Studienarbeit

Entwurf und Dokumentation eines Life Cycle Modells in einem FBS System

Alexandre Hanft

Betreuer:
Prof. Dr. Hans-Dieter Burkhard, Mirjam Minor
Humboldt-Universität zu Berlin, Institut für Informatik
Lehr- und Forschungsgebiet Künstliche Intelligenz
Unter den Linden 6, 10099 Berlin

Berlin, den 14.09.2004

Erläuterungen zur Typographie

Allgemeine *Hervorhebungen erfolgen gesperrt kursiv*.
Verweise auf andere Abschnitte erfolgen ebenso kursiv.
In `Courier New` werden Benutzereingaben, Code und Dateinamen dargestellt.
Aufgeführte Dialoge und besondere Termini wie «case authoring» werden in französische Anführungszeichen eingeklammert.
Variable Bezeichnungen wie in Bestandsliste*_AnuName_Datum*.xls werden kursiv ausgezeichnet.
Literaturverweise sind in eckige Klammern gefasst [Lenz99] und optional mit einer Seitenangabe versehen: [Richter2003, S. 412].

Inhaltsverzeichnis

1 Einleitung

1.1 Einleitung Fallbasiertes Schließen

Die Methode des Fallbasierten Schließens (kurz FBS, engl Case-Based Reasoning, CBR) zählt zu den wissensbasierten Systemen auf dem Gebiet der Künstlichen Intelligenz. Dabei wird zur Lösung eines Problems auf schon gemachte Erfahrungen bei der Lösung ähnlicher Probleme zurückgegriffen [Richter2003].
Insbesondere das Textuelle Fallbasierte Schließen, welches vornehmlich mit schwach strukturiertem Wissen in natürlichsprachlicher Textform arbeitet, wurde schon in vielen Projekten am Lehrstuhl für Künstlichen Intelligenz des Instituts für Informatik untersucht [LenzEtal98].

1.1.1 Einführung in das Fallbasiertes Schließen

Ein FBS System besteht im Groben aus einer Fallbasis, die eine Menge von sogenannten Fällen enthält, einer Ähnlichkeitsmodellierung und einer Retrieval genannten „Suchmaschine“. Die Interaktion des Akteurs, welcher menschlicher Natur als auch ein Agent sein kann, mit dem FBS geschieht durch Fälle. Ein Fallbasiertes System kann grafisch so dargestellt werden (Bild 1):

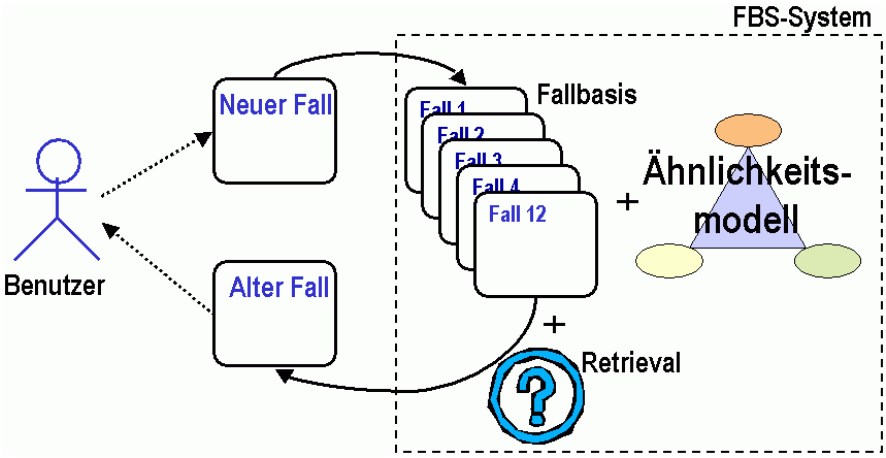

Bild 1: Einfaches Modell eines Fallbasierten Schließen Systems

Ein Akteur legt neues Wissen bzw. Erfahrung in Form eines neuen Falls in der Fallbasis ab. Initial kann die Fallbasis auch durch Aufbereitung anderer verfügbarer Dokumente aufgebaut werden. Auf diese vorherigen Erfahrungen greift er später in Form von „alten“ Fällen zu. Dazu stellt er an das System eine Anfrage, Query genannt, die einen unvollständigen Fall darstellt. Mithilfe der Ähnlichkeitsmodellierung werden durch das Retrieval aus der Fallbasis ähnliche Fälle zur aktuellen Anfrage herausgesucht.
Ähnlichkeit im Fallbasierten Schließen ist als a-priori-Kriterium der Ersatz für die Kriterien Nützlichkeit und Akzeptanz, welche erst nach dem

Verwenden des herausgesuchten Falls bewertbar sind. Die Grundannahme des FBS dabei ist, dass ähnliche Probleme auch ähnliche Lösungen haben, die alten Lösungen also adäquat für das aktuelle Problem verwendet werden können bzw. für die Problemlösung nützlich sind.
Die Ähnlichkeitsmodellierung besteht aus einem Indexlexikon, Information Entities (Informationseinheiten) und Ähnlichkeitsbeziehungen. Eine Information Entity (IE) bezeichnet die kleinste im FBS modellierte Wissenseinheit, einen atomaren Begriff. Jeder Fall wie auch die Anfrage wird (in Form von Relevanzkanten) durch eine Menge von IE's repräsentiert. Die Ähnlichkeitskanten spannen Ähnlichkeitsbeziehungen zwischen diesen IE's auf. Für das Retrieval gibt es verschiedene Techniken, z.B. kd-Bäume oder Case-Retrieval-Netze (CRN). Bei CRNs sind die Fälle beispielsweise durch ein Case Retrieval Net [Lenz99] so im Arbeitsspeicher organisiert, dass eine Suche über einen Spreading Activation Mechanismus sehr schnell erfolgt.

Die für das Fallbasierte Schließen wohl am meisten zitierte Charakterisierung ist der „4-R-Zyklus" von Amodt & Plaza [AamodtPlaza94]. Er beschreibt in Bild 2 das FBS als einen sich wiederholenden Kreis, der aus vier Prozessen besteht: Retrieve, Reuse, Revise und Retain.

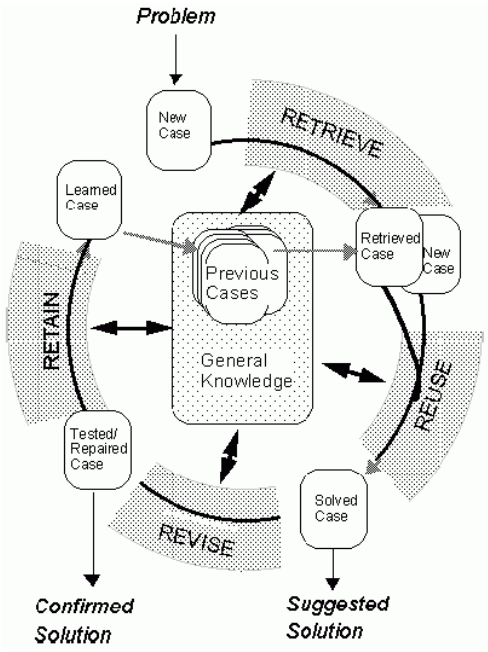

Bild 2: 4 R Zyklus nach Aamodt & Plaza (1994)

Retrieve besteht genau genommen aus zwei Teilen:

- dem Erzeugen eines unvollständigen Neuen Falls als Query mit der aktuellen Problemstellung und dem
- Retrieval: Suche nach ähnlichen Fällen.

Reuse bezeichnet die

- Benutzung des Wissens aus gefundenen Fällen für den Neuen Fall, (z.B. Übernahme der Lösung, Adaption durch Regeln).

Revise besteht aus dem

- Einsatz der vorgeschlagenen Lösung, der zu einer Bewertung des Neuen Falls führt. Evtl. muss die vorgeschlagene Lösung aufgrund der Praxis angepasst bzw. geändert werden.

Die anfängliche Problemstellung und die tatsächlich verwendete Lösung ergeben zusammen den kompletten neuen Fall.
Beim Retain wird

- der neue Fall in der Fallbasis abgespeichert, so dass er für das nächste Retrieval mit zur Verfügung steht.

Das im Folgenden vorgestellte Projekt TestManager führt genau genommen davon nur die Prozesse Retrieval und Retain aus, die verbleibenden Prozesse Reuse und Revise muss der Benutzer selbst erledigen.

1.2 Einleitung Projekt TestManager

Das Projekt beschäftigt sich mit der Unterstützung bei der Erstellung und Verwaltung von Testfällen für den Test einer Software aus der Erdgas-Domäne. Das System soll den Nutzern als verteiltes Client-Server System im Intranet zur Verfügung stehen. Aufbauend auf dem bestehenden System ExperienceBook [Kunze98] wurde dafür das System TestManager geschaffen.

1.3 Unterschiede zu bestehendem System

1.3.1 Ständig wachsende Fälle

Testfälle sind im Gegensatz zu Fällen in bisherigen Systemen wie ExperienceBook oder SIMATIC [LenzEtal99] nicht statisch, sondern entstehen und verändern sich während aller Phasen der Softwareentwicklung. Erst sind Software-Tests sehr unspezifisch, enthalten kaum mehr als den Titel und eine grobe Testaufgabe. Später werden die Tests an die Gegebenheiten der zu testenden Software angepasst, konkrete Schritte zur Testdurchführung notiert. Weitere Anpassungen und Ergänzungen an der Test-Spezifikation sind nach der Durchführung des Tests notwendig. Schließlich wird nach dem bestandenem Revisionstest der Status des Testfalls auf „erfolgreich ausgeführt“ geändert.

1.3.2 Alle Benutzer sind Experten und Autoren

Die Benutzer des TestManager sind gleichberechtigte Softwaretester bzw. Entwickler, sie sollen die Fälle ohne die Mithilfe eines Administrators einpflegen und ändern können. Änderungen sollen möglichst unmittelbar danach auch für die anderen Benutzer sichtbar sein um mehrfache Eingaben von Tests zu vermeiden. Daraus folgt auch die Anforderung, dass gleichzeitiges, konkurrierendes Editieren am gleichen Fall erkannt, ermöglicht und im Konfliktfall behandelt werden muss. Unabhängig davon soll derselbe Fall einem anderen Benutzer als Retrieval Ergebnis präsentiert werden können.

1.3.3 Weitere Unterschiede

Wegen des fehlenden Administrators müssen auch das Indexlexikon und die Ähnlichkeitskanten durch den Benutzer gepflegt werden können, was aber nicht Gegenstand dieser Arbeit ist.
Ein Fall soll mehreren Kategorien und Themen zugeordnet werden können, so dass bei den Retrieval-relevanten Attributen mehrwertige Attribut-Werte-Paare möglich sein müssen.
Weitere Anpassungen des Fallformats waren in dem für ein neues Projekt üblichen Rahmen.

1.3.4 Woraus besteht ein Fall?

Ein Fall besteht aus Texten und enthält:

- einen Titel,
- Testbeschreibung,
- Instruktion zur Testausführung,
- Erwartetes Ergebnis des Tests,
- Zeit und Autor der Erstellung und Änderung,
- sowie Zuordnungen der betroffenen Kategorie und des Themenbereichs,
- Status ob erfolgreich ausgeführt und
- Projektzuordnung

Ein Beispiel wird in *2.4 Syntax eines Falls auf Seite 13* vorgestellt.

1.4 Das Attribut-Werte-Paar

In fallbasierten Systemen werden oft die sogenannten Attribut-Werte-Paare (kurz AVP) [Richter2003, S. 412] eingesetzt. Es besteht aus einem Attributnamen und einem dazugehörigen Attributwert. Ersterer als auch beide können durch eine Auswahl vordefinierter Begriffe eingeschränkt sein. In der Darstellung werden beide meist durch ein ´=´ getrennt:
`Attributtext = Werttext`
AVPs können einfache oder mehrfache AVPs sein. Bei einem einfachem AVP ist das mehrfache Vorkommen des gleichen Attributnamen (mit unterschiedlichen Werten) nicht erlaubt, bei mehrfachen schon. Fälle haben in der Regel mehrere AVPs, bei mehrwertigen können sie dann mehrere Paare mit dem gleichen Attributnamen haben.

2 Der Entwurf des DataManagers

Die Komponente DataManager ist die Schnittstelle des TestManagers zu den Falldaten. Lesen, Abspeichern und Löschen der Testfälle erfolgt ausschließlich über den DataManager. Um die gleichzeitige Verwendung von Fällen in Retrieval- und Maintenance-Prozessen zu gewährleisten, unterscheidet DataManager zwischen unterschiedlichen Sichtweisen, „alten“ und „neuen“ Fällen und implementiert eine flache Versionskontrolle. Der Begriff *maintenance* wird in dieser Arbeit anstelle der Übersetzung Wartung verwendet, da er in der wissenschaftlichen Literatur der gebräuchliche Fachbegriff ist.
Einen Überblick der Umgebung des DataManagers im TestManager zeigt Bild 3: Der MailRoomServer ist die zentrale Schnittstelle zur Weboberfläche und leitet auf Fälle bezogene Anfragen an den DataManager weiter. Im Falle einer Suche (Retrieval) wird die Anfrage an den RetrievalServer weitergeleitet, der für die Präsentation des Ergebnisses selbst Anfragen an den DataManager stellt. Ebenso fordert er beim Aufbau des CRN die Fälle zum Parsen der IEs ab.

Systemumgebung des DataManager

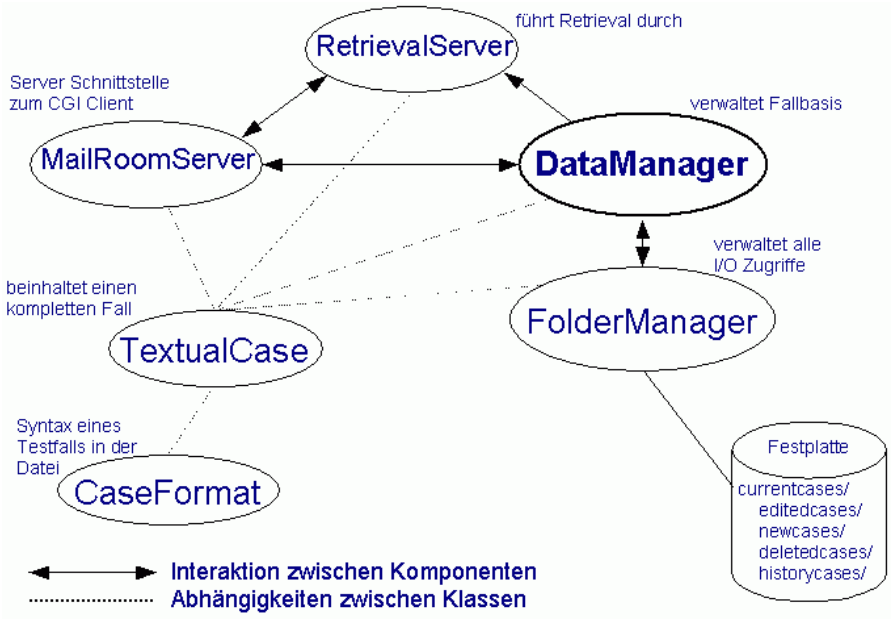

Bild 3: Überblick über die Umgebung des DataManagers im System

Der DataManager bearbeitet nach der Initialisierung der Reihe nach Anfragen, die vom MailRoomServer oder RetrievalServer kommen, z.B. Abspeichern eines neuen Falls, Liefern des kompletten Inhalts eines Falls, Angeben von Kurzdarstellungen bestimmter Fälle, Einlesen aller Fälle zum Aufbau des CRNs und weiteres. Der FolderManager als eine Komponente des DataManagers kapselt den kompletten I/O-Zugriff auf das Dateisystem. Alle genannten Komponenten hängen von der Klasse TextualCase ab, da Fälle zwischen den Komponenten in Form von TextualCase Objekten ausgetauscht werden.

2.1 Modell des Life Cycle eines Falls

Das Modell life cycle eines Falls wurde wie in [MinorHanft2000] beschrieben entworfen und bildet die Entwicklung eines unvollständigen Falls bis zum kompletten Fall ab. Der Begriff Life Cycle wird in dieser Arbeit trotz adäquat vorhandener Übersetzung Lebenszyklus nicht übersetzt, da er in der Literatur verwandter Fachgebiete wie im Software Engineering schon länger verwendet wird. Ein Beispiel für einen typischen Life Cycle eines Testfalls zeigt Bild 4. Dabei wird ein Fall mit seiner Erzeugung „geboren". Der Testfall kann zu diesem Zeitpunkt auch nur als vage Idee oder aus einer einzelnen Textzeile bestehen. Mit jeder Änderung und Ergänzung, z.B. dem Eintragen der Test-Instruktionen oder dem erwartetem Ergebnis reift der Fall. Nach jeder Durchführung des Tests wird das Resultat „erfolgreich" oder „nicht erfolgreich" vermerkt. Mit dem eventuellen Löschen eines Falls aus der Fallbasis stirbt ein Fall und dessen life cycle endet.

Modell des Life Cycle eines Testfalls

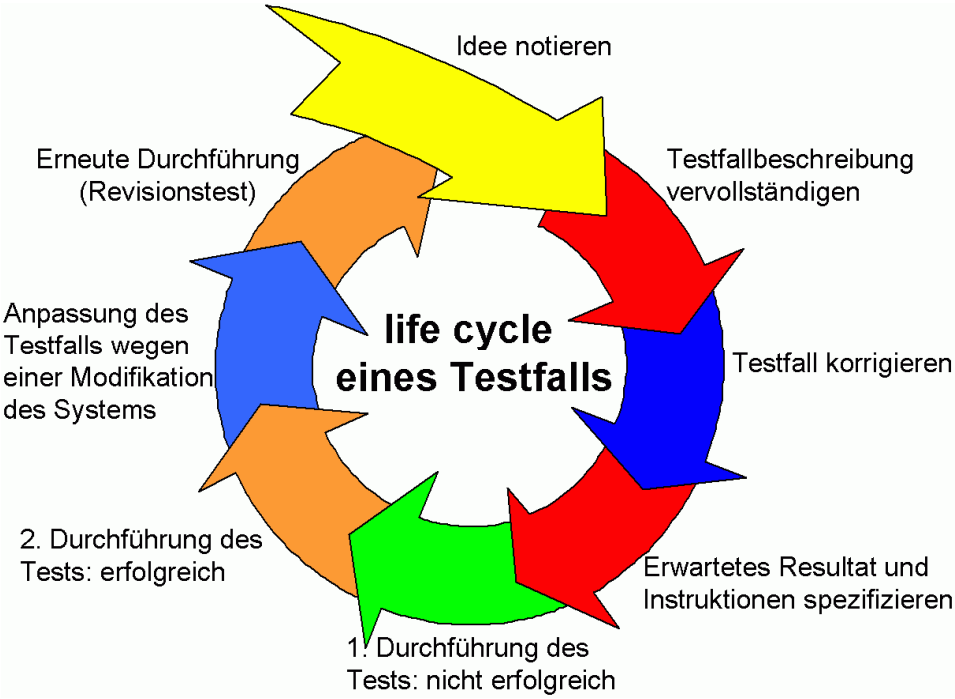

Bild 4: Ein typischer Life Cycle eines (Test-)Falls

Jeder Fall erhält zur Identifizierung eine lebenslängliche, eineindeutige Nummer – die persistente Fall-Id.
Um die unterschiedlichen Stadien des Reifegrades eines Testfalls zu unterscheiden, wurde die Revisionsnummer eingeführt, die sich mit jeder (abgespeicherten) Änderung am Fall erhöht. Eine höherer Revisionsnummer zeigt also das Fortschreiten im Lebenszyklus bzw. den höheren Reifegrad an.
Der Prozess des Erzeugens, Editierens und Löschens von Fällen wird fortan als *maintenance* bezeichnet.
Grundlage für das Retrieval ist die Vergleichbarkeit fragmentarischer (unvollständiger) mit vollständigen Fällen, welches über eine dreischichtige Ähnlichkeitsfunktion erreicht wird (siehe dazu wieder [MinorHanft2000]). Dazu wird vereinfacht gesprochen die Ähnlichkeitsfunktion nur auf Abschnitte des Falls angewendet, welche in der Query als auch in dem

untersuchten Fall gefüllt sind. Die Summe dieser partiellen Ähnlichkeiten wird als globale Ähnlichkeit jeweils mit der Anzahl der Abschnitte normiert, welche in Fall und Query gefüllt sind. Dadurch bleibt die Ähnlichkeit robust gegenüber der Vervollständigung von Fällen, d.h. vollständige Fälle sind nicht per se ähnlicher zur Query als unvollständige.

Bestandteile des Life Cycle

Das Modell des Life Cycle eines Falls besteht also aus:

- Revisionsnummer eines Falls,
- persistenter, eindeutiger Fall-Identifer und
- dreischichtiger Ähnlichkeitsfunktion zur Berechnung über unterschiedlich komplette Fälle.

2.1.1 Unterschiedliche Sichten zur Wahrung der Konsistenz im CRN trotz maintenance

Unterschiedliche Sichten zur Konsistenz-wahrung notwendig

Neben einem jederzeit möglichen Retrieval soll die unmittelbare Integration von Änderungen an Fällen ermöglicht werden.
Nach bisherigem Stand des bestehenden FBS Systems muss für die Berücksichtigung neuer und geänderter Fälle beim Retrieval das Case Retrieval Net (CRN) mit beträchtlichem Zeitaufwand neu aufgebaut werden. Während dieser Zeit wäre aber kein Retrieval möglich.
Die Alternative einer Integration neuer Fälle in ein bestehendes CRN ist auch denkbar, wurde aber wegen des Implementierungsaufwandes verworfen.
Bei einem Retrieval sollen gefundene Fälle mit den für das Retrievalergebnis relevanten IEs im Text präsentiert werden. Durch mögliche Änderungen an diesem Fall ist aber nicht sichergestellt, dass der neue Falltext noch die gleichen repräsentierenden IEs enthält. Daher sind mehrere Revisionen eines Falls zu behandeln und zwei verfügbar zu halten, eine für das Retrieval (auf dem das CRN basiert) und die aktuelle, falls sie durch einen oder mehrere Benutzer geändert wurde.

2.2 Anforderungen an den DataManager

Die zu verwaltenden Falldaten werden zur Laufzeit durch die Benutzer verändert. Aufgabe ist also die vollautomatische Integration neuer Fälle sowie die Erweiterung und Löschung bestehender Fälle. Trotzdem sollen die Falldaten möglichst ständig verfügbar sein.
Gleichzeitiges Bearbeiten von Fällen von mehreren Autoren muss durch eine Konfliktlösung vor gegenseitigem Überschreiben geschützt sein.

Weitere Anforderung: Kurzdarstellung eines Falls

Des weiteren sollen dem Benutzer (auf dessen Oberfläche) die Fälle übersichtlich mit Kurzbeschreibung in einer Liste präsentiert werden, welche immer auf dem neustem Stand ist und nach Fallnummern und Änderungszeitpunkt sortiert werden kann. Zuerst werden also dem Benutzer nur Kurzdarstellungen jedes Falls angezeigt und bei Bedarf jeweils ein kompletter Fall nachgeladen und angezeigt. Dies entspricht dem Laden von Beiträgen aus Newsgroups. Um den Oberflächen-Client und die Transportwege zu entlasten, soll diese Aufbereitung der Kurzdarstellung auch im DataManager erfolgen.

Zusammenfassung der Anforderungen

Zusammengefasst stehen folgende Anforderungen:

1. Anzeigen eines kompletten Falls der neusten Revision und der im Retrieval verwendeten Revision,
2. Änderung eines bestehenden Falls,
3. Behandlung von Konflikten beim gleichzeitigen Editieren des gleichen Falls,
4. automatische Integration eines neuen Falls in die Fallbasis (Abspeichern),
5. Löschen eines nicht mehr benötigten Falls,
6. Erstellen von Fall-Kurzdarstellungen, schnelle Bereitstellung,
7. Sortieren der Fall-Kurzdarstellungen nach verschiedenen Kriterien und
8. Lieferung von Fällen nach ID oder Fallindex.

2.3 Umsetzung des Case Life Cycle

Umsetzung des Life Cycle durch flache Versionsverwaltung

Zur Verwaltung der unterschiedlichen Revisionen eines Falls und der Erkennung von Konflikten bei konkurrierendem (Schreib-)Zugriff auf den gleichen Fall organisiert der DataManager eine flache Versionsverwaltung für die Falldaten. Flach wird diese bezeichnet, da sie keine Hierarchie der Revisionen erlaubt, wie das z.B. bei CVS mit Branches möglich ist.
Da das CRN neu aufgebaut werden kann, während Benutzer mit dem System arbeiten, kann für den Fall-Identifer nicht der sonst aus dem CRN benutzte Fall-Index verwendet werden. IDs gelöschter Fälle können nicht wieder verwendet werden.
Die Versionsverwaltung ohne Sperren ist beim Einsatzszenario übers Internet/Intranet mit diskontinuierlichen Verbindungen vorteilhafter als ein Locking Mechanismus, da er Deadlocks oder die Notwendig des automatischen Lösen der Sperren vermeidet. Da das Internet/Intranet keine kontinuierlichen Verbindungen voraussetzt kann das System auch nicht von solchen ausgehen. Würden beim Ändern von Fällen diese für andere Benutzer gesperrt werden, müsste ein automatisches Lösen der Sperren (z.B. nach einer Zeitspanne) implementiert werden. Netzwerkverbindungen können abreißen, Computer abstürzen oder Mitarbeiter die Bearbeitung (und Sperrung) eines Falls über die Mittagspause oder den Feierabend „vergessen“ und somit die Aufhebung der Sperre verhindern.

2.3.1 «case authoring» und «retrieval» Sicht

Um für das Retrieval stets eine konsistente Fallbasis (und CRN) zu haben, als auch nach dem Editieren eines Falls die letzte Änderung sichtbar zu machen, verwaltet der DataManager zwei Sichten auf jeden Fall: die «case authoring» Sicht und «retrieval» Sicht. Bei ersterer werden immer die neusten Änderungen sichtbar, während bei der zweiten sich der Fall nach Aufbau des CRN nicht mehr ändert [MinorHanft1999], Kap 3.

Dies verdeutlicht Bild 5: Nach dem Initialisieren des DataManagers (und danach erfolgtem Aufbau des CRN) zeigen beide Sichtweisen auf die gleiche Revision eines Falls.
Wird ein Fall geändert, unterscheiden sich beide Sichtweisen, die «case authoring» view zeigt auf den Fall mit einer aktuelleren, höheren Revision. Mit dem nächsten Update (siehe *3.8.2 Update Seite 26*) der Fallbasis (und Neuaufbau des CRN) werden beide Sichten wieder vereint.

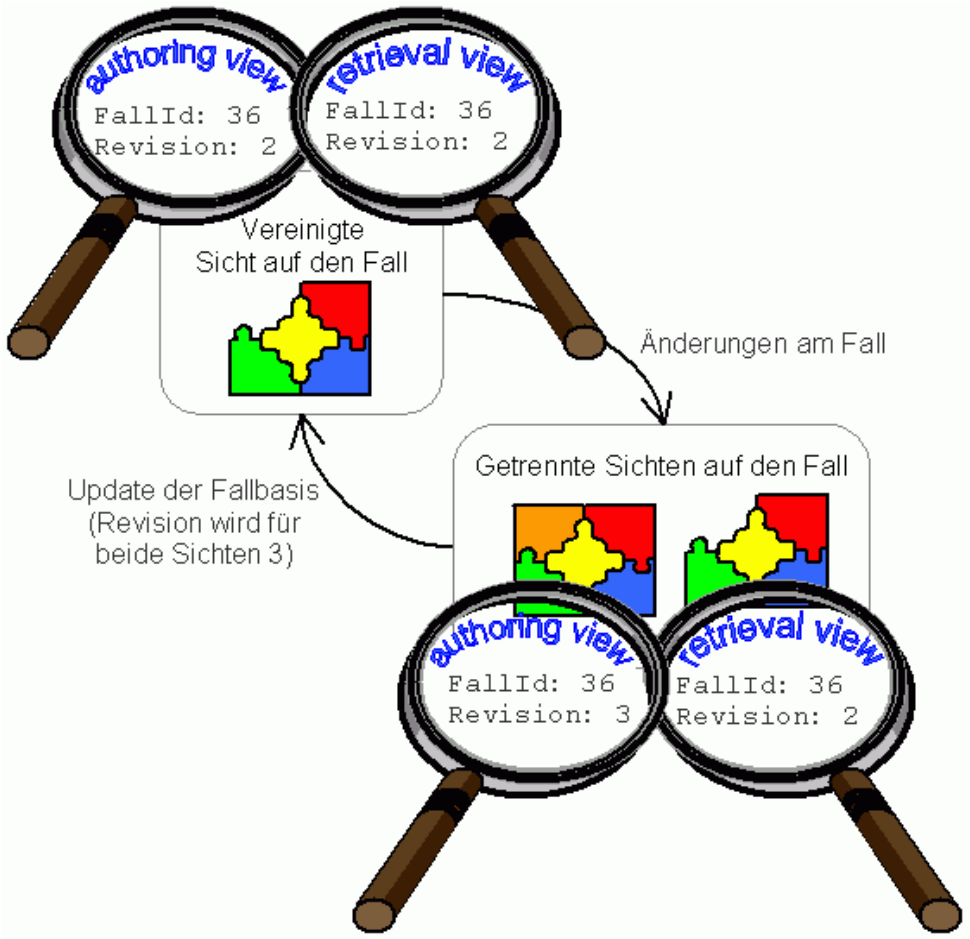

Bild 5: Unterschiedliche Sichtweisen vor und nach der Fallmodifikation

Unveränderbar ist nur die Fall-Id. Bis auf die vom DataManager verwalteten Infoattribute *erstelltAm*, *erstelltVon*, *geaendertAm* sowie die *Revision* können alle anderen Bestandteile des Falls in jeder Revision geändert und ergänzt werden.

2.4 Syntax eines Falls

Das Fallformat,. in welcher Form der Fall in der Datei abgespeichert wird, ist abgesehen von der fehlenden XML-Deklaration ein XML-Dokument. Es ist unterteilt in mehrere Abschnitte. Jeder Fall wird in einer separaten Datei, der Falldatei gespeichert. Die DTD sieht wie folgt aus:

Fehler! Kein gültiger Dateiname.
Dies ist das vollständige Beispiel eines Falls, wie er in der Datei abgespeichert wird:

Beispiel eines Falls

```
<CASE>
<CASE_NUMBER>
1
</CASE_NUMBER>
<TITLE>
</TITLE>
<RETRIEVAL_ATTRIBUTES>
Kategorie = Leitsystem
Kategorie = Simulation
Themenbereich = Oberflaeche
Themenbereich = Optimierung
Themenbereich = Regelsystem
Status = Idee fuer einen Test
</RETRIEVAL_ATTRIBUTES>
<INFO_ATTRIBUTES>
erstelltAm = 12. 03. 1999
erstelltVon = Minor
geaendertAm = 11. 04. 1999
geaendertVon = Hanft
Revision = 2
Sortierzeit = 910015418
</INFO_ATTRIBUTES>
<DESCRIPTION>
Der Test soll zeigen, wie man auf einer oder
mehreren Zeilen ein Fallformat entwirft.
</DESCRIPTION>
<INSTRUCTION>
1. Oberflaeche starten
2. Bedenken der Konsequenzen
3. Aufschreiben
4. von anderen ueberpruefen lassen
</INSTRUCTION>
<EXPECTED_RESULT>
Erwartet wird eine schnelles, auch eine oder
mehrere Zeilen umfassendes, flexibel einzusetzendes
Konzept.
</EXPECTED_RESULT>
</CASE>
```

Der XML-Syntax entsprechend wird jeder Abschnitt durch ein dem gleichlautendem SectionTag vorangestelltem Schrägstrich abgeschlossen. Fehlende Abschlüsse führen beim Scannen zu Fehlermeldungen. Die Reihenfolge der Abschnitte ist festgeschrieben. Eventuelle Leerzeilen zwischen den Abschnitten bleiben unberücksichtigt.
Im Folgenden werden die einzelnen Abschnitte, deren Format und Bedeutung erläutert:
`<CASE_NUMBER>` ist der Fall-Identifer als Long-Wert, beginnend bei 1.

TextualCase hat zwei Abschnitte mit Atribut-Werte-Paaren, einen *Infoattributes* genannten für informative und organisatorische Zwecke sowie der *Retrievalattributes* genannte Abschnitt für im Retrieval relevante Daten wie z.B. die zu testende Kategorie.
Wie in *1.4 Einleitung auf Seite 5* beschrieben werden alle Attribute als Attributname = Attributwert dargestellt, führende und nachfolgende Leerzeichen bei den Attribut-Werte-Paaren bleiben unberücksichtigt.

`<RETRIEVAL_ATTRIBUTES>` Retrieval-Attribute

Status ist entweder "Idee für einen Test" oder "Testfall". Das Attribut *Status* ist einwertig.
Kategorie und *Themenbereich* können mehrere Werte zugeordnet sein.

`<INFO_ATTRIBUTES>` Info-Attribute

Die Attribute *Sortierzeit*, *Revision*, *erstelltAm*, *geaendertAm*, *erstelltVon*, *geaendertVon* sind einwertig.

Sortierzeit =	eine Zahl vom Typ time_t (=long), welche die Systemzeit der letzten Änderung zwecks besserer Vergleichbarkeit speichert, entspricht TextualCase.dateChanged
Revision =	Revision des editierten Falls, beginnend bei 1 (int)
erstelltAm =	Datum der Erstellung als String (wird vom DataManager aus der ersten Sortierzeit berechnet) Format: TT. MM. JJJJ
geaendertAm =	Datum der letzten Änderung als String (wird vom DataManager aus Sortierzeit berechnet) Format: TT. MM. JJJJ
erstelltVon =	Name des Autors als string
geaendertVon =	Name des Editors als string

`<DESCRIPTION>`, `<RESULT>` und `<INSTRUCTION>` sind (mehrzeilige) Texte vom Typ string. drei Text Abschnitte

`<DESCRIPTION>`	ist die Beschreibung des Testfalls, wird im Member TextualCase.theDescription abgelegt
`<RESULT>`	ist das erwartete Ergebnis des Tests, entspricht TextualCase.theExpectedResult
`<INSTRUCTION>`	ist die Handlungsanweisung zur Durchführung des Tests, entspricht TextualCase.theInstruction

Für die Verwendung des Fallformats im Programm enthält die Datei caseFormat.h diesbezügliche Festlegungen der Abschnittsnamen und AttributWert-Namen, so dass eine Formatänderung keine Änderungen im Quelltext bei der Verwendung, sondern nur in dieser Header-Datei nach sich zieht.
Das hier gezeigte Fallformat wird unmittelbar in die Klasse *TextualCase* umgesetzt.

2.4.1 Der gültige Dateiname

Für eine Fall-Id ist der Dateiname exakt nach folgender Signatur vorgeschrieben. Auf das Präfix folgt mit gegebener Anzahl führender Nullen das Dateisuffix.
Die Werte stehen in der Datei `tm.ini`

```
filePraefix = TM01Case_
fileSuffix = .txt
countDigitsFileName = 10
```

so dass sich z.B. für die Id 45 `TM01Case_0000000045.txt` ergibt.

2.5 Kurzdarstellungen eines Falls

Aus dem oben gezeigten Fallformat werden zur Darstellung der Fälle in Listen davon nur die Fall-Id, die zur Sortierung und Präsentation erforderlichen InfoAttribute sowie die 1. Zeile der Fallbeschreibung benötigt. Die Sortierzeit (Zeit der letzten Änderung) ist zur schnellen Sortierung direkt als `member time_t theSortTime` schon in *Basic-TextualCase* vorhanden. *TextualCaseShort* stellt die geforderte kürzere Darstellung des Falls dar.

3 Implementierung des DataManagers

Für die komplette Verwaltung der Fallbasis inklusive der Modellierung des life cycle eines jeden Falls ist die Komponente DataManager im TestManager zuständig, welche im folgenden Kapitel beschrieben wird. Sie verfügt dafür über eine Schnittstelle, die über `getCase()`, `saveCase()` und `saveNewCase()` das Lesen und Abspeichern von Fällen erlaubt. Einzelne oder eine Liste von Fall-Kurzdarstellungen werden über `getCasePresentation()` und `getCasePresentations()` zurückliefert. Eine Übersicht der beteiligten Klassen zeigt Bild 6.

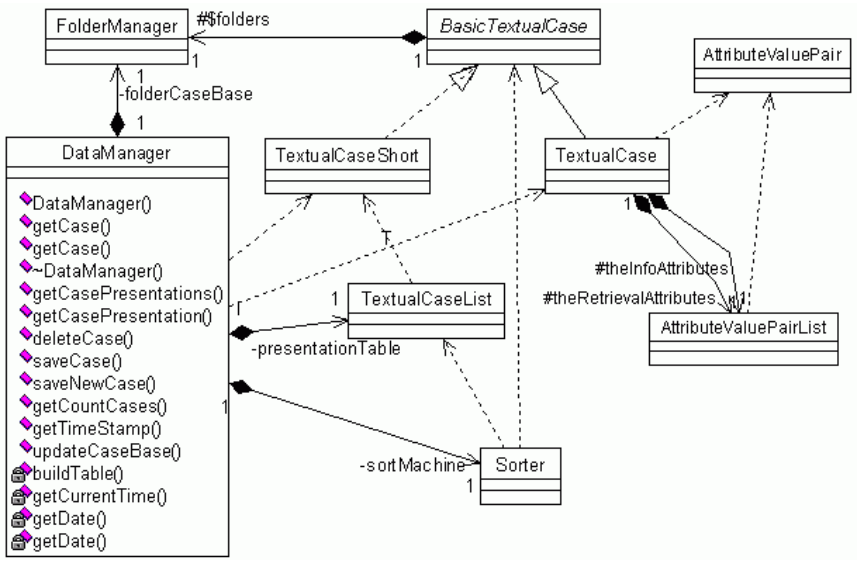

Bild 6: UML Klassendiagramm: *DataManager und benutzte Klassen*

Zuerst werden die verwendeten Basisklassen AttributeValuePair und die Listenklasse AttributeValuePairList beschrieben. BasicTextualCase ist die Basis für TextualCase und TextualCaseShort sowie TextualQuery, die nicht im Diagramm aufgeführt ist. Sie werden danach ausführlich erläutert.
Wie Bild 6 zeigt, hat der DataManager folgende drei Komponenten:

- FolderManager für den I/O Zugriff,
- PresentationTable fürs Vorhalten der Kurzdarstellungen aller Fälle und
- den Sorter für die Sortierung der Falldaten.

Diese Komponenten werden anschließend dargestellt.
Zusammenfassend wird danach die Schnittstelle der DataManagers inklusive der Bedeutung der Parameter jeder Methode vorgestellt. Außerdem wird dabei besonders auf die Konfliktlösung bei konkurrierendem Schreiben eingegangen.

3.1 Die Klasse AttributeValuePair

Die Klasse *AttributeValuePair* modelliert ein Attribut-Werte-Paar (AVP), wobei Attributname als auch Attributwert vom Typ her strings sind.

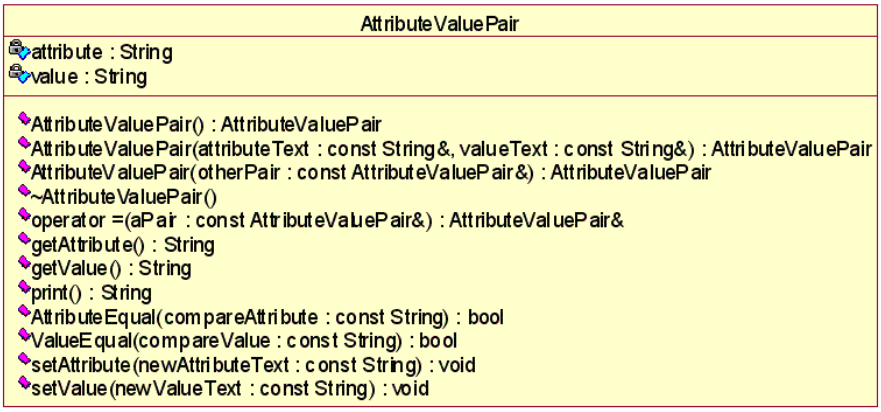

Bild 7: UML Klassendiagramm[1] *AttributeValuePair*

Wie Bild 7 zeigt, enthält die Klasse die Methoden `getAttribute()`, `getValue()`, `setAttribute()` sowie `setValue()` zum Lesen und Ändern der AttributNamen bzw. -Werte. Folgende Methode überprüfen auf Gleichheit: `operator =()` das ganze AVP, `AttributeEqual()` den Attributnamen und `ValueEqual()` den Attributwert.

3.2 Die Aufzählung AttributeValuePairList

Verwaltung mehrerer AVPs

Zur Verwaltung von AVPs innerhalb eines Falls wurde die Klasse *AttributeValuePairList* implementiert. Dies ist eine Ableitung von einem DYN_ARRAY Template (siehe 4.2 Dynamisches Array auf Seite 43). Die Suche nach einem Wert zu einem bestimmten Attribut erfolgt mittels `getValue()`, die Überprüfung ob ein angegebenes Attribut oder Paar enthalten ist, durch `findPos()`. Mit `setPair()` wird ein neues AVP angefügt. Ist der Parameter `changeFlag` auf true (=default) gesetzt, wird kein neues AVP angehängt, wenn der Attributname schon vorkommt, sondern der vorhandene AttributWert überschrieben. Dies entspricht der Implementation eines einwertigen AVP. Mit `changeFlag = false` wird immer ein neues AVPs angehängt, was einem mehrwertigen AVP entspricht. Ein Klassendiagramm ist in Bild 23 auf Seite 44 aufgeführt.

1.1.1.1

[1] Das verwendete UML-Werkzeug Rational Rose 98i stellt die Sichtbarkeit in einer an MS Visual C++ angelehnten Notation dar:

Sichtbarkeit	UML 1.4	Rose
public	+	
protected	#	
private	-	
friend	$	

3.3 BasicTextualCase

abstrakte Basisklasse eines Falls

BasicTextualCase stellt die abstrakte Basisklasse dar, welche die Gemeinsamkeiten der unten erläuterten Klassen *TextualCase* und *TextualCaseShort* sowie das Handling für den I/O-Zugriff beinhaltet. Bild 8 zeigt einen Überblick über die TextualCase Familie: Von BasicTextualCase sind TextualCase und TextualCaseShort abgeleitet. Während ersterer einen kompletten Fall aufnimmt, beinhaltet letzterer nur die Kurzdarstellung eines Falls. TextualCaseList ist eine mit TextualCaseShort als Parameter instanzierte Klasse des Templates `LONG_DYN_ARRAY` (gekennzeichnet mit einem T), welche mit der STL-Klasse `<vector>` vergleichbar ist. Ein TextualQuery beinhaltet eine Abfrage eines Benutzers.

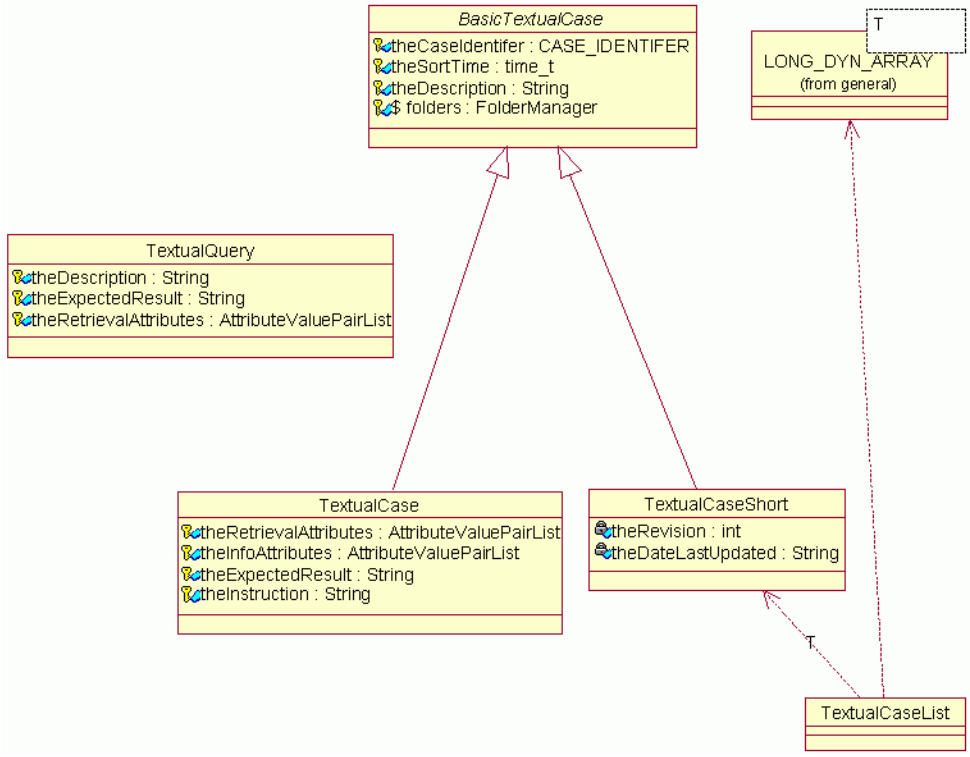

Bild 8: UML-Klassendiagramm: Übersicht über die TextualCase-Familie

Die Sortierzeit (Zeit der letzten Änderung) ist sowohl als AVP in den InfoAttributen als auch zur schnelleren Sortierung noch einmal extra als Member `time_t theSortTime` vorhanden.
Ein TextualCase ist wie ein Büro-Rollcontainer – er wird an einer Stelle gefüllt, zur einer anderen Stelle geschoben und dort ausgelesen oder weiter gefüllt.
Die lebenslängliche Fall-Id ist vom Typ
`typedef long` **`CASE_IDENTIFER`** und beginnt bei 1.
Im Bild 9 sind alle Attribute und Methoden der Klasse *BasicTextualCase* aufgeführt. Leicht zu erkennen sind die zu den Attributen der Klasse korrespondierenden Set- und Get-Methoden. Die zum Scanner gehörenden Methoden werden im nächsten Abschnitt erläutert.

Bild 9: UML Klassendiagramm abstrakte Klasse *BasicTextualCase*

3.3.1 Der Scanner

Der Scanner als eingebettete Klasse erledigt das Einlesen des TextualCase Objekts aus einer Falldatei. Scanner und Parser sind vollständig getrennt, wobei *BasicTextualCase* nur ersteren Dienst übernimmt. Der Parser befindet sich im tmRetrievalManager, welcher beim Aufbau des CRN die Fälle vom DataManager anfordert.

Stream-Handling in Hilfsklasse FolderManager

Die Methode zum Scannen heißt `read(const bool newerFlag)` und läuft in drei Schritten ab:

1. Öffnen der Datei
2. Einlesen der Zeichen /Scannen aus *inputFile*
3. Schließen der Datei

Das Öffnen und Schließen der Falldatei zum Schreiben/Lesen übernimmt der FolderManager, in *BasicTextualCase* als Klassenvariable `gateway`. Da das Lesen und Schreiben unter C++ über Streams erfolgt, stellt der FodlerManager mit `readOpenCase()` einen Zugriff auf die richtige Datei über sein public Member `ifstream inputFile` her, aus der dann gelesen wird. Nach dem Scannen wird abschließend der Stream vom FolderManager wieder mit `readClose()` geschlossen. Detaillierte Ausführungen dazu finden sich unter *3.8.3 Schreiben und Lesen von Fällen auf Seite 26*.

Der Parameter `const bool newerFlag` wird an den FolderManager durchgereicht und entscheidet, ob ein Fall in der aktuellsten Revision oder der Retrieval Revision gefordert ist. Diese Semantik gilt für jeden

Parameter `newerFlag` innerhalb des DataManager Moduls. Siehe dazu auch *2.3.1 «case authoring» und «retrieval» Sicht auf Seite 12*.
Objekte der Familie *BasicTextualCase* kümmern sich nicht darum, welche Datei geöffnet wird, ob eine neue Datei angelegt wird oder eine alte überschrieben werden kann. Dies wird über die Parameter Fall-Id und das newerFlag erst im FolderManager entschieden.

Scanner basiert auf expectTag() und doUntilTag() und nextLine()

Der Einlesevorgang wird strukturiert durch drei Methoden `nextLine()`, `expectTag()` und `doUntilTag()`.

```
void nextLine( String & aLine )
    throw( EOFFileIOException, ReadFileIOException);
```

Einlesen erfolgt zeilenweise

liest eine Zeile aus `ifstream` in einen String `aLine` ein, wobei die Funktion statt direktem `ifstream.getline()` dafür sorgt, dass Leerzeilen überlesen werden. Bei Lesefehlern wird eine *EOFFileIOException* oder *ReadFileIOException* geworfen. Eine Übersicht zu den Exception Klassen des Projekts steht in Bild 25 auf Seite 46.

```
void expectTag( String & aLine, const String & aTag )
     throw( CaseFileSyntaxException, FileIOException );
```

expectTag() prüft Tag ab

überprüft ob die nächste eingelesene Zeile genau das geforderte Tag `aTag` enthält. Im positiven Fall wird die Zeile mit dem Tag kommentarlos passiert, ansonsten eine *CaseFileSyntaxException* geworfen.

Darauf aufbauend ruft

```
void doUntilTag( StringFunc todo, const String & aTag);
```

todo() mit jeder Zeile bis Tag

für jede Zeile die Funktion `todo()` mit der aktuellen Zeile als Parameter auf, solange nicht der String `aTag` in der neu eingelesenen Zeile entdeckt wird. `doUntilTag()` reicht die Exceptions von `nextLine()` weiter.

Alle an `todo` übergebenen Funktionen müssen die Signatur *StringFunc erfüllen:

```
typedef void ( BasicTextualCase::*StringFunc ) ( const String &);
```

Einfachster zur `*StringFunc` Signatur passender Vertreter ist

```
void doNothing( const String & aLine ) { }
```

und verarbeitet nichts von der aktuellen Zeile. Somit eignet sie sich zum Überspringen von Sections.

Für jede Section eine read... Funktion

Jede Section wird dann mithilfe von einem doUntil Aufruf eingelesen, wobei die übergebenen StringFunc Funktionen jede einzelne Zeile in der Section verarbeiten.

```
virtual void readDescription (const String & aLine );
virtual void readRetrievalAttributes (const String & aLine )=0;
virtual void readInfoAttributes (const String & aLine ) = 0;
virtual void readInstruction (const String & moreInstruction )=0;
virtual void readExpectedResult(const String & moreResult )=0;
```

Die erste ist virtuell zur unterschiedlichen Behandlung der Fallbeschreibung (Description Section) in den Unterklassen *TextualCase* und *TextualCaseShort*. Die letzten vier sind auch pure virtuell (in UML Notation: {abstrakt}), weil eine sinnvolle Redefinition zwecks adäquater Behandlung erst in der Unterklasse *TextualCase* erfolgen kann. In *TextualCaseShort* bleiben diese Funktionen außer *readInfoAttributes()* leer, da diese Abschnitte dort nicht berücksichtigt werden. Im folgenden Bild 10 sei noch mal eine Übersicht über die zum Scannen verwendeten Methoden und betroffenen Attribute gegeben:

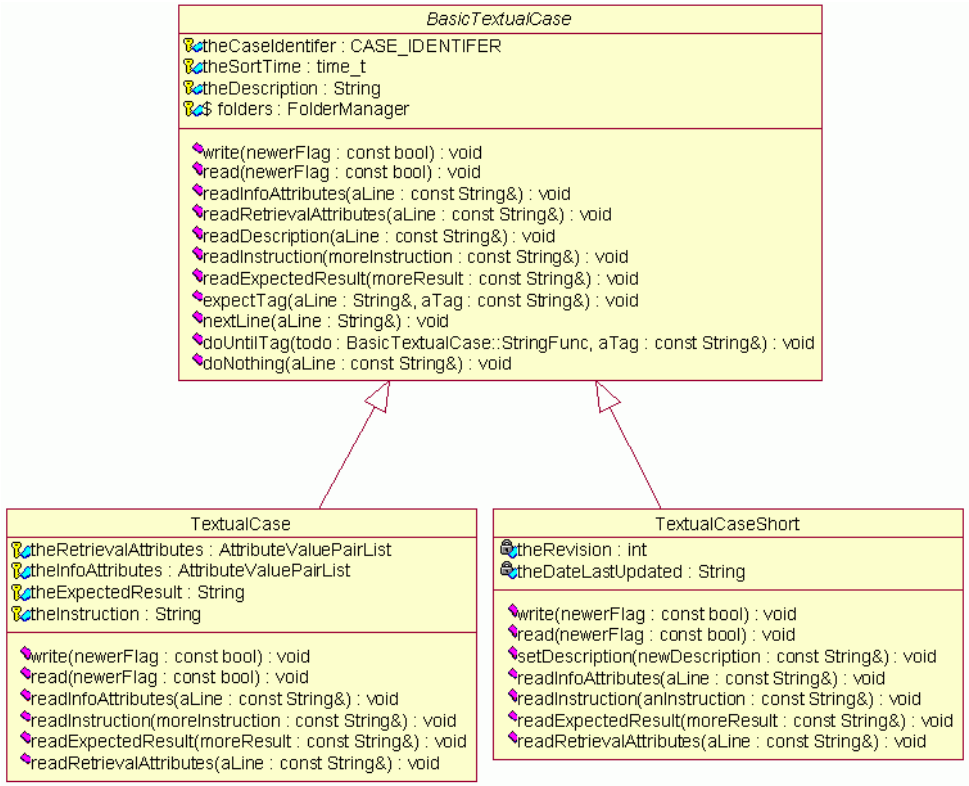

Bild 10: UML-Klassendiagramm:
Virtuelle und redefinierte Methoden zum Scannen

3.3.2 Fall abspeichern

Mithilfe der Funktion

```
virtual void write( const bool newerFlag )
    throw( FileIOException ) = 0
```

der Klasse BasicTextualCase wird ein Fall analog zum Scannen in drei Schritten auf die Festplatte geschrieben. Der FolderManager stellt dabei über sein public Member `ofstream outputFile` den Zugriff auf die richtige Datei her:

1. Schreibendes Öffnen der Datei mit *FolderManager::writeOpenCase()*
2. Ausgeben des Fallformats mit printCaseFormat() nach FolderManager::outputFile
3. Schließen der Datei mit FolderManager::writeClose()

Die Funktion `printCaseFormat()` gibt dabei den Fall so aus wie er vom Scanner gelesen wird und wie in *2.4 Syntax eines Falls auf Seite 13* vorgestellt. Für die Klasse TextualCaseShort ist write() auch definiert, lässt aber einige Abschnitte mangels Daten frei.

Kontrollierter Zugriff über DataManager

Die Methoden `read()` und `write()` sind protected. Da nur DataManager friend der *BasicTextualCase* ist, geschieht das Lesen und Schreiben nur durch Operationen und unter Kontrolle des *DataManager*.

3.4 TextualCase

Das oben in 2.4 auf Seite 13 gezeigte Fallformat wird unmittelbar in die Klasse *TextualCase* umgesetzt. Bild 11 zeigt alle bei der Klassendefinition neuen und redefinierten Methoden aus der Basisklasse.

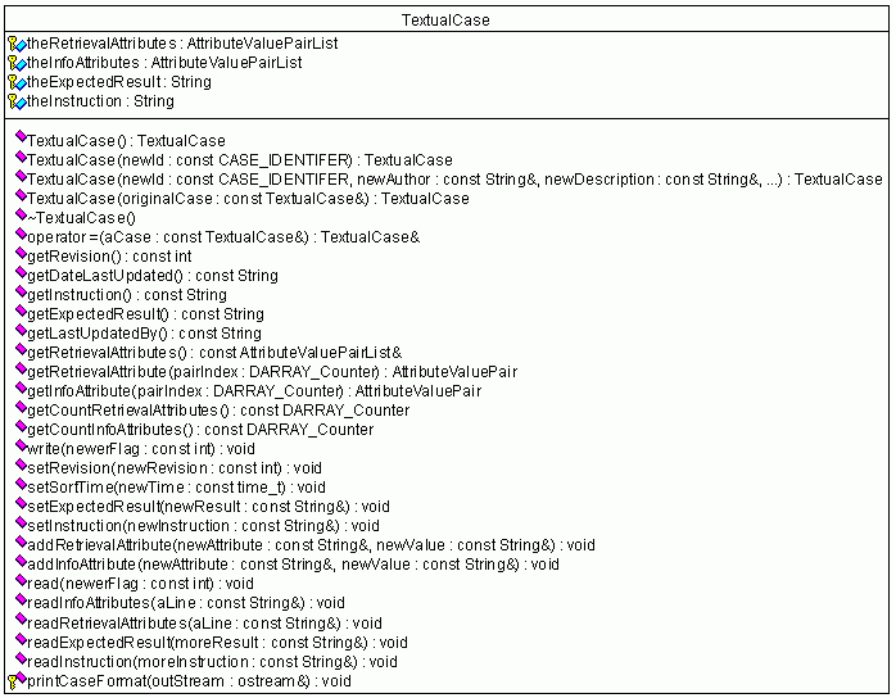

Bild 11: UML-Klassendiagramm: TextualCase

Für die Instruktionen und das erwartete Testresultat sind die Member `theInstruction` und `theExpectedResult` mit ihren korrespondierenden set- und get Methoden vorhanden.
Für die Attribut-Werte-Paare der Retrieval-Attribute und Info-Atrribute existieren vom Typ AttributeValuePairList die Member `theInfo-Attributes` und `theRetrievalAttributes`. Alle Retrieval-Attribute werden mit `getRetrievalAttributes()` als AttributeValue-PairList zurückgeliefert. Daneben existieren noch für ein Attribut die korrenspondierenden get- und set-Methoden.
Das AVP Sortierzeit (Zeit der letzten Änderung) ist sowohl als Info-Attribut als auch zur schnelleren Sortierung noch einmal extra als member `time_t theSortTime` vorhanden. Ebenso greifen `setSortTime()`, `setRevision()`, `getRevision()`, `getLastUpdatedBy()` sowie `getDateLastUpdated()` auf die Liste der Info-Atrribute zu.

3.5 TextualCaseShort

Diese Klasse dient zur Kurzdarstellung eines Falls in Listen für die Oberfläche entsprechend der Anforderung Nr. 5 an den DataManager.
Zur einfachen Erzeugung aus einem TextualCase existiert ein Copy-Konstruktor mit der Basisklasse als Argument:

```
TextualCaseShort& operator= (const BasicTextualCase & aCase)
```

Bild 12 zeigt die Klassendefinition mit allen neuen und redefinierten Attributen und Methoden.

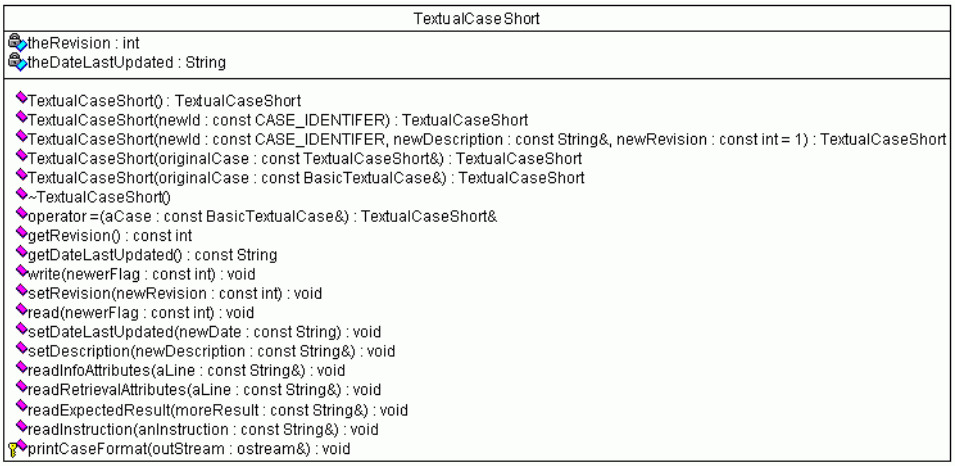

Bild 12: UML-Klassendiagramm: TextualCaseShort

Die bei der Schwesterklasse *TextualCase* in der Liste der InfoAtrribute gespeicherten AVPs *Revision*, *Sortierzeit* und *geaendertAm* werden hier in den atomaren Membern `theRevision`, und `theDateLastUpdated` festgehalten. Diese werden dann auch von der redefinierten `readInfoAttributes()` beim Scannen gefüllt.
Ebenso wird `setDescription()` redefiniert, da von dem zugewiesenem Text nur die ersten 60 Zeichen als Kurzbeschreibung verwendet werden.

3.6 TextualCaseList

TextualCaseList verwaltet ein dynamisches Array von TextualCaseShort Objekten und ist eine Template Instanz von `LONG_DYN_ARRAY` (*siehe 4.2 Dynamisches Array auf Seite 43f*), vergleichbar mit der Klasse <vector> aus der STL:

```
typedef LONG_DYN_ARRAY< TextualCaseShort > TextualCaseList;
```

Sie dient vornehmlich für die `PresentationTable` und das Zurückgeben von mehreren Fall-Kurzdarstellungen bei `getCasePresentations()`.

3.7 TextualQuery

Ein TextualQuery Objekt enthält die eingegebenen Werte einer Anfrage für das Retrieval. Es hat dieselbe Struktur wie *TextualCase*, aber ohne Fall-Id, I/O-Zugriff, Scanner und die Info-Attribute. Daher ist die Klasse auch nicht von BasicTextualCase abgeleitet. Bild 13 zeigt ihr Klassendiagramm.

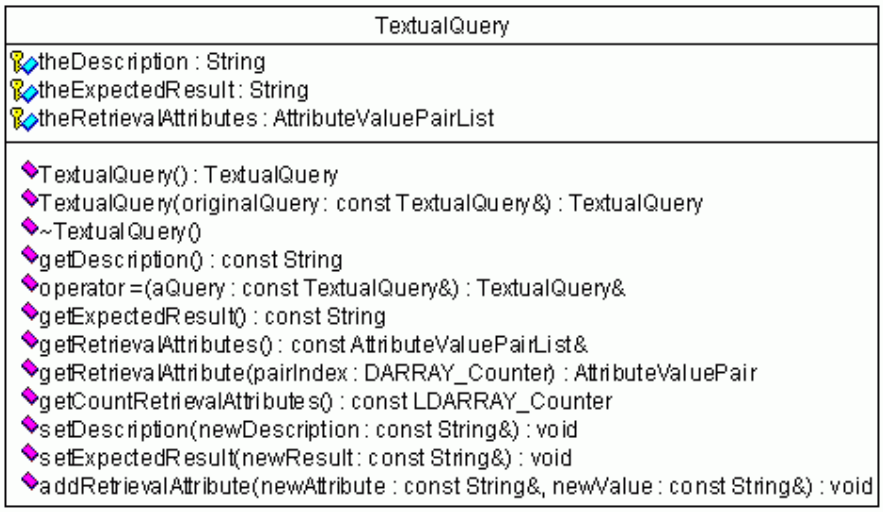

Bild 13: UML-Klassendiagramm: TextualQuery

3.8 FolderManager

Der FolderManager kapselt vollständig den Zugriff auf das Dateisystem. Er verwaltet die Dateien mit den Falltexten, welche zusammen die (persistente) Fallbasis auf Disk bilden. Zugriffe auf die Dateien der Fallbasis von Ableitungen von *BasicTextualCase* und des *DataManager* laufen nur über die Schnittstelle des *FolderManager*.

3.8.1 Struktur der Fallbasis

Die Fallbasis besteht physisch aus Dateien, eine für jeden Fall. Zur Verwaltung der Änderungen befindet sich die Fallbasis in einem Hauptverzeichnis *curentcases/*. Die Dateien im Hauptverzeichnis werden zur Laufzeit nicht verändert, damit eine fürs CRN konsistente Fallbasis vorliegt («retrieval» Sicht), d.h. zu im Retrieval aktivierten Fällen werden auch die geparsten Falltexte gefunden.
Die zur Laufzeit des CRN getätigten Änderungen werden in extra Unterverzeichnissen abgespeichert.

Verzeichnis-Struktur:

currentcases/ (Hauptverzeichnis)
enthält alle Fälle des aktuellen CRN (nach vollständigem Update), jeden Fall in einer separaten Datei

Darin befinden sind die Unterverzeichnisse:

deletedcases/
: enthält die kopierten Dateien der Fälle aus currentcases/, welche gelöscht wurden (bis zum vollständigen Update verbleiben sie wegen der Konsistenz des CRN für das Retrieval noch in *currentcases/*)

newcases/
: enthält neu angelegte Fälle, welche beim Update nach *currentcases/* verschoben werden

editedcases/
: enthält die neusten Daten der editierten Fälle, während die alten Revisionen des Falls noch bis zum vollständigen Update in *currentcases/* stehen

historycases/
: enthält die Dateien der Fälle aus *currentcases/*, welche beim Bereinigen der Unterverzeichnisse (vollständiges Update) mit neueren Daten aus *edited/* überschrieben oder gelöscht wurden. Damit wird eine History-Funktion (Backup) für fälschlich überschriebene, gelöschte Fälle realisert.

3.8.2 Update der Fallbasis

Update vereinigt «case authoring» und «retrieval» Sichten

Durch ein Update der Fallbasis werden die beiden Sichtweisen «case authoring» und «retrieval» wieder vereinigt. Dabei werden alle Dateien durch den FolderManager mittels *update()* aus den Unterverzeichnissen gelöscht bzw. verschoben. Bei einem anschließendem Neuaufbau des CRN werden dann die neusten Fälle und Veränderungen mitberücksichtigt.
Das Update wird automatisch beim Herunterfahren des *Foldermanagers* mit *update()* oder über die Schnittstelle des DataManagers mit *updateCaseBase()* ausdrücklich zur Laufzeit des Systems angestossen.
Dazu prüft der DataManager-Konstruktor mit *FolderManager::-isCaseFileInDirectory()* ob die Fallbasis ein richtiges Update erfahren hat und veranlasst im negativen Fall ein Update.

Der Vorteil der Verwaltung in Verzeichnissen statt in Listen im Speicher liegt darin, dass bei einem Absturz bzw. nicht ordnungsgemäßen Herunterfahren des TestManager Systems (d.h. ohne Update) das Update nachgeholt wird und die ja schon abgespeicherten neuen und geänderten Fälle nicht verloren gehen sondern in die Fallbasis integriert werden.

3.8.3 Schreiben und Lesen von Fällen

Der eigentliche Schreib-/ Lesezugriff zum Speichern und Lesen der Fälle geschieht folgendermaßen:

1. FolderManager öffnet Stream
2. TextualCase liest aus/schreibt in Stream
3. FolderManager öffnet Stream

FolderManager stellt über seine public member `ifstream inputFile` und `ofstream outputFile` den Zugriff auf den jeweiligen Stream bereit, welche mit folgenden Methoden verändert wird:

\ Zweck Operation	Datei lesen (Scannen)	Datei schreiben (Fall speichern)
Öffnen	*readOpenCase(...)*	*writeOpenCase(...)*
Schließen	*readClose()*	*writeClose()*

Der FolderManager entscheidet bei `readOpenCase()` (bzw. `writeOpenCase()`) nur anhand der Parameter *Fall-ID* und des *NewerFlag*, welche Datei geöffnet wird, ob eine neue Datei angelegt wird oder eine alte überschrieben werden kann. Der Dateiname wird mit `buildName()` gebildet. Bild 14 zeigt das UML-Aktivitätsdiagramm der Funktion `readOpenCase()`.

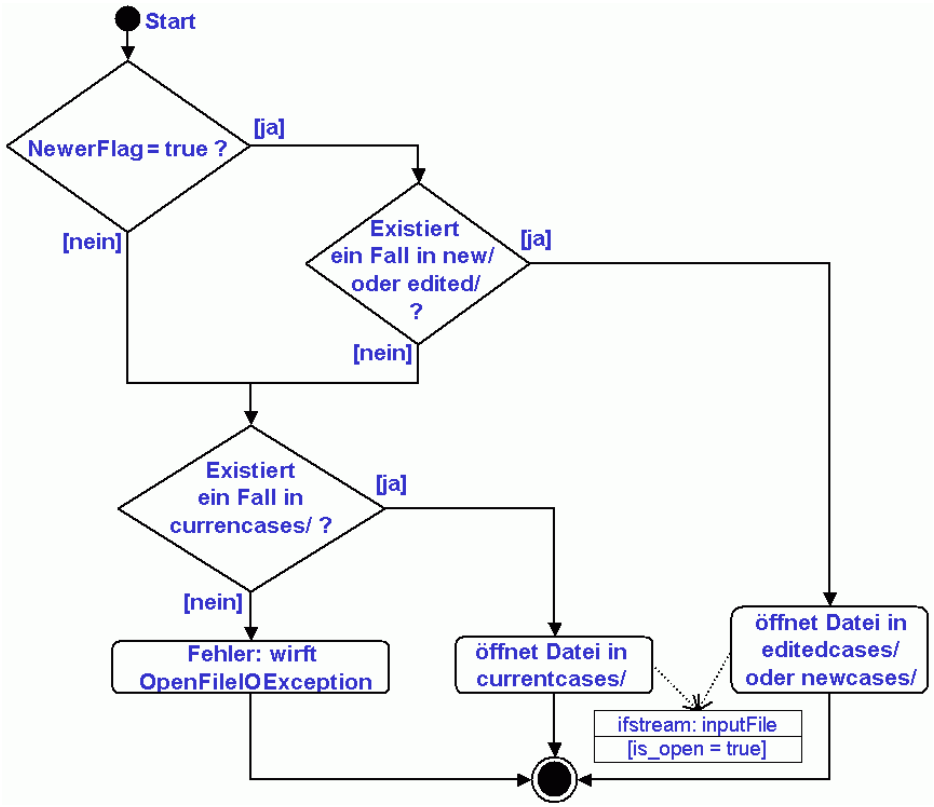

Bild 14: UML-Aktivitätsdiagramm:
Ablauf der Funktion *readOpenCase()*

Ist der Parameter NewerFlag true, wird im Hauptverzeichnis nachgeschaut ob dort eine Datei mit der Fall-ID existiert. Im anderen Falle wird erst in editcases/ nachgeschaut und danach in newCases/. Wurde dabei eine Datei gefunden wird sie geöffnet, ansonsten eine Ausnahme OpenFileIO-Exception geworfen.

Die Funktion *writeOpenCase()* ist im Abschnitt *3.11.3.1 Konfliktlösung bei konkurrierendem Speichern auf Seite 39* näher erläutert und wirft im Fehlerfall ebenfalls eine OpenFileIOException.

Die zeitliche Abfolge und den Überblick über die beteiligten Objekte beim Liefern eines Falls zeigt folgendes Sequenzdiagramm im Bild 15, ausgehend von der „Poststelle“ TMMailRoom, welche mithilfe der Klasse Server über Sockets mit der CGI-Schnittstelle des Webservers kommuniziert. Bei Sockets holt der Server gesendete Daten immer mit `receive()` ab, während er selbst welche mit `send()` zum Client schickt. Startpunkt ist die Funktion `mallotMail()`, innerhalb dieser lauscht der Server am socket mit `get_new_Client()` auf die Verbindungsanfrage eines neuen Clients.

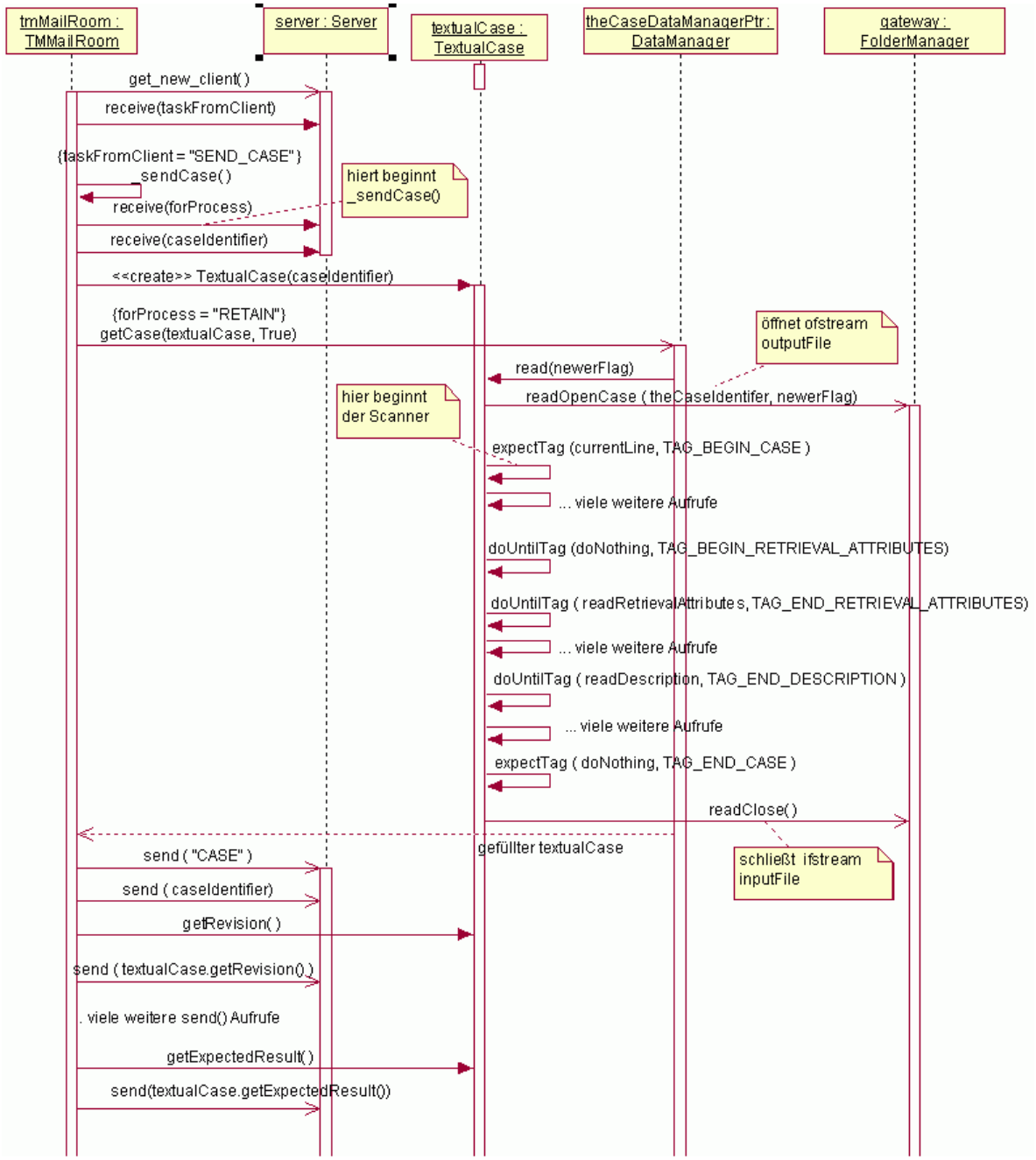

Bild 15: UML-Sequenzdiagramm: Kontrollfluss beim Anfordern eines kompletten Fall(texte)s

Hat er einen neuen Client, holt er sich die erste Nachricht mit `receive (taskFromClient)` vom Client. Ist diese gleich „SEND_CASE“, ruft TMMailRoom an sich selbst `_sendCase()` auf. Darin holt er sich vom Client die gewünschte Sicht und den CaseIdentifer. Ein neues Objekt *TextualCase* mit der FallID wird erzeugt und über den DataManager mittels `getCase(textualCase, True)` aus der Fallbasis gefüllt. Ist wie hier im Beispiel forProcess = „RETAIN“, wird dabei als Parameter NewerFlag=true übergeben. Näheres zum Scanner findet sich unter *3.3.1 Der Scanner auf Seite 20*. Anschließend leitet der Server mittels `send(„CASE“)` die Antwort ein und übermittelt alle Daten des Falls in dem er sie von textualCase abfragt und dann mit `send()` zum Client schickt.
Die Reihenfolge und Inhalte der Kommunikation zwischen Server und Client ist in Kommunikationsprotokollen festgelegt.

3.8.3.1 Hilfsfunktionen buildName() und exist()

buildName() hinterläßt Pfad in completeFileName

Der Pfad- und Dateiname wird mit *buildName()* gebildet und im member `completeFileName` gespeichert, so dass er auch nach Funktionsende noch verfügbar ist. Dazu setzt er den Pfad aus dem pathCaseBase + einem evtl. Parameter `subPath` zusammen, der Dateiname bildet sich aus filePräfix + *caseId* + fileSuffix. Die Präfix- und Suffix-Konstanten sind in der *tm.ini* gespeichert.
Da readOpenCase wie writeOpenCase buildName aufrufen, kann die Datei über *fstream::open() mit `completeFileName` als Parameter danach gleich geöffnet werden.

Zur Feststellung, ob bestimmte Dateien vorhanden sind, dient
`bool exist( const CASE_IDENTIFER caseId, const bool newerFlag ).`

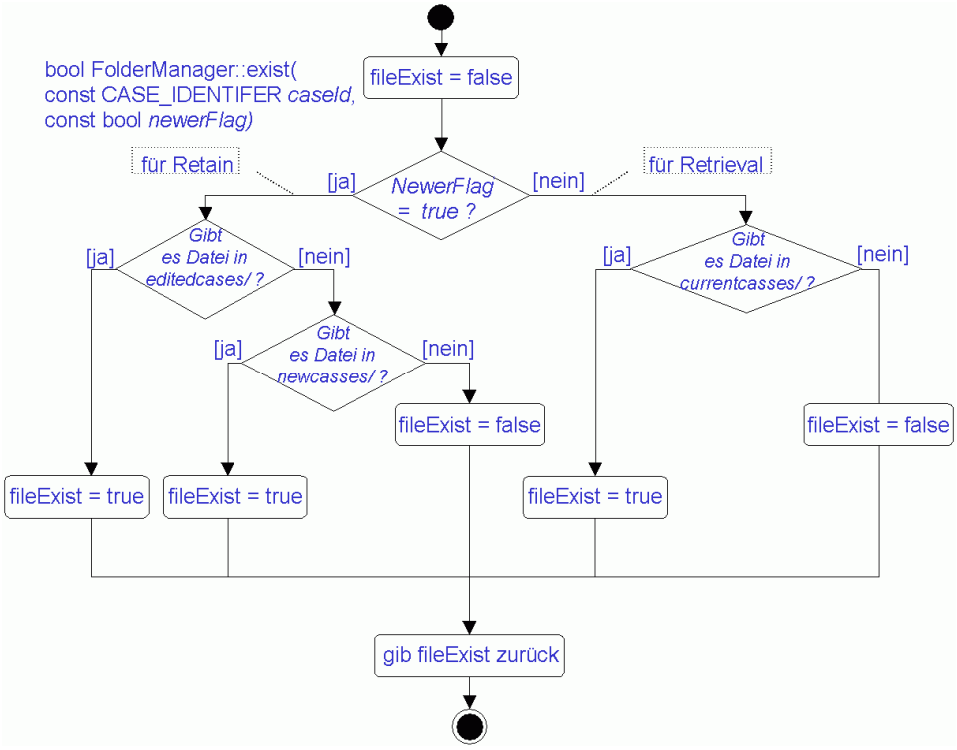

Bild 16: UML-Aktivitätsdiagramm: Ablauf der Funktion exist()

Das Bild 16 zeigt den Kontrollfluß innerhalb dieser Funktion. Sie prüft mit Parameter `NewerFlag = true (false)`, ob wirklich ein neuerer (alter) Fall existiert. Bei true wird dazu erst im *editedcases/* Verzeichnis und im erfolglosen Fall danach in *newcases/* gesucht. Wird auch hier keine Falldatei gefunden, liefert die Funktion false zurück. Ob daneben noch eine „alte“ Falldatei existiert, ist dabei nicht relevant. Ist aber mit `NewerFlag = false` die Retrieval Sicht gefordert, so wird zurückgeliefert ob eine Falldatei im Verzeichnis currencases/ vorhanden ist.

3.8.4 Historie aller Revisionen

Das System besitzt zur Verfolgung der wachsenden Fälle eine History-Funktionalität. Dabei werden für jeden Fall alle Revisionen (abgesehen von der aktuellen, letzten) in einer Datei in *historycases/* gesichert.
Innerhalb von `writeOpenCase()`, welche den Stream zum Schreiben öffnet, wird mittels `backupCaseFile()` der „alte“ Fallinhalt der Revision, welche ja danach überschrieben wird, an diese Datei in *historycases/* angehängt. Dabei wird immer die aktuelle Revision aus der «case authoring» Sicht berücksichtigt, beim ersten Editieren (seit dem CRN Aufbau) also die Datei aus *currentcases/* bzw. bei neuen Fällen aus *newcases/* kopiert, während es beim weiteren Editieren jene aus *editedcases/* ist.
Eine vom Benutzer aufrufbare Undo-Funktion existiert nicht, wäre damit aber einfach zu realisieren.

3.8.5 Löschen eines Falls

Beim Löschen von Fällen mittels *FolderManager::deleteCaseFile()* wird ein Fall in der «case authoring» Sicht gelöscht. Aus «retrieval» Sicht existiert er aus Konsistenzgründen weiterhin. Erst nach einem Update der Fallbasis verschwindet mit der Vereinigung der «case authoring» und «retrieval» Sicht der Fall, sein Life Cycle ist beendet. Dabei wird die letzte aktuelle Revision aus der «case authoring» Sicht wieder mittels backupCaseFile() in die Datei im *historycases/* Verzeichnis gesichert. Deswegen bleiben alle Revisionen auch nach dem Update im *historycases/* Verzeichnis erhalten.

3.8.6 Suchen nach vorhandenen Falldateien

Beim Start des Systems haben weder DataManager noch FolderManager Informationen über vorhandene Fälle. Daher durchsucht die Funktion `getCaseIdsInDirectory()` ein Verzeichnis und gibt in einem DYN_ARRAY alle ID’s von Falldateien zurück, die dem Muster des Dateinamens entsprechen.

```
CASE_INDEX getCaseIdsInDirectory( CASE_IDENTIFER & highestId,
    ID_LDARRAY *idsOfDirectory, String directoryName =
pathCaseBase )
    throw( NoDirFileIOException );
bool isCaseFileInDirectory( String directoryName = pathCaseBase )
    throw( NoDirFileIOException );
```

Dem gegenüber ermittelt `isCaseFileInDirectory()`, ob der angegebene Dateiname im Verzeichnis unterhalb von *currentcases/* existiert und einen gültigen Dateinamen darstellt.

Dabei wird für beide die Funktion `extractIdentifer()` benutzt, um aus einem Dateinamen die Fall-Id zu ermitteln. Nur wenn die Signatur exakt entsprechend *2.4.1 Der gültige Dateiname auf Seite 15* übereinstimmt, wird *true* zurückgegeben und der Wert in *caseId* gespeichert:

```
inline bool
extractIdentifer( String & caseFileName, CASE_IDENTIFER & caseId ).
```

Die Signatur ist eineindeutig, so dass es zu jeder Fall-Id nur einen Dateinamen gibt, der von `extractIdentifer()` akzeptiert wird.

3.9 PresentationTable

Zur Erfüllung der Anforderung der schnellen Lieferung von Fall-Kurzdarstellungen an den DataManager wird im Speicher eine Tabelle mit allen Fall-Kurzdarstellungen gehalten. Darin stehen die „alten“ für die «retrieval» Sicht entsprechend der Reihenfolge beim Einlesen und daran anschließend die „neueren“ für die «case authoring» Sicht entsprechend der Bearbeitungsreihenfolge.
Die nächste freie Stelle in der Tabelle an der eine Kurzdarstellung eines neuen oder geänderten Falles abgelegt werden kann besagt
`CASE_INDEX nextFreeTableIndex.`

Die aktuelle Anzahl der Fälle der Fallbasis steht in den geschützten Attributen
`CASE_INDEX countRetainCases` und
`CASE_INDEX countRetrievalCases`
und kann über die Methode `getCountCases(bool newerFlag)` abgefragt werden (siehe *3.11.5 Größe und Aktualität der Fallbasis auf Seite 40*).

Diese Tabelle ist ein LONG_DYN_ARRAY von *TextualCaseShort*-Objekten. Zur Aufzählung wird der Typ
`typedef long` **`CASE_INDEX`** verwendet, dessen Zählung bei 0 beginnt.
Der Aufbau dieser PresentationTable als auch des Sorters erfolgt durch
`void buildTable() throw( FileIOException )`
im Konstruktor des DataManager als auch bei Aufruf von `updateCaseBase()` nach dem Update der Fallbasis.

3.10 Der Sorter

Um einen Zugriff geordnet nach Änderungszeit, Fall-Id und abhängig von der Sichtweise «case authoring» oder «retrieval» auf die Kurzdarstellungen der Fälle in der `PresentationTable` zu haben, existiert der Sorter als Member `SortMachine` im DataManager. Durch die vollständige Trennung der Daten (Kurzdarstellungen) von den Sortiermechanismen und -indizes wird eine spätere Austauschbarkeit der Sortieralgorithmen ohne weitere Änderungen im DataManager ermöglicht.
Die Anpassungen an verschiedene Sortierkriterien wird mit dem Template `SortEntry` innerhalb des Sorters erleichtert. Der Sorter enthält eine STL <list> für jedes einzelne Sortierkriterium in der «case authoring» Sicht. Lisen ermöglichen das schnelle Anfügen und Löschen in der Mitte.

Außerdem gibt es ein schneller zugreifbares DYN_ARRAY für die Fälle aus der „unveränderlichen“ «retrieval» Sicht, da hier nach dem initialen Aufbau keine Änderungen notwendig sind.
Zum jetzigen Zeitpunkt kann nach Änderungszeit und Fall-Id sortiert werden.

3.10.1 Sorter-Schnittstelle

Der Sorter liefert über `getsorted()` in `tableIndexList` die Indizes geordnet nach dem Sortierkriterium in der PresentationTable zurück, wobei mit `fromCaseIndex` und `toCaseIndex` die Menge eingeschränkt werden kann.

```
CASE_INDEX getSorted( const CASE_INDEX fromCaseIndex,
    const CASE_INDEX toCaseIndex, const bool newerFlag,
    const String sortOrder, INDEX_LDARRAY & tableIndexList ) const
```

Als Parameter `sortOrder` sind „byID“ und „byTime“ zugelassen, ansonsten wird eine WrongSortOrderException geworfen.

Für die Aktualisierung bei Änderungen existieren noch die Methoden

```
createCase( const BasicTextualCase & newcase,
    CASE_INDEX & tableIndex ) throw( DataManagerException ),

changeCase( const BasicTextualCase & changedCase, CASE_INDEX
    & saveTableIndex ) throw( NotExistDataManagerException     und

deleteCase( const CASE_IDENTIFER wasteId )
    throw( NotExistDataManagerException ),
```

an welche jeweils der geänderte TextualCase oder TextualCaseShort und die Position in der `presentationTable` übergeben wird, damit Liste und DYN_ARRAY im Sorter entsprechend den Sortierkriterien aktualisiert werden können. Diese werden in *3.10.3 Aktualisierung bei einem neu erzeugten Fall auf Seite 34ff* eingehend erläutert.

3.10.2 SortEntry-Template

Da die Sortierung nach verschiedenen Kriterien immer nach dem gleichen Muster abläuft, wird dafür ein Template definiert: `SortEntry`. Das Template enthält einen Eintrag für einen Index in der `presentation-Table` und für das Sortierkriterium den zu bindenden Parameter T als Member. Um die Sortierung über der STL Liste zu gewährleisten, muss für den Template-Parameter die **<**-Operation spezifiziert werden.
Darüber hinaus muss in *BasicTextualCase* eine entsprechende get-Funktion für die Gewinnung des zu sortierenden Wertes aus dem Fall existieren.

Entsprechend den beiden Sortierkriterien ID und Sortierzeit gibt es zwei benannte Instanziierungen:

```
typedef SortEntry< CASE_IDENTIFER > ID_ENTRY;
typedef SortEntry< time_t > TIME_ENTRY;
```

Daraus ergeben sich folgende Member im Sorter:

Member im Sorter	DYN_ARRAY für «retrieval» Sicht	list<> für «case authoring» Sicht
Änderungszeit	timeSortedOld	timeSortedNew
Fall-Id	idSortedOld	idSortedNew

Bild 17 zeigt eine Übersicht über die erzeugten Strukturen: In der presentationTable liegen die Kurzdarstellungen der Fälle 1 bis 5 nach dem CRN Aufbau. Es liegen noch keine Falländerungen vor. Die DYN_ARRAYs `idSortedOld` und `timeSorted` enthalten der Sortierreihenfolge nach Einträge welche jeweils das Sortierkriterium Id bzw. Zeit und den Index der PresentationTable enthalten. Z.B. ist der neueste Fall mit der Sortierzeit T=240 jener mit dem Index 1, an diesem Index steht Fall 3. `idSortedNew` und `timeSortedNew` spiegeln in Listen die gleiche Reihenfolge wider.

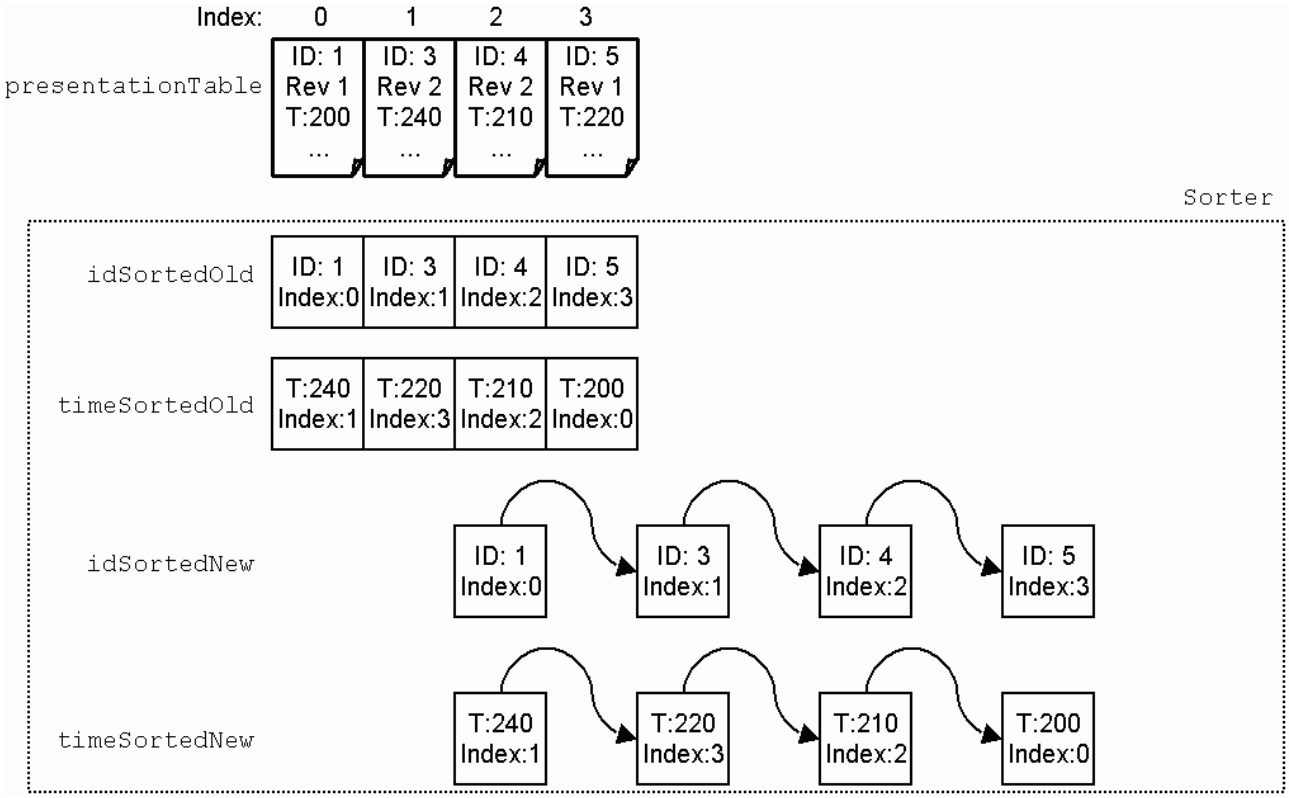

Bild 17: Strukturen der PresentationTable und des Sorters

3.10.3 Aktualisierung bei einem neu erzeugten Fall

Ausgehend von der im Bild 17 gezeigten Ausgangssituation wird ein neuer Fall mit der ID 6 zur Sortierzeit 342 an Index 4 abgelegt. In Bild 18 werden die Ergänzungen blau markiert. Durch

```
idSortedNew.push_back(ID_ENTRY( tableIndex, newcase.getCaseIdentifer() ))
timeSortedNew.push_front(TIME_ENTRY( tableIndex, newcase.getSortTime() ))
```

wird hinten an `idSortedNew` ein Eintrag mit der neuen Fall-ID 6 und dem Index 4 angehängt sowie an `timeSortedNew` ein TIME_ENTRY Element mit der Zeit 342 und gleichem Index 4 vorangestellt. `idSortedOld` und `timeSorted` bleiben unverändert, weil sich aus «retrieval» Sicht nichts ändert.

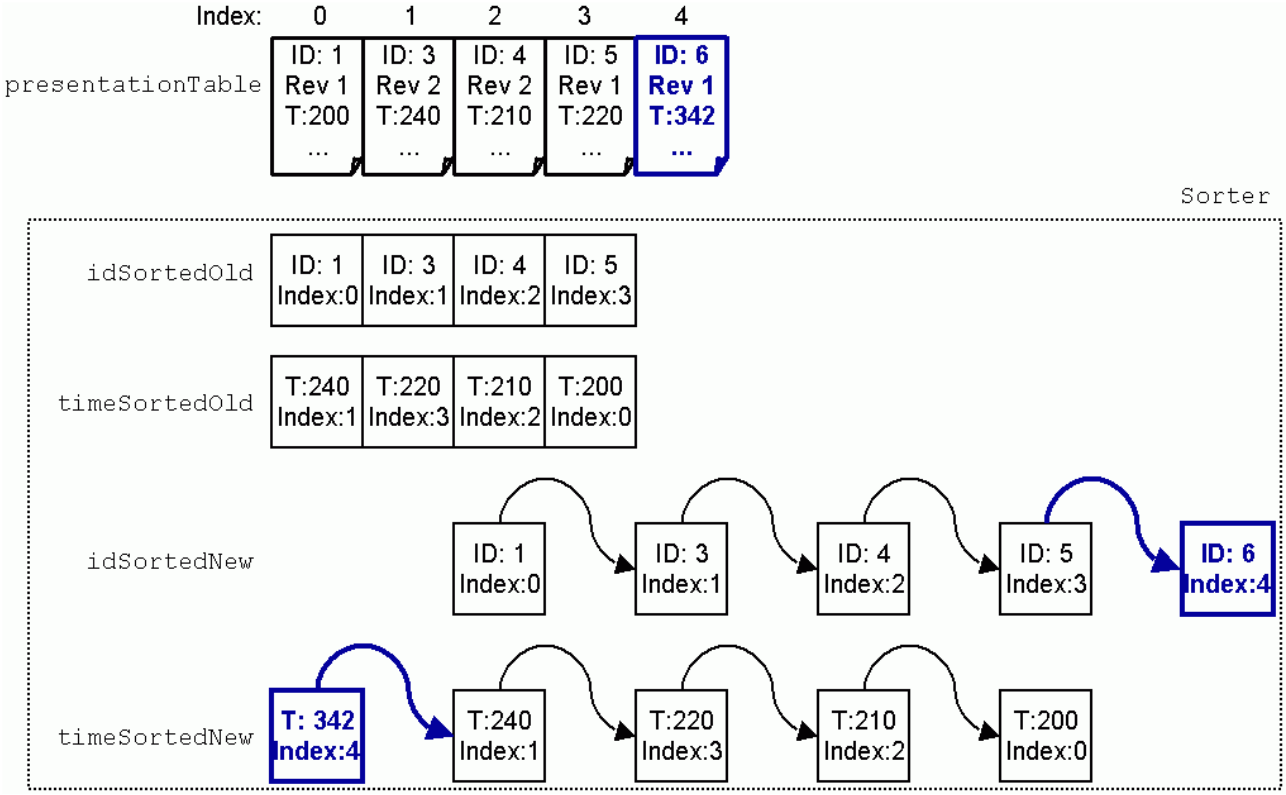

Bild 18: Aktualisierung des Sorters nach Einfügen eines neuen Falls

3.10.4 Aktualisierung bei Änderung an einem Fall / neuer Revision

Ausgehend von der im Bild 18 gezeigten Situation nach dem Anlegen eines neuen Falls wird ein bestehender Fall mit der ID 4 zur Zeit 360 geändert, die Revision erhöht sich auf 3. Bild 19 zeigt die Veränderungen blau an. In `idSortedNew` wird der Eintrag mit der Fall-ID 4 gesucht, der dort vorhandene Index in `movedTableIndex` gesichert und auf 5 geändert. In timeSortedNew wird das Element, welches `movedTable-Index` enthält, gelöscht und ein neues TIME_ENTRY Element mit der Zeit 360 und gleichem Index 5 vorangestellt. Wie im letzen Abschnitt bleiben `idSortedOld` und `timeSorted` unverändert, weil sich aus «retrieval» Sicht nichts ändert. Daher muss auch der „alte“ Textual-CaseShort Eintrag an Index 2 in der `PresentationTable` bestehen bleiben.

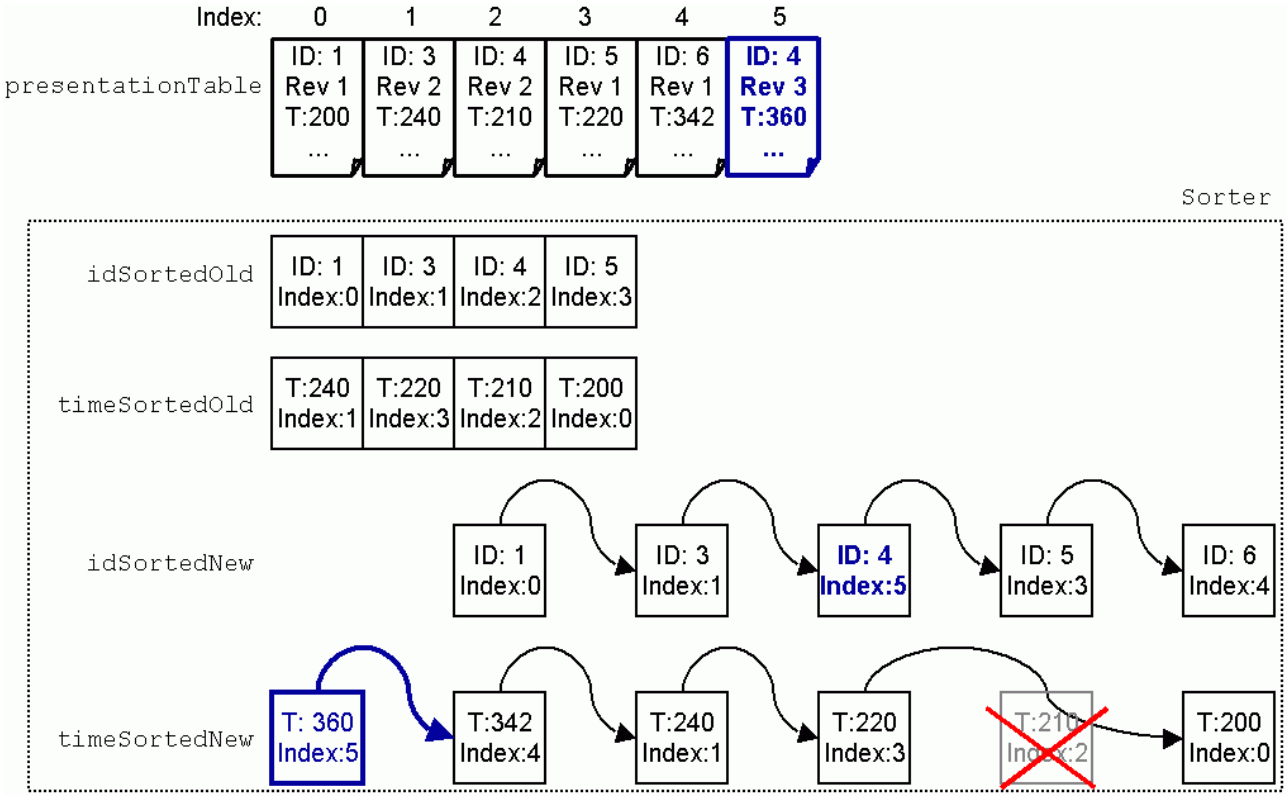

Bild 19: Aktualisierung des Sorters nach Änderungen an einem bestehendem Fall

3.10.5 Aktualisierung bei einem gelöschten Fall

Als Ausgangssituation dient jene in Bild 19 gezeigte. Der Fall mit der ID 5 wird gelöscht. Zur Aktualisierung des Sorters wird in `idSortedNew` der Eintrag mit der Fall-ID 5 gesucht, der dort vorhandene Index in `deletedTableIndex` gesichert und der komplette ID_ENTRY Eintrag aus der Liste entfernt. In `timeSortedNew` wird das Element, welches `deletedTableIndex` enthält, ebenfalls gelöscht. Der TextualCaseShort Eintrag in der `PresentationTable` bleibt dabei bestehen, bis durch ein Update des DataManager werden die `PresentationTable` und die `SortMachine` wieder neu aufgebaut werden.

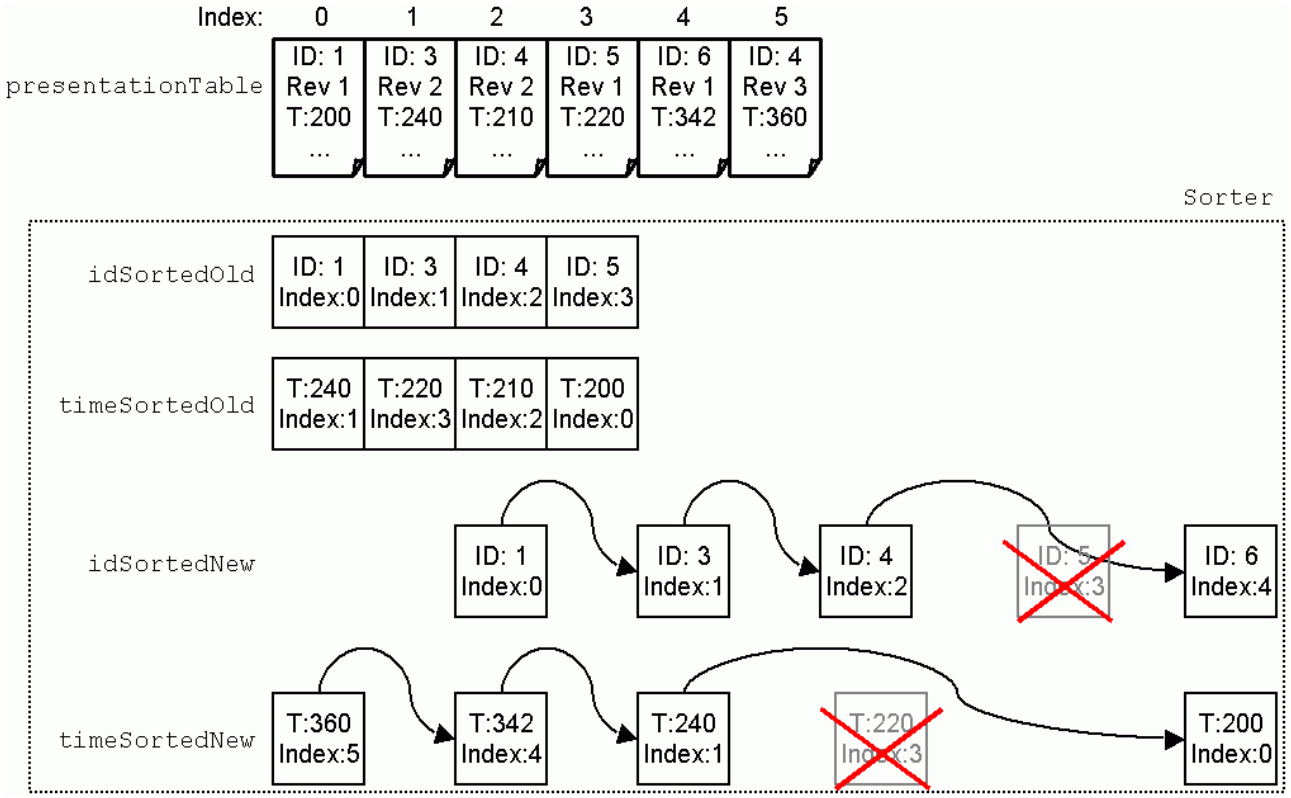

Bild 20: Aktualisierung des Sorters nach Löschen eines Falls

3.10.6 Suchmechanismus

Im Augenblick wird der gewünschte erste Index durch lineare Suche ermittelt. Die Zeitkomplexität fürs Suchen beträgt O(N log N).
Bei großen Fallbasen ist als Erweiterung vorstellbar, Zeiger auf Anfänge von Intervallen dieser Indizes zu halten. Diese müssen aber bei jeder Änderung der Reihenfolge aktualisiert werden. Eine Alternative wäre die Verwendung binärer Bäume anstelle der Liste.

3.11 Schnittstelle des DataManagers

Nachfolgend werden die öffentlichen Operationen des DataManagers aufgezählt, dessen Parameter und Vorbedingungen erläutert.

3.11.1 Liefern eines kompletten Falls

Dazu stehen zwei Operationen zur Verfügung:

```
void getCase( TextualCase & completeCase, const bool newerFlag )
    throw( DataManagerException );
CASE_IDENTIFER getCase( const CASE_INDEX caseIndex,
    TextualCase & completeCase )
    throw( DataManagerException )
```

Beide laden einen Fall aus der Fallbasis und speichern ihn in der übergebenen Referenz auf TextualCase completeCase.
Bei ersterem `void getCase()` muss completeCase eine gültige Fall-ID haben. `newerFlag` entscheidet ob die Revision des Falls aus «case authoring» oder «retrieval» Sicht gefordert ist.

Bei der zweiten Funktion wird der geforderte Fall über den Index spezifiziert. Sie wird für das Parsen des Falls zum Aufbau des CRN benötigt, da das CRN intern nur mit lückenlosem Index arbeitet. Die Abbildung Index→ID durch `sortedIds` wird in `buildTable()` aufgebaut und enthält bei 0 beginnend alle Ids aufsteigend sortiert, so wie die Falldeskriptoren im CRN dann aufgereiht sind. Ist der `caseIndex` zu groß oder <0, wird eine NotExistDataManagerException geworfen. Da diese Funktion von `TMRetrievalManager::_loadCaseBase()` immer für die «retrieval» Sicht aufgerufen wird, fehlt hier `newerFlag`. Sie gibt außerdem die ID des geladenen Falls zurück.

Beide Funktionen werfen eine Exception `DataManagerException`, wenn keine derartige Datei vorhanden ist.

3.11.2 Liefern von Fall-Kurzdarstellungen

Die Anforderung des Zugriffs auf nach bestimmten Kriterien sortierten Fall-Kurzdarstellungen realisieren `getCasePresentations()` und `getCasePresentation()`, wobei letztere nur eine einzige Kurzdarstellung zurück gibt. Erstere gibt zusätzlich noch die Anzahl der (Indizes auf) die Kurzdarstellungen zurück.

```
CASE_INDEX getCasePresentations( const CASE_INDEX fromCaseIndex,
    const CASE_INDEX toCaseIndex, const bool newerFlag,
    const String sortOrder, TextualCaseList & someCasePresentations)
    throw( DataManagerException );
void getCasePresentation( const CASE_INDEX caseIndex,
    TextualCaseShort & casePresentation )
    throw( DataManagerException );
```

Dies erfolgt in zwei Schrittten:

1. Der Sorter liefert über `getsorted()` in `tableIndexList` die Indizes geordnet nach den Sortierkriterien zurück:
2. Gefüllt wird der Rückgabeparameter `someCasePresentations` mit Fallrepräsentationen aus `presentationTable` entsprechend den Indizes in `tableIndexList`.

3.11.3 Speichern eines Falls

Dazu stehen zwei Operationen zur Verfügung:

```
int saveCase( TextualCase & savedCase, const bool forceFlag )
    throw( DataManagerException, SystemCallException );
CASE_IDENTIFER saveNewCase( TextualCase & newCase )
    throw( DataManagerException, SystemCallException );
```

Zur Speicherung eines neuen Falls dient `saveNewCase` , welche die ID des neuen Falls zurückliefert.
Der Fall in `newCase` sollte das Infoattribut *erstelltVon* enthalten. Die Attribute *erstelltAm, geaendertAm*, *Sortierzeit*, *Revision* werden automatisch eingefügt.
Um die Eindeutigkeit der Fall-ID zu gewährleisten, merkt sich der DataManager beim Einlesen der vorhandenen Fälle die größte ID in `CASE_IDENTIFER largestId`. Hat der übergebene Fall `newCase` keine gültige ID (=0 oder <`largestId`), wird diese auf eine gültige gesetzt, d.h. auf die inkrementierte `largestId`.

`saveCase` speichert einen vorhandenen Fall, und gibt die neue Revision zurück oder 0 wenn nicht gespeichert. Der Fall in `savedCase` muss das Infoattribut *Revision* und sollte *geaendertVon*, *erstelltVon*, *erstelltAm* enthalten.

Beide Funktionen werfen eine Exception `DataManagerException`, wenn Schreib-/Lesefehler auftraten, das InfoAttribut Revision nicht vorhanden ist oder das Ändern von *geaendertVon*, *erstelltVon* oder *erstelltAm* fehlschlug.
Der Sorter `sortMachine` und die Tabelle der Fallrepäsentationen presentationTable werden ebenso wie die Zeit der letzten Änderung `time_t caseBaseUpdated` aktualisiert.

3.11.3.1 Konfliktlösung bei konkurrierendem Speichern

Folgendes Konfliktszenario gilt es zu erkennen und zu behandeln:

1. Benutzer A läßt sich Fall X anzeigen.
2. Benutzer B läßt sich den gleichen Fall X anzeigen.
3. Benutzer B editiert den Fall und speichert die Änderungen ab.
4. Benutzer A möchte ebenfalls seine Änderungen am Fall X speichern.
   Ohne Versionskontrolle würde er die Änderungen von B überschreiben!
5. Die hier implementierte Versionskontrolle stellt fest, dass seit dem letzten Laden des Falls Änderung(en) erfolgten und fragt A ob er seine Änderungen trotzdem abspeichern will.

Beim ersten Versuch den Fall zu speichern, ist das `forceFlag` von `saveCase()` auf `false` gesetzt, wird folgende Überprüfung ausgeführt: Da die Revision erst nach erfolgreichem Abspeichern erhöht wird, besagt die Revision im übergebenen `savedCase`, welche als Grundlage des Editierens verwendet wurde. Dazu wird ein neuer Fall erzeugt und aus der Fallbasis geladen um dessen Revision mit der übergebenen zu vergleichen. Ist die Revision in der Fallbasis (hier immer aus «case authoring» Sicht betrachtet) nun höher als die des übergebenen Falls, wurden in der Zwischenzeit schon Änderungen am Fall gespeichert, die mit dem jetzigen Fall überschrieben würden. In diesem Konfliktfall liefert die Funktion 0 für nicht gespeichert zurück. Die aufrufende Komponente fragt daraufhin den Benutzer, welcher dann über ein Abspeichern oder Anzeigen der zwischenzeitlichen Änderungen entscheiden kann.
Bejaht dieser das Überschreiben, wird ein zweiter Versuch zum Speichern gestartet, diesmal aber mit `forceFlag = true`. Es findet dann keine Konflikterkennung statt, ohne weitere Nachfrage wird der Fall als neue Revision gespeichert und überschreibt damit eventuell erfolgte zwischenzeitliche Änderungen. Nicht desto trotz wird die überschriebene Revision aber in der Historie gespeichert. Um aber auch in diesem Fall die Revision zu inkrementieren wird auch hier der TextualCase vorher aus dem Dateisystem gelesen. Grafisch lässt sich dies im Bild 21 so darstellen:

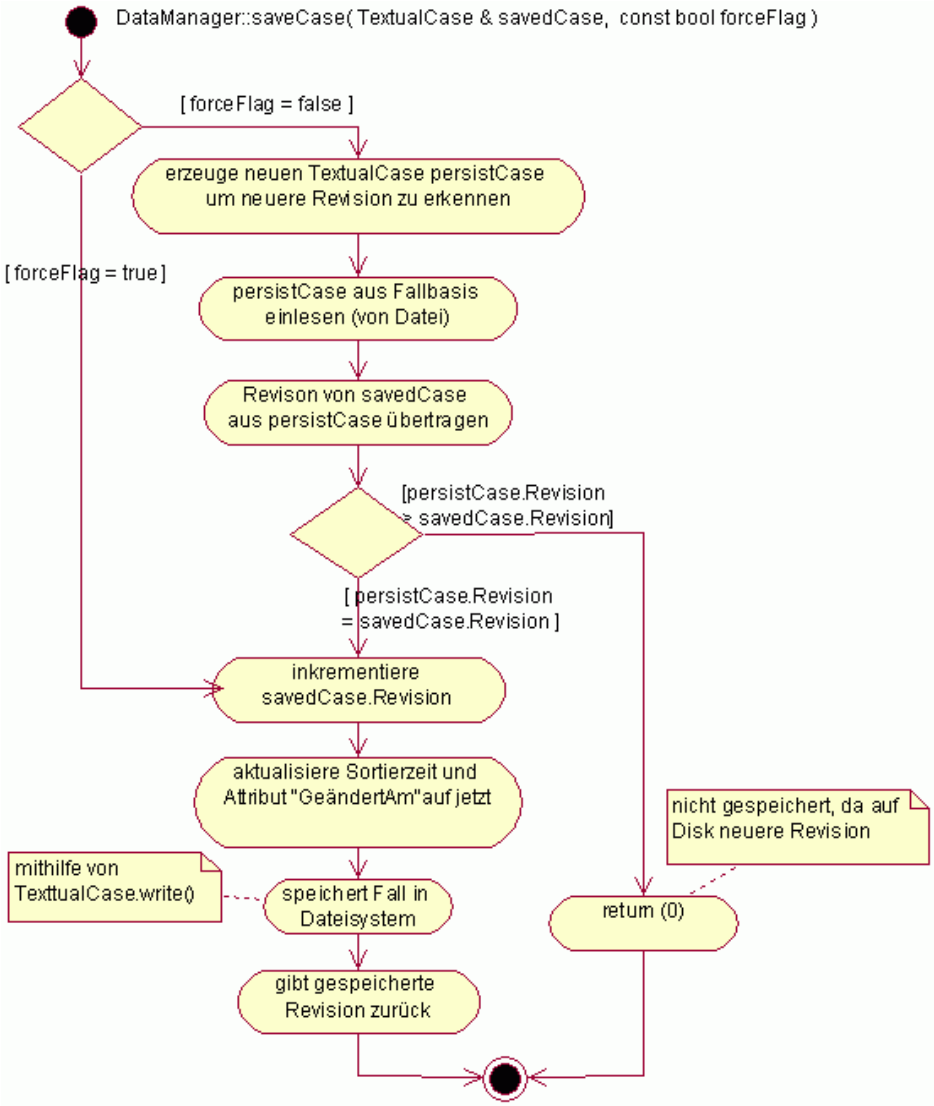

Bild 21: UML-Aktivitätsdiagramm: Ablauf der Funktion saveCase()

3.11.4 Löschen eines Falls

Ein Fall wird durch

```
void deleteCase( const CASE_IDENTIFER CaseId, const bool newerFlag)
    throw( DataManagerException );
```

gelöscht, wobei das `newerFlag` nicht benutzt wird, das Löschen sich also immer auf die «case authoring» Sicht bezieht. Dabei wird vor dem „Löschen“ durch den FolderManager die letzte Revision gesichert. Im DataManager selbst wird der Sorter und das Datum der letzten Änderung angepasst.

3.11.5 Größe und Aktualität der Fallbasis

Die Größe der Fallbasis kann über

```
CASE_INDEX getCountCases( const bool newerFlag )
```

abgefragt werden, wobei `newerFlag` = true (false) die Größe aus «retrieval» («case authoring») Sicht ermittelt.

Die UNIX-Zeit der letzten Änderung der Fallbasis gibt `time_t getTimeStamp()` zurück und bezieht sich direkt auf `caseBaseUpdated`.

3.12 Test des DataManagers

Für verwendete jede Klasse wurde ein Testprogramm *Klasse*_Test.cpp geschrieben, welches die Schnittstelle dieser Klasse testet. Unterteilt wird dabei in mehrere Testfälle, welche unterschiedliche Teile und Aspekte der Schnittstelle behandeln. Insgesamt sind es über 60 Testfälle. Für jede Klasse wurde ebenso eine Textdatei *Klasse*_Test.txt aus den Ausschriften des Testprogramms erzeugt. Die Ausgaben wurden dann von der Testperson in Augeschein genommen, ob sie den erwarteten Werten entsprechen. Bei einer Abweichung wurde der Quellcode verbessert und die Tests erneut ausgeführt, bis bei keinem Testfall eine Abweichung mehr zu den Erwartungen auftritt.

Exemplarisch sei hier das Testprotokoll textualCase_Test.txt der Klasse TextualCase aufgeführt:

Testprotokoll für TextualCase

```
=====================================================
3. Version ergaenzt fuer getRetrievalAttributes()
   (mit Debug Level $2, auch mit Exceptions)
=====================================================


! Test-Case a):
!  Try to get an not existing Case with Id 42
cout ioCase
Fall-Id: 42
theSortTime: 0
RetrievalAttribute:
InfoAttribute:
Testfallbeschreibung:

erwartetes Resultat:


ioCase.read( false )
$6 Datei existiert nicht
currentcases/TM01Case_0000000042.txt:
$6 OpenFileIOException with No file found for
reading: currentcases/TM01Case_0000000042.txt for
case with Id 42! Fehler waehrend read() bei a)
gefangen


! Test-Case b):
!  Change Case with Id 46 and save them
$2 RO oeffnet Fall in pathCaseBase
currentcases/TM01Case_0000000046.txt
Fall-Id: 46
theSortTime: 900000080
```

```
RetrievalAttribute:
Kategorie = Leitsystem
Kategorie = Simulation
Themenbereich = Oberflaeche
Status = Testfall
InfoAttribute:
erstelltAm = 12. 08. 1998
erstelltVon = hanft
geaendertAm = 05. 10. 1998
geaendertVon = minor
Revision = 4
Sortierzeit = 900000080
Kategorie = Systmetest
Testfallbeschreibung:
Testfallbeschreibungen werden oft geaendert.
erwartetes Resultat:
Unix ist an den Umlauten schuld.


$2 copy 1nd source
currentcases/editedcases/TM01Case_0000000046.txt
$2 to target:
currentcases/historycases/TM01Case_0000000046.txt
$2 WO oeffnet editierten Fall in
currentcases/editedcases/TM01Case_0000000046.txt


! Test-Case c):
!  holt retrieval attribute Liste

!  Ausgabe einzeln ueber GetNthElement() :
Pair Nr. 0 Kategorie = Leitsystem
Pair Nr. 1 Kategorie = Simulation
Pair Nr. 2 Themenbereich = Oberflaeche
Pair Nr. 3 Status = Testfall

!  Ausgabe ueber AttributeValuePairList::printAll()
:
Kategorie = Leitsystem
Kategorie = Simulation
Themenbereich = Oberflaeche
Status = Testfall
```

In Testfall a) wird versucht einen nicht existierenden Fall zu öffnen, erwartet wird dabei die `OpenFileIOException`. Testfall b) zeigt einen Fall 46, der aus der «retrieval» Sicht aus `currentcases/` gelesen wird. Beim Abspeichern als neue Revision zeigen die Ausschriften, dass dabei die Falldatei vom Verzeichnis `editedcases/` zur Sicherung der Historie ins Verzeichnis `historycases/` kopiert wird. Anschließend wird der Fall aus der «authoring» Sicht aus dem Verzeichnis `editedcases/` gelesen. Im Testfall c) wird der Reihe nach auf die Retrievalattribute mittels `GetNthElement()` zugegriffen, die abschließende Ausgabe über `printAll()` zeigt dass dies auch alle und auch genau diese waren.

4 Benutzte Komponenten

4.1 Die Klasse String

Für die String-Verarbeitung im Projekt TestManager wurde ein eigener String Typ verwendet, der vom STL <string> abgeleitet ist. Dieser eigene Typ war notwendig um für bestimmte Methoden eine vom STL string abweichende Semantik zu definieren, auf Compilern unterschiedlicher Hersteller einen String mit immer gleicher Semantik zu haben und um zusätzlich benötigte Funktionalitäten anzubieten.
Hier eine Übersicht über die Klasse:

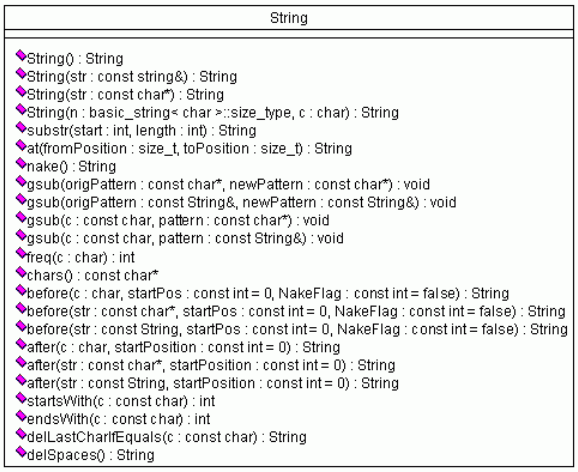

Bild 22: UML-Klassendiagramm vom String

4.2 Dynamisches Array

`DYN_ARRAY` wurde ursprünglich von Mario Lenz zur Vermeidung von Speicherlecks bei der list<> Implementierung des Borlands C++ Compilers entwickelt und ist mit dem STL Template <vector> vergleichbar. Zum Löschen von Elementen wurde noch `RemoveNthElement()` ergänzt.
`DARRAY_Counter` dient statt int als Aufzählungstyp (Laufvariable), zur besseren Zweckbestimmung der Variablen.
Für die Aufbewahrung von Fällen wurde ein analoger Container mit long als Aufzählungstyp erstellt: `LONG_DYN_ARRAY` mit `LDARRAY_Counter`. Bei `DYN_ARRAY` ist die Anzahl der enthaltenen Elemente auf den Zahlenbereich von int (>0) beschränkt, während dies bei `LONG_DYN_ARRAY` long ist.
Beide Counter-Definitionen finden sich in DataManagerDefs.h, währenddessen die Templates im Modul GENERAL in den Dateien dynArray.h und longDynArray.h lokalisiert sind.

4.2.1 Einsatz des Templates

`DYN_ARRAY` ist auch die Basis für eine Ableitung zu `AttributeValuePairList`, was in *3.2 Die Aufzählung AttributeValuePairList auf Seite 18* oben gezeigt wird.

Zur besseren Lesbarkeit wurden häufig benutzte Instanzierungen für Fälle mit Typdefinitionen versehen:

```
typedef LONG_DYN_ARRAY< CASE_IDENTIFER > ID_LDARRAY;
typedef LONG_DYN_ARRAY< CASE_INDEX > INDEX_LDARRAY;
typedef DYN_ARRAY< CASE_IDENTIFER > ID_DARRAY;
typedef DYN_ARRAY< CASE_INDEX > INDEX_DARRAY;
```

Die letzten beiden Definitionen werden allerdings im TestManager nicht benutzt.
Im folgenden Bild 23 seien die strukturellen Beziehungen zwischen Template und AttributeValuePairList dargestellt. `AttributeValuePairList` ist nicht unmittelbar vom Typ `DYN_ARRAY <AttributeValuePair>`, also das mit dem `AttributValuePair` instanziiertes Template, sondern eine Ableitung davon, weil noch zusätzliche Funktionalität, wie z.B. die Suche mittels `findPos()` gefordert ist.

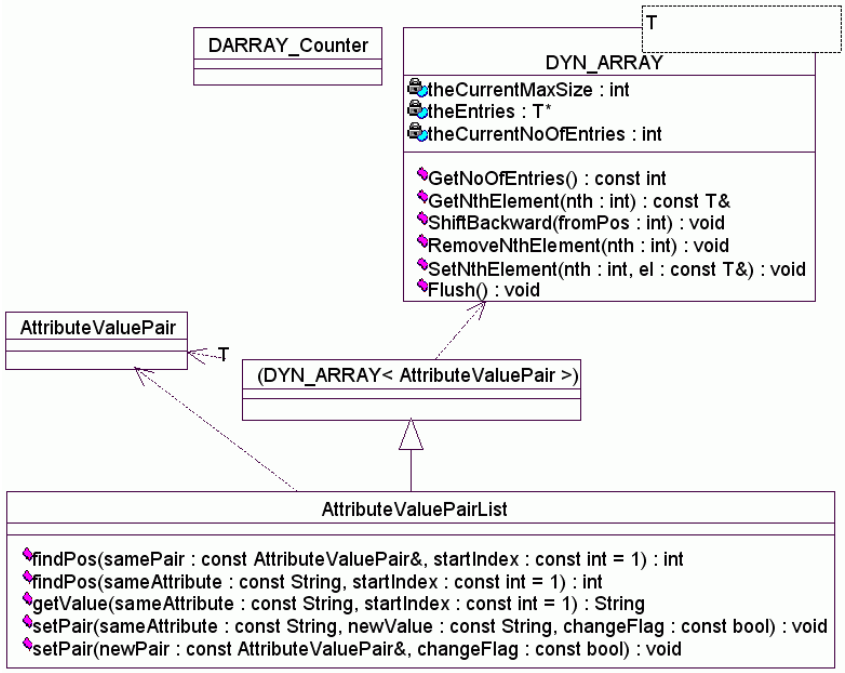

Bild 23: UML-Klassendiagramm: AttributeValuePairList ist eine Ableitung der Instanzierung von DYN_ARRAY

4.3 Ausnahmebehandlung

4.3.1 Basisklasse MyException

Die Basisklasse aller verwendeten Ausnahmen im TestManager ist `MyException`. Sie enthält die komplette notwendige Funktionalität, so dass bei Ableitungen nur Konstruktoren hinzukommen. Bild 24 zeigt die Klassendefinition. Sie verfügt über ein Makro `throwException()`, welches die auslösende Codestelle (Datei und Zeile) mit `setFileName()` und `setLineNumber()` im Quelltext generiert.
Es gibt zwei unterschiedliche Meldungstexte, einen kürzeren und einen ausführlicheren, die sich mit `getDescription()` und `getDetailDescription()` abfragen lassen.

Bild 24: Member und Methoden der Exception-Basisklasse

Hier seien einige Ableitungen beispielhaft aufgeführt.

4.3.2 NotFoundException

Die NotFoundException wird geworfen, wenn erwartete Elemente nicht gefunden werden, z.B. Attribute in AttributeValuePairList. Diese Exception gehört zum Teil zum erwarteten Programmablauf.

4.3.3 FileIOException

Eine FileIOException oder deren Ableitung wird bei nicht zu öffnenden Dateien, nicht lesbaren Verzeichnissen, Lesefehlern und EOF innerhalb von Dateien geworfen. Durch Spezialisierung wird angezeigt ob der Fehler z.B. beim Öffnen einer Datei auftrat: `OpenFileIOException`.

4.3.4 SyntaxFileException

SyntaxFileException wird bei Fehlern im Fallformat in der Datei geworfen.

Im folgenden Bild 25 sei eine Übersicht über die im Projekt TestManager verwendeten Exception Klassen dargestellt. Die obersten Ableitungen stellen Kategorien von Fehlerklassen dar, welche bei weiterer notwendiger Unterscheidung weiter spezialisiert werden. Alle fettgedruckten Klassen werden im DataManager verwendet.

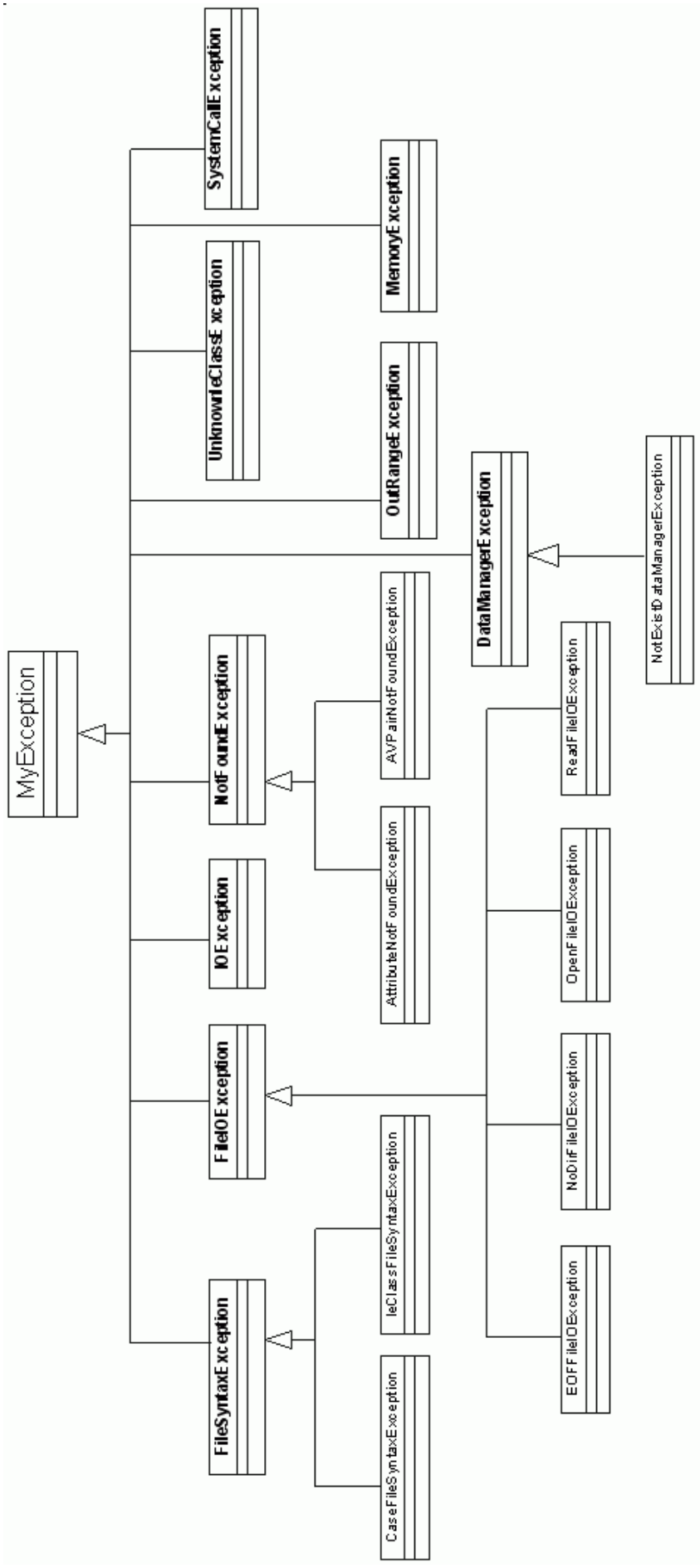

Bild 25: Übersicht aller verwendeten Exception Klassen

Abbildungsverzeichnis

Literatur

[AamodtPlaza94] A. Aamodt, E. Plaza (1994); AICom - Artificial Intelligence Communications, IOS Press, Vol. 7: 1, pages 39–59.

[BuschmannEtal1998] Frank Buschmann, et al: „Pattern-orientierte Software-Architektur, Ein Pattern-System“, Addison Wesley Longmann, Bonn 1998

[Kunze98] Mirjam Kunze: Das ExperienceBook - Dokumentation eines fallbasierten Systems zur Unterstützung der Systemadministration Diplomarbeit, Humboldt-Universität zu Berlin, 1998. http://www.informatik.hu-berlin.de/~minor/Publications/diplomarbeit.zip

[LenzEtal98] M. Lenz, A. Hübner, M. Kunze: Textual CBR. In: Case-Based Reasoning Technology - From Foundations to Applications, Lenz M., Burkhard HD., Bartsch-Spörl B., Wess S. (Hrsg.), LNAI 1400, Springer Verlag, Berlin, 1998.

[LenzEtal99] M. Lenz, K.-H. Busch, A. Hübner, and S. Wess. The SIMATIC Knowledge Manager. In D. W. Aha and H. Mũnoz-Avila. Exploring Synergies of Knowledge Management and Case-Based Reasoning. Technical Report AIC-99-008, Naval Research Lab 1999, pages 40–45.

[Lenz99] M.Lenz: Case Retrieval Nets as a Model for Building Flexible Information Systems, Dissertation, Humboldt-Universität zu Berlin, 1999.

[MinorHanft1999] Mirjam Minor, Alexandre Hanft: Cases with a Life-Cycle. in: S. Schmitt, I. Vollrath (Eds.): Challenges for Case-Based Reasoning - Proceedings of the ICCBR'99 Workshops, Universität Kaiserslautern, 1999.

[MinorHanft2000] Mirjam Minor, Alexandre Hanft: The Life Cycle of Test Cases in a CBR System. Proc. EWCBR-2000, LNAI 1898, pages 455–466, Springer Verlag, 2000.

[Richter2003] Michael M. Richter: Fallbasiertes Schließen S. 407–430 in Görz G., Rollinger C.-R., Schneeberger, J. (Hrsg): Handbuch der Künstlichen Intelligenz, Oldenburg Wissenschaftsverlag, München Wien, 4. Auflage, 2003