. Daher basiert es auf der Verwaltung der Segmente, die die einzelnen Objekte und Links zusammenfassen. Neben diesen Daten verwaltet ein Segment eine Tabelle (`Link-Names`) zur Umsetzung der Linknamen in deren interne numerische Darstellung und eine Tabelle (`ACL-Directory`) zur Abbildung der Access Control Lists (`ACL`) auf numerische Werte, die an den Objekten gesetzt werden (vgl. Abbildung 4.2, `ACL-No`).

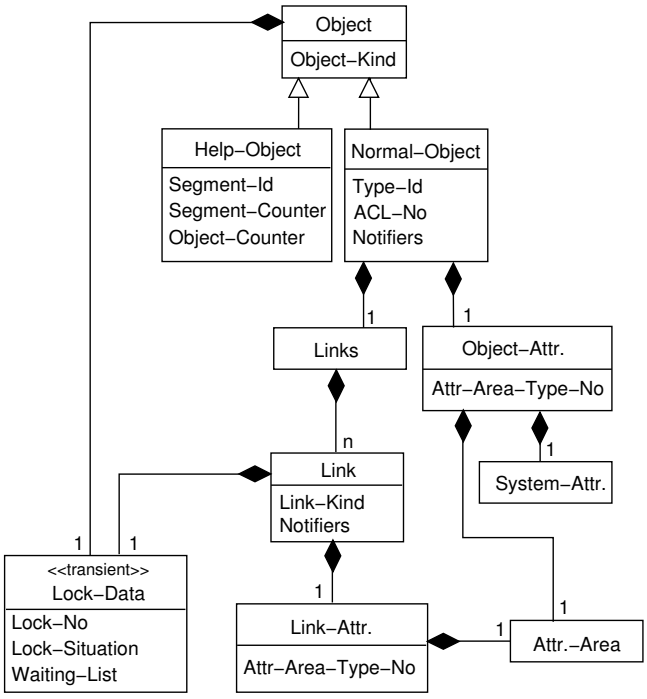


Abbildung 4.2: Interne Strukturierung der Objekte

Die für den Benachrichtigungsmechanismus benötigten Daten sind einerseits in einer systemweiten Verwaltung (`Notification-Management`) der Benachrichtigungswünsche der Prozesse, andererseits in Hinweise (`Notifiers`, siehe Abbildung 4.2) an Objekten und Links über bestehende Benachrichtigungswünsche. Diese Aufteilung entkoppelt das Auslösen einer Benachrichtigung und die Zustellung an die Prozesse, die eine Benachrichtigung wünschen.
Des weiteren gibt es Daten für die Systemwiederherstellung (Recovery), die auf zwei Datenstruk-

turen verteilt sind. Einerseits verwaltet jeder Prozeß (`Process`) eine transiente[1], transaktionsspezifische Liste (`Undo-Log`) mit durchgeführten Änderungsoperationen, um bei einem Transaktionsabbruch die Änderungen rückgängig machen zu können. Für den Fall eines Systemfehlers gibt es eine segmentspezifische Liste (`Segment-Log`) mit Änderungen, um die Änderungen von abgeschlossenen Transaktionen wiederherstellen zu können.

Die Objekte selbst sind weiter strukturiert. Hier gibt es die `Help-Objects`, die zur Verwaltung von segmentübergreifenden Links benötigt werden. Die Nutzdaten der Objekte sind im `Normal-Object` abgelegt. Dieses verwaltet die Attribute `Object-Attributes` und die ausgehenden Links `Links`, die selbst wiederum ihre Attribute verwalten `Attribute-Area`. Die Objekt-Attribute bestehen aus den `System-Attributes` zur Speicherung von Zeitstempeln und der Anzahl von eingehenden und ausgehenden Links, unterteilt nach Link-Kategorien. Zusätzlich gibt es noch die benutzerdefinierten Attribute `Attribute-Area`. Aufgrund des Sichtenmechanismus für die Multiple-View-Integration muß die Verwaltung der benutzerdefinierten Attribute dynamisch sein. Abhängig von der in einem Prozeß gesetzten Sicht, kann es erforderlich sein, die Attribut-Verwaltung um weitere Attribute zu erweitern. Das ist der Fall, wenn auf die Attribute im Rahmen einer Sicht erstmalig schreibend zugegriffen wird.
Aufgrund der feinkörnigen Sperrverwaltung ist es notwendig, daß an jedem Objekt und Link die aktuelle Sperrsituation (`Lock-Data`) vermerkt wird. Diese beinhaltet zusätzlich eine Liste von Transaktionen, die auf die Zuteilung einer Sperre warten.

4.1.2 Verwaltung der selbstreferentiellen Metadaten

Die Entwurfstransaktionen (ETA) und die Konfigurationen werden selbstreferentiell[2] im OMS als Objekte verwaltet, wie es im SDS `scm` in Abbildung 3.5 auf Seite 96 und im Anhang A dargestellt ist. Diese Operationen können weitgehend unter Verwendung der existierenden Schnittstellen zur Bearbeitung von Objekten und Links realisiert werden. Diese Objekte und Links dienen zur Verwaltung der im Versionierungskonzept benötigten Metadaten.
Es gibt jedoch einige Besonderheiten zu beachten. Um zu verhindern, daß diese Metadaten unter Umgehung der dafür vorgesehenen API modifiziert werden können, sind nur Lesezugriffe erlaubt. Im SDS ist daher der Zugriff auf Typebene auf reine Lesezugriffe eingeschränkt. Innerhalb der Verwaltungsoperationen für ETA, Konfigurationen und Dokumente dürfen daher keine Typrechte geprüft werden. Diese laufen daher im Kontext des Datenbank-Benutzers `root`, der vergleichbar mit dem gleichlautenden Unix-Benutzer Sonderrechte besitzt.
Zusätzlich ist zu berücksichtigen, daß es sich bei der Verwaltung dieser Metadaten einerseits um komplexe Änderungen im OMS handelt, bei denen teilweise mehrere Links oder Objekte betroffen sind, andererseits daß diese Metadaten Grundlage für die kooperative Arbeit mehrerer Entwickler sind. Daher muß besonderer Augenmerk auf die Konsistenz und das Verhindern von Parallelitätsanomalien gelegt werden.

4.1.2.1 Metadaten-Transaktion

Zur Sicherstellung der Konsistenz der Metadaten müssen alle Funktionen, die ETA, Konfigurationen oder Dokumente verwalten, im Rahmen von Transaktionen ausgeführt werden. Somit ist sichergestellt, daß bei Auftreten eines Fehlers bereits durchgeführte Änderungen wieder zurückgenommen und daß in der Zwischenzeit konkurrierende Operationsaufrufe bis zum Abschluß der

[1] Da die persistenten Daten nur beim Beenden des Prozesses auf den persistenten Speicher geschrieben werden, ist die transiente Verwaltung ausreichend. Ein Systemfehler führt somit automatisch zum Rücksetzen aller ausgeführten Änderungen.

[2] Ergänzend zu der selbstreferentiellen Verwaltung, ist es zur Verbesserung der Laufzeit notwendig, einige Daten, wie z. B. den Konfiguration-Identifizierer, in transienten Datenstrukturen zwischen zu speichern.

laufenden Operation verzögert werden. Die Grundlage zur Serialisierung der Zugriffe basiert auf dem Setzen von Sperren an allen betroffenen Objekten und Links. Hierbei ist jedoch zu beachten, daß die Sperren nach Abschluß der Operation wieder zurückgezogen werden müssen, damit kooperierende Entwickler Zugriff auf die Metadaten haben. Es ist somit nicht möglich, diese Operationen im Rahmen der in H-PCTE verwendeten Werkzeugtransaktionen (WTA) auszuführen, da diese die Sperren erst nach Abschluß der Wurzel-Transaktion zurückziehen. Die Metadaten könnten bei Auftreten eines Fehler zurückgenommen werden und konkurrierende Operationen würden verzögert, jedoch wäre eine Kooperation durch die existierenden Sperren unmöglich. Beispielsweise könnten kooperierende Entwickler keine Sicherungspunkt-Konfigurationen anlegen oder Kommentare am Konfigurationsobjekt setzen (siehe Abschnitt 4.2).
Würde man das Setzen von Sperren innerhalb der Metadaten-Operationen unterlassen, kämen als weitere Problematik Abhängigkeiten zwischen unterschiedlichen WTA hinzu. Metadaten, die von einer WTA angelegt worden sind, müssen durch eine andere WTA modifiziert werden, wie z. B. existierende Links müssen durch andere Links mit neuen Zielobjekten ausgetauscht werden. Diese Abhängigkeiten müßte man beim Beenden der beteiligten WTA berücksichtigen, was jedoch nahezu unmöglich ist und auch die Wartbarkeit erheblich verschlechtern würde.

Die Lösung besteht darin, einen weiteren Transaktionstyp (*Metadaten-Transaktion*) einzuführen, der zur Verwaltung der Metadaten dient. Die Metadaten-Transaktionen unterscheiden sich von WTA dadurch, daß sie eine deutlich kürzere Laufzeit besitzen, keine Hierarchie bilden und die gesetzten Sperren sofort beim Beenden zurückziehen. Alle Operationen, die die Metadaten modifizieren, müssen im Rahmen einer Metadaten-Transaktion ausgeführt werden. Das stellt sicher, daß bei Auftreten eines Fehlers alle Änderungen zurückgenommen und daß konkurrierende Änderungen serialisiert werden.

Diese Lösung erfordert jedoch eine genauere Betrachtung der durchzuführenden Aktionen bei Abbruch von WTA, da das auch Auswirkungen auf die Metadaten besitzen kann.

4.1.2.2 Abbruch einer Werkzeugtransaktion

Für den nicht-kooperativen Fall ist der Abbruch einer WTA einfach zu handhaben. Die beim Start angelegten Metadaten müssen wie die Nutzdaten entfernt werden. Für einen Transaktionsabbruch auf Benutzeranforderung könnte diese Funktionalität in der entsprechenden Schnittstelle realisiert werden.
Bei der kooperativen Arbeit mehrerer WTA kommt erschwerend hinzu, daß einzelne WTA abgebrochen und andere WTA erfolgreich beendet werden könnten. Die Konsistenz der Nutzdaten ist durch die Sperren sichergestellt, so daß bei dem Abbruch einer WTA, die anderen ungehindert und ohne Verlust von eigenen Änderungen weiterarbeiten können. Der Abbruch aller WTA ist somit nicht notwendig. Es kann daher der Fall auftreten, daß die WTA, die ursprünglich die Metadaten anlegte, abgebrochen wird und alle anderen WTA weiterarbeiten. Würde der Abbruch der initialen WTA auch das Löschen der Metadaten bedeuten, müßten die kooperativen WTA abgebrochen werden, was nicht notwendig ist. Ein einfacher Eintrag in den Undo-Log der initialen WTA, der das Löschen der Metadaten auslöst, ist somit nicht die Lösung.
Eine mögliche Lösung ist, mit jeder Konfiguration einen Zähler zu verwalten, der angibt, wieviel WTA kooperativ an dieser Konfiguration arbeiten. Für jede neu gestartete WTA wird dieser Zähler in der Metadaten-Transaktion inkrementiert. Beim erfolgreichen Beenden einer WTA erfolgt keine Änderung des Zählers, nur bei einem Transaktionsabbruch wird der Zähler dekrementiert. Nach dem Dekrementieren muß geprüft werden, ob der Zähler den Wert Null erreicht hat. In diesem Fall arbeitet keine WTA mehr an der Konfiguration und es wurde auch noch keine erfolgreich beendet, so daß die betreffenden Metadaten wieder entfernt werden müssen. Somit ist sichergestellt, daß, sofern es noch eine aktive WTA für eine Konfiguration gibt, die

Metadaten nicht fälschlicherweise gelöscht werden und daß nach Abbruch aller WTA keine ungültigen Metadaten im OMS verbleiben.
Das Dekrementieren des Zähler einschließlich Überprüfung und Löschen der Metadaten, sofern das notwendig ist, kann durch einen zusätzlichen Eintrag in den Undo-Logs aller beteiligten WTA erreicht werden. Dieser Eintrag veranlaßt die oben beschriebenen Aktionen.

Auf einen Transaktionsabbruch aufgrund eines Systemfehlers ließe sich jedoch so nicht reagieren und die Metadaten verblieben im OMS. Das Problem in diesem Fall ist, daß die Metadaten-Transaktion abgeschlossen ist, die auf den durch sie angelegten Metadaten basierenden WTA jedoch unterbrochen wurden. Wenn die WTA nicht als `*_EDITOR_TRANSACTION` (siehe Abschnitt 2.3.5 auf Seite 86) gestartet wurde, gehen durch den Systemfehler alle durch die WTA angelegten Versionen verloren und der Konfiguration wären keine Versionen zugeordnet. Für diesen Fall muß jede WTA in den Segment-Logs notieren, auf welcher Metadaten-Transaktion sie aufsetzt, diese vermerken die Konfiguration, auf die sie sich beziehen. Bei der Systemwiederherstellung, müssen dann die Abhängigkeiten zwischen den WTA, den Metadaten-Transaktionen und den betroffenen Konfigurationen berücksichtigt werden.
Bei der Wiederherstellung müssen alle Metadaten-Transaktionen in Abhängigkeit von der Konfiguration verwaltet werden, d.h. es gibt eine Verwaltung der neu angelegten Konfigurationen, die Verweise auf die korrespondierenden Metadaten-Transaktionen besitzen. Parallel muß für die abgebrochenen WTA bestimmt werden, ob sie (partiell) wiederholt wird. Wenn sie wiederholt wird, behält die entsprechende Metadaten-Transaktion ihre Gültigkeit. Wird die WTA nicht wiederholt, so sind auch die Änderungen der korrespondierenden Metadaten-Transaktion ungültig. Die Metadaten-Transaktion wird dann als ungültig gekennzeichnet. Nachdem dieser Test für alle WTA und Metadaten-Transaktionen ausgeführt wurde, kann für die Konfigurationen entschieden werden, ob sie benötigt werden. Das ist der Fall, wenn mindestens eine Metadaten-Transaktion weiterhin gültig ist. Falls eine Konfiguration nicht mehr benötigt wird, muß sie durch eine inverse Metadaten-Transaktion gelöscht werden. Diese Tests einschließlich der Kennzeichnung muß vor der Wiederherstellung durchgeführt werden.

4.1.2.3 Anlegen einer Sicherungspunkt-Konfiguration

Bei der Realisierung der Sicherungspunkt-Konfigurationen gibt es einige Fälle zu berücksichtigen. Das dem Sicherungspunkt korrespondierende Konfigurationsobjekt wird ebenfalls in einer Metadaten-Transaktion angelegt. Hierfür wird zuvor die im H-PCTE-Prozeß laufende WTA beendet und somit alle Änderungen dieser WTA persistent[3]. Nachdem das neue Konfigurationsobjekt angelegt wurde, wird eine nachfolgende WTA auf der neuen Konfiguration gestartet. Aus Sicht des Werkzeugs ist das Beenden der WTA, das Anlegen der neuen Konfiguration und das Starten der Nachfolge-WTA atomar in der Funktion `HPcte_tta_create_savepoint` realisiert.
Besitzt die WTA, die die Sicherungspunkt-Konfiguration anlegte, kooperierende WTA, so müssen diese über den Sicherungspunkt benachrichtigt werden, damit alle weiteren Änderungen der neuen Konfiguration zugeordnet werden können. Die unterliegenden WTA werden jedoch nicht beendet und neu gestartet, stattdessen erhalten diese lediglich eine Nachricht unter Verwendung des Benachrichtigungsmechanismus von H-PCTE und nutzen anschließend die neue Konfiguration als Arbeitskonfiguration. Hierdurch erhält man das in Abschnitt 3.3.2.2 beschriebene Verhalten. Ein Abbruch einer kooperierenden WTA resultiert in der Rücknahme aller Änderungen dieser WTA, ohne daß die anderen WTA davon beeinflußt werden. Der Abbruch der WTA, die die Sicherungspunkt-Konfiguration anlegte, führt nur zu der Rücknahme der Änderungen, die der neuen Arbeitskonfiguration zugeordnet sind.

[3]Nur unter der Voraussetzung, daß alle umgebenden WTA ebenfalls erfolgreich beendet werden.

4.1.2.4 Zweigverwaltung

Die selbstreferentielle Verwaltung der Konfigurationen ist prinzipiell ausreichend, um die Zweig-ID vergeben und bei Nebenentwicklungszweigen die Konfiguration bestimmen zu können, an der der Zweig angelegt wurde (Ursprungskonfiguration). Hierfür müßten bei jeder Abfrage dieser Daten alle Konfigurationsobjekte unter Berücksichtigung der Struktur eingelesen werden. Das würde die Laufzeit stark verlängern. Daher ist es notwendig, diese Daten in einer Datenstruktur zwischenzuspeichern, um einen performanten Zugriff darauf zu ermöglichen.
Das Einlesen der Konfigurationen stellt jedoch auch einen initialen Aufwand dar, der die zum Start der Werkzeuge benötigte Zeit erhöht. Daher sollten diese Daten persistent abgelegt werden. Zu berücksichtigen ist dann jedoch, daß die neu vergebenen Zweig-ID einerseits eindeutig sein müssen, andererseits daß der Zugriff auf die Ursprungskonfigurationen von Nebenentwicklungszweigen performant sein muß[4].
Die erste Forderung läßt sich durch eine zentrale Verwaltung der Konfiguration-ID erreichen, was jedoch der zweiten Forderung widerspricht. Berücksichtigt man, daß neue Zweige eher selten angelegt werden, daß aber lesende Zugriffe auf die Zweigverwaltung häufig sind, so kann der H-PCTE-Server um eine zentrale Zweigverwaltung erweitert werden. Diese ist primär für die Vergabe neuer Zweig-ID zuständig. Jeder H-PCTE-Klient erhält darüber hinaus eine Kopie der Zweigverwaltung, die für Abfragen von Ursprungskonfigurationen genutzt wird. Beim Anlegen eines neuen Zweiges fordert ein Klient eine neue Zweig-ID an, diese trägt er in seine lokale Kopie ein. Der H-PCTE-Server muß darüber hinaus nach der Vergabe einer neuen Zweig-ID die anderen Klienten über die neue Zweig-ID informieren und die Daten der zentralen Zweigverwaltung speichern, um neu gestarteten Klienten eine aktuelle Kopie zur Verfügung zu stellen. Dieses Speichern stellt jedoch keinen Laufzeit-Engpaß dar, da Klient und Server nach der Vergabe asynchron weiterarbeiten.

4.1.3 Verwaltung der Versionen

Die Verwaltung und der Zugriff auf die Versionen erfordert umfassende Änderungen an der Verwaltung der Objekte und Links. Zu berücksichtigen sind hier diverse Fragestellungen beim Anlegen von neuen Versionen, Fragen die den Übergang von versionierten und nicht versionierten Objekten betreffen und Überlegungen zur technischen Umsetzung der Versionsverwaltung.

4.1.3.1 Anlegen von Versionen

Es gibt drei verschiedene Gründe, warum eine Version neu angelegt werden muß:

1. Modifizieren eines Objektes oder Links
2. Anlegen eines neuen Objektes oder Links
3. Löschen eines Objektes oder Links

[4]Müßte bei jeder Abfrage der Ursprungskonfiguration eine Anfrage an den Server gestellt werden, würde das viele Operationsausführungen deutlich verlangsamen. Das Bestimmen der Ursprungskonfiguration ist nicht ausschließlich bei Modifikationen an Konfigurationszweigen notwendig, sondern auch häufig dann, wenn auf eine Objekt- oder Linkversion zugegriffen wird. Das resultiert daher, daß nur dann eine neue Version angelegt wird, wenn diese verändert wurde. Daher kann z. B. der Fall eintreten, daß ein Werkzeug in einem Nebenentwicklungszweig arbeitet, aber von dem Objekt, auf das zugegriffen werden soll, noch keine Version in diesem Zweig existiert. Dann muß die Ursprungskonfiguration des Nebenentwicklungszweiges bestimmt werden, um auf die dieser Konfiguration zugeordneten Objektversion zugreifen zu können.

Im ersten Fall existiert bereits mindestens eine Version, die in einer WTA modifiziert werden soll. Mögliche Änderungen sind das Setzen eines neuen Attribut-Wertes oder das Erzeugen oder Löschen eines Links an einem Objekt. Letzteres ist auf implementierungs-spezifische Gründe (siehe Abschnitt 4.1.3.3) zurückzuführen. Der zweite Fall ist trivial, da noch keine Version existiert, muß eine neue angelegt werden.
Das Löschen eines Objektes oder Links stellt auch eine Änderung derselben dar, so daß eine neue Version anzulegen ist. Diese Version unterscheidet sich jedoch von den Vorgängerversionen dadurch, daß sie nur eine Kennzeichnung ist, daß das Objekt oder der Link gelöscht wurde. Das Anlegen der als gelöscht markierten Version ist notwendig, da die Objekte oder Links aufgrund der Versionierung nicht vollständig im OMS gelöscht werden dürfen, um Zugriffe auf ältere Versionen zu ermöglichen.
Aufgrund des navigierenden Zugriffs würde es ausreichen, nur von gelöschten Links eine neue Version anzulegen. Wenn die Links gelöscht sind, sind auch deren Zielobjekte nicht mehr zugreifbar. Das Problem ist jedoch, daß in laufenden H-PCTE-Prozessen noch Objekt-Referenzen existieren können, die auf das gelöschte Objekt verweisen. Ein Zugriff auf das Objekt mittels dieser Referenzen wäre somit noch möglich, was jedoch durch die neue Version als Löschmarkierung verhindert wird.
Diese Überlegungen gelten jedoch nur für `Normal-Objects`. `Help-Objects` brauchen nicht als gelöscht markiert werden, da diese ausschließlich OMS-intern für Verwaltungszwecke genutzt werden und daher keine Referenzen auf sie existieren können.

Die unterschiedlichen Ereignisse, die das Anlegen einer neuen Version erfordern, würden Änderungen an vielen Schnittstellen von H-PCTE erforderlich machen. Die Überprüfung, ob eine neue Version benötigt wird und das anschließende Anlegen läßt sich jedoch zentral in einer Funktion des OMS realisieren. Versionen werden nur im Rahmen von WTA angelegt. Diese fordern, daß vor der eigentlichen Änderung das Objekt oder der Link gesperrt wird. Kombiniert man jetzt Sperranforderung und Versionierung (siehe Abschnitt 4.1.4), läßt sich H-PCTE um die Versionierungsfunktionalität erweitern, ohne alle Schnittstellen erweitern zu müssen. Zusätzlich zu dem Anlegen der neuen Version sind Erweiterungen im Bereich des Recovery (siehe Abschnitt 4.1.5) notwendig, um bei einem Transaktionsabbruch die angelegten Versionen wieder zu entfernen.

4.1.3.2 Versionierte und unversionierte Objekte und Links

In einer Instanz von H-PCTE gibt es versionierte und unversionierte Objekte und Links. Der Grund hierfür liegt in der ausschließlichen Versionierung innerhalb von WTA. Jeder Zugriff außerhalb einer WTA führt *nicht* zur Versionierung. Die explizite Verwaltung der Dokumente durch die Entwurfstransaktionen ermöglicht eine klare Unterscheidung zwischen versionierten und unversionierten Dokumenten.
Alle Versionen erhalten, wie bereits erwähnt, als Versionsidentifizierer die Konfigurations-ID der Arbeitskonfiguration, der sie zugeordnet sind. Zur Vereinfachung der Objekt- und Link-Verwaltung wird eine Pseudo-Konfiguration eingeführt, der alle unversionierten Objekte und Links zugeordnet werden. Diese Pseudo-Konfiguration besitzt die Zweig-ID '0' und die Konfigurationsnummer '0'. Beide zusammen bilden die Konfigurations-ID der Pseudo-Konfiguration. Alle unversionierten Objekte und Links erhalten diese Konfigurations-ID. Somit ist eine einheitliche Verwaltung von versionierten und unversionierten Objekten und Links möglich[5].

[5]Diese Lösung eröffnet auch die Möglichkeit, im Rahmen einer WTA versionierte und unversionierte Dokumente zu bearbeiten. Ob ein Objekt versioniert werden müßte, ließe sich anhand der Konfigurations-ID feststellen. Alle Objekte die die Konfigurations-ID '0;0' besitzen sind unversioniert und dürfen bei Änderungen nicht versioniert werden, von allen anderen muß eine Version angelegt werden. Wird ein Objekt erzeugt, ist anhand

Durch die klare Trennung von versionierten und unversionierten Dokumenten im Rahmen der ETA gibt es nur an den Arbeitsbereichsobjekten einen Übergang von unversionierten zu versionierten Objekten. Dieser steht jedoch vollständig unter der Kontrolle der Verwaltungsfunktionen für der ETA. Daher stellen diese Übergänge kein Problem dar.
Ohne die klare Trennung von versionierten und unversionierten Dokumenten würde es an mehreren Objekten Übergänge von unversionierten zu versionierten Objekten geben, die nicht durch eine spezielle Schnittstelle kontrolliert sind. Das würde verschiedene Probleme verursachen, teilweise konzeptuelle aber auch implementierungsspezifische.
Greift man auf die unversionierten Objekte außerhalb einer WTA zu, so sind alle Links auf versionierte Objekte nicht sichtbar, d.h. es ist für ein Werkzeug nicht ersichtlich, daß dieses Objekt weitere Objekte referenziert. Soll dieses unversionierte Objekt oder aber auch das gesamte komplexe Objekt, sofern das atomare Objekt Teil von einem ist, gelöscht werden, müssen entweder alle versionierten Objekte ebenfalls gelöscht werden oder das Löschen muß verweigert werden. Der erste Fall verbietet sich, da er dem Löschen aller Versionen gleichzusetzen ist. Er widerspricht also der Grundidee der Versionierung. Im zweiten Fall würde das Löschen eines Objektes verweigert, weil etwas existiert, was aber gar nicht sichtbar ist.
Der umgekehrte Fall, daß von einem versionierten Objekt unversionierte Objekte angelegt werden, stellt einen Widerspruch in sich selbst dar. Ein unversioniertes atomares Objekt als Bestandteil eines versionierten komplexen Objektes gehört zu dessen Version. Würde man das unversionierte Objekt in einer WTA löschen, so würde man auch eingefrorene Versionen des komplexen Objektes, also des Dokumentes, nachträglich ändern. Da diese jedoch eingefroren sind, dürfen sie nicht änderbar sein. Allgemein stellt sich die Frage, welche Anwendungsfälle für diese Kombination von unversionierten und versionierten Objekten existieren.
Erschwerend käme hinzu, daß die System-Attribute eines Objektes nicht berechnet werden, sondern am Objekt selbst gespeichert sind. An Objekten, die Links auf versionierte Objekte besitzen, sind die Werte der System-Attribute jedoch davon abhängig, ob auf diese Objekte in oder außerhalb einer WTA zugegriffen wird, also ob die versionierten Objekte sichtbar oder unsichtbar sind. Die System-Attribute der unversionierten Objekte müssen daher auch versioniert werden.

4.1.3.3 Implementierungsaspekte

Bei der Realisierung der Verwaltung der einzelnen Versionen von Objekten und Links gilt es, zuvor deren interne Datenhaltung genauer zu analysieren. Ein Objekt besteht aus:

- den Attributen, die sich in:
  - die System-Attribute und
  - benutzerdefinierten Attribute aufteilen,
- der Verwaltung der Links,
- der am Objekt gesetzten ACL,
- Informationen für die Sperrverwaltung,
- Informationen für den Benachrichtigungsmechanismus und
- weitere (hier nicht relevante) Verwaltungsdaten.

des Ursprungsobjekts feststellbar, ob das neue Objekt versioniert oder unversioniert sein muß. Aufgrund des fehlenden Einsatzbereichs für die kombinierte Verwaltung von versionierten und unversionierten Dokumenten wurde hiervon Abstand genommen.

Die Links besitzen nur benutzerdefinierte Attribute, Informationen für den Benachrichtigungsmechanismus und die Sperrverwaltung. Betrachtet man die Abhängigkeiten dieser Daten, so kann man drei Arten von Daten unterscheiden:

1. persistente, versionierbare Daten: z. B. Attribute, Links, Verwaltungsdaten der Versionsverwaltung selbst und die ACL

2. persistente, unversionierte Daten: z. B. `Object-Kind` (Unterscheidung zwischen: `Normal-Object`, `Help-Object` und Löschmarkierung), Objekt-ID

3. transiente versionsspezifische Daten: z. B. Informationen für die Sperrverwaltung und den Benachrichtigungsmechanismus

Zur Vermeidung von redundanter Speicherung von versionsunabhängigen Daten müßten diese unabhängig von den versionsspezifischen Daten verwaltet werden. Ergänzend werden rein konzeptuell betrachtet alle Attribute einzeln versioniert (siehe Abschnitt 3.3.3). Diese feinkörnige Versionierung bedeutet jedoch für die Implementierung einen erhöhten Aufwand, einmal an Speicherbedarf und auch an Laufzeit.
Man müßte für alle versionsspezifischen Daten eines Objektes eine eigene Versionsverwaltung implementieren. Das reduziert den Speicherbedarf für die Nutzdaten, jedoch bedeutet es einen hohen Aufwand an Verwaltungsinformationen: alle Versionen müssen einer Konfiguration zugeordnet werden (zwei zusätzliche Integer-Werte für jedes Attribut, auch für Attribute vom Typ Boolean!) und es werden weitere Informationen für das Recovery (siehe Abschnitt 4.1.5 auf Seite 123) benötigt.
Beim Zugriff auf diese Daten müßte immer die passende Version der Attribute, Sperrdaten usw. bestimmt werden. Das würde die Laufzeit erheblich verlängern.

Daher ist ein Kompromiß zwischen dem Speicherbedarf für die Nutzdaten der Versionen, für die Verwaltungsinformationen und der Laufzeit zum Zugriff auf eine einzelne Version zu finden. Dieser Kompromiß ist jedoch stark von der speziellen Modellierung abhängig, d.h. je nach Werkzeug, im OMS gespeicherten Dokumenttypen und deren Versionierung können bei einer feinkörnigen Implementierung der Versionsverwaltung die Nutz- oder die Metadaten überwiegen. Enthalten die Attribute z. B. sehr viele lange Texte, überwiegen die Nutzdaten, enthalten die Attribute jedoch eher kurze Texte oder handelt es sich sogar um Attribute der Typen: Integer, Float, Enumeration oder Boolean, überwiegen die Metadaten deutlich. Ein Beispiel hierfür sind feinkörnig modellierte UML-Diagramme, sofern der Quelltext nicht in einem Attribute gespeichert wird[6].
Berücksichtigt man, daß H-PCTE darauf ausgerichtet ist, die Dauer der Operationsausführungen gering zu halten, um die Arbeit interaktiver Werkzeuge direkt auf der Schnittstelle von H-PCTE zu ermöglichen, so muß das Ziel sein, die Antwortzeiten der einzelnen Operationen durch die Einführung der Versionierung nicht zu stark zu verlängern.

Die gewählte Lösung versioniert aus diesen Überlegungen das Objekt als Ganzes, wobei die Links unabhängig vom Objekt versioniert werden. Das bedeutet zwar eine redundante Speicherung von Nutzdaten, ermöglicht aber geringe Zugriffszeiten, da nur einmal pro Operationsausführung die Version des Objektes bestimmt werden muß. Die eigenständige Versionierung aller Daten eines Objektes oder Links hätte auch zu einer vollständigen Reimplementierung der

[6]Bei der in H-PCTE vorliegenden Implementierung von Textattributen nehmen lange Textattribute jedoch eine Sonderstellung ein, da sie extern gespeichert werden und am Objekt nur Verweise verwaltet werden. Hier wäre auch die ergänzende Nutzung von Techniken zur Versionierung von Texten denkbar, um die einzelnen Versionen kompakt zu speichern.

gesamten Datenhaltung von H-PCTE geführt, da diese auf eine zusammenhängende Objekt-Struktur abgestimmt ist.
In der gewählten Lösung konnte die Versionsverwaltung vollständig in den Modulen `Objects` bzw. `Links` realisiert werden. Die Versionen werden getrennt nach der Zugehörigkeit zu den Konfigurationszweigen und zu den Konfigurationen eines Zweiges gespeichert, siehe Abbildung 4.3.

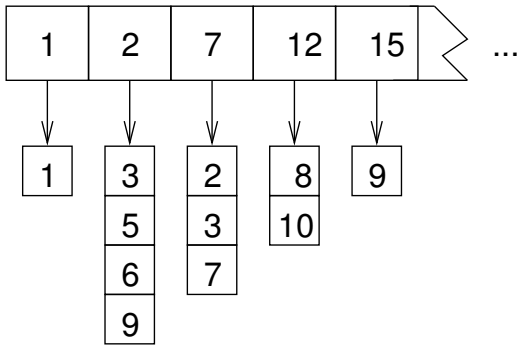


Abbildung 4.3: Interne Versionsverwaltung in `objects` und `links`

Durch die Zuordnung der Versionen zu den Konfigurationen besitzen die Versionen keine fortlaufende Nummer, wie es auch in der Abbildung dargestellt ist. Durch die fortlaufende Vergabe der Nummern ist jedoch sichergestellt, daß alle Versionsnummern aufsteigend sortiert vorliegen. Beim Eintragen einer neuen Version in die Versionsverwaltung kann diese einfach angehängt werden, ohne existierende Versionen verschieben zu müssen, das gilt sowohl bei neuen Zweigen als auch bei neuen Konfigurationen innerhalb eines Zweiges.
Um die konzeptuell unabhängige Versionierung der Attribute erreichen zu können, wird an jeder Objektversion zusätzlich ein Bitfeld gespeichert, welches angibt, ob ein Attribut in dieser Version verändert wurde.

4.1.4 Sperrverwaltung

In dem von Platz [187] vorgestellten Sperrkonzept bezieht sich eine Sperre aufgrund der nicht vorhandenen Versionsverwaltung immer auf das ganze Objekt/Link bzw. einzelne Teile davon. Durch die Einführung von Versionen würde das Sperren des ganzen Objektes/Links (oder Teilen hiervon) mit allen Versionen eine zu große Einschränkung in der kooperativen Arbeit bedeuten. Sperren an Versionen, die ein Werkzeug nicht anzeigt, sollten keine Auswirkungen besitzen. Auf den exakt verwendeten Sperr-Modus gehen wir hier jedoch nicht im Detail ein, da die Sperrmodi unverändert bleiben. Die Abhängigkeiten und Kompatibilitäten zwischen den Sperren kann in [187] nachgelesen werden.

Bei der einfachen Erweiterung des Sperrkonzepts, könnte man auf die Idee kommen, die Sperren an jeder Version zu setzen. Das hat jedoch einige Nachteile. Der entscheidende ist, daß im Rahmen einer WTA nicht eine Version eines Objektes/Links gesperrt wird, sondern mindestens zwei Versionen, sofern vor dem Schreibzugriff ein Lesezugriff erfolgt. In diesem Fall wird als erstes eine Lesesperre auf der letzten Version des betroffenen Versionszweiges gesetzt. Diese Version ist jedoch eingefroren, so daß für Änderungen eine neue Version angelegt werden muß. Auf der Version müßte dann die Schreibsperre angefordert werden. Ist diese neue Version jedoch die erste Version innerhalb eines neuen Zweiges, so kann die Sperre sofort zugeteilt werden, ansonsten muß erst die Kompatibilität getestet werden, *bevor* die Version angelegt werden kann, auf der die Sperre dann anschließend zu setzen ist.
Hinzu kommt, daß beim Anlegen einer Sicherungspunkt-Konfiguration von allen bereits gesperrten Objekten bei einem weiteren Schreibzugriff eine neue Version angelegt werden muß. Alle

Sperren müssen dann auf die neue Version übertragen werden, so daß sehr viele Versionen eines Objektes/Links gesperrt worden sein können. Das stellt einerseits einen hohen Verwaltungsaufwand dar und führt zu einer komplexen Logik bei der Sperranforderung, auch durch die unterschiedlicheReihenfolge von Sperranforderung und Anlegen der Version, abhängig davon, ob die Version einen neuen Zweig eröffnet oder nicht.

Bei genauerer Betrachtung erkennt man jedoch, daß das Sperren eines Zweiges sinnvoller ist als das Sperren einzelner Versionen. In jedem Zweig gibt es nur eine Version, die nicht eingefroren ist. Alle anderen Versionen sind nicht mehr änderbar. Das Setzen einer Schreibsperre ist von daher nur an der nicht eingefrorenen Version sinnvoll. Lesesperren können zwar an jeder Versionen gesetzt werden, jedoch haben sie bei eingefrorenen Versionen keine Auswirkung, da diese in jedem Fall nur lesbar und nicht änderbar sind.
Das vereinfacht das Setzen von Sperren deutlich. Es müssen nur noch Sperren pro Zweig verwaltet werden, von daher müssen keine Sperren mehr auf eine neu angelegte Version übertragen werden. Das vereinfacht auch das Zurückziehen der Sperren, da die Übertragung von Sperren nicht mehr protokolliert werden muß. Alle Kompatibilitätstests für neue Sperren sind nur noch auf dem Zweig durchzuführen. Eine weitere Vereinfachung ergibt sich bei der Zuteilung von Lesesperren. Ist die zu sperrende Version nicht in der Arbeitskonfiguration der WTA enthalten[7] –sie ist also eingefroren –, so kann die Lesesperre sofort zugeteilt werden, da keine Schreibsperre existieren kann. Zu beachten ist jedoch der Fall des Setzens einer Lesesperre an einem Versionszweig, für den jedoch noch keine Version existiert. Dieser Fall tritt auf wenn innerhalb eines Konfigurationszweigs das Objekt oder der Link noch nicht verändert wurde. Innerhalb dieses Konfigurationszweigs wird dann auf die Objektversion zugegriffen, die der Konfiguration des Ursprungskonfigurationszweigs zugeordnet ist, an der der neue Zweig eröffnet wurde. Aus diesem Grund müssen die Sperren getrennt von den Versionen verwaltet werden.

Bei Schreibsperren ist zu beachten, daß diese *vor* dem Anlegen der neuen Version anzufordern sind, da es inkompatible Sperren geben kann, die das Zuteilen der Schreibsperre verhindern, und somit die neuen Version (noch) nicht benötigt wird.

Für ro-WTA wird keine eigene Arbeitskonfiguration angelegt, so daß sich die Lesesperren nicht in jedem Fall auf die jüngste Version eines Zweiges beziehen. Hier muß deshalb geprüft werden, ob die ro-WTA auf einer eingefrorenen Konfiguration aufsetzt. In diesem Fall können die Lesesperren sofort zugeteilt werden, andernfalls muß noch die Kompatibilität geprüft werden.
Mit Hilfe der Sperranforderung kann bei den ro-WTA sichergestellt werden, daß alle Änderungen mit einem Fehler abgebrochen werden. Vor jeder Änderung wird in H-PCTE eine Sperre angefordert. Handelt es sich um eine Schreibsperre, so wird die Zuteilung sofort verweigert und ein Fehler zurückgeliefert.

4.1.5 Recovery

Durch die Versionierung muß das Recovery erweitert werden. Einerseits muß bei einem Abbruch einer WTA sichergestellt sein, daß alle durchgeführten Änderungen wieder zurückgenommen werden, unter Berücksichtigung der Version, auf der sie ursprünglich durchgeführt wurden. Andererseits müssen bei einem Systemfehler wieder alle Änderungen von abgeschlossenen WTA bzw. von nicht abgeschlossenen WTA bis zum jeweils jüngsten Sicherungspunkt wiederhergestellt werden. Auch in diesem Fall müssen alle Änderungen auf der richtigen Version durchgeführt werden.
Um das zu erreichen, muß einerseits jeder Wechsel der Arbeitskonfiguration, z. B. durch Anlegen einer Sicherungspunkt-Konfiguration, im Undo-Log der WTA gespeichert sein. Das ermöglicht

[7] Von dem Objekt wurde im Rahmen der WTA noch keine neue Version angelegt.

das Zurücknehmen von Änderungen, ohne bei jeder Änderung die betroffene Version mitspeichern zu müssen, diese läßt sich aufgrund der im Undo-Log gespeicherten K-ID der Arbeitskonfiguration bestimmen. Im Gegensatz hierzu kann der Wechsel der Arbeitskonfiguration nicht im Segment-Log gespeichert werden, da der Log einerseits nicht an eine einzige WTA gebunden ist, andererseits eine WTA auf mehreren Segmenten arbeiten könnte. Für das Wiederherstellen von Änderungen muß daher im Segment-Log bei allen Änderungen die Version, auf die sich eine Änderung bezieht, mit gespeichert werden.

Anlegen von Versionen. Beim Recovery muß man zwei Fälle im Zusammenhang mit dem Anlegen von neuen Versionen berücksichtigen. Durch die Kombination von Sperranforderung und dem Anlegen einer neuen Version, kann der Fall auftreten, daß eine Schreibsperre explizit angefordert wurde, die das Anlegen einer neuen Version zur Folge hat. Im weiteren Verlauf der WTA wird die neue Version jedoch nicht verändert, so daß sie nicht notwendig ist. Diesen Fall kann man erkennen, indem man die aufgrund der expliziten Sperranforderung erzeugte Version kennzeichnet. Wird anschließend schreibend auf diese Version zugegriffen, so entfernt man diese Markierung wieder. Beim Zurückziehen der Sperren nach Abschluß der WTA kann man prüfen, ob die Version modifiziert wurde, falls nicht, kann die Version wieder entfernt werden. Das hat keine Auswirkungen auf das Dokument, da sich diese Version nicht von ihrer Vorgängerversion unterscheidet.

Eine weitere Besonderheit beim Recovery im Zusammenhang mit dem Anlegen von Versionen beruht auf der Tatsache, daß mehrere WTA auf dieselbe Version schreibend zugreifen können. Aufgrund des Schreibzugriffs der ersten WTA wird eine neue Version angelegt, die auch von kooperierenden WTA genutzt werden kann. Wenn alle WTA erfolgreich beendet werden, gibt es kein Problem. Wird jedoch die erste WTA abgebrochen, so würden auch alle durch diese WTA angelegten Versionen wieder entfernt und die Arbeitsgrundlage der kooperierenden WTA entzogen. Dieser Fall muß gesondert berücksichtigt werden.

Die Lösung ist vergleichbar mit der, die für die Konfigurationsobjekte in Abschnitt 4.1.2.2 auf Seite 116 vorgestellt wurde. Der Eintrag in den Undo-Log der WTA stellt nur noch einen Hinweis dar, daß eine neue Version angelegt werden müßte. Dieser Eintrag ist von allen WTA in ihren Undo-Logs vorzunehmen. Jede neu angelegte Version erhält einen Zähler, der angibt, wieviele WTA diese Version benötigen. Dieser Zähler wird inkrementiert, wenn eine kooperierende WTA diese Version auch anlegen würde, sofern sie nicht bereits existieren würde. Wird eine WTA erfolgreich beendet, erfolgt keine Änderung des Zählers. Bei einem Abbruch einer WTA wird der Zähler wieder dekrementiert und anschließend überprüft, ob der Zähler den Wert Null wieder erreicht hat. In diesem Fall kann die Version wieder gelöscht werden, da es keine WTA mehr gibt, die diese Version benötigt.

Dieser Zähler kann auch als Markierung für die Problematik der expliziten Sperranforderung genutzt werden. Bei der expliziten Sperranforderung und der damit verbundenen Erzeugung der neuen Version wird der Zähler mit Null initialisiert. Beim ersten Zugriff und der damit verbundenen impliziten Sperranforderung wird der Zähler inkrementiert. Ist der Zählerstand beim Zurückziehen der Sperre unverändert, also Null, so kann die Version wieder entfernt werden.

4.1.6 Benachrichtigungsmechanismus

Der Benachrichtigungsmechanismus muß, wie die Sperrverwaltung, erweitert werden, um die Versionen berücksichtigen zu können. Es dürfen nur Prozesse über Änderungen informiert wer-

den, die an Versionen durchgeführt wurden, die im Prozeß resp. der ausgeführten WTA[8] sichtbar[9] sind.

Im Gegensatz zur Sperrverwaltung werden keine Versionsinformationen mit den Benachrichtigungswünschen gespeichert. Ob ein Prozeß benachrichtigt werden muß, wird vor der Zuteilung entschieden. Der Grund hierfür ist die deutlich größere Anzahl von Objekten im Vergleich zu H-PCTE-Prozessen. Das bietet den Vorteil, daß nicht an jeder Version Informationen über die zu benachrichtigenden Prozesse gespeichert werden müssen. Bei der Durchführung einer Änderung ist es ausreichend, die zugehörige K-ID der Nachricht beizufügen. Vor der Zustellung kann dann die mitgelieferte K-ID mit der ID der Arbeitskonfiguration der Prozesse abgeglichen werden. Stimmen die Identifizierer überein, so muß die Nachricht dem jeweiligen Prozeß zugestellt werden, andernfalls nicht.
Beim Abgleich der K-ID sind die Zweig-ID und die Konfigurationsnummer innerhalb des Zweiges zu berücksichtigen. Man kann sich nicht auf die Zweig-ID beschränken. Der Grund hierfür sind die ro-WTA. Diese können auf jeder Konfiguration eines Zweiges aufsetzen. Würde man nur die Zweig-ID berücksichtigen, würden auch Prozesse benachrichtigt, für die die geänderte Version nicht sichtbar ist[10].

4.2 Erweiterungen in PI-SET

Durch die gewählte Lösung, die Versionierungsfunktionalität in die Werkzeugtransaktionen zu integrieren, die im Rahmen von Entwurfstransaktionen laufen, ließ sich PI-SET mit minimalen Aufwand erweitern. Nach der Anmeldung an PI-SET muß der Entwickler eine Aufgabe auswählen und bekommt dann eine Liste mit Dokumenten angezeigt, die er bearbeiten kann bzw. zu der er neue Dokumente hinzufügen kann. Die Werkzeuge mit denen er die Dokumente bearbeitet, werden in unversionierten Werkzeugtransaktionen ausgeführt.
Durch die Architektur von PI-SET war es ausreichend, die Aufgaben in PI-SET auf die ETA des Versionierungskonzepts abzubilden und die Schnittstellen zum Starten und Beenden der unversionierten WTA durch die Schnittstellen mit Versionsunterstützung auszutauschen. Diese Erweiterungen erlauben eine rudimentäre Nutzung der Versionierungsfunktionalität (ohne die Dokumentverwaltung durch die ETA), erforderten jedoch keine Änderungen an der graphischen Benutzungsschnittstelle von PI-SET. Zur Auswahl einer bestimmten Dokumentversion war es jedoch notwendig, die Werkzeuge zu erweitern, damit diese alle Konfigurationen einer ETA anzeigen. Diese Anzeige wird dann verwendet, um eine Konfiguration und somit um Dokumentversionen auszuwählen.

Diese Lösung bietet jedoch noch keine Möglichkeit, Änderungskommentare einzugeben. In Abschnitt 3.1 haben wir ein Konzept vorgestellt, wie Änderungskommentare zu jedem beliebigen Zeitpunkt eingegeben werden können. Zu deren Realisierung gibt es verschiedene Möglichkeiten. Der Werkzeugentwickler kann in einem benutzerspezifischem SDS den Objekttyp der Konfiguration wie folgt erweitern:

- um ein Text-Attribut, in dem die Kommentare abgelegt werden. Diese Lösung hat bei der kooperativen Arbeit an einer Konfiguration (siehe Abschnitt 3.3.2) den Nachteil, daß

[8]Die Benachrichtigungen werden dem Prozeß und nicht der WTA zugestellt, da ein Prozeß nicht in jedem Fall eine WTA ausführen muß, aber auch an Änderungen an unversionierten Objekten oder Links interessiert sein kann.

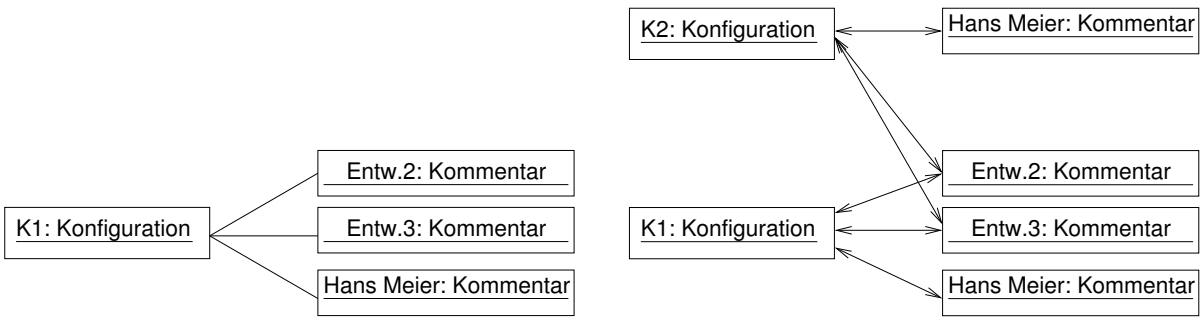
[9]Sichtbar sind außerhalb einer WTA alle unversionierten Objekte und Links bzw. Alle Objekt- und Linkversionen, die der Arbeitskonfiguration einschließlich deren Vorgänger-Konfigurationen der ausgeführten WTA zugeordnet sind.

[10]Für den Fall, daß Prozesse auch über Änderungen an für sie nicht sichtbaren Versionen informiert werden müssen, läßt sich die gewählte Lösung leicht den Erfordernissen anpassen.

das Attribut durch Schreibsperren gegen parallele Zugriffe gesperrt ist und somit nur ein Entwickler einen Kommentar schreiben könnte.

- um einen „Schlaufen-Link“ pro Entwickler mit Kardinalität $1 : n$, der vom Konfigurationsobjekt ausgeht und es selbst wieder als Zielobjekt besitzt. An einem Link-Attribut könnten die Kommentare eines Entwicklers gespeichert werden. Das besitzt den Nachteil, daß beim Anlegen einer Sicherungspunkt-Konfiguration die Kommentare der Entwickler, die den Sicherungspunkt *nicht* angelegt haben, nicht eindeutig einer Konfiguration zugeordnet werden können.

- um ein Kommentar-Objekt für jeden an der kooperativen Arbeit beteiligten Entwickler. Dieses Kommentar-Objekt besitzt ein Text-Attribut, welches die Änderungskommentare eines Entwicklers speichert. Wenn dann ein Sicherungspunkt angelegt wird, kann von jedem beteiligten Konfigurationsobjekt ein Link auf das Kommentar-Objekt angelegt werden. Beispiel: Zu Beginn der Kooperation gibt es eine Konfiguration $K1$, an der drei Entwickler kooperieren (siehe Abbildung 4.4(a)). Der Entwickler „Hans Meier“ legt einen Sicherungspunkt an, so daß alle neuen Versionen der drei Entwickler jetzt der neuen Konfiguration $K2$ zugeordnet werden. Die Änderungskommentare von „Hans Meier“ können der richtigen Konfiguration zugeordnet werden, da er bis zum Anlegen des Sicherungspunktes alle Kommentare eingegeben haben wird. Man kann davon ausgehen, daß alle Kommentare, die er nach dem Anlegen des Sicherungspunktes eingeben wird, sich auf die neuen Änderungen beziehen werden. Davon kann man bei den anderen beiden Entwicklern jedoch nicht ausgehen, da sie i.d.R. keine Kenntnis über den Sicherungspunkt haben. Von daher kann die SEU von den neuen Konfigurationsobjekt einen Link auf das ursprüngliche Kommentarobjekt anlegen, so daß die Änderungskommentare der anderen beiden Entwickler den beiden Konfiguration $K1$ und $K2$ zugeordnet sind (siehe Abbildung 4.4(b)).



(a) vor Anlegen des Sicherungspunktes

(b) nach Anlegen des Sicherungspunktes

Abbildung 4.4: Zusammenhang von Konfigurationsobjekten und Kommentarobjekten

Sinnvoll im Rahmen der kooperativen Entwicklung ist die dritte vorgestellte Möglichkeit zur Speicherung der Änderungskommentare. Um jeden Entwickler über neue Kommentare der kooperativ tätigen Entwickler informieren zu können, ist es möglich, sich über neue Links am Konfigurationsobjekt und Änderungen an den Kommentarattributen der Kommentarobjekte benachrichtigen zu lassen. Als Schlüsselattribute für die Links auf die Kommentarobjekte kann die Benutzer-ID der Entwickler genutzt werden.

4.3 Zusammenfassung

In diesem Kapitel haben wir einige Aspekte diskutiert, die bei der Umsetzung des im vorherigen Kapitels vorgestellten Versionsverwaltungskonzeptes zu berücksichtigen sind. Eine wichtige Erkenntnis ist, daß zur Bearbeitung der selbstreferentiell verwalteten Metadaten nicht die regulären Werkzeugtransaktionen nutzbar sind. Insbesondere durch die kooperative Entwicklung stellen diese Metadaten die Grundlage für mehrere Entwickler dar. Die Konsistenz der Metadaten muß zwar sichergestellt werden, ein Entwickler darf sie jedoch nicht über einen längeren Zeitraum exklusiv sperren.
Eine weitere Besonderheit, die Metadaten und die Versionen der Objekte/Links gemeinsam haben, ist, daß es bei der kooperativen Entwicklung unerheblich ist, welcher Entwickler die Metadaten resp. Versionen anlegte. Man muß jedoch sicherstellen, daß bei einem Abbruch der Tätigkeit eines Entwicklers, die durch ihn angelegten Daten, die von anderen kooperativen Entwicklern genutzt werden, nicht wieder gelöscht werden. Die Einträge in den Logs für das Recovery stellen daher nur Hinweise dar, daß die angelegten Metadaten oder Versionen im Rahmen der Transaktion benötigt werden. Dieser Eintrag ist durch alle Transaktionen vorzunehmen. Nur wenn alle Transaktionen abgebrochen werden, dürfen diese Daten wieder gelöscht werden.
Ein weiterer Aspekt ist das Setzen der Sperren. Hier ist es ausreichend, die Sperren auf einen Versionszweig zu setzen. Die Version als solche muß nicht einzeln gesperrt werden. Der Grund liegt darin, daß im vorgestellten Versionierungskonzept immer nur eine Version eines Zweiges modifizierbar ist.
Bei der Speicherung der Versionen ist einerseits das Verhältnis des Umfangs der Metadaten zu den Nutzdaten zu berücksichtigen, andererseits darf der Aufwand zur Konstruktion einer Version nicht zu groß sein, um die Antwortzeiten interaktiver Werkzeuge nicht zu sehr zu verlängern. Aus dieser Sicht sollte die technische Umsetzung nicht starr an das Konzept gebunden sein. Es kann sinnvoll sein, intern die Objekte als Ganzes zu versionieren, obwohl das Konzept die Versionierung einzelner Bestandteile eines Objektes vorsieht.

Diese Besonderheiten treten insbesondere durch die Möglichkeit auf, kooperativ an einer Version zu arbeiten. Da keines der in Kapitel 2 vorgestellten SKM-Systeme entsprechende Funktionalitäten anbietet, sind dort vergleichbare Konzepte nicht notwendig.

Kapitel 5

Das Differenz- und Mischkonzept für UML-Diagramme

In diesem Kapitel wird die gewählte Lösung zur *Anzeige* von Differenzen und zum Mischen von Versionen *eines* UML-Diagramms beschrieben. Die Anzeige von Differenzen hängt stark vom Dokumenttyp ab. Daher stellen wir zu Beginn in Abschnitt 5.1 ein Konzept zur Anzeige von Differenzen zwischen UML-Diagrammen vor, welches die Unzulänglichkeiten existierender Werkzeuge (siehe Abschnitt 2.2) beheben soll. Anschließend betrachten wir in Abschnitt 5.2 die Details der einzelnen Diagrammtypen, um anhand von diesen Details eine differenziertere Differenzdarstellung zu erhalten. In Abschnitt 5.3 wird dieses Konzept dahingehend weiterentwickelt, daß die Differenzen gruppiert und gruppenweise angezeigt werden können, um die Übersichtlichkeit zu erhöhen. Abschnitt 5.4 erweitert die Differenzanzeige um Funktionen zum Mischen der Versionen. In Abschnitt 5.5 werden die Ergebnisse zusammengefaßt.

5.1 Die Anzeige der Differenzen: Das Vereinigungsdokument

Die Anzeige von Differenzen zwischen zwei Dokumenten, im folgenden als *Basisdokumente* bezeichnet, ist entscheidend von den Eigenschaften des Dokumenttyps abhängig. Diese beeinflussen die verwendete Methode zur Anzeige der Differenzen. Die meisten Differenz(anzeige)-Werkzeuge arbeiten ausschließlich auf Textdokumenten und nicht auf Diagrammen. Die beiden Dokumenttypen unterscheiden sich jedoch in zwei wichtigen Eigenschaften:

- räumliche Ausdehnung der Dokumente
- Auswirkung der Anordnung der Dokumentteile auf die Semantik

Die graphische Darstellung eines Diagramms entspricht der eines Graphen mit Knoten und Kanten[1] und besitzt daher nicht nur eine Dokumentlänge, sondern auch eine Breite, so daß mehrere Diagrammelemente nebeneinander angeordnet sein können. Der Begriff Diagrammelement bezeichnet im weiteren Knoten und Kanten eines Diagramms. Textdokumente besitzen zwar auch eine Breite, jedoch kann man nicht beliebige Zeilen nebeneinander anordnen, ohne die Aussage des Textes zu verändern[2]. Im Gegensatz zu Textdokumenten kann man die Diagrammelemente

[1] In Abschnitt 5.2 betrachten wir die einzelnen Diagrammtypen der UML im Detail.

[2] Nur unter den Annahme, daß man aufeinander folgende Zeilen zu einer neuen Zeile verbindet, kann man zwei Zeilen nebeneinander anordnen, ohne die Aussage zu verändern. Das gilt jedoch nicht für alle Arten von Texten.

wahlfrei im Diagramm verschieben, ohne die Semantik des Dokumentes zu verändern. Daraus folgt, daß die Anordnung der Diagrammelemente, also das Layout der Diagramme, i.d.R. für die Semantik ohne Bedeutung ist[3].

Zur Differenzanzeige von Diagrammen eignet sich die überlagerte Darstellung (siehe Abschnitt 2.2.1) der Basisdokumente. Es ist bei dieser Anzeigetechnik zwar nicht möglich, das Layout beider Basisdokumente beizubehalten. Es ist jedoch möglich, ein Layout zu erzeugen, das dem Layout eines Basisdokuments ähnlich ist (siehe Abschnitt 5.1.1). Das bietet den Vorteil, daß der Anwender des Werkzeugs mindestens eines der Dokumente wiedererkennen kann. Aus diesem Grund verwenden wir diese Methode zur Anzeige von Differenzen für die Diagramme.

Das aus der Differenzberechnung resultierende Diagramm zur Anzeige der Differenzen enthält somit alle Diagrammelemente beider Basisdokumente. Die durch die Differenzberechnung als korrespondierend erkannten Diagrammelemente sind nur einmal enthalten, und die Diagrammelemente, die spezifisch für eines der Basisdiagramme sind, erhalten unterschiedliche farbige Markierungen abhängig vom Basisdiagramm (siehe Abschnitt 5.1.2). Somit stellt das resultierende Diagramm ein *Vereinigungsdokument* der Basisdokumente dar. Das Vereinigungsdokument ist jedoch *kein* Mischdokument, da die Konflikte zwischen den Basisdiagrammen nicht gelöst sind.

Im Vergleich mit den Basisdiagrammen besitzt das Vereinigungsdiagramm eine andere Graphstruktur, die durch die spezifischen Diagrammelemente der einzelnen Basisdiagramme hervorgerufen wird. Diese andere Graphstruktur bedingt daher ein anderes Layout für das Vereinigungsdokument als das der Basisdokumente. Durch die Notwendigkeit, das Layout im Vereinigungsdokument zu verändern, ist es bei diesem Konzept nicht möglich, Differenzen im Layout der Basisdokumente anzuzeigen. Wenn das Differenzanzeige-Werkzeug das leisten soll, kann dieses Konzept nicht verwendet werden.

5.1.1 Layout des Vereinigungsdokumentes

Durch die andere Graphstruktur des Vereinigungsdokumentes kann das Layout nicht vollständig von einem Basisdokument übernommen werden. Das Vereinigungsdokument muß daher ein neues Layout erhalten. Dieses Layout sollte jedoch ähnlich dem Layout eines Basisdokuments sein, um die Wiedererkennung zu vereinfachen.
Diese Randbedingung erleichtert die Erstellung des Layouts, da bereits für Teilmengen des Vereinigungsdokumentes Layoutdaten vorliegen, die berücksichtigt werden können. Im Gegensatz hierzu berechnen Layout-Algorithmen, wie z. B. der Sugiyama-Algorithmus oder dessen Erweiterungen [202] für UML-Diagramme, das Layout für das gesamte Diagramm neu.
Die Auswahl des Basisdokuments, welches als Grundlage für das Layout des Vereinigungsdokumentes verwendet wird, kann entweder durch den Anwender des Differenzanzeige-Werkzeugs vorgegeben oder automatisch anhand der Anzahl der Diagrammelemente bestimmt werden. Das Layout des Vereinigungsdokumentes ist dem Layout des gewählten Basisdokuments um so ähnlicher, je weniger Diagrammelemente neu positioniert werden müssen. Somit ist es sinnvoll, das Basisdokument mit den meisten Diagrammelementen als Layoutvorlage zu verwenden.

5.1.2 Markierung der Unterschiede

Das Vereinigungsdokument besteht aus den gemeinsamen Diagrammelementen und denen, die spezifisch für ein Basisdiagramm sind. Die gemeinsamen Diagrammelemente sind i.d.R. nicht von besonderer Bedeutung für einen Entwickler. Dieser interessiert sich vornehmlich für die

[3]Das gilt nicht für Sequenzdiagramme (siehe Abschnitt 5.2.6.1).

Unterschiede zwischen den Dokumenten, also für die basisdiagrammspezifischen Elemente. Diese müssen somit in geeigneter Art und Weise markiert werden. Für Programm-Quelltexte hat Yang [239] mögliche Methoden untersucht. Er entschied sich für eine farbliche Markierung mit einer geringen Anzahl an Farben, da diese für Werkzeugnutzer leicht zu erfassen ist. Andere Markierungen, wie z. B. unterschiedliche Linientypen/-stärken oder Schriftarten, sind schwerer zu unterscheiden.
Für die Markierung der basisdiagrammspezifischen Elemente nutzen wir u.a. auch aus diesem Grund farbige Markierungen. Ein weiterer Grund, der gegen die Verwendung von unterschiedlichen Linientypen/-stärken und Schriftarten spricht, ist, daß in den UML-Diagrammen unterschiedliche Schriftarten und Linientypen Verwendung finden.

Anzahl der Farben bei zwei Basisdiagrammen. Die Bedeutung und die Anzahl der verwendeten Farben hängt von der Anzahl der zu vergleichenden Diagramme ab. Im einfachsten Fall vergleichen wir nur zwei Diagramme miteinander (2-Wege-Differenzberechnung), können also keine Aussage darüber treffen, welches Element in welchem Diagramm erzeugt oder gelöscht wurde. Das ist i.d.R. nur mit der Existenz eines gemeinsamen Vorgängers möglich oder wenn man alle Änderungen auf eines der beiden Basisdiagramme bezieht. Elemente, die nicht im Bezugsdiagramm vorhanden sind, kann man somit als erzeugt bezeichnen und die Elemente, die nur im Bezugsdiagramm enthalten sind als gelöscht. Für diesen Fall sind drei Farben ausreichend, um das Vereinigungsdiagramm zu zeichnen: eine Farbe für gemeinsame Elemente und je eine Farbe für die spezifischen Elemente.
Eine eindeutige Aussage, ob ein Element erzeugt oder gelöscht wurde, kann man nur dann treffen, wenn beide zu vergleichenden Diagramme zwei Varianten mit einer gemeinsamen Vorgängerversion sind. In diesem Fall kann das Delta anhand der 3-Wege-Differenzberechnung bestimmt werden und somit sind erzeugte und gelöschte Diagrammelemente eindeutig identifizierbar. Im Unterschied zur 2-Wege-Differenzberechnung benötigen wir bereits fünf Farben: eine Farbe für gemeinsame Elemente, je eine Farbe für erzeugte Elemente beider Basisdokumente und je eine Farbe für gelöschte Elemente.

Hier stellt sich die Frage, ob der Informationsgewinn durch die 3-Wege-Differenzberechnung die geringere Übersichtlichkeit durch die höhere Anzahl an Farben aufwiegt. Gegen die 3-Wege-Differenzberechnung spricht, daß üblicherweise die Unterschiede zwischen zwei Dokumenten von Interesse sind, entweder zwischen einer Vorgängerversion und einem Nachfolger oder zwischen zwei Varianten, aber eher selten zwischen zwei Varianten und der gemeinsamen Vorgängerversion. Die Vorgängerversion sollte jedoch beim Mischen von Varianten berücksichtigt werden (siehe Abschnitt 5.4).
Ein weiterer Grund ist die schlechtere Unterscheidbarkeit von fünf Farben gegenüber von drei Farben. Daher verwenden wir im folgenden die 2-Wege-Differenzberechnung und somit maximal drei Farben: die Farbe Schwarz für die gemeinsamen Diagrammelemente und die Farben Rot und Grün für die speziellen Diagrammelemente. Eine Farbe gibt also nur das Enthaltensein in einem Basisdiagramm an und nicht, ob es gelöscht oder erzeugt wurde.

Abbildung 5.1 zeigt ein Beispiel eines Vereinigungsdokumentes unter Verwendung der 2-Wege-Differenzberechnung. Die Darstellung der Differenzen zwischen den zwei Versionen der einzelnen Diagrammtypen wird in Abschnitt 5.2 detailliert diskutiert.

Anzahl der Farben bei mehreren Basisdiagrammen. Wenn mehr als zwei Varianten eines Dokumentes (ohne Vorgängerversion) verglichen werden sollen, steigt die Anzahl der benötigten Farben exponentiell an: $Anzahl = 2^n - 1$. Dies resultiert daher, daß die spezifischen Elemente in einer beliebigen Kombination der Basisdiagramme enthalten sein können. Daher ist die farbige Markierung nur für eine geringe Anzahl zu vergleichender Dokumente

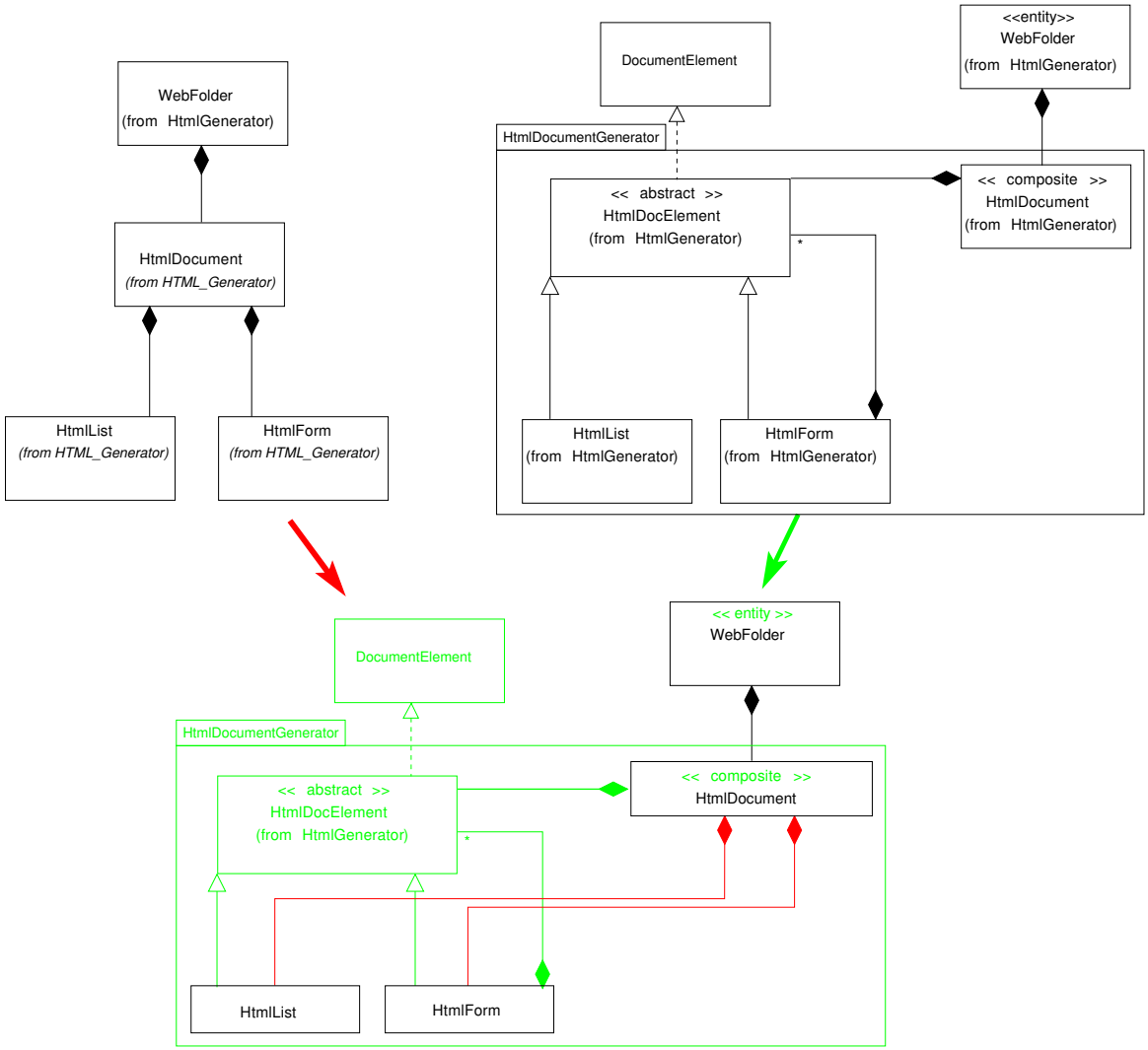


Abbildung 5.1: Beispiel eines Vereinigungsdokuments

sinnvoll. Wenn man mehrere Dokumente vergleichen möchte, muß das Werkzeug Funktionen anbieten, um die Anzahl der benötigten Farben gering zu halten. Die Farben lassen sich mit steigender Anzahl immer schlechter unterscheiden. Eine mögliche Lösung wäre hier, für alle spezifischen Elemente eine Farbe zu verwenden. Die Zugehörigkeit der spezifischen Elemente zu den jeweiligen Basisdiagrammen ließe sich dann mit Werkzeugunterstützung darstellen. Entweder zeigt das Werkzeug eine Liste mit Bezeichnern der Basisdiagramme pro spezifisches Element an (z. B. in einem Tooltip) oder das Werkzeug bietet die Möglichkeit, alle spezifischen Elemente von bestimmten Basisdiagrammen auszublenden. Wenn die Anzahl der anzuzeigenden Basisdiagramme auf zwei reduziert wurde, können die Differenzen 3-farbig dargestellt werden. Inwieweit diese beiden Ansätze für eine große Anzahl von Basisdiagrammen benutzbar ist, soll hier nicht evaluiert werden.

5.2 Differenzen zwischen UML-Diagrammen

Die Vorschläge zur Differenzberechnung zwischen UML-Modellen [140, 205, 5] berechnen ausschließlich die Differenzen zwischen zwei Editiermodellen. Die Anzeige der Differenzen bleibt unberücksichtigt. In diesem Abschnitt wollen wir die Grundlage erarbeiten, um die Differenzen zwischen einzelnen Versionen eines Diagramms detailliert anzeigen und später auch mischen

zu können. Von Interesse sind hier nur die Differenzen im Modell und nicht die Änderungen am Layout. Desweiteren konzentrieren wir uns nur auf die Arten von Differenzen, die zwischen zwei Versionen eines Diagrammtyps auftreten können, sowie auf deren Darstellung. Die Differenzberechnung betrachten wir in diesem Abschnitt nicht. Das holen wir in Abschnitt 6.2.2 nach.

Um eine detaillierte Darstellung realisieren zu können, ist eine genaue Betrachtung der einzelnen Diagrammelemente, deren Repräsentation im Metamodell der UML sowie der Darstellung im Diagramm notwendig. Zur klaren Unterscheidung zwischen den Elementen im Diagramm und denen im Modell sprechen wir von Diagrammelementen, wenn wir die im Diagramm visuell dargestellten Elemente meinen, und von Modellobjekten, wenn wir Objekte des Editiermodells meinen.

Die Detaillierung der Differenzdarstellung ist abhängig vom verwendeten Editier-Metamodell und den darauf definierten Operationen. Hier setzen wir ein feinkörniges Editier-Metamodell (siehe Abschnitt 1.2.2.1) voraus, welches mit dem in der UML-Spezifikation [181] vorgestellten Metamodell vergleichbar ist.

5.2.1 Klassendiagramme

In diesem Abschnitt wollen wir uns als erstes die Eigenschaften eines Klassendiagramms verdeutlichen, um dann anschließend daraus die möglichen Differenzen zwischen zwei Klassendiagrammen ableiten zu können.

Eigenschaften von Klassendiagrammen. Die graphische Darstellung eines Klassendiagramms entspricht der eines gerichteten Graphen, mit getypten Knoten und Kanten. Die Kanten bezeichnen wir im folgenden als Beziehungen. Abbildung 5.2 zeigt beispielhaft ein Klassendiagramm. Die üblicherweise verwendeten Knotentypen sind: Klassen, Pakete, Interfaces, Objekte und Bemerkungen.

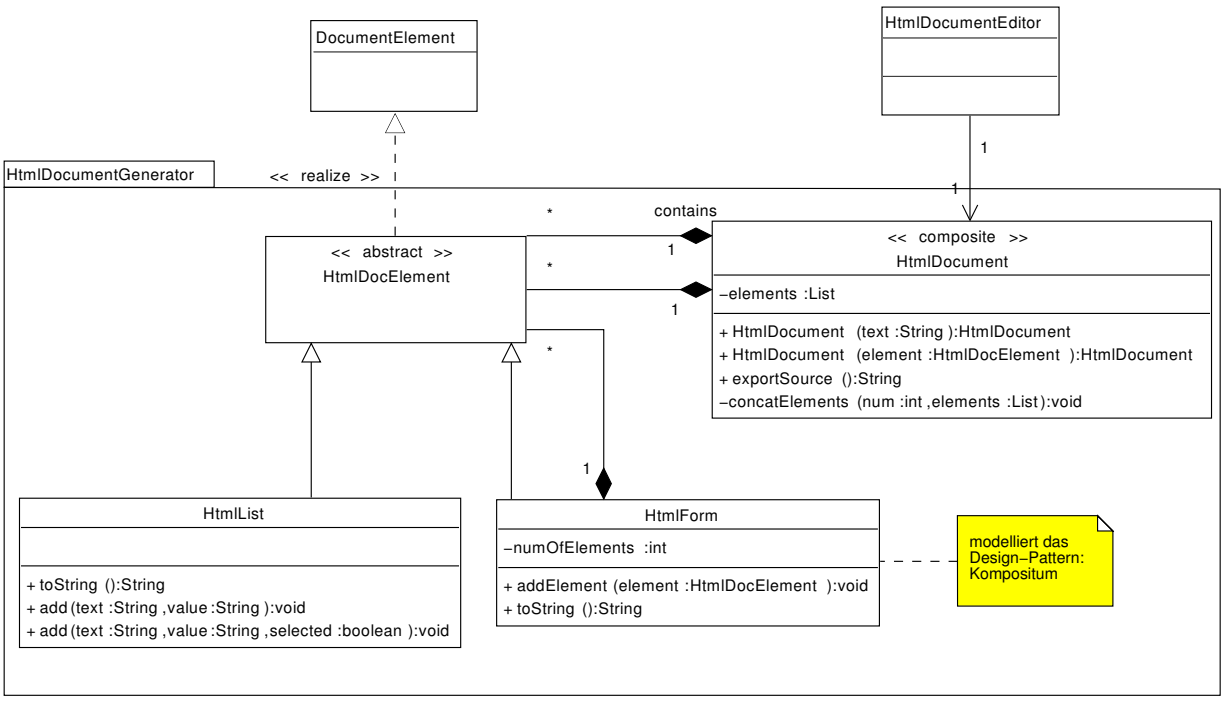


Abbildung 5.2: Beispiel eines Klassendiagramms

Die Knotentypen lassen sich unter Berücksichtigung der graphischen Darstellung und des Metamodells [181, Seiten: 2-13, 2-16] in drei Gruppen einteilen:

1. Knoten mit textuell dargestellten Komponenten: Klassen, Interfaces und Objekte
2. Knoten mit Komponenten, die graphisch dargestellt werden: Pakete
3. Knoten nur mit textuellen Attributen: Bemerkung

Der Begriff Komponente leitet sich hierbei direkt aus dem Metamodell ab: Die Diagrammelemente, die im Metamodell über eine Kompositionsbeziehung mit einem übergeordneten Element in Beziehung stehen, bezeichnen wir als Komponenten. Beispiele für Komponenten sind Methoden und Attribute von Klassen, die Methoden von Interfaces oder die Attribute von Objekten. Die Methoden enthalten wiederum selbst Komponenten: die Parameter. Bei Differenzen zwischen zwei Klassen kann es sich um Differenzen zwischen den Klassen selbst oder um Differenzen zwischen den Komponenten der Klassen handeln. Das muß bei der Darstellung berücksichtigt werden.
Die Komponenten von Paketen können im Paketsymbol oder als eigenständiges Klassendiagramm dargestellt werden, so daß für diese eine gesonderte Betrachtung nicht notwendig ist. Zu berücksichtigen ist jedoch, daß Beziehungen zwischen Knoten außerhalb eines Pakets und Knoten innerhalb eines Pakets existieren können.
Modellobjekte mit Komponenten stellen für ihre Komponenten den Namensraum ([181]: Namespace) dar, der auch an den Knoten notiert werden kann.
Neben den Komponenten enthalten die Knoten auch atomare Attribute, die den Knoten selbst näher beschreiben (z. B. Bezeichner und Sichtbarkeiten) sowie Referenzen auf andere Modellobjekte (z. B. Typ oder Stereotyp). Der Unterschied zwischen einer Komponente und einem referenzierten Modellobjekt ist, daß letztere im Metamodell durch eine Assoziation und nicht über eine Kompositionsbeziehung mit dem korrespondierenden Objekt in Verbindung steht. Somit ist das referenzierte Objekt eigenständig.

Ein Knoten kann mehrere Komponenten oder Modellobjekte gleichen Typs referenzieren. Diese können als Menge oder als Liste[4] im Metamodell realisiert sein. Das Metamodell definiert auch Kardinalitäten zwischen den Modellobjekten, so daß ein Knoten nicht beliebig viele Referenzen oder Komponenten besitzen darf. Einen Überblick[5] gibt Tabelle 5.1. Darin sind neben den zu einem Diagrammelement zugehörigen atomaren Attributen, Komponenten und Referenzen auch die Typen der Attribute, die Kardinalitäten sowie die Verwaltungsart (Menge oder Liste) der Referenzen und Komponenten angegeben.
Neben den Knoten gibt es in Klassendiagrammen auch Typen von Beziehungen: Assoziationen, Vererbungsbeziehungen, Implementierungsbeziehungen, Beziehungen zwischen einer Bemerkung und einem anderen Diagrammelement. Die Beziehungen können gemäß UML-Spezifikation in Abhängigkeit vom Beziehungstyp zwei oder z.T. auch mehr Knoten im Diagramm verbinden. Die Anzahl der verbundenen Diagrammelemente ist abhängig vom Beziehungstyp und vom verwendeten Werkzeug. Die Beziehungstypen unterscheiden sich einerseits durch die graphische Darstellung, aber auch durch ergänzende Angaben wie z. B. Kardinalitäten. Die Beziehungen sind lt. UML-Spezifikation als eigenständige Modellobjekte realisiert. Eine Besonderheit stellen im Metamodell der UML die Assoziationen dar, die neben atomaren Attributen (z. B. der Bezeichner der Beziehung) auch Komponenten, und zwar die Assoziationsendpunkte, besitzen, die die Anbindung an die beteiligten Diagrammknoten darstellen. Tabelle 5.2 hierüber gibt einen Überblick.

[4]Unter Liste verstehen wir hier eine Menge mit einer Ordnungsrelation auf den Elementen.
[5]Die Attribute, Komponenten und Referenzen der Diagrammelemente sind nur exemplarisch angegeben.

Diagrammelement	atomare Attribute	Komponenten	Referenzen
Klasse	**Klassenname:** *Text* **Sichtbarkeit:** *Enum*: public, protected, package, private **Modifizierer:** *Enum*: abstract, final, root, active	**Methoden:** *Liste* **Attribute:** *Liste*	**Stereotyp:** *Menge*
Interface	**Bezeichner:** *Text*	**Methoden:** *Liste*	**Stereotyp:** *Menge*
Objekt	**Bezeichner:** *Text*	**Attribute:** *Liste, ggf. mit Werten*	**Stereotyp:** *Menge*
Paket	**Bezeichner:** *Text* **Modifizierer**	**graph Darstellung:** *Klassen, Interfaces, Objekte, Pakete*	**Stereotyp:** *Menge*
Methode	**Bezeichner:** *Text* **Sichtbarkeit:** *Enum* **Modifizierer:** *Enum:* abstract, final, root, query, static	**Parameter:** *Liste*	**Stereotyp:** *Menge* **Rückgabetyp:** *1*
Parameter	**Bezeichner:** *Text*		**Typ:** *1* **Stereotyp:** *Menge*
Attribut	**Bezeichner:** *Text* **Sichtbarkeit:** *Enum* **Modifizierer:** *Enum*: static, final, transient, volatile **Wert:** *Text*		**Typ:** *1* **Stereotyp:** *Menge*
Bemerkung	**Text:** *Text*		

Tabelle 5.1: Eigenschaften der Knoten-Elemente eines Klassendiagramms

Zusammenfassend lassen sich die Diagrammelemente eines Klassendiagramms folgendermassen einteilen:

- Knoten mit
  - reinen Textattributen
  - textuell dargestellten Komponenten
  - graphisch dargestellten Komponenten
- Beziehungen zwischen den Knoten, teilweise mit textuellen Annotationen
- Listen von Diagrammelementen
- Mengen von Diagrammelementen

Diagrammelement	atomare Attribute	Komponenten	Referenzen
Assoziation	**Name:** *Text* **Modifizierer:** *Enum*: abstract, final, root	**Assoziationsendpunkt:** *Liste*	**Stereotyp:** *Menge*
Assoziations-endpunkt	**Name:** *Text* **Aggregation:** *Enum*: none, Aggregation, Komposition **Modifizierer:** *Enum*: abstract, final, root **Gerichtet:** *Boolean* **Kardinalität:** *Enum* **Sichtbarkeit:** *Enum*		**Klasse:** *1* **Objekt:** *1* **Stereotyp:** *Menge*
Vererbung	**Name:** *Text* **Deskriminator:** *Text*		**Klassen:** *2* **Stereotyp:** *Menge*
Implementierung	**Name:** *Text*		**Klasse:** *1* **Interface:** *1* **Stereotyp:** *Menge*
Beziehung an Bemerkung			**Diagrammel.:** *1* **Bemerkung:** *1*

Tabelle 5.2: Eigenschaften der Beziehungstypen eines Klassendiagramms

Differenzen zwischen Klassendiagrammen. Aus den oben genannten Überlegungen lassen sich jetzt die möglichen Änderungen und damit auch die möglichen Differenzen zwischen Klassendiagrammen ableiten. Die grundlegende Art von Differenzen sind:

- neue/gelöschte Knoten: z. B. Klassen, Interfaces oder Pakete
- neue/gelöschte Beziehungen: z. B. Assoziationen, Vererbungsbeziehungen
- Intra-Knoten- und Intra-Beziehungsänderungen. Hierzu zählen Wertänderungen von atomaren Modell-Attributen. Beispiele sind:
  - reine Textattribute, z. B. der Klassenname
  - Aufzählungstypen, z. B. die Sichtbarkeit von Methoden, Attributen, usw.
  - Boolesche Werte

  Mögliche Änderungen sind das Umbenennen einer Klasse, einer Rolle usw., Ändern von Kardinalitäten an Beziehungen, aber auch das Ändern des Typs einer Assoziation (Assoziation, Aggregation und Komposition), sofern das Metamodell dem der UML-Spezifikation entspricht.

- Differenzen zwischen zwei Varianten einer Bemerkung beschränken sich auf geänderten Inhalt, also Differenzen zwischen zwei (kurzen) Texten. Hierfür lassen sich die bekannten Techniken zur Anzeige von Differenzen zwischen Texten verwenden. Eine gesonderte Betrachtung ist somit nicht notwendig.

Weitere Intra-Knoten-/Intra-Beziehungsänderungen sind: Umordnen von Listen, Änderungen an oder von referenzierten Modell-Objekten. Dazu zählen Umsetzen der Referenz auf ein anderes Modell-Objekt (z. B. Auswahl eines anderen Typs für einen Parameter), aber auch Ändern des im Diagrammelement angezeigten atomaren Modell-Attributs des referenzierten Modell-Objektes (z. B. Ändern des Bezeichners eines Stereotyps). Es gibt eine Besonderheit für bestimmte Knotentypen: Das Ändern des Stereotyps kann gemäß UML-Spezifikation zu einer anderen graphischen Darstellung führen. Ein Beispiel hierfür sind Klassen: Das Klassensymbol kann durch verschiedene Icons ersetzt werden, je nach verwendetem Stereotyp.
Desweiteren gibt es noch Inter-Knotenänderungen. Hier sind insbesondere Verschiebungen von Komponenten zu nennen, wie z. B. verschieben einer Methode von einer Klasse zu einer anderen Klasse oder verschieben von Klassen zwischen unterschiedlichen Paketen. Beziehungen, genauer gesagt die Endpunkte von Beziehungen, können auch zwischen Diagrammknoten verschoben werden, so z. B. Assoziationsendpunkte zwischen Klassen.
Folgende Übersicht faßt die Änderungen zusammen:

1. Struktur-Änderungen
   (a) neue/gelöschte Knoten
   (b) neue/gelöschte Beziehungen
2. Intra-Knoten/Beziehungsänderungen
   (a) atomare Wertänderungen: Texte, Aufzähltyp oder Boolescher Wert
   (b) neue/gelöschte Elemente in Listen und Mengen
   (c) Umordnen einer Liste
   (d) Änderungen an komplexen graphischen Elementen
   (e) referenzierte externe Objekte
3. Inter-Knoten/Beziehungsänderungen: (werkzeugabhängig)
   (a) Verschieben von Komponenten zwischen Knoten
   (b) Verschieben von Beziehungen

Darstellung der Differenzen. Die Darstellung der Differenzen zwischen Diagrammen ist abhängig vom verwendeten Editortyp zur Differenzdarstellung. Bisher haben wir in diesem Kapitel Editoren unterstellt, die die Diagramme als Graph anzeigen. Jedoch lassen sich, abhängig vom Editor, in der Graph-Repräsentation eines UML-Diagramms nicht alle Daten eingeben oder anzeigen, so daß auch noch formular-orientierte Editoren für bestimmte Diagrammelemente, wie z. B. Klassen notwendig sind. Zu Beginn diskutieren wir die Darstellung von Differenzen in graphischen und anschließend in formular-orientierten Editoren.

Graphische Editoren. Die Darstellung von erzeugten/gelöschten Knoten oder Beziehungen ist einfach möglich, indem der Knoten oder die Beziehung vollständig mit allen Komponenten und Attributen eingefärbt wird. Bei Beziehungen gilt dies unabhängig davon, ob sie zwischen zwei Knoten des Diagramms oder zwischen zwei graphischen Komponenten von Knoten besteht (z. B. zwischen einer Klasse in einem Paket und einer Klasse in einem anderen Paket). Ein Beispiel hierfür wurde bereits in Abbildung 5.1 auf Seite 132 gezeigt.
Neben der Differenzdarstellung ist bei erzeugten/gelöschten Knoten und Beziehungen noch das Layout zu berücksichtigen. Bei Klassendiagrammen sollten Vererbungsbeziehungen vertikal und Assoziationen horizontal verlaufen, die Klassensymbole sollten dementsprechend angeordnet werden, wobei die Länge der einzelnen Beziehungen so kurz wie möglich sein sollte.

Die Darstellung von Intra-Knoten/Intra-Beziehungsänderungen wollen wir aufgrund der vielfältigen Arten von Differenzen näher betrachten. Die einfachste Differenz ist ein geändertes atomares Modell-Attribut mit rein textuellem Inhalt (z. B. der Klassenname). In diesem Fall kann man beide Werte nebeneinander unterschiedlich gefärbt darstellen. Bei genauer Betrachtung der graphischen Darstellung eines Diagramms gibt es neben den atomaren textuellen Modell-Attributen noch weitere Diagrammelemente, die im Metamodell zwar durch eigenständige Modellobjekte (Komponenten oder referenzierte Modellobjekte) realisiert sind, im Diagramm jedoch nur als Text (i.d.R. der Bezeichner) dargestellt werden, wie z. B. Methoden, Attribute, Parameter oder auch Stereotypen. Änderungen der Bezeichner dieser Modellobjekte sowie der Austausch des referenzierten Modellobjekts zeigen sich nur durch einen geänderten Text im Diagramm. Daher können Differenzen so dargestellt werden wie die Differenzen bei atomaren Textattributen. Diese Darstellungsform ist für jeden Bezeichner separat möglich, wie in Abbildung 5.3 dargestellt.
Für den Fall, daß das atomare Modell-Attribut ein Aufzählungstyp ist oder rein Boolesche Werte annehmen kann, hängt die Methode der Darstellung der Differenzen von deren Repräsentation im Diagramm ab. Bei rein textueller Darstellung (z. B. Kardinalität) oder bei der Darstellung durch einzelne Symbole (z. B. Sichtbarkeit von Methoden und Attributen) können beide Versionen ebenfalls unterschiedlich gefärbt nebeneinander dargestellt werden. Wird der Wert jedoch z. B. durch eine andere Schriftart wie bei abstrakten Klassen oder durch Unterstreichung des Bezeichners dargestellt, muß man auf eine textuelle Repräsentation ausweichen. Für das Beispiel der abstrakten Klasse kann man das Schlüsselwort `abstract` gefärbt im Klassensymbol angeben, anstatt Kursivschrift zu verwenden.

Komplexere Intra-Knotenänderungen betreffen die als Listen oder Mengen dargestellten Komponenten der Diagrammelemente. Mengen mit unterschiedlichen Elementen lassen sich einfach darstellen, indem die unterschiedlichen Elemente an einer beliebigen Position eingefärbt dargestellt werden. Ein Beispiel ist die Menge der Stereotypen eines Diagrammelementes. Bei den Stereotypen einer Klasse ist noch zu beachten, daß das Klassensymbol in Abhängigkeit von den gewählten Stereotypen auch als Icon dargestellt werden kann, falls das eingesetzte Werkzeug dies unterstützt. In diesem Fall sollte die Darstellung der Stereotypen als Menge ohne Icon-Darstellung der Klasse bevorzugt werden, um die Übersichtlichkeit zu verbessern.
Listen (z. B. Attributliste oder Parameterliste) unterscheiden sich von Mengen darin, daß auf allen Elementen der Liste eine Ordnung definiert ist, die die Position der Elemente in der Liste bestimmt. Die Ordnung ist entweder implizit durch ein Werkzeug bzw. eine gegebene Ordnungsrelation oder explizit durch den Werkzeuganwender vorgegeben. Der erste Fall stellt für die Differenzanzeige kein Problem dar. Die Elemente der Liste im Vereinigungsdiagramm werden anhand der impliziten Ordnung in der Liste angeordnet und die basisdiagrammspezifischen Elemente werden markiert.
Den zweiten Fall müssen wir separat betrachten. Hier ist die Listenposition der basisdiagrammspezifischen Elemente zu berücksichtigen. Die Listenposition ist für die Liste im Vereinigungs-

diagramm keine absolute Angabe, sondern eine relative Angabe, die als Bezugsgröße die korrespondierenden Listenelemente beider Basisdiagramme vor und nach dem betrachteten Elemente besitzt.
Gibt es mehrere basisdiagrammspezifische Listenelemente an unterschiedlichen Positionen, können diese in die Liste des Vereinigungsdiagramms ohne Veränderung der Position übernommen und markiert werden.
Neben unterschiedlichen Elementen in Listen können die Elemente in der Liste explizit umsortiert worden sein, so daß sie an unterschiedlichen Positionen stehen. Hier gibt es die Möglichkeit, die Liste in beiden Sortierungen nebeneinander darzustellen, jedoch ist es so nicht einfach möglich, die Listenposition eines Elements in beiden Versionen der Liste zu erkennen. Eine andere Methode ist, die Liste in einer Sortierung anzuzeigen und die Listenposition eines Elements in der passenden Farbe vor den Listeneintrag zu schreiben (siehe Abbildung 5.3: Methodenliste und Parameterliste von der Methode `concatElements`). Bei letzterer Methode kann man die Listenpositionen der Elemente leichter erkennen.
Eine Kombination der beiden oben genannten Arten von Differenzen in Listen ist auch möglich. So kann ein Entwickler eine Liste umsortiert haben, um anschließend neue Elemente hinzuzufügen und andere Elemente zu löschen. In diesem Fall ist die Differenzdarstellung der Liste eine Kombination aus den beiden genannten Darstellungsformen. Die einzige zu klärende Frage ist, an welchen Positionen die neuen Elemente in der Liste im Vereinigungsdokument eingetragen werden. Sind die Elemente nur in der Liste des Basisdokuments enthalten, dessen Sortierung verwendet wird, so können die Elemente an ihren Original-Positionen eingetragen werden. Gibt es die Elemente nur in der Liste des Basisdokuments, dessen Sortierung in der Differenzdarstellung nicht verwendet wird, so kann man diese Elemente zusammen an den Anfang oder an das Ende der Liste setzen oder auch hinter das Element, hinter dem sie auch in der Basisversion eingetragen sind. Ist diese Position durch eine neues Element der anderen Basisversion belegt, so müssen diese Elemente nacheinander im Vereinigungsdokument dargestellt werden.

Die wesentlichen Änderungen an komplexen graphischen Komponenten sind erzeugte oder gelöschte Komponenten. Da der Graph der Komponenten als ein Subdiagramm interpretierbar ist, gibt es keinen Unterschied zur Darstellung von erzeugten/gelöschten Diagrammelementen des umgebenden Diagramms, so daß auch dieselben Darstellungsformen verwendet werden können.

Eine weitere Änderungsart sind Interknoten-Änderungen, wie z. B. das Verschieben von Methoden zwischen Klassen oder das Verschieben von Klassen aus/in Pakete. Um diese Differenzen darstellen zu können, muß ein (komplexes) Modellobjekt zweimal im Diagramm gezeichnet werden, einmal an der alten und einmal an der neuen Position, jeweils in der Farbe des entsprechenden Basisdiagramms. Da die Verschiebung so nicht erkennbar ist, sollte sie durch ein zusätzliches Symbol gekennzeichnet sein. Falls mehrere Diagrammelemente verschoben und die Bezeichner der Elemente verändert wurden, ist eine Zuordnung der Paare der verschobenen Element nur schwer möglich. Daher sollte ein Differenzanzeige-Werkzeug Funktionen anbieten, um die Paare der verschobenen Elemente bei Bedarf hervorzuheben, indem ein Element durch den Anwender markiert und das korrespondierende Element durch das Werkzeug blinkend dargestellt wird.
Eine weitere Form von Interknoten-Änderungen ist das Verschieben von Beziehungsendpunkten. In diesem Fall müßte die Beziehung dreigeteilt dargestellt werden: in einen gemeinsamen Teil, der schwarz gefärbt ist und zwei spezifische Teile, die dann eingefärbt gezeichnet werden. Das reduziert jedoch die Übersichtlichkeit und ist auch kompliziert zu realisieren, so daß man die Beziehung besser zweimal darstellt.

Unabhängig von der Art der Verschiebung sinkt mit deren Anzahl die Übersichtlichkeit des Vereinigungsdiagramms, da die verschobenen Diagrammelemente doppelt gezeichnet wer-

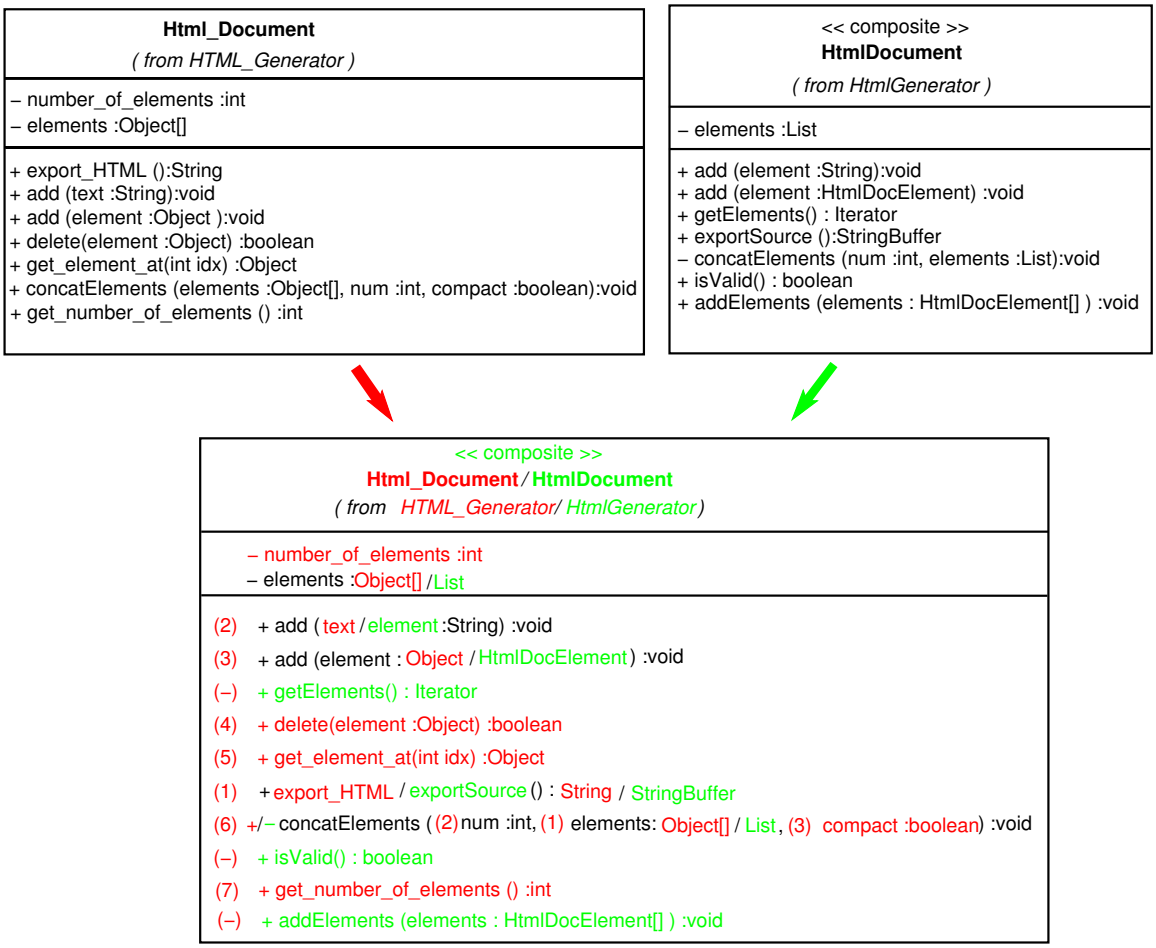


Abbildung 5.3: Beispiel der Differenzanzeige einer Klasse

den. Verschobene Knoten reduzieren die Übersichtlichkeit mehr als verschobene Kanten, da auch die Kanten zu den verschobenen Knoten eingezeichnet werden müssen und somit auch doppelt dargestellt werden. Von daher ist diese Art der Differenzdarstellung für verschobene Diagrammelemente nur praktikabel, wenn nur eine geringe Anzahl verschoben wurde.

Formular-orientierte Editoren. Formular-orientierte Editoren verwendet man i.a., um die Eigenschaften einzelner Elemente eines graphischen Editors zu bearbeiten, die in der graphischen Darstellung nicht oder nur schlecht änderbar sind. Bei diesen Eigenschaften handelt es sich i.d.R. um Attribute eines Modellobjekts. Abhängig vom Editor zeigt dieser noch eine Liste/Baumstruktur der Komponenten des zu bearbeitenden Modellobjekts. Die vorkommenden Differenzen beschränken sich somit auf geänderte atomare Attribute des Modellobjekts oder geänderte Listeneinträge und Reihenfolgen. Ein Beispiel ist die Auswahl eines Modifiers oder der Sichtbarkeit von Methoden. Der Aufbau der Bedienoberfläche eines formular-orientierten Editors ist i.d.R. einfacher als der eines graphischen Editors. Formular-orientierte Editoren bestehen im wesentlichen aus Texteingabefeldern oder aus Listen von Texten. Desweiteren können sie auch eine Baumdarstellung von Komponenten beinhalten.
Differenzen zwischen zwei kurzen Texten oder zwischen zwei Listen können wie Intra-Knotenänderungen behandelt werden. Die Darstellung von Änderungen einer Baumstruktur kann vergleichbar mit der Lösung im ModelIntegrator gehandhabt werden. Die Umsortierung von Kindknoten eines Vaterknotens kann so gehandhabt werden wie bei Listen. Problematisch sind Umsortierungen, die mehrere Knoten betreffen. Eine mögliche Lösung ist, die Knoten

gefärbt an beiden Positionen darzustellen, wobei ein weiteres Symbol kennzeichnet, daß die Knoten verschoben wurden.

5.2.2 Objektdiagramme

Objektdiagramme zeigen Instanzen von Klassen. Da Klassendiagramme auch Objekte beinhalten können und gemäß der UML-Spezifikation dann als Objektdiagramme bezeichnet werden, sofern sie ausschließlich Objekte enthalten, ist eine gesonderte Betrachtung nicht notwendig. Alle Aussagen zu Klassendiagrammen gelten somit auch für Objektdiagramme.

5.2.3 Anwendungsfalldiagramme

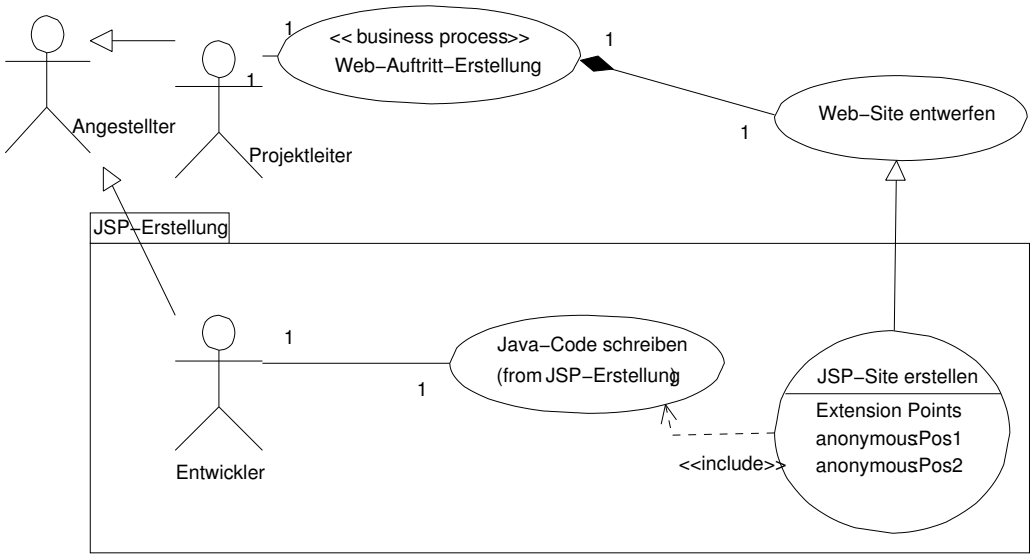


Abbildung 5.4: Beispiel eines Anwendungsfalldiagramms

Die wesentlichen Diagrammelemente in Anwendungsfalldiagrammen (ein Beispiel ist in Abbildung 5.4 dargestellt) sind: Akteure, Anwendungsfälle, Pakete sowie mehrere Beziehungstypen. Die Akteure können gemäß ihrer Spezifikation auch als Klassen dargestellt werden, somit gilt für sie das bereits für Klassen Gesagte. Die Pakete beinhalten andere Knotentypen als in den Klassendiagrammen, was deren Eigenschaften jedoch nicht ändert, so daß auch für diese keine weitere Betrachtung notwendig ist.
Die Anwendungsfälle können eine Menge von Extension-Points beinhalten, deren Darstellung sich nicht wesentlich von der Darstellung der Menge von Stereotypen unterscheidet.
Neben den bereits bekannten Beziehungstypen gibt es in Anwendungsfalldiagrammen noch Include- und Extend-Beziehungen. Erstere verbinden zwei Anwendungsfälle. Gemäß der UML-Spezifikation [181] sind sie als eigenständige Modellobjekte realisiert, die direkt auf die beteiligten Modellobjekte der Anwendungsfälle verweisen. Die Extend-Beziehungen verbinden nicht nur zwei Anwendungsfälle, wie die Include-Beziehungen, sondern auch noch eine Liste von Extension-Points, die jedoch nicht in der graphischen Darstellung des Diagramms angezeigt wird. Die Zuordnung von Extension-Points zu einer Extend-Beziehung läßt sich nur in einem eigenständigen formular-orientierten Editor unter Verwendung einer Liste vornehmen.
In Tabelle 5.3 ist eine Übersicht der Diagrammelemente der Anwendungsfalldiagramme dargestellt. Diese Analyse zeigt, daß diese keine weiteren Eigenschaften besitzen als die bisher

Diagrammelement	atomare Attribute	Komponenten	Referenzen
Anwendungsfall (Usecase)	**Name:** *Text* **Modifizierer:** *Enum*: abstract, final, root	**Methoden:** *Liste* **Attribute:** *Liste*	**ExtensionPoint:** *Menge* **Stereotyp:** *Menge*
ExtensionPoint	**Name:** *Text* **Location:** *Text*		**Extend-Beziehung:** *Menge*
Actor	s. Klasse	s. Klasse	s. Klasse
Extend-Beziehung	**Bedingung:** *Boolescher Ausdruck*		**Extension-Point:** *Liste*

Tabelle 5.3: Eigenschaften der Diagrammelemente eines Anwendungsfalldiagramms

bekannten Eigenschaften (siehe Abschnitt 5.2.1). Für die Differenzanzeige gibt es daher keine Besonderheiten zu berücksichtigen.

5.2.4 Zustandsdiagramme

Zustandsdiagramme stellen im Gegensatz zu den Klassendiagrammen keine statische Struktur eines Modells dar, sondern sie modellieren das dynamische Verhalten. Daher enthalten sie andere Diagrammelement-Typen, siehe Abbildung 5.5. Diese beschränken sich im wesentlichen auf Zustände und deren Subtypen sowie die Transitionen als einzigen Beziehungstyp. Die Transitionen verbinden zwei Zustände beliebigen Typs miteinander und besitzen Komponenten, die das Verhalten beim Zustandsübergang genauer definieren. Die Komponenten werden als Texte an der Transition notiert. Die Differenzdarstellung muß daher nicht weiter diskutiert werden.

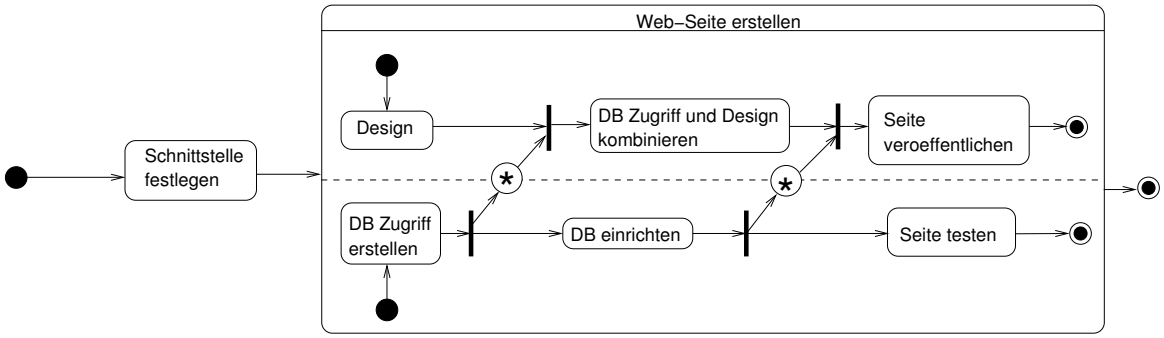


Abbildung 5.5: Beispiel eines Zustandsdiagramms

Der am einfachsten strukturierte Subtyp eines Zustands ist der *Pseudo-State*. Er hat eine einfache graphische Struktur (Kreis oder Linie), die vom Wert des Metamodell-Attributs `Kind` bestimmt wird. Dieses Attribut gibt die Art (junction, fork, deep history, initial, choice, join, shallow history) des Pseudo-States an. Neben der Änderung des Bezeichners könnte je nach Realisierung des Werkzeugs auch das Attribut `Kind` verändert werden und damit das Aussehen. Da mit der Änderung dieses Metamodell-Attributs auch die Semantik des Pseudo-States verändert wird, stellt sich die Frage, ob es überhaupt sinnvoll ist, diese Änderung in einem Werkzeug zu erlauben. Das Ändern eines Pseudo-States ist in jedem Fall durch Löschen und Neuanlegen möglich. Daher betrachten wir die Typänderung nicht weiter.

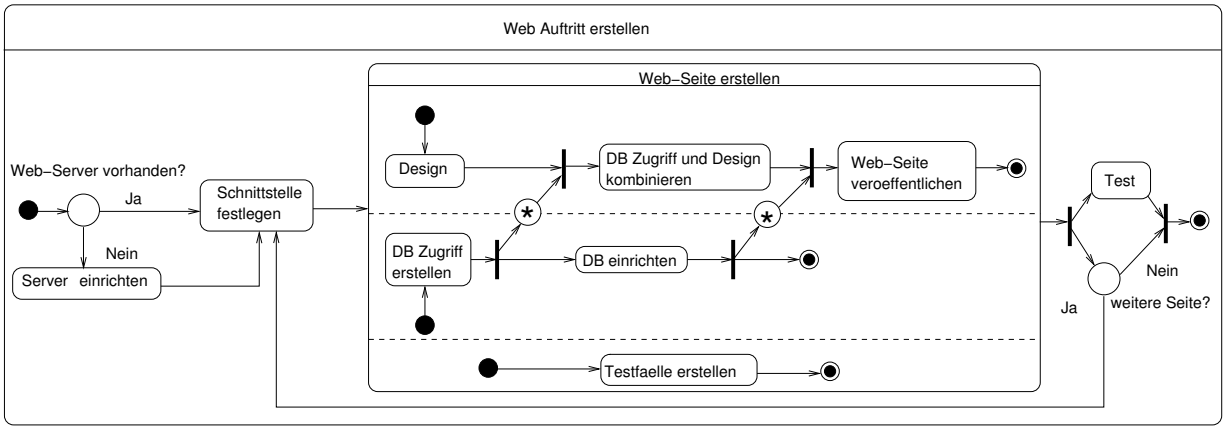
Neben den Pseudo-States gibt es weitere einfach strukturierte Knotentypen, wie den *Final-State* oder die *Synch-States*. Diese Knotentypen besitzen einen Bezeichner und eine Stereotyp-Menge. Komplexere Knotentypen sind die Zustände, diese können drei Komponenten besitzen: *Entry-, Exit-, Do-Activity*. Sie spezifizieren die beim Eintritt, beim Verlassen oder beim Verbleiben im Zustand ausgeführten Aktivitäten und werden im Knotensymbol durch Texte dargestellt. Desweiteren können sie eine Menge von internen Zustandsübergängen enthalten, die ebenfalls als eine Menge von Texten im Knotensymbol dargestellt werden. Diese Knotentypen besitzen hinsichtlich der Differenz-Darstellung keine nicht bereits bekannten Eigenschaften, somit erübrigt sich eine weitergehende Diskussion.
Composite States (z. B. Zustand `Web Auftritt erstellen` in Abbildung 5.6(a)) können eine Menge von Sub-Zustandsdiagrammen beinhalten, die jeweils als *Region* bezeichnet werden. Enthält der Composite State mehr als eine Region, spricht man von einem *Concurrent State* (z. B. Zustand `Web-Seite erstellen` in Abbildung 5.5), da die Sub-Diagramme parallele Abläufe modellieren, die mit Hilfe von *Synch-States* synchronisiert werden können. In diesem Fall gibt es Transitionen, die regionübergreifend sind. Die Sub-Diagramme in den Regionen können aber auch unabhängig sein. Für die Darstellung des Vereinigungsdiagramms von Zustandsdiagrammen ist es daher wichtig, ob der Zustandsdiagramm-Editor es erlaubt, einen Composite State in einen Concurrent State (oder umgekehrt) zu konvertieren oder nicht. Bei genauer Betrachtung entspricht die Konvertierung dem Erzeugen oder Löschen einer Region, so daß wir uns bei der Betrachtung hierauf beschränken können.
Nebenbei sei bemerkt, daß das Einfügen von Regionen unproblematisch ist, ganz im Gegensatz zum Löschen einer Region, wenn sie mittels Synch-States von anderen Regionen abhängig ist. Daher unterstützt nicht jeder Zustandsdiagramm-Editor diese Art von Änderungen.
Falls der Editor das Erzeugen und Löschen von Regionen nicht erlaubt, beschränken sich die Differenzen auf Änderungen an den Sub-Diagrammen und auf verschobene Knoten. Die einzelnen Regionen kann man als unabhängige Knoten mit graphisch dargestellten Komponenten interpretieren. Die Differenzdarstellung ist somit analog zur Differenzdarstellung der Pakete in Klassendiagrammen.
Falls der Editor jedoch das Anlegen und Löschen von Regionen unterstützt, ist es wahrscheinlich, daß mit dem Löschen einer Region auch die darin enthaltenen Komponenten gelöscht werden, da diese ein unabhängiges Sub-Zustandsdiagramm modellieren. In diesem Fall kann man die Region einschließlich der enthaltenen Knoten im Vereinigungsdiagramm einfärben. Falls die Knoten vor dem Löschen einer Region verschoben wurden, müssen sie entsprechend dargestellt werden.

Ein Beispiel für ein Vereinigungsdiagramm von zwei Versionen eines Zustandsdiagramms zeigt Abbildung 5.6. Das Diagramm aus Abbildung 5.6(a) zeigt eine erweiterte Version des Diagramms aus Abbildung 5.5. Alle Diagrammelemente spezifisch zur ersten Version sind rot und alle Elemente, die nur zur zweiten Version gehören, sind grün gefärbt.

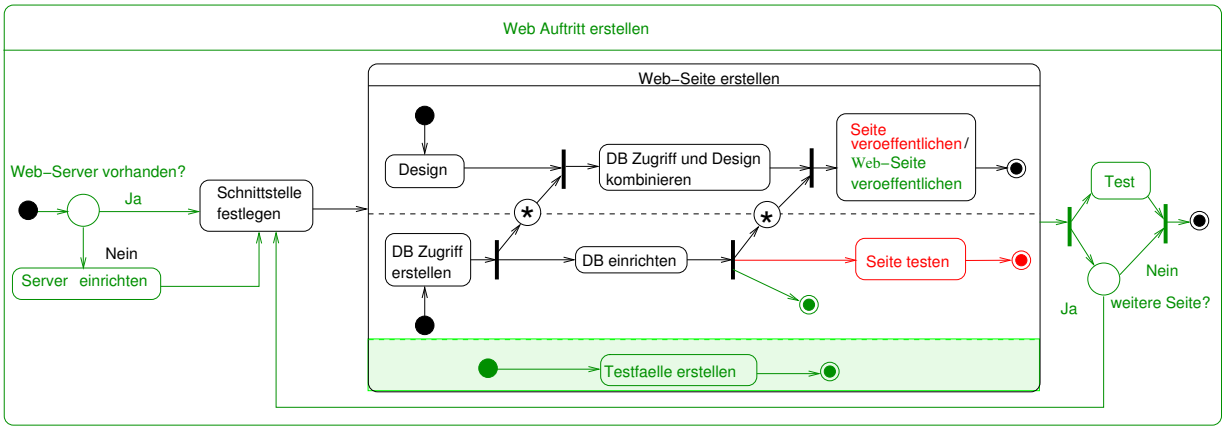
Aus dieser Betrachtung geht hervor, daß das Vereinigungsdiagramm von Zustandsdiagrammen mit den bereits eingeführten Darstellungsmöglichkeiten gezeichnet werden kann.

5.2.5 Aktivitätsdiagramme

Aktivitätsdiagramme sind eine erweiterte Form der Zustandsdiagramme. Daher teilen sie sich viele Diagrammelement-Typen mit ihnen. Zusätzlich gibt es einige weitere einfach strukturierte Knotentypen (Control Icons und Object Flow State), die nur textuell dargestellte Komponenten enthalten. Diese erfordern keine spezielle Betrachtung der Darstellung im Vereinigungsdiagramm, da diese äquivalent zu den bereits bekannten Techniken ist. Jedoch gibt es das Konzept der *Swimlanes*, welches wir genauer analysieren müssen. Sie gruppieren die Diagrammknoten



(a) Erweitertes Zustandsdiagramm



(b) Vereinigungsdiagramm der Zustandsdiagramme aus den Abbildungen 5.5 (rot) und Abbildung 5.6(a) (grün)

Abbildung 5.6: Beispiel eines Zustandsdiagramms einschl. Vereinigungsdiagramm

und entsprechen häufig den organisatorischen Einheiten des zugrundeliegenden Geschäftsmodells. Im Metamodell der UML sind sie auf die Partitionen abgebildet, die die Zustände in Sub-Diagramme einteilen [181, Seiten: 2-174, 3-161ff.]. Vertikale Linien teilen das Diagramm visuell in einzelne Swimlanes. Die Reihenfolge der Swimlanes im Diagramm hat keine semantische Bedeutung. Das Hinzufügen oder Entfernen von Swimlanes in einem Diagramm ist einfach möglich, weil die Swimlanes nur die Knoten eines existierenden Diagramms gruppieren, aber keine Sub-Diagramme darstellen. Gemäß der UML-Spezifikation [181] sind die Knoten innerhalb einer Swimlane *keine* Komponenten der Swimlane, sondern werden nur referenziert.
Das (nachträgliche) Anlegen von neuen Swimlanes erfordert wahrscheinlich das Verschieben von Knoten innerhalb des Diagramms, um sie den einzelnen Swimlanes zuzuordnen. Somit ergeben sich die folgenden Differenzen zwischen zwei Diagrammversionen, die die Swimlanes betreffen:

- verschobene Knoten
- umbenannte Swimlanes
- vertauschte Swimlanes
- neue/gelöschte Swimlanes

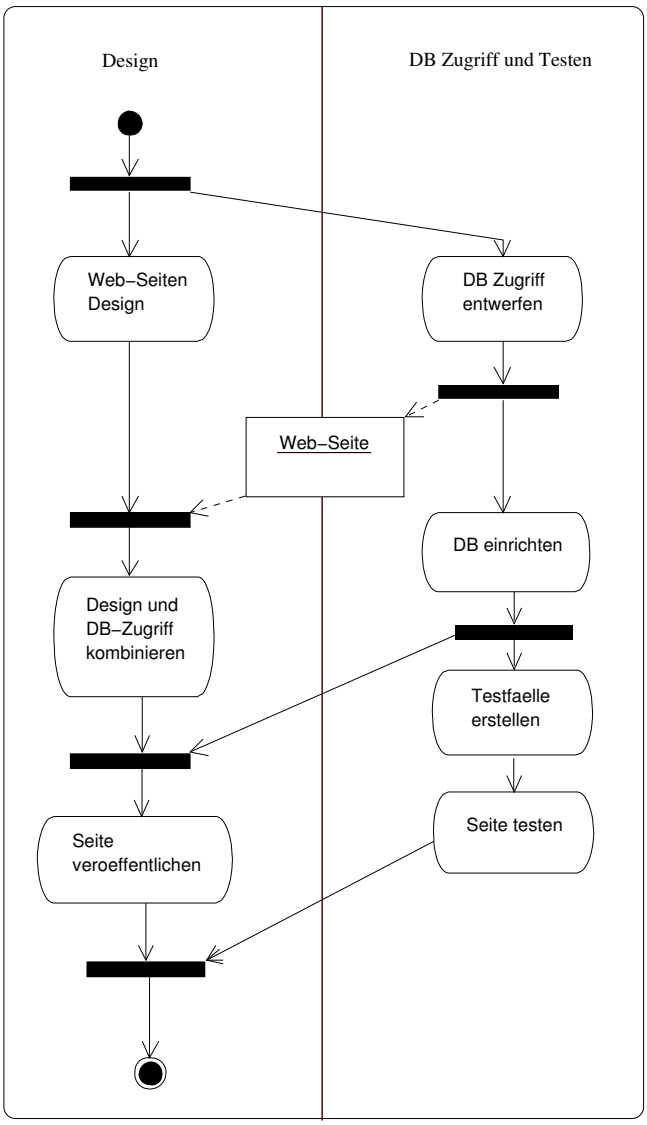
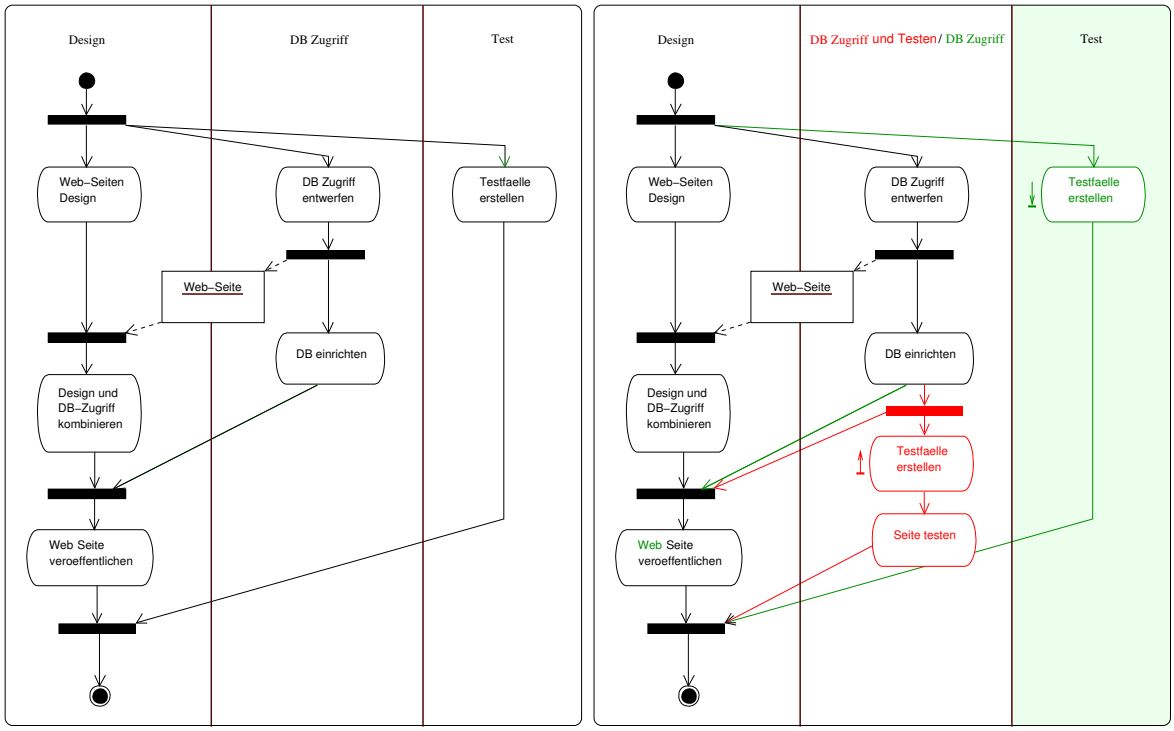


Abbildung 5.7: Beispiel eines Aktivitätsdiagramms

Zwischen Swimlanes verschobene Knoten können so dargestellt werden wie zwischen Paketen verschobene Klassen. Der Bezeichner einer Swimlane wird als Text dargestellt, so daß Umbenennungen entsprechend dargestellt werden können. Interessanter ist die Darstellung von vertauschten oder sogar erzeugten/gelöschten Swimlanes. Es handelt sich um eine Layoutänderung, falls die Swimlanes vertauscht wurden. Für das Vereinigungsdokument verwendet man dann das Layout eines Basisdiagramms. Die vertauschte Reihenfolge der Swimlanes kann man so kennzeichnen wie die Umsortierungen bei Listen. Die Darstellung wird komplizierter, wenn Swimlanes erzeugt und gelöscht wurden. In diesem Fall müssen die Swimlanes beider Basisdiagramme angezeigt werden. Die Reihenfolge sollte dabei möglichst beibehalten werden. Durch einen schwach gefärbten Hintergrund kann man die Zugehörigkeit einer Swimlane zu einem Basisdiagramm kennzeichnen. Knoten, die durch Hinzufügen oder Löschen von Swimlanes verschoben wurden, kann man entsprechend darstellen. Die Lesbarkeit des Vereinigungsdiagramms reduziert sich dann allerdings deutlich mit der Anzahl von geänderten Swimlanes und betroffenen Knoten, da diese einschließlich deren Transitionen doppelt gezeichnet würden. Diese Darstellungsform eignet sich somit nur für eine geringe Anzahl an geänderten Swimlanes und Knoten.



(a) Erweitertes Aktivitätsdiagramm

(b) Vereinigungsdiagramm der Aktivitätsdiagramme aus den Abbildungen 5.7 (rot) und Abbildung 5.8(a) (grün)

Abbildung 5.8: Beispiel eines Aktivitätsdiagramms einschl. Vereinigungsdiagramm

5.2.6 Interaktionsdiagramme

Interaktionsdiagramme modellieren das Verhalten eines Systems unter Berücksichtigung der Struktur der beteiligten Instanzen des Modells und die Kommunikation zwischen ihnen. Die Struktur der beteiligten Instanzen des Modells einschließlich der Beziehungen zwischen ihnen bezeichnet man als Kollaboration, die zwischen ihnen ausgetauschten Nachrichten als Interaktion. Kollaborationen zwischen Klassen können, wie in Abbildung 5.9 gezeigt, dargestellt werden. Interaktionsdiagramme stellen die Kollaborationen im Detail dar. Sie gibt es in zwei Ausprägungen:

1. Sequenzdiagramme
2. Kollaborationsdiagramme

Beide Diagrammtypen basieren auf denselben Daten, so daß eine Transformation des einen Diagrammtyps in den anderen möglich ist. Aus diesem Grund bestehen beide Diagrammtypen auch weitgehend aus denselben Diagrammelementen, deren Darstellung sich jedoch unterscheidet. Der primäre Grund hierfür ist ein anderer Betrachtungswinkel auf die Daten. Sequenzdiagramme konzentrieren sich auf die Interaktion zwischen den Objekten, und Kollaborationsdiagramme betrachten primär die Kollaboration. Die wichtigsten Diagrammelemente sind die Objekte und die Nachrichten, die als Pfeile dargestellt werden.
Die Sequenzdiagramme modellieren die Nachrichten in Form von gerichteten Kanten zwischen den Lebenslinien der Objekte. Die Lebenslinien sind vergleichbar mit einer Zeitachse, die senkrecht von oben nach unter oder horizontal von links nach rechts verläuft und bei Realzeitanwendungen an eine Metrik gebunden werden kann. Die Reihenfolge der Nachrichten zwischen den

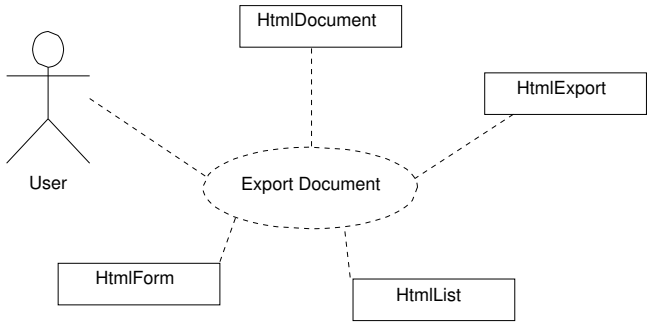


Abbildung 5.9: Beispiel einer Kollaboration

Objekten entlang der Zeitachse legt die Reihenfolge fest. Daher gibt es nur noch einen Freiheitsgrad für die Positionierung der Diagrammelemente, was eine Besonderheit dieses Diagrammtyps ist. Im Gegensatz hierzu liegt der Schwerpunkt der Kollaborationsdiagramme auf der Struktur, so daß sie keine separate Zeitachse besitzen. Daher wird die Nachrichtenreihenfolge durch (automatisch) berechnete Sequenznummern dargestellt. Die Sequenznummern sind im Vergleich zu den bisher betrachteten Diagrammtypen eine Neuheit, da sie automatisch berechnet, aber auch manuell durch den Anwender vergeben werden können.
Die Reihenfolge der Nachrichten ist im UML-Metamodell durch explizite Beziehungen zwischen den entsprechenden Modellobjekten festgelegt, so daß das Einfügen oder Löschen von Methodenaufrufen im Editiermodell der Diagramme einfach möglich ist. Die Umsetzung in konkreten Werkzeugen unterscheidet sich jedoch für Sequenz- und Kollaborationsdiagramme.

In Sequenzdiagrammen lassen sich durch Einfügen oder Löschen eines (neuen) Aufrufpfeiles zwischen bestehenden Pfeilen weitere Nachrichten zwischen Objekten einfügen oder existierende löschen. Diese Änderungen sind für Kollaborationsdiagramme erheblich komplexer, da eine bestehende Nachrichtenreihenfolge um weitere Nachrichten erweitert oder um existierende Nachrichten gekürzt werden müßte. Das hat die Neuberechnung aller Sequenznummern zur Folge. Zu beachten ist hier, daß diese komplexen Änderungen des Editiermodells nicht direkt im Vereinigungsdiagramm darstellbar sind. Dieses kann nur den Zustand vor und nach den Änderungen widerspiegeln.

5.2.6.1 Sequenzdiagramme

Eine weitere Besonderheit der Sequenzdiagramme sind neben der expliziten Zeitachse die Lebenslinien der Objekte. Diese geben die Dauer der Existenz eines Objektes an. Die Zerstörung eines Objektes innerhalb des Diagramms kennzeichnet man durch ein großes `X`. Die Lebenslinie kann in 2 oder mehr Linien aufgeteilt werden, um bedingte Verzweigungen oder Rekursion explizit darzustellen. Da die Lebenslinien eigenständige Diagrammelemente sind, genau wie die Objekte selbst, können Differenzen zwischen zwei Diagrammversionen, die die Lebenslinie oder die Objekte betreffen, durch bereits bekannte Techniken dargestellt werden. Die Reihenfolge der Objekte ist semantisch nicht bedeutsam, so daß die Reihenfolge aus den Basisdiagrammen übernommen werden kann (vgl. Abbildung 5.10). An den Lebenslinien können weiterhin entlang der Zeitachse Laufzeiten oder Verzögerungen notiert sein. Dabei handelt es sich um Texte, deren Differenzdarstellung analog zur Differenzdarstellung anderer Texte ist.
Wir müssen jedoch die Nachrichten näher betrachten. Zwei Pfeile mit einem gemeinsamen Startpunkt, aber unterschiedlichen Zielen drücken eine bedingte Verzweigung aus. An den Pfeilen wird dann die Bedingung notiert. Weiterhin beschreiben unterschiedliche Pfeilspitzen die Art einer Nachricht: Methodenaufruf, asynchrone Kommunikation oder Rücksprung aus einer Methode. Betrifft eine Änderung nur die Bedingung, läßt sich diese durch die Markierung des

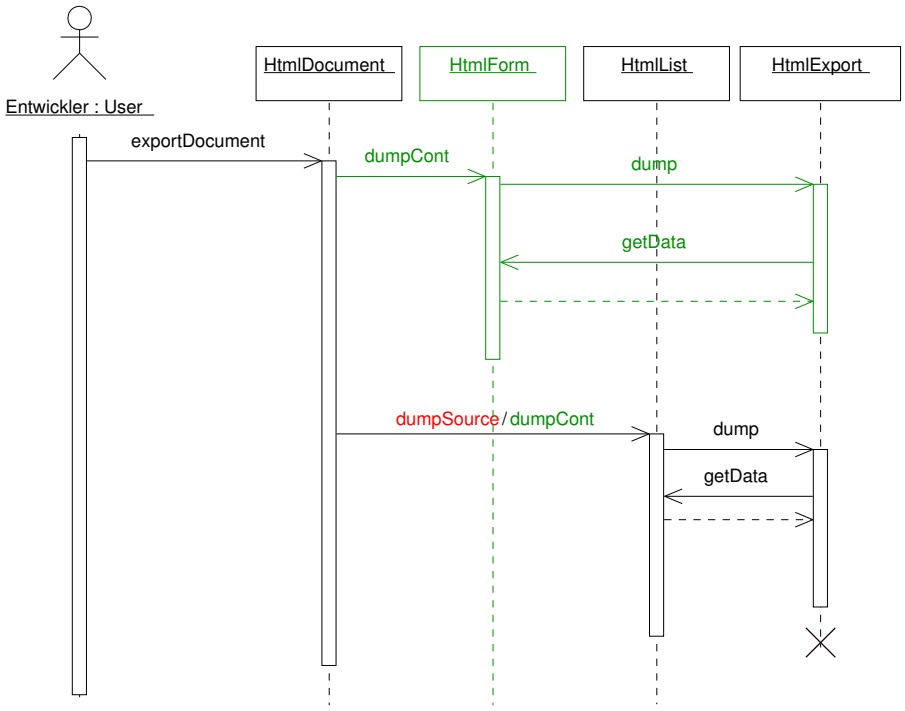


Abbildung 5.10: Beispiel eines Vereinigungsdiagramms von Sequenzdiagrammen

Textes kennzeichnen. Die Änderung der Art einer Nachricht von z. B. einem Methodenaufruf in eine asynchrone Kommunikation läßt sich durch eine doppelte und farbige Darstellung des entsprechenden Pfeils im Diagramm kennzeichnen. Das Hinzufügen, Löschen oder Verschieben von Pfeilen ist genauso zu handhaben wie z. B. das Hinzufügen oder Löschen von Beziehungen in Klassendiagrammen, unabhängig davon, ob sie einen gemeinsamen Startpunkt oder verschiedene Startpunkte besitzen.

5.2.6.2 Kollaborationsdiagramme

In Kollaborationsdiagrammen kann man zwischen aktiven und passiven Objekten unterscheiden. Die aktiven Objekte veranlassen die Kommunikation, während die passiven Objekte primär Daten speichern. Im Diagramm unterscheidet man die aktiven von den passiven Objekten durch einen stärker gezeichneten Rahmen. Es ist jedoch auch möglich, das Schlüsselwort `active` als Stereotyp zu verwenden. Häufig enthalten die aktiven Objekte auch Komponenten. Die Differenzdarstellung dieser Diagrammelemente unterscheidet sich nicht von der Darstellung bei anderen komplexen Diagrammelementen und Texten.
Die Objekte sind durch Assoziationen miteinander verbunden. Die Richtung der Nachrichten zwischen den Objekten wird durch einen kleinen zusätzlichen Pfeil an den Assoziationen gekennzeichnet. Gibt es mehrere Nachrichten zwischen zwei Objekten in derselben Richtung, notiert man sie als Liste am Richtungspfeil. Jede Nachricht ist gekennzeichnet durch einen Bezeichner (z. B. der Methodenname) und eine Sequenznummer, die die Reihenfolge angibt. Die Sequenznummern können fortlaufend sein oder auch eine hierarchische Struktur besitzen. Differenzen, die die Bezeichner betreffen, können wie bereits bekannt gehandhabt werden (vgl. Abbildung 5.11). Die Reihenfolge der Nachrichten in der Liste ist durch die Reihenfolge aller Nachrichten im Diagramm festgelegt, so daß die Sortierung der Liste der Nachrichten nicht direkt änderbar ist und somit deren Differenzdarstellung von den Sequenznummern abhängt.
Je nach eingesetztem Werkzeug können die Sequenznummern durch den Anwender manuell eingegeben oder durch das Werkzeug berechnet werden. Erster Fall ist trivial, daher wird dessen

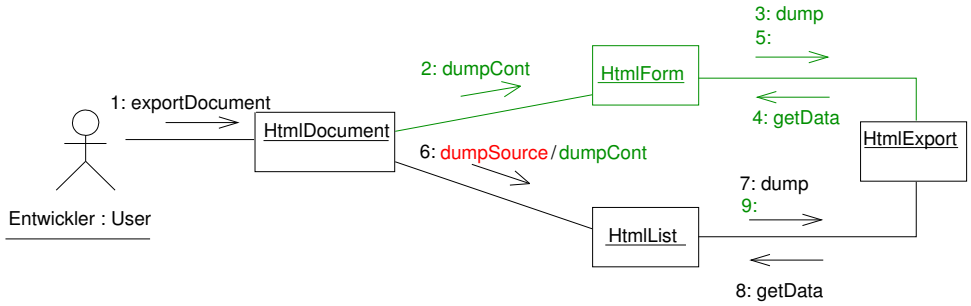


Abbildung 5.11: Beispiel eines Vereinigungsdiagramms von Kollaborationsdiagrammen

Differenzdarstellung nicht weiter diskutiert. Die automatische Berechnung der Sequenznummern basiert auf der Reihenfolge der Nachrichten, die durch den Anwender angelegt, gelöscht oder verschoben werden können. Änderungen an der Reihenfolge der Nachrichten wirken sich somit auf die Sequenznummern aus, ohne daß der Anwender diese explizit ändern muß. Es stellt sich die Frage, ob Differenzen zwischen automatisch berechneten Werten von Interesse sind. Insbesondere ruft das Einfügen einer Nachricht in eine lange Sequenz von Nachrichten viele Folgeänderungen an den Sequenznummern hervor, die alle im Vereinigungsdiagramm gekennzeichnet würden und somit die Übersicht verschlechtern würden. Daher erscheint es sinnvoll, nur die geänderten Sequenznummern farbig zu markieren, die durch Ändern der zugehörigen Nachricht neu vergeben wurden, und nicht die Sequenznummern, die daraufhin neu berechnet wurden.

5.2.7 Implementierungsdiagramme

Implementierungsdiagramme gibt es in zwei Ausprägungen:

1. Komponentendiagramme

2. Verteilungsdiagramme

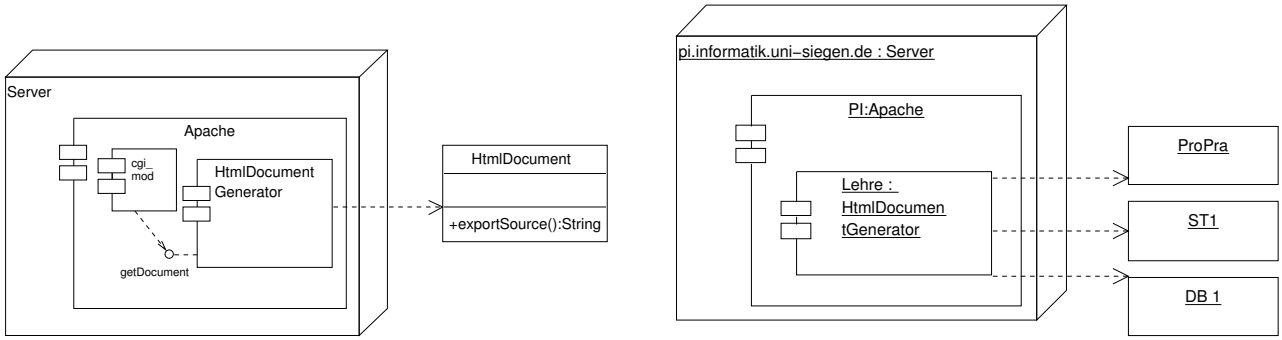
Die Komponentendiagramme stellen die Abhängigkeiten zwischen Softwarekomponenten[6] einschließlich der Classifier (wie z. B. Klassen, Interfaces, usw.) und den Artefakten, die sie implementieren, wie z. B. Source Code, Skripten usw. dar. Dieser Diagrammtyp enthält nur Typen und keine Instanzen, ganz im Gegensatz zu den Verteilungsdiagrammen[7], die die Konfiguration von den Instanzen der Komponenten und den Bearbeitungsknoten[8] visualisieren.

Die Komponenten und Knoten können andere Diagrammknoten beinhalten und sie sind durch einen Typ spezifiziert. Handelt es sich um eine Instanz, wird der Typ noch um einen Bezeichner ergänzt. Je nach eingesetztem Werkzeug können die von den Komponenten angebotenen Schnittstellen durch kleine Kreise, die durch eine gestrichelte Linie mit der Komponente verbunden ist, dargestellt werden. Somit läßt sich die Abhängigkeit von einer bestimmten Schnittstelle einer Komponente darstellen (siehe Abbildung 5.12). Die Reihenfolge der Schnittstellen-Darstellungen ist beliebig.
Die Implementierungsdiagramme haben keine besonderen Eigenschaften, die bei der Differenzdarstellung berücksichtigt werden müßten, von daher erübrigt sich eine Diskussion.

[6] Softwarekomponenten sind modulare, verteilbare und austauschbare Teile eines Systems, welche eine Implementierung beinhalten und eine Schnittstelle nach außen anbieten.

[7] Diese Diagramme werden teilweise auch als Einsatzdiagramme bezeichnet.

[8] Ein Bearbeitungsknoten ist ein physikalisches Objekt, welches eine Bearbeitungseinheit darstellt. Beispiele sind: Prozessoren, externe Geräte usw.



(a) Komponentendiagramm (b) Verteilungsdiagramm

Abbildung 5.12: Beispiele für Implementierungsdiagramme

5.2.8 Arten von Differenzen

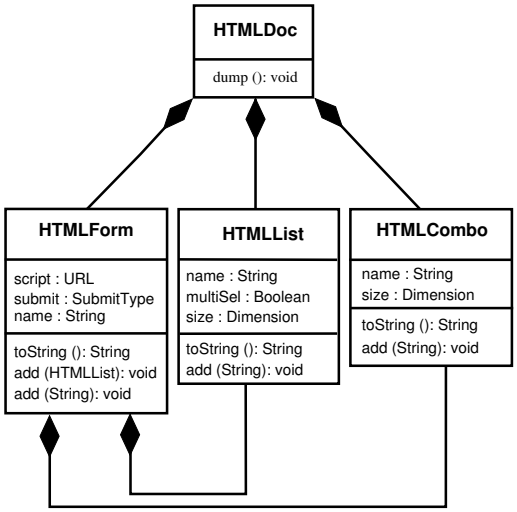
Zusammenfassend läßt sich nach der Betrachtung der einzelnen Diagrammtypen der UML sagen, daß sich die Arten von Differenzen und deren Darstellung, die bei der Betrachtung von Klassendiagrammen in Abschnitt 5.2.1 herausgearbeitet wurden, bis auf wenige Ausnahmen auch auf die anderen Diagrammtypen der UML anwendbar sind. Die Ausnahmen sind insbesondere automatisch berechnete Werte wie die Sequenznummern in Kollaborationsdiagrammen. Sofern weitere Diagrammtypen keine Besonderheiten aufweisen, kann das Gesagte auch auf andere Diagrammtypen übertragen werden, beispielsweise auf ER-Diagramme oder Datenflußdiagramme. Die farbige Darstellung der versionsspezifischen Diagrammelemente ist auf alle Diagrammtypen anwendbar.

5.3 Gruppieren von Differenzen

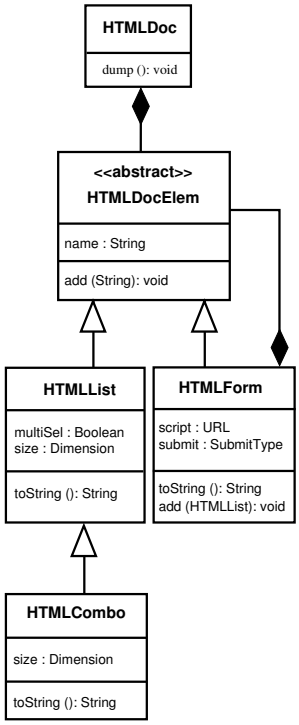
Ein etwas umfangreicheres Beispiel für eine Differenzdarstellung zeigt Abbildung 5.13. Das Ausgangsdiagramm (Abbildung 5.13(a)) zeigt eine vereinfachte Klassenstruktur zur Modellierung von HTML-Seiten. Diese wird durch Generalisierung und Einführung des Kompositum-Patterns umstrukturiert (Abbildung 5.13(b)). Die Erweiterung durch das Visitor-Pattern bietet anschließend die Möglichkeit, die modellierten HTML-Seiten zu exportieren (Abbildung 5.13(c)). Abbildung 5.13(d) zeigt abschließend die Differenzen zwischen der Ausgangsversion und der dritten Version des Diagramms. Wie man leicht erkennt, sinkt mit steigender Anzahl der Differenzen die Übersichtlichkeit und damit der Nutzen des Vereinigungsdiagramms. Diese Darstellung wäre in dieser Form nur für kleine Diagramme bzw. für Diagramme mit wenigen Differenzen nutzbar.

Jedoch enthalten Diagramme, die sich über mehrere Versionen entwickelt haben, u.U. viele Differenzen. Eine Möglichkeit, die Übersichtlichkeit des Vereinigungsdiagramms zu erhöhen, besteht nun darin, nicht alle Differenzen gleichzeitig anzuzeigen, sondern diese zu gruppieren und nur einzelne Differenz-Gruppen farbig zu markieren. Die nicht markierten Differenz-Gruppen sollte man von den unveränderten Diagrammelementen unterscheiden können, jedoch sollten sie nicht mehr so auffällig sein. Eine Lösung ist, sie in einem grauen Farbton darzustellen. Prinzipiell sind mehrere Methoden denkbar, die Differenzen zu gruppieren:

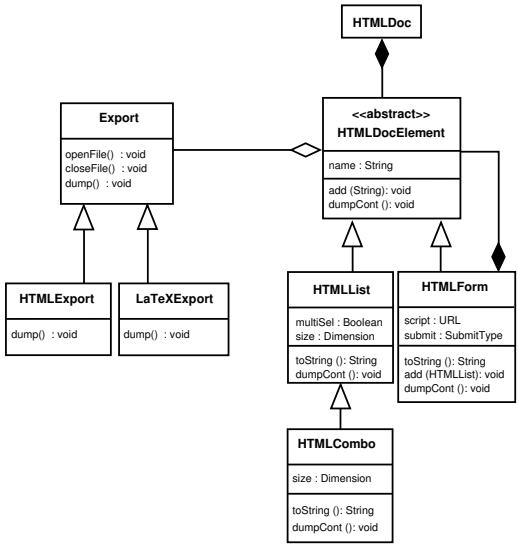
- **Versionsinformationen nutzen**: Sind die zu vergleichenden Dokumente unter Verwendung des Versionsverwaltungskonzeptes aus Kapitel 3 erstellt worden, so kann man die Differenzen einzelnen Konfigurationen zuordnen. Anhand der Konfigurationen lassen sich



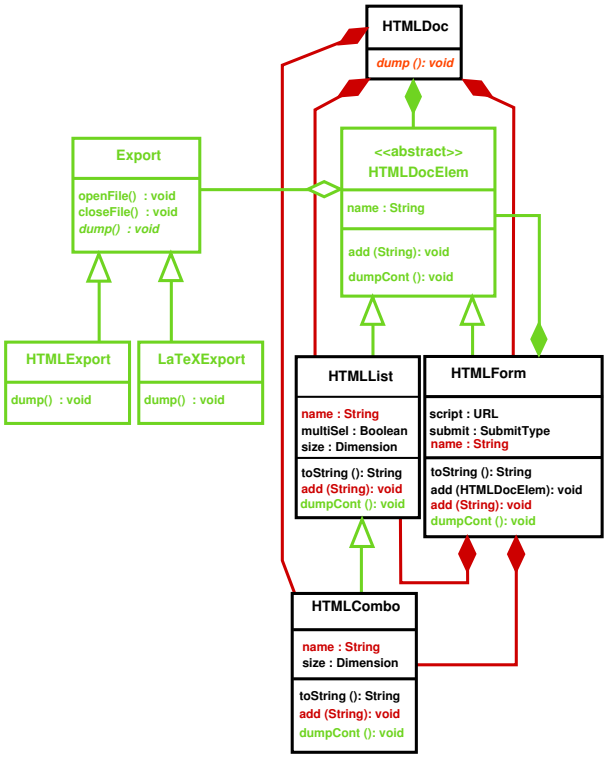
(a) Ausgangsversion



(b) Umstrukturierung



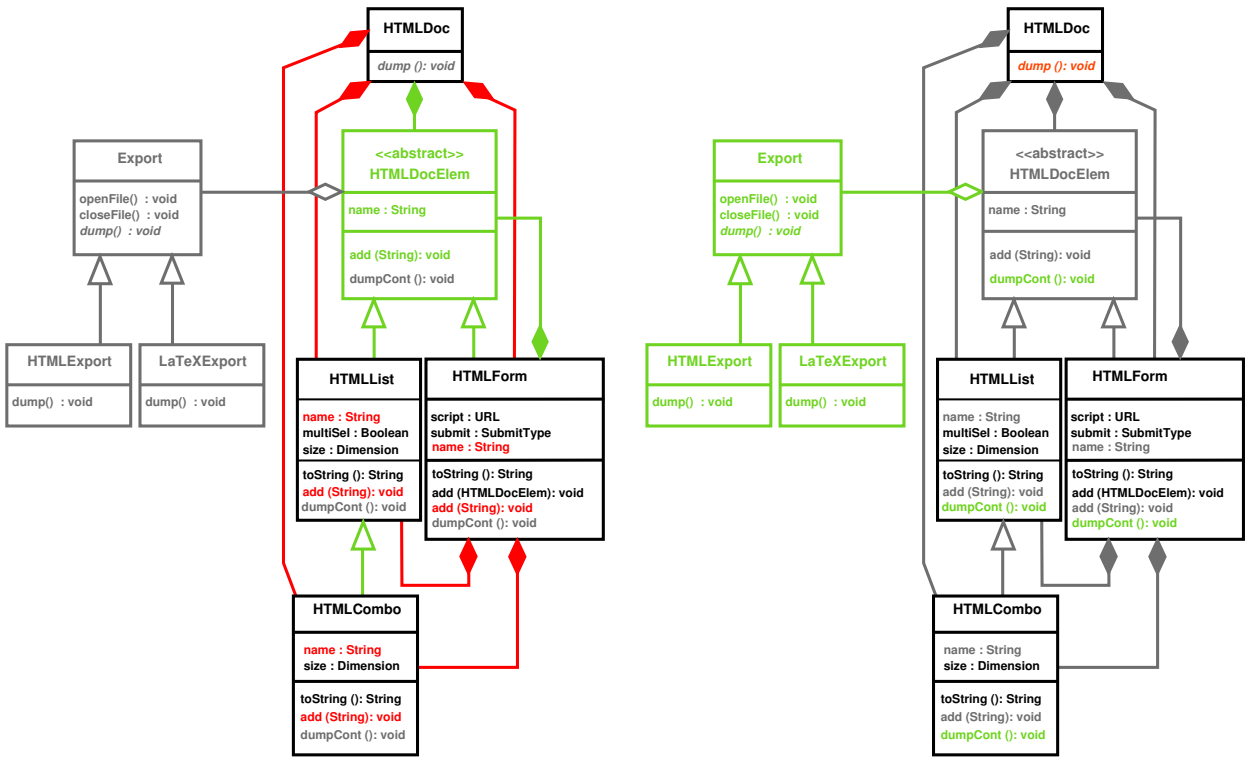
(c) Erweiterung um Exportfunktionalität



(d) Differenzdarstellung zwischen 5.13(a) und 5.13(c)

Abbildung 5.13: Beispiel für die Entwicklung eines Klassendiagramms einschließlich der Differenzanzeige

die Differenzen dann gruppieren. Unter der Annahme, daß in einer Konfiguration nur Änderungen zusammengefaßt sind, die in einem direktem Zusammenhang stehen, ist somit die Gruppierung auf die zusammenhängenden Änderungen ausgerichtet. Ein Beispiel hierfür ist die Entwicklung aus Abbildung 5.13. Wenn das Ausgangsdiagramm und jede Erweiterung im Rahmen einer Werkzeugtransaktion durchgeführt wurde, kann man die Differenzen zwei Konfigurationen zuordnen, je eine für jede Erweiterung des Ausgangsdiagramms. Daraus folgen dann die in Abbildung 5.14 dargestellten Vereinigungsdiagramme mit gruppierten Differenzen.



(a) Umstrukturierung durch Generalisierung

(b) Erweiterung um Exportfunktionalität

Abbildung 5.14: Beispiel für die gruppierte Anzeige von Differenzen

Zu beachten ist jedoch, daß wiederholte Änderungen an einem Diagrammelement unberücksichtigt bleiben, da die Anzeige nur die Differenzen enthält und nicht alle Änderungen. Wurde beispielsweise eine Klasse mehrfach umbenannt, so hat nur die letzte Umbenennung Auswirkungen auf die gruppierte Differenzdarstellung.

Ergänzend kann man die zu den Konfigurationen gehörenden Änderungskommentare und die an der Erstellung beteiligten Entwickler in dem Differenzwerkzeug anzeigen.

- **Editiermodell nutzen**: Berücksichtigt man das Editiermodell, so kann man die Differenzen anhand der Editiermodellelement-Typen gruppieren. Beispielsweise kann man die Anzeige der Differenzen auf Änderungen an Attributen oder Methoden der Klassen beschränken oder sich nur Änderungen von Assoziationen oder Vererbungsbeziehungen anzeigen lassen. Das ist insbesondere dann vorteilhaft, wenn nur bestimmte Arten von Änderungen interessant sind.

  Diese Art der Gruppierung ist jedoch spezifisch für jeden Diagrammtyp und abhängig vom Editiermodell. Liegt ein grobkörniges Editiermodell vor, so lassen sich weniger Grup-

pen bilden, als bei einem feinkörnigen Editiermodell, welches sich eng an die UML-Spezifikation anlehnt. Prinzipiell lassen sich die Differenzen anhand aller einzeln modellierten Editiermodell-Typen gruppieren. Welche Gruppierungen sinnvoll sind, muß jedoch für jeden Diagrammtyp, Werkzeug und Anwendungsfall einzeln entschieden werden.

- **explizite und implizite Differenzen unterscheiden**: Bei einigen Diagrammtypen kann man auch zwischen expliziten und impliziten Differenzen unterscheiden. Explizite Differenzen sind auf Änderungen zurückzuführen, die durch den Werkzeuganwender direkt vorgenommen wurden. Beispiele sind unterschiedliche Bezeichner einer Klasse, Methode, usw. Bei impliziten Differenzen wirken sich Änderungen an einem Diagrammelement auf weitere Diagrammelemente aus. Wenn eine Klasse z. B. als Attributtyp in anderen Klassen verwendet wird und im Laufe der Entwicklung umbenannt wird, so erscheint der neue Name der Klasse nicht ausschließlich an der Klasse selbst, sondern auch als neuer Attributtyp bei allen Attributen, die diese Klasse als Typ referenzieren. Wurde jedoch der Typ eines einzelnen Attributs direkt geändert, also eine andere Klasse ausgewählt, so handelt es sich um eine explizite Differenz.

  Explizite Differenzen sind also Differenzen, die an dem Diagrammelement angezeigt werden, wo die Änderung vorgenommen wurde und implizite Differenzen gehen auf Änderungen zurück, die an anderen Diagrammelementen vorgenommen wurden, die jedoch mit dem angezeigten Diagrammelement in Zusammenhang stehen.

  Impliziten Differenzen können auf referenzierte Diagrammelemente oder auf berechnete Informationen zurückzuführen sein. In Interaktionsdiagrammen werden z. B. die Sequenznummern teilweise durch Werkzeuge automatisch anhand der Einfügereihenfolge der einzelnen Methoden berechnet. Bietet das Werkzeug dann die Möglichkeit, weitere Methoden in die Sequenz einzufügen, hat das Auswirkungen auf die nachfolgenden Sequenznummern. Diese sind somit implizit verändert worden.

  Enthält ein Diagramm viele referenzierte oder berechnete Informationen, so können einige wenige Änderungen zu einer Vielzahl an visualisierten Differenzen führen, wovon die meisten nur indirekt auf die Änderungen zurückzuführen sind. Somit ist die Gruppierung nach expliziten und impliziten Differenzen sinnvoll.

- **Kombinationen der oben genannten Gruppierungen**: Es sind auch Kombinationen der genannten Gruppierungsmöglichkeiten denkbar. Man könnte z. B. die Differenzanzeige auch auf Änderungen beschränken, die nur bestimmte Diagrammelement-Typen direkt betreffen und durch einzelne Konfigurationen zusammengefaßt werden.

5.4 Mischen von Software-Dokumenten

Die Anzeige von Differenzen zwischen UML-Diagrammen ist der erste Schritt zum Mischen von zwei Versionen eines Diagramms. Auf den Ergebnissen des vorhergehenden Abschnitts aufbauend diskutieren wir in diesem Abschnitt das Konzept zum Mischen von zwei Versionen eines UML-Diagramms.
Das Mischkonzept basiert auf den Diagrammen als Ganzes und nicht auf einzelnen Objektversionen, die zusammen ein Diagramm bilden. Letzteres hätte den Vorteil, daß die Versionsverwaltung alle geänderten Versionen liefern könnte, jedoch würde bei einer Betrachtung der einzelnen Objektversionen der Kontext unberücksichtigt bleiben. Der ist jedoch wichtig, um unterschiedliche Arten von Konflikten unterscheiden zu können, die wir in Abschnitt 5.4.1 vorstellen. In Abschnitt 5.4.2 stellen wir das Mischkonzept vor. Die Darstellung von Konflikten

diskutieren wir in Abschnitt 5.4.3 und das vorgeschlagene Konzept zum Lösen der Konflikte in Abschnitt 5.4.4.

5.4.1 Konflikte beim Mischen von Diagrammen

Das Ergebnis von Abschnitt 5.2.8 ist eine Liste von möglichen Änderungen an Diagrammen, woraus die Differenzen zwischen Diagrammen abgeleitet wurden. Auf Basis dieser Liste von möglichen Änderungen kann man auch bestimmen, welche Konflikte beim Mischen von Diagrammen auftreten können. Zu beachten ist bei der Betrachtung von Misch-Konflikten, daß ein Konflikt immer nur dann auftritt, wenn beide zu mischende Versionen gleichberechtigt sind, d.h. daß ein symmetrisches Mischen vorliegt. Erhalten die Änderungen einer der beteiligten Versionen eine höhere Priorität beim Mischen, so treten keine nicht automatisch lösbaren Konflikte auf, da bei konkurrierenden Änderungen die Änderung der höher gewichteten Version gewählt wird.

In Tabelle 5.4 sind die möglichen Änderungen an Diagrammen aus Abschnitt 5.2.8 noch einmal aufgelistet. Um die Übersichtlichkeit der beiden darauf folgenden Tabellen 5.5 und 5.6 zu erhöhen, ist jeder Änderung eine Kennzahl zugeordnet, die in diesen beiden Tabellen die durchgeführten Änderungen beschreibt. Zur Verdeutlichung der möglichen Änderungen sei noch kurz wiederholt, daß es sich bei den Elementen in Listen/Mengen (z. B. Liste der Attribute einer Klasse oder Menge der Stereotypen) auch um Komponenten eines Diagrammelementes handelt.

Art der Änderung	**Abkürzung**
Struktur-Änderungen	
neuer Diagramm-Knoten	1
gelöschter Diagramm-Knoten	2
neue Beziehungen am Diagramm-Knoten	3
gelöschte Beziehungen vom Diagramm-Knoten	4
Intra-Knoten/Beziehungsänderungen	
atomare Wertänderung: Texte, Aufzähltyp oder Boolescher Wert	5
neue Komponenten/Element in Listen/Mengen	6
gelöschte Komponenten/Element aus Listen/Mengen	7
Umordnen einer Liste	8
extern referenzierte Elemente austauschen: z. B. Attributtyp ändern	9
Inter-Knoten/Beziehungsänderungen: (werkzeugabhängig)	
Verschieben einer Komponenten zum betrachteten Diagr.-Knoten hin	10
Verschieben einer Komponenten vom betrachteten Diagr.-Knoten weg	11
Verschieben einer/der Beziehungen (zum betrachteten Diagr.-Knoten hin)	12
Verschieben einer/der Beziehungen (vom betrachteten Diagr.-Knoten weg)	13

Tabelle 5.4: Mögliche Änderungen an Diagrammen (aus Abschnitt 5.2.1)

Extern referenzierte Elemente sind Beziehungen im Metamodell, die sich in den Diagrammen so auswirken, daß eine Änderung an einem Diagrammelement auch Auswirkungen auf andere Diagrammelemente hat. Als Beispiel sei das Umbenennen einer Klasse genannt. Bei dieser Klasse kann es sich um den Typ eines Attributes einer anderen Klasse handeln. Die Umbenennung hat in diesem Fall die Wirkung, daß sich auch die Bezeichnung des Attributtyps ändert. Dieser Fall ist von der Wahl eines anderen Attributtypen zu unterscheiden. Diese Änderung hat

nur Auswirkungen auf das Attribut selbst, da die ursprüngliche Klasse nicht verändert wurde (Kennzahl 9 aus Tabelle 5.4).
Weiterhin müssen wir das Verschieben von Elementen oder Beziehungen genauer betrachten. Für Konflikte, also konkurrierende Änderungen an *einem* Diagrammelement, ist es relevant, ob eine Komponente von dem betrachteten Diagrammelement ausgehend zu einem anderen Diagrammelement hin geschoben wurde oder umgekehrt. Das Verschieben zu einem anderen Diagrammelement ist vergleichbar mit dem Löschen; das Verschieben zu dem betrachteten Diagrammelement ist vergleichbar mit dem Hinzufügen. Aus diesem Grund wird dieser Unterschied bei diesen Änderungstypen berücksichtigt.

Für die folgenden Betrachtungen gehen wir vom symmetrischen Mischen aus. In Tabelle 5.5 betrachten wir nur Änderungen an einem Diagrammelement. Unter Diagrammelement verstehen wir hierbei einen Knoten im Diagramm, Beziehungen zwischen Knoten oder auch Komponenten eines Knotens. Demzufolge kann sich z. B. eine atomare Wertänderung auf das Umbenennen einer Klasse, der Rolle einer Beziehung oder einer Methode beziehen. Eine Liste von Komponenten betrachten wir auch als ein Diagrammelement, so daß das Hinzufügen einer Listen-Komponente und das Umordnen der Liste zwei Änderungen an demselben Diagrammelement sind. Die Entscheidung, ob es sich um ein Diagrammelement handelt wird dabei unter Berücksichtigung des Metamodells getroffen. Bei dem Hinzufügen von zwei gleich benannten Klassen in ein Diagramm handelt es sich demzufolge um zwei Diagrammelemente.

Änderungsarten	1	2	3	4	5	6	7	8	9	10	11	12	13
1	-	-	-	-	-	-	-	-	-	-	-	-	-
2	-		×		×	×		×	×	×		×	
3	-	×	-	-	-	-	-	-	-	-	-	-	-
4	-		-		×	×		×	×	-	-	×	×
5	-	×	-	×	×	-	×						
6	-	×	-	×	-	-	-	×	-	-	-	-	-
7	-		-		×	-			×	-	×		
8	-	×	-	×		×		×		×			
9	-	×	-	×		-	×		×				
10	-	×	-	-		-	-	×		-	-		
11	-		-	-		-	×			-	×		
12	-	×	-	×		-						-	-
13	-		-	×		-						-	×

Tabelle 5.5: Verhältnis von Änderungen an *einem* Diagrammelement in verschiedenen Versionen

(Kennzahlen lt. Tab. 5.4; ×:= Konflikt ; - := nicht möglich ; Fall 8: Umordnen einer Liste betrifft mehrere Elemente, von daher hier nicht berücksichtigt)

Bei genauer Betrachtung der möglichen Konflikte fällt auf, daß die meisten Konflikte auf Löschungen zurückzuführen sind (*Löschkonflikte*). Die entsprechenden Fälle sind zur besseren Übersicht in der Tabelle 5.5 grau hinerlegt. Beispiele hierfür sind z. B. das Löschen einer Klasse in einer Version und das Hinzufügen von neuen Methoden in einer anderen Version (Fall: 2 vs. 6). Ein weiteres Beispiel ist das Löschen einer Beziehung in einem Kollaborationsdiagramm, der in der anderen Version eine andere Methode einer existierenden Klasse zugeordnet wurde (Fall: 4 vs. 9). Interessant ist noch zu bemerken, daß das Löschen von verschiedenen Teilen eines Diagrammelementes oder des Elementes selbst keine Konflikte verursacht. Das ist darauf

zurückzuführen, daß z. B. das Löschen eines Diagramm-Knotens auch das Löschen seiner Komponenten impliziert und somit kompatibel mit dem Löschen einer Komponente oder Beziehung ist.
Eine weitere Besonderheit von Konflikten, die auf Löschungen zurückzuführen sind, ist, daß ein Konflikt vorliegen kann, wenn Änderungen an zwei verschiedenen Knoten des Editiermodells, die auf einem Pfad von der Wurzel zu den Blättern liegen, durchgeführt wurden. Beispiele hierfür sind das Löschen einer Klasse und das Ändern eines Methodenbezeichners. Beide Änderungen sind an unterschiedlichen Knoten des Editiermodells durchgeführt worden. Das Löschen ist jedoch rekursiv auf alle Komponenten angewendet worden, so daß dieser Konflikt bei genauerer Betrachtung dem Konflikt Löschen vs. Umbenennen der Komponente entspricht.
Die verbleibenden Konflikte sind entweder auf gleichartige Änderungen am selben Diagrammelement zurückzuführen (z. B. konkurrierendes Ändern eines Bezeichners oder Auswahl eines anderen referenzierten Elementes) oder aber auf das Umsortieren einer Liste. Beim Umsortieren ist insbesondere das Hinzufügen von neuen Elementen problematisch, da diese eine andere Position erhalten müssen als in der Version, in der sie hinzugefügt wurden. Diese Art des Konflikts unterscheidet sich von den anderen beiden Konfliktarten dahingehend, daß beide Änderungen in die Mischversion einfließen können. Für das neue Listenelement muß man lediglich eine Position in der Liste angeben.
Im Gegensatz zu dem Hinzufügen von Listenelementen ist das Löschen von Elementen aus einer Liste in Kombination mit dem Umordnen eher unproblematisch, da das Element aus der Liste einfach entfernt werden kann, ohne daß das die Ordnung der Liste sich ändern würde.
Bei Konflikten, die auf gleichartige Änderungen zurückzuführen sind, muß man noch den Sonderfall berücksichtigen, daß in beiden Versionen dieselbe Änderung durchgeführt wurde. Beispielsweise wurde eine Klasse in beiden Versionen in gleicher Art und Weise umbenannt (Fall: 5 vs. 5) oder es wurde der selbe Typ für ein Attribut gewählt (Fall: 9 vs. 9). In diesen Fällen führen die konkurrierenden Änderungen zu keinem Konflikt.

In Tabelle 5.6 betrachten wir die Auswirkungen von Änderungen, die an zwei unterschiedlichen Diagrammelementen durchgeführt wurden. Diese Elemente müssen jedoch im selben Namensraum liegen, es muß sich also z. B. um zwei Klassen in einem Paket, um zwei Beziehungen zwischen den selben Diagramm-Knoten oder um zwei Methoden der selben Klasse handeln.
Die Konflikte, die in diesem Fall auftreten, bezeichnen wir als syntaktische Konflikte und sie unterscheiden sich von den Konflikten, die auf das Mischen von geänderten Diagrammelementen zurückzuführen sind (i.f. *Mischkonflikte*; s. auch Def. in Abschnitt 1.2.4). Bei syntaktischen Konflikten handelt es sich um zwei Elemente im selben Namensraum, die keinen eindeutigen Bezeichner besitzen. Als Beispiele sind hier zu nennen das Hinzufügen einer neuen Klasse und das Umbenennen einer existierenden Klasse (Fall: 1 vs. 5). Wenn die neue Klasse und die umbenannte Klasse den selben Bezeichner besitzen liegt ein syntaktischer Konflikt vor, da das in UML-Diagrammen nicht erlaubt ist[9]. Allgemein betrachtet, ist ein syntaktischer Konflikt ein Konflikt an dem zwei oder mehr Elemente beteiligt sind und der im Widerspruch zu den Konsistenzkriterien des modellierten Dokumentes steht. Im Gegensatz hierzu verletzen Mischkonflikte die Konsistenz des Editiermodells, wohingegen das Editiermodell bei syntaktischen Konflikten konsistent bleibt.

Betrachtet man die Tabelle, so fällt auf, daß bei Auftreten eines syntaktischen Konflikts immer ein Diagrammelement hinzugefügt wurde – von dem Fall, daß zwei Elemente durch Umbenennen denselben Bezeichner erhalten haben, abgesehen.

[9]Syntaktische Konflikte können nicht ausschließlich beim Mischen von zwei Versionen eines Dokumentes auftreten. Ein syntaktischer Konflikt tritt auch auf, wenn in einer Version eines Klassendiagramms eine neue Klasse angelegt wird, deren Bezeichner identisch mit dem Bezeichner einer bereits existierenden Klasse ist. Mischkonflikte treten jedoch ausschließlich beim Mischen auf.

Änderungsarten	1	2	3	4	5	6	7	8	9	10	11	12	13
1	S				S								
2													
3			S		S							S	
4													
5	S		S		S	S				S		S	
6					S	S			S	S			
7													
8													
9						S							
10					S	S				S			
11													
12					S							S	
13													

Tabelle 5.6: Verhältnis von Änderungen an *zwei unterschiedlichen* Diagrammelementen im selben Namensraum in verschiedenen Versionen

(Kennzahlen lt. Tab. 5.4; S := syntaktischer Konflikt)

Zusammenfassend läßt sich sagen, daß Mischkonflikte immer nur bei Änderungen an einem Diagrammelement auftreten, bei syntaktischen Konflikten sind jedoch immer zwei unterschiedliche Diagrammelemente im selben Namensraum beteiligt. Darauf läßt sich auch die Eigenschaft zurückführen, daß Mischkonflikte oft durch Löschungen verursacht werden und syntaktische Konflikte durch das Hinzufügen von Elementen. Da wir das symmetrische Mischen betrachtet haben, folgt daraus auch die Eigenschaft, daß die Tabellen selbst symmetrisch sind, bezogen auf ihre Diagonale.

Die Untersuchung der möglichen Konflikte beim Mischen zeigt weiter, daß viele Änderungen, die an zwei Versionen eines Diagramms gemacht wurden, sich automatisch mischen lassen, ohne daß Konflikte auftreten. Die notwendige Voraussetzung hierfür ist das feinkörnige Editier-Metamodell der UML-Diagramme.

Syntaktische Konflikte liegen in den Konsistenzkriterien der modellierten Diagramme begründet. Die Überprüfung der syntaktischen Konsistenz der modellierten Dokumente ist eine umfassende Aufgabe, die nur durch zusätzliche Werkzeuge sichergestellt werden kann. Diese Werkzeuge sind auch für unversionierte Dokumente sinnvoll nutzbar. Daher sind diese Analysatoren unabhängig vom Mischen zu betrachten. Die Konsistenzsicherung der Diagramme ist eine eigenständige Forschungsfrage, die hier nur kurz für syntaktische Konflikte skizziert wurde. Eine umfassende Bearbeitung würde den Rahmen dieser Arbeit übersteigen. Davon bleibt jedoch die Sicherstellung der Konsistenz des Editiermodells unberührt. Das Mischwerkzeug muß sicherstellen, daß das Editiermodell der Mischversion in sich konsistent ist. Setzt man voraus, daß beide Basisversionen ein konsistentes Editiermodell besitzen und daß alle Funktionen des Mischwerkzeugs die Konsistenz nicht verletzen, so erhält man auch wieder ein konsistentes Editiermodell.

5.4.2 Das Mischkonzept für Software-Dokumente

Bei der Konzeption eines Mischverfahrens ist zu entscheiden, wie die Konflikte zu lösen sind. Bei strukturierten Dokumenten könnte man ein Mischkonzept nutzen wie in COOP/Orm [6] (siehe Abschnitt 2.2.3.1). Bei der Traversierung der Syntaxbäume[10] beider Versionen muß der Werkzeuganwender die Konflikte sofort lösen. Der Vorteil ist, daß man die aufgetretenen Konflikte nicht verwalten muß und nach Abschluß des Mischens die Mischversion vollständig vorliegt. Der entscheidende Nachteil ist jedoch, daß man nur die Knoten des Syntaxbaums betrachten kann, die bereits gemischt wurden. Konflikte können jedoch auch zwischen Knoten auftreten, die auf unterschiedlichen Ebenen des Syntaxbaums liegen, wie im vorherigen Abschnitt diskutiert. Desweiteren erhält der Werkzeuganwender keinen Überblick aller aufgetretenen Konflikte, der u.U. für einige Mischentscheidungen notwendig wäre.
Aus diesen Gründen ist es sinnvoll, ein erweitertes Vereinigungsdokument als Grundlage der manuellen Mischentscheidungen zu nutzen. In diesem erweiterten Vereinigungsdokument sind bereits alle automatisch entscheidbaren Konflikte gelöst, so daß der Werkzeuganwender seine Mischentscheidungen auf Basis dieser bereits teilweise gemischten Version treffen kann.

Abhängig von der Größe der zu mischenden Diagrammversionen und von der Anzahl der Differenzen zwischen ihnen kann die Anzahl der manuell zu lösenden Konflikte sehr groß sein. Daher sollte das Mischkonzept eine Möglichkeit vorsehen, so viele Konflikte wie möglich mit so wenigen Entscheidungen wie nötig lösen zu können. Der im *Object Merging Framework* [170] vorgeschlagene Ansatz, Tabellen zu erstellen, anhand derer Mischentscheidungen getroffen werden (vgl. Abschnitt 2.2.3.2), ist aufgrund der komplexen Metamodelle der UML-Diagramme nicht praktikabel. Man müßte für alle in den Metamodellen verwendeten Typen im voraus angeben, wie ein Konflikt zu lösen ist. Von dem Umfang dieser Konfigurationsdaten einmal abgesehen, hätte der Werkzeuganwender vor der Beendigung der Mischfunktion keine exakte Vorstellung vom Ergebnis. Diese Funktionalität muß demzufolge in den Mischwerkzeugen selbst und nicht in der Mischfunktion realisiert werden.
Aus diesen Vorüberlegungen folgt, daß ein 3-stufiges Mischkonzept sinnvoll ist:

1. *Mischfunktion erstellt die Pre-Mischversion.* Die Pre-Mischversion ist ein erweitertes Vereinigungsdiagramm. Im Gegensatz zu dem Vereinigungsdiagramm sind nach den Regeln des 3-Wege-Mischens alle automatisch entscheidbaren Konflikte vorläufig gelöst. Diese automatischen Mischentscheidungen sind manuell änderbar. Die Konflikte und vorläufigen Mischentscheidungen sind in der Repräsentation als Pre-Misch-Diagramm entsprechend gekennzeichnet (siehe Abschnitt 5.4.3).

2. *Der Werkzeuganwender löst die Konflikte.* Die Pre-Mischversion enthält neben den Konflikten Informationen aus der Versionsverwaltung, anhand derer alle Änderungen (in Konflikt stehende wie auch automatisch gemischte) den Konfigurationen zugeordnet werden können, in denen sie ursprünglich durchgeführt wurden. Mit diesen Informationen lassen sich mehrere Konflikte mit einer Entscheidung lösen (siehe Abschnitt 5.4.4).

   Teilweise kann es notwendig sein, bereits getroffene Misch-Entscheidungen zurückzunehmen und die andere Alternative zu wählen. Aus diesem Grund sind alle Mischentscheidungen (manuelle und automatisch getroffene) in diesem Schritt noch verfügbar und im Editiermodell des Vereinigungsdiagramms gespeichert.

3. *Die Mischversion des Diagramms wird erzeugt.* Anhand aller an der Pre-Mischversion gespeicherten Mischentscheidungen wird nach dem Lösen der Konflikte die Mischversion erzeugt (siehe Abschnitt 6.3.2).

[10] Die Spannbäume auf den Editiermodellen.

5.4.3 Anzeige der Pre-Mischversion

Die Pre-Mischversion eines Diagramms stellt einen Zwischenschritt beim Mischen von zwei Versionen eines Diagramms dar. Von der Semantik unterscheidet es sich vom Vereinigungsdiagramm, welches zur Anzeige von Differenzen genutzt wird, dahingehend, daß für konfliktfreie Änderungen bereits automatisch eine manuell änderbare Mischentscheidung getroffen wurde. Diese und die Konflikte sind in der Pre-Mischversion jeweils in einer eignen Farbe gekennzeichnet.
Durch die Notwendigkeit, die existierenden Konflikte zu lösen, sollten diese für den Werkzeuganwender deutlich erkennbar sein. Zu deren Kennzeichnung wird für die Pre-Mischversion, im Vergleich zum Vereinigungsdiagramm, eine erweiterte Darstellung eingesetzt. Alle automatischen Mischentscheidungen sind blau gezeichnet. Das beruht auf der Überlegung, daß die bereits getroffenen Entscheidungen für einen Werkzeuganwender erkennbar sein sollten, jedoch nicht so auffällig sein dürfen wie die zu lösenden Konflikte. Die für die Darstellung der Konflikte verwendeten Farben entsprechen denen der Differenzdarstellung. Alle Änderungen, die auf eine Diagrammversion zurückzuführen sind, werden daher in derselben Farbe dargestellt, entweder in rot oder in grün.
Ein weiterer Unterschied zum Vereinigungsdiagramm besteht darin, daß mehr Informationen visualisiert werden müssen. Durch die Nutzung des 3-Wege-Mischverfahrens kann man zwischen hinzugefügten und gelöschten Diagrammelementen unterscheiden. Für automatisch getroffene Mischentscheidungen gilt daher zusätzlich, daß gelöschte Diagrammelemente blau durchgestrichen gezeichnet werden, im Gegensatz zu hinzugefügten Elementen, die ausschließlich an deren blauer Darstellung erkennbar sind.
Bei Konflikten gilt es die unterschiedlichen Arten[11] von Konflikten zu berücksichtigen. Deren Darstellung ist i.w. an die Darstellung von Differenzen im Vereinigungsdiagramm angelehnt:

- *Löschkonflikt*: Das gelöschte Diagrammelement wird durchgestrichen gezeichnet. Die Farbe der Linien entspricht dabei der Farbe, die zur Darstellung der Änderungen der Version verwendet wird, in der das Element gelöscht wurde. Die in Konflikt stehende Änderung wird in der anderen Farbe eingezeichnet.

- *gleichartige Änderung*: Die Darstellung beruht auf der Differenzdarstellung in dem korrespondierendem Vereinigungsdiagramm. Beide Varianten werden gleichzeitig angezeigt.

- *Umordnen einer Liste von Komponenten*: Die Listenelemente werden in der ursprünglichen Reihenfolge eingezeichnet. Die neue Reihenfolge und die in Konflikt stehende Änderungen werden entsprechend der in Abschnitt 5.2.1 beschriebenen Methode dargestellt.

Um die Aufmerksamkeit besser auf die Konflikte zu lenken, können diese durch einen farbigen Hintergrund und ein ergänzendes Hinweissymbol hervorgehoben werden. Sinnvoll ist das insbesondere bei Konflikten, die nicht direkt erkennbar sind. Die Farbe des Hinweissymbols orientiert sich dabei an der Farbe, die zur Darstellung der Änderungen einer bestimmten Version verwendet wird. Das Hinweissymbol kann auch dazu genutzt werden, um den Werkzeuganwender auf Konflikte aufmerksam zu machen, die in der aktuellen Anzeige des Werkzeugs nicht sichtbar sind. Ein Beispiel hierfür könnte ein Mischkonflikt an einer ergänzenden Spezifikation einer Klasse sein.

Diese beschriebenen Darstellungsarten für Konflikte sind ausreichend für alle Diagrammtypen der UML, da die Arten von Konflikten anhand der Liste der möglichen Änderungen an den UML-Diagrammen erarbeitet wurde. Für einzelne Diagrammtypen kann eine geringfügige Anpassung der Darstellung notwendig sein.

[11] Das sind gemäß Abschnitt 5.4.1 alle möglichen Konfliktarten.

Ein Beispiel. In Abbildung 5.15 auf der nächsten Seite ist ein Beispiel dargestellt, welches die Darstellung einer Pre-Mischversion verdeutlicht. Abbildung 5.15(a) zeigt die Basisversion eines (vereinfachten) Diagramms zur Modellierung von HTML-Dokumenten, das in der ersten Variante (Abbildung 5.15(b)) um weitere Klassen zum Export der modellierten HTML-Dokumente erweitert wurde. Desweiteren sind die vorhandenen Klassen um einige Methoden ergänzt worden und die Klasse `HTMLDocElement` wurde umbenannt. In der zweiten Variante wurde eine Anbindung an ein CGI-Skript (die Klasse `Application`) realisiert und in einer weiteren Werkzeugsitzung wurden vorhandene Klassen umbenannt, die Klasse `HTMLCombo` gelöscht und die Klasse `Form` um ein Attribut erweitert sowie die Methodenliste umgeordnet.
Die resultierende Pre-Mischversion der Diagramme zeigt Abbildung 5.15(d). Die meisten Änderungen, die in den beiden Varianten durchgeführt wurden, konnten automatisch gemischt werden. Konflikte traten lediglich bei der Umbenennung einer Klasse, dem Löschen der Klasse `HTMLCombo`, und dem Umordnen der Methodenliste auf.

Erweiterungen. Zur Verbesserung der Übersichtlichkeit ist folgende erweiterte Funktionalität möglich:

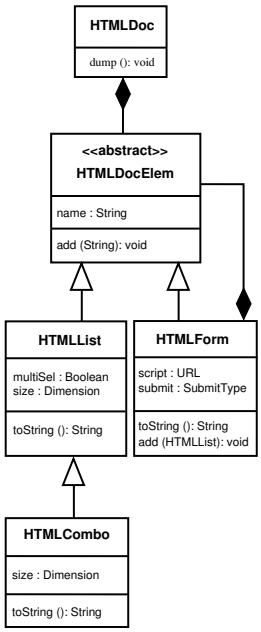
- *ausblenden der autom. gelöschten Diagrammelemente*: Alle Diagrammelemente, die von der Mischfunktion als gelöscht markiert wurden, also im Diagramm durchgestrichen angezeigt werden, können ausgeblendet werden. Das Mischwerkzeug kann eine Funktion anbieten, die diese Diagrammelemente bei Bedarf ausblendet oder wieder einblendet.

- *automatisch hinzugefügte Diagrammelemente schwarz zeichnen*: Wenn der Werkzeuganwender alle automatisch hinzugefügten Diagrammelemente akzeptiert, können diese schwarz dargestellt werden. Bei Bedarf können diese Elemente auch wieder blau gezeichnet werden, falls der Werkzeuganwender einige dieser Entscheidungen nachträglich ändern möchte.

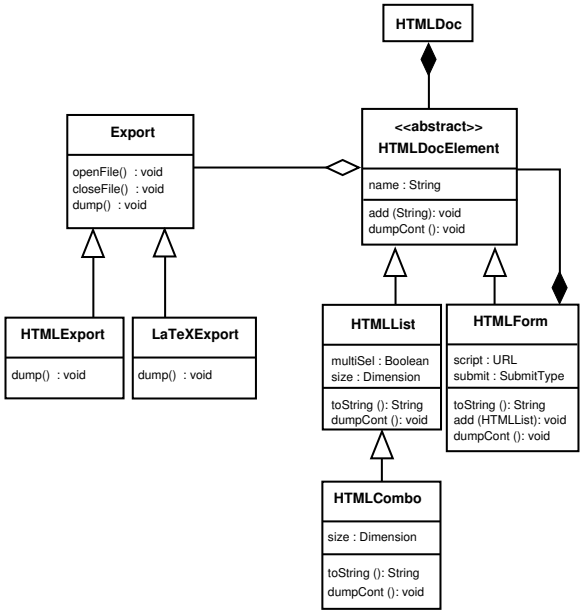
- *farbige Kennzeichnung automatisch getroffener Mischentscheidungen*: Die Darstellung der automatisch gemischten Änderungen an Diagrammelementen in blauer Farbe kann u.U. nicht ausreichend sein, um das Ergebnis zu beurteilen und um zu entscheiden, ob weitere Änderungen notwendig sind. Beispielsweise wurden alle Subtypen der Klasse `HTMLDocElement` in Abbildung 5.15(b) um die Methode `dumpCont` erweitert. Da in der zweiten Variante des Diagramms (Abbildung 5.15(c)) diese Klasse einen weiteren Subtyp (`Input`) erhalten hat, der jedoch nicht diese Methode enthält, ist die Pre-Mischversion des Diagramms semantisch inkonsistent. Diese Inkonsistenz läßt sich leichter erkennen, wenn ausschließlich die Änderungen angezeigt werden, die auf der ersten Variante basieren, wie es in Abbildung 5.16 dargestellt ist. In diesem Fall sind alle Änderungen der zweiten Variante blau und Konflikte grau gefärbt.

- *gruppierte Darstellung der Konflikte*: Vergleichbar mit der gruppierten Darstellung von Differenzen im Vereinigungsdiagramm kann man auch die Konflikte anhand der Konfigurationen gruppieren, in der die Änderungen durchgeführt wurden. Hier ist jedoch zu beachten, daß Konflikte auf zwei verschiedene Konfigurationen zurückzuführen sind, da die in Konflikt stehenden Änderungen in zwei verschiedenen Konfigurationszweigen durchgeführt wurden. Die Gruppierung muß daher auf beiden Konfigurationen basieren.

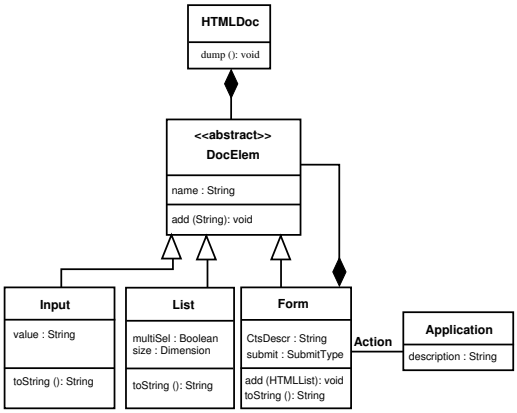
- *Konflikt-Liste*: Zusätzlich zur Darstellung als Diagramm erscheint eine Liste aller beim Mischen aufgetretenen Konflikte sinnvoll. Die Konflikte könnte man darin gruppiert anzeigen. Eine mögliche Gruppierung ist:



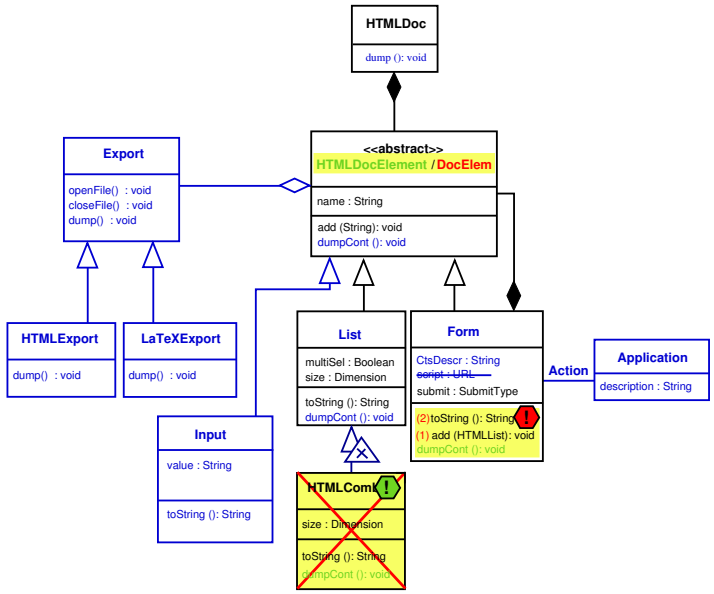
(a) Basisversion



(b) Variante 1: Erweiterung um Exportfunktionalität



(c) Variante 2: Anbindung an eine Applikation



(d) Automatisch gemischte Version mit Konflikten

Abbildung 5.15: Beispiel für die Entwicklung eines Klassendiagramms mit Pre-Mischversion

- automatisch gelöscht
- automatisch hinzugefügt
- manuell gelöste Konflikte
- ungelöste Konflikte

Eine weitere Gruppierung anhand der Konfliktarten ist ergänzend möglich.

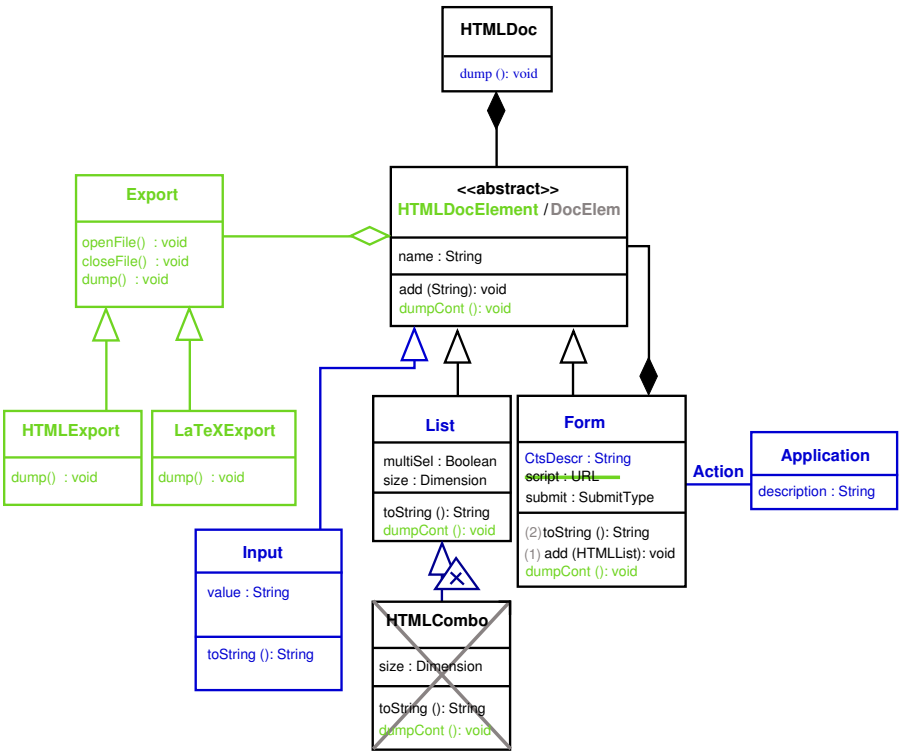


Abbildung 5.16: Beispiel für Erweiterungen

- Zusatzdaten anzeigen: Zu jeder Änderung und zu jedem Konflikt sind die Konfigurationen bekannt, in der die Änderungen ursprünglich durchgeführt wurden. An den Konfigurationen sind ggf. Änderungskommentare vermerkt. Diese können ebenfalls angezeigt werden, um dem Werkzeuganwendern ergänzende Informationen bieten zu können, die es erleichtern, semantische Konflikte zu finden (im Beispiel: Erweiterung der Klassen um eine Methode).

5.4.4 Lösen der Konflikte in Werkzeugen

Aufbauend auf der Darstellung der Pre-Mischversion eines Diagramms können die nicht automatisch entscheidbaren Konflikte gelöst werden. Diese müssen durch den Werkzeuganwender manuell entschieden werden. Hierzu bieten sich drei Vorgehensweisen an:

1. *Konflikte einzeln lösen*: Bei einer geringen Anzahl an Konflikten kann man diese einzeln betrachten und sequentiell lösen. Hierfür muß der Werkzeuganwender die gewünschte Variante zur Lösung des Konflikts auswählen.

2. *alle Konflikte zu Gunsten einer Variante lösen*: Einige konventionelle Werkzeuge zum Mischen von Texten bieten die Möglichkeit, einer der beiden zu mischenden Varianten eine höhere Gewichtung zu geben. Dadurch kann die Mischfunktion für alle Änderungen eine automatische Entscheidung treffen. Bei dieser Lösung sind zwei Methoden möglich:

   (a) Das Mischwerkzeug wählt für alle Differenzen die Änderung der höher gewichteten Variante. Der Anwender hat dann die Möglichkeit einzelne Entscheidungen abzuändern.

(b) Alle anhand eines 3-Wege-Mischalgorithmus entscheidbaren Differenzen werden automatisch gewählt. Die verbleibenden nicht automatisch entscheidbaren Konflikte werden zugunsten der höher gewichteten Variante ausgewählt.

Diese Lösung eignet sich nur für Fälle, in denen alle Änderungen einer Variante in der Mischversion enthalten sein müssen und nur in Ausnahmefällen die Änderung der niedriger gewichteten Variante in die Mischversion Eingang finden sollen. Für diese Ausnahmen ist es jedoch notwendig, die automatisch getroffenen Entscheidungen abändern zu können (siehe Abschnitt 5.4.4.1).

3. *eine Gruppe von Konflikten gemeinsam lösen*: Einer Variante des Dokumentes eine höhere Gewichtung beim Mischen zu geben, ist in vielen Fällen nicht das Mittel der Wahl, da zu viele Änderungen auf die falsche Art entschieden werden und somit die Wahrscheinlichkeit hoch ist, eine falsche Mischentscheidung zu übersehen.

   Sinnvoll ist eine feinkörnige Möglichkeit, mehrere Konflikte gleichzeitig zu lösen. Hierfür bietet es sich an, die durch die Versionsverwaltung gelieferten Informationen zu nutzen.

   Unter der Annahme, daß in einer Werkzeugsitzung logisch zusammengehörige Änderungen durchgeführt wurden, sind diese in einzelnen Konfigurationen zusammengefaßt. Wenn mehrere Konflikte auf Änderungen, die denselben Konfigurationen zugeordnet sind, basieren, lassen sich diese auch gemeinsam lösen. Hierfür wird dann nicht eine Diagramm-Variante höher gewichtet, sondern eine Konfiguration, die an mehreren Konflikten beteiligt ist.

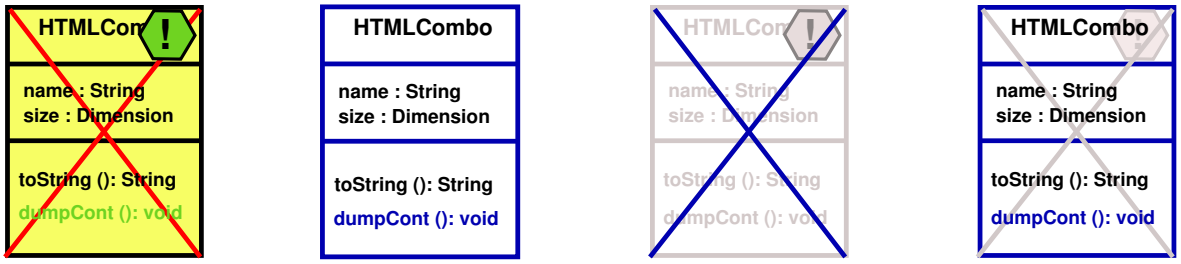
   Diese Art der Lösung von Konflikten kann mit der gruppierten Anzeige von Konflikten kombiniert werden, so daß nur Konflikte farblich markiert sind, die auf gemeinsame Konfigurationen zurückführbar sind.

4. *Konflikte kooperativ lösen*: Beim Mischen von Varianten eines umfangreichen Dokumentes können sehr viele Konflikte auftreten. Diese können auf die Änderungen einer großen Gruppe von Entwicklern zurückzuführen sein. Aus dieser Sicht wäre es vorteilhaft, wenn alle beteiligten Entwickler an der Lösung der Konflikte kooperativ mitwirken. Wenn die Pre-Mischversion unter der Kontrolle des in Kapitel 3 vorgestellten Versionsverwaltungskonzeptes steht und als persistente Version im OMS gespeichert ist (siehe Abschnitt 6.2), ist die Kooperation der Entwickler möglich (siehe Abschnitt 3.3.2).

5.4.4.1 Mischentscheidungen ändern

Es gibt mehrere Gründe, warum einmal getroffenen Mischentscheidungen geändert werden müssen. Beispielsweise kann eine Variante/Konfiguration höher gewichtet worden sein als die andere oder das Lösen eines Konfliktes erfordert eine andere Entscheidung bei einem bereits gelösten Konflikt. Für diese Fälle muß das Mischwerkzeug es erlauben, bereits getroffene Mischentscheidungen abzuändern. Dabei sollte es unerheblich sein, ob die Mischentscheidung automatisch oder manuell getroffen wurde.
Eine notwendige Voraussetzung hierfür ist, daß die Mischentscheidungen noch nicht sofort umgesetzt wurden, sondern so lange verzögert werden, bis alle Konflikte gelöst sind, was durch das drei-stufige Mischkonzept gegeben ist. Eine weitere Voraussetzung ist eine flexible Funktionalität hinsichtlich der angezeigten Konflikte und deren Lösungen. Erweitert man die in Abschnitt 5.4.3 beschriebene Darstellung der Konflikte um die Möglichkeit, alle bereits gelösten Konflikte wieder darzustellen, so können diese auch in der anderen Form gelöst werden. Um diese von den noch zu lösenden Konflikten unterscheiden zu können, sollten sie eine andere

farbliche Kennzeichnung erhalten. Hier bietet sich die Darstellung für die automatisch gelösten Konflikte an. Zu berücksichtigen ist jedoch, daß man die gewählte Lösung eindeutig erkennen muß, insbesondere bei Löschkonflikten ist das relevant.
Abbildung 5.17 zeigt beispielhaft, wie ein Löschkonflikt einer Klasse (hier die Klasse `HTMLCombo` aus Abbildung 5.15(d)) dargestellt werden kann. Hier sind der ungelöste Konflikt (Abbildung 5.17(a)), der gelöste Konflikt durch Auswahl der Variante 1 aus Abbildung 5.15(d) (Abbildung 5.17(b)) und die zur Änderung der Konfliktlösung benötigte Darstellung (Abbildung 5.17(c) und 5.17(d)) angegeben. Bei letzterer ist die gewählte Variante blau und die nicht gewählte Variante grau gezeichnet.



(a) ungelöster Konflikt

(b) gelöster Konflikt

(c) Anzeige des früheren Konflikts nach dem Löschen

(d) Anzeige des früheren Konflikts nach Wahl der Änderung

Abbildung 5.17: Darstellungsarten eines Konflikts

Die Darstellung der anderen Konflikte kann analog geschehen. Bei Konflikten, die auf gleichartigen Änderungen an einem Diagrammelement basieren, kann die gewählte Variante blau und die nicht gewählte Variante in der der Diagrammversion zugehörigen Farbe gezeichnet werden. Entsprechendes gilt für Konflikte, die auf dem Umordnen einer Liste beruhen.

5.5 Zusammenfassung

In diesem Kapitel haben wir die einzelnen UML-Diagramme betrachtet und auf die möglichen Arten von Änderungen hin untersucht. Im wesentlichen handelt es sich dabei um neue/gelöschte Diagramm-Knoten bzw. Komponenten, um neue/gelöschte Beziehungen zwischen Diagramm-Knoten, um Wertänderungen an Diagrammelementen wie z. B. geänderte Bezeichner und um das Umordnen von Listen von Komponenten. Anhand der Änderungen haben wir die möglichen Differenzen zwischen zwei Versionen eines Diagramms herausgearbeitet und einen Vorschlag gemacht, wie diese visualisiert werden können. Der Grundgedanke dabei ist, beide zu vergleichenden Versionen in einem Diagramm darzustellen und dabei die gemeinsamen Diagrammelemente schwarz sowie alle ausschließlich zu einer Version gehörenden Diagrammelemente in Rot respektive Grün zu zeichnen.
Im Vergleich mit existierenden Werkzeugen zur Bestimmung von Differenzen müssen bei UML-Diagrammen mehrere Abstraktionsebenen der Dokumente berücksichtigt werden. Die Bestimmung der Differenzen muß auf Basis des Editiermodells stattfinden, während zur Anzeige das Diagramm die geeignete Darstellungsform ist. Existierende Differenzwerkzeuge beschränken sich auf eine Abstraktionsebene. Für strukturierte Dokumente in einem OMS, für XML-Dokumente oder UML-Diagramme ist das die Repräsentation als Editiermodell (siehe Abschnitt 2.2.2.2). Der ModelIntegrator ist ein Beispiel hierfür.

Auf die möglichen Änderungen aufbauend, haben wir untersucht, welche Konflikte beim Mischen von zwei Varianten eines UML-Diagramms auftreten können. Bei den Konflikten kann man drei unterschiedliche Arten unterscheiden: (1) Löschkonflikte (z. B. Diagrammelement gelöscht vs. Komponente hinzugefügt), (2) Umordnen von Listen (z. B. Attribut-Liste umgeordnet vs. neues Attribut hinzugefügt) und (3) gleichartige Änderungen an einem Diagrammelement, wie z. B. das Ändern eines Bezeichners. Die Darstellung der Pre-Mischversion ist dabei an die Differenzdarstellung angelehnt.
Das Mischen selber sollte in drei Stufen erfolgen. Als erstes wird ein vorläufiges Mischdiagramm erstellt, in der alle automatisch entscheidbaren Änderungen übernommen und Konflikte markiert werden, die die Entwickler im zweiten Schritt unter Ausnutzung von Versionsinformationen lösen. Nach dem Lösen aller Konflikte wird die endgültige Mischversion durch das Werkzeug generiert.
Im Gegensatz zu existierenden Mischwerkzeugen ist es durch das Mischen in drei Stufen möglich, bereits getroffene Konfliktlösungen zu jedem Zeitpunkt während des zweiten Schrittes zurückzunehmen und die jeweils andere Lösung auszuwählen. Durch die Integration mit der Versionsverwaltung können auch mehrere Konflikte gemeinsam gelöst werden.

Kapitel 6

Differenzbestimmung und Mischen im Metamodell

Im vorigen Kapitel haben wir Fragestellungen betrachtet, die die Differenzanzeige und das Mischen von UML-Diagrammen betreffen, ohne auf das Editiermodell näher einzugehen. Das holen wir in diesem Kapitel nach. Das Kapitel beginnen wir mit einer Betrachtung, wie sich die Differenzen und Konflikte im Editiermodell widerspiegeln (Abschnitt 6.1). Eine Beobachtung ist, daß eine einzige Differenz im Diagramm einer Vielzahl an Differenzen im Editiermodell entsprechen kann. Die Algorithmen zur Berechnung der Differenzen und zum Mischen von Versionen beschreiben wir in Abschnitt 6.2. Im Gegensatz zu konventionellen Differenz- und Mischwerkzeugen werden die Differenzen nicht durch eigenständige Werkzeuge berechnet und gemischt, sondern durch eine separate Funktion, die ein persistentes Differenz- bzw. Pre-Mischmodell anlegt (Abschnitt 6.3). Der Vorteil liegt einerseits in der leichten Realisierbarkeit der Anzeigewerkzeuge durch die OMS-orientierte Werkzeugarchitektur, andererseits in der Möglichkeit, Konflikte kooperativ zu lösen. Das persistente Pre-Mischmodell muß zum Abschluß des Mischvorgangs noch in das endgültige Mischmodell überführt werden (Abschnitt 6.3.2).

6.1 Differenzen und Konflikte

In Abschnitt 5.2 haben wir die möglichen Typen von Änderungen an den UML-Diagrammen erarbeitet und als Ergebnis eine Liste mit Änderungen erhalten. Diese Liste beschreibt die Änderungen an der graphischen Darstellung der Diagramme, jedoch nicht, wie sich diese Änderungen im Editiermodell widerspiegeln. Das holen wir in diesem Abschnitt nach. Diese Betrachtung ist angelehnt an die Modellierung der UML-Diagramme in H-PCTE, sie läßt sich jedoch auf das Metamodell der UML übertragen. Durch die Existenz von attributierten Beziehungen in H-PCTE gibt es im Vergleich zur UML-Spezifikation bei der Modellierung von Beziehungen einige Unterschiede, die wir an den entsprechenden Stellen kurz skizzieren.

Die Änderungsliste in Abschnitt 5.2 haben wir anhand der Änderungen an den Diagrammelementen sortiert. Betrachtet man die Änderungen an den Diagrammelementen und vergleicht sie mit den korrespondierenden Änderungen am Syntaxbaum[1], so fällt auf, daß unterschiedliche Änderungen an den Diagrammen zu vergleichbaren Änderungen an den Syntaxbäumen führen. Daher wählen wir hier eine andere Sortierung, die an den möglichen Änderungen an den Syntaxbäumen (neue, gelöschte oder verschobene Knoten sowie geänderte Attribute und

[1] Wenn wir im folgenden den Begriff *Syntaxbaum* verwenden, verstehen wir darunter den Spannbaum des Editiermodells, der durch die Komponentenbeziehungen aufgespannt wird und *nicht* den abstrakten Syntaxbaum.

referenzierte Knoten) angelehnt ist. Zu den jeweiligen Änderungen im Syntaxbaum geben wir die Liste von korrespondierenden Änderungen im Diagramm an.

- neue Knoten im Syntaxbaum:

  Korrespondierende Änderungen im Diagramm:

  - neue Diagramm-Knoten
  - neue Komponenten in Listen und Mengen eines Diagramm-Knotens

  Das Anlegen von neuen Diagramm-Knoten oder Komponenten führt auch zum Anlegen von neuen Knoten im Syntaxbaum. Da ein Diagrammelement, z. B. eine Klasse oder eine Methode, durch mehr als einen Knoten im Syntaxbaum modelliert wird, ist oft nicht nur ein einzelner Knoten, sondern ein Teilbaum hinzugefügt worden.

  Die einzelnen Änderungen an den Diagrammen unterscheiden sich im Syntaxbaum nur durch die Typen der neu angelegten Syntaxbaum-Knoten und durch die Größe der Teilbäume. Der Teilbaum, der eine Klasse repräsentiert, besitzt i.d.R. mehr Ebenen, als der Teilbaum, der eine Methode repräsentiert.

  Die Aussagen gelten für die Modellierung in H-PCTE wie auch für eine Modellierung, die zur UML-Spezifikation [181] konform ist.

- gelöschte Knoten im Syntaxbaum:

  Korrespondierende Änderungen im Diagramm:

  - gelöschte Diagramm-Knoten
  - gelöschte Komponenten aus Listen und Mengen eines Diagramm-Knotens

  Hier gilt das für neue Knoten im Syntaxbaum gesagte, nur daß die Syntaxbaum-Knoten resp. Teilbäume nicht hinzugefügt, sondern gelöscht wurden.

- modifizierte Attribute:

  Korrespondierende Änderungen im Diagramm:

  - atomare Wertänderungen an Diagramm-Knoten
  - Umordnen einer Komponenten-Liste eines Diagramm-Knotens

  Die Bezeichner der Diagrammelemente, wie z. B. der Klassenname oder weitere textuelle und numerische Werte, sind als Attribute der Syntaxbaum-Knoten realisiert. Geänderte Bezeichner wirken sich daher als geänderte Attribute eines Knotens im Syntaxbaum aus, sowohl in H-PCTE als auch in einer UML-Spezifikation konformen Modellierung.

  Die Ordnung einer Liste läßt sich auf unterschiedliche Arten realisieren. In H-PCTE wurde hierfür ein Link-Attribut am Link zwischen Diagrammelement und Komponenten (z. B. zwischen Klasse und Methode) verwendet, welches die Position innerhalb der Liste angibt. Das Umordnen einer Liste führt also zu Änderungen an dem Positionsattribut der Links.

  Die UML-Spezifikation definiert keine konkrete Realisierung bis auf die Tatsache, daß eine Ordnung existiert. Die konkrete Realisierung bleibt dem Werkzeugentwickler überlassen. Da i.d.R. die wenigsten Werkzeuge auf einem Datenmodell mit attributierten Beziehungen aufsetzen, müssen diese eine andere Lösung wählen, z. B. eine Verwaltung der Komponenten mit expliziter Ordnung. Beispiele sind Arrays oder Listen, abhängig von der gewählten Programmiersprache und von den verfügbaren Bibliotheken. In einer XMI-Datei [180]

entspricht die Reihenfolge der XML-Elemente der Reihenfolge der Komponenten eines Diagrammelementes.

- verschieben von Teilbäumen:

  Korrespondierende Änderungen im Diagramm:

  – Verschieben von Komponenten zwischen Diagramm-Knoten

  Das Verschieben von Komponenten eines Diagrammelementes führt zu verschobenen Teilbäumen innerhalb des Editiermodells. Will man bei der Differenzberechnung auch verschobene Teilbäume erkennen, so ist eine einfache Traversierung nicht ausreichend, siehe Abschnitt 6.2. Dieses Verhalten beruht auf der Komponentenhierarchie im Editier-Metamodell der Diagramme. Die Aussagen gelten für die Modellierung in H-PCTE wie auch eine Modellierung, die zur UML-Spezifikation [181] konform ist.

- ändern von referenzierten Syntaxbaum-Knoten:

  Korrespondierende Änderungen im Diagramm:

  – neue Beziehungen zwischen Diagramm-Knoten
  – gelöschte Beziehungen zwischen Diagramm-Knoten
  – referenzierte externe Objekte (z. B. Typ eines Objektes, Parameters, ...)
  – Verschieben von Beziehungen zwischen Diagramm-Knoten

  Die Auswirkungen dieser Änderungen an den Diagrammen sind vom Editier-Metamodell abhängig. Nutzt dieses explizite Beziehungen wie z. B. die Links in H-PCTE, können die genannten Änderungen auf geänderte Links abgebildet werden. Neue oder gelöschte Beziehungen zwischen Diagrammelementen sind daher durch neue oder gelöschte Links im Editiermodell realisiert. Für referenzierte externe Objekte gilt entsprechendes.

  Im Gegensatz hierzu sind alle Beziehungen zwischen Diagrammelementen in der UML-Spezifikation als eigenständige Objekte modelliert, die die Diagrammelemente anhand eindeutiger Objekt-ID referenzieren. Neue oder gelöschte Beziehungen im Diagramm entsprechen daher neuen oder gelöschten Objekten in dem Teil des Editiermodells, der die Beziehungen modelliert. Referenzierte externe Objekte und verschobene Beziehungsendpunkte lassen sich im Editiermodell durch geänderte Objekt-ID realisieren.

6.1.1 Konflikte

Anhand dieser Überlegungen können wir erarbeiten, auf welche Änderungen bestimmte Konflikte beim Mischen von zwei Versionen eines Diagramms zurückzuführen sind. Das Ergebnis aus Abschnitt 5.4.1 ist die Erkenntnis, daß es drei Konfliktklassen gibt:

1. Konflikte durch gleichartige Änderungen
2. Konflikte durch Umordnen von Listen
3. Löschkonflikte

Konflikte, die auf gleichartige Änderungen zurückzuführen sind, lassen sich leicht feststellen. Beide Syntaxbäume müssen sich am selben Knoten, Attribut oder Beziehung unterscheiden:

- atomare Wertänderung: Das selbe Attribut wurde geändert.
- Umordnen einer Liste: Die Positionsattribute bzw.[2] Verwaltungs-Struktur wurde geändert.
- extern referenzierte Elemente austauschen: Der Link bzw. die Objekt-ID wurde geändert.
- Verschieben einer Komponente vom betrachteten Diagramm-Knoten weg: Der Teilbaum besitzt in beiden Versionen unterschiedliche Vaterknoten.
- Verschieben einer Beziehungen vom betrachteten Diagramm-Knoten weg: Der Link bzw. die Objekt-ID wurde geändert.

Das Umordnen einer Komponentenliste verursacht nicht ausschließlich bei einer konkurrierenden Umordnung einen Konflikt, sondern auch, wenn in der anderen Version eine neue Komponente hinzugefügt wurde. In diesem Fall besteht der Konflikt aus geänderten Positionsattributen bzw. Verwaltungsstruktur und einem neuen Knoten.
Löschkonflikte sind immer auf unterschiedliche Änderungen zurückzuführen. Hierbei wurde ein Knoten oder Teilbaum des Syntaxbaums in einer Version gelöscht und in der anderen Version wurde der Teilbaum um einen Knoten erweitert oder ein Attribut geändert.
Die letzten beiden Konfliktklassen machen deutlich, daß ein Konflikt unterschiedliche Teile des Syntaxbaums betreffen kann. Das muß man bei der Implementierung des Misch-Algorithmus berücksichtigen.
Zu beachten ist jedoch, daß einige Konflikte auch Auswirkungen auf ihre Vaterobjekte (bezogen auf den Syntaxbaum) besitzen. Diese Aussage gilt für Konflikte, die auf das Löschen von Teilbäumen zurückzuführen sind. In diesem Fall gibt es an einem Objekt einen Konflikt infolge des Löschens und einer konkurrierenden Änderung. Wenn nicht ausschließlich das Objekt selbst gelöscht wurde, sondern auch sein Vaterobjekt, so steht die konkurrierende Änderung auch mit dem Vaterobjekt in Konflikt und muß dort notiert werden. Das gilt für alle Objekte bis zur Wurzel des gelöschten Teilbaums.

6.1.2 Schlüsselattribute an Beziehungen

Abhängig vom Editier-Metamodell können Beziehungen zwischen Objekten des Editiermodells eigenständige Elemente sein wie in PCTE oder „nur“ einfache Referenzen unter Verwendung von eindeutigen Objekt-Identifizierern wie z. B. in XMI.
Im letzteren Fall brauchen diese nicht weiter betrachtet zu werden, da sie keine eigenen Attribute besitzen können, diese sind bei Bedarf durch die Einführung von Hilfsobjekten zu modellieren, wie in der UML-Spezifikation. Im Gegensatz hierzu besitzen Link-Typen in PCTE zwei Arten von Attributen: Schlüsselattribute und Nicht-Schlüsselattribute. Diese kann man zur Modellierung bestimmter Dokument-Eigenschaften nutzen. Eine wichtige Frage hierbei betrifft die Semantik der Attribute: Dienen die Schlüsselattribute z. B. nur zur Unterscheidung einzelner Links ohne weitere Semantik oder identifizieren sie die Ziel-Objekte. Man kann folgende Fälle unterscheiden:

- Schlüsselattribute besitzen eine Semantik:

[2] Die erste angegebene Änderung bezieht sich auf H-PCTE und die zweite auf eine Modellierung konform zur UML-Spezifikation.

  - für den Link selbst: Das Schlüsselattribut trifft eine Aussage über die Beziehung der beiden Objekte.
  - für das Zielobjekt: Das Schlüsselattribut trifft eine Aussage über das Zielobjekt.

- Nicht-Schlüsselattribute besitzen eine Semantik: Ein Beispiel hierfür sind die Positionsattribute, die die Ordnung auf der Liste der Methoden oder Attribute einer Klasse definieren. In diesem Fall treffen sie eine Aussage über die Zielobjekte.

- Schlüsselattribute besitzen keine Semantik: In diesem Fall dienen sie ausschließlich zur Unterscheidung der einzelnen Links und treffen keine Aussage über den Link selbst oder dessen Zielobjekt.

Besitzen die Schlüsselattribute eines Link-Typs eine Semantik, so muß diese gesondert beim Mischen von Versionen berücksichtigt werden. Das ist abhängig von der Modellierung und dem Anwendungsfall, so daß keine allgemeingültige Aussage getroffen werden kann.
Für den Fall, daß die Schlüsselattribute ausschließlich zur Unterscheidung der einzelnen Links dienen, können beim Mischen von zwei Versionen eines Diagramms neben den oben diskutierten Konflikten weitere Konflikte auftreten. Diese lassen sich jedoch automatisch lösen. Ein Beispiel hierfür ist, daß in den zu mischenden Versionen jeweils ein neuer Link vom selben Objekt ausgehend angelegt wurde, der in beiden Versionen dasselbe Schlüsselattribut, jedoch unterschiedliche Zielobjekte besitzt. Beim Mischen stellt das zwar einen Konflikt dar, der sich jedoch durch die automatische Vergabe eines anderen Schlüssels lösen läßt.

6.2 Der Differenz- und Misch-Algorithmus

Die Berechnung der Differenzen und das Mischen von Versionen eines Dokumentes ist stark von dessen Editier-Metamodell abhängig. Je mehr Informationen man bei der Berechnung berücksichtigen kann, um so höher ist die Qualität des Ergebnisses. Ein Beispiel hierfür ist die Bedeutung der Schlüsselattribute von Beziehungen, wie sie in Abschnitt 6.1.2 diskutiert wurde. Nur mit dieser Kenntnis kann man die Differenzen richtig interpretieren und anschließend mischen. Daher beschränken wir uns in dieser Arbeit auf die Differenzberechnung und das Mischen von UML-Diagrammen, deren Editier-Metamodell eine baumartige Komponentenstruktur definiert[3]. Für andere Dokumenttypen muß der Algorithmus ggf. angepaßt werden. Der auf dem Editiermodell aufgespannte Syntaxbaum ist die Grundlage der Differenzberechnung, die in Algorithmus 1 skizziert wird.
Bei der Berechnung der Differenzen traversiert man die beiden Syntaxbäume mit den Wurzel-Objekten beginnend. Die Traversierung basiert auf dem Algorithmus der Breitensuche in Bäumen. Für alle Objekte einer Ebene eines Baums sucht man im anderen Baum nach korrespondierenden Objekten (siehe Algorithmus 2). Die Zuordnung der Objekte ist davon abhängig, ob es sich bei den beiden Syntaxbäumen um zwei Versionen eines Baums handelt oder nicht. Falls es zwei Versionen sind, kann man die Objekt-Identifizierer für die Zuordnung nutzen, da sich diese bei Änderungen an den Bäumen nicht ändern. Andernfalls kann man nur unter Berücksichtigung der Dokumentsemantik eine Entscheidung treffen (siehe Abschnitt 6.2.2).
Die so gefundenen korrespondierenden Objekte können anschließend verglichen und die Differenzen bestimmt werden. Beide (korrespondierende Objekte und Differenzen) werden in dem

[3]Die dieser Arbeit zugrundeliegende Realisierung basiert zwar auf einem deutlich einfacheren Editiermodell, als wie es durch die UML-Spezifikation vorgegeben wird, die Idee basiert jedoch auf der genannten Voraussetzung.

Algorithmus 1 Berechne die Differenzen zwischen zwei Dokumenten

```
function diff_objects( Object root1, Object root2 ) : Set
Set q, set1, set2;
Set result;
Set shifted1, shifted2; // possibly shifted objects
Tuple x;
Object obj1, obj2;
// insert document root
q.add ( new Tuple( root1, root2 ) );

// traverse object structure
while  q != ∅ do
  for all x in q do
    q.delete( x );
    // find difference of attributes of current object
    result.add( new Object ( diff_attrs( x ) ) );
    // collect child objects
    set1 = x[1].get_child_objects();
    set2 = x[2].get_child_objects();
    q.add( find_matching_objects( set1, set2 ) );
    // Sets of objects only contained in the first or
    // second document or possibly shifted
    shifted1.addAll( set1 );
    shifted2.addAll( set2 );
  end for
  // Notice: q == ∅
  q = find_matching_objects( shifted1, shifted2 )
end while
// Sets of objects only contained in one base document
result.addAll( shifted1 );
result.addAll( shifted2 );
return result;
```

Algorithmus 2 Suche nach korrespondierenden Objekten

```
function find_matching_objects( Set set1, Set set2 ) : Set
Object obj1, obj2;
Set q;
for all obj1 in set1 do
  obj2 = set2.search_obj( obj1 );
  if  obj2 and not q.contains( obj2 )  then
    q.add( new Tuple( obj1, obj2 ) );
    set1.delete( obj1 );
    set2.delete( obj2 );
  end if
end for
return q;
```

Editiermodell des Vereinigungsdiagramms[4] gespeichert, siehe Abschnitt 6.3. Die beiden in den Bäumen gefundenen Objektversionen können sich in den Attributen und in ihren Komponenten unterscheiden.
Die Komponenten eines Objektes können als Teilbäume des Syntaxbaums betrachtet werden, deren Wurzel das jeweilige Komponenten-Objekt ist. Es können auch einige Objekte (mit ihren Komponenten) innerhalb des gesamten Syntaxbaums verschoben worden sein. Diese verschobenen Objekte können nicht bei der ersten Traversierung erkannt werden. Zu diesem Zeitpunkt ist nur bekannt, daß diese Objekte in einem Syntaxbaum vorhanden sind, während sie im anderen an dem Ausgangsobjekt fehlen. Um die verschobenen Objekte finden zu können, müssen alle Objekte, die nach der ersten Traversierung als spezifisch für einen Syntaxbaum erkannt wurden, für jeden Syntaxbaum getrennt in Listen gesammelt werden. Die in den Listen gespeicherten Objekte kann man im nächsten Schritt abgleichen. Enthalten die Listen korrespondierende Objekte, so sind diese innerhalb des Syntaxbaums verschoben worden. Alle Objekte, die nicht zugeordnet werden konnten, sind entweder in dem einem Syntaxbaum neu erzeugt oder in dem anderen gelöscht worden. Diese Unterscheidung ist nur mit Hilfe einer gemeinsamen Vorgängerversion der Syntaxbäume möglich, die bei der Differenzberechnung jedoch nicht betrachtet wird. Das Verschieben ist auch ohne die Vorgängerversion aufgrund der gleichen Objektidentifizierer erkennbar.
In H-PCTE kann zum Verschieben eines Teilbaums innerhalb des Objekt-Graphen die Funktion `Pcte_link_replace` genutzt werden. Wenn diese Funktion zusätzliche Informationen an den verschobenen Objekten speichert, um sie von neu erzeugten Objekten unterscheiden zu können, so kann die Suche nach verschobenen Objekten vereinfacht werden, indem nur noch als verschoben markierte Objekte gesammelt und anschließend abgeglichen werden. Die Anzahl der zu vergleichenden Objekte ist in diesem Fall erheblich geringer.

Das beschriebene Verfahren zur Differenzberechnung läßt sich zu einem 3-Wege-Mischalgorithmus erweitern. Hierfür traversiert man nicht ausschließlich die beiden zu mischenden Versionen, sondern auch die gemeinsame Vorgängerversion. Bei der Zuordnung der Objekte müssen dann die drei gefundenen Objektversionen verglichen und eine Version mit Hinweisen zum Mischen erstellt werden. Die Logik entspricht der des 3-Wege-Mischens, siehe Abschnitt 1.2.4. Aus den gemischten Objektversionen entsteht so schrittweise das *Pre-Misch-Editiermodell* (Abkürzung für: Editiermodell des Pre-Mischdiagramms; s. Abschnitt 5.4.2, 6.3 und [232]).

6.2.1 Layoutdaten

Zu berücksichtigen ist bei der Differenzberechnung, daß die Syntaxbäume auch Layoutdaten enthalten, wie z. B. bei der Modellierung der UML-Diagramme in PI-SET. Wenn die Layoutdaten bei der Differenzberechnung berücksichtigt werden, führt das zu sehr vielen Differenzen [191], die nicht relevant sind. Aus diesem Grund muß der Algorithmus 1 so erweitert werden, daß man die Differenzberechnung auf bestimmte Objekt-, Link- oder Attributtypen beschränken kann. Das läßt sich realisieren, indem man der Funktion zur Differenzberechnung eine Liste dieser Typen als Parameter übergibt, die dann bei der Berechnung der Differenzen unberücksichtigt bleiben.
Beide Diagrammtypen, das Vereinigungsdiagramm wie auch das Pre-Mischdiagramm, sollten aber Layoutdaten besitzen, damit die Diagramme angezeigt werden können. Eine pragmatische Lösung besteht darin, nach der Differenzberechnung bzw. dem Erstellen der Pre-Mischversion die Layoutdaten zu ergänzen. Das ist möglich durch Übernahme der Layoutdaten von einem

[4] Auch wenn dieses Modell nach seiner Erstellung nicht mehr verändert wird, verwenden wir trotzdem die Bezeichnung Editiermodell, da es ein erweitertes Editiermodell ist.

der beiden Basisdiagramme. Die Diagrammelemente aus dem jeweils anderen Basisdiagramm muß der Werkzeuganwender entweder manuell positionieren oder man kann deren Layoutdaten ebenfalls übernehmen. Bei der Übernahme ist jedoch zu berücksichtigen, daß dann einige Diagrammelemente überlappend angezeigt werden können. Die Übernahme der Layoutdaten kann entweder auch durch die Differenzfunktion geschehen oder in einem zweiten Schritt nach der Berechnung, abhängig von der Modellierung.

6.2.2 Suche nach korrespondierenden Objekten

Bei der Bestimmung von Differenzen und beim Mischen ist das Finden von korrespondierenden Objekten in den beiden Syntaxbäumen ein entscheidender Faktor, der die Qualität des Ergebnisses wesentlich bestimmt. Enthalten die Objekte eindeutige Identifizierer, die sich als Schlüssel nutzen lassen, so vereinfacht das die Suche nach den korrespondierenden Objekten erheblich, da die Gleichheit der Identifizierer gleichbedeutend ist mit der Korrespondenz der Objekte. Entsprechende Identifizierer definieren die meisten Editier-Metamodelle, wie in z. B. H-PCTE, aber auch XMI [180].

Jedoch lassen sich die Identifizierer nicht in jedem Fall nutzen, wenn sie z. B. ausschließlich innerhalb eines Syntaxbaums eindeutig sind. Basieren alle Änderungen auf einem Syntaxbaum mit Identifizierern, die durch Änderungen nicht modifiziert werden, so können diese zum Abgleich der verschiedenen Versionen eines Syntaxbaums genutzt werden. Voraussetzung hierfür ist, daß die Identifizierer persistent mit dem Syntaxbaum gespeichert werden. Im Fall von UML-Diagrammen bedeutet das, daß das gesamte Diagramm einschließlich der Identifizierer persistent gespeichert werden muß. Das ist entweder in einer Datei (z. B. XMI) oder im Quelltext innerhalb von Kommentaren möglich.

Sind die Identifizierer jedoch nicht vorhanden, z. B. weil die zu vergleichenden Versionen des Syntaxbaums durch das Parsen von Quelltext und der anschließenden Konvertierung in die UML-Diagramme entstanden sind, so besitzen die Objekte zwar eindeutige Identifizierer, diese unterscheiden sich jedoch zwischen den Versionen der Syntaxbäume, so daß andere Methoden (siehe Abschnitt 2.2.2.2 und [230]) zur Suche der korrespondierenden Objekte eingesetzt werden müssen.

Je mehr Informationen dabei zur Verfügung stehen, um so leichter lassen sich die Objekte identifizieren. Besitzt ein Objekt z. B. viele Attribute und Komponenten, die evtl. auch ihrerseits wieder Komponenten besitzen, so steht eine Vielzahl an nutzbaren Informationen zur Verfügung. Anhand des Vergleichs der Attribute und Komponenten kann man dann auf die Korrespondenz der Objekte selbst schließen. Daraus folgt auch, je weniger Informationen (wenige Attribute oder Komponenten) zur Verfügung stehen, desto unsicherer wird das Ergebnis.

In dieser Arbeit beschränken wir uns auf den Fall von Syntaxbäumen, die unter Versionskontrolle stehen. Daher bleiben die Objekt-Identifizierer erhalten und können verwendet werden. Der andere Fall, daß die Identifizierer nicht genutzt werden, ist ein eigenes und komplexes Forschungsgebiet, welches den Rahmen dieser Arbeit übersteigen würde. Ein möglicher Ansatz wird in [230] beschrieben.

Abhängigkeit der Differenzanzeige. Das Ergebnis der Differenzberechnung ist jedoch davon abhängig, ob Identifizierer oder andere Eigenschaften der Syntaxbäume verwendet wurden, um korrespondierende Objekte zu finden. Basiert die Differenzberechnung auf den Identifizierern, so beeinflussen die auf den Diagrammen ausgeführten Änderungen die Differenzberechnung:

Fall 1: Ein Entwickler löscht eine existierende Klasse in einem Diagramm und erzeugt anschließend eine neue Klasse.

Fall 2: Ein Entwickler ändert den Bezeichner der Klasse, die Bezeichner der Methoden, Attribute und Parameter in der Weise, daß sie exakt so aussieht wie die Klasse aus Fall 1.

Die Syntaxbäume der beiden Diagramme unterscheiden sich dann jedoch in ihren Identifizierern. Im ersten Fall, hat die neue Klasse[5] einen neuen Identifizierer erhalten und im zweiten Fall besitzt die Klasse die bisherigen Identifizierer. Das führt bei der Differenzanzeige zu folgendem Unterschied. Im ersten Fall wird eine Klasse als neu angelegt und eine als gelöschte erkannt. Die Differenzanzeige für den zweiten Fall enthält nur eine Klasse, deren Bezeichner, Methoden und Attribute geändert wurden. Insbesondere bei der Umstrukturierung von Klassenstrukturen, z. B. durch die Einführung einer neuen Vererbungsbeziehung durch Generalisierung, kann diese Art von Modifikationen auftreten.
Nachteilig ist dieses Verhalten in dem Fall, daß die neue Oberklasse viele Methoden und Attribute der alten Klasse erhält und so der ursprünglichen Version der Klasse ähnlicher ist als die Klasse, die aufgrund der Identifizierer von der Differenzfunktion als korrespondierend bestimmt wurde.

6.3 Die Editiermodelle des Vereinigungsdiagramms und des Pre-Misch-Diagramms

Bei der Realisierung der Differenzberechnung ist die Frage zu klären, wie die Differenz- und Mischwerkzeuge die OMS-orientierte Werkzeugarchitektur nutzen können. Eine in diesem Zusammenhang zu klärende Frage ist, ob die Differenzberechnung durch das OMS oder durch die Werkzeuge durchgeführt wird. Die Integration in die Werkzeuge hat den Vorteil, daß diese speziell auf die Erfordernisse der Dokumente und die Anforderungen der Werkzeuge angepaßt werden kann. Die Integration in die Werkzeuge hat jedoch den Nachteil, daß für jedes Werkzeug diese Funktion eigenständig zu realisieren ist. Die Realisierung in H-PCTE hätte den Vorteil, daß die Funktion nur einmal implementiert werden müßte, diese ist dann generisch für verschiedene Dokumenttypen und Arbeitsschemata. Der Nachteil wäre jedoch, daß die Schnittstelle entweder sehr komplex oder nicht allgemein genug wäre.
Die Differenzberechnung wird daher als Bibliothek realisiert, die die Schnittstelle von H-PCTE nutzt und selbst von den Werkzeugen genutzt werden kann. Die benötigten Parameter sind dabei die beiden Dokumente, deren Konfigurationsidentifizierer und eine Liste mit Objekt-, Link-, und Attributtypen, die bei der Differenzberechnung nicht berücksichtigt werden sollen. Das vereint die Vorteile beider oben genannten Lösungen.

Die Trennung der Differenzberechnung von den Werkzeugen wirft die Frage auf, wie die Differenzen den Werkzeugen mitgeteilt werden können. Eine Lösung könnte sein, eine spezielle Schnittstelle anzubieten, die die Werkzeuge zur Abfrage der Differenzen nutzen könnten. Diese Schnittstelle und die Werkzeuge wären jedoch sehr komplex, da mittels der neuen Schnittstelle die Differenzen wie auch die Gemeinsamkeiten beider zu vergleichender Syntaxbäume an die Werkzeuge weitergereicht werden müßten. In diesem Fall läßt sich die OMS-orientierte Werkzeugarchitektur nicht nutzen.
Die OMS-orientierte Werkzeugarchitektur erfordert, daß die Werkzeuge direkt auf den im OMS gespeicherten Daten arbeiten. Die Lösung besteht darin, daß die Differenzfunktion die Unter-

[5] Wenn hier von Klasse gesprochen wird, ist der Teilbaum im Syntaxbaum gemeint, der die Klasse repräsentiert.

schiede nicht an die Werkzeuge direkt liefert, sondern ein Editiermodell des Vereinigungsdiagramms im OMS persistent erzeugt, welches die Gemeinsamkeiten und die Unterschiede der zu vergleichenden Editiermodelle beinhaltet. Dieses Editiermodell des Vereinigungsdiagramms kann dann durch die Werkzeuge eingelesen und angezeigt werden. Die Werkzeuge müssen in diesem Fall nur die zusätzlichen Daten – die Differenzen – interpretieren und anzeigen, ansonsten sind keine Änderungen an den Werkzeugen notwendig. Abhängig von der Architektur der Werkzeuge läßt sich das mit geringem Aufwand realisieren [231, 232].
Der Nachteil dieser Realisierung besteht darin, daß das Editiermodell des Vereinigungsdiagramms persistent im OMS angelegt wird und wieder gelöscht werden muß. Weiterhin sollte das Editiermodell des Vereinigungsdiagramms unversioniert erzeugt werden, damit es ohne die Nutzung spezieller Administrationsfunktionen der Versionsverwaltung entfernt werden kann. Würde das Editiermodell des Vereinigungsdiagramms in den Versionsbaum eingefügt, so würde das Löschen des Editiermodell des Vereinigungsdiagrammss nur zu einer Löschmarkierung führen, es aber nicht wirklich löschen.
Vergleichbare Überlegungen gelten für das Pre-Misch-Editiermodell. Es wäre denkbar, das Pre-Misch-Editiermodell im Versionszweig anzulegen, indem die gemischte Version des Diagramms später abgelegt werden soll. Das hat neben den für das Editiermodell des Vereinigungsdiagramms diskutierten Nachteilen den weiteren Nachteil, daß alle nach dem Mischen nicht mehr benötigten Mischkonflikte ebenfalls im Versionszweig gespeichert wären. Daher wird das Pre-Misch-Editiermodell ebenfalls unversioniert erzeugt.

6.3.1 Das Editiermodell des Vereinigungsdiagramms

Das Editiermodell des Vereinigungsdiagramms unterscheidet sich vom Editiermodell konventioneller Diagramme in einigen Eigenschaften:

- Es basiert auf einer 2-Wege-Differenzberechnung.
- Erweiterungen im Editier-Metamodell: Die Editier-Metamodelle der einzelnen UML-Diagramme müssen so erweitert werden, daß die zusätzlichen Differenzinformationen mitgespeichert werden können. Insbesondere handelt es sich dabei um die Informationen, daß einzelne Objekte des Editiermodells des Vereinigungsdiagramms nur in einer Basisversion vorhanden sind und daß die Werte der Objekt- oder Link-Attribute sich unterscheiden.

  Ferner sind für die gefilterte Anzeige der Differenzen noch die Informationen notwendig, welcher Konfiguration eine bestimmte Änderung, die zu einer Differenz führte, zuzuordnen ist. Die Konfigurationsidentifizierer werden durch das OMS an die Differenzfunktion geliefert.
- Konsistenz. B.dingungen: Die Konsistenz. B.dingungen des Editier-Metamodells des Vereinigungsdiagramms müssen gegenüber dem Editier-Metamodell der Diagramme gelockert werden. Das betrifft die Kardinalitäten der Links. Beispielsweise besitzen Attribute in Klassendiagrammen exakt einen Typ. Wenn dieser durch einen Link auf die entsprechende Klasse modelliert ist, so darf maximal ein Link existieren. Das läßt sich durch die Definition von Kardinalitäten sicherstellen. Wenn der Attributtyp sich jedoch zwischen beiden Diagrammversionen unterscheidet, so müssen im Vereinigungsdiagramm beide Attributtypen gespeichert sein. Es müssen also zwei Links existieren, was jedoch durch die ursprüngliche Kardinalität verhindert wird.

  Das Ändern der Kardinalitäten kann einen nicht zu unterschätzenden Aufwand in einer OMS-orientierten Architektur darstellen, da die Kardinalitäten aller Beziehungen in allen

Arbeitsschemata angepaßt werden müssen. Das führt i.d.R. zu neuen Schemata, die wieder alle Typdefinitionen einschl. der Erweiterungen enthalten. Falls die Kardinalitäten jedoch durch die Werkzeuge und nicht durch das Repository anhand der Definitionen im Schema überprüft werden, so entsteht kein Mehraufwand.

Das Pre-Misch-Editiermodell. Das Pre-Mischdiagramm wird im Pre-Misch-Editiermodell gespeichert. Dieses unterscheidet sich vom Editiermodell des Vereinigungsdiagramms nur darin, daß es durch eine 3-Wege-Mischfunktion entstanden ist und daher zusätzlich die Mischentscheidungen aber auch die Objektidentifizierer der Basisversionen speichert. Die Mischentscheidungen dürfen nicht sofort im Pre-Misch-Editiermodell nachvollzogen werden, da ansonsten der Werkzeuganwender keine Möglichkeit hätte, einmal getroffene Entscheidungen zu ändern.

6.3.2 Erzeugen des Mischmodells

Nachdem beim Mischen alle Konflikte gelöst sind, muß aus dem Pre-Misch-Editiermodell das Editiermodell des gemischten Diagramms (*Misch-Editiermodell*) erzeugt werden, indem alle Mischentscheidungen, die man auch als ein über das Pre-Misch-Editiermodell verteiltes Edit-Skript interpretieren kann, ausgeführt werden.
Die Mischentscheidungen dürfen jedoch nicht auf dem Pre-Misch-Editiermodell selbst ausgeführt werden, da es außerhalb des Versionszweigs liegt, in dem das Mischdokument erzeugt werden soll (Zielversionszweig). Alle Mischentscheidungen, die sich nur auf Objekte beziehen, die bereits in dem Zielversionszweig enthalten sind, können direkt auf den entsprechenden Objekten bzw. ihren Versionen ausgeführt werden. Vor der Ausführung von Mischentscheidungen, die sich auf Objekte beziehen, die ausschließlich in dem Dokument des einzumischenden Versionszweigs enthalten sind, müssen diese Objekte in den Zielversionszweig importiert werden. Hierfür dient eine neue Schnittstelle von H-PCTE (`HPcte_object_and_link_import_version`).

6.4 Zusammenfassung

In diesem Kapitel haben wir betrachtet, wie sich die Differenzen und Konflikte zwischen Versionen eines UML-Diagramms in den korrespondierenden Syntaxbäumen widerspiegeln. Bei den Differenzen handelt es sich im wesentlichen um erzeugte, gelöschte oder verschobene Teilbäume des Editiermodells und geänderte Attribute der Editiermodell-Knoten. Die Konflikte, die beim Mischen von zwei Syntaxbäumen auftreten, betreffen entweder einzelne Knoten, sofern es sich um gleichartige Änderungen handelt, oder auch ganze Teilbäume. Das ist der Fall, wenn in einer Version ein Teilbaum gelöscht und in der anderen Version einzelne Knoten modifiziert wurden.
Handelt es sich bei den zu vergleichenden oder zu mischenden Syntaxbäumen um zwei Versionen eines Baumes, deren Knoten eindeutige Identifizierer besitzen, die an allen Versionen eines Knotens unverändert bleiben, so lassen sich die Differenzen durch eine einfache Traversierung der Syntaxbäume und einen Vergleich der einzelnen Versionen der Knoten feststellen. Durch die in Kapitel 3 beschriebene Versionierung sind diese Voraussetzungen in dieser Arbeit gegeben. Ohne Identifizierer bzw. wenn sich die Identifizierer zwischen den Versionen unterscheiden, muß bei der Bestimmung der Differenzen der Inhalt der Dokumente berücksichtigt werden.
Die Möglichkeit einzelne Instanzen von Typen des Editier-Metamodells bei der Differenzberechnung unberücksichtigt zu lassen, ist ein Unterscheidungsmerkmal im Vergleich zu anderen Lösungen. Ein Anwendungsfall sind die Layoutdaten der Diagramme. Die Differenzberechnung

würde ansonsten zu viele Konflikte liefern. Ein weiterer Unterschied liegt darin, daß die Differenzen bzw. die Mischversion in einem Differenz-/Pre-Misch-Editiermodell gespeichert werden, anstatt sie direkt an die Werkzeuge weiterzureichen. Das bietet den Vorteil, daß die OMS-orientierte Werkzeugarchitektur auch für die Differenz- und Mischwerkzeuge nutzbar ist.
Ein weiterer Vorteil des persistenten Pre-Misch-Editiermodells liegt darin, daß die Anwender des Mischwerkzeugs die volle Kontrolle über alle Mischentscheidungen haben und selbst bereits getroffene Mischentscheidungen wieder abändern können.

Kapitel 7

Zusammenfassung und Ausblick

Die Softwareentwicklung ist heute ohne den Einsatz von Werkzeugen und Modellierungs-Sprachen wie zum Beispiel der UML nicht mehr denkbar. Während der Entwicklung eines Projektes werden insbesondere in den frühen Phasen viele Analyse- und Entwurfsdokumente erstellt, die verschiedene Entwicklungsstadien durchlaufen. Die eingesetzten Softwareentwicklungsprozesse [113, 185, 208] fordern die Wiederholbarkeit und die Nachvollziehbarkeit der Entwicklung. Hierfür müssen einzelne Versionen, die bestimmten Kriterien entsprechen, archiviert werden. Für diese Aufgaben haben sich Versionsverwaltungs-Systeme als gewinnbringend erwiesen. Jedoch unterstützen existierende Systeme die Dokumente der frühen Entwicklungsphasen nur unzureichend, da diese auf textuelle Dokumente und nicht auf Diagramme spezialisiert sind. Durch die umfangreicher werdenden Projekte ist auch die Größe der Entwicklergruppen, die kooperativ an einem Projekt arbeiten, stetig gestiegen. Existierende Werkzeuge sind i.d.R. nur für den Einsatz an einem Arbeitsplatz ausgelegt, so daß einzelne Entwickler die Dokumente nur sequentiell bearbeiten können. Die kooperative Entwicklung wird nicht unterstützt. Parallel entwickelte Versionen eines Dokumentes müssen die Entwickler meist manuell mischen, da nur sehr wenige Werkzeuge das Mischen von Versionen eines UML-Diagramms ermöglichen. Diese nutzen dann das Editiermodell der Dokumente sowohl intern zur Berechnung der Differenzen und zum Mischen als auch zu deren Anzeige. Die Darstellung der Diagramme als Editiermodell ist jedoch eine andere Art der Repräsentation als die graphische Darstellung der Diagramme. Somit müssen die Anwender dieser Werkzeuge zwei Aufgaben lösen. Einerseits die mentale Übersetzung des Editiermodells einschließlich der Differenzen und Konflikte in die graphische Darstellung als Diagramm, andererseits die Interpretation der Differenzen und das Lösen der Konflikte. Eine Unterstützung durch ein Versionsverwaltungs-System kann beide Problemstellungen lösen.

Die konventionellen Versionsverwaltungs-Systeme bieten jedoch nur eine unzureichende Unterstützung für die Dokumente der frühen Phasen der Softwareentwicklung, wie zum Beispiel UML-Diagramme. Diese Dokumente unterscheiden sich von den Dokumenten der späten Phasen insbesondere dadurch, daß letztere nur eine Abstraktionsebene besitzen, die textuelle Repräsentation[1]. Diese nutzen die Werkzeuge intern zur Verwaltung, zur persistenten Speicherung und zur Darstellung, die durch den Anwender bearbeitet wird. Im Gegensatz hierzu liegen UML-Diagramme in bis zu drei unterschiedlichen Repräsentationen vor. Der Anwender nutzt zur Erstellung und Modifizierung die graphische Darstellung, die Werkzeuge arbeiten intern mit einer Repräsentation als Editiermodell und speichern diesen in einem binären oder textuellen Format, wie z. B. XMI.

Zur Versionsverwaltung eignet sich ausschließlich die Repräsentation als Editiermodell, das

[1] Bei genauer Betrachtung gibt es zwar auch bei Texten mehrere Abstraktionsebenen, dieses unterscheiden sich jedoch nur in wenigen Details.

gleiche gilt für die Berechnung von Differenzen und für das Mischen von Versionen. Jedoch ist diese Repräsentation zur Darstellung ungeeignet. Die Anwender erwarten eine Darstellung als Diagramm, die bisher von keinem konventionellen Werkzeug angeboten wird.

7.1 Zusammenfassung

Der Beitrag dieser Arbeit besteht in der Entwicklung eines Versionsverwaltungskonzeptes, welches einerseits die kooperative Entwicklung von UML-Diagrammen ermöglicht, andererseits Werkzeuge zur Differenzanzeige und zum Mischen von Versionen unterstützt. Vorausgesetzt werden die feinkörnige Modellierung der Diagramme und Werkzeuge, die nach dem Prinzip der OMS-orientierten Werkzeugarchitektur arbeiten.

Versionsverwaltung von UML-Diagrammen. Die Versionierung der Dokumente ist an die im Rahmen der Entwicklung anfallenden Aufgaben angelehnt. Die Aufgaben werden durch Entwurfstransaktionen (ETA) modelliert, deren primäre Aufgabe in der Bereitstellung der zur Lösung einer Aufgabe benötigten Versionen der Dokumente besteht. Zur Bearbeitung einer Aufgabe sind i.d.R. mehrere Werkzeugsitzungen notwendig, so daß die ETA nicht an einzelne Werkzeugsitzungen gebunden werden können. Die ETA werden daher persistent im OMS verwaltet.
Jede ETA besitzt ihren eigenen Arbeitsbereich in dem die Dokumente und alle Versionen, die im Rahmen dieser ETA angelegt wurden, verwaltet werden. Die Softwareentwicklungswerkzeuge können über spezielle Schnittstellen auf die ETA und die durch sie verwalteten Dokumente zugreifen.

Die kooperative konkurrierende Bearbeitung einer Diagrammversion durch mehrere Entwickler ist in einer ETA möglich. Die Konsistenz der Diagrammversion wird durch den Einsatz von Werkzeugtransaktionen sichergestellt. Die Grundlage ist die feinkörnige Modellierung und Speicherung der Diagramme in H-PCTE sowie der feinkörnige Sperr- und der Benachrichtigungsmechanismus.

Die Versionierung von feinkörnig modellierten Daten ist manuell nicht möglich. Ein Diagramm besteht aufgrund der feinkörnigen Modellierung aus mehreren hundert Objekten, die einzeln versioniert werden müssen. Durch eine gemeinsame Versionierung würden Informationen verloren gehen, die bei der Differenzberechnung nützlich sind. Daher muß das Anlegen der Objektversionen automatisch geschehen. Das ist durch die Erweiterung der Werkzeugtransaktionen (WTA) möglich. Bei jedem Zugriff fordert die WTA eine Sperre auf dem betroffenem Objekt, Link oder Attribut an. Anhand des Sperrmodus kann entschieden werden, ob ein Lese- oder ein Schreibzugriff durchgeführt werden soll. Handelt es sich um einen Lesezugriff wird keine neue Version benötigt. Soll jedoch eine Änderungen durchgeführt werden, so muß eine neue Version angelegt werden. Es ist ausreichend von jedem Objekt, Link oder Attribut nur eine neue Version innerhalb einer WTA anzulegen.
Die Vielzahl an angelegten Versionen in einer WTA faßt eine Konfiguration – die Arbeitskonfiguration – zusammen. Die in einer WTA zugreifbaren Versionen werden beim Start der WTA durch die Auswahl einer existierenden Konfiguration – der Basiskonfiguration – bestimmt. Die Arbeitskonfiguration wird zur Nachfolger-Konfiguration der Basiskonfiguration. Besitzt eine Basiskonfiguration bereits eine Nachfolger-Konfiguration, so wird ein neuer Konfigurationszweig angelegt. Die Konfigurationen bilden somit die Versionshistorie der Dokumente einer ETA.
In einer WTA sind nur Objekt-, Link- und Attributversionen zugreifbar, die in der Arbeitskonfiguration oder ihren Vorgängern gespeichert sind. Versionen, die in anderen Konfigurationen vorliegen, sind nicht zugreifbar.

Die Konfigurationen bilden neben der Versionierung auch die Grundlage der konkurrierenden und der isolierten Arbeit von Entwicklern. Wählen zwei Entwickler die selbe Konfiguration als Basiskonfiguration und haben sich nicht für die isolierte Arbeit entschieden, so arbeiten sie konkurrierend auf der selben Diagrammversion. Ihre Änderungen sind durch den Sperrmechanismus gegeneinander geschützt, wobei sie der Benachrichtigungsmechanismus über Änderungen informiert. Zur Sicherung konsistenter Zwischenversionen können die Entwickler unabhängig von einander Sicherungspunkt-Konfigurationen anlegen. Bei einem Transaktionsabbruch bleiben dann die Änderungen des Entwicklers, der den Sicherungspunkt anlegte, erhalten.

Differenzanzeige und Mischen von UML-Diagrammen. Im Rahmen der kooperativen Entwicklung entstehen mehrere Versionen der einzelnen Diagramme, so daß die Anzeige und das Mischen von Differenzen zwischen einzelnen Versionen eine große Bedeutung erlangen. Die Berechnung der Differenzen und das Mischen von ihnen ist unter Ausnutzung der durch das OMS und die Versionsverwaltung zur Verfügung gestellten Objekt- und Versionsidentifizierern möglich. Zur Berechnung müssen die Syntaxbäume der Diagramme traversiert und die zusammengehörenden Versionen der einzelnen Objekte gesucht werden. Sind sie anhand der Identifizierer gefunden, lassen sich die Differenzen durch einen einfachen Vergleich bestimmen. Interessant ist jedoch die Frage, wie diese Differenzen an die Werkzeuge weitergereicht werden und wie diese sie anzeigen.
Die Differenzen und die vorläufige Mischversion, die zum Lösen von Mischkonflikten benötigt wird, werden in einem erweiterten Editiermodell persistent im OMS gespeichert. Das bietet die Möglichkeit, die OMS-orientierte Werkzeugarchitektur auch für die Differenz- und Mischwerkzeuge nutzen zu können. Diese erleichtert einerseits die Werkzeugentwicklung. Andererseits erreicht man so eine Trennung von Berechnung und Anzeige der Differenzen bzw. vom Erstellen der vorläufigen Mischversion und dem Lösen der Konflikte.

Die Anzeige der Differenzen bzw. der vorläufigen Mischversion basiert auf einer gemeinsamen Anzeige beider Diagrammversionen – den Basisdiagrammen – einschließlich der Differenzen bzw. Konflikte. Alle Diagrammelemente, die beiden Diagrammversionen gemeinsamen sind, werden schwarz gezeichnet. Die Diagrammelemente, die nur in einem Basisdiagramm enthalten sind, werden eingefärbt. Jedem Basisdiagramm wird hierfür eine eigene Farbe zugeordnet, die die Zugehörigkeit eines Diagrammelementes zum Basisdiagramm kennzeichnet.

Das Mischen basiert auf einem 3-Wege-Misch-Algorithmus, so daß bereits viele Differenzen automatisch entschieden werden können. Diese bereits gemischten Differenzen werden in der vorläufigen Mischversion blau gezeichnet. Alle manuell gelösten Konflikte erhalten auch eine blaue Kennzeichnung. Eingefärbt sind demnach in der vorläufigen Mischversion ausschließlich die Konflikte.
Die Anwender haben auch die Möglichkeit, bereits gelöste Konflikte, seien es die automatisch gemischten Differenzen oder die manuell gelösten Konflikte, nachträglich zu ändern und die jeweils andere Lösung auszuwählen. Aus diesem Grund können die Konfliktlösungen noch nicht direkt im Editiermodell realisiert werden, sondern sie müssen als eine Art von Edit-Skript an den einzelnen Objekten des Editiermodells gespeichert werden. Nach dem vollständigen Lösen aller Konflikte wird dann die vorläufige Mischversion in die endgültige Mischversion durch Ausführen der Edit-Skripte umgewandelt.

Haben sich die zu vergleichenden oder zu mischenden Diagramme eine längere Zeit unabhängig entwickelt, so gibt es viele Unterschiede und Konflikte, die angezeigt werden. Diese Vielzahl an eingefärbten Diagrammelementen würde die Anzeige nahezu unbrauchbar machen. Unter Ausnutzung der Versionshistorie oder des Editiermodells kann man die eingefärbten Differenzen und Konflikte gruppieren. Bei Verwendung der Versionshistorie kann man die Konfigurationen zum gruppieren nutzen, bei Verwendung des Editiermodells ist die Gruppierung anhand

von Diagrammelementtypen wie z. B. Klassen, Methoden oder Interfaces möglich. Die so gebildeten Gruppen können dann einzeln eingefärbt gezeichnet werden. Alle gruppierten Elemente, die zwischenzeitlich nicht von Interesse sind, können dann grau gezeichnet werden, wodurch die Aufmerksamkeit der Entwickler primär auf die auffällig gefärbten Diagrammelemente gelenkt wird.

7.2 Übertragbarkeit

Die dieser Arbeit zugrunde liegenden Annahmen und Voraussetzungen sind bei den wenigsten CASE-Werkzeugen gegeben, insbesondere basieren sie nicht auf einem Repository zur Datenhaltung, dessen Dienste genutzt werden können. Daher stellt sich die Frage, wie sich die vorgestellten Konzepte auf andere Systeme übertragen lassen.

Entwurfstransaktionen. Der Einsatz von Entwurfstransaktionen ist im Rahmen mehrbenutzerfähiger CASE-Werkzeuge mit zentraler Datenhaltung sinnvoll. Die weitverbreiteten Einzelarbeitsplatz-Werkzeuge ziehen für sich allein betrachtet jedoch keinen direkten Vorteil daraus. Kombiniert man die Werkzeuge mit einer Versionsverwaltung, so kann das Konzept der Entwurfstransaktionen darin Anwendung finden. Die Entwurfstransaktionen können dann genutzt werden, um einzelne Dokumentversionen ihren korrespondierenden Aufgaben zuzuordnen. Im wesentlichen handelt es sich dann um die Kombination von Versionsverwaltung und Projektmanagement. Ein Unterschied zum vorgeschlagenem Konzept besteht darin, daß die CASE-Werkzeuge von den Werkzeugen zur Versionsverwaltung getrennt sind und somit die Vorteile der integrierten Lösung fehlen. Diese sind ein unterbrechungsfreies Arbeiten, das mit der Auswahl einer Aufgabe beginnt und nahtlos mit der Bearbeitung der Dokumente fortgesetzt werden kann.

Versionierung und Werkzeugtransaktionen. Konventionelle CASE-Werkzeuge arbeiten intern auch mit feinkörnig modellierten Dokumenten, sie bieten jedoch keinen direkten Zugriff darauf. Diese interne Repräsentation wird beim Speichern der Dokumente in eine binäre oder textuelle Repräsentation umgewandelt, so daß eine eigenständige Versionsverwaltung nur auf diese Repräsentation aufsetzen kann. Um das vorgeschlagene Versionsverwaltungskonzept für konventionelle Werkzeuge nutzbar machen zu können, müßten die existierenden Werkzeuge so erweitert werden, daß sie zusätzliche Schnittstellen für die Versionsverwaltung bieten.
Eine Möglichkeit wäre die Erweiterung des Editiermodells um Versionsidentifizierer. Bei einer Änderung eines Knotens müßte dann das Werkzeug wissen, welche neue Versionsnummer die externe Versionsverwaltung für die persistente Repräsentation vergeben wird und könnte diese an den modifizierten Knoten setzen. Die Knoten selbst bräuchten nicht im Werkzeug versioniert zu werden. Diese Aufgabe würde die externe Versionsverwaltung übernehmen.
Diese Lösung würde zwar keine Werkzeugtransaktionen bieten, die Idee, daß jede Werkzeugsitzung zu neuen Versionen der Dokumente führt, wäre jedoch umsetzbar. Die weitere Aufgabe der Werkzeugtransaktionen, die Sicherstellung der Konsistenz bei kooperativer Arbeit, läßt sich nur mit Unterstützung eines Repositorys erreichen.

Differenzberechnung, -anzeige und Mischen. Setzt man voraus, daß konventionelle CASE-Werkzeuge ein feinkörniges Editier-Metamodell nutzen, welches eindeutige Identifizierer besitzt, die sich zwischen einzelnen Werkzeugsitzungen nicht ändern, so läßt sich die Berechnung und Anzeige von Differenzen direkt übernehmen. Besitzt das Editiermodell keine Identifizierer

oder werden diese zwischen einzelnen Werkzeugsitzungen verändert, so muß die Differenzberechnung auf die Semantik des Editiermodells abgestützt werden. Die Anzeige kann aber in jedem Fall übernommen werden. Gleiches gilt für das Mischen von Dokumenten.

7.3 Ausblick

Die in dieser Arbeit vorgeschlagenen Konzepte bieten vier Ansatzpunkte für Erweiterungen oder Vertiefungen:

1. Empirische Untersuchungen
2. Erweiterte Funktionalität der Entwurfstransaktionen
3. Verbesserung der Versionsverwaltung
4. Optimierung der Differenz- und Mischwerkzeuge

Empirische Untersuchungen. In dieser Arbeit wurde nicht geklärt, wie sich die vorgeschlagenen Konzepte auf konkrete Projekte in der Softwareentwicklung anwenden lassen und ob sie durch die Entwickler akzeptiert werden. Das muß in weiterführenden Arbeiten angegangen werden.

Erweiterte Funktionalität der Entwurfstransaktionen. Das vorgeschlagene Entwurfstransaktionskonzept ist eine Lösung, um die angelegten Versionen auf die zugrunde liegenden Aufgaben abzubilden. Die Entwurfstransaktionen (ETA) bieten aber noch Erweiterungspotential:

- Anbindung an das Projektmanagement: Da die ETA einzelne Aufgaben modellieren, bieten sie sich als Anknüpfungspunkt für Projektmanagementsysteme an.
- Weitere Funktionalitäten des Softwarekonfigurations-Managements (SKM) anbinden: SKM besteht nicht nur aus der Versionierung, sondern umfaßt weitere Aufgabenbereiche, wie z. B. Änderungsmanagement oder Fehlermanagement. Beide Aufgabenbereiche basieren auf einer Aufgabenverwaltung, so daß auch hier Anknüpfungspunkte existieren.
- Verteilte Entwicklung: Bei verteilter Entwicklung stellt sich immer die Frage, welche Daten an den einzelnen verteilten Arbeitsplätzen benötigt werden. Unter der Annahme, daß die ETA die zur Bearbeitung einer Aufgabe benötigten Dokumente verwalten, ist es möglich, sie als Grundlage für die Verteilung zu nutzen.
- Zugriffskontrolle: In größeren Projekten arbeiten viele Entwickler zusammen, die jeweils unterschiedliche Aufgaben wahrnehmen. Die Dokumente sollten jedoch nicht durch alle Entwickler modifiziert werden dürfen. Die Einführung von Zugriffskontrollmechanismen erscheint in diesem Zusammenhang sinnvoll. Berücksichtigt man, daß die Entwickler in einzelne Gruppen eingeteilt werden, die bestimmte Aufgaben bearbeiten sollen, so kann man die ETA um Funktionen zur Zugriffskontrolle erweitern. In diesem Fall können die ETA mit der im OMS nachgebildeten Gruppenstruktur verknüpft werden, so daß nur bestimmte Entwicklergruppen Zugriff auf bestimmte ETA besitzen. Abhängig von den Funktionen der Zugriffskontrollmechanismen können unterschiedliche Arten von Zugriffen erlaubt oder verboten werden.

Verbesserung der Versionsverwaltung. Im Rahmen der Versionsverwaltung gibt es einige Möglichkeiten für eine Anpassung an bestimmte Anwendungsfälle[2] oder Optimierungen.

- Die Funktionsweise der Werkzeugtransaktionen (WTA) bestimmt entscheidend die Möglichkeiten der Versionsverwaltung. Ein Beispiel ist das Anlegen der Sicherungspunkt-Konfigurationen. Im vorgeschlagenen Konzept stellt eine Sicherungspunkt-Konfiguration nur einen Rücksetzpunkt für die anlegende WTA dar, alle anderen WTA werden nicht beeinflußt. Unter bestimmten Umständen könnte es aber sinnvoll sein, auch die Änderungen von ihnen zu sichern.

- Das Konzept unterscheidet nicht zwischen versionierten und unversionierten Objekten, Links oder Attributen. Für einige Anwendungsfälle könnte es aber interessant sein, daß in einer WTA nicht von allen veränderten Objekten, Links oder Attributen eine neue Version angelegt wird, sondern daß sie unversioniert bleiben, wie es z. B. Adele [80] bietet. Ansatzpunkte hierfür sind Lösungen, die auf den Typen aufsetzen. Die Versionsverwaltung könnte anhand des Typs entscheiden, ob eine neue Version angelegt werden muß oder nicht.

- Aufgrund der feinkörnigen Modellierung bestehen die Dokumente aus einer Vielzahl an Objekten, so daß i.d.R. ein Direktzugriff auf bestimmte Versionen eines Objektes nicht sinnvoll erscheint und daher auch nicht umgesetzt wurde. Für bestimmte Anwendungsfälle könnte der Direktzugriff wünschenswert sein, einschließlich des Zugriffs auf den Versionsbaum eines Objektes. Dieser ist im vorgeschlagenen Konzept nur indirekt über die Konfigurationen möglich, wie er bei der Differenzberechnung zur Zuordnung von Änderungen zu Konfigurationen eingesetzt wird.

- Aufgrund technischer Gründe, die eine vollständige Reimplementierung der Objektverwaltung von H-PCTE bedeutet hätten, wurde auf die Trennung von unversionierten und versionierten Teilen der internen Objektstruktur von H-PCTE verzichtet, so daß auch von unversionierten Bestandteilen eine neue Version angelegt wird, die in einem größeren Speicherbedarf resultiert. Bei einer Neukonzeption von H-PCTE sind diese Aspekte zu berücksichtigen.

Optimierung der Differenz- und Mischwerkzeuge. Bei den Differenz- und Mischwerkzeugen gibt es zwei Ansatzpunkte für Erweiterungen. Das Konzept zur Anzeige von Differenzen stellt einen ersten Vorschlag dar. Ein Problem des Vorschlags liegt darin, daß das Layout der Diagramme verändert werden muß. Solange das Layout nicht relevant ist, funktioniert diese Lösung. Besitzt das Layout jedoch eine Relevanz, so müssen anderen Lösungen zur Darstellung der Differenzen gefunden werden, evtl. könnte eine 3-dimensionale Darstellung einen Lösungsansatz darstellen.

Die Differenz- und Mischwerkzeuge dienen in erste Linie zur Visualisierung von Differenzen zwischen Versionen. In der Softwareentwicklung tritt jedoch des öfteren der Fall auf, daß mehrere Varianten einer Software existieren, an denen dieselben Änderungen durchgeführt werden müssen. Bei der bisherigen Lösung müssen alle Varianten einzeln bearbeitet werden. Erweitert man die Differenzwerkzeuge jedoch so, daß die Varianten gleichzeitig angezeigt und auch bearbeitet werden können, so könnte das eine Lösung sein, die es erlaubt, Änderungen, die für alle Varianten gelten, nur einmal durchführen zu müssen.

[2] Dem Autor sind keine solchen Anwendungsfälle im Rahmen der Entwicklung von UML-Diagrammen bekannt. In anderen Anwendungsbereichen könnte jedoch ein Bedarf existieren.

Anhang A

Schema Erweiterung

```
sds scm:

------------------------------------------------------------------
----------------- IMPORTE  ---------------------------------------
------------------------------------------------------------------

------- Object Types

import objecttype    system-object ;
import objecttype    system-common_root;
import objecttype    system-activity;

import objecttype    security-user_group;
import objecttype    security-user;

import objecttype    hpcte-segment;

------- Attribute Types

import attribute system-number;
import attribute system-system_key;
import attribute system-exact_identifier ;
import attribute hpcte-user_name;


------------------------------------------------------------------
----------------- ATTRIBUTTYPEN  ---------------------------------
------------------------------------------------------------------

attribute Identifier             : (read) natural;
attribute Config_Rev_Id          : (read) natural;
attribute Branch_Id              : (read) natural;
attribute Origin_Branch_Id       : (read) natural;
attribute Origin_Config_Rev_Id   : (read) natural;

attribute Object_Id              : (read) natural;
attribute Branch_Name            : (read) string;
attribute Branch_Description     :        string;
```

```
-----------------------------------------------------------------------
---------------- LINKTYPEN  -------------------------------------------
-----------------------------------------------------------------------



----------------- Existenz erhaltende Links

linktype    document_in_workspace : (navigate) existence link  (number)
to document_root reverse document_of;

linktype    document_of       : (navigate) reference link      (number)
to workspace reverse document_in_workspace;

-----------------

linktype    design_transactions          : (navigate) existence link
to design_transaction_directory reverse design_transactions_of;
linktype    design_transactions_of       : (navigate) reference link
to common_root reverse design_transactions;

-----------------

linktype    known_design_transaction     : existence link  (number)
to design_transaction reverse known_design_transaction_of;
linktype    known_design_transaction_of  : reference link
to design_transaction_directory reverse known_design_transaction;

-----------------
-- DTA besitzen eine Baumstruktur

linktype    known_sub_design_transaction    : existence link  (number)
to design_transaction reverse known_sub_design_transaction_of;
linktype    known_sub_design_transaction_of : reference link
to design_transaction reverse known_sub_design_transaction;


-----------------
-- Basis-Konfiguration einer DTA

linktype    base_configuration           : (navigate) existence link
to configuration reverse base_configuration_of;
linktype    base_configuration_of        : (navigate) reference link  (number)
to design_transaction reverse base_configuration;

-----------------
-- Konfigurationen bilden einen gerichteten azyklischen Graphen

linktype    known_successor              : (navigate) existence link  (number)
 to configuration reverse known_predecessor;
linktype    known_predecessor            : (navigate) reference link  (number)
```

```
to configuration reverse known_successor;

-----------------


linktype    assigned_workspace          : (navigate) existence link
to workspace reverse workspace_of;
linktype    workspace_of                : (navigate) reference link
        to design_transaction reverse assigned_workspace;

-----------------

----------------- Referenz Links

-----------------

linktype    base_design_transaction     : (navigate) reference link
to design_transaction reverse base_design_transaction_of;
linktype    base_design_transaction_of  : (navigate) reference link  (number)
to design_transaction reverse base_design_transaction;

-----------------

linktype    known_branch                : (navigate) reference link  (number)
to configuration reverse known_branch_of;
linktype    known_branch_of             : (navigate) reference link  (number)
to design_transaction  reverse known_branch;

-----------------


linktype    main_branch                 : (navigate) reference link
to configuration  reverse main_branch_of;
linktype    main_branch_of              : (navigate) reference link  (number)
to design_transaction reverse main_branch;

-----------------

linktype    created_by                  : (navigate) reference link  (number)
to user reverse created_configuration;
linktype    created_configuration       : (navigate) reference link  (number)
to configuration reverse created_by;

------------------------------------------------------------------
-------------- Objekte -------------------------------------------
------------------------------------------------------------------

  extend objecttype common_root with
  link

        design_transactions;
```

```
  end common_root;


-----------------



  objecttype design_transaction_directory: child type of activity
  with
     link

        known_design_transaction;
        design_transactions_of;

  end design_transaction_directory;

-----------------


  objecttype design_transaction: child type of object
  with

     attribute

        is_main_dta     : (read) enumeration
                (MAIN_DTA, SUB_DTA, UNDEFINED) := UNDEFINED;

        Identifier;
        state           : (read) enumeration
                (CREATED, INITIALISED, RUNNING, SYNCHRONIZED, COMMITED) := CREATED;

         Cooperation_Mode     : (read) enumeration
                (EXCLUSIVE, COOPERATIVE) := COOPERATIVE;


     link

        base_configuration;
        base_design_transaction;
        base_design_transaction_of;

        main_branch;
        known_branch;

        assigned_workspace;

        known_sub_design_transaction;
        known_sub_design_transaction_of;

        known_design_transaction_of;

  end design_transaction;
```

```
-----------------

  objecttype configuration: child type of object
  with

     attribute

        Branch_Id;
        Config_Rev_Id;
        Process_Counter      : (read) natural;
        Cooperation_Mode     : (read) enumeration
                (ISOLATED, COOPERATIVE) := COOPERATIVE;

        State                : (read) enumeration
                (RUNNING, FINISHED) := RUNNING;

     link

        main_branch_of;
        known_predecessor;
        known_successor;
        created_by;
        known_branch_of;
        base_configuration_of;

  end configuration;


-----------------

  objecttype workspace: child type of segment
  with
     link

          workspace_of;
          document_in_workspace;

  end workspace;


-----------------

  objecttype document_root : child type of object
  with
     link
          document_of;
  end document_root;

----------------- Erweiterungen
```

```
 extend objecttype user
 with
    link

        created_configuration;
        owner_of;

  end user;

-----------------

 extend linktype known_branch
 with
     attribute
          Branch_Name;
          Branch_Description;
 end known_branch;


end scm;
```

Literaturverzeichnis

[1] Abramowicz, K. ; Dittrich, K. R. ; Laengle, R. ; Ranft, M. ; Raupp, T. ; Rehm, S. : DAMOKLES - Architektur, Implementierung, Erfahrungen. In: *Informatik - Forschung und Entwicklung* 6 (1991), Nr. 1, S. 1–13. – ISSN 0178–3564

[2] Abramowicz, K. ; Rehm, S. ; Raupp, T. ; Ranft, M. ; Längle, R. ; Härtig, M. ; Gotthard, W. ; Dittrich, K. R.: Support for Design Processes in a Structurally Object-Oriented Database System. In: Dittrich, K. R. (Hrsg.): *Proceedings of the 2nd International Workshop on Object-Oriented Database systems* Bd. 334. Berlin - Heidelberg - New York : Springer–Verlag, September 1988. – ISBN 3–540–50345–5, S. 80–97

[3] Abu-Shakra, M. ; Fisher, G. L.: Multi-Grain Version Control in the Historian System. In: [153], S. 46–56

[4] Al-Khudair, A. ; Gray, W. A. ; Miles, J. C.: Object-Oriented Versioning in a Concurrent Engineering Design Environment. In: Read, B. (Hrsg.): *Advances in Databases 18th British National Conference on Databases, BNCOD 18 Chilton, UK, July 9-11, 2001, Proceedings* Bd. 2097. Berlin - Heidelberg - New York : Springer–Verlag

[5] Alanen, M. ; Porres, I. : Difference and Union of Models. In: *UML 2003 - The Unified Modeling Language: Modeling Languages and Applications* Bd. 2863. Berlin - Heidelberg - New York : Springer–Verlag, Oktober 2003, S. 2–17

[6] Asklund, U. : Identifying Conflicts During Structural Merge. In: Magnusson, B. (Hrsg.): *Proceedings of NWPER'94, Nordic Workshop on Programming Environment Research*, 1994

[7] Asklund, U. : *Configuration Management for Distributed Development – Practice and Needs*, Dept. of Computer Science, Lund University, Licentiate thesis, 1999

[8] Asklund, U. ; Bendix, L. ; Christensen, H. B. ; Magnusson, B. : The Unified Extensional Versioning Model. In: [75], S. 100–122

[9] Balzert, H. : *Lehrbuch der Software-Technik: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*. Spektrum Akadem. Verlag, 1998

[10] Bamberger, J. : Essence of the Capability Maturity Model. In: *IEEE Computer* 30 (1997), Juni, Nr. 6, S. 112–114

[11] Barghouti, N. S.: Supporting Cooperation in the MARVEL Process-Centered SDE. In: Weber, H. (Hrsg.): *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments*. New York, NY, USA : ACM Press, Dezember 1992. – Published as SIGSOFT Software Engineering Notes, Volume 17, Number 5., S. 21–31

[12] Barghouti, N. S. ; Kaiser, G. E.: Concurrency Control in Advanced Database Applications. In: *ACM Computing Surveys* 23 (1991), September, Nr. 3, S. 269–317. – ISSN 0360–0300

[13] Barnard, D. T. ; Clarke, G. ; Duncan, N. : Tree-to-tree Correction for Document Trees / Departement of Computing andf Information Science Queen's University Kingston Ontario, Canada. 1995. – Forschungsbericht

[14] Belkhatir, N. ; Estublier, J. ; Melo, W. L.: ADELE 2 – An approach to software development coordination. In: Fuggetta, A. (Hrsg.) ; Conradi, R. (Hrsg.) ; Ambriola, V. (Hrsg.): *Proceedings of the* 1^{st} *European Workshop on Software Process Modeling*, 1991, S. 89–100

[15] Bendix, L. : Fully Supported Recursive Workspaces. In: [209], S. 256–261

[16] Bendix, L. : Experiences with an Object-Based approach to Configuration Management. In: *7th International Workshop on Software Configuration Management, Boston, Massachusetts, May 18-19, 1997.* Berlin - Heidelberg - New York : Springer–Verlag, Mai 1997

[17] Bendix, L. : Experience from Teaching Configuration Management. In: [103],

[18] Bendix, L. ; Asklund, U. : Summary of the Subworkshop on Change Management. In: *Nordic Journal of Computing* 6 (1999), Nr. 1, S. 129–

[19] Bendix, L. ; Larsen, P. N. ; Nielsen, A. I. ; Petersen, J. L. S.: CoEd—A Tool for Versioning of Hierarchical Documents. In: [153], S. 174–187

[20] Bendix, L. ; Vitali, F. : VTML for Fine-Grained Change Tracking in Editing Structured Documents. In: [75], S. 139–156

[21] Berger, M. ; Schill, A. ; Völksen, G. : Supporting Autonomous Work and Reintegration in Collaborative Systems. In: Conen, W. (Hrsg.) ; Neumann, G. (Hrsg.): *Coordination Technology for Collaborative Applications - Organizations, Processes, and Agents* Bd. 1364. Berlin - Heidelberg - New York : Springer–Verlag, 1998. – ISBN 3–540–64170–X, S. 177–198

[22] Berghoff, J. ; Drobnik, O. ; Lingnau, A. ; Mönch, C. : Agent-based configuration management of distributed applications. In: *Proceedings of the Third International Conference on Configurable Distributed Systems. Annapolis, MD, USA, 6-8 May 1996.* Los Alamitos, CA, USA : IEEE Comput. Soc. Press, 1996, S. 52–9

[23] Bergstraesser, T. ; Bernstein, P. ; Pal, S. ; Shutt, D. : Versions and workspaces in Microsoft Repository. In: *1999 ACM SIGMOD International Conference on Management of Data. Philadelphia, PA, USA. 1-3 June 1999.* Bd. 28, 1999, S. 532–533

[24] Berliner, B. : CVS II: Parallelizing Software Development. In: USENIX Association (Hrsg.): *Proceedings of the Winter 1990 USENIX Conference, January 22–26, 1990, Washington, DC, USA.* Berkeley, CA, USA : USENIX, Januar 1990, S. 341–352

[25] Berten, A. ; Kamphusmann, T. ; Kelter, U. ; Monecke, M. ; Ohst, D. ; Platz, D. ; Wagner, C. : Jahresbericht zum Arbeitsbereich Datenbankmanagementsysteme fur das Projektjahr 1998 – Sonderforschungsbereich 240 Teilprojekt Z2 / Praktische Informatik, Universität-Gesamthochschule Siegen. 1998 (27). – Internes Memorandum

[26] Berzins, V. (Hrsg.): *Software Merging and Slicing*. IEEE Computer Society Press, 1995

[27] Bieliková, M. : Software configuration management (CEEPUS invited lecture). In: Kalpic, D. (Hrsg.) ; Dobric, V. H. (Hrsg.): *19th International Conference on Information Technology Interfaces ITI'97, Pula, Croatia*, 1997, S. 543–550

[28] Bieliková, M. ; Návrat, P. : Modelling Versioned Hypertext Documents. In: [153], S. 188–197

[29] Borghoff, U. M. ; Schlichter, J. H.: *Rechnergestützte Gruppenarbeit*. 2. Berlin - Heidelberg - New York : Springer–Verlag, 1998

[30] Bouazza, A. ; Molli, P. : Unifying coupled and uncoupled collaborative work in virtual teams. In: *ACM CSCW'2000 workshop on collaborative editing systems, Philadelphia, Pennsylvania, USA, de'cembre 2000*, 2000

[31] Bruegge, B. ; Dutoit, A. H.: *Object-Oriented Software Engineering: Conquering Complex and Changing Systems*. Prentice-Hall, 2000. – ISBN ISBN 0–13–017452–1

[32] Buffenbarger, J. : Syntactic Software Merging. In: [72], S. 153–172

[33] Burger, C. : Team awareness with mobile agents in mobile environments. In: *Proceedings of the 7th International Conference on Computer Communications and Networks (IC3N'98)*. Los Alamitos : IEEE Computer Society, Oktober 1998, S. 45–49

[34] Cagan, M. : Untangling Configuration Management. In: [72], S. 35–52

[35] Campbell, I. (Hrsg.): *Proceedings of the PCTE 1993 Conference*. 1993

[36] Cederqvist, P. : *Version Management with CVS*. Box 2044, S-580 02 Linkoping, Sweden: Signum Support AB, November 1993

[37] Cellary, W. ; Durand, D. ; Haake, A. ; Hicks, D. ; Vitali, F. ; Whitehead, J. : Things Change: Deal with it! Versioning, Cooperative Editing and Hypertext. In: *Proceedings of the Seventh ACM Conference on Hypertext*, 1996 (Panel), S. 259

[38] Ceri, S. ; Fraternali, P. ; Paraboschi, S. : XML: Current Developments and Future Challenges for the Database Community. In: Zaniolo, C. (Hrsg.) ; Lockemann, P. C. (Hrsg.) ; Scholl, M. H. (Hrsg.) ; Grust, T. (Hrsg.): *Advances in Database Technology - EDBT 2000 7th International Conference on Extending Database Technology, Konstanz, Germany, March 2000. Proceedings* Bd. 1777. Berlin - Heidelberg - New York : Springer–Verlag, 2000, S. 3–17

[39] Chawathe, S. ; Garcia-Molina, H. : Meaningful Change Detection in Structured Data. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1997, S. 26–37

[40] Chawathe, S. S. ; Rajaraman, A. ; Garcia-Molina, H. ; Widom, J. : Change detection in hierarchically structured information. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1996, S. 493–504

[41] Chien, S.-Y. ; Tsotras, V. J. ; Zaniolo, C. ; Zhang, D. : Storing and Querying Multiversion XML Documents using Durable Node Numbers. In: *Proc. of The 2nd International Conf. on Web Information Systems Engineering (WISE), Kyoto, Japan, Dec. 2001*, 2001

[42] Choi, E. J. ; Kwon, Y. R.: An Efficient Method for Version Control of a Tree Data Structure. In: *Software Practice and Experience* 27 (1997), Juli, Nr. 7, S. 797–811. – ISSN 0038–0644

[43] Chou, H.-T. ; Kim, W. : A Unifying Framework for Version Control in a CAD Environment. In: *Proceedings of the Twelfth International Conference on Very Large Data Bases (VLDB), Kyoto, Japan*, Morgan Kaufmann, August 1986, S. 336–344

[44] Christensen, H. B.: The Ragnarok Architectural Software Configuration Management Model. In: Sprague, Jr., R. H. (Hrsg.): *Proceedings of the Thirty-Second Annual Hawaii International Conference On System Sciences, January 5-8, 1999, Maui, Hawaii*, 1999

[45] Christensen, H. B.: Tracking Change in Rapid and eXtreme Development: A Challenge to SCM Tools. In: [103],

[46] Chu-Carroll, M. C. ; Sprenkle, S. : Coven: brewing better collaboration through software configuration management. In: *Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*, ACM Press, 2000. – ISBN 1–58113–205–0, S. 88–97

[47] Chu-Carroll, M. C. ; Wright, J. ; Shields, D. : Supporting aggregation in fine grained software configuration management. In: *Proceedings of the tenth ACM SIGSOFT symposium on Foundations of software engineering*, ACM Press, 2002. – ISBN 1–58113–514–9, S. 99–108

[48] Clarke, S. ; Harrison, W. ; Ossher, H. ; Tarr, P. : Subject-Oriented Design: Towards Improved Alignment of Requirements, Design and Code. In: *SIGPLAN Notices* 34 (1999), Oktober, Nr. 10

[49] Coallier, F. : How ISO 9001 fits into the software world. In: *IEEE Software* 11 (1994), Januar, Nr. 1, S. 98–100

[50] Cobéna, G. ; Abiteboul, S. ; Marian, A. : Detecting Changes in XML Documents. In: *18. International Conference on Data Engineering (ICDE) San Jose, California, USA, February 26-March 1, 2002*, 2002

[51] Colinas, M. F. ; Oussalah, C. ; Talens, G. : Versions of Simple and Composite Objects. In: Agrawal, R. (Hrsg.) ; Baker:1999:PD, S. (Hrsg.) ; Bell, D. (Hrsg.): *Very Large Data Bases, VLDB '93: Proceedings of the 19th International Conference on Very Large Data Bases, August 24–27, 1993, Dublin, Ireland.* Palo Alto, Calif., USA : Morgan Kaufmann Publishers, 1993, S. 62–72

[52] Conradi, R. (Hrsg.): *Proceedings of the 7th Workshop on System Configuration Management (SCM-7), at ICSE'97 Boston, MA, USA, May 18-19, 1997.* Bd. *1235.* Berlin - Heidelberg - New York : Springer–Verlag, 1997 (Lecture Notes in Computer Science (LNCS))

[53] Conradi, R. ; Hagaseth, M. ; Liu, C. : Planning Support for Cooperating Transactions in EPOS. In: *Information Systems* 20 (1995), Nr. 4, S. 317–336

[54] Conradi, R. ; Malm, C. C.: Cooperating Transactions and Workspaces in EPOS: Design and Preliminary Implementation. In: Andersen, R. (Hrsg.) ; Bubenko, Jr., J. A.

(Hrsg.) ; Sølvberg, A. (Hrsg.): *Advanced Information Systems Engineering 3th International Conference, CAiSE'91, Trondheim, Norway, May 13 – 15* Bd. 498. Berlin - Heidelberg - New York : Springer–Verlag, 1991, S. 375–392

[55] Conradi, R. ; Westfechtel, B. : Version Models for Software Configuration Management / RWTH Aachen, Germany. 1996 (AIB 96-10). – Forschungsbericht

[56] Conradi, R. ; Westfechtel, B. : Configuring Versioned Software Products. In: [209], S. 88–109

[57] Conradi, R. ; Westfechtel, B. : Towards a Uniform Version Model for Software Configuration Management. In: [52], S. 1–17

[58] Conradi, R. ; Westfechtel, B. : Version models for software configuration management. In: *ACM Computing Surveys* 30 (1998), Juni, Nr. 2, S. 232–282. – ISSN 0360–0300

[59] Conradi, R. ; Westfechtel, B. : SCM: Status and Future Challenges. In: [75], S. 228–231

[60] Cosquer, F. J. N. ; Veríssimo, P. ; Krakowiak, S. ; Decloedt, L. : Support for Distributed CSCW Applications. In: Krakowiak, S. S. S. (Hrsg.): *Advances in Distributed Systems – Advanced Distributed Computing: From Algorithms to Systems* Bd. 1752. Berlin - Heidelberg - New York : Springer–Verlag, 2000, S. 295–326

[61] Courington, W. : *The Network Software Environment.* : Sun Microsystems, Inc., 1989

[62] Crnkovic, I. : A Change Process Model in an SCM Tool, Presentation. In: *Euromicro 98 Conference, Västeras, Sweden, August*, 1998

[63] Crnkovic, I. : Software Process Measurements using Software Configuration Management. In: *The 11th European Software Control and Metrics Conference Munich, Germany, May*, 2000

[64] Cronk, R. D.: Tributaries and Deltas: Tracking software change in multiplatform environments. In: *Byte Magazine* 17 (1992), Januar, Nr. 1, S. 177–186. – ISSN 0360–5280

[65] Däberitz, D. : *Der Bau von Software-Entwicklungsumgebungen mit Hilfe von Nicht-Standard-Datenbanken*, Praktische Informatik, Universität-Gesamthochschule Siegen, Diss., 1997

[66] Däberitz, D. ; Kelter, U. : Rapid Prototyping of Graphical Editors in an Open SDE. In: *Proceedings 7th Conf. on Software Engineering Environments (SEE'95), Noordwijkerhout, Netherlands* IEEE Press, 1995, S. 61–72

[67] Dart, S. A.: Spectrum of Functionality In Configuration Management Systems / Software Engineering Institute, Carnegie-Mellon University. Pittsburgh, Pennsylvania 15213 : Software Engineering Institute, Carnegie-Mellon University, Dezember 1990 (CMU/SEI-90-TR-11). – Forschungsbericht

[68] Dart, S. A.: Concepts in Configuration Management Systems. In: [83], S. 1–18

[69] Dayal, U. ; Hsu, M. ; Ladin, R. : A Transactional Model for Long-Running Activities. In: *Proceedings of the 17th Conference on Very Large Databases (VLDB), Barcelona*, Morgan Kaufman, September 1991

[70] Denert, E. (Hrsg.) ; Hoffman, D. (Hrsg.) ; Ludewig, J. (Hrsg.) ; Parnas, D. (Hrsg.): *Dagstuhl Seminar Report Nr. 230: Software Engineering Research and Education: Seeking a new Agenda*. Schloss Dagstuhl, feb 1999 . – 14.02.– 19.02.. – ISSN 0940–1121

[71] Dittrich, K. R. ; Gotthard, W. ; Lockemann, P. C.: DAMOKLES- A Database System for Software Engineering Environments. In: *Proceedings of the International Workshop on Advanced Programming Environments, Trondheim, Norway* Bd. 244. Berlin - Heidelberg - New York : Springer–Verlag, Juni 1986, S. 353–371

[72] Estublier, J. (Hrsg.): *Software Configuration Management: Selected Papers of the ICSE SCM-4 and SCM-5 Workshops*. Bd. *1005*. Berlin - Heidelberg - New York : Springer–Verlag, 1995 (Lecture Notes in Computer Science (LNCS))

[73] Estublier, J. : Work Space Management in Software Engineering Environments. In: [209], S. 127–138

[74] Estublier, J. : Distributed Objects for Concurrent Engineering. In: *Proceedings of the 9th International Symposium on Software Configuration Management (SCM-9), Toulouse, France, September 5-7, 1999* [75], S. 172–185

[75] Estublier, J. (Hrsg.): *Proceedings of the 9th International Symposium on Software Configuration Management (SCM-9), Toulouse, France, September, 1999*. Bd. *1675*. Berlin - Heidelberg - New York : Springer–Verlag, 1999 (Lecture Notes in Computer Science (LNCS))

[76] Estublier, J. : Software configuration management: a roadmap. In: *ICSE - Future of SE Track*, 2000, S. 279–289

[77] Estublier, J. : Objects Control for Software Configuration Management. In: Dittrich, K. R. (Hrsg.) ; Geppert, A. (Hrsg.) ; Norrie, M. C. (Hrsg.): *Advanced Information Systems Engineering, 13th International Conference, CAiSE 2001, Interlaken, Switzerland, June 4-8, 2001, Proceedings* Bd. 2068, Springer, 2001. – ISBN 3–540–42215–3, S. 359–373

[78] Estublier, J. ; Belkhatir, N. ; Ahmed-Macer, M. ; Melo, W. L.: Process Centered SEE and Adele. In: Forte, G. (Hrsg.) ; Madhavji, N. H. (Hrsg.) ; Müller, H. A. (Hrsg.): *5th International Workshop Computer-Aided Software Engineering*, 1992, S. 156–165

[79] Estublier, J. ; Casallas, R. : The Adele Configuration Manager. In: [217], S. 99–133. – ISBN 0–471–94245–6

[80] Estublier, J. ; Casallas, R. : Three dimensional versioning. In: [72], S. 118–135

[81] Feiler, P. H.: Software process support through software configuration management. In: Perry, D. E. (Hrsg.): *Proceedings of the* 5^{th} *International Software Process Workshop*, 1989, S. 58–60

[82] Feiler, P. H.: Configuration Management Models in Commercial Environment / Software Engineering Institute, Carnegie-Mellon University. 1991 (CMU/SEI-91-TR-7 ADA235782). – Forschungsbericht

[83] Feiler, P. H. (Hrsg.): *Proceedings of the 3rd International Workshop on Software Configuration Management (SCM-3), Trondheim, Norway, June 12-14, 1991*. ACM Press, 1991

[84] Feldman, S. I.: Make – A program for maintaining computer programs. In: *Software – Practice and Experience* 9 (1979), März, Nr. 3, S. 255–265

[85] Fischer, B. ; Grosch, F.-J. ; Kievernagel, M. ; Zeller, A. : Die inferenzbasierte Softwareentwicklungsumgebung NORA / Techn. Universität Braunschweig. 1993/94 (93-09). – Informatik-Bericht

[86] Eclipse Foundation. *Eclipse*. http://www.eclipse.org/. 2004

[87] Fowler, G. ; Korn, D. ; North, S. ; Rao, H. ; Vo, K.-P. : Libraries and File System Architecture. In: Krishnamurthy, B. (Hrsg.): *Practical Reusable UNIX Software*. John Wiley & Sons, 1995, S. 25–90

[88] Fowler, G. ; Korn, D. ; Rao, H. : n-DFS: The Multiple Dimensional File System. In: [217], S. 135–154. – ISBN 0–471–94245–6

[89] Frühauf, K. ; Zeller, A. : Software Configuration Management: State of the Art. In: [75], S. 217–227

[90] Frühauf, K. ; Ludewig, J. ; Sandmayr, H. : *Software-Projektmanagement und -Qualitätssicherung*. 3. vdf Hochschulverlag AG, 2000

[91] Gamma, E. ; Helm, R. ; Johnson, R. ; Vlissides, J. : *Design Patterns: Elements of Reusable Object-oriented Software*. Reading : Addison Wesley, 1996. – ISBN 0–201–63361–2

[92] Glinz, M. : An Integrated Formal Model of Scenarios Based on Statecharts. In: Schäfer, W. (Hrsg.) ; Botella, P. (Hrsg.): *5th European Software Engineering Conference, Sitges, Spain, September 25-28, 1995, Proceedings* Bd. 989. Berlin - Heidelberg - New York : Springer–Verlag, 1995. – ISBN 3–540–60406–5, S. 254–271

[93] Godart, C. ; Canals, G. ; Charoy, F. ; Molli, P. : About Some Relationships Between Configuration Management, Software Process and Cooperative Work: The COO Environment. In: [72], S. 173–178

[94] Goldstein, I. P. ; Bobrow, D. G.: A Layered Approach to Software Design / Xerox Palo Alto Research Center. 1980 (CSL-80-5). – Forschungsbericht

[95] Grundy, J. C. ; Hosking, J. G.: Constructing Multi-View Editing Environments using MViews. In: Glinert, E. P. (Hrsg.) ; Olsen, K. A. (Hrsg.): *Proc. IEEE Symp. Visual Languages, VL*, IEEE Computer Society, 24–27 August 1993. – ISBN 0–8186–3970–9, S. 220–224

[96] Gulla, B. : A Browser for a Versioned Entity-Relationship Database. In: *International Workshop on Interfaces to Database Systems (IDS'92), Glasgow, Scotland*, 1992

[97] Gulla, B. ; Karlsson, E.-A. ; Yeh, D. : Change-Oriented Version Descriptions in EPOS. In: *Software Engineering Journal* 6 (1991), November, Nr. 6, S. 378–386

[98] Haase, O. : *Mengenorientierte Anfragen auf partiell zugreifbaren Datenbanken*, Universtität -GH- Siegen, Praktische Informatik, Diss., 1997

[99] Härder, T. ; Mahnke, W. ; Ritter, N. ; Steiert, H.-P. : Generating Versioning Facilities for a Design-Data Repository Supporting Cooperative Applications. In: *International Journal of Cooperative Information Systems (IJCIS)* 9 (2000), Nr. 1-2, S. 117–146

[100] Henrich, A. : P-OQL: an OQL-oriented Query Language for PCTE. In: *Proceedings of the 7th Conference on Software Engineering Environments (SEE '95), Noordwijkerhout, Niederlande*, IEEE Computer Society Press, 1995, S. 48–60

[101] Henrich, A. : *Management von Softwareprojekten.* Oldenbourg, 2002 (Lehr- und Handbücher der praktischen Informatik)

[102] Hesse, W. : RUP: A Process Model for Working with UML. In: Siau, K. (Hrsg.) ; Halpin, T. (Hrsg.): *Unified Modeling Language: Systems Analysis, Design and Development Issues.* Idea Publishing Group, 2001, Kapitel 4, S. 61–74

[103] van der Hoek, A. (Hrsg.): *Tenth International Workshop on Software Configuration Management (SCM-10) New Practices, New Challenges, and New Boundaries May 14-15, 2001 Toronto, Canada (a workshop of 23th ICSE 2001).* http://www.ics.uci.edu/~andre/scm10/, 2001

[104] van der Hoek, A. ; Heimbigner, D. ; Wolf, A. L.: A Generic, Peer-to-Peer Repository for Distributed Configuration Management. In: *Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany, March*, 1996

[105] Hunt, J. J. ; Tichy, W. F. *Selected Patterns for Software Configuration Management – Preliminary version*

[106] Hunt, J. J. ; Vo, K.-P. ; Tichy, W. F.: An Empirical Study of Delta Algorithms. In: [209], S. 49–66

[107] IBM. *Rational Rose.* http://www.ibm.com. 2004

[108] International Organization for Standardization: *ISO 9000: Quality management and quality assurance standards; Part 3: Guidelines for the application of ISO 9001 to the development, supply and maintenance of software.* Geneva, Switzerland: International Organization for Standardization, 1991

[109] International Organization for Standardization (ISO): *ISO/IEC 13719-1:1998: Information technology — Portable Common Tool Environment (PCTE) — Part 1: Abstract specification.* Geneva, Switzerland: International Organization for Standardization, 1998. – 460 S

[110] International Organization for Standardization (ISO): *ISO/IEC 13719-2:1998: Information technology — Portable Common Tool Environment (PCTE) — Part 2: C programming language binding.* Geneva, Switzerland: International Organization for Standardization, 1998. – 145 S

[111] International Organization for Standardization (ISO): *ISO/IEC 13719-3:1998: Information technology — Portable Common Tool Environment (PCTE) — Part 3: Ada programming language binding.* Geneva, Switzerland : International Organization for Standardization, 1998. – 161 S

[112] International Organization for Standardization (ISO): *Interface Definition Language*. Geneva, Switzerland: International Organization for Standardization (ISO), 1999

[113] International Organization for Standardization (ISO): *ISO 9001: Quality systems – Model for quality assurance in design/development, production, installation and servicing*. Geneva, Switzerland: International Organization for Standardization (ISO), 2000

[114] Jing, J. ; Helal, A. S. ; Elmagarmid, A. : Client-server computing in mobile environments. In: *ACM Computing Surveys* 31 (1999), Juni, Nr. 2, S. 117–157. – ISSN 0360–0300

[115] Johnson, M. K.: Diff, Patch, and Friends. In: *Linux Journal* 28 (1996), August. – ISSN 1075–3583

[116] Käfer, W. ; Schöning, H. : Mapping a Version Model to a Complex-Object Data Model. In: *Proceedings of the Eigth International Conference on Data Engineering (ICDE), Tempe, Arizona*. Los Alamitos, California : IEEE Computer Society Press, Februar 1992, S. 348–357

[117] Katz, R. H.: Toward a Unified Framework for Version Modeling in Engineering Databases. In: *ACM Computing Surveys* 22 (1990), Dezember, Nr. 4, S. 375–408. – ISSN 0360–0300

[118] Kay, M. H. ; Rivett, P. J. ; Walters, T. J.: The Raleigh Activity Model: Integrating Versions, Concurrency, and Access Control. In: Gray, P. M. D. (Hrsg.) ; Lucas, R. J. (Hrsg.): *Proceedings of Advanced Database Systems. 10th British National Conference on Databases, BNCOD 10* Bd. 618. Berlin - Heidelberg - New York : Springer–Verlag, Juli 1992. – ISBN 3–540–55693–1, S. 175–191

[119] Kelter, U. : Concurrency control for design objects with versions in CAD databases. In: *Information Systems* 12 (1987), Nr. 2, S. 137–143

[120] Kelter, U. : Integrationsrahmen für Software-Entwicklungsumgebungen. In: *Informatik-Spektrum* 16 (1993), Oktober, Nr. 5, S. 281–285. – Springer–Verlag

[121] Kelter, U. : *Einführung in H-PCTE*. Siegen: Universität-Gesamthochschule Siegen, Juni 1998

[122] Kelter, U. : *Reihe Informatik*. Bd. 51: *Parallele Transaktionen in Datenbanksystemen*. Mannheim [u.a.] : Bibliographisches Institut, 1985

[123] Kelter, U. : Transaktionskonzepte für Non-Standard-Datenbanksysteme. In: *Informationstechnik* 30 (1988), S. 17–26

[124] Kelter, U. : H-PCTE – a high-performance object management system for system development environments. In: *Proceedings COMPSAC Illinois, September 23-25*, IEEE Press, 1992, S. 45–50

[125] Kelter, U. : Fine-Grained Data in PCTE: Notions, Issues and Proposed Solutions. In: Lindquist, T. (Hrsg.) ; Kaehnemann, H. (Hrsg.): *Proceedings of the PCTE 1994 Conference*, 1994, S. 41–57

[126] Kelter, U. : Differenzen und Mischen von Dateien. In: *Softwaretechnik 2.* Praktische Informatik, Universität Siegen, 2002. – Lehrmodul zur Vorlesung

[127] Kelter, U. : Einführung in CVS. In: *Softwaretechnik 1.* Praktische Informatik, Universität Siegen, 2003. – Lehrmodul zur Vorlesung

[128] Kelter, U. : Einführung in das Konfigurationsmanagement. In: *Softwaretechnik 1.* Praktische Informatik, Universität Siegen, 2003. – Lehrmodul zur Vorlesung

[129] Kelter, U. : Dokumentdifferenzen. In: *Softwaretechnik 2.* Praktische Informatik, Universität Siegen, 2004. – Lehrmodul zur Vorlesung

[130] Kelter, U. ; Monecke, M. : Eine Architektur zur effizienten Konstruktion verteilter datenbankbasierter Anwendungen. In: *Proceedings der 5. Fachkonferenz Smalltalk und Java in Industrie und Ausbildung (STJA), 28.-30. September 1999, Erfurt*, 1999

[131] Kelter, U. ; Monecke, M. ; Platz, D. : Realisierung von verteilten Editoren in Java auf Basis eines aktiven Repositories. In: *Proceedings der GI-Fachtagung Java-Informations-Tage (JIT) 12.-13.11.1998, Frankfurt* GI, S. 340–353

[132] Kelter, U. ; Monecke, M. ; Platz, D. : Constructing Distributed SDEs using an Active Repository. In: *Proc. 1st Intl. Symposium on Constructing Software Engineering Tools (COSET '99); 17.-18.05.1999, Los Angeles, CA*, 1999, S. 149–158

[133] Kent, W. : An Overview of the Versioning Problem. In: Clifford, J. (Hrsg.) ; Lindsay, B. G. (Hrsg.) ; Maier, D. (Hrsg.): *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, May 31 - June 2, 1989, Portland, Oregon*, 1989, S. 5–7

[134] Kesim, F. N. ; Sergot, M. : Versioning of Objects in Deductive Databases. In: Ceri, S. (Hrsg.) ; Tanaka, K. (Hrsg.) ; Tsur, S. (Hrsg.): *Proceedings of the Third International Conference on Deductive and Object-Oriented Databases , DOOD '93, Phoenix, Arizona, USA, December 6-8, 1993* Bd. 760. Berlin - Heidelberg - New York : Springer–Verlag, 1993. – ISSN 0302–9743, S. 459–472

[135] Klahold, P. ; Schlageter, G. ; Unland, R. ; Wilkes, W. : A Transaction Model Supporting Complex Applications in Integrated Information Systems. In: Navathe, S. B. (Hrsg.): *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data, Austin, Texas, 28–31 May 1985*, 1985, S. 388–401

[136] Klingemann, J. ; Tesch, T. ; Wäsch, J. : Enabling Cooperation among Disconnected Mobile Users. In: *Proceedings of the Second IFCIS Conference on Cooperative Information Systems (CoopIS-97), Kiawah Island, South Carolina, USA, June 24-27*, IEEE Computer Society, 1997

[137] Kobialka, H.-U. ; Meyke, C. : Configurations are Versions, Too. In: *Proceedings of the 4th International Workshop on Software Configuration Management (Preprint), Baltimore, Maryland*, 1993

[138] Korth, H. ; Kim, W. ; Bancilhon, F. : On Long-Duration CAD Transactions. In: *Information Sciences* 46 (1988), S. 73–107

[139] Korth, H. F. ; Silberschatz, A. : *Database System Concepts.* 2. McGraw-Hill International, 1991 (Computer Science Series)

[140] Koskinen, J. ; Peltonen, J. ; Selonen, P. ; Systä, T. ; Koskimies, K. : Towards tool assisted UML development environments. In: Gyimothy, T. (Hrsg.): *Seventh Symposium on Programming Languages and Tools, SPLST'2001, Szeged, Hungary, June*, Szeged, Hungary: University of Szeged, 2001, S. 1–15

[141] Lago, P. ; Conradi, R. : Transaction Planning to Support Coordination. In: [72], S. 145–151

[142] Larsson, M. ; Crnkovic, I. : New Challenges for Configuration Management. In: [75], S. 232–243

[143] Leblang, D. B.: The CM Challenge: Configuration Management That Works. In: [217], S. 1–38. – ISBN 0–471–94245–6

[144] Leblang, D. B. ; Chase, Jr., R. P. ; Spilke, H. : Increasing Productivity with a Parallel Configuration Manager. In: *Proceedings of the International Workshop on Software Version and Configuration Control (SCM), Grassau, Germany*. Stuttgart : Teubner Verlag, Januar 1988, S. 21–37

[145] Lee, B. G. ; Narayanan, N. H. ; Chang, K. H.: An integrated approach to distributed version management and role-based access control in computer supported collaborative writing. In: *The Journal of Systems and Software* 59 (2001), November, Nr. 2, S. 119–134. – ISSN 0164–1212

[146] Lie, A. ; Conradi, R. ; Didriksen, T. M. ; Karlsson, E. ; Hallsteinsen, S. O. ; Holager, P. : Change Oriented Versioning in a Software Engineering Database. In: [216], S. 56–65

[147] Lin, Y.-J. ; Reiss, S. P.: Configuration management with logical structures. In: *Proceedings of the 18th International Conference on Software Engineering*, IEEE Computer Society Press, 1996, S. 298–307

[148] Lindholm, T. : *A 3-way Merging Algorithm for Synchronizing Ordered Trees - the 3DM merging and differencing tool for XML*, Helsinki University of Technology, Departement of Computer Science, Laboratory of Information Processing Science, Diplomarbeit, 2001

[149] Lippe, E. ; Florijn, G. : Implementation Techniques for Integral Version Management. In: America, P. (Hrsg.): *Proceedings of the 5th European Conference on Object-Oriented Programming, ECOOP '91, Geneva, Switzerland, July 15-19, 1991* Bd. 512, 1991. – ISSN 0302–9743, S. 342–359

[150] Lippe, E. ; van Oosterom, N. : Operation-based Merging. In: Weber, H. (Hrsg.): *SIGSOFT '92: Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments*, 1992, S. 78–87

[151] Llirbat, F. ; Simon, E. ; Tombroff, D. : Using Versions in Update Transactions: Application to Integrity Checking. In: *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases*, 1997, S. 96–105

[152] MacKay, S. A.: The State-of-the-Art in Concurrent, Distributed Configuration Management. In: [72], S. 180–193

[153] Magnusson, B. (Hrsg.): *Proceedings of the 8th International Symposium on System Configuration Management (SCM-8), Brussels, Belgium, July 20-21, 1998*. Bd. *1439*. Berlin - Heidelberg - New York : Springer–Verlag, 1998 (Lecture Notes in Computer Science (LNCS))

[154] Magnusson, B. ; Asklund, U. : Fine Grained Version Control of Configurations in COOP/Orm. In: [209], S. 31–48

[155] Magnusson, B. ; Asklund, U. ; Minör, S. : Fine-Grained Revision Control for Collaborative Software Development. In: *Proceedings of the ACM SIGSOFT '93 Symposium on the Foundations of Software Engineering, Los Angeles, California*, 1993, S. 33–41

[156] Mahler, A. : Variants: Keeping Things Together and Telling Them Apart. In: [217], S. 73–97. – ISBN 0–471–94245–6

[157] Maioli, C. ; Sola, S. ; Vitali, F. : Versioning Issues in a Collaborative Distributed Hypertext System / University of Bologna (Italy). Department of Computer Science. 1993 (BOLOGNA UBLCS-93-6). – Technical Report. – 14 S

[158] Marian, A. ; Abiteboul, S. ; Cobena, G. ; Mignet, L. : Change-Centric Management of Versions in an XML Warehouse. In: Apers, P. M. G. (Hrsg.) ; Atzeni, P. (Hrsg.) ; Ceri, S. (Hrsg.) ; Paraboschi, S. (Hrsg.) ; Ramamohanarao, K. (Hrsg.) ; Snodgrass, R. T. (Hrsg.): *Proceedings of the Twenty-seventh International Conference on Very Large Data Bases: Roma, Italy, 11–14th September, 2001*. Los Altos, CA 94022, USA : Morgan Kaufmann Publishers, 2001. – ISBN 1–55860–804–4, S. 581–590

[159] Meckenstock, A. G.: *Synchronisations- und Recoverykonzepte für Transaktionen in kooperativen Entwurfsumgebungen*. Shaker Verlag, Juni 1997

[160] Mei, H. ; Zhang, L. ; Yang, F. : A software configuration management model for supporting component-based software development. In: *ACM SIGSOFT Software Engineering Notes* 26 (2001), Nr. 2, S. 53–58. – ISSN 0163–5948

[161] Mens, T. : A State-of-the-Art Survey on Software Merging. In: *IEEE Transactions on Software Engineering* 28 (2002), Nr. 5, S. 449–462. – ISSN 0098–5589

[162] Meyers, S. : Difficulties in integrating multiview development systems. In: *IEEE Software* 8 (1991), Januar, Nr. 1, S. 49–57. – ISSN 0740–7459

[163] Midha, A. K.: Software Configuration Management for the 21st Century. In: *Bell Labs Technical Journal* Winter (1997), S. 154–165

[164] Miller, T. : Configuration Management with NSE. In: Long, F. (Hrsg.): *Software Engineering Environments, International Workshop on Environments, Chinon, France, September 1989* Bd. 467. Berlin - Heidelberg - New York : Springer–Verlag, September 1989, S. 99–106

[165] Mitschang, B. ; Härder, T. ; Ritter, N. : Conflict Management in CONCORD. In: *5th International Conference on Data and Knowledge Bases for Manufacturing and Engineering (DKSME'96), Tempe, Arizona*, 1996

[166] Monecke, M. : Komponentenbasierte Konstruktion flexibler Software-Entwicklungswerkzeuge. In: *erscheint in: Proc. GI-Fachtagung Informatik 2000, Junge Informatik, Berlin, September 2000*

[167] Monecke, M. : *Adaptierbare CASE-Werkzeuge in prozeßorientierten Software-Entwicklungsumgebungen*, Praktische Informatik, Universität Siegen, Diss., 2003

[168] Monk, S. R. ; Sommerville, I. : A Model for Versioning of Classes in Object-Oriented Databases. In: Gray, P. M. D. (Hrsg.) ; Lucas, R. J. (Hrsg.): *Proceedings of Advanced Database Systems. 10th British National Conference on Databases, BNCOD 10* Bd. 618. Berlin - Heidelberg - New York : Springer–Verlag, Juli 1992. – ISBN 3–540–55693–1, S. 42–58

[169] Munch, B. P. ; Larsen, J.-O. ; Gulla, B. ; Conradi, R. ; Karlsson, E.-A. : Uniform Versioning: The Change-Oriented Model. In: *Proceedings of the 4th International Workshop on Software Configuration Management (Preprint), Baltimore, Maryland*, 1993, S. 188–196

[170] Munson, J. P. ; Dewan, P. : A Flexible Object Merging Framework. In: *Proceedings of ACM CSCW'94 Conference on Computer-Supported Cooperative Work*, 1994 (Technologies for Sharing I), S. 231–242

[171] Myers, E. W.: An $O(ND)$ difference algorithm and its variations. In: *Algorithmica* 1 (1986), S. 251–256

[172] Nagl, M. : Software-Entwicklungsumgebungen: Einordnung und zukünftige Entwicklungslinien. In: *Informatik-Spektrum* 16 (1993), Oktober, Nr. 5, S. 273–280. – Springer–Verlag

[173] Nguyen, M. N. ; Conradi, R. : Workspace Management: Supporting Cooperative Work. In: *Proceedings International Conference for Young Computer Scientists (ICYCS), Bejing, China*, 1993

[174] Ohst, D. : *Ein 64-Bit-Speicherkonzept für das hauptspeicherorientierte Objektmanagementsystem H-PCTE*, Praktische Informatik, Universität Siegen, Diplomarbeit, 1998

[175] Ohst, D. ; Kelter, U. : A Fine-grained Version and Configuration Model in Analysis and Design. In: *Proc. of the IEEE International Conference on Software Maintenance 2002 (ICSM 2002), 3-6 October 2002, Montreal, Canada*, 2002

[176] Ohst, D. ; Welle, M. ; Kelter, U. : Differences between Versions of UML Diagrams. In: *Proc. of the fourth joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2003), September 1-5, 2003, Helsinki, Finland*, ACM Press, 2003, S. 227–236

[177] Olson, T. G. ; Gates, L. P. ; Mullaney, J. L. ; Over, J. W. ; Reizer, N. R. ; Kellner, M. I. ; Phillips, R. W. ; DiGennaro, S. J.: A Software Process Framework for the SEI Capability Maturity Model: Repeatable Level / Software Engineering Institute, Carnegie-Mellon University. Pittsburgh, Pennsylvania : Software Engineering Institute, Carnegie-Mellon University, Juli, 2 1993 (CMU/SEI-93-SR-007). – SPECIAL REPORT CMU/SEI-93-SR-007

[178] OMG: *Meta-Object Facility ((MOF)), version 1.4.* : OMG, 2002

[179] OMG: *OMG-XML Metadata Interchange (XMI) Specification, v1.2.* : OMG, Januar 2002. – Formal version of the XMI specification, v1.2. This document supersedes formal/00-11-02.

[180] OMG: *OMG-XML Metadata Interchange (XMI) Specification, v1.2.* : OMG, jan 2002. – Formal version of the XMI specification, v1.2. This document supersedes formal/00-11-02.

[181] OMG: *Unified Modeling Language Specification.* OMG, März 2003. – Version 1.5 formal/03-03-01

[182] Oquendo, F. ; Berrada, K. ; Gallo, F. ; Minot, R. ; Thomas, I. : Version management in the PACT integrated software engineering environment. In: Ghezzi, C. (Hrsg.); McDermid, J. A. (Hrsg.): *Proceedings of the 2nd European Software Engineering Conference* Bd. 387. Berlin - Heidelberg - New York : Springer–Verlag, September 1989. – ISBN 3–540–51635–2, S. 222–242

[183] OReilly, C. ; Morrow, P. ; Bustard, D. : Improving Conflict Detection in Optimistic Concurrency Control Models. In: Westfechtel, B. (Hrsg.) ; van der Hoek, A. (Hrsg.): *Software Configuration Management ICSE Workshops SCM 2001 and SCM 2003 Toronto, Canada, May 14-15, 2001 and Portland, OR, USA, May 9-10, 2003* Bd. 2649. Berlin - Heidelberg - New York : Springer–Verlag, 2003. – Selected Papers, S. 191–

[184] Paulk, M. C.: Comparing ISO 9001 and the Capability Maturity Model for software. In: *Software Quality Journal* 2 (1993), Dezember, Nr. 4, S. 245–256

[185] Paulk, M. C. ; Curtis, B. ; Chrissis, M. B.: Capability Maturity Model for Software, Version 1.1 / Software Engineering Institute, Carnegie-Mellon University. Pittsburgh, PA : Software Engineering Institute, Carnegie-Mellon University, 1993 (CMU/SEI-93-TR-24). – Forschungsbericht

[186] Perry, D. E. ; Siy, H. P. ; Votta, L. G.: Parallel changes in large-scale software development: an observational case study. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 10 (2001), Nr. 3, S. 308–337. – ISSN 1049–331X

[187] Platz, D. : *Ein Werkzeugtransaktionskonzept für Objekt-Managementsysteme als Basis von Software-Entwicklungsumgebungen*, Praktische Informatik, Universität Siegen, Diss., Juni 1999

[188] Platz, D. ; Kelter, U. : Konsistenzerhaltung von Fensterinhalten in einer Repository-basierten SEU. In: *Proceedings GI-Fachtagung Softwaretechnik 96, Koblenz, 12.-13.09.1996* GI-Fachausschuš 2.1, 1996

[189] tkdiff Project. *tkdiff.* http://sourceforge.net/projects/tkdiff/. 2004

[190] Ramakrishnan, R. ; Ram, D. J.: Modeling Design Versions. In: Vijayaraman, T. M. (Hrsg.) ; Buchmann, A. P. (Hrsg.) ; Mohan, C. (Hrsg.) ; Sarda, N. L. (Hrsg.): *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, Morgan Kaufmann, 1996. – ISBN 1–55860–382–4, S. 556–566

[191] Rho, J. ; Wu, C. : An Efficient Version Model of Software Diagrams. In: *Proc. 5th Asia-Pacific Software Engineering Conf., 2-4 December 1998 in Taipei, Taiwan, ROC* IEEE Computer Society, 1998

[192] Ritter, N. : The C3-Locking-Protocol - A Concurrency Control Mechanism for Design Environments. In: *Softwaretechnik in Automation und Kommunikation (STAK'96)*, 1996

[193] Robbins, J. E. ; Redmiles, D. F.: Cognitive support, UML adherence, and XMI interchange in Argo/UML. In: *Information and Software Technology* 42 (2000), Nr. 2, S. 79–89

[194] Rochkind, M. J.: The source code control system. In: *Proceedings of the 1st National Conference on Software Engineering*, IEEE Computer Society Press, September 1975. – This paper was cited as the best/most influential paper by the program committee for ICSE 11 in 1985, S. 37–43

[195] Rochkind, M. J.: The Source Code Control System. In: *IEEE Transactions on Software Engineering* 1 (1975), Dezember, Nr. 4, S. 364–370

[196] Rockel, I. : *Versionierungs- und Mischkonzepte für UML Diagramme*, University of Paderborn, Department of Mathematics and Computer Science, Paderborn, Germany, Diplomarbeit, September 2000

[197] Sachweh, S. ; Schäfer, W. : Objektorientierte Spezifikation und Realisierung einer Umgebung für das Konfigurationsmanagement. In: *Softwaretechnik'96, Koblenz*, 1996, S. 49–56

[198] Saheb, M. ; Karoui, R. ; Sedillot, S. : Open Nested Transactions: A Support for Increasing Performance and Multi-tier Applications. In: Saake, G. (Hrsg.) ; Schwarz, K. (Hrsg.) ; Türker, C. (Hrsg.): *Transactions and Database Dynamics 8th International Workshop on Foundations of Models and Languages for Data and Objects, Dagstuhl Castle, Germany, September 1999. Selected Papers* Bd. 1773. Berlin - Heidelberg - New York : Springer–Verlag, 1999, S. 167–192

[199] Santoyridis, I. ; Carnduff, T. W. ; Gray, W. A. ; Miles, J. C.: An Object Versioning System to Support Collaborative Design within a Concurrent Engineering Context. In: Small, C. (Hrsg.) ; Douglas, P. (Hrsg.) ; Johnson, R. (Hrsg.) ; King, P. (Hrsg.); Martin, N. (Hrsg.): *Advances in Databases 15th British National Conference on Databases, BNCOD 15, Birkbeck College, University of London, UK* Bd. 1271. Berlin - Heidelberg - New York : Springer–Verlag, Juli 1997, S. 184–199

[200] Sarma, A. ; Noroozi, Z. ; van der Hoek, A. : Palantir: Raising Awareness among Configuration Management Workspaces. In: *Proceedings of the Twenty-Fifth International Conference on Software Engineering, Portland, Oregon, May 2003*, 2003

[201] Schönberger, S. ; Keller, R. K. ; Khriss, I. : Algorithmic Support for Model Transformation in Object-Oriented Software Development. In: *Concurrency and Computation: Practice and Experience* 13 (2001), April, Nr. 5, S. 351–383. – Object Systems Section

[202] Seemann, J. : Extending the Sugiyama Algorithm for Drawing UML Class Diagrams: Toward Automatic Layout of Object-Oriented Software Diagrams. In: Di Battista, G. (Hrsg.): *Proc. 5th Int. Symp. Graph Drawing, GD*, Springer-Verlag, 18–20 September 1997 (Lecture Notes in Computer Science, LNCS 1353). – ISBN 3–540–63938–1, S. 415–424

[203] Selkow, S. M.: The tree-to-tree editing problem. In: *Information Processing Letters* 6 (1977), Dezember, Nr. 6, S. 184–186. – ISSN 0020–0190

[204] Sellentin, J. ; Frank, A. ; Mitschang, B. : TOGA — A Customizable Service for Data-Centric Collaboration. In: Jarke, M. (Hrsg.) ; Oberweis, A. (Hrsg.): *Proceedings of the 11th International Conference on Advanced Information Systems Engineering,*

CAiSE'99, Heidelberg, Germany, June 14-18, 1999 Bd. 1626, 1999. – ISSN 0302–9743, S. 301–316

[205] Selonen, P. : Set Operations for Unified Modeling Language. In: *Proceedings of the Eight Symposium on Programming Languages and Tools, SPLST'2003, Kuopio, Finland, June*, Kuopio, Finland: University of Kuopio, 2003, S. 70–81

[206] Selonen, P. ; Koskimies, K. ; Sakkinen, M. : How to Make Apples from Oranges in UML. In: Sprague, Jr., R. H. (Hrsg.): *Proc. 34th Annual Hawaii International Conference on System Sciences (HICSS-34)*, IEEE Computer Society, 2001

[207] Snelting, G. ; Zeller, A. : Inferenzbasierte Werkzeuge in NORA. In: *Proceedings of Softwaretechnik '93*, 1993 (GI Softwaretechnik Trends)

[208] Rational Software: *The Rational Unified Process, version 5.0*. Cupertino, CA, 1998.: Rational Software, 1998

[209] Sommerville, I. (Hrsg.): *Proceedings of the 6th International Workshop on Software Configuration Management (SCM-6), Berlin, Germany, March 25-26, 1996*. Bd. *1167*. Berlin - Heidelberg - New York : Springer–Verlag, 1996 (Lecture Notes in Computer Science (LNCS))

[210] SUN. *Sun Product Documentation, Teamware, Filemerge*. docs.sun.com. 2004

[211] SunSoft: SPARCworks/TeamWare Solutions Guide / SunSoft. 1994 (801-7186-05). – Forschungsbericht

[212] Tai, K.-C. : The Tree-to-Tree Correction Problem. In: *Journal of the ACM* 26 (1979), Juli, Nr. 3, S. 422–433. – ISSN 0004–5411

[213] Thomas, I. : PCTE interfaces: Supporting tools in software-engineering environments. In: *IEEE Software* 6 (1989), November, Nr. 6, S. 15–23. – ISSN 0740–7459

[214] Thomas, I. : Version and Configuration Management on a Software Engineering Database. In: [216], S. 23–25

[215] Thomas, I. ; Nejmeh, B. A.: Definitions of Tool Integration for Environments. In: *IEEE Software* 9 (1992), März, Nr. 2, S. 29–35. – ISSN 0740–7459

[216] Tichy, W. F. (Hrsg.): *Proceedings of the 2nd International Workshop on Software Configuration Management, October 24, 1989, Princeton,New Jersey, USA*. ACM Press, 1989

[217] Tichy, W. (Hrsg.): *Configuration Management*. Baffins Lane, Chichester, West Sussex PO19 1UD, England : John Wiley and Sons, Ltd., 1994. – ISBN 0–471–94245–6

[218] Tichy, W. F.: A Data Model for Programming Support Environments. In: Schneider, H.-J. (Hrsg.) ; Wasserman, A. I. (Hrsg.): *Proceedings of the IFIP WG 8.1 Working Conference on Automated Tools for Information System Design and Development, New Orleans, Louisiana*, 1982, S. 31–48

[219] Tichy, W. F.: Design, Implementation, and Evaluation of a Revision Control System. In: *Proceedings of the 6th ACM/IEEE International Conference on Software Engineering (ICSE), Tokyo, Japan*, 1982, S. 58–67

[220] Tichy, W. F.: The String-to-String Correction Problem with Block Moves. In: *ACM Transactions on Computer Systems* 2 (1984), November, Nr. 4, S. 309–321

[221] Tichy, W. F.: RCS: A System for Version Control. In: *Software Practice and Experience* 15 (1985), Juli, Nr. 7, S. 637–654. – ISSN 0038–0644

[222] Vitali, F. : Versioning Hypermedia. In: *ACM Computing Survey*, 1999

[223] Wagner, T. A. ; Graham, S. L.: Integrating incremental analysis with version management. In: *Proceedings of ESEC '95 - 5th European Software Engineering Conference. Sitges, Spain. 25-28 Sept. 1995.* Berlin - Heidelberg - New York : Springer–Verlag, 1995, S. 205–18

[224] Wagner, T. A. ; Graham, S. L.: Efficient self-versioning documents. In: *Proceedings IEEE COMPCON 97. Digest of Papers. San Jose, CA, USA. IEEE Comput. Soc. 23-26 Feb. 1997.*, EEE Comput. Soc. Press, Los Alamitos, CA, USA, 1997, S. 62–67

[225] Wakeman, L. ; Jowett, J. : *PCTE - The standard for open repositories.* Hemel Hempstead, Hertfordshire, UK : Prentice Hall, 1993

[226] Walborn, G. D. ; Chrysanthis, P. K.: PRO-MOTION: Management of Mobile Transactions. In: *Proceedings of the 11th ACM Annual Symposium on Applied Computing, Special Track on Database Technology (SAC), San Jose, California*, 1997, S. 101–108

[227] Wang, Y. ; DeWitt, D. J. ; Cai, J.-Y. : X-Diff: An Effective Change Detection Algorithm for XML Documents. In: *19th International Conference on Data Engineering, March 5 - March 8, 2003 - Bangalore, India*, 2003

[228] Wäsch, J. : *Transactional Support for Cooperative Applications*, Technische Universität Darmstadt, Diss., Juni 1999

[229] Weber, D. W.: Change Sets Versus Change Packages: Comparing Implementations of Change-Based SCM. In: [52], S. 25–35

[230] Wehren, J. : *Ein XMI-basiertes Differenzwerkzeug für UML-Diagramme*, Universität Siegen, Diplomarbeit, 2004. – In Vorbereitung

[231] Welle, M. : *Ein konfigurierbares graphisches Differenz-Werkzeug für feinkörnig modellierte Analyse- und Entwurfs-Diagramme*, Praktische Informatik, Universität Siegen, Diplomarbeit, October 2002

[232] Welle, M. : *Ein interaktives Mischwerkzeug für UML-Diagramme*, Praktische Informatik, Universität Siegen, Diplomarbeit, September 2004. – In Vorbereitung

[233] Westfechtel, B. : *Informatik-Fachberichte*. Bd. 280: *Revisions- und Konsistenzkontrolle in einer integrierten Softwareentwicklungsumgebung*. Berlin - Heidelberg - New York : Springer–Verlag, 1991

[234] Westfechtel, B. : Structure-Oriented Merging of Revisions of Software Documents. In: [83], S. 68–79

[235] Westfechtel, B. ; Conradi, R. : Software Configuration Management and Engineering Data Management: Differences and Similarities. In: [153], S. 95–106

[236] White, B. A.: *Software Configuration Management Strategies and Rational ClearCase*. Addison-Wesley, 2000. – ISBN 0–201–60478–7

[237] Whitehead, Jr., E. J.: SCM and Hypertext Versioning: A Compelling Duo. In: [209],

[238] Widjojo, S. ; Hull, R. ; Wile, D. : A Specificational Approach to Merging Persistent Object Bases. In: Dearle, A. (Hrsg.) ; Shaw, G. (Hrsg.) ; Zdonik, S. (Hrsg.): *Implementing Persistent Object Bases: Principles and Practice (The Fourth Intl. Workshop on Persistent Object Systems)*. Martha's Vineyard, MA : Morgan-Kaufmann, September 1990, S. 267–278

[239] Yang, W. : How to Merge Program Texts. In: *The Journal of Systems and Software* 27 (1994), November, Nr. 2, S. 129–141

[240] Zeller, A. : *Configuration Management with Version Sets: A Unified Software Versioning Model and its Applications*, Fachbereich für Mathematik und Informatik, Technischen Universität Braunschweig, Diss., April 1997

[241] Zeller, A. ; Snelting, G. : Handling Version Sets through Feature Logic / Institut für Programmiersprachen und Informationssyteme, Abteilung Softwaretechnologie, Technische Universität Gausstrasse 17, D-38092 Braunschweig, Deutschland. 1995 (94-04). – Forschungsbericht

[242] Zeller, A. ; Snelting, G. : Unified Versioning through Feature Logic / Institut für Programmiersprachen und Informationssyteme, Abteilung Softwaretechnologie, Technische Universität Braunschweig. 1997 (96-01). – Forschungsbericht

[243] Zhang, K. ; Shasha, D. : Simple fast algorithms for the editing distance between trees and related problems. In: *SIAM Journal of Computing* 18 (1989), S. 1245–1262

[244] Zuendorf, A. ; Wadsack, J. ; Rockel, I. : Merging Graph-Like Object Structures. In: [103],

Glossar

Änderungsbasierte Versionierung Die Verwaltung von Versionen auf Basis der durchgeführten Änderungen an den Dokumenten.

Basisdokumente Dokumente, die als Eingabe für die Differenzberechnung dienen.

Delta Alle Differenzen, in denen sich zwei Dokumente unterscheiden, bezeichnet man als Delta.

Diagrammelement Elemente eines Diagramms, die innerhalb eines Diagramms dargestellt werden. Ein Diagrammelement kann ggf. gleichzeitig die Daten mehrerer Modellobjekte darstellen, z. B. ist ein Klassensymbol durch einen Teilbaum im Editiermodell repräsentiert.

Differenz Unter einer Differenz versteht man einen Dokumentteil eines Dokumentes A, das keinen korrespondierenden Dokumentteil in einem Dokument B besitzt oder den inhaltlichen Unterschied zwischen zwei korrespondierenden Dokumentteilen.

Differenzen Der Begriff Differenzen bezeichnet in dieser Arbeit eine Teilmenge der in einem Delta zusammengefaßten Differenzen.

Dokumentteil Ein Dokumentteil bezeichnet einen Teil eines Dokumentes (oder anders formuliert: eine Komponente des Editiermodells) und ist abhängig von den Dokumenttypen. Bei Textdokumenten sind Dokumentteile z. B. Folgen von Zeilen, und bei UML-Diagrammen in der Repräsentation eines Editiermodells können einzelne Objekte oder Teilbäume einen Dokumentteil bilden.

Edit-Skript Folge von Änderungsanweisungen oder Differenzen, die eine Dokumentversion in eine andere überführen.

Editiermodell Eine werkzeugspezifische Realisierung des abstrakten Syntaxbaums eines Dokumentes, die sich in einigen Details vom abstrakten Syntaxbaum unterscheidet. Es repräsentiert die modellierten Eigenschaften eines Dokumentes.

Gerichtetes Delta Eine spezielle Form eines Delta, welche das Delta in Form von Änderungsanweisungen beschreibt, die das eine Dokumente in das andere Dokument überführen.

Konfiguration Die Menge der Versionen von mehreren Software-Elementen wie z. B. Objekte oder Dateien, die zusammen ein konsistentes Software-Produkt bilden, bezeichnet man als Konfiguration.

Konflikt siehe Mischkonflikt.

Korrespondierende Dokumentteile Dokumentteile zweier Dokumente korrespondieren wenn sie hinreichend viele übereinstimmende Merkmale besitzen. Diese allgemeine Definition läßt sich bei versionierten Dokumenten präzisieren: Je ein Dokumentteil eines Dokumentes A und eines Dokumentes B korrespondieren, wenn sie durch Änderungen an einem Dokumentteil eines Dokumentes C aus diesem hervorgegangen sind und das Dokument C ein gemeinsamer Vorgänger von A und B ist.

Mischkonflikt Ein Konflikt ist eine Differenz eines gerichteten Deltas, für die nicht automatisch entschieden werden kann, ob der Zustand vor oder nach Anwenden der Differenz in das Mischdokument übernommen werden soll.

Mischversion Auch Mischdokument genannt, bezeichnet das Dokument bzw. die Version, die durch das Mischen von zwei Dokumenten/Versionen entstanden ist.

Modellobjekte Objekte, die in Abhängigkeit vom verwendeten Editiermodell, die Nutzdaten und deren Struktur repräsentieren.

Pre-Misch-Diagramm Die graphische Darstellung eines Diagramms in dem bereits automatisch entscheidbare Differenzen vorläufig gemischt wurden.

Pre-Misch-Editiermodell Das Editiermodell des Pre-Misch-Diagramms.

Revisionen Linear aufeinander folgende Versionen eines Dokuments bezeichnet man als Revisionen. Im Gegensatz zu Varianten treten sie nicht zeitgleich auf.

Software Configuration Management bezeichnet ein Teilgebiet der Softwareentwicklung, welches sich insbesondere mit der Verwaltung von Dateien, deren Versionen und Beziehungen untereinander beschäftigt.

Software-Produkt bezeichnet alle Dokumente, die während der Softwareentwicklung erstellt werden. Neben dem Quelltext gehören auch Analyse- und Entwurfsdokumente hierzu, wie sie durch die UML beschrieben werden.

Softwarekomponenten sind modulare, verteilbare und austauschbare Teile eines Systems, welche eine Implementierung beinhalten und eine Schnittstelle nach außen anbieten.

Syntaktischer Konflikt Ein syntaktischer Konflikt ist *kein* Konflikt im eigentlichen Sinn. An ihm sind Änderungen an zwei oder mehreren Elementen beteiligt und er steht im Widerspruch zu den Konsistenzkriterien des modellierten Dokumentes. Ein Beispiel sind Namenskollisionen, die durch konkurrierende Änderungen verschiedener Diagrammelemente aufgetreten sind.

Syntaxbaum Der Begriff Syntaxbaum bezeichnet in dieser Arbeit den Spannbaum eines Editiermodells, der durch die Komponentenbeziehungen aufgespannt wird. Er entspricht *nicht* dem abstrakten Syntaxbaum.

Variante Zwei Versionen eines Dokumentes, die zeitgleich existieren und sich in bestimmten Eigenschaften unterscheiden.

Vereinigungsdiagramm Die graphische Darstellung der Differenzen und Gemeinsamkeiten zweier Versionen eines Diagramms, indem beide Versionen überlagert angezeigt werden.

Version Mehrere Zustände eines Dokumentes, die durch sequentielle oder parallele Bearbeitung aus einem Ursprungszustand entstanden sind.

Zustandsbasierte Versionierung Die Verwaltung von Versionen eines Dokumentes, die auf den Zuständen vor und nach den durchgeführten Änderungen basiert.

Index