



Master Thesis

Semantisches Filesystem ObjectSFS

Author(s):

Imfeld, Alexander

Publication Date:

2002

Permanent Link:

<https://doi.org/10.3929/ethz-a-004439723> →

Rights / License:

[In Copyright - Non-Commercial Use Permitted](#) →

This page was generated automatically upon download from the [ETH Zurich Research Collection](#). For more information please consult the [Terms of use](#).



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

**Department of
Computer Science**

Institute for
Information Systems
Database Research
Group

Diplomarbeit

22.4 - 21.8.2002

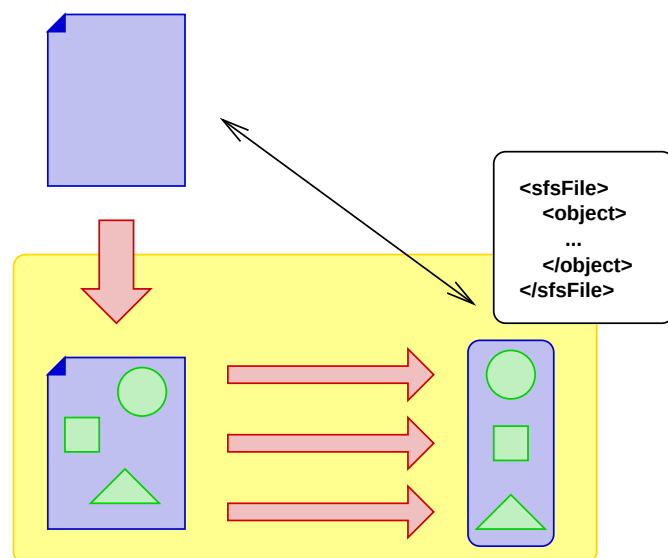


Alexander Imfeld
(D-INFK)

Professor
Prof.Dr.H.-J.Schek

Assistentin
Pamela Ravasio

Semantisches Filesystem ObjectSFS



Inhaltsverzeichnis

1. Einleitung	1
1.1. Motivation	1
1.2. Ziele	3
1.3. Überblick	4
2. Dateisysteme	5
2.1. Dateiformat	5
2.2. Dateiattribute	6
2.3. Organisation der Dateien	6
2.4. Zusammenfassung	8
3. Semantische Dateisysteme	11
3.1. Dateisemantik	11
3.2. Architekturen Semantischer Dateisysteme	12
3.3. Aktuelle Entwicklungen und Systeme	16
3.4. Zusammenfassung und Diskussion	18
4. ObjectSFS	21
4.1. Typen	21
4.2. Objekte	26
4.3. Dateien	33
4.4. Konfiguration	36
4.5. Verarbeitung	38
4.6. Zusammenfassung	50
5. Grafische Benutzeroberfläche	53
5.1. Configurator	54
5.2. Filemanager	59
5.3. Objektbrowser	61
5.4. Zusammenfassung	63

6. Implementation	65
6.1. Plugins	65
6.2. Aufnahme von Dateien	69
6.3. Konsistenz	69
6.4. Persistenz	72
6.5. Zusammenfassung	74
7. Konfigurationen	75
7.1. Extraktion vorhandener Metadaten	75
7.2. Dynamische Klassifikation von Texten	80
8. Konklusion	85
8.1. Diskussion und offene Fragen	85
8.2. Erweiterungsmöglichkeiten	86
A. Installation	89
B. Klassendiagramm	91
C. Beispielkonfigurationen	93
C.1. Beispielkonfiguration <i>meta</i>	93
C.2. Beispielkonfiguration <i>text</i>	100
D. Aufgabenstellung	105
D.1. Introduction	105
D.2. Brief characterization of a SFS	105
D.3. Project plan	105
Literaturverzeichnis	107
Index (Typen, Java-Klassen und -Interfaces)	109

1. Einleitung

1.1. Motivation

Informationen werden in heutigen Desktopsystemen überwiegend in Dateien gespeichert. Die Dateien kapseln von Applikationen erzeugte Daten, welche über programminterne Verarbeitungsschritte oder auf Grund von Benutzerinteraktionen (z.B. Tastatureingaben) mit der Applikation entstehen. Tabelle 1.1 listet einige mögliche Dateidaten mit- samt der verantwortlichen Erzeugerapplikation auf.

Unformatierter Text	← Editor
Formatiertes Textdokument	← Textverarbeitungs- oder Textsatzprogramm
Tabelle (Spreadsheet)	← Tabellenkalkulationsprogramm
Technische Zeichnung	← CAD-Programm
Grafik	← Bildverarbeitungsprogramm
Quellcode	← Editor oder Codegenerator
Maschinencode	← Compiler

Tabelle 1.1.: Von Applikationen erzeugte Dateidaten

Semantische Informationen über Dateien charakterisieren den Inhalt einer Datei, die Dateidaten. Sie beschreiben mehr als nur die physikalischen Eigenschaften (wie Anzahl Bytes usw.) einer Datei. Sie ergeben sich aus einem *Verständnis*, einer *Interpretation* des Dateiinhalts und geben den Dateidaten eine *Bedeutung*. Zum Beispiel stellt die Aussage, dass es sich beim Inhalt einer Datei um ein Textdokument handelt, eine semantische Information über diese Datei dar. Weitere semantische Informationen über eine Datei könnten folgende Angaben umfassen:

- Autor des Textdokuments
- Titel des Textdokuments
- Im Text vorkommende Schlüsselwörter
- Art des Textes (Brief, Rechnung, ...)
- Von der Art des Textes abhängige Angaben (Empfänger des Briefes, Rechnungsbetrag, ...)

Semantische Informationen über Dateien unterstützen den Benutzer oder die Applikationen hauptsächlich bei der Suche nach einer Datei oder einer Menge von Dateien auf Grund von inhaltsspezifischen Angaben. Sie liefern für eine bestimmte Datei aber auch einen kurzen Überblick über die Art und die Attribute des Inhalts.

Dateisysteme ermöglichen und regeln den Zugriff auf Dateien. Die heute gebräuchlichen Dateisysteme sind allerdings nicht oder nur sehr beschränkt (im Rahmen von einigen festgelegten Standard-Dateiattributen) dazu in der Lage, semantische Informationen über Dateien zu extrahieren, ins System einzubinden und für den Dateizugriff anzubieten. Dateisysteme betrachten die Dateien im Allgemeinen als eine Sequenz von Bytes, ohne die Bedeutung oder die logische Struktur dieser Bytesequenz zu kennen. Der Zugriff auf Dateien geschieht bei den angesprochenen Dateisystemen in der Regel über den Namen der Datei. Ein inhaltsbasierter Zugriff auf Dateien, wie er unter Einbezug semantischer Informationen möglich wäre, wird nicht unterstützt.

Die semantischen Informationen zu den Dateien liegen teilweise in den Applikationen, welche diese Dateien erzeugen und manipulieren. Eine typische Textverarbeitung beispielsweise hat Kenntnis über die logische Struktur der von ihr verarbeiteten Dateien. Sie arbeitet auf der Ebene von Überschriften, Abschnitten und Worten. Oft unterstützt sie auch die Angabe von zusätzlichen Informationen des Benutzers (wie zum Beispiel Autor, Stichwörter usw.), welche den Dateiinhalt näher charakterisieren.

Die von Applikationen angebotenen semantischen Informationen zu den von ihnen erzeugten oder verarbeiteten Dateien sind allerdings beschränkt. Ausserdem gehen sie selten über Strukturinformationen und benutzerspezifizierte Informationen hinaus. Eine gewöhnliche Textverarbeitung kennt die Anzahl der Wörter in einem Text, ist aber nicht dazu in der Lage, die Schlüsselwörter zu extrahieren oder die Art des geschriebenen Textes zu bestimmen (z.B. *Brief* oder *Rechnung*). Ein übliches Bildbearbeitungsprogramm weiss um die Dimension einer Grafik, erkennt aber nicht das dargestellte Motiv.

Spezialisierte Programme wurden und werden entwickelt um nichttriviale semantische Informationen aus Dateien mittels Analyse der Dateidaten zu extrahieren. So gibt es heute Programme zur Indexierung oder zum Zusammenfassen von Texten, zur Text- und Spracherkennung, zur Bildanalyse und Mustererkennung usw.

Semantische Informationen zu Dateien sind grundsätzlich vorhanden oder können extrahiert werden. Bietet sich der Zugang zu den Informationen allerdings nur über entsprechende Applikationen oder Analyseprogramme, so wird die Sicht auf die Informationen durch die jeweilige Applikation eingeschränkt und verliert ihre globale Verfügbarkeit. Eine Einbettung der semantischen Informationen in das Dateisystem hätte eine einheitliche Verwaltung derselben zur Folge und würde die Integration der Information auf verschiedenen Ebenen ermöglichen [23]:

- Integration auf der Ebene der Schnittstellen: Der Zugriff und die Manipulation der Informationen erfolgt einheitlich und standardisiert.

- Integration auf der Ebene von Anfragen und Suche: Die Informationen sind zentral verfügbar und über eine klar definierte Schnittstelle zugänglich.
- Integration auf der Ebene von Referenzen: Beziehungen können erkannt werden, da eine globale Sicht auf die Informationen besteht.
- Integration auf der Ebene von Namensräumen: Die Konsistenz der Namensgebung ist gewährleistet. Eine semantische Information kann somit eine global definierte *Bedeutung* bekommen.
- Integration auf weiteren Ebenen wie Transaktionen, Change-Management, Sicherheit, Fehlertoleranz usw.

1.2. Ziele

Das Ziel dieser Arbeit ist die Implementation einer Schicht über einem konventionellen Dateisystem, welche semantische Informationen aus Dateien extrahiert, verwaltet und nach aussen zugänglich macht. Neben der automatischen Extraktion von Informationen soll das System auch semantische Informationen einbeziehen, die vom Benutzer oder von anderen Applikationen für Dateien erstellt wurden.

Die semantische Schicht bietet einen einheitlichen Zugang auf semantische Informationen über Dateien, ohne Anforderungen an das darunterliegende Dateisystem zu stellen. Dieser Ansatz erlaubt das Erstellen eines plattformunabhängigen Systems, welches eine erweiterte Schnittstelle zu den Dateien anbietet.

Die Implementation des Systems soll den folgenden Anforderungen gerecht werden:

- Plattformunabhängigkeit (Implementation in Java)
- Einfacher Aufbau
- Generizität: Möglichst grosse Dateitypenunabhängigkeit
- Flexible Schnittstelle für den Zugriff auf semantische Informationen
- Möglichkeit der Integration benutzerspezifischer semantischer Informationen
- Konfigurierbarkeit, um verschiedene Benutzerbedürfnisse abzudecken
- Erweiterbarkeit

1.3. Überblick

Kapitel 2 gibt einen Überblick über traditionelle Dateisysteme und Dateisystemkonzepte, welche im Rahmen dieser Arbeit von Bedeutung sind. In Kapitel 3 folgt eine Einführung in das Gebiet der semantischen Dateisysteme. Die verschiedenen Architekturen semantischer Dateisysteme werden anhand von konkreten Implementationen vorgestellt. Anschliessend folgt ein Einblick in die Entwicklung aktueller verwandter Systeme. Kapitel 4 liefert einen ersten Überblick über die Konzepte von OBJECTSFS, dem in dieser Arbeit entstandenen semantischen Dateisystem, während Kapitel 5 die grafische Benutzeroberfläche einer Test-Applikation zur Interaktion mit OBJECTSFS vorstellt. Kapitel 6 geht danach vertieft auf einige Implementationsaspekte von OBJECTSFS ein. In Kapitel 7 werden zwei Konfigurationen für OBJECTSFS untersucht und deren Leistungsfähigkeit in der Praxis diskutiert. Zum Abschluss des Hauptteils dieser Arbeit findet sich in Kapitel 8 eine Diskussion des entstandenen Systems, eine Ausführung zu aufgetretenen Problemen und offenen Fragen, sowie eine Übersicht über Erweiterungsmöglichkeiten von OBJECTSFS.

Im Anhang findet sich neben einer Installationsanleitung für OBJECTSFS und einer Übersicht über die vom System zusätzlich benötigten Komponenten, ein detailliertes Klassendiagramm, welches den Aufbau von OBJECTSFS veranschaulichen soll. Ausserdem liegen im Anhang auch die in Kapitel 7 besprochenen Konfigurationen vor. Sie sind in komplettem Umfang gezeigt und können zu Referenzzwecken verwendet werden. Zum Schluss folgt die Aufgabenstellung, welche der vorliegenden Diplomarbeit zu Grunde liegt.

2. Dateisysteme

Ein klassisches Dateisystem stellt einen integralen Bestandteil eines Betriebssystems dar. Durch das Ablegen von Daten in Dateien und deren Speicherung in einem Dateisystem werden die drei wesentlichen Anforderungen erfüllt, die sich an die langfristige Speicherung von Informationen stellen [21]:

- Speicherung sehr grosser Mengen von Informationen.
- Persistenz der Informationen. Die Informationen überleben die Terminierung der Prozesse, die diese erzeugen.
- Mehrere Prozesse können gleichzeitig auf die gespeicherten Informationen zugreifen.

Die Aufgaben eines Dateisystems setzen sich aus der Verwaltung von Dateien und aus dem Bereitstellen von Mechanismen für den Zugriff auf Dateien zusammen. Diese Aufgaben müssen einerseits auf der physikalischen Ebene (Abbildung von Dateien auf das physikalische Speichermedium), andererseits auf der logischen Ebene (Darstellung des Dateisystems dem Benutzer und anderen Applikationen gegenüber) vom Dateisystem bewältigt werden.

Im Folgenden sollen verschiedene Aspekte klassischer Dateisysteme vorgestellt und ihre Rolle im Hinblick auf semantische Informationen und den Dateizugriff diskutiert werden.

2.1. Dateiformat

Auf der Ebene des Dateisystems wird jede Datei als eine Sequenz von Bytes behandelt. Das gemeinsame Modell einer Datei zwischen dem Dateisystem und einer Applikation ist ein Bytestrom. Das Dateisystem nimmt einen Bytestrom von einer Applikation zur Speicherung in einer Datei entgegen, während die Applikation einen Bytestrom als Folge einer Leseoperation aus einer Datei vom Dateisystem empfängt. Die Applikation weiss dabei nichts über die interne Speicherung des Bytestroms im Dateisystem und das Dateisystem seinerseits weiss nichts über die applikationsspezifische Interpretation des gelieferten Bytestroms.

Die uniforme Sicht auf eine Datei als Bytestrom verunmöglicht es dem klassischen Dateisystem, Semantische Informationen aus dem Inhalt der Datei zu extrahieren. Die Dateidaten haben für das Dateisystem keine definierte Semantik, diese wird den Daten erst durch die entsprechende Applikation verliehen.

2.2. Dateiattribute

Das Dateisystem speichert üblicherweise für jede Datei neben dem Dateinamen und den Dateidaten eine kleine Menge von Attributen. Die Anzahl und die Zusammensetzung der gespeicherten Attribute unterscheidet sich von System zu System. Tabelle 2.1 listet einige der am häufigsten verwendeten Dateiattribute auf [21].

Für die meisten in Tabelle 2.1 aufgelisteten Dateiattribute gilt, dass sie keine semantische Information über die Datei liefern (Als Ausnahmen können *Erzeuger*, *Eigentümer*, *ASCII/Binär-Flag* und das *Erzeugungsdatum* angesehen werden.). Die Attribute dienen hauptsächlich betriebssysteminternen Zwecken wie zum Beispiel dem Zugriffsschutz oder der Regelung des internen Datenzugriffs.

Gewisse Dateisysteme wie zum Beispiel NTFS (*New Technology Filesystem*) [12] oder HFS (*Macintosh Filesystem*) [7] erlauben es dem Programmierer, neben den Basis-Dateiattributen weitere benutzerdefinierte Dateiattribute anzulegen. Die Attribute werden zwar vom Dateisystem verwaltet, ihre Semantik ist aber durch die Applikation bestimmt, welche die Attribute angelegt hat. Eine sinnvolle Verwendung dieser Informationen durch andere Applikationen erfordert die Standardisierung dieser Attribute¹.

2.3. Organisation der Dateien

Gebräuchliche Dateisysteme sind in der Regel hierarchisch organisiert. Dateien werden in Verzeichnissen abgelegt, welche ihrerseits eine spezielle Art von Dateien verkörpern. Die Verzeichnisse bilden eine Baumstruktur mit den eigentlichen Dateien als Blätter des Baumes. Abbildung 2.1 zeigt ein hierarchisches Dateisystem mit Verzeichnissen und Dateien aus der Sicht des Benutzers.

Das hierarchische Dateisystem, wie es in Abbildung 2.1 gezeigt ist, benutzt die Metapher der Mappe oder des Ordners für Verzeichnisse. Ein solcher Ordner dient als Behälter für eine beliebige Anzahl anderer Ordner oder Dateien.

Eine Datei in einem hierarchischen Dateisystem ist eindeutig durch ihren Pfad im Verzeichnisbaum bestimmt. Der Name des Pfades setzt sich aus den Namen der Verzeichnisse (durch einen Separator getrennt) gefolgt vom Namen der Datei zusammen (‘/usr/bin/less’ bezeichnet beispielsweise die Datei mit dem Namen ‘less’, die sich im

¹In Windows 2000 werden zu den Dateien einige wenige festgelegte Attribute wie zum Beispiel *author* im Dateisystem abgelegt. Diese Attribute können für einfache Suchanfragen über den *Explorer* verwendet werden [12].

Zugriffsrechte	Bestimmen, wer auf welche Art auf die Datei zugreifen darf.
Passwort	Passwort, das für den Zugriff auf die Datei benötigt wird.
Erzeuger	Identifikation des Benutzers, der die Datei erzeugt hat.
Eigentümer	Aktueller Eigentümer der Datei (<i>owner</i> in Unix).
Gruppe	Identifikation der Benutzergruppe, der die Datei zugeordnet ist (<i>group</i> in Unix).
Read-Only-Flag	Zeigt an, ob der Zugriff auf die Datei nur lesend oder auch schreibend erfolgen darf.
Hidden-Flag	Signalisiert, ob die Datei in einem Verzeichnis-Listing erscheinen soll.
System-Flag	Zeigt an, ob es sich bei der Datei um eine Systemdatei handelt.
Archiv-Flag	Zeigt an, ob von der Datei eine Sicherungskopie erstellt wurde.
ASCII/Binär-Flag	Markiert die Datei als ASCII- oder Binärdatei.
Flag für den wahlfreien Zugriff	Gibt an, ob der Zugriff auf die Datei sequentiell oder wahlfrei erfolgen kann.
Temporär-Flag	Signalisiert, ob die Datei bei Prozessende gelöscht werden soll.
Sperre-Flag	Markiert die Datei als gesperrt.
Erzeugungsdatum	Datum und Uhrzeit der Dateierzeugung.
Zeit des letzten Zugriffs	Datum und Uhrzeit des letzten Zugriffs auf die Datei.
Zeit der letzten Änderung	Datum und Uhrzeit der letzten Änderung des Dateiinhalts.
Aktuelle Grösse	Grösse der Datei in Bytes.
Maximalgrösse	Grösse, bis zu der die Datei maximal wachsen darf.

Tabelle 2.1.: Eine Auswahl häufig verwendeter Dateiattribute

Verzeichnis 'bin' befindet, das im Verzeichnis 'usr' liegt, welches seinerseits im Wurzelverzeichnis gespeichert ist.).

Dateien können vom Benutzer in beliebigen Verzeichnissen abgelegt werden. Die Benennung der Dateien und der Verzeichnisse ist ebenfalls dem Benutzer überlassen. Das bedeutet, dass der Benutzer für die *logische* Organisation seiner Dateien verantwortlich ist. Der Benutzer gewinnt dadurch grösstmögliche Freiheit in der Strukturierung des Verzeichnisbaums und in der Ablage und Benennung seiner Dateien, muss sich aber selbst um die Systematik seiner Ablage kümmern. Die Position einer Datei im Verzeichnisbaum

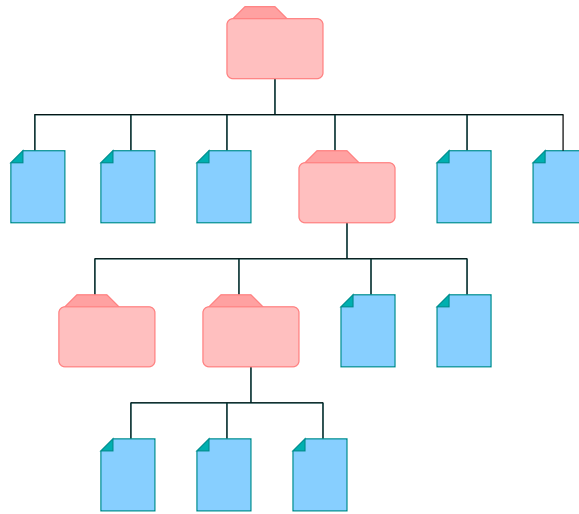


Abbildung 2.1.: Hierarchisches Dateisystem

liefert keine vom System definierte semantische Information über die Datei. In der Praxis kann man zwar davon ausgehen, dass der Benutzer seine Dateien nach gewissen einfachen semantischen Kriterien im Verzeichnisbaum anordnet (z.B. Bildserien thematisch sortiert in eigenen Verzeichnissen oder Textdokumente nach Briefen, Rechnungen usw. auf Verzeichnisse aufgeteilt). Aus dieser Annahme heraus lassen sich allerdings für den Benutzer keine zusätzlichen semantischen Informationen über die Dateien gewinnen².

2.4. Zusammenfassung

Ein klassisches Dateisystem bietet keine Unterstützung für die Extraktion von semantischen Informationen aus Dateidaten. Das Dateisystem speichert Dateidaten als einen unstrukturierten Bytestrom, ohne diesen interpretieren zu können. Die Semantik wird dem Bytestrom erst von der Erzeugerapplikation auferlegt.

Neben Dateidaten speichern Dateisysteme eine kleine Menge von Dateiattributen, welche jedoch keine inhaltsbezogenen Merkmale der Dateien festhalten. Die Dateiattribute dienen hauptsächlich systeminternen Verwaltungszwecken.

Die am weitesten verbreitete Form des Dateisystems ist das hierarchische Dateisystem, welches über Verzeichnisse baumartig strukturiert ist. Der Zugriff auf Dateien erfolgt bei hierarchischen Dateisystemen über einen Pfad im Verzeichnisbaum, dessen Struktur für Systemdateien durch das Betriebssystem und für Benutzerdateien durch

²Der Benutzer gewinnt keine Information, die er nicht schon selbst aus der Position der Datei im Verzeichnisbaum entnehmen kann.

den Benutzer bestimmt ist. Die vom Benutzer willkürlich wählbare Struktur des Verzeichnisbaums, welcher seine Dateien aufnimmt, lässt keine nichttrivialen semantischen Rückschlüsse über diese Dateien zu.

3. Semantische Dateisysteme

Semantische Dateisysteme übernehmen die Aufgaben eines klassischen Dateisystems, indem sie Dateien verwalten und den Zugriff auf Dateien ermöglichen und regeln. Sie bieten zusätzlich semantische Informationen zu den verwalteten Dateien und verwenden diese, um die Möglichkeiten des Dateizugriffs zu erweitern.

3.1. Dateisemantik

Semantische Informationen über eine Datei beschreiben die Semantik dieser Datei, welche kurz als Dateisemantik bezeichnet wird. Die Dateisemantik umfasst nach [23] folgende Arten von Datei-Informationen:

- *definitional* (z.B. Informationen über den Typ der Datei)
- *associative* (z.B. Schlüsselwörter, die den Dateiinhalt charakterisieren)
- *structural* (z.B. Organisation der Daten, Beziehungen innerhalb von Dateien und zwischen Dateien)
- *behavioral* (z.B. Change-Management)
- *environmental* (z.B. Erschaffer der Datei)

Klassische Dateisysteme bieten unter Umständen über Dateiattribute (siehe Abschnitt 2.2) eine kleine Menge dateisemantischer Informationen. So tragen beispielsweise die Angabe zum Erschaffer der Datei (*environmental* Information) oder das ASCII/Binär-Flag (*definitional* Information) zur Beschreibung der Dateisemantik bei. Die Anzahl der dateisemantischen Informationen ist aber in klassischen Dateisystemen stark beschränkt und ein Zugriff auf Dateien, basierend auf diesen Informationen, nicht möglich.

Ein Semantisches Dateisystem unterscheidet sich von einem klassischen Dateisystem dadurch, dass es umfangreiche semantische Informationen zu Dateien verwaltet und diese für einen erweiterten Dateizugriff zur Verfügung stellt.

3.2. Architekturen Semantischer Dateisysteme

Bei Architekturen Semantischer Dateisysteme unterscheidet man grundsätzlich zwischen zwei Ansätzen [23]:

- integrierter (integrated) Ansatz
- erweiterter (augmented) Ansatz

3.2.1. Integrierter Ansatz

Bei integrierten Ansätzen sind semantische Informationen direkt in ein erweitertes Dateisystem zusammen mit den Dateidaten eingebettet. Durch die Integration der semantischen Informationen ergibt sich eine implizite Synchronisation zwischen den Dateidaten und den dazugehörigen semantischen Informationen: Das System kann die Konsistenz der semantischen Informationen gewährleisten, da es selbst die Dateidaten verwaltet und daher immer ein aktuelles Bild von ihnen besitzt.

Semantische Dateisysteme nach dem integrierten Ansatz können nach ihrem Grad der Kopplung mit dem eigentlichen Dateisystem und dem Betriebssystem klassifiziert werden. Man unterscheidet die folgenden zwei Arten:

- Schwach gekoppelte Systeme: Sie erweitern ein Dateisystem indem sie darauf aufsetzen, jedoch einen eigenen Mechanismus für die Datenspeicherung implementieren (unter Verwendung des eigentlichen Dateisystems).
- Eng gekoppelte Systeme: Sie sind direkt in das eigentliche Dateisystem eingebaut.

Beispiel SHORE

SHORE (*Scalable Heterogeneous Object REpository*) [1] ist ein an der Universität von Wisconsin entwickeltes Objekt-System, welches Technologien von objektorientierten Datenbanken (OODB) und Dateisystemen vereint.

Das SHORE-System besteht im Wesentlichen aus den folgenden drei Komponenten:

- *Shore Storage Manager* (SM): Verantwortlich für die Speicherung und Verwaltung von Objekten.
- *Shore Value-Added Server* (SVAS): Ermöglicht und regelt den Zugriff auf gespeicherte Objekte.
- *Shore Data Language* (SDL): Eine an ODL (*Object Definition Language*) des ODMG Konsortiums (*Object Data Management Group*) [13] angelehnte Sprache zur Definition von Objekttypen in SHORE.

Benannte Objekte werden in SHORE über einen Unix-ähnlichen Namensraum organisiert. Ein Objekt kann ein Verzeichnis, einen symbolischen Link oder ein beliebiges anderes typisiertes Objekt darstellen (als Gegenstück zu einer normalen Datei in Unix). Der Typ des Objekts ist dabei entweder ein von SHORE definierter eingebauter Typ oder ein benutzerdefinierter Typ.

Der Zugriff auf Objekte erfolgt wie von klassischen Dateisystemen her gewohnt über Verzeichnisse oder über einen für jedes Objekt global eindeutig definierten *Object Identifier* (OID). Zudem bietet SHORE eine Möglichkeit für den typisierten Zugriff auf Objekte.

Anwendungen können auf ihre Daten auch in gewohnter untypisierter Art zugreifen. SHORE implementiert die entsprechenden Unix Dateisystem-Aufrufe, um diese Art von Zugriff über das SHORE-System zu ermöglichen.

Das SHORE-Framework dient als Basis verschiedener Datenbankprojekte (Paradise [15], PREDATOR [17], Lambda-DB [10]) und eignet sich dank seines Objektmodells und der verschiedenen Zugriffsmöglichkeiten auf Objekte als Grundlage für die Implementation eines integrierten (schwach gekoppelten) Semantischen Dateisystems.

3.2.2. Erweiterter Ansatz

Bei einem erweiterten Ansatz liegen die semantischen Informationen in einer Abstraktionsschicht über dem traditionellen Dateisystem, das die Dateidaten speichert. Die Schnittstelle zum ursprünglichen Dateisystem bleibt dabei unverändert. Eine Synchronisation zwischen den Dateidaten und den semantischen Informationen ergibt sich nicht so einfach wie bei einem integrierten Ansatz, da das Semantische Dateisystem in der Regel nicht über Änderungen im darunterliegenden Dateisystem informiert wird.

Es lassen sich zwei Architekturen von Semantischen Dateisystemen nach dem erweiterten Ansatz unterscheiden:

- *Query shipping* Architekturen: Benutzeranfragen, welche an die Abstraktionsschicht des Systems gestellt werden, werden direkt zu den entsprechenden Dateien transportiert, wo sie über einen geeigneten Mechanismus bearbeitet werden.
- *Index shipping* Architekturen: Metadaten werden aus den Dateien extrahiert und in der Abstraktionsschicht des Systems zur Verfügung gestellt. Benutzeranfragen werden in der Abstraktionsschicht behandelt.

Viele Implementationen Semantischer Dateisysteme basieren auf einer Architektur gemäss dem erweiterten Ansatz (SFS [5], TSIMMIS [3], Harvest [6], InfoHarness [20], u.a.). Die Wahl dieser Architektur erlaubt eine unkomplizierte Integration des Systems in bestehende Benutzerumgebungen. Sie stellt weniger Anforderungen an das bestehende Dateisystem und an den Benutzer, als dies eine Architektur nach dem integrierten

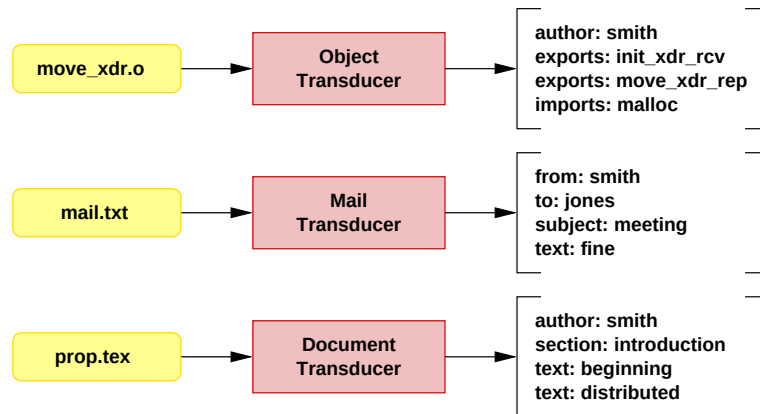


Abbildung 3.1.: Beispiele von Transducern

Ansatz tut. Diese Tatsache macht den erweiterten Ansatz speziell für Prototypen und plattformunabhängige Implementationen interessant.

Beispiel SFS

SFS ist ein am MIT entwickeltes (*index shipping*) Semantisches Dateisystem [5], welches das NFS (Network Filesystem) Protokoll implementiert. Alle Zugriffe auf das Dateisystem geschehen über gewöhnliche NFS-Anfragen. Diese Art der Implementation bietet folgende zwei Vorteile:

- Der Benutzer kann mit einem beliebigen NFS-Client auf das Dateisystem zugreifen. Der Einsatz des SFS bleibt vor dem Benutzer verborgen, er kann wie gewohnt mit dem Dateisystem interagieren.
- SFS wird über das NFS-Protokoll direkt über alle Änderungen am Dateisystem informiert und kann so die semantischen Informationen zu den Dateidaten konsistent halten.

Mittels sogenannten *Transducern* (siehe Abbildung 3.1) generiert SFS zu den im System gespeicherten Dateien Attribute. Ein Attribut ist ein Name-Wert Paar, das eine Eigenschaft der Datei beschreibt (z.B. ein Attribut mit Name **author** und einem Wert **smith**, das den Namen des Autors dieser Datei angibt). Für verschiedene Dateitypen werden verschiedene Transducer programmiert, welche aus dem Dateiinhalt die dateispezifischen Attribute extrahieren. Abbildung 3.1 zeigt Beispiele von Transducern für Objektdateien, Maildateien und T_EX-Dateien, sowie deren Ausgaben für eine bestimmte Eingabedatei.

SFS unterstützt *virtuelle Verzeichnisse*. Ein virtuelles Verzeichnis hat den Namen eines Attributes und enthält alle Dateien, für die ein solches Attribut extrahiert wurde.

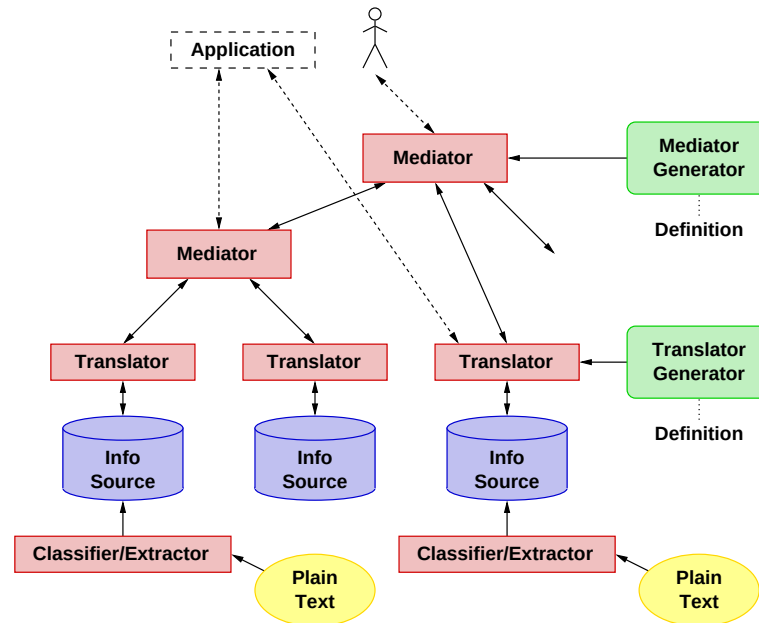


Abbildung 3.2.: Architektur von TSIMMIS

Der Inhalt von virtuellen Verzeichnissen kann durch Angabe eines Attribut-Wertes auf Dateien eingeschränkt werden, für die ein Attribut mit angegebenem Wert extrahiert wurde. Virtuelle Verzeichnisse verhalten sich für den Benutzer wie normale Verzeichnisse. Idealerweise merkt der Benutzer nicht, dass der Inhalt dieser Verzeichnisse zur Zeit der Anfrage generiert wurde.

Beispiel TSIMMIS

TSIMMIS [3] ist ein in Stanford in Zusammenarbeit mit IBM entwickeltes System. Ziel von TSIMMIS ist es, integrierten Zugriff auf verschiedene, heterogene Informationsquellen anzubieten. TSIMMIS kann als Beispiel eines (*query shipping*) Semantischen Dateisystems angesehen werden.

Abbildung 3.2 zeigt die Architektur von TSIMMIS, welche im folgenden beschrieben wird.

TSIMMIS definiert ein Objektmodell (*Object Exchange Model* oder *OEM*) für die Darstellung von Metadaten. Ein *Translator* ist für die Verarbeitung von Anfragen und für die Konversion der Daten-Objekte aus einer Informationsquelle in das gemeinsame Objektmodell verantwortlich. Eine Informationsquelle kann dabei ein Dateisystem, eine Datenbank, eine digitale Bibliothek, ein Information-Retrieval System usw. repräsentieren. Besteht eine Informationsquelle aus unstrukturierten Informationen, so werden

diese Informationen über einen *Classifier* klassifiziert und wichtige Attribute über einen *Extractor* extrahiert. Die so gewonnenen Attribute sind dann über den Translator für diese Informationsquelle zugänglich. Auf ein Dateisystem als Informationsquelle übertreten würde dies bedeuten, dass ein Classifier den Typ einer Datei bestimmt (z.B. eine E-Mail Nachricht) und ein Extractor dann die für diesen Typ relevanten Attribute (*Sender*, *Empfänger*, *Datum*, *Betreff*, *Inhalt* usw.) extrahiert und verfügbar macht.

Eine Applikation oder ein Benutzer richtet Anfragen an das System in einer eigens für OEM entwickelten Sprache: *OEM-QL*. OEM-QL Anfragen liefern OEM-Objekte als Resultat. Die Anfragen können direkt an einen Translator oder einen sogenannten *Mediator* gestellt werden. Ein Mediator ist ein Modul, das die Informationen aus einer oder mehreren Informationsquellen verfeinert. Er ist in der Regel darauf spezialisiert gewisse Typen von Informationen (z.B. aktuelle Ereignisse) zu verarbeiten und passt Anfragen und Resultate geeignet an.

Da ein Translator und ein Mediator beide OEM-QL Anfragen akzeptieren, können Mediatoren beliebig auf anderen Mediatoren oder Translatoren aufsetzen.

Um die Benützung und die Anpassung des Systems zu vereinfachen, unterstützt TSIMMIS das automatische Erzeugen von Translatoren und Mediatoren über entsprechende *Generatoren*, welche eine einfache Definitionssprache verarbeiten.

3.3. Aktuelle Entwicklungen und Systeme

3.3.1. Semantic Web

Eine Arbeitsgruppe des W3C um Tim Berners-Lee beschäftigt sich mit der Entwicklung eines *Semantic Web* [24]. Semantic Web ist als Erweiterung des bestehenden Internets zu verstehen, mit dem Ziel, einen Standard für die Beschreibung von Daten im Internet aufzustellen. Es soll eine standardisierte Darstellung für semantische Informationen über Webinhalte entstehen, welche eine bessere Zusammenarbeit zwischen Menschen und Maschinen bei der Verarbeitung und beim Zugriff auf diese Inhalte ermöglicht.

Die dem Semantic Web zu Grunde liegende Technologie heisst RDF (*Resource Description Framework*) [24] und lässt sich in XML (*Extended Markup Language*) [24] darstellen. Die Möglichkeit zur Darstellung in XML erlaubt einen einfachen Austausch von RDF-Angaben.

RDF ist eine Sprache für die Beschreibung von Ressourcen¹ und eignet sich speziell für die Repräsentation von Metadaten über Web-Ressourcen wie zum Beispiel *Titel*, *Autor* oder *Änderungsdatum* einer Webseite oder eines Webdokuments.

Eine RDF-Beschreibung lässt Aussagen der Art `http://www.example.org/index.html has a Creator whose value is John Smith` zu und besteht aus drei Teilen:

¹Jedes Objekt, das sich über eine URI (*Uniform Resource Identifier*) adressieren lässt, ist eine Resource.



Abbildung 3.3.: RDF-Beschreibung als gerichteter beschrifteter Graph

- Ein Subjekt (`http://www.example.org/index.html`). Das Subjekt ist die Ressource, die beschrieben wird. Die Ressource wird über eine URI (*Uniform Resource Identifier*) adressiert.
- Ein Prädikat (`Creator`). Das Prädikat charakterisiert die Beschreibung und definiert deren Bedeutung.
- Ein Objekt (`John Smith`). Das Objekt ist ein Wert für den Inhalt der Beschreibung.

Abbildung 3.3 zeigt das obige Beispiel einer RDF-Beschreibung in einer graphischen Darstellung als gerichteter und beschrifteter Graph.

Die XML Darstellung des Beispiels sieht wie folgt aus:

```

<rdf:RDF>
  <rdf:Description about='http://www.example.org/index.html'>
    <s:Creator>John Smith</s:Creator>
  </rdf:Description>
</rdf:RDF>
  
```

Der Namensraum für die in RDF-Beschreibungen verwendeten Elemente (wie `RDF`, `Description` usw.) ist mit `rdf` bezeichnet. Die Angabe `s` ist ein Verweis auf den Namensraum, in dem das Element `Creator` über ein RDF-Schema definiert ist.

Ein RDF-Schema [24] dient dazu, ein Vokabular für RDF-Aussagen festzulegen. Über den Verweis auf ein RDF-Schema und über die Verwendung der dort spezifizierten Elemente als Prädikate, wird die Bedeutung einer RDF-Aussage festgelegt.

Es sind Bestrebungen im Gange, Mengen von Elementen für die Beschreibung von Ressourcen zu standardisieren. Ein Beispiel für eine solche Bestrebung stellt die *Dublin Core Metadata Initiative* [2] dar, welche Elemente für die Beschreibung von Dokumenten definiert. Die dort definierten Elemente (*Title*, *Creator*, *Subject*, *Description*, *Publisher*, u.a.) beschreiben Ressourcen in einer Art, wie man sie beispielsweise von Bibliothekskatalogen her kennt.

3.3.2. Microsoft Universal Data Access

UDA (*Universal Data Access*) [22] bezeichnet Microsoft's Strategie für den einheitlichen Zugriff auf Daten aus verschiedenen Datenquellen wie zum Beispiel Datenbanken oder Dateisystemen. Abbildung 3.4 zeigt die Architektur von Microsoft UDA.

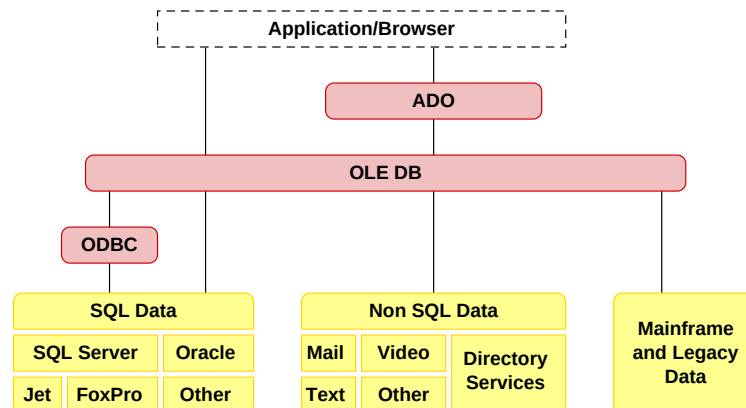


Abbildung 3.4.: Microsoft UDA-Architektur

UDA basiert auf OLE DB, das eine Menge von (COM-) Schnittstellen für den Datenzugriff definiert. Applikationen können diese Schnittstellen direkt oder über ADO (*ActiveX Data Objects*) verwenden. Eine Datenquelle, welche Daten über UDA zur Verfügung stellen möchte, implementiert die OLE DB Schnittstellen. So gibt es beispielsweise eine ODBC (*Open Database Connectivity*) Implementation, welche Daten in ODBC-Datenbanken über OLE DB verfügbar macht.

Die Definition von gemeinsamen Schnittstellen für den Zugriff auf Daten (bzw. Metadaten) gibt den Benutzern dieser Schnittstellen eine gemeinsame Basis: Die Operationen und die Resultate haben eine global definierte Semantik.

3.4. Zusammenfassung und Diskussion

Die in diesem Kapitel beschriebenen Ansätze und Implementationen von Semantischen (Datei-) Systemen verwenden alle ein internes Modell für die Repräsentation von semantischen Informationen über Objekte (bzw. Dateidaten). Das Modell ist die Grundlage für die Speicherung von Metadaten über diese Objekte und bildet die Abstraktion für den Zugriff auf Informationen.

Da das Modell für die Metadaten eine zentrale Rolle in einem Semantischen (Datei-) System einnimmt, ist ein gutes Design dieses Modells von grosser Bedeutung für die Mächtigkeit und die Flexibilität des Systems. Die meisten der in diesem Kapitel vorgestellten Systeme verwenden ein Objektmodell für die Darstellung semantischer Informationen, das beliebig strukturierte Beschreibungen von Daten zulässt (z.B. SDL in SHORE, OEM in TSIMMIS oder RDF-Schema im Semantic Web). Das in Abschnitt 3.2.2 beschriebene SFS bildet dabei eine Ausnahme, indem es nur Name-Wert

Paare als Metadaten erlaubt, was die Möglichkeiten zur Beschreibung von komplizierten Objekten einschränkt.

Die hier vorgestellten Semantischen Dateisysteme haben zumeist den Nachteil, dass sie auf bestimmte Umgebungen abgestimmt sind und nur mit gewissen Dateitypen zurecht kommen. Die Anpassung an neue Situationen ist aufwendig und erfordert in der Regel einen Eingriff in das System. So muss in SFS beispielsweise für jeden neuen Dateityp ein Transducer programmiert werden und in TSIMMIS müssen Classifier und Extractor angepasst werden. (Semantic Web und Microsoft UDA umgehen das Problem, indem sie sich nicht um das Generieren von Metadaten kümmern. Semantic Web definiert nur das Modell und das Format der Metadaten, während UDA eine Menge von Schnittstellen für den Zugriff auf Metadaten spezifiziert.)

Ein weiterer Mangel der besprochenen Systeme ist die fehlende Möglichkeit zur Integration von bereits vorhandenen oder vom Benutzer spezifizierten Metadaten. SFS und TSIMMIS verlassen sich ganz auf die Resultate von Transducern bzw. Classifier und Extractor, welche unter Umständen falsche Informationen liefern oder nicht in der Lage sind, die gewünschten Informationen zu extrahieren.

4. ObjectSFS

OBJECTSFS ist ein Prototyp eines objektorientierten, konfigurierbaren Semantischen Dateisystems. Die Architektur von OBJECTSFS basiert auf dem in Abschnitt 3.2 beschriebenen erweiterten Ansatz, was eine einfache Integration in bestehende Systeme ermöglicht. Über die Wahl des erweiterten Ansatzes und der Implementationssprache Java wird eine weitgehende Plattformunabhängigkeit von OBJECTSFS erreicht.

OBJECTSFS verarbeitet Dateien zu strukturierten Metadaten-Objekten (ähnlich einem Verbund oder einer Struktur wie sie z.B. in der Programmiersprache C als *struct* vorkommt). Die Verarbeitung wird durch eine Konfiguration gesteuert und umfasst die folgenden zwei Schritte (siehe Abschnitt 4.5):

1. Klassifikation der Datei.
2. Extraktion eines Metadaten-Objekts für die Datei.

Das Extrahieren von Metadaten qualifiziert OBJECTSFS als ein *index shipping* System. Objekte werden in einer Abstraktionsschicht gemäss einem einfachen Objektmodell für Anfragen verfügbar gemacht.

Die folgenden Abschnitte beschreiben das Objektmodell (Typen und Objekte) von OBJECTSFS, die Repräsentation von Dateien und die Konfiguration des Systems. Das Objektmodell bildet die Grundlage der danach vorgestellten Verarbeitung von Dateien zu Fileobjekten. Eine Vertrautheit mit dem Modell ist daher eine Voraussetzung für das Verständnis des Prinzips der Verarbeitung in OBJECTSFS.

4.1. Typen

Typen stellen das zentrale Element des Objektmodells von OBJECTSFS dar. Sie bestimmen die Struktur von Objekten und bilden die Basis der Extraktion, der Speicherung und der Abfrage von Objekten.

4.1.1. Typsystem

Abbildung 4.1 zeigt ein Diagramm der Klassen rund um das Typsystem von OBJECTSFS.

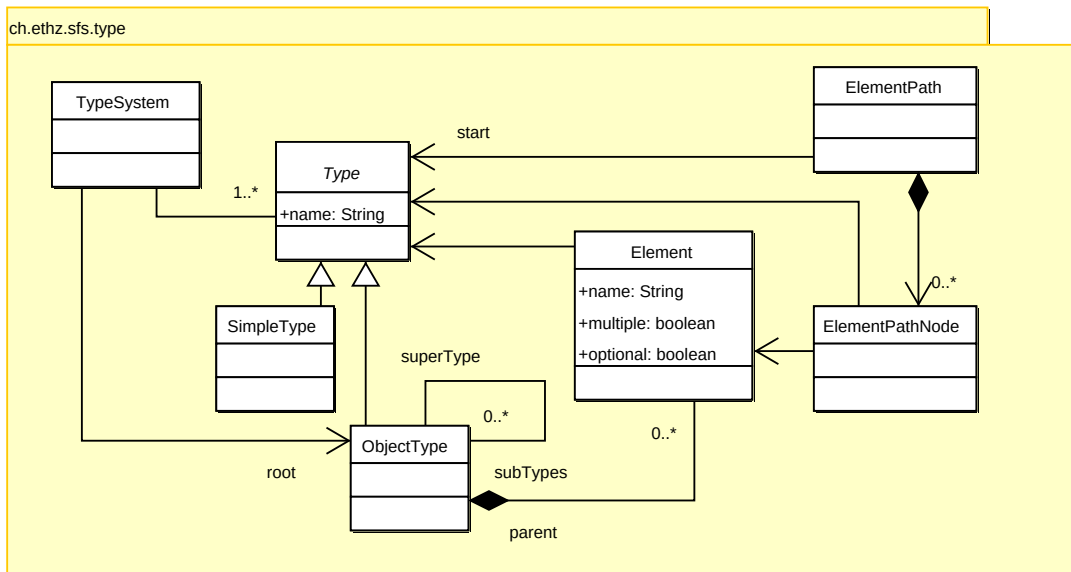


Abbildung 4.1.: Klassendiagramm zum Typsystem von OBJECTSFS

Die Konzepte des OBJECTSFS-Typsens sind an Konzepte aus der XML-Schema Spezifikation [24] angelehnt. Der Zusammenhang wird in der Beschreibung der einzelnen Konzepte aufgezeigt.

Typ

Ein Typ (*Type*) ist entweder ein eingebauter einfacher Typ (*SimpleType*) oder ein komplexer Typ (*ObjectType*). Einfache Typen und komplexe Typen sind von ihrer Funktion und ihrer Struktur her zu vergleichen mit den XML-Schema Typen *simpleType* und *complexType* [24].

Jeder Typ hat einen Namen, welcher ihn innerhalb des Typsystems eindeutig kennzeichnet. Eine spezielle Klasse (*TypeSystem*) verwaltet alle im System definierten Typen und ermöglicht den Zugriff auf diese.

Einfacher Typ

Einfache Typen (*SimpleType*) sind in OBJECTSFS vordefiniert. Beispiele hierfür sind **string** für eine Zeichenkette, **integer** für eine Ganzzahl oder **date** für ein Datum.

Die einfachen Typen in OBJECTSFS sind kompatibel zu den gleichnamigen *simpleTypes* in XML-Schema. OBJECTSFS definiert nur einige wenige einfache Typen und lässt keine Ableitung dieser Typen zu. Für das Erstellen von Metadaten-Objekten und

die Beschreibung von Dateien und Datei-Objekten reichen die in OBJECTSFS definierten einfachen Typen (`string`, `integer`, `date`) allerdings aus.

Objekttyp

Ein komplexer Typ, kurz Objekttyp (*ObjectType*), setzt sich aus verschiedenen Elementen zusammen. Er ist vergleichbar mit einem strukturierten Typ wie er beispielsweise in der XML-Schema Spezifikation als *complexType* oder in der Programmiersprache C als *struct* anzutreffen ist.

Jeder Objekttyp erweitert einen anderen Objekttyp¹. Das Typsystem definiert einen Basistyp (`Root`), der keinen anderen Objekttyp erweitert und von dem jeder ObjektTyp direkt oder indirekt abgeleitet ist. Der `Root`-Objekttyp ist über die spezielle Klasse *TypeSystem* zugänglich.

Abbildung 4.2 zeigt ein Beispiel eines Typsystems. Die Objekttypen, ihre Elemente, sowie die Vererbungshierarchie ist grafisch dargestellt.

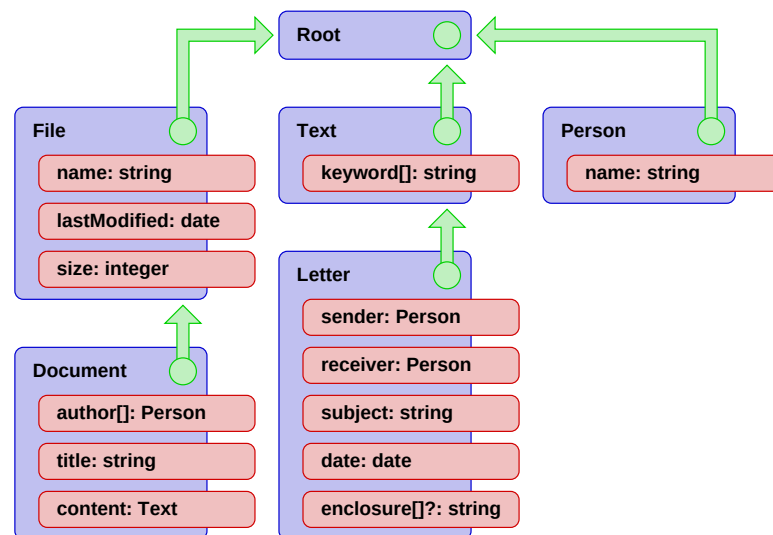


Abbildung 4.2.: Beispiel für Objekttypen in einem Typsystem

`Root` ist der Basistyp des Typsystems aus Abbildung 4.2. Er wird erweitert durch die Objekttypen `File`, `Text` und `Person`. `File` steht für eine Datei und definiert Elemente für die gebräuchlichsten Dateiattribute (`name`, `lastModified` und `size`). `Document` ist eine Erweiterung des Objekttyps `File` und verkörpert Dateien, deren Inhalt aus einem formatierten Text (z.B. eine PDF- oder eine RTF-Datei) besteht. In `Document` sind einige zusätzliche Elemente definiert (zum Beispiel `title` für den Titel des Dokuments).

¹Es gibt keine *multiple inheritance*.

Text ist für die Beschreibung von unformatierten Text gedacht und definiert dazu ein **keyword** Element. Das **content** Element von **Document** ist ein Beispiel eines Elements mit Typ **Text**, welches für die Beschreibung von Dokumenten verwendet wird. **Text** wird durch den Objekttyp **Letter** erweitert, welcher für die Beschreibung von Text mit einem speziellen Aufbau verwendet wird, wie er für Briefe üblich ist. In **Letter** sind dazu die Elemente zur Charakterisierung von Briefen (**sender**, **receiver** usw.) definiert.

Element

Ein Element hat einen Namen und einen bestimmten Typ. Jedes Element ist für einen Objekttyp (*parent*) definiert und ist vergleichbar mit einem *element*, wie es in XML-Schema für einen *complexType* definiert wird.

Das einzige für den in Abbildung 4.2 gezeigten Objekttyp **Person** definierte Element heisst **name** und hat den einfachen Typ **string**. Für **Letter** ist ein Element namens **sender** definiert, das den Objekttyp **Person** hat. Der *parent*-Objekttyp des **title** Elements ist **Document**.

Ein Element kann mehrfach in einem Objekttyp vorkommen (genauer als Feld in einer Instanz eines Objekttyps), die Möglichkeit dazu wird mit einem Flag (*multiple*) markiert. Wir nennen ein solches Element ein *multiple* Element. Das Element **keyword** des in Abbildung 4.2 dargestellten Objekttyps **Text** ist ein Beispiel eines multiplen Elements. Die eckigen Array-Klammern ([]) nach dem Namen des Elements stellen diesen Sachverhalt dar.

Ein anderes Flag (*optional*) besagt, dass das Element wahlweise im Objekttyp vorkommen kann oder auch nicht (auch hier bezieht sich das optional präziserweise auf das entsprechende Feld einer Instanz des Objekttyps). Ein solches Element nennen wir ein *optionales* Element. Das Element **enclosure** aus Abbildung 4.2 ist ein optionales Element des Objekttyps **Letter**. Dieser Umstand wird durch ein Fragezeichen hinter dem Element-Namen dargestellt. Gleichzeitig ist **enclosure** auch ein multiples Element, was durch die ebenfalls vorhandenen eckigen Klammern verdeutlicht wird.

Vererbung beeinflusst die Menge der Elemente von Objekttypen: Abgeleitete Objekttypen erben die Elemente des Supertyps, den sie erweitern. So erbt der Objekttyp **Document** aus Abbildung 4.2 beispielsweise die Elemente **name**, **lastModified** und **size** des Objekttyps **File**, während das multiple Element **keyword** des Objekttyps **Text** auf **Letter** vererbt wird.

Elementpfad

Ein Elementpfad (*ElementPath*) adressiert ein Element (bzw. ein Unterelement) über einen Starttyp (*start*) und eine Kette von Elementpfad-Knoten (*ElementPathNode*). Der Starttyp markiert den Startpunkt des Pfades. Ein Elementpfad-Knoten steht im wesentlichen für ein Element und ist mit einem Typ (*Type*) verknüpft. Für die in diesem

Abschnitt folgende Beschreibung können wir annehmen, dass der Typ des Elementpfad-Knotens dem Typ des enthaltenen Elements entspricht².

Jedes Element eines Elementpfad-Knotens muss ein gültiges Element des Objekttyps des vorhergehenden Elementpfad-Knotens sein (oder des Starttyps, falls es sich um den ersten Elementpfad-Knoten im Pfad handelt). Der Typ eines Knotens, der als Vorgänger eines anderen Knotens im Elementpfad auftritt, darf also nie ein einfacher Typ (*SimpleType*) sein (dasselbe gilt für den Starttyp).

Die Schreibweise für Elementpfade richtet sich nach dem folgenden Schema: $T/e_1 : t_1/\dots/e_n : t_n$, wobei T für den Starttyp und e_i bzw. t_i für das Element bzw. den Typ des i -ten Elementpfad-Knotens eines Elementpfades der Länge n stehen. Gilt $n > 0$ so müssen alle t_j mit $j < n$ als auch T für einen Objekttyp stehen. Weiter muss e_k ein Element von t_{k-1} für $2 < k < n$ und e_1 ein Element von T sein.

Gültige Elementpfade im Zusammenhang mit den in Abbildung 4.2 dargestellten Objekttypen sind:

- `Document/title:string`: Adressiert das Element `title` des Objekttyps `Document`.
- `Document/author:Person/name:string`: Adressiert ein Unterelement des Objekttyps `Document`.
- `Letter/keyword:string`: Adressiert das multiple Element `keyword`, das der Objekttyp `Letter` von `Text` geerbt hat.
- `Root`: Leerer Pfad mit Starttyp `Root`. Ein Elementpfad, dessen Starttyp keine Elemente definiert, kann keinen Elementpfad-Knoten aufnehmen.

4.1.2. Typdefinitionen in XML-Schema

Mit den vorgestellten Konzepten von Typen (*SimpleType* und *ObjectType*) und Elementen lassen sich Definitionen für Objekttypen vornehmen. Für die Notation von Typdefinitionen kann eine Untermenge der XML-Schema Notation [24] verwendet werden. Die Verwendung einer XML-basierten Darstellung erlaubt eine unkomplizierte Serialisierung und den einfachen Austausch von Typdefinitionen.

Die Abbildung von OBJECTSFS Typsystem Konzepten auf eine Notation gemäss XML-Schema Spezifikation ist in Tabelle 4.1 unter Verwendung einer an EBNF (*Erweiterte Backus-Naur-Form*) angelehnten Schreibweise gezeigt³.

²In Abschnitt 4.2 wird die Bedeutung des Typs des Elementpfad-Knotens in Zusammenhang mit der Adressierung von Objektfeldern und der Unterstützung von Polymorphismus erklärt

³Der Übersichtlichkeit halber wird in den hier gezeigten XML-Schema Notationen jeweils auf die Angabe des Schema Elements (`<schema>`) und des Namensraums verzichtet. Die verwendeten XML-Elemente für die Typdefinitionen stammen aus dem XML-Schema Namensraum (<http://www.w3.org/2001/XMLSchema>).

<i>ObjectType</i>	<code><complexType name=' <i>ObjectTypeId</i> '> <extension base=' <i>ObjectTypeId</i> '> [<i>ElementSequence</i>] </extension> </complexType></code>
<i>ElementSequence</i>	<code><sequence> (<i>Element</i>)* </sequence></code>
<i>Element</i>	<code><element name=' <i>ElementId</i> ' type=' <i>TypeId</i> ', [<i>ElementOptional</i>] [<i>ElementMultiple</i>] /></code>
<i>ElementId</i>	<i>Identifier</i>
<i>ElementOptional</i>	<code>minOccurs='0'</code>
<i>ElementMultiple</i>	<code>maxOccurs='unbounded'</code>
<i>TypeId</i>	<code>(<i>SimpleTypeId</i> <i>ObjectTypeId</i>)</code>
<i>SimpleTypeId</i>	<code>(string integer date)</code>
<i>ObjectTypeId</i>	<i>Identifier</i>

Tabelle 4.1.: Schema für die Notation von Typdefinitionen

Ein *Identifier* stellt einen Namen für einen Objekttyp oder für ein Element dar. Elementnamen werden üblicherweise klein, die Namen von Objekttypen gross geschrieben. Als Name eines Objekttyps darf kein Name eines vordefinierten einfachen Typs (`string`, `integer`, `date`) verwendet werden.

Multiple Elemente werden in der vorgestellten XML-Schema Notation über das Attribut `maxOccurs='unbounded'` gekennzeichnet. Standardmässig besitzt dieses Attribut gemäss XML-Schema Spezifikation den Wert 1. Analog dazu werden optionale Element mit dem Attribut `minOccurs='0'` versehen. Auch dieses Attribut besitzt standardmässig den Wert 1.

Tabelle 4.2 zeigt die in Abbildung 4.2 vorgestellten Objekttypen in ihrer XML Darstellung gemäss der in Tabelle 4.1 eingeführten Notation⁴. Die Darstellung ist kompatibel mit den in XML-Schema spezifizierten Typdefinitionen.

4.2. Objekte

Ein Objekt in OBJECTSFS ist ein abstraktes Abbild einer Datei oder eines Datei-Objekts (eines Teils der Datei). Ein OBJECTSFS-Objekt kapselt die semantischen Informationen zu einer Datei.

Die Struktur eines Objekts wird durch einen Objekttyp bestimmt. Die Assoziation mit einem Objekttyp ermöglicht den typbasierten Zugriff auf Objekte.

⁴Für leere Elemente wird hier die übliche verkürzte Schreibweise verwendet: `<xyz/>` statt `<xyz></xyz>`.

<code><complexType name='Root'/></code>
<code><complexType name='File'><extension base='Root'><sequence></code> <code><element name='name' type='string' /></code> <code><element name='lastModified' type='date' /></code> <code><element name='size' type='integer' /></code> <code></sequence></extension></complexType></code>
<code><complexType name='Text'><extension base='Root'><sequence></code> <code><element name='keyword' type='string' maxOccurs='unbounded' /></code> <code></sequence></extension></complexType></code>
<code><complexType name='Person'><extension base='Root'><sequence></code> <code><element name='name' type='string' /></code> <code></sequence></extension></complexType></code>
<code><complexType name='Document'><extension base='File'><sequence></code> <code><element name='author' type='Person' maxOccurs='unbounded' /></code> <code><element name='title' type='string' /></code> <code><element name='content' type='Text' /></code> <code></sequence></extension></complexType></code>
<code><complexType name='Letter'><extension base='Text'><sequence></code> <code><element name='sender' type='Person' /></code> <code><element name='receiver' type='Person' /></code> <code><element name='subject' type='string' /></code> <code><element name='date' type='date' /></code> <code><element name='enclosure' type='string'</code> <code>minOccurs='0' maxOccurs='unbounded' /></code> <code></sequence></extension></complexType></code>

Tabelle 4.2.: Objekttypen in XML Notation

4.2.1. Objektsystem

Abbildung 4.3 zeigt ein erweitertes Klassendiagramm, welches das Konzept der Objekte zusammen mit dem Typsystem von OBJECTSFS darstellt.

Fileobjekt

Ein Fileobjekt (*FileObject*) beschreibt eine Entität, welche eine ganze Datei umfasst oder im Inhalt einer Datei vorkommt (Beispiele für Fileobjekte sind eine Person, ein Dokument, ein Brief, eine E-Mail Datei oder eine E-Mail Nachricht).

Jedes Fileobjekt ist eine *Instanz* eines bestimmten Objekttyps, welcher die Struktur des Objektes festlegt. Wir nennen diesen Objekttyp die *Klasse* des Fileobjekts. Ein

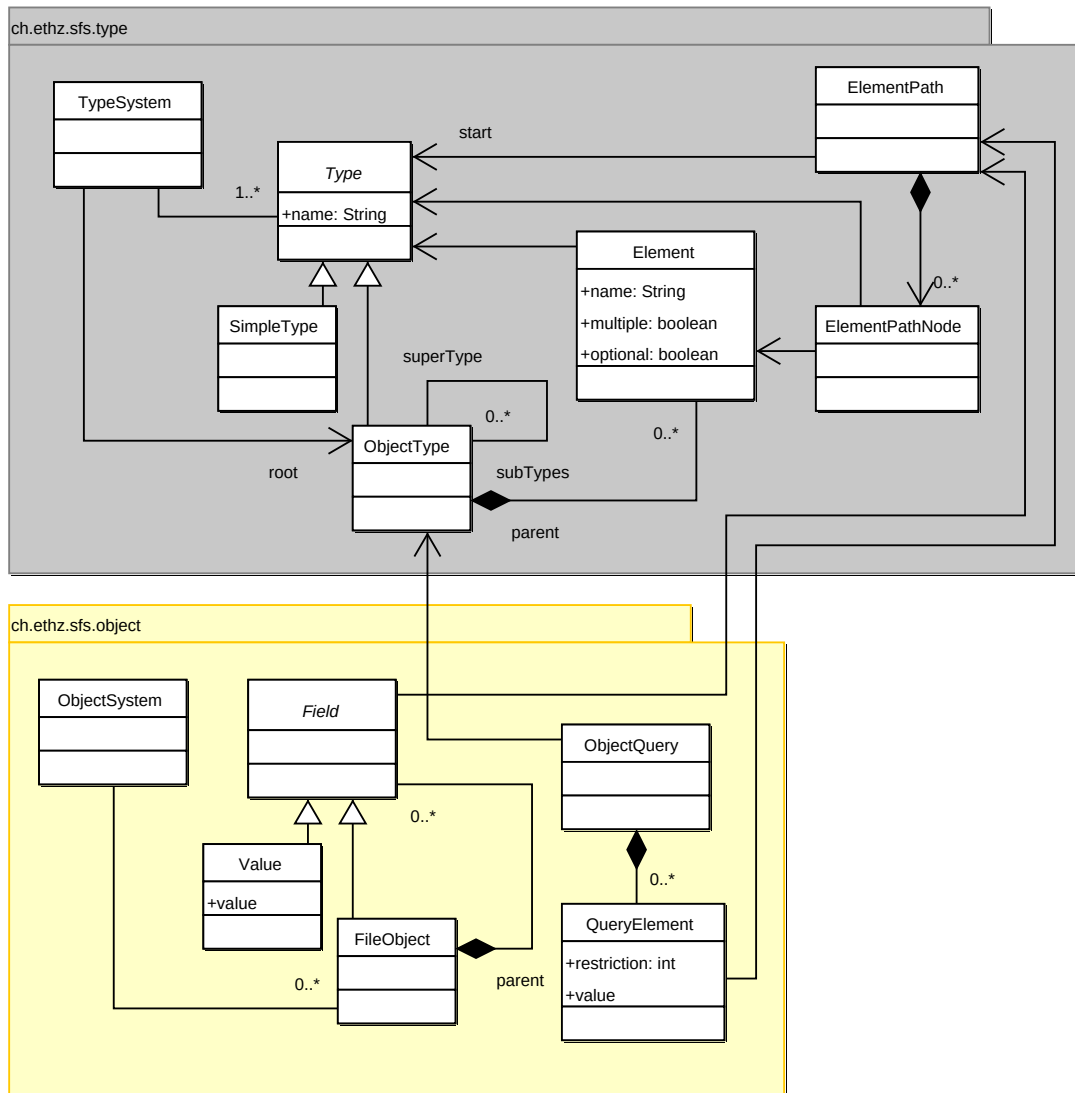


Abbildung 4.3.: Klassendiagramm zum Objektsystem von OBJECTSFS

Fileobjekt ist implizit mit einem Objekttyp über einen Elementpfad verknüpft. Der Mechanismus dazu ist im nächsten Abschnitt beschrieben.

Der Zugriff auf Fileobjekte geschieht über eine spezielle Klasse (*ObjectSystem*), welche alle im System vorhandenen Fileobjekte kapselt.

Abbildung 4.4 zeigt eine grafische Darstellung eines Fileobjekts. Die Typdefinitionen zu dem gezeigten Beispiel sind in Abbildung 4.2 dargestellt. Die Beziehungen zwischen

Feldern des Fileobjekts und Elementen von Objekttypen werden im folgenden Abschnitt diskutiert.

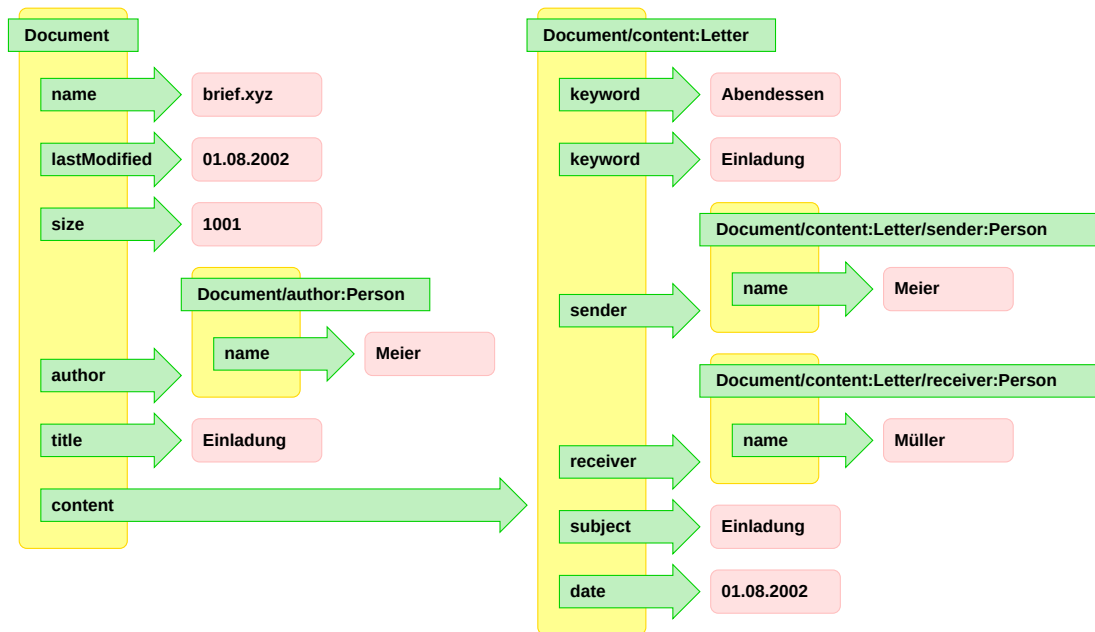


Abbildung 4.4.: Grafische Darstellung eines Fileobjekts

Feld

Ein Fileobjekt besteht aus Feldern (*Field*). Die Klasse des Fileobjekts bestimmt die Art und die Anzahl der Felder eines Fileobjekts: Das Fileobjekt besitzt ein Feld für jedes Element der Klasse, für multiple Elemente können mehrere, für optionale Elemente müssen keine Felder existieren (hängt von der Instanz ab). Die Klasse des in Abbildung 4.4 dargestellte Fileobjekts ist **Document**. Das Fileobjekt besitzt je ein Feld für jedes in **Document** definierte oder auf **Document** vererbte Element (**name**, **lastModified**, **size**, **author** usw.). Abbildung 4.4 zeigt zusätzlich ein Fileobjekt (als Feld des oben beschriebenen Fileobjekts) der Klasse **Letter**. Dieses Fileobjekt besitzt mehrere Felder für das multiple Element **keyword** und kein Feld für das optionale Element **enclosure**.

Für die Adressierung von Feldern werden Elementpfade verwendet. Jedes Feld ist mit einem Elementpfad verknüpft und damit auch implizit mit einem Element (falls der Elementpfad nicht leer ist) und mit einem Typ. Das mit dem Feld assoziierte Element wird durch den letzten Knoten im Pfad bestimmt. Der Typ des Feldes ergibt sich aus dem Typ dieses Knotens bzw. aus dem Starttyp des Elementpfades, falls der Pfad leer ist. Handelt es sich beim Typ des Feldes um einen einfachen Typ, so steht das Feld für

einen Wert (*Value*) des entsprechenden Typs. Ist der Typ des Feldes aber ein Objekttyp, so steht dieses Feld für ein weiteres Fileobjekt⁵. Wir nennen dieses Feld ein *Subobjekt* des ursprünglichen Fileobjekts.

Der Elementpfad `Document/author:Person` adressiert für das in Abbildung 4.4 gezeigte Fileobjekt ein Feld, das seinerseits ein Fileobjekt der Klasse `Person` darstellt. Das Feld (Subobjekt) ist mit dem `author` Element des Objekttyps `Document` assoziiert. Über den Pfad `Document/author:Person/name:string` wird ein Feld – bzw. ein Wert (`Meier`) – adressiert, das mit dem `name` Element des Objekttyps `Person` verknüpft ist und den Typ `string` hat.

Das Fileobjekt aus Abbildung 4.4 besitzt auch ein Feld bzw. ein Subobjekt für den Inhalt des Dokuments (Element `content` des Objekttyps `Document`). Das `content` Element eines `Document` Objekttyps hat den Typ `Text`. Die Klasse des entsprechenden Feldes im Fileobjekt ist allerdings ein Subtyp von `Text`, nämlich `Letter`. Das Beispiel zeigt das Prinzip des verwendeten Polymorphismus: Ein Feld eines Fileobjekts kann eine Instanz einer Klasse aufnehmen, welche der Klasse des zugeordneten Elements *oder* einer Subklasse davon entspricht. Dieser Polymorphismus für Felder wird über die Angabe eines Typs im Elementpfad-Knoten unterstützt. Der für den Knoten definierte Typ bestimmt den aktuellen Typ des mit dem entsprechenden Element assoziierten Feldes. Das Feld für das `content` Element wird also für das in Abbildung 4.4 gezeigte Fileobjekt über den Pfad `Document/content:Letter` adressiert. Der Typ des Feldes wird über den Typ des letzten Elementpfad-Knotens (`Letter`) bestimmt und nicht über den Typ des `content` Elements (`Text`).

Der gezeigte Polymorphismus erlaubt den einheitlichen Umgang mit gleichgestaltigen Objekten, ohne die speziellen Eigenschaften dieser Objekte einzuschränken. Das über den Elementpfad `Document/content:Letter` adressierte Feld des in Abbildung 4.4 gezeigten Fileobjekts kann sowohl als Instanz eines `Text`- als auch eines `Letter`-Objekttyps angesehen werden.

Wert

Ein Wert *Value* ist ein Feld, das mit einem einfachen Typ assoziiert ist. Das Format des Wertes richtet sich nach dessen Typ und entspricht den üblichen Konventionen: Ein `string`-Wert ist eine beliebige Zeichenkette, ein `integer` eine Ganzzahl, während ein `date` als formatierte Zeichenkette (z.B. `16.08.1977`) dargestellt wird.

Ein Beispiel für einen `string` Wert im Fileobjekt aus Abbildung 4.4 ist `Abendessen` (Adressiert über `Document/content:Letter/keyword:string`). `Document/size:integer` adressiert einen `integer` Wert (1001) und `Document/date:date` einen `date` Wert (`01.08.2002`).

⁵bzw. eine Referenz auf ein Fileobjekt

Anfragen

Grundsätzlich erfolgt der Zugriff auf Fileobjekte über das Objektsystem (*ObjectSystem*) auf Grund ihrer Klasse. Das Objektsystem von OBJECTSFS unterstützt aber auch einfache Anfragen für einen selektiveren Zugriff.

Eine Anfrage (*ObjectQuery*) basiert auf einem Objekttyp und besteht aus einer beliebigen Anzahl von Anfrage-Elementen (*QueryElement*). Der Objekttyp, für den die Anfrage erstellt wird, bestimmt die Klasse der zurückgelieferten Objekte. Jedes Anfrage-Element definiert dabei eine Restriktion auf den Wert eines Feldes. Das Feld ist über einen Elementpfad bestimmt und alle zurückgelieferten Fileobjekte müssen die für dieses Feld festgesetzte Restriktion erfüllen.

Ein Anfrage-Element kann eine der folgenden Restriktionen aufnehmen (Die angeführten Beispiele beziehen sich auf die in Abbildung 4.2 dargestellten Typdefinitionen. Die Resultate der Anfragen sind hier beispielhaft für ein Objektsystem angegeben, das nur das in Abbildung 4.4 gezeigte Fileobjekt mit seinen Subobjekten enthält. Die Resultate haben hier die Form von Elementpfaden und bezeichnen so die von der Anfrage zurückgelieferten Objekte als Felder des Fileobjekts. Das Fileobjekt selbst wird mit **Document** bezeichnet.):

- **Gleichheit:** Der im Anfrage-Element spezifizierte Wert muss mit dem Wert des Feldes übereinstimmen.

Beispiel einer Anfrage

Alle Fileobjekte der Klasse **Document** deren Feld **Document/author:Person/name:string** den Wert **Meier** aufweist.

Resultat der Anfrage

Document.

- **Enthalten:** Der angegebene Wert muss im Wert des Feldes enthalten sein.

Beispiel einer Anfrage

Alle Fileobjekte der Klasse **Letter** deren Feld **Letter/keyword:string** den Wert **essen** enthält.

Resultat der Anfrage

Document/content:Letter.

- **Grösser:** Der Wert des Feldes muss eine höhere Ordnung besitzen als der im Anfrage-Element spezifizierte Wert. (**string** Werte werden auf Grund ihrer alphabetischen Ordnung verglichen. Für **integer** und **date** Werte gelten die natürlichen Ordnungen).

Beispiel einer Anfrage

Alle Fileobjekte der Klasse **Document** deren Feld **Document/size:integer** einen Wert grösser als 1000 aufweist.

Resultat der Anfrage
Document.

- **Kleiner:** Der Wert des Feldes muss eine niedrigere Ordnung besitzen als der im Anfrage-Element spezifizierte Wert.

Beispiel einer Anfrage

Alle Fileobjekte der Klasse **Letter** deren Feld **Letter/date:date** einen Wert kleiner als 02.08.2002 aufweist.

Resultat der Anfrage

Document/content:Letter.

- **Regulärer Ausdruck:** Der Wert des Anfrage-Elements ist ein regulärer Ausdruck und der als Zeichenkette interpretierte Feld-Wert enthält das spezifizierte Muster.

Beispiel einer Anfrage

Alle Fileobjekte der Klasse **Person** deren Feld **Person/name:string** das Muster (Meier|Müller) enthält.

Resultat der Anfrage

Document/author:Person,
Document/content:Letter/sender:Person und
Document/content:Letter/receiver:Person.

4.2.2. Objektrepräsentation in XML

Neben einer internen Repräsentation von Fileobjekten über Java-Klassen lassen sich Fileobjekte auch im XML-Format darstellen. Eine XML-Repräsentation eines Fileobjekts richtet sich nach der XML-Schema Repräsentation der Klasse des Fileobjekts (siehe Abschnitt 4.1.2).

Wie bei den Typdefinitionen bietet die XML Darstellung von Fileobjekten den Vorteil des einfachen Austausches. Ausserdem können andere Applikationen oder Benutzer Fileobjekte in XML Notation erzeugen oder editieren. Über diesen Mechanismus werden Metadaten aus Angaben des Benutzers in OBJECTSFS integriert (siehe Abschnitt 4.3.2).

Tabelle 4.3 zeigt ein Beispiel einer Objektrepräsentation in XML-Format. Es handelt sich dabei um die Repräsentation des in Abbildung 4.4 dargestellten Fileobjekts vom Typ Document.

Das Fileobjekt wird durch ein **object** Element dargestellt und seine Klasse über ein **type**-Attribut festgelegt. Das Präfix **xsi** bezieht sich auf den Namensraum für XML-Schema Instanzen (<http://www.w3.org/2001/XMLSchema-instance>).

Das **type**-Attribut findet auch beim Feld für das **content** Element Verwendung. Die Angabe des Typs (**Letter**) dient dort zur Festlegung des aktuellen Typs dieses Feldes und überschreibt somit den für das **content** Element definierten Typ (**Text**).

```
<object xsi:type='Document'>
  <name>brief.xyz</name>
  <lastModified>01.08.2002</lastModified>
  <size>1001</size>
  <author>
    <name>Meier</name>
  </author>
  <title>Einladung</title>
  <content xsi:type='Letter'>
    <keyword>Abendessen</keyword>
    <keyword>Einladung</keyword>
    <sender>
      <name>Meier</name>
    </sender>
    <receiver>
      <name>Müller</name>
    </receiver>
    <subject>Einladung</subject>
    <date>01.08.2002</date>
  </content>
</object>
```

Tabelle 4.3.: XML-Repräsentation eines Fileobjekts

Das Attribut erlaubt also die Umsetzung des oben beschriebenen Feld-Polymorphismus auf die XML-Repräsentation von Fileobjekten.

4.3. Dateien

4.3.1. Dateisystem

Abbildung 4.5 zeigt die Klassen rund um das Dateisystem und ihre Beziehung zu den Klassen des Objektmodells von OBJECTSFS.

OBJECTSFS erstellt für jede ins System aufgenommene Datei⁶ eine interne Repräsentation: ein *SFSFile*. Jedes *SFSFile* ist mit einer Datei des eigentlichen Dateisystems über einen Pfad verknüpft. Der Pfad bezeichnet die Position der Datei im Verzeichnisbaum zur Zeit der Aufnahme ins System. Ausserdem werden in *SFSFile* zwei Zeitstempel gespeichert: Das Datum der letzten Änderung der Datei (zur Zeit der Aufnahme) und der

⁶Eine Datei muss explizit zum Dateisystem von OBJECTSFS hinzugefügt werden, um von OBJECTSFS verwaltet zu werden.

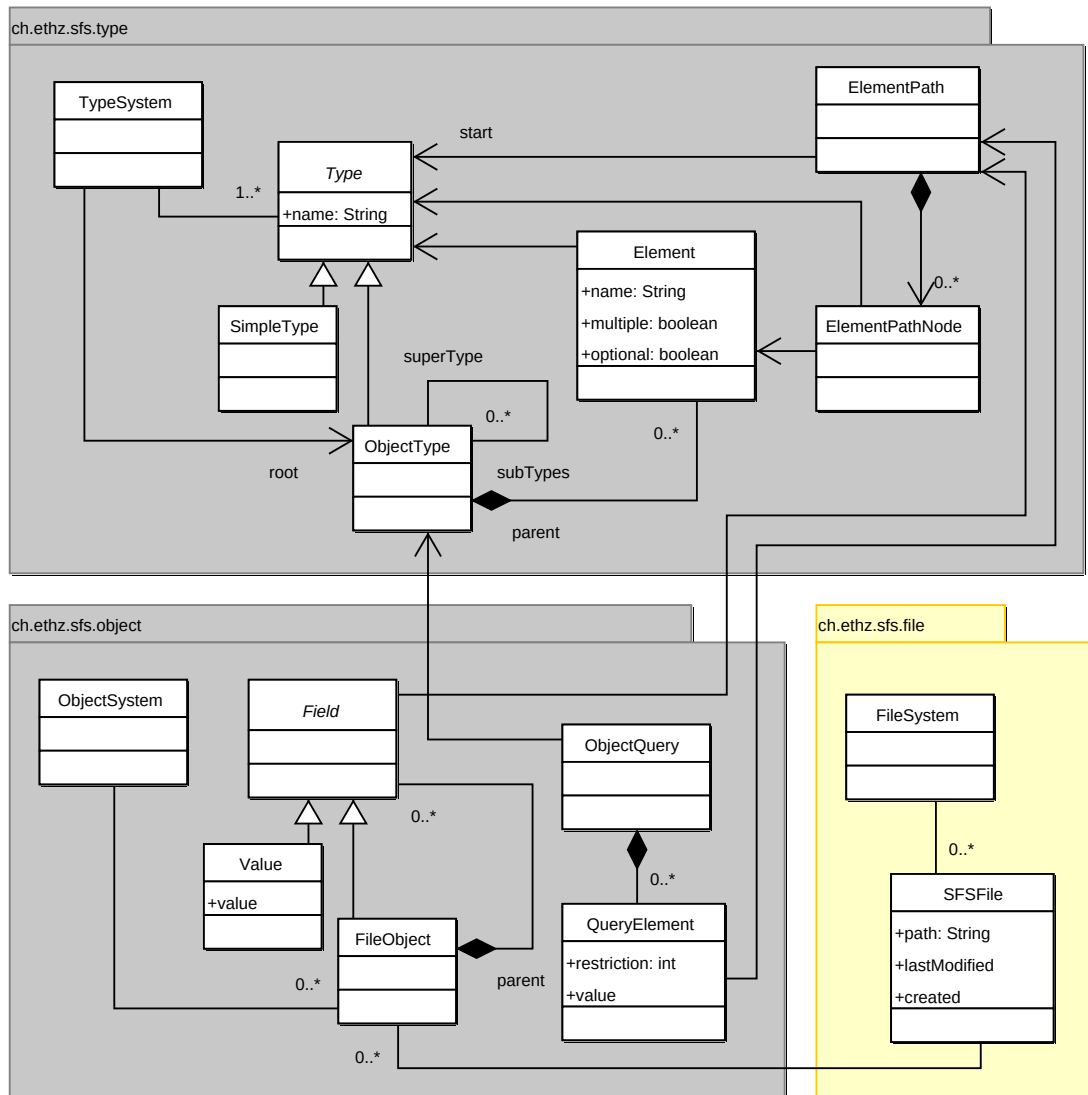


Abbildung 4.5.: Klassendiagramm zum Dateisystem von OBJECTSFS

Zeitpunkt der Aufnahme ins System. Diese beiden Zeitstempel sowie die Pfadangabe werden für die Sicherstellung der Konsistenz zwischen Daten im eigentlichen Dateisystem und in OBJECTSFS benötigt. In Abschnitt 6.3 wird dieser Zusammenhang genauer erklärt.

4.3.2. Datei-Metadaten

Die semantischen Informationen zu einer Datei – sogenannte *Datei-Metadaten* – werden durch Fileobjekte verkörpert, welche mit einem SFSFile verknüpft sind.

Datei-Metadaten können im XML-Format dargestellt werden, was die einfache Integration von extern spezifizierten Metadaten erlaubt (siehe weiter unten). Eine XML-Repräsentation von Datei-Metadaten entspricht dabei einer Instanz eines XML-Dokuments, welche nach dem in Tabelle 4.4 dargestellten Gerüst, ergänzt durch die XML-Repräsentation der enthaltenen Fileobjekte (siehe Abschnitt 4.2.2), aufgebaut ist.

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<sfsFile xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'>
  <object xsi:type= ...
    ...
  </object>
</sfsFile>
```

Tabelle 4.4.: XML-Repräsentation von Datei-Metadaten

Die XML-Repräsentation der Datei-Metadaten lässt sich über das in Tabelle 4.5 gezeigte Gerüst eines XML-Schemas validieren (auf seine Schema-Konformität hin überprüfen). Das gezeigte Schema muss dabei um die Definitionen der Objekttypen des Typsystems, gemäss den in Tabelle 4.1 gezeigten Regeln, erweitert werden.

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<schema xmlns='http://www.w3.org/2001/XMLSchema'>
  <element name='sfsFile'>
    <element name='object' type='Root'
      minOccurs='0' maxOccurs='unbounded' />
  </element>
  <complexType name='Root' ...
    ...
  </complexType>
</schema>
```

Tabelle 4.5.: Gerüst eines XML-Schemas für Datei-Metadaten

Die Fileobjekte, welche die semantischen Informationen zu einer Datei darstellen, entstehen über die systeminterne Verarbeitung der Datei (siehe Abschnitt 4.5) oder werden aus extern definierten Metadaten übernommen. OBJECTSFS integriert extern definierte Metadaten zu einer Datei `foo.bar`, falls sich diese im XML-Format (gemäss

Tabelle 4.4) in einer Datei `foo.bar_sfs.xml` befinden⁷. Existiert eine solche Metadaten-Datei zum Zeitpunkt der Aufnahme einer Datei, so findet keine Verarbeitung dieser Datei statt. Die Fileobjekte zur Beschreibung der Datei werden direkt aus der Metadaten-Datei gewonnen.

4.4. Konfiguration

Eine Konfiguration (*SFSConfiguration*) stellt die Grundlage für die Verarbeitung von Dateien zu Datei-Metadaten (Fileobjekten) dar (siehe Abschnitt 4.5). Sie erlaubt die Steuerung des Verhaltens und die einfache Anpassung von OBJECTSFS an eine bestimmte Umgebung, ohne einen Eingriff auf der Stufe des Systems zu erfordern.

Eine Konfiguration setzt sich aus einem Typsystem (*TypeSystem*) und Verarbeitungsinformationen (*InfoSystem*) zusammen, wie aus dem in Abbildung 4.6 gezeigten Klassendiagramm ersichtlich ist.

Zusätzlich kann eine Konfiguration gewisse Einstellungen über *ConfigPreferences* aufnehmen. Diese Einstellungen werden zu einem späteren Zeitpunkt behandelt (Kapitel 6).

4.4.1. Verarbeitungsinformationen

Verarbeitungsinformationen kommen in zwei Varianten daher:

- Informationen zur Bestimmung der Klasse eines Fileobjekts (*FilterInfo*).
- Informationen zur Extraktion von Feldern eines Fileobjekts (*ExtractorInfo*).

Das in Abbildung 4.6 dargestellte Klassendiagramm zeigt, dass Verarbeitungsinformation einerseits für Objekttypen (*ObjectTypeInfo*), andererseits für Elemente (*ElementInfo*) definiert werden können. Die Verknüpfung von Verarbeitungsinformationen mit einem Element wird über einen Elementpfad realisiert. Der Zugriff auf Verarbeitungsinformationen geschieht grundsätzlich über *InfoSystem*, Verarbeitungsinformationen zu einem Element werden zusätzlich über eine spezielle Klasse (*ObjectInfo*) verwaltet, welche ebenfalls mit einem Elementpfad verknüpft ist. *ObjectInfo* kapselt alle Verarbeitungsinformationen zu den Elementen des durch den Elementpfad bestimmten Objekttyps (des Objekttyps des letzten Elementpfad-Knotens). Verarbeitungsinformationen zum Element `Document/author:Person/name:string` sind beispielsweise in einem *ElementInfo* für diesen Pfad enthalten, das ausgehend von *InfoSystem* über ein *ObjectInfo* für den Pfad `Document/author:Person` zugänglich ist.

⁷Die Datei kann vom Benutzer oder von einer Applikation angelegt werden. In Abschnitt 5.2.1 wird beschrieben, wie eine solche Datei im Rahmen von OBJECTSFS erstellt oder verändert werden kann.

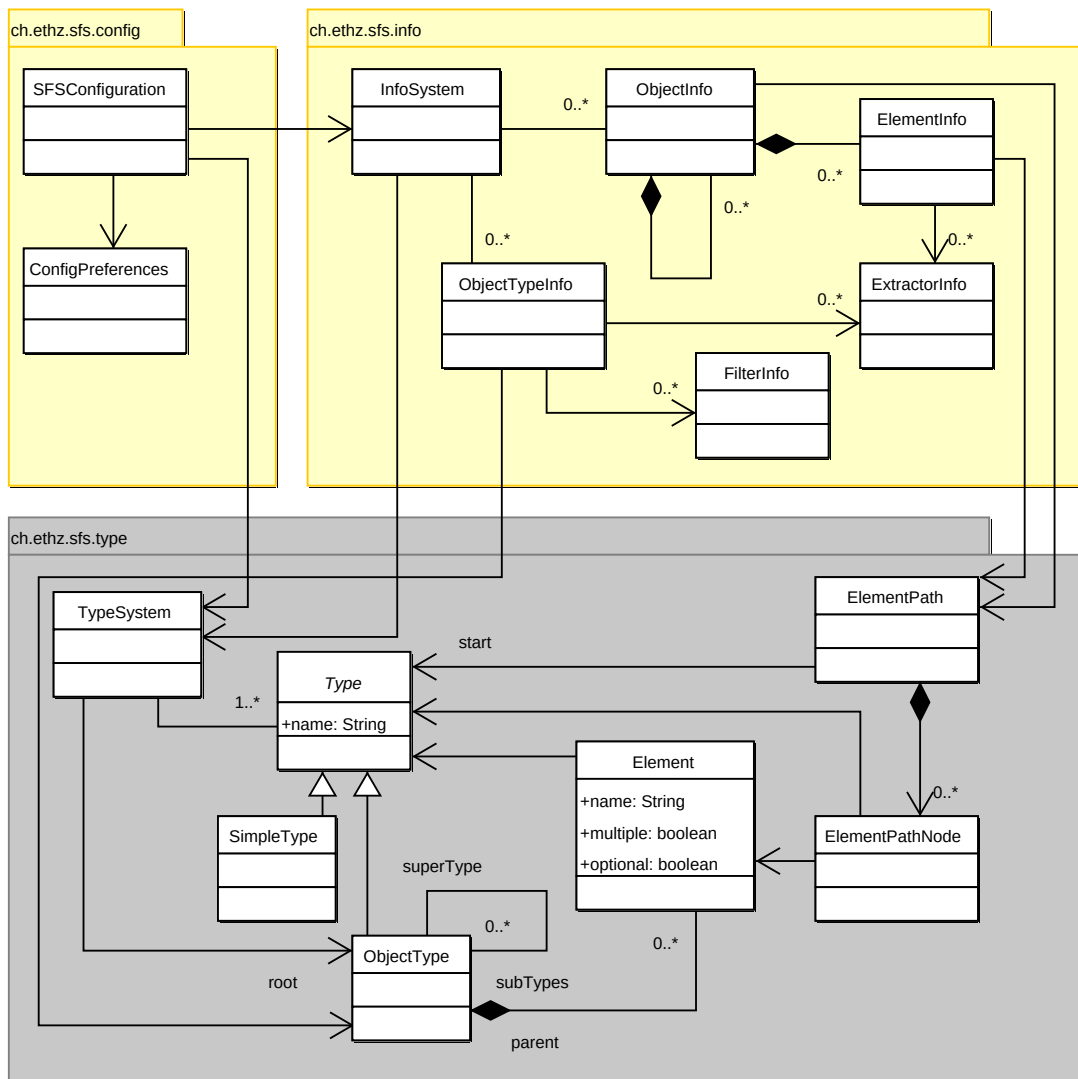


Abbildung 4.6.: Klassendiagramm zu Konfigurationen in OBJECTSFS

Die Verarbeitungsschritte, welche über *Filter* (definiert in *FilterInfo*) und *Extraktoren* (definiert in *ExtractorInfo*) ausgeführt werden, sowie die Verwendung der hier beschriebenen Verarbeitungsinformationen sind in Abschnitt 4.5 beschrieben.

4.4.2. XML-Darstellung einer Konfiguration

Konfigurationen lassen sich einfach in ein XML-Format überführen. Dazu werden die Objekttypen des Typsystem (gemäss dem in Tabelle 4.5 vorgestellten XML-Schema), die Verarbeitungsinformationen (<infoSystem>), sowie die Konfigurationseinstellungen (<preferences>⁸) in XML-Format gebracht und in das in Tabelle 4.6 dargestellte Gerüst eingefügt.

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<<configuration name='...'>>
  <schema xmlns='http://www.w3.org/2001/XMLSchema'>
    ...
  </schema>
  <infoSystem>
    ...
  </infoSystem>
  <preferences ... />
</configuration>
```

Tabelle 4.6.: Darstellung einer Konfiguration in XML-Format

Beispiele für komplette Konfigurationen und ihre XML-Darstellung finden sich im Anhang C. Sie werden in Kapitel 7 beschrieben.

4.5. Verarbeitung

Die Verarbeitung von Dateien zu Datei-Metadaten (bzw. Fileobjekten) geschieht über die in den Verarbeitungsinformationen einer Konfiguration spezifizierten *Filter* und *Extraktoren*.

Filter und Extraktoren arbeiten auf einem *Kontext*. Ein Filter entscheidet für einen gegebenen Kontext, ob er ihn akzeptiert. Ein Extraktor erzeugt zu einem bestimmten Eingabe-Kontext einen Ausgabe-Kontext.

4.5.1. Kontext

Ein Kontext repräsentiert den aktuellen Stand der Verarbeitung einer Datei und dient als Grundlage für das Erzeugen von Fileobjekten zur Beschreibung dieser Datei. Kontexte liefern die Daten für die Felder von Fileobjekten. Diese werden direkt als Wert interpretiert (bei Wert-Feldern) oder einer weiteren Verarbeitung unterzogen (bei Feldern, die ein Subobjekt verkörpern).

⁸Die konkreten Einstellungen werden als Attribute des <preferences>-Element angegeben.

Abbildung 4.7 zeigt die Klassen zu einem Kontext (*ProcessingContext*) in UML Notation.

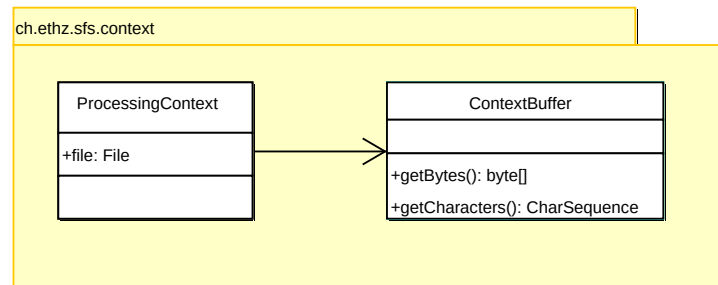


Abbildung 4.7.: Klassen zur Repräsentation eines Kontextes

Ein Kontext enthält immer einen Verweis auf eine Datei (*file*) und auf einen Kontextbuffer (*ContextBuffer*). Die Datei, auf die der Kontext verweist, ist die momentan verarbeitete Datei. Der Kontextbuffer kapselt die Daten des Kontextes und stellt Methoden für den Zugriff auf die Daten zur Verfügung. Die Daten liegen grundsätzlich in Byte-Form vor (Zugriff über *getBytes()*), können aber vom Kontextbuffer zu Zeichen codiert und als Zeichenkette zurückgegeben werden (*getCharacters()*).

Am Anfang der Verarbeitung einer Datei steht ein leerer Kontext. Dieser Kontext wird vom System für die Datei angelegt und wird fortan mit *Initial-Kontext* bezeichnet. Aus dem Initial-Kontext entstehen im Laufe der Verarbeitung weitere Kontexte durch die Anwendung von Extraktoren (siehe Abschnitt 4.5.3). Extraktoren können aus einem Eingabe-Kontext einen beliebigen Ausgabe-Kontext generieren. Die Kontextdaten (die Daten des Kontextbuffers) eines Ausgabe-Kontextes sind in der Regel eine Untermenge der Daten des Eingabe-Kontextes oder sind direkt aus der Datei gewonnen.

Es gibt eine enge Verwandtschaft zwischen einem Kontext und einem Feld: Jedes Fileobjekt und jedes Feld eines Fileobjekts, das während der Verarbeitung entsteht, ist implizit mit einem Kontext assoziiert. Wertfelder beziehen ihren Wert direkt aus diesem Kontext. Dateien bzw. Objekte werden auf Grund des Kontextes mit Hilfe von Filtern klassifiziert (siehe Abschnitt 4.5.2).

Während der Verarbeitung entspricht ein Fileobjekt bzw. ein Feld also einem Kontext. Um diese Beziehung auszudrücken werden im folgenden Text Ausdrücke wie *Objekt-Kontext* (bzw. *Feld-Kontext*) oder *Kontext-Objekt* (bzw. *Kontext-Feld*) verwendet. Ein *Objekt-Kontext* (bzw. *Feld-Kontext*) bezeichnet dabei den zu einem Fileobjekt (bzw. zu einem Feld) gehörigen Kontext, während ein *Kontext-Objekt* (bzw. *Kontext-Feld*) das Fileobjekt (bzw. das Feld) bezeichnet, das mit dem Kontext assoziiert ist.

Abbildung 4.8 zeigt die Extraktion eines `Document/title:string` Feldes eines Fileobjekts vom Typ `Document`, um die Verarbeitung von Kontexten zu illustrieren⁹.

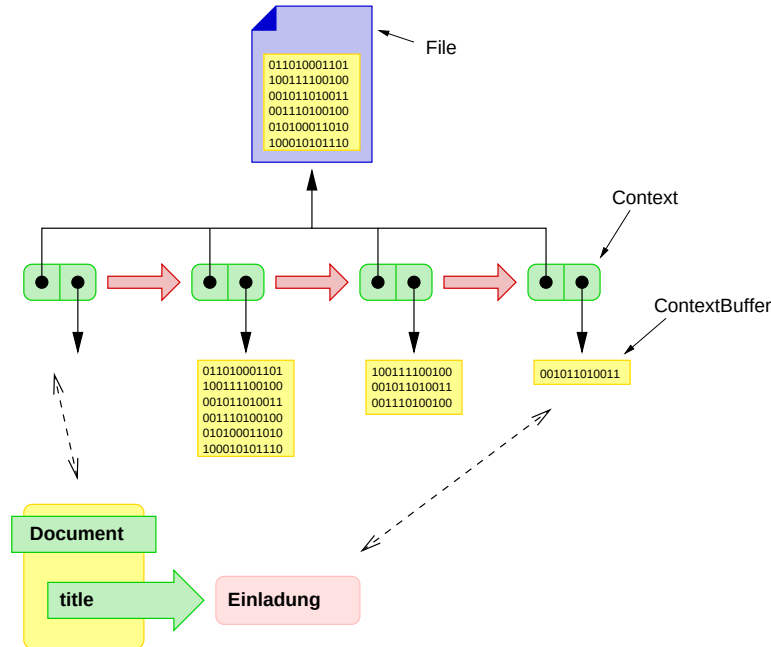


Abbildung 4.8.: Extraktion eines Feldes über die Verarbeitung von Kontexten

Die Abbildung zeigt vier Kontexte (`Context`), die im Laufe der Verarbeitung entstehen. Der erste Kontext stellt den Initial-Kontext der Verarbeitung dar. Er ist mit der zu verarbeitenden Datei (`File`) verknüpft und sein Kontextbuffer (`ContextBuffer`) ist leer. Das dargestellte Fileobjekt vom Typ `Document` ist mit diesem Kontext assoziiert (Der Kontext wurde vorgängig als `Document` über einen Filter klassifiziert. Siehe dazu Abschnitt 4.5.2). Die folgenden Kontexte entstehen über interne Verarbeitungsschritte. Die Verarbeitung erfolgt über die Anwendung von Extraktoren (siehe Abschnitt 4.5.3). Im gezeigten Beispiel erzeugt der erste Extraktor aus dem Initial-Kontext einen Ausgabe-Kontext, welcher in seinem Kontextbuffer den Inhalt der Datei kapselt. Dieser Extraktor erzeugt die Kontextdaten also direkt aus Daten aus der Datei, ohne die (leeren) Kontext-Daten des Initial-Kontextes zu berücksichtigen. Die weiteren Extraktoren erzeugen Kontexte, deren Daten eine Untermenge der Daten des Eingabe-Kontextes darstellen. Sobald der Kontext für das `Document/title:string` Feld bereitsteht, kann der Wert dieses Feldes aus dem Feld-Kontext (bzw. aus den Daten im Kontextbuffer) gewonnen werden.

⁹Das gezeigte Beispiel und die folgenden Beispiele beziehen sich auf die in Abbildung 4.2 eingeführten Objekttyp-Definitionen und das in Abbildung 4.4 dargestellte Fileobjekt.

4.5.2. Filter

Filter werden für Objekttypen definiert (*FilterInfo* in *ObjectTypeInfo*). Für einen Objekttyp können mehrere Filter angelegt werden.

Am Anfang der Verarbeitung einer Datei wird der Initial-Kontext über Filter klassifiziert. Mit anderen Worten werden mit Hilfe von Filtern die Objekttypen der Fileobjekte bestimmt, welche die Datei beschreiben sollen. Wir nennen diesen Vorgang *Initial-Klassifikation* oder *Klassifikation einer Datei*. Als Grundlage für die Klassifikation eines Initial-Kontextes kann ein Filter nur die mit dem Kontext assoziierte Datei verwenden, da der Kontextbuffer anfangs leer ist.

Um zu entscheiden, ob sich eine Datei zu einem bestimmten Objekttyp klassifizieren lässt, werden alle Filter, welche für diesen Objekttyp definiert sind, auf den Kontext angewendet. Akzeptieren alle diese Filter den fraglichen Kontext (Filter liefern einen Wahrheitswert zu einem Eingabe-Kontext), so wird für die Datei ein Fileobjekt des entsprechenden Typs angelegt. Das Fileobjekt ist implizit mit dem Initial-Kontext assoziiert.

In Abbildung 4.9 ist dieser Sachverhalt beispielhaft dargestellt. Das Bild zeigt einen für den Objekttyp **Document** definierten Filter, der einen Eingabe-Kontext (Initial-Kontext) klassifiziert (wir nehmen an, dass keine weiteren Filter für **Document** definiert sind).

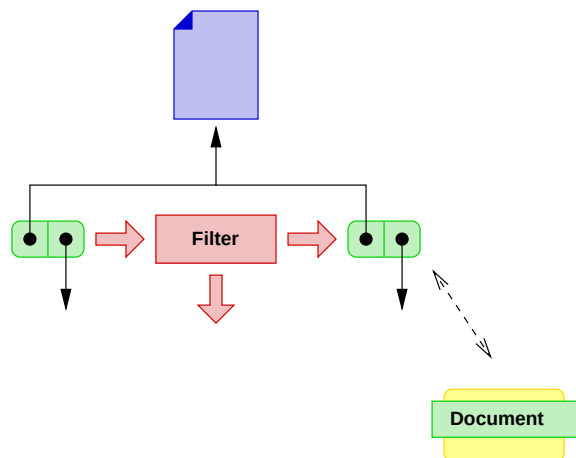


Abbildung 4.9.: Beispiel einer Initial-Klassifikation

Akzeptiert der in Abbildung 4.9 dargestellte Filter seinen Eingabe-Kontext, so wird ein Fileobjekt vom Typ **Document** angelegt (andernfalls wird kein neues Objekt angelegt). Das Fileobjekt dient der Aufnahme von semantischen Informationen zur momentan verarbeiteten Datei.

Da eine Datei über mehrere Fileobjekte verschiedener Klassen beschrieben werden kann, wird der Vorgang wiederholt, bis die Datei (bzw. der Initial-Kontext) vollständig klassifiziert ist und alle Fileobjekte feststehen. Eine Datei kann allerdings nie über zwei Fileobjekte beschrieben werden, deren Klassen direkt oder indirekt voneinander abgeleitet sind (z.B. `File` und `Document`). Die spezialisiertere Klasse hat immer den Vorrang (`Document`). Die Verwendung der allgemeineren Klasse (`File`) erübrigt sich, da sie in der spezialisierteren Klasse *enthalten* ist (Die spezialisiertere Klasse weist alle Elemente der allgemeineren Klasse auf.).

Abbildung 4.10 zeigt den Ablauf der Initial-Klassifikation in Form eines Diagramms. D bezeichnet die zu klassifizierende Datei, C steht für den zugehörigen Initial-Kontext, T bezeichnet einen Objekttyp und $F(T)$ die für T definierten Filter.

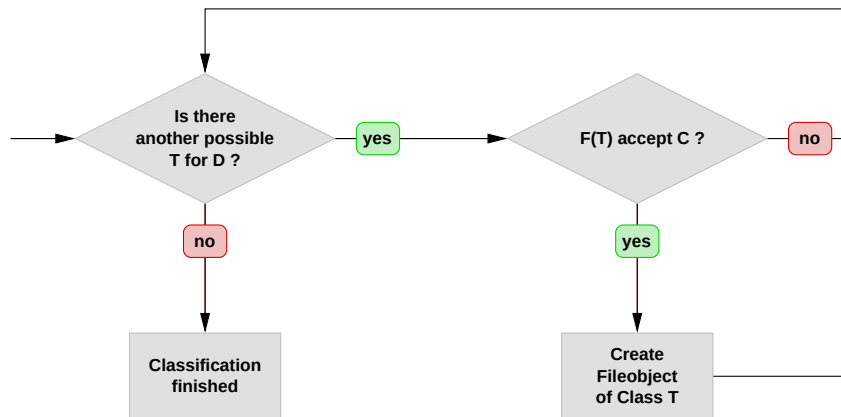


Abbildung 4.10.: Ablauf der Initial-Klassifikation

Im Laufe der Verarbeitung kommen Filter erneut zum Einsatz, um den *dynamischen Typ* eines Subobjekts zu bestimmen. Grundsätzlich ist die Klasse eines Subobjekts durch den Typ des entsprechenden Feldes bestimmt. Über die Anwendung von Filtern auf den Objekt-Kontext kann das Subobjekt allerdings als Instanz einer Subklasse dieser (durch den Feld-Typ definierten) Klasse identifiziert werden (Polymorphismus). Wir nennen diesen Vorgang *dynamische Klassifikation* oder *Klassifikation eines Objekts*. Die dynamische Klassifikation eines Fileobjekts basiert in der Regel auf den Daten im Kontextbuffer des entsprechenden Objekt-Kontextes. Anders als bei der Initial-Klassifikation ist dieser Kontextbuffer im Allgemeinen nicht leer.

Abbildung 4.11 zeigt die dynamische Klassifikation eines Objekts auf Grund seines Kontextes. Das durch das Feld `Document/content:Text` dargestellte Subobjekt wird mittels eines Filters als Instanz von `Letter` erkannt (Der Filter ist der einzige für `Letter` definierte Filter und akzeptiert den Eingabe-Kontext.). Der so bestimmte dynamische Typ des Feldes (hier `Letter`) bestimmt die weitere Verarbeitung.

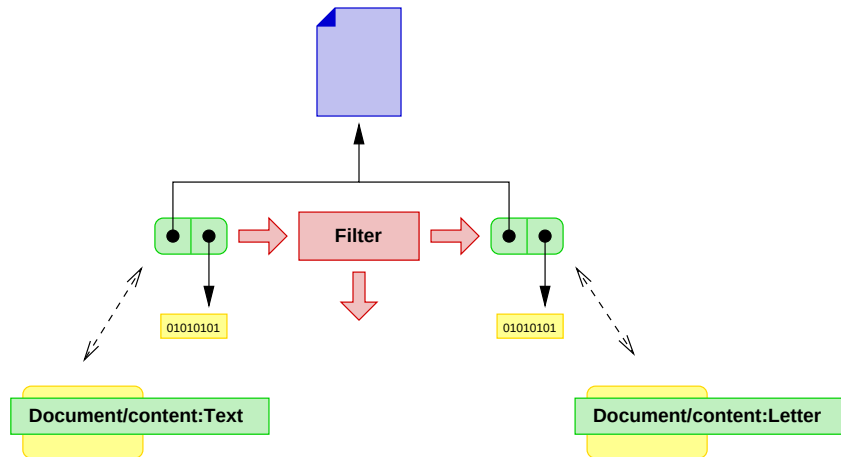


Abbildung 4.11.: Beispiel für die dynamische Klassifikation eines Fileobjekts

Der dynamische Typ eines Subobjekts kann nur eine Erweiterung des eigentlichen Typs dieses Subobjekts sein. Es werden also nur die Filter angewendet, die für Subtypen dieses Typs definiert sind. Der dynamische Typ bestimmt sich dann aus dem am meisten spezialisierten Objekttyp (der am weitesten von Root entfernte Objekttyp in der Vererbungshierarchie) für den folgendes gilt:

- Mindestens ein Filter ist für den Objekttyp definiert.
- Alle für den Objekttyp definierten Filter akzeptieren den Kontext des Subobjekts.

Lässt sich kein dynamischer Typ für ein Subobjekt bestimmen, so behält es seinen aktuellen Typ bei.

4.5.3. Extraktoren

Extraktoren können für Objekttypen und für Elemente von Objekttypen, beschrieben durch einen Elementpfad, definiert werden (*ExtractorInfo* in *ObjectTypeInfo* bzw. *ElementInfo*).

Extraktoren übernehmen grundsätzlich die folgenden zwei Aufgaben:

1. Extrahieren des Kontextes (bzw. der Kontexte) für ein Feld (Wert-Feld oder Subobjekt) eines Fileobjekts.
2. Extrahieren eines Eingabe-Kontextes für einen Filter oder für einen anderen Extraktor (Kaskadierung von Extraktoren).

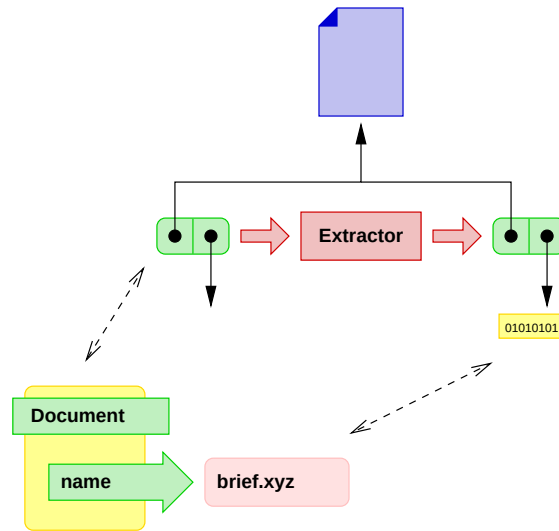


Abbildung 4.12.: Funktionsprinzip eines Extraktors

Abbildung 4.12 zeigt das Funktionsprinzip eines Extraktors. Ein Extraktor nimmt einen Kontext als Eingabe und liefert einen Kontext (oder auch mehrere Kontexte) als Ausgabe.

Das Beispiel aus Abbildung 4.12 zeigt einen für die Extraktion des Feldes `Document/name:string` zuständigen Extraktor. Das Kontext-Objekt des Eingabe-Kontextes ist ein Fileobjekt vom Typ `Document`. Der Kontextbuffer des Eingabe-Kontextes ist leer (Es handelt sich um einen als `Document` klassifizierten Initial-Kontext.). Der Extraktor liefert den Feld-Kontext, der den Wert des Feldes `Document/name:string` bestimmt. Der extrahierte Wert ist `brief.xyz`.

Für multiple Elemente extrahiert ein Extraktor unter Umständen mehrere Feld-Kontexte. Abbildung 4.13 zeigt ein Beispiel eines für die Extraktion von `Document/content:Text/keyword:string` Feldern verantwortlichen Extraktors. Die Anzahl der extrahierten Kontexte bestimmt die Anzahl der entsprechenden Felder des Fileobjekts.

Der Extraktor aus Abbildung 4.13 extrahiert zwei Kontexte, welche die Werte für die Felder `Document/content:Text/keyword:string` des abgebildeten Fileobjekts bestimmen: `Abendessen` und `Einladung`.

Extraktoren können auch den Kontext eines Subobjekts bestimmen. Dies ist in Abbildung 4.14 an einem Beispiel gezeigt.

Der in Abbildung 4.14 dargestellte Extraktor bestimmt aus dem Eingabe-Kontext für das Fileobjekt `Document/content:Letter` den Kontext für das Subobjekt `Document/content:Letter/sender:Person`.

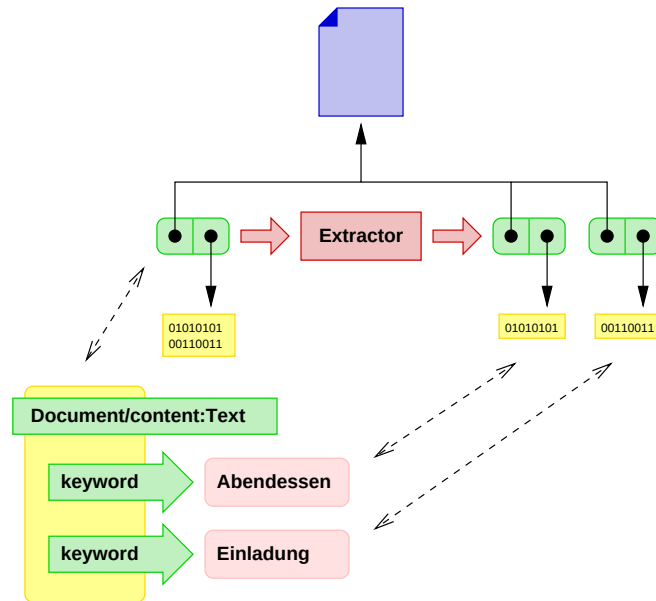


Abbildung 4.13.: Extraktion mehrerer Feld-Kontexte

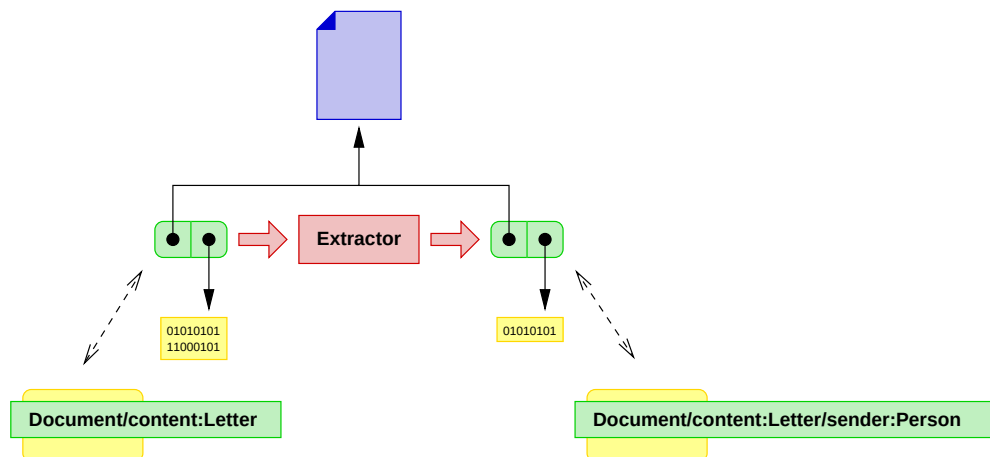


Abbildung 4.14.: Extraktion des Kontextes eines Subobjekts

Kaskadierung von Extraktoren

Der Eingabe-Kontext eines Filters oder eines Extraktors ist bestimmt durch das Ergebnis eines vorhergehenden Verarbeitungsschrittes oder durch den Ausgabe-Kontext eines dazwischengeschalteten Extraktors, den wir *Basis-Extraktor* nennen. Basis-Extraktoren können über Angaben in den Verarbeitungsinformationen (in *FilterInfo*

bzw. *ExtractorInfo*) für Filter und Extraktoren definiert werden. Die Angabe von Basis-Exraktoren in den Verarbeitungsinformationen erlaubt die Kaskadierung von Extraktoren zur Bestimmung des Eingabe-Kontextes.

Bei Verarbeitungsinformationen, die für einen Objekttyp (über *ObjectTypeInfo*) definiert sind, beschränkt sich die Wahl des Basis-Exraktors auf alle Extraktoren, die für diesen Objekttyp definiert sind (über *ExtractorInfo* in *ObjectTypeInfo*). Ein für den Objekttyp **Document** definierter Filter zur Klassifikation eines Kontextes kann jeden ebenfalls für diesen Objekttyp definierten Extraktor als Basis-Extraktor verwenden. So kann der Filter seinen Eingabe-Kontext beispielsweise durch einen Extraktor bestimmen lassen, der den Inhalt der zu verarbeitenden Datei extrahiert und in seinem Ausgabe-Kontext zur Verfügung stellt.

Für einen Objekttyp definierte Extraktoren werden nie zur Extraktion eines Feldes verwendet. Aus diesem Grund können sie nur von anderen Extraktoren als Basis-Exraktoren verwendet werden.

Ein Extraktor, der für ein über einen Elementpfad adressiertes Element definiert ist (über *ExtractorInfo* in *ElementInfo*) kann als Basis-Extraktor diejenigen Extraktoren verwenden, die für dasselbe Element oder für dessen *parent*-Objekttyp definiert sind. So kann ein für das **Document/title:string** Element definierter Extraktor einen ebenfalls für dieses Element oder einen für **Document** (im entsprechenden *ObjectTypeInfo*) definierten Extraktor als Basis verwenden. Dieses Konzept der Basis-Exraktoren erlaubt die Wiederverwendung von Extraktoren und erleichtert so die Konfiguration des Systems¹⁰. So können beispielsweise die Extraktoren für die Extraktion der Felder eines **Document** Fileobjekts alle auf ein und denselben für den Objekttyp **Document** definierten Basis-Extraktor zurückgreifen. Dieses Beispiel ist in Abbildung 4.15 dargestellt.

Die Extraktoren für die Felder **Document/title:string** und **Document/author:Person** (Extractor 3 und Extractor 2) sind in Abbildung 4.15 beide mit demselben für den Objekttyp **Document** definierten Basis-Extraktor verknüpft (Extractor 1). Der Eingabe-Kontext der beiden Extraktoren wird von diesem Basis-Extraktor bestimmt. Aus dem Eingabe-Kontext extrahiert der Extraktor für das Feld **Document/title:string** (Extractor 3) den Wert **Einladung**, während der Extraktor für das Feld **Document/author:Person** (Extractor 2) einen Kontext für das entsprechende Subobjekt liefert.

Extraktion von Feldern und Pfadrelaxierung

Die Hauptaufgabe eines für ein Element definierten Extraktors ist es, die Felder (bzw. die Kontexte für die Felder) eines Fileobjekts zu extrahieren. Für jedes Feld wird dazu der geeignete Extraktor (bzw. die geeigneten Extraktoren) bestimmt: Ist für den Ele-

¹⁰ Ausserdem steigert die Wiederverwendung von Extraktoren die Effizienz der Verarbeitung: Dazu werden die Ausgabe-Kontexte von Basis-Exraktoren zwischengespeichert. Sie müssen dann bei erneuter Verwendung des Extraktors nicht mehr neu bestimmt werden.

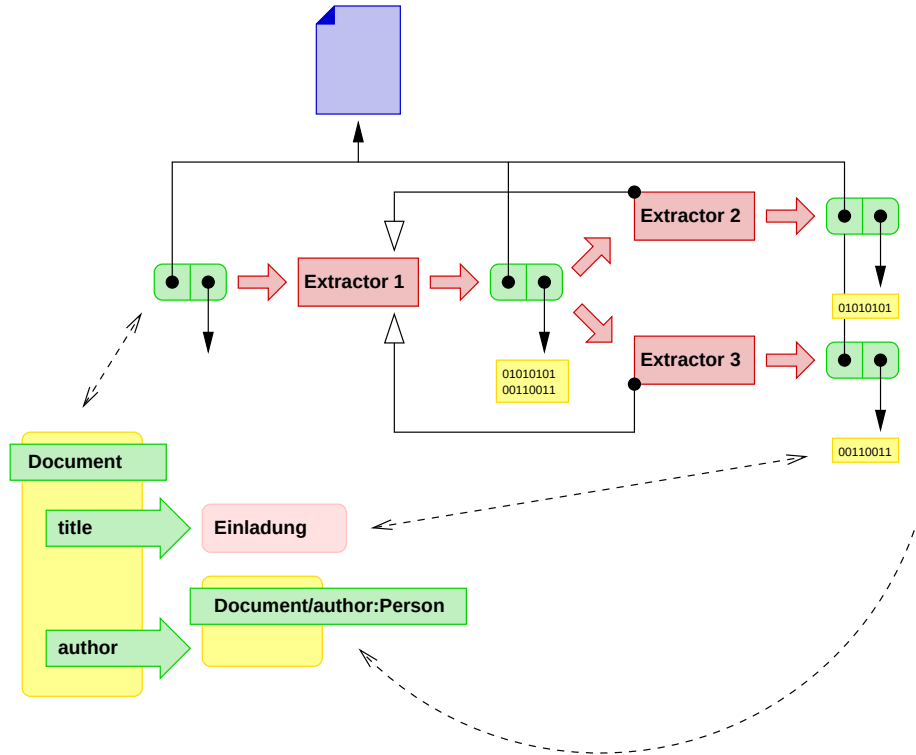


Abbildung 4.15.: Kaskadierung von Extraktoren

mentpfad zum Feld ein Extraktor (oder mehrere Extraktoren) definiert (*ExtractorInfo* im entsprechenden *ElementInfo*), so wird dieser Extraktor zur Extraktion des Feldes verwendet. Ist kein Extraktor für diesen Elementpfad definiert, wird ein Extraktor verwendet, der für einen weniger spezifischen – einen *relaxierten* – Elementpfad definiert ist. Im folgenden wird zuerst die Technik der Pfadrelaxierung beschrieben, um dann auf die Auswirkungen bei der Extraktion von Feldern und den Nutzen bei der Erstellung von Konfigurationen einzugehen.

Tabelle 4.7 listet einige Beispiele von Elementpfaden mit den zugehörigen relaxierten Elementpfaden auf. Die gezeigten Beispiele sollen die folgenden Ausführungen illustrieren.

Jede der folgenden Operationen führt einen Elementpfad in einen relaxierten Elementpfad über:

- Ersetzen des Starttyps des Elementpfades durch seinen Supertyp: $T/e_1 : t_1/\dots/e_n : t_n \longrightarrow S/e_1 : t_1/\dots/e_n : t_n$, wobei S der Supertyp von T ist, und e_1 ein gültiges Element von S sein muss.

Document/author:Person/name:string	→ Person/name:string
Letter/sender:Person/name:string	→ Person/name:string
Document/content:Text/keyword:string	→ Text/keyword:string
Document/content:Letter/keyword:string	→ Letter/keyword:string
Letter/keyword:string	→ Text/keyword:string

Tabelle 4.7.: Elementpfade und ihre Relaxierung

Beispiel

Letter/keyword:string → Text/keyword:string.

- Entfernen des ersten Knotens aus dem Elementpfad. Der Typ des entfernten Knotens wird zum neuen Starttyp des Elementpfades: $T/e_1 : t_1 / \dots / e_n : t_n \rightarrow t_1/e_2 : t_2 / \dots / e_n : t_n$.

Beispiel

Document/author:Person/name:string → Person/name:string.

Die Relaxierung eines Pfades setzt voraus, dass der Pfad über mindestens ein Element verfügt ($n \geq 1$). Ist der Pfad leer und besteht nur aus einem Starttyp ($n = 0$), so kann dieser nicht relaxiert werden.

Die erste Operation hat Vorrang beim Relaxieren eines Elementpfades, kann jedoch nur angewendet werden, wenn der Starttyp des Elementpfades T einen Supertyp (S) hat und das erste Element des Pfades e_1 ein gültiges Element dieses Supertyps ist. Kann die erste Operation nicht durchgeführt werden (z.B. für Document/author:Person/name:string, da author kein gültiges Element des Supertyps File von Document ist), so kommt die zweite Operation zur Anwendung. Abbildung 4.16 zeigt die Schritte beim Relaxieren eines Elementpfades P in einem Ablaufdiagramm.

Die Suche nach Extraktoren über das Relaxieren von Pfaden beseitigt die Notwendigkeit, für jeden Elementpfad einen Extraktor definieren zu müssen. Werden Felder eines Fileobjekts einer bestimmten Klasse stets auf die gleiche Art extrahiert, so genügt es die Extraktoren für die Elemente des entsprechenden Typs (der Klasse des Fileobjekts) zu definieren. Gibt es eine Situation, in der dieses Feld auf eine spezielle Art extrahiert werden muss, so kann das Standardverhalten über die Angabe eines entsprechenden Extraktors für einen spezifischeren Elementpfad überschrieben werden. Folgendes Beispiel soll dies verdeutlichen: Die Schlüsselwörter (keyword) eines Textes (Text) werden in der Regel immer auf die gleiche Art und Weise extrahiert. Es genügt also einen Schlüsselwortextraktor für den Elementpfad Text/keyword:string zu definieren. Sollen nun die Schlüsselwörter eines Textes (Text) extrahiert werden, der den Inhalt (content) eines Dokuments (Document) darstellt, so wird über Pfadrelaxierung (Document/content:Text/keyword:string → Text/keyword:string) sichergestellt, dass der für den Elementpfad Text/keyword:string definierte Extraktor verwendet wird. Soll bei der Ex-

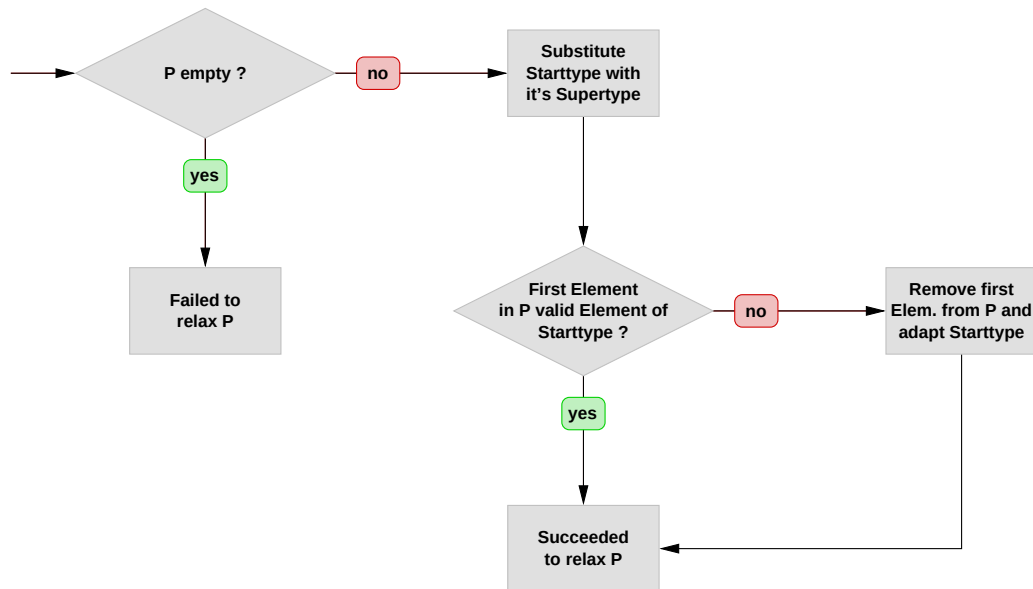


Abbildung 4.16.: Schritte beim Relaxieren eines Elementpfades

traktion von Schlüsselwörtern aus dem Inhalt eines Dokuments jedoch ein spezieller Extraktionsmechanismus zum Zug kommen (etwa weil die Schlüsselwörter in den dokumenteigenen Metadaten auftreten), so kann das Standardverhalten (das Verwenden eines für einen relaxierten Pfad definierten Extraktors) durch die Angabe eines Extraktors für den Elementpfad `Document/content:Text/keyword:string` überschrieben werden.

Sind einmal die Extraktoren für die Extraktion eines Feldes bestimmt (ev. unter Verwendung von Pfadrelaxierung), so werden sie auf den Eingabe-Kontext angewendet, um den Ausgabe-Kontext zu bestimmen. Der Ausgabe-Kontext kann sich dabei aus dem Resultat verschiedener Extraktoren zusammensetzen, falls mehrere Extraktoren für die Extraktion des Feldes zu einem (nicht-multiplen) Element definiert sind. Die Kontextbuffer werden dazu entsprechend zusammengefasst (aneinandergehängt), um den Kontextbuffer für den resultierenden Ausgabe-Kontext zu bilden.

Abbildung 4.17 zeigt die Vorgehensweise bei der Extraktion eines Feldes F , das über einen Elementpfad P adressiert ist.

Die Abbildung zeigt auch das Verhalten des Systems, wenn kein neuer Kontext für ein Feld extrahiert werden kann (weil keine geeigneten Extraktoren definiert sind und die Pfadrelaxierung schlussendlich scheitert). In diesem Fall wird der alte Kontext (der Eingabe-Kontext) zum neuen Kontext des Feldes und die Verarbeitung wird mit diesem (unveränderten) Kontext fortgesetzt.

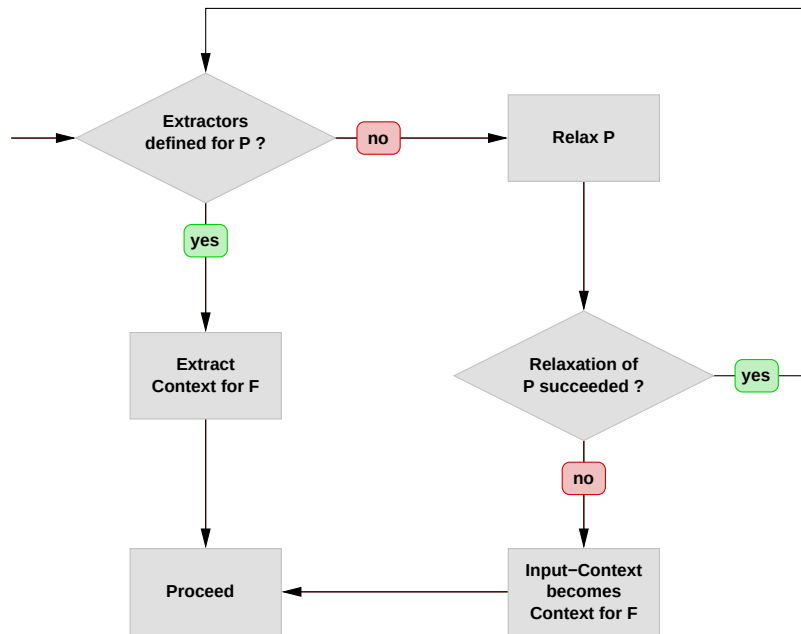


Abbildung 4.17.: Extraktion eines Feld-Kontextes

4.6. Zusammenfassung

OBJECTSFS definiert ein an XML-Schema angelehntes Objektmodell, welches die Konzepte der Objektorientierung (Vererbung, Polymorphismus) berücksichtigt. Das Modell erlaubt das Erstellen von Objekttypen, welche über eine Konfiguration mit Verarbeitungsinformationen versehen werden. Die Konfiguration steuert das Verhalten des Systems bei der Verarbeitung von Dateien zu strukturierten Metadaten-Objekten. Die einfache Konfigurierbarkeit von OBJECTSFS erlaubt die schnelle Anpassung an sich verändernde Umgebungen und erhöht damit die Flexibilität des Systems.

Verarbeitungsinformationen teilen sich auf in die Angabe von Filtern und von Extraktoren. Die Klassifikation von Dateien und Objekten geschieht über Filter, während die Extraktion von Feldern über Extraktoren geschieht. Filter und Extraktoren arbeiten jeweils auf einem Eingabe-Kontext. Der Eingabe-Kontext resultiert dabei aus dem (leeren) Initial-Kontext über verschiedene Verarbeitungsschritte unter Anwendung von Extraktoren oder speziell definierter Extraktor-Ketten.

OBJECTSFS implementiert eine eigene Dateiverwaltung über dem eigentlichen Dateisystem. Dateien müssen explizit zum Dateisystem von OBJECTSFS hinzugefügt oder aus diesem entfernt werden. Für jede aufgenommene Datei stellt das System semantische Informationen, sogenannte Datei-Metadaten zur Verfügung. Die Datei-Metadaten stellen entweder das Resultat einer internen Verarbeitung dar oder werden von aussen zur

Verfügung gestellt. Spezielle Mechanismen müssen implementiert werden, um die Konsistenz zwischen Dateidaten und Datei-Metadaten zu gewährleisten (siehe Abschnitt 6.3).

Typsystem, Fileobjekte, Datei-Metadaten und Konfigurationen lassen sich in XML-Format darstellen. Das XML-Format erlaubt den unkomplizierten Austausch von Daten und bildet die Grundlage für die einfache Integration extern definierter Metadaten. Die Überführung des Typsystems in ein XML-Schema ermöglicht zudem die Validierung dieser Metadaten durch externe Applikationen.

5. Grafische Benutzeroberfläche

Eine Test-Applikation bietet eine grafische Benutzerschnittstelle zu den wesentlichen Komponenten von OBJECTSFS. Die Schnittstelle der Test-Applikation erlaubt die Interaktion des Benutzers mit dem System und wird in diesem Kapitel vorgestellt.

Abbildung 5.1 zeigt den von der Test-Applikation gebotenen Einstiegspunkt für die Interaktion mit OBJECTSFS.

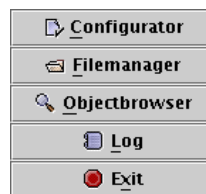


Abbildung 5.1.: Von der Test-Applikation gebotener Einstiegspunkt für den Zugriff auf die Hauptkomponenten von OBJECTSFS

Die Test-Applikation umfasst drei wesentliche Komponenten:

- **Configurator:** Erlaubt die Erstellung und Verwaltung von Konfigurationen.
- **Filemanager:** Bietet Zugang zum Dateisystem von OBJECTSFS und ermöglicht das Hinzufügen und Entfernen von Dateien.
- **Objectbrowser:** Bietet Zugang zu den Fileobjekten. Der Objektbrowser unterstützt das Erstellen einfacher Anfragen an das Objektsystem und kümmert sich um die Darstellung der zurückgelieferten Resultate.

Daneben lässt sich über die Test-Applikation ein Fenster zur Anzeige von Ausgaben des Systems öffnen, ein sogenanntes *Log* (siehe Abbildung 5.2).

OBJECTSFS erzeugt verschiedene Log-Nachrichten, die beispielsweise Veränderungen am Typsystem, den internen Zustand der Verarbeitung von Dateien oder eingetretene Ausnahmebedingungen dokumentieren. Das Log-Fenster stellt die Log-Nachrichten des Systems dar, gefiltert nach ihrer Wichtigkeit¹. Dazu lässt sich über das Menü des

¹Die Log-Nachrichten werden mit einem *Log-Level* (siehe auch Java API-Dokumentation zum Paket `java.util.logging` [9]) versehen, das die Wichtigkeit der Nachricht bezeichnet.

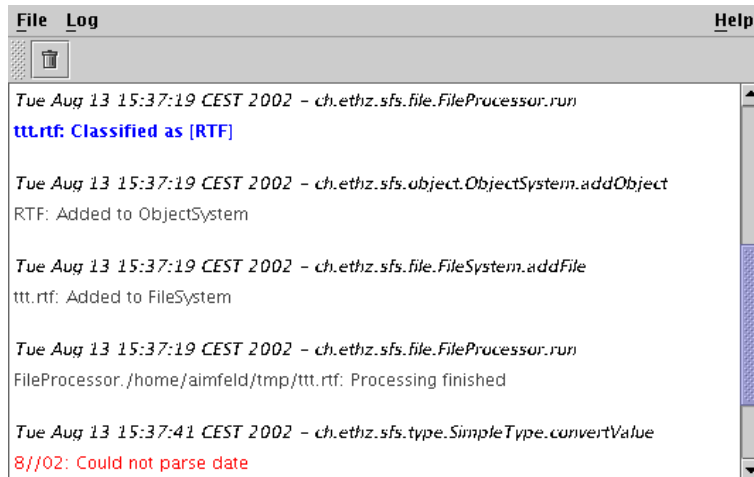


Abbildung 5.2.: Log-Fenster für die Anzeige von Log-Nachrichten des Systems

Log-Fensters ein Schwellwert festsetzen, so dass nur noch diejenigen Nachrichten angezeigt werden, deren Wichtigkeit über dem eingestellten Schwellwert liegen (z.B. nur noch diejenigen Nachrichten, die über aufgetretene Ausnahmerebedingungen informieren).

5.1. Configurator

Der Configurator (*ConfigEditor*) ist ein Editor für Konfigurationen (siehe Abschnitt 4.4). Er ermöglicht das Erstellen, Speichern und Laden von Konfigurationen. Das Speichern und Laden geschieht über einen *Konfigurationsmanager* (*ConfigManager*). Der Konfigurationsmanager unterstützt verschiedene Formate für die Speicherung von Konfigurationen. Sie können einerseits als serialisierte Objekte (über den Java Serialisierungs-Mechanismus [9]) oder in ihrer XML-Repräsentation (siehe Abschnitt 4.4.2) abgelegt werden. Zusätzlich erlaubt der Konfigurationsmanager den Export eines Typsystems in XML-Schema Darstellung (siehe Abschnitt 4.3.2). Das Schema kann von externen Applikationen für die Validierung von Datei-Metadaten verwendet werden.

Über den Configurator lässt sich eine Konfiguration zur *Systemkonfiguration* bestimmen. Die Systemkonfiguration bezeichnet die vom System für die Verarbeitung von Dateien verwendete Konfiguration². Die Systemkonfiguration definiert ausserdem das aktuelle Typsystem für OBJECTSFS.

Abbildung 5.3 zeigt die Benutzeroberfläche des Configurators, welche sich in vier Bereiche aufteilt: In einem ersten Bereich (A) lässt sich die Typhierarchie des Typsystems durch hinzufügen von Subtypen und durch entfernen von Objekttypen editieren. Ein

²Sie wird in einer Datenbank abgelegt (siehe Abschnitt 6.4).

zweiter Bereich (B) steht für das Hinzufügen, Löschen und Bearbeiten von Elementen des selektierten Objekttyps zur Verfügung. In einem dritten Bereich (C) lassen sich Filter und Extraktoren für den selektierten Objekttyp definieren. Ein vierter Bereich schließlich (D) bietet die Möglichkeit, Extraktoren für das durch den aktuellen Elementpfad bezeichnete Element zu definieren.

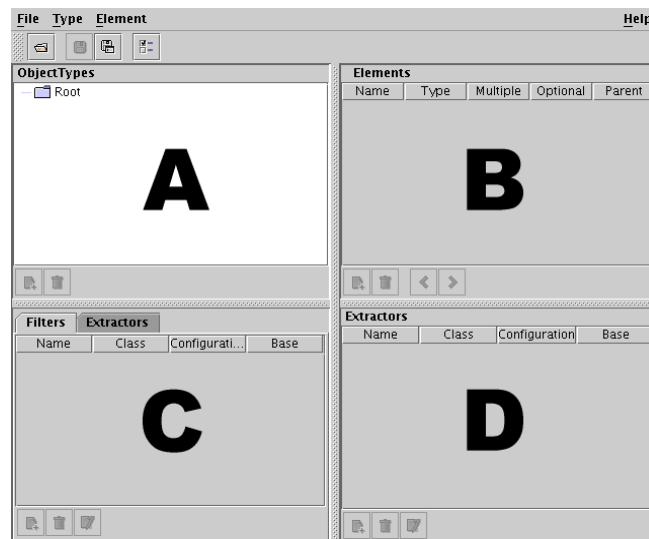


Abbildung 5.3.: Konfigurationseditor für das Erstellen, Speichern und Laden von Konfigurationen

5.1.1. Typhierarchie

Die Typhierarchie des Typsystems geht von einem Basistyp (**Root**) aus. **Root** lässt sich nicht aus der Hierarchie entfernen und jeder neu hinzugefügte Objekttyp ist ein direkter oder indirekter Subtyp dieses Basistyps.

Abbildung 5.4 zeigt ein Beispiel einer Konfiguration der Typhierarchie. (Das gezeigte Beispiel basiert auf einer stark vereinfachten Konfiguration, ausreichend für die Illustration der beschriebenen Konzepte. Umfangreichere Konfigurationen für OBJECTSFS werden in Kapitel 7 vorgestellt.)

Neben dem Basistyp (**Root**) sind in der Konfiguration aus Abbildung 5.4 sechs Objekttypen definiert: Ein Objekttyp **File** als Erweiterung des Basistyps **Root**, der erweitert wird durch den Objekttyp **Document**, welcher seinerseits durch **RTF** erweitert wird. Daneben gibt es je einen Objekttyp **Text** und **Person**, welche den Basistypen **Root** erweitern. Ein letzter Objekttyp **Letter** ist als Erweiterung von **Text** definiert.

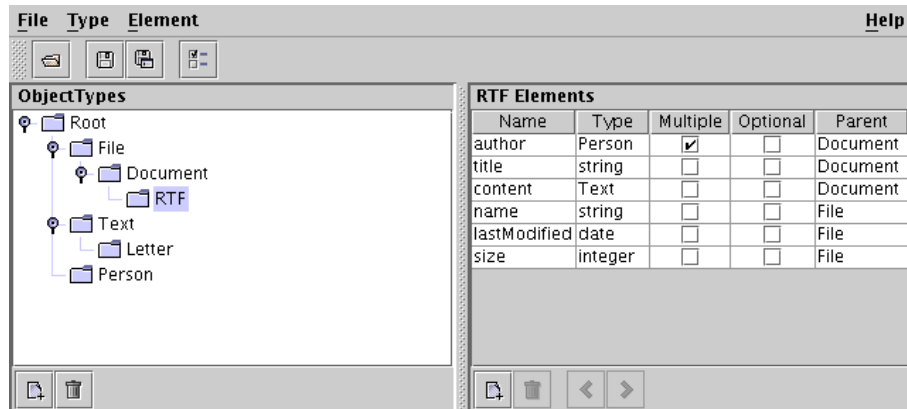


Abbildung 5.4.: Beispiel einer Konfiguration des Typsystems

5.1.2. Elemente

Für die Objekttypen in der Typhierarchie lassen sich gemäß dem in Kapitel 4 vorgestellten Objektmodell Elemente definieren. Abbildung 5.4 zeigt die Elemente, welche dem Objekttyp RTF zugeordnet sind.

Die Elemente **author** und **title** sind als Elemente von **Document** definiert, während die Elemente **name**, **lastModified** und **size** vom Objekttyp **File** geerbt sind. Das **Parent**-Feld der abgebildeten Tabelle zeigt den Objekttyp, für den ein Element definiert ist.

Das Element **author** in Abbildung 5.4 wurde als multipel (**Multiple**) markiert. Instanzen von **Document** können somit mehrere Felder für das **author** Element aufweisen (Dies entspricht dem Umstand, dass ein Dokument von mehreren Autoren stammen kann.).

Die Elemente eines Objekttyps lassen sich direkt über die Tabelle editieren. Die editierbaren Zellen sind Name, Typ (**Type**) und Multiplizität (**Multiple**) des Elements. Ein geerbtes Element lässt sich nicht editieren. Das Element **title** lässt sich beispielsweise bei selektiertem **Document** Objekttyp editieren, nicht aber das Element **name**. So ist sichergestellt, dass die Kontrolle über ein Element nur von einem Objekttyp (dem Objekttyp, der das Element definiert) ausgeht. Diese Einschränkung verhindert unerwünschte Nebeneffekte beim Ändern eines Elements und fördert so die Übersichtlichkeit der Konfiguration.

5.1.3. Filter

Filter lassen sich ähnlich wie Elemente einem Objekttyp zuordnen (über die in Abschnitt 4.4 beschriebenen *FilterInfo*). Abbildung 5.5 zeigt einen für den Objekttyp RTF definierten Filter.

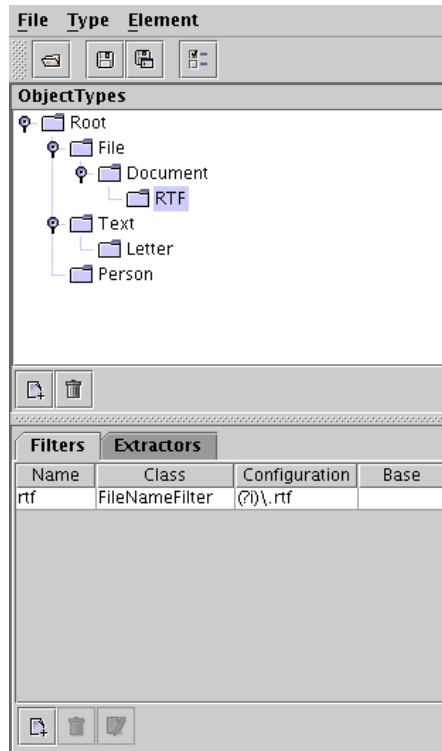


Abbildung 5.5.: Ein für einen Objekttyp definierter Filter

Für einen Filter werden dessen Name (**rtf**), die (Java-) Klasse, über die der Filter implementiert ist (*FileNameFilter*), eine String-Repräsentation der aktuellen Konfiguration (`((?i)\.rtf)`), sowie der Name eines eventuell für den Filter spezifizierten Basis-Extraktors (siehe Abschnitt 4.5) angezeigt.

Die Konfiguration lässt sich bei einem konfigurierbaren Filter über einen Dialog, wie er in Abbildung 5.6 gezeigt wird, vornehmen. Der Dialog zeigt eine kurze Beschreibung und eine grafische Komponente zur Konfiguration des Filters. Beim Betätigen des *Okay*-Buttons wird die neue Konfiguration gesetzt.

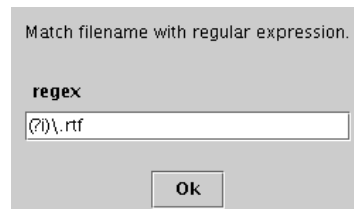


Abbildung 5.6.: Konfiguration eines Filters

Beim Dialog aus Abbildung 5.6 handelt es sich um einen Konfigurationsdialog für einen *FileNameFilter*. Dieser Filter wird in Abschnitt 6.1.2 mit allen anderen Filtern beschrieben, welche im Rahmen von OBJECTSFS implementiert wurden.

Die in Abbildung 5.6 gezeigte Konfiguration eines *FileNameFilter* besteht aus dem regulären Ausdruck `(?i)\.rtf`³. Der so konfigurierte Filter akzeptiert einen Kontext genau dann, wenn im Namen der Datei für die der Kontext steht, die Zeichenkette `.rtf` auftritt (unter beliebiger Verwendung von Gross- oder Kleinschreibung).

5.1.4. Extraktoren

Abbildung 5.7 stellt dar, wie über den Configurator Extraktoren für Objekttypen und Elemente definiert werden können (dies geschieht intern über die in Abschnitt 4.4 beschriebenen *ExtractorInfo*).

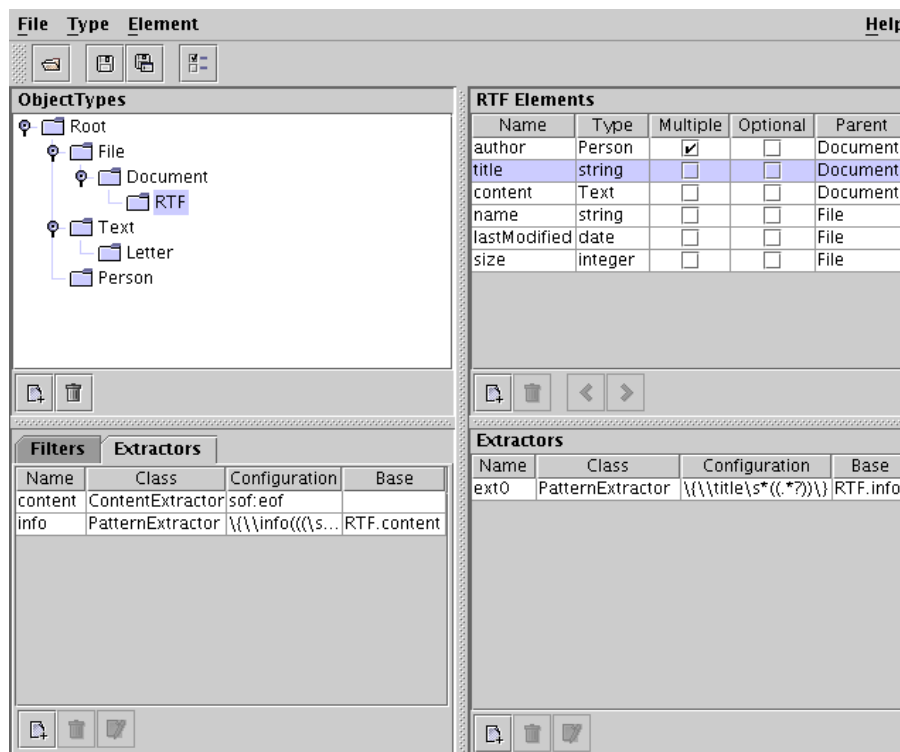


Abbildung 5.7.: Für Objekttypen und Elemente definierte Extraktoren

³OBJECTSFS verwendet die mit Java 1.4 eingeführten Klassen zur Verarbeitung von regulären Ausdrücken (Paket `java.util.regex`). Eine Zusammenfassung der gültigen Konstrukte für reguläre Ausdrücke findet sich in der entsprechenden Java API-Dokumentation [9].

Die Abbildung zeigt drei Extraktoren. Zwei davon sind für den Objekttyp `RTF` definiert (`content` und `info`), ein weiterer für das Element `RTF/title:string (ext0)`. Für Elemente mit Objekttyp (z.B. das durch den Pfad `RTF/author:Person` adressierte Element) können auch Extraktoren für Unterelemente (`RTF/author:Person/name:string`) definiert werden. Der Configurator erlaubt dazu das Navigieren in der Elementstruktur (über die zwei Buttons `<` und `>`), um die entsprechenden (Unter-) Elemente sichtbar zu machen.

Wie bei den Filtern wird für einen Extraktor neben seinem Namen und der (Java-) Klasse, über die er implementiert ist (`Class`), die aktuelle Konfiguration (`Configuration`) und der Basis-Extraktor angezeigt (`Base`). Die Konfiguration eines Extraktors verläuft über einen *Konfigurationsdialog* wie er in Abbildung 5.6 für einen Filter gezeigt ist.

Über die Angabe von Basis-Extraktoren lassen sich die Extraktoren kaskadieren (siehe Abschnitt 4.5). Im Beispiel aus Abbildung 5.7 ist eine Kaskade von drei Extraktoren zu sehen: Die Extraktion eines `title` Feldes eines Fileobjekts der Klasse `RTF` geschieht über den für das Element `RTF/title:string` definierten Extraktor (`ext0`). Der Basis-Extraktor von `ext0` ist als `RTF.info` angegeben. Der Eingabe-Kontext für den Extraktor `ext0` wird also vom Extraktor `info` bestimmt, der für den Objekttyp `RTF` definiert ist. Der Extraktor `info` wird seinerseits vom `content` Extraktor mit einem Eingabe-Kontext versorgt.

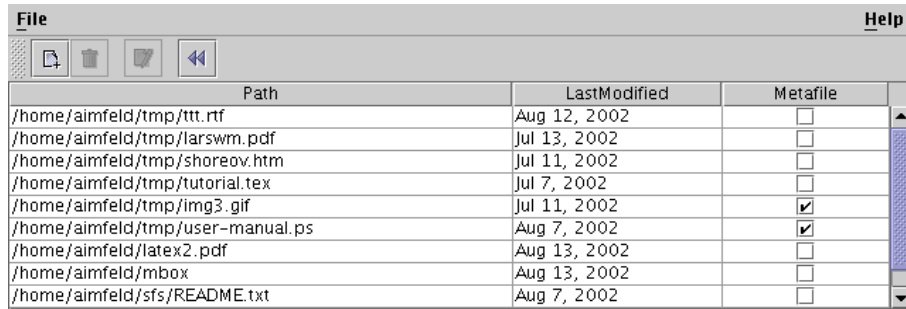
Die OBJECTSFS Implementation liefert analog zu den Filtern eine Reihe von Extraktoren für den Einsatz in einer Konfiguration. In Abschnitt 6.1.2 sind die implementierten Extraktoren mit einer kurzen Beschreibung ihrer Funktionsweise aufgelistet.

5.2. Filemanager

Der Filemanager, dessen Benutzeroberfläche in Abbildung 5.8 gezeigt ist, ermöglicht den Zugriff auf das Dateisystem von OBJECTSFS. Er listet alle Dateien auf, die sich momentan im System befinden, und zeigt für jede Datei das Datum der letzten Änderung (zum Zeitpunkt der Aufnahme). Ausserdem zeigt er für eine Datei an, ob für diese externe Metadaten existieren (gemäss den Ausführungen in Abschnitt 4.3.2).

Der Filemanager bietet drei Funktionen für die Manipulation der Dateiliste an:

- Hinzufügen einer Datei oder eines Verzeichnisses. Das Hinzufügen eines Verzeichnisses bewirkt, dass alle Dateien, die sich im Verzeichnis befinden, zum System hinzugefügt werden.
- Löschen einer Datei, die sich im System befindet. Die Datei wird nicht im darunterliegenden Dateisystem gelöscht, sie wird lediglich aus OBJECTSFS entfernt.
- Löschen (bzw. Entfernen) aller Dateien aus OBJECTSFS.



Path	LastModified	Metafile
/home/aimfeld/tmp/ttt.rtf	Aug 12, 2002	☐
/home/aimfeld/tmp/larswm.pdf	Jul 13, 2002	☐
/home/aimfeld/tmp/shoreov.htm	Jul 11, 2002	☐
/home/aimfeld/tmp/tutorial.tex	Jul 7, 2002	☐
/home/aimfeld/tmp/img3.gif	Jul 11, 2002	☒
/home/aimfeld/tmp/user-manual.ps	Aug 7, 2002	☒
/home/aimfeld/latex2.pdf	Aug 13, 2002	☐
/home/aimfeld/mbox	Aug 13, 2002	☐
/home/aimfeld/sfs/README.txt	Aug 7, 2002	☐

Abbildung 5.8.: Filemanager für OBJECTSFS

5.2.1. Editieren von Datei-Metadaten

Datei-Metadaten können über eine beliebige Applikation generiert oder durch den Benutzer angelegt und editiert werden. OBJECTSFS bietet im Rahmen des Filemanagers ebenfalls eine Möglichkeit für das Anlegen und das Editieren von Datei-Metadaten. Für jede Datei lässt sich dazu ein Editor starten, wie er in Abbildung 5.9 gezeigt ist.

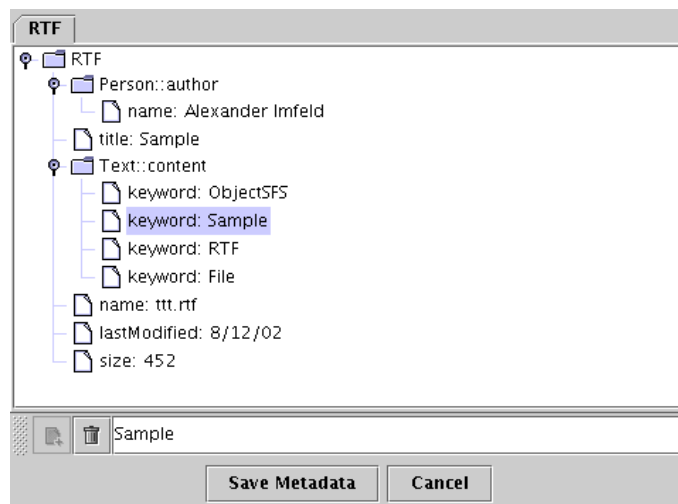


Abbildung 5.9.: Editieren von Datei-Metadaten

Der Editor zeigt das Fileobjekt (bzw. die Fileobjekte) für diese Datei in einer Baumstruktur und erlaubt das Hinzufügen und Löschen von Feldern. Die Werte von Feldern lassen sich zudem über ein Textfeld editieren.

Der *Save Metadata*-Button speichert die so erzeugten oder veränderten Metadaten in der Metadaten-Datei mit der Endung `_sfs.xml`. Die Metadaten werden im XML-Format

gespeichert, gemäss den in Abschnitt 4.3.2 aufgestellten Konventionen zur Repräsentation von Datei-Metadaten.

5.3. Objektbrowser

Für den Zugriff auf das Objektsystem von OBJECTSFS dient der Objektbrowser, der in Abbildung 5.10 zu sehen ist.

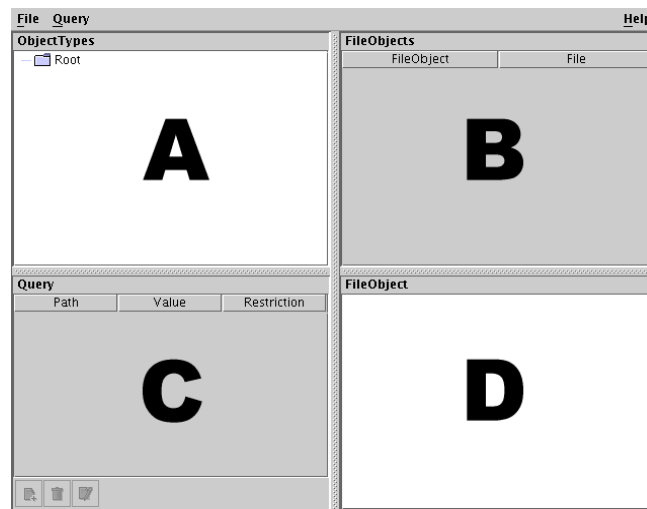


Abbildung 5.10.: Objektbrowser für OBJECTSFS

Die Benutzeroberfläche des Objektbrowser gliedert sich in vier Bereiche: Ein erster Bereich (A) zeigt die Typhierarchie der Systemkonfiguration. In einem zweiten Bereich wird eine Liste von Fileobjekten angezeigt. Die angezeigten Fileobjekte sind alle Instanzen des im Bereich A selektierten Objekttyp (bzw. Instanzen eines Subtyps dieses Objekttyps). Der Bereich C dient dem Erstellen einer Anfrage an das Objektsystem. Wurde eine Anfrage gestellt, so zeigt Bereich B die Fileobjekte, welche die in der Anfrage aufgestellten Bedingungen erfüllen. Der letzte Bereich (D) zeigt die Felder eines in Bereich B selektierten Fileobjekts in einer Baumstruktur. Die Baumstruktur entspricht der Elementstruktur des Objekttyps (bzw. der Klasse) des selektierten Fileobjekts.

5.3.1. Suchanfragen

Eine Suchanfrage (siehe Abschnitt 4.2.1) an das Objektsystem von OBJECTSFS geschieht immer im Zusammenhang mit dem im Objektbrowser selektierten Objekttyp. Abbildung 5.11 zeigt ein Beispiel einer Suchanfrage für den Objekttyp Document.

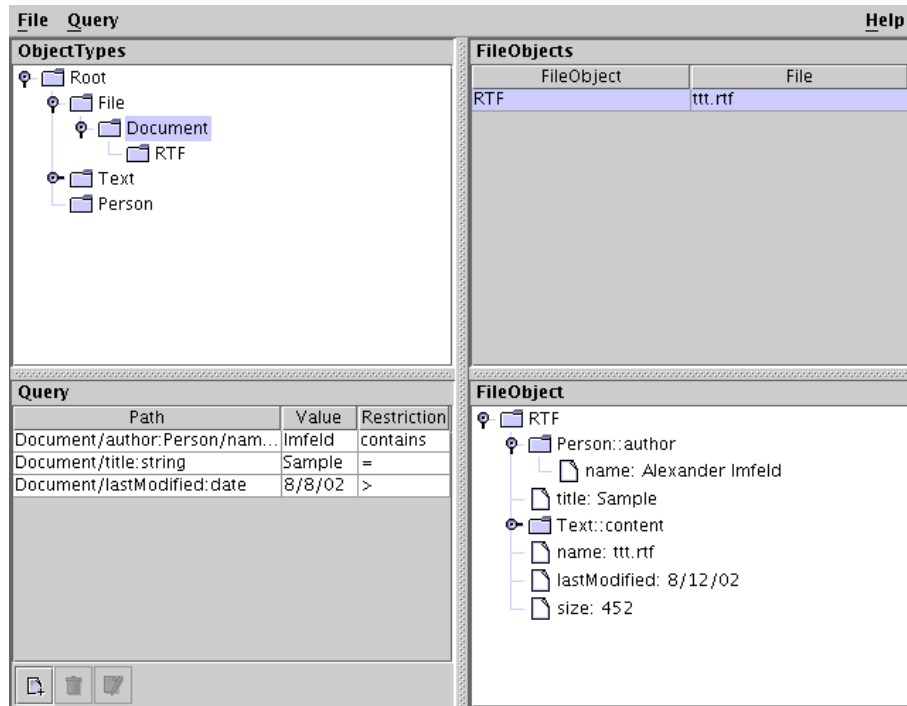


Abbildung 5.11.: Suchanfrage an das Objektsystem von OBJECTSFS

Die gezeigte Suchanfrage besteht aus drei Anfrage-Elementen: Einem Element zur Einschränkung des `Document/author:Person/name:string` Feldes, einem Element zur Einschränkung von `Document/title:string` und einem Element zur Einschränkung von `Document/lastModified:date`. Die Elementpfade zu den eingeschränkten Feldern sind in der Spalte `Path` zu sehen. In der Spalte `Value` sind die Werte dieser Felder spezifiziert (`Imfeld`, `Sample` und `8/8/02`⁴) während die letzte Spalte die Art der Restriktion (`Restriction`) angibt (= für *Gleichheit*, `contains` für *Enthalten*, `<` bzw. `>` für *Kleiner* bzw. *Grösser* und `~` für *Regulärer Ausdruck*).

Die Abbildung 5.11 zeigt auch das vom Objektsystem zurückgelieferte Fileobjekt. Es ist die einzige Instanz der selektierten Klasse (`Document`), die alle spezifizierten Restriktionen erfüllt.

⁴Die Darstellung eines Datums richtet sich nach den länderspezifischen Einstellungen des Betriebssystems.

5.4. Zusammenfassung

Eine Test-Applikation implementiert eine grafische Benutzeroberfläche, welche Zugang zu den drei Hauptkomponenten (Konfiguration, Dateisystem und Objektsystem) von OBJECTSFS bietet. Die Interaktion mit OBJECTSFS geschieht typischerweise in drei Phasen unter Verwendung dieser Hauptkomponenten: In einer ersten Phase wird eine Systemkonfiguration für OBJECTSFS über den Configurator erstellt. In einer zweiten Phase werden Dateien über den Filemanager zum Dateisystem von OBJECTSFS hinzugefügt, nachdem sie über einen in Abschnitt 6.2 genauer vorgestellten Mechanismus zu Fileobjekten verarbeitet worden sind, welche semantische Informationen über die Dateien enthalten. Die Fileobjekte werden im Objektsystem von OBJECTSFS gespeichert. Der Zugriff auf Dateien basierend auf den im Objektsystem abgelegten semantischen Informationen kann in einer dritten Phase über den Objektbrowser erfolgen.

6. Implementation

Dieses Kapitel behandelt ausgewählte Aspekte der Implementation von OBJECTSFS. Es geht vertieft auf einige bereits in Kapitel 4 und Kapitel 5 angesprochene Punkte ein.

6.1. Plugins

Um die einfache Erweiterbarkeit von OBJECTSFS zu gewährleisten, können dem System Filter und Extraktoren in Form von *Plugins* hinzugefügt werden. Ein Plugin für OBJECTSFS muss folgende Voraussetzungen erfüllen:

- Das Plugin ist eine Java-Klasse, welche das Interface *SFSPlugin* implementiert.
- Die Klasse liegt im Paket `ch.ethz.sfs.plugins` und ist über den Klassenpfad auffindbar.

Für Filter muss zusätzlich gelten:

- Die Klasse implementiert das Interface *SFSFilter*.

Und für Extraktoren:

- Die Klasse implementiert das Interface *SFSExtractor*.

6.1.1. Plugin Schnittstellen

Abbildung 6.1 zeigt die drei Interfaces (*SFSPlugin*, *SFSFilter* und *SFSExtractor*) mit ihren Schnittstellen in einem Klassendiagramm.

Die Methode *getDescription* dient dazu, eine kurze Beschreibung eines Plugins zu bekommen. Die Beschreibung wird beispielsweise in einem Dialog zur Konfiguration des Plugins dargestellt (siehe Abbildung 5.6).

Nicht jedes Plugin ist konfigurierbar (insbesondere gilt dies für einen *DefaultFilter* und einen *DefaultExtractor*). Ob ein Plugin konfigurierbar ist, wird über die Methode *isConfigurable* angezeigt. Konfigurierbare Plugins liefern über die Methode *getConfigPanel* eine grafische Komponente, über welche die aktuelle Konfiguration des Plugins angezeigt und editiert werden kann. Die zurückgelieferte Komponente wird ebenfalls in einem Dialog zur Konfiguration des Plugins (siehe Abbildung 5.6) verwendet.

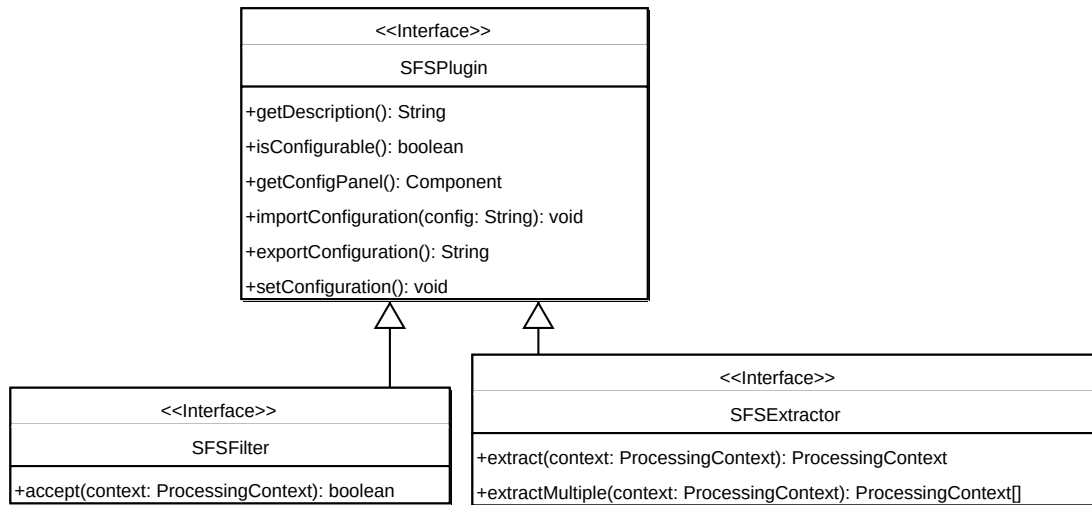


Abbildung 6.1.: Schnittstellen von Plugins

Zwei Methoden (*importConfiguration* und *exportConfiguration*) sind für den Import und den Export einer *Plugin-Konfiguration* aus einer String-Repräsentation bzw. in eine String-Repräsentation verantwortlich. Die String-Repräsentation der Plugin-Konfiguration wird zum Beispiel im Configurator (siehe Abschnitt 5.1) für Filter und Extraktoren angezeigt. Sie ist aber auch wichtig für die Darstellung einer Konfiguration im XML-Format (die String-Repräsentation einer Plugin-Konfiguration findet sich direkt in der XML Darstellung der entsprechenden Filter bzw. Extraktoren) und das Einlesen einer solchen Konfiguration (siehe Abschnitt 4.4).

Eine letzte Methode (*setConfiguration*) meldet einem Plugin den Abschluss der Konfiguration über die zurückgelieferte grafische Komponente (*getConfigPanel*). Das Plugin kann darauf die vorgenommene Konfiguration übernehmen. Die Methode wird beim Bestätigen einer Plugin-Konfiguration über den Konfigurationsdialog (siehe Abbildung 5.6) aufgerufen.

Ein Filter definiert eine zusätzliche Methode (*accept*), welche einen Kontext entgegennimmt und einen Wahrheitswert zurückliefert. Der Wahrheitswert besagt, ob der Filter den Kontext akzeptiert oder nicht (siehe Abschnitt 4.5.2).

Für Extraktoren sind zwei zusätzliche Methoden definiert (*extract* und *extractMultiple*). Die Methoden erzeugen zu einem Eingabe-Kontext einen Ausgabe-Kontext bzw. mehrere Ausgabe-Kontexte. Die Wahl der Methode hängt vom Element ab, das mit dem zu extrahierenden Feld verknüpft ist. Handelt es sich um ein multiples Element so wird *extractMultiple* verwendet, ansonsten kommt *extract* zum Zug (siehe Abschnitt 4.5.3).

6.1.2. Implementierte Plugins

Filter

Tabelle 6.1 listet alle mit OBJECTSFS implementierten Filter auf. Die Funktionsweise eines Filters geht aus einer kurzen Beschreibung hervor.

<i>DefaultFilter</i>	Standard-Filter. Akzeptiert keinen Kontext.
<i>FileNameFilter</i>	Akzeptiert einen Kontext auf Grund eines konfigurierbaren regulären Ausdrucks, welcher im Dateinamen der mit dem Kontext assoziierten Datei vorkommen muss.
<i>PatternFilter</i>	Akzeptiert einen Kontext auf Grund eines regulären Ausdrucks, welcher in der String-Repräsentation des Kontextbuffers vorkommen muss.

Tabelle 6.1.: Mit OBJECTSFS implementierte Filter

Die Anzahl der implementierten Filter ist sehr beschränkt. Die mit OBJECTSFS gesammelte Erfahrung hat allerdings gezeigt, dass die vorgestellten Filter in der Praxis ausreichen. Ein Fileobjekt für eine Datei kann in den meisten Fällen auf Grund des Dateinamens (bzw. der Endung des Dateinamens) klassifiziert werden (unter Umständen braucht es zusätzlich eine Untersuchung des Dateiinhalts auf das Auftreten eines Musters). Die dynamische Klassifikation von Fileobjekten geschieht zumeist über Muster-suche in einem Kontextbuffer.

Extraktoren

Die mit OBJECTSFS implementierten Extraktoren sind in Tabelle 6.2 mitsamt einer kurzen Beschreibung ihrer Funktionsweise aufgelistet.

Die implementierten Extraktoren wurden möglichst allgemein und generisch gehalten, um die Menge der Extraktoren überschaubar zu halten und ihre Einsatzmöglichkeiten so wenig wie möglich einzuschränken. Die meisten der implementierten Extraktoren sind deshalb unabhängig vom Dateityp, operieren auf einfachen Zeichenketten und können gut kombiniert werden. Nicht alle Dateien lassen sich allerdings effizient mit dateitypunabhängigen Extraktoren verarbeiten (siehe auch Kapitel 7). Aus diesem Grund sind einige Extraktoren für den Umgang speziell mit diesen Dateien entstanden (*PDFInfoExtractor*, *RTFTextExtractor*). Ausserdem erlaubt OBJECTSFS über einen speziellen Extraktor, einen *Executer*, die Ausführung externer Programme. Der *Executer* eignet sich beispielsweise für die Dateikonvertierung in ein Text-Format (Man denke an die vielen vor allem aus der Unix-Welt stammenden *xyz2txt* Programme). Ist der Dateiinhalt einmal als Text verfügbar, so lässt er sich leichter mit den anderen Extraktoren (z.B. mit einem *PatternExtractor*) verarbeiten.

<i>DefaultExtractor</i>	Standard-Extraktor. Gibt den Eingabe-Kontext unverändert als Ausgabe-Kontext zurück.
<i>ContentExtractor</i>	Extrahiert den Inhalt der Datei, welche mit dem Eingabe-Kontext assoziiert ist.
<i>PatternExtractor</i>	Extrahiert Daten gemäss einem regulären Ausdruck. Die extrahierten Daten entsprechen der Subsequenz der Eingabe-Daten, die mit der ersten im regulären Ausdruck spezifizierten <i>capturing group</i> ((...)) übereinstimmen (siehe dazu API-Dokumentation zu <code>java.util.regex.Pattern</code> [9]).
<i>Splitter</i>	Teilt Daten an den Stellen, die mit dem konfigurierten regulären Ausdruck übereinstimmen.
<i>Replacer</i>	Ersetzt über einen regulären Ausdruck gefundene Daten durch einen String.
<i>FileAttributeExtractor</i>	Extrahiert ein Dateiattribut der mit dem Eingabe-Kontext assoziierten Datei. Unterstützte Attribute sind der Dateiname, das Datum der letzten Änderung und die Dateigrösse.
<i>DateExtractor</i>	Extrahiert ein Datum nach einem vorgegeben Muster (Beispiel eines solchen Musters: <code>yy.MM.dd</code>).
<i>NumberExtractor</i>	Konvertiert Bytes in eine Zahl.
<i>PDFInfoExtractor</i>	Extrahiert aus der (PDF-)Datei, welche mit dem Eingabe-Kontext assoziiert ist, das Info-Objekt, welches die Metadaten der PDF-Datei enthält.
<i>RTFTextExtractor</i>	Extrahiert den Text aus der (RTF-)Datei, welche mit dem Eingabe-Kontext assoziiert ist.
<i>Executer</i>	Führt ein Programm aus und verwendet die Ausgabe des Programms als Ausgabe-Kontext. Die Konfiguration umfasst den Namen des Programms und das Argument für den Programmaufruf (Datei des Kontextes oder Kontextdaten).

Tabelle 6.2.: Mit OBJECTSFS implementierte Extraktoren

6.1.3. Laden von Plugins

Der Zugriff auf Plugin-Implementationen (d.h. Filter und Extraktoren) geschieht über einen *Pluginmanager* (*PluginManager*). Der Pluginmanager veranlasst dazu einen *Plugin-Lader* (*PluginLoader*), alle Klassen aus dem Plugin-Paket (`ch.ethz.sfs.plugins`) zu laden, welche die geforderten Interfaces (*SFSPlugin* und *SFSFilter* bzw.

SFSExtractor) implementieren. Die Plugins stehen danach für die Verwendung im System zur Verfügung.

6.2. Aufnahme von Dateien

Bevor eine Datei zum Dateisystem von OBJECTSFS hinzugefügt wird (z.B. über den in Abschnitt 5.2 vorgestellten Filemanager), erfolgt eine Verarbeitung dieser Datei. Die Verarbeitung wird von einem *FileProcessor* kontrolliert. Der *FileProcessor* prüft als erstes das Vorhandensein einer Metadaten-Datei (siehe Abschnitt 4.3.2). Dazu sucht der *FileProcessor* im selben Verzeichnis der zu verarbeitenden Datei eine bis auf die Endung (`_sfs.xml`) gleich benannte Datei. Findet er eine solche Datei, so erzeugt er die Metadaten-Objekte aus der dort vorgefundenen XML-Repräsentation der Fileobjekte. Andernfalls führt er eine Verarbeitung der Datei (gemäss den Ausführungen aus Abschnitt 4.5) auf Grund der Systemkonfiguration durch.

Die Verarbeitung besteht kurz zusammengefasst aus den folgenden Schritten:

1. Erstellen eines neuen (leeren) Kontextes (Initial-Kontext) für die Datei.
2. Klassifizieren des initialen Kontextes.
3. Für jeden (bei der Klassifizierung) gefunden Objekttyp wird ein Fileobjekt erstellt.
4. Die Felder der erstellten Fileobjekte werden (rekursiv) extrahiert. Subobjekte werden vor der Extraktion ihrer Felder dynamisch klassifiziert.

Die so entstandenen Fileobjekte (aus der Metadaten-Datei oder aus der beschriebenen Verarbeitung der Datei) werden vom *FileProcessor* zum Objektsystem von OBJECTSFS hinzugefügt. Schliesslich informiert der *FileProcessor* das Dateisystem von OBJECTSFS, welches die verarbeitete Datei aufnimmt.

Die vom *FileProcessor* durchgeführten Schritte laufen im Hintergrund ab: Er ist als *Thread* implementiert. Die Priorität des *FileProcessor* lässt sich in den Konfigurationseinstellungen (siehe Abschnitt 4.4) angeben. Die Einstellungen der Systemkonfiguration bestimmen somit die Priorität eines neu gestarteten *FileProcessor* und damit auch die Priorität der Verarbeitung von Dateien.

6.3. Konsistenz

OBJECTSFS muss die Aktualität und die Konsistenz einerseits zwischen den Dateien in OBJECTSFS (bzw. internen Repräsentationen von Dateien) und den dazugehörigen semantischen Informationen (Fileobjekte), andererseits zwischen diesen Dateien und den Dateien im darunterliegenden Dateisystem gewährleisten. In Abbildung 6.2 sind die verschiedenen Ebenen abgebildet, die es dabei von OBJECTSFS zu berücksichtigen gilt.

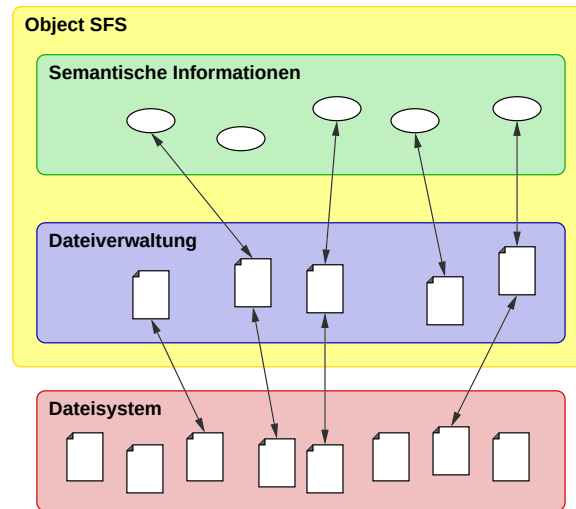


Abbildung 6.2.: Die verschiedenen Ebenen, auf denen Konsistenz zu gewährleisten ist

Die folgenden Ereignisse betreffen die Konsistenz zwischen den Dateien in OBJECTSFS und den semantischen Informationen über diese Dateien (bzw. die Konsistenz zwischen dem OBJECTSFS-Dateisystem und dem OBJECTSFS-Objektsystem):

- Hinzufügen von Dateien zu OBJECTSFS.
- Entfernen von Dateien aus OBJECTSFS.

Da die Änderungen über OBJECTSFS geschehen, ist es einfach, die Konsistenz zu gewährleisten: Im ersten Fall werden die semantischen Informationen gemäß Verarbeitungsinformationen in der Systemkonfiguration extrahiert (bzw. aus vorhandenen Datei-Metadaten übernommen) und als Fileobjekte zum Objektsystem hinzugefügt. Im zweiten Fall werden die Fileobjekte, die mit den entfernten Dateien assoziiert sind, aus dem Objektsystem gelöscht.

Die nächsten Ereignisse betreffen die Konsistenz zwischen den Dateien in OBJECTSFS und den Dateien im eigentlichen Dateisystem:

- Änderung einer Datei im Dateisystem.
- Löschen einer Datei im Dateisystem.
- Verschieben einer Datei im Dateisystem.

Die Ereignisse sind nur dann interessant, wenn es sich bei den betroffenen Dateien um Dateien handelt, die vorgängig in OBJECTSFS aufgenommen wurden.

Da die Änderungen nicht über OBJECTSFS geschehen, muss OBJECTSFS periodisch die Konsistenz zwischen den Dateien in OBJECTSFS und ihrem Gegenstück im Dateisystem prüfen. Dies geschieht über einen Thread (*FileObserver*), welcher in regelmässigen Abständen aktiv wird und die Existenz und die Änderungsdaten der Dateien überprüft (Die Zeit zwischen zwei aktiven Phasen lässt sich in den in Abschnitt 4.4 vorgestellten Konfigurationseinstellungen angeben. Der *FileObserver* bezieht den Wert aus der aktuellen Systemkonfiguration.).

Hat sich eine Datei verändert, so wird sie neu verarbeitet. Der *FileObserver* kann Änderungen einer Datei feststellen, indem er das Fileattribut, welches das Datum der letzten Änderung angibt, mit dem Datum der letzten Änderung vergleicht, das in der internen Repräsentation der Datei gespeichert ist (siehe Abschnitt 4.3). Letzteres beschreibt das Datum der letzten Änderung zum Zeitpunkt der Aufnahme ins System.

Wurde eine Datei gelöscht oder verschoben, so wird sie aus dem Dateisystem von OBJECTSFS entfernt¹.

Die folgenden Ereignisse werden ebenfalls vom *FileObserver*-Thread bemerkt und bewirken eine erneute Verarbeitung der Datei durch den *FileProcessor* (siehe Abschnitt 6.2):

- Erstellen von externen Metadaten zu einer Datei.
- Änderung der externen Metadaten zu einer Datei.
- Löschen von externen Metadaten zu einer Datei.

Die Überprüfung der extern spezifizierten Datei-Metadaten durch den *FileObserver* sichert die Konsistenz zwischen diesen externen Angaben und den intern dafür erstellten Fileobjekten. Er verwendet dazu das in der internen Repräsentation einer Datei gespeicherte Datum der Aufnahme dieser Datei. Wurden extern spezifizierte Datei-Metadaten nach der Aufnahme der eigentlichen Datei geändert oder erzeugt (kann über einen Vergleich mit dem entsprechenden Fileattribut der Metadaten-Datei festgestellt werden), so ist die Konsistenz der Daten nicht mehr sichergestellt und der *FileProcessor* muss die entsprechende Datei erneut verarbeiten. Da die interne Repräsentation einer Datei das Vorhandensein von extern spezifizierten Metadaten zum Zeitpunkt der Aufnahme festhält, kann auch das Löschen (oder Verschieben) einer Metadaten-Datei vom *FileObserver* erkannt werden. Auch in diesem Fall löst der *FileObserver* eine Neuverarbeitung der Datei durch den *FileProcessor* aus.

¹Um festzustellen, dass eine Datei verschoben und nicht gelöscht wurde, müsste OBJECTSFS das ganze Dateisystem nach der Datei durchsuchen. Das Verschieben einer Datei im Dateisystem wird daher aus Effizienzgründen gleichbehandelt wie das Löschen einer Datei im Dateisystem.

6.3.1. Änderung der Systemkonfiguration

Die Systemkonfiguration definiert das Typsystem, das den Aufbau der Fileobjekte bestimmt. Eine Änderung der Systemkonfiguration und damit des Typsystem bewirkt eine Inkonsistenz zwischen den in OBJECTSFS enthaltenen Fileobjekten und diesem Typsystem.

Es bieten sich drei Strategien an, auf eine Änderung der Systemkonfiguration zu reagieren:

1. Löschen aller Fileobjekte und Entfernen aller Dateien aus OBJECTSFS.
2. Erneute Verarbeitung (über den *FileProcessor*) aller enthaltenen Dateien und damit implizite Aktualisierung der Fileobjekte.
3. Vergleich der neuen mit der alten Systemkonfiguration. Nur von den Unterschieden betroffene Dateien neu verarbeiten.

Jede der aufgeführten Strategien bietet Vor- und Nachteile: Die erste Variante ist sicher die effizienteste, entspricht jedoch nicht dem vom Benutzer erwarteten Systemverhalten. Die zweite Variante ist ein *Brute-Force*-Ansatz und wenig effizient. Auf der anderen Seite ist sie weniger fehleranfällig als die dritte Variante und für völlig verschiedene Systemkonfigurationen auch gleich effizient. Für die dritte Variante gilt, dass sich der Aufwand für die Analyse zweier Konfigurationen nicht immer lohnt, unter Umständen allerdings (bei ähnlichen Konfigurationen bzw. Konfigurationen, die voneinander *abstammen*) erfordert sie weniger Aufwand bei der Neuverarbeitung von Dateien.

Da es sich bei OBJECTSFS um einen Prototyp handelt und die Effizienz des Systems von zweitrangiger Bedeutung ist, wird in OBJECTSFS die zweite Strategie verfolgt. Eine Anpassung des Verhaltens gemäss der dritten Strategie sollte allerdings ohne grösseren Aufwand möglich sein².

6.4. Persistenz

Das Dateisystem und das Objektsystem von OBJECTSFS, sowie die Konfigurationen (speziell die Systemkonfiguration) müssen persistent gemacht werden.

Konfigurationen können über den Configurator in verschiedenen Formaten in Dateien gespeichert werden (siehe Abschnitt 5.1). Die Systemkonfiguration, das Filesystem und das Objektsystem werden in einer Datenbank abgelegt.

²Ein Ansatz wäre das Verwenden von Zeitstempeln und eindeutigen Bezeichnern für Elemente einer Konfiguration (z.B. für Objekttypen oder für *FilterInfo*). Änderungen der Konfigurations-Elemente könnten so über die Zeitstempel festgestellt werden, welche den Zeitpunkt der letzten Änderung derselben festhalten (z.B. hätte die Änderung einer Filter-Konfiguration die Anpassung des Zeitstempels im entsprechenden *FilterInfo*, *ElementInfo*, *ObjectInfo* und *InfoSystem* zur Folge).

Die Wahl der Datenbank fiel auf die Open-Source-Software *Ozone* [14]. Ozone ist eine in Java implementierte objektorientierte Datenbank. (Java-) Objekte können direkt in der Datenbank abgelegt werden, was eine nahtlose und unkomplizierte Ergänzung von OBJECTSFS um einen Persistenzmechanismus für Filesystem, Objektsystem und Systemkonfiguration ermöglicht. Es ist keine objekt-relationale Abbildung von OBJECTSFS-Komponenten erforderlich. Der Zugriff auf die persistenten Komponenten kann ausserdem statt über eine Datenbankabfragesprache (wie SQL oder OQL) über normale Java-Methodenaufrufe erfolgen.

In Abbildung 6.3 ist das Prinzip der Kommunikation mit Ozone-Datenbank-Objekten grafisch dargestellt. Es folgt eine kurze Erklärung dieser Abbildung. Für detailliertere Ausführungen sei auf die Dokumentation von Ozone verwiesen [14].

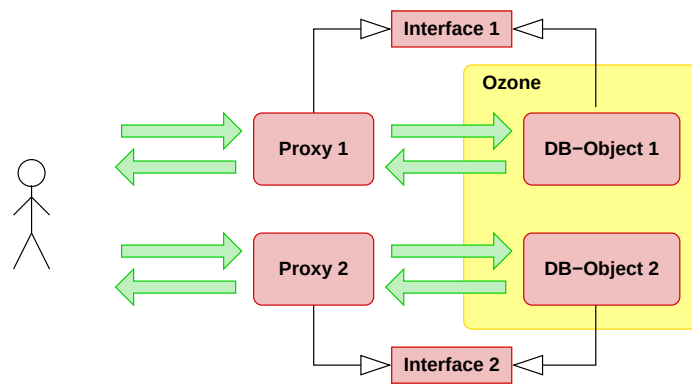


Abbildung 6.3.: Kommunikation mit Ozone-Datenbank-Objekten über Proxy-Objekte

Die Ozone-Datenbank läuft in einer separaten JVM (*Java Virtual Machine*). Der Zugriff auf Objekte in Ozone geschieht über Methoden, welche in einem (Java-) Interface definiert sind. Für jedes Datenbank-Objekt (Filesystem, Objektsystem, Konfiguration) ist ein solches Interface vorhanden. Die Methoden des Interfaces werden vom jeweiligen Datenbank-Objekt implementiert.

Ozone generiert zu den gespeicherten Objekten sogenannte *Proxy-Objekte* mit gleicher Signatur. Ein Proxy-Objekt leitet einen Methodenaufruf auf der Client-Seite (aus OBJECTSFS) an die Datenbank weiter, wo er vom Datenbank-Objekt ausgeführt wird. Das Resultat gelangt zurück auf die Client-Seite zum Proxy-Objekt, von wo aus es dem Aufrufer der Methode zurückgegeben wird. Abbildung 6.3 stellt diese Kommunikation mit Datenbank-Objekten über Proxies dar.

Das Filesystem und das Objektsystem von OBJECTSFS werden vollständig in Ozone als Datenbank-Objekte gespeichert. Der Zugriff auf von OBJECTSFS verwaltete Dateien und Fileobjekte geschieht innerhalb der Ozone Datenbank (innerhalb der JVM von Ozone). Auf diese Weise wird der Kommunikationsaufwand zwischen Client (OBJECTSFS) und der Ozone-Datenbank minimiert.

6.5. Zusammenfassung

Über die Implementation von Plugins lässt sich OBJECTSFS einfach um neue Filter und Extraktoren ergänzen. Die Plugin-Architektur erlaubt die flexible Erweiterung des Systems.

Dateien werden von einem *FileProcessor* verarbeitet, welcher das Vorhandensein von extern spezifizierten Metadaten prüft. Er sorgt dafür, dass Fileobjekte zur Beschreibung der Dateien erstellt und zu OBJECTSFS hinzugefügt werden.

Ein *FileObserver* ist verantwortlich für die Konsistenz zwischen den in OBJECTSFS aufgenommenen Dateien (bzw. deren internen Repräsentation) und den Dateien im darunterliegenden Dateisystem bzw. den dazu erstellten semantischen Informationen (Fileobjekten). Zusätzlich kümmert sich der *FileObserver* um die Konsistenz zwischen intern gespeicherten semantischen Informationen und extern spezifizierten Datei-Metadaten.

Die Persistenz der drei Hauptkomponenten von OBJECTSFS wird von Ozone gewährleistet. Ozone ist eine in Java implementierte objektorientierte Datenbank und lässt sich dadurch nahtlos mit OBJECTSFS verbinden.

7. Konfigurationen

Konfigurationen für OBJECTSFS sind von zentraler Bedeutung für den Einsatz des Systems. Die Effizienz, sowie die Quantität und die Qualität der Informationsausbeute von OBJECTSFS hängt von der Güte der verwendeten Konfiguration ab.

Verschiedene Umgebungen und Einsatzzwecke erfordern eine speziell angepasste Konfiguration. Die Kenntnis verwendeter Dateitypen spielt eine grosse Rolle beim Erstellen einer *optimalen* Konfiguration. Insofern ist eine Konfiguration nicht dateitypunabhängig. Ein geschickter Aufbau des Typsystems fördert allerdings eine dateitypunabhängige Sicht auf die erzeugten Fileobjekte. Dies geschieht im Wesentlichen durch das Definieren von Supertypen (z.B. `Document`), welche die gemeinsamen Eigenschaften der (dateityp-spezifischen) Subtypen (z.B. `RTF`) zusammenfassen.

Im Folgenden werden zwei mögliche Ansätze für die Konfiguration von OBJECTSFS vorgestellt. Die beschriebenen Beispielkonfigurationen zeigen die Möglichkeiten, aber auch die Grenzen des Systems auf. Sie können leicht angepasst und als Basis für eigene Erweiterungen verwendet werden.

Die zwei in den Beispielkonfigurationen verwendeten Ansätze können einfach in einer einzigen Konfiguration vereint werden. Sie werden hier getrennt behandelt, damit die ihnen zu Grunde liegenden Konzepte besser zum Ausdruck kommen.

7.1. Extraktion vorhandener Metadaten

Viele Dateitypen bieten die Möglichkeit, Metadaten in den Dateidaten zu integrieren (Diese Metadaten werden applikationsspezifisch angelegt und sind nicht mit den in Abschnitt 4.3.2 vorgestellten Datei-Metadaten zu verwechseln.). Die folgende Aufzählung zeigt einige Beispiele solcher Dateitypen, mitsamt den unterstützten Metadaten¹:

- **PDF** (*Portable Document Format* [16]): Unterstützt die Angabe von Metadaten in einem speziellen PDF-Objekt, dem *Info Dictionary*. Die Metadaten umfassen unter anderem Informationen zu Autor (`/Author (...)`), Titel (`/Title (...)`), Thema (`/Subject (...)`) und Schlüsselwörtern (`/Keywords (...)`). Die Angaben lassen sich über einen PDF-Viewer (z.B. Acrobat Reader) einsehen.

¹Eine umfassende Referenz zu Dateiformaten findet sich unter [25].

- **RTF** (*Rich Text Format* [18]): Metadaten werden in einer *Information Group* gespeichert (`{\info ...}`). RTF unterstützt ähnliche Metadaten-Angaben wie PDF. Die Informationen werden über spezielle Konstrukte dargestellt, die jeweils durch ein spezifisches Kontrollwort eingeleitet sind (`{\author ...}`, `{\title ...}`, `{\subject ...}`, `{\keywords ...}` usw.).
- **HTML** (*HyperText Markup Language* [19]): HTML erlaubt die Integration von Metadaten im Header der Datei (`<head> ...</head>`). Die Metadaten werden in einem speziellen Tag (`<meta ...>`) angegeben. So lassen sich Informationen zu Autor (`author`) und Schlüsselwörtern (`keywords`) in HTML integrieren. Auch eine Beschreibung des Inhalts ist möglich (`description`). Ausserdem findet sich im Header üblicherweise ein `title`-Tag, das den Titel der entsprechenden HTML-Seite enthält.
- **MP3** (*MPEG Audio Layer 3* [11]): Über ID3-Tags [8] kann eine MP3-Datei um einfache Informationen zum enthaltenen Musikstück ergänzt werden. Es sind dies Angaben zum Namen des Künstlers und des Titels des Stücks. Zusätzlich kann das Album, von dem das Stück stammt, und das Musikgenre spezifiziert werden.

Daneben gibt es textbasierte Dateien, die eine einfache und wohldefinierte Struktur aufweisen. Die Extraktion von semantischen Informationen aus dem Dateiinhalt wird dadurch stark vereinfacht. Die angesprochenen Dateien sind oft über spezielle Tags oder Schlüsselwörter strukturiert oder enthalten einen Header, der den Dateiinhalt beschreibt:

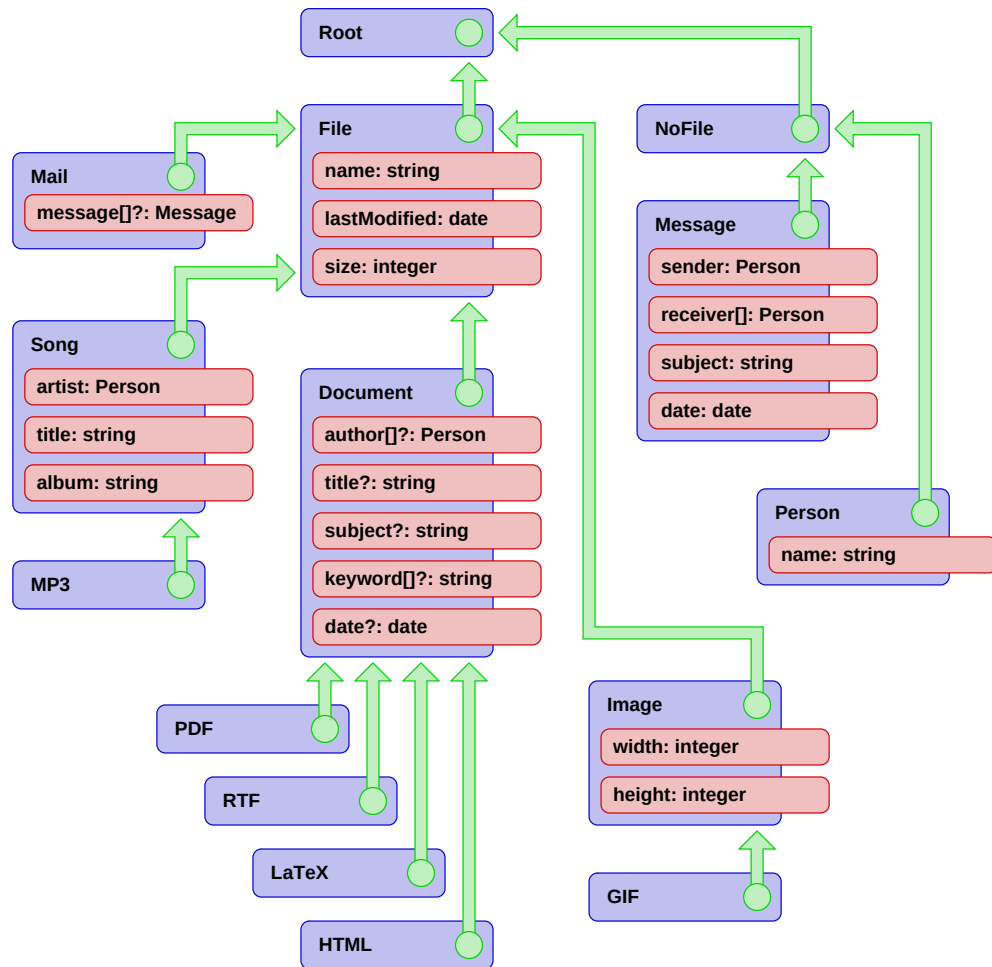
- **Mail-Datei**: Eine Mail-Datei enthält Mail-Nachrichten, deren Header eine standardisierte Struktur aufweist. Er besteht aus definierten Schlüsselwörtern (`From`, `To`, `Subject`, `Date` usw.) und den entsprechenden Werten.
- **L^AT_EX-Datei**: Sie enthält spezielle Befehle zur Beschreibung der Dokumentenstruktur (`\chapter{...}`, `\section{...}` usw.). Zusätzlich sind in L^AT_EX Befehle für die Angabe von Titel, Autor und Datum definiert (`\title{...}`, `\author{...}` und `\date{...}`).
- **Sourcecode**: Sourcecode (Java, C, C++ usw.) besteht aus Schlüsselwörtern, die durch die entsprechende Programmiersprache bestimmt sind. Dies ermöglicht eine einfache Analyse der Struktur des Sourcecodes.

Die Liste der Beispiele liesse sich ohne weiteres fortsetzen².

7.1.1. Beispielkonfiguration

Ein vollständiges Beispiel einer Konfiguration zur Extraktion von applikationsspezifischen Metadaten aus den Dateidaten findet sich in Anhang C.1. Die Konfiguration trägt den Namen *meta* und liegt im XML-Format (siehe Abschnitt 4.4.2) vor.

²Adressbuch-Dateien, vCards, BibT_EX-Dateien, Konfigurationsdateien usw.

Abbildung 7.1.: Typsystem der Beispielkonfiguration *meta*

Die Konfiguration (Typsystem und Verarbeitungsinformationen) wird im Folgenden kurz vorgestellt, um danach ihre Leistungsfähigkeit im praktischen Einsatz zu diskutieren.

Typsystem

Abbildung 7.1 zeigt das Typsystem der Beispielkonfiguration in einer grafischen Darstellung (wie sie in Abschnitt 4.1.1 eingeführt wurde).

Die Konfiguration *meta* definiert vier zentrale Objekttypen: **Document**, **Mail**, **Song** und **Image**. Diese Typen definieren die Elemente, die von ihren Subtypen geerbt werden. Die Subtypen (**PDF**, **RTF**, **LaTeX**, **HTML**, **MP3** usw.) verkörpern konkrete Dateitypen. Sie

definieren keine zusätzlichen Elemente. Ihr Zweck ist es, dateitypspezifische Verarbeitung zu ermöglichen (siehe nächster Abschnitt).

Die vier oben erwähnten zentralen Objekttypen (**Document**, **Mail**, **Song** und **Image**) bieten eine dateitypunabhängige Sicht auf Instanzen ihrer Subtypen. Sie sind von **File** abgeleitet und erben die dort definierten Elemente. **File** definiert Elemente für einige gebräuchliche Fileattribute (**name**, **lastModified** und **size**).

Neben **File** gibt es einen weiteren von **Root** abgeleiteten Objekttyp: **NoFile**. Dieser Objekttyp dient der Abgrenzung zwischen Fileobjekten, welche ganze Dateien beschreiben (z.B. **Document**), und Fileobjekten, welche Dateiteile (bzw. Entitäten, die in Dateien vorkommen) verkörpern (z.B. **Message**). Erstere sind Instanzen von **File** oder einer Subklasse davon, während letztere Instanzen von Subklassen von **NoFile** sind.

NoFile wird erweitert durch **Person** und **Message**. Eine Instanz von **Message** dient der Beschreibung von E-Mail Nachrichten (Element **message** in **Mail**). Zur Beschreibung von Personen (über ihren Namen) existiert der Typ **Person** mit dem Element **name**. Die Beschreibung von Personen hätte auch über ein Element mit Typ **string** im entsprechenden Objekttyp realisiert werden können (z.B. Typ **string** für das Element **author** in **Document**). Ein separater Objekttyp **Person** erlaubt allerdings eine spätere Suche nach Personen (Instanzen von **Person**) über eben diesen Objekttyp. Die Suche ist dabei unabhängig vom Objekttyp des Fileobjekts, in dem die Person als Feld auftritt. Man stelle sich vor, **Document** würde ein Element **author** mit Typ **string** definieren. Die Suche nach einer Person über ihren Namen könnte so nur über die Suche nach einem **Document**-Fileobjekt (oder einem anderen Fileobjekt, das ein Feld zur Aufnahme eines Personennamens besitzt) erfolgen, das die gewünschte Restriktion auf dem **author**-Feld erfüllt (siehe Abschnitt 4.2.1). Tritt die Person (bzw. ihr Name) beispielsweise auch als Sender einer E-Mail-Nachricht auf (**sender** in **Message** mit fiktivem Typ **string**), so beeinflusst das das Resultat der oben beschriebenen Suche nicht. Die Suche kann nur Instanzen des spezifizierten Objekttyps (hier **Document**) zurückliefern.

Verarbeitungsinformationen

Instanzen der Objekttypen **PDF**, **RTF**, **LaTeX**, **HTML**, **MP3** und **GIF** sind für die Beschreibung von Dateien gleichnamigen Dateityps vorgesehen. Für diese Objekttypen sind Filter definiert, die eine Datei auf Grund ihres Namens klassifizieren (genau genommen wird der bei der Verarbeitung der Datei angelegte Initial-Kontext klassifiziert. Siehe auch Abschnitt 4.5.2). Dies geschieht über *FileNameFilter*, die den Namen der Datei auf eine bestimmte Endung hin untersuchen. Der Filter für den Objekttyp **PDF** ist beispielsweise mit dem regulären Ausdruck `(?i)\.pdf` (siehe Java API-Dokumentation zum Paket `java.util.regex`) konfiguriert. Die Verarbeitung stellt so sicher, dass für eine Datei die auf `.pdf` endet, ein Fileobjekt der Klasse **PDF** angelegt wird. Analoges gilt für die anderen oben erwähnten Objekttypen.

Eine E-Mail-Datei kann in der Regel nicht auf Grund des Dateinamens klassifiziert werden. Die Klassifikation geschieht daher auf Grund des Dateiinhalts. Dazu ist für den Objekttyp `Mail` ein *PatternFilter* definiert, welcher einen ebenfalls für `Mail` definierten Extraktor als Basis verwendet. Beim Extraktor handelt es sich um einen *ContentExtractor*, welcher den Dateiinhalt extrahiert und im Kontextbuffer seines Ausgabe-Kontextes verfügbar macht. Der *PatternFilter* sucht in diesen Kontext-Daten nach einem bestimmten Muster. Er ist so konfiguriert, dass er E-Mail-Dateien an einem Schlüsselwort (`Message-ID:`) erkennt, das in jedem Header einer E-Mail-Nachricht vorhanden sein muss.

Jede Datei kann mindestens zu einem `File` klassifiziert werden. Dies wird durch den für `File` definierten Filter sichergestellt. Der Filter akzeptiert beliebige Dateinamen.

Ist eine Datei einmal klassifiziert und ein Fileobjekt (oder mehrere Fileobjekte) zur Beschreibung dieser Datei angelegt, so beginnt die Extraktion der Felder gemäss den in der Konfiguration spezifizierten Extraktoren.

Jedes Feld für ein in `File` definiertes Element wird über einen eben für dieses Element definierten Extraktor extrahiert. Dies geschieht über einen *FileAttributeExtractor*.

Die Objekttypen `PDF`, `RTF`, `LaTeX`, `HTML`, `MP3` und `GIF` definieren Extraktoren für die von `Document`, `Song` bzw. `Image` geerbten Elemente. Die entsprechenden Felder werden also typspezifisch extrahiert: Das Feld für das Element `title` wird für ein Fileobjekt der Klasse `RTF` anders extrahiert als für ein Fileobjekt der Klasse `PDF`.

Die Extraktion von Feldern verläuft in der Regel nach dem folgenden Prinzip: Ein für die Klasse des Fileobjekts definierter Extraktor extrahiert aus dem Dateiinhalt die für die Aufnahme der Metadaten verantwortlichen Daten. Dieser Extraktor dient dann als Basis-Extraktor für diejenigen Extraktoren, die für die Extraktion der Felder verantwortlich sind. Letztere Extraktoren extrahieren dann aus dem Eingabe-Kontext die für sie relevanten Daten. Am Beispiel der Extraktion der Felder eines Fileobjekts der Klasse `PDF` soll dies verdeutlicht werden: Für den Objekttyp `PDF` ist ein *PDFInfoExtractor* definiert. Dieser Extraktor dient der Extraktion des Info-Dictionary aus einer PDF-Datei. Die Felder (bzw. die entsprechenden Feld-Kontexte) `PDF/author:Person` und `PDF/title:string` werden je über einen *PatternExtractor* extrahiert. Für beide Extraktoren ist der *PDFInfoExtractor* als Basis-Extraktor definiert. Das konfigurierte Muster dient der Extraktion des entsprechenden Schlüsselwortes (`/Autor (...)` bzw. `/Title (...)`) und der dazugehörigen Daten aus dem Info-Dictionary, das der *PDFInfoExtractor* in seinem Ausgabe-Kontext zur Verfügung stellt.

Die meisten der in der Konfiguration *meta* verwendeten Extraktoren dienen der Extraktion von Mustern (*PatternExtractor*). Weitere Einzelheiten zu *meta* werden hier nicht weiter beschrieben und sind direkt aus Anhang C.1 zu entnehmen³.

³Eine Konfiguration lernt man allerdings dann am besten kennen, wenn man sie selbst in OBJECTSFS einsetzt und ihre Wirkung bei der Verarbeitung von Dateien beobachtet.

Diskussion

In Dateien vorhandene Metadaten oder Header-Informationen können zuverlässig extrahiert werden. Sie sind meistens über bestimmte Schlüsselwörter zugänglich. Da allerdings jeder Dateityp ein anderes Format für die Speicherung von Metadaten definiert, wird eine Konfiguration zur Extraktion dieser Metadaten dateitypabhängig und umfangreich. Die Art der Dateiverarbeitung muss für jeden Dateityp neu definiert werden.

Eine Abhilfe für dieses Problem ist in Form von RDF [24] in Sicht. RDF dient der Beschreibung von Ressourcen (z.B. Dateien) und wurde bereits in Abschnitt 3.3.1 vorgestellt. Einige Dateitypen bieten schon heute die Möglichkeit, Metadaten in Form von RDF-Beschreibungen in ihre Dateien zu integrieren (z.B. HTML oder PDF). In Zukunft werden voraussichtlich noch mehr Dateitypen dazukommen, so dass RDF sich zu einem Standard für die Darstellung von Metadaten entwickeln könnte. Eine einheitliche und standardisierte Form für die Darstellung von Metadaten würde die Extraktion dieser Daten stark vereinfachen.

Unabhängig von den verschiedenen Darstellungsformen für Metadaten stellt sich die Frage, wie verbreitet die Beschreibung von Dateiinhalten über Metadaten ist. Diese Frage ist insofern zentral, als es viele Systeme gibt, die auf das Vorhandensein von Metadaten angewiesen sind (z.B. Dokumenten-Management-Systeme, Semantische Dateisysteme und das in Abschnitt 3.3.1 vorgestellte Semantic Web).

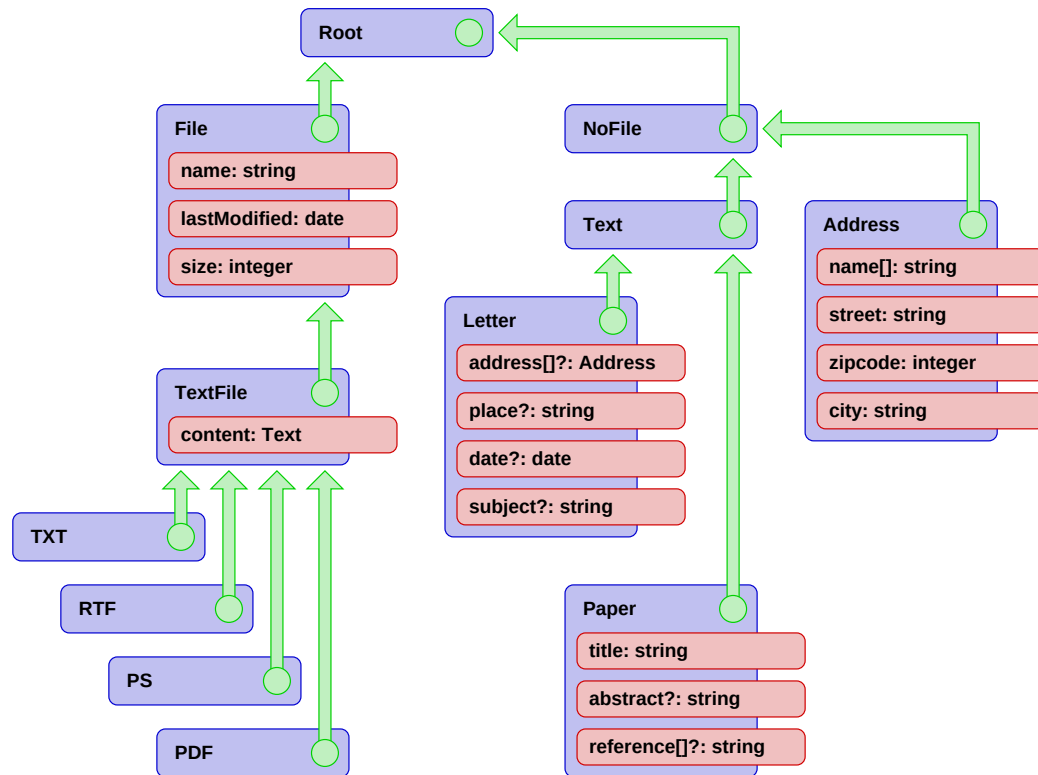
Leider zeigt sich, dass viele Dateien trotz Möglichkeiten zur Aufnahme von Metadaten keine solchen Daten aufweisen⁴. Dazu kommt, dass auch bei vorliegenden Metadaten ihre Aktualität und Korrektheit kaum überprüft werden kann.

7.2. Dynamische Klassifikation von Texten

Manche Dateien können keine Metadaten aufnehmen oder die Metadaten sind nicht vorhanden (bzw. nicht verlässlich). Sind die Dateien zusätzlich schlecht strukturiert (enthalten sie keine vordefinierten Tags oder Schlüsselwörter), so müssen sie nach einem anderen als dem in Abschnitt 7.1 vorgestellten Ansatz behandelt werden. Ihre Verarbeitung muss auf Grund des Dateiinhalts geschehen. Bei textbasierten Dateien kann dies über die in OBJECTSFS implementierten Plugins geschehen, die auf Mustersuche basieren.

Das grundsätzliche Vorgehen ist das (dynamische) Klassifizieren dieser Dateien (bzw. entsprechender Fileobjekte) auf der Basis ihres Text-Inhalts. Nach der Klassifikation wird versucht, die Felder für die klassenspezifischen Elemente zu extrahieren.

⁴Aus diesem Grund sind beispielsweise die Elemente für den Objekttyp `Document` in *meta* als optional definiert (siehe Abbildung 7.1).

Abbildung 7.2.: Typsystem der Beispielformatierung *text*

7.2.1. Beispielformatierung

Im Anhang C.2 findet sich ein Beispiel einer Konfiguration für textbasierte Dateien. Die Konfiguration heisst *text* und versucht die Dateien, wie im vorhergehenden Abschnitt kurz beschrieben, zu verarbeiten.

Typsystem

Abbildung 7.2 zeigt das Typsystem der Konfiguration *text* in einer grafischen Darstellung (siehe Abschnitt 4.1.1).

Das Typsystem ist ähnlich aufgebaut wie das von der Konfiguration *meta* definierte Typsystem (siehe Abbildung 7.1). Der Basistyp *Root* wird wiederum durch die zwei Objekttypen *File* und *NoFile* erweitert. *File* definiert die schon in Abschnitt 7.1.1 vorgestellten Elemente *name*, *lastModified* und *size*.

Die Konfiguration ist auf textbasierte Dateien ausgerichtet, die über Instanzen eines *TextFile* beschrieben werden. *TextFile* ist der einzige von *File* abgeleitete Typ und definiert ein *content*-Element, das allen seinen Subtypen (TXT, RTF, PS und PDF) gemein-

sam ist. Auch hier verkörpern die Subtypen von **TextFile** Dateitypen und ermöglichen die dateitypspezifische Extraktion des geerbten **content**-Elements (siehe nächster Abschnitt).

Das **content**-Element ist vom Typ **Text**. Es steht für den Textinhalt einer textbasierten Datei. Die Verarbeitung soll extrahierten Text dynamisch zu einem Brief oder einer wissenschaftlichen Arbeit klassifizieren⁵. Dazu wird der **Text**-Objekttyp durch die zwei Objekttypen **Letter** und **Paper** erweitert. **Letter** und **Paper** definieren Elemente, die der Beschreibung von Briefen und wissenschaftlichen Arbeiten dienen. **Letter** verwendet dazu den Objekttyp **Address**, der neben **Text** als Erweiterung von **NoFile** definiert ist.

Verarbeitungsinformationen

Über *FileNameFilter* wird sichergestellt, dass Dateien entsprechender Dateitypen zu TXT, RTF, PDF bzw. PS klassifiziert werden. Dies geschieht analog zu den Ausführungen in Abschnitt 7.1.1 über den Vergleich des Dateinamens (genauer der Endung des Dateinamens) mit einem konfigurierten Muster.

Die Felder für die in **File** definierten Elemente werden über Extraktoren extrahiert, die für eben diese Elemente (in **File**) definiert sind. Der Textinhalt einer Datei wird dann typspezifisch für das in **TextFile** definierte **content**-Element durchgeführt. Für RTF geschieht dies beispielsweise über einen *RTFTextExtractor*. Bei reinen Textdateien (mit der Endung **.txt**) wird dazu einfach der gesamte Dateiinhalt mittels eines Extraktors (*ContentExtractor*) extrahiert. Die Extraktion des **content**-Feldes wird bei Fileobjekten der Klasse **PDF** oder **PS** von einem *Executer* übernommen. Der *Executer* startet ein externes Programm, das eine entsprechende Konvertierung der Datei zu einem Text vornimmt (*pdf2txt* bzw. *ps2txt* oder ähnliche Programme⁶).

Ein einmal extrahiertes Feld (Subobjekt) vom Typ **Text** kann dynamisch zu einem **Paper**- oder einem **Letter**-Objekttyp klassifiziert werden. Dazu wird der Text über Filter untersucht. Die Filter akzeptieren Text mit einem bestimmten Aufbau oder Text, in dem spezielle Ausdrücke auftreten. Der Aufbau eines Briefes beispielsweise richtet sich nach gewissen Regeln: Auf die (optional vorhandenen) Adressen von Sender und Empfänger folgt eine Angabe des Datums (unter Umständen auch eines Orts) und eine Betreffszeile. Eine charakteristische Anrede (**Sehr geehrte ...** oder **Guten Tag ...**) leitet den eigentlichen Briefftext ein, der von einer Grussformel (**Mit freundlichen Grüßen ...**) beendet wird.

Für wissenschaftliche Arbeiten gilt, dass sie in der Regel mit dem Titel der Arbeit und Angaben zum Autor (oder den Autoren) beginnt. Weiter enthält sie üblicherwei-

⁵Es sind weitere Beispiele denkbar: Zeitungsartikel, Rechnungen, Fax-Nachrichten, Lebensläufe, u.a.

⁶Zum Beispiel liefert Ghostscript [4] entsprechende Programme zur Konvertierung von Postscript oder PDF zu Text. Die Programme sind allerdings zum Teil wenig effizient und bekunden oft Mühe mit der Erhaltung von Zeilenumbrüchen.

se eine kurze Zusammenfassung der Arbeit (*Abstract*) eine Einleitung (*Introduction*), den Hauptteil der Arbeit, eine Konklusion (*Conclusion*) und ein Literaturverzeichnis (*Bibliography*, *References*).

Die erwähnten Filter zur Klassifikation von Texten nützen die beschriebenen Eigenschaften von Briefen und wissenschaftlichen Arbeiten aus. Der für **Letter** definierte Filter (*PatternFilter*) durchsucht den Text (genauer die Kontextdaten des Objekt-Kontextes eines Fileobjekts der Klasse **Text**) nach einer Anrede (Dies geschieht über den regulären Ausdruck (`Sehr\sgeehrte|Guten\sTag`)). Enthält der fragliche Text eine solche Anrede, so wird das entsprechende Fileobjekt als **Letter** klassifiziert. Der Filter kann beliebig erweitert werden um beispielsweise auch andere Anreden oder die Grussformel zu berücksichtigen.

Beim Filter, der für den Objekttyp **Paper** definiert ist, handelt es sich ebenfalls um einen *PatternFilter*. Er untersucht seinen Eingabe-Kontext auf das Vorhandensein der oben beschriebenen Elemente, die den Aufbau einer wissenschaftlichen Arbeit charakterisieren. Konkret sucht er nach den üblichen Überschriften für eine Kurzzusammenfassung (**Abstract** bzw. **Zusammenfassung** oder **Kurzzusammenfassung**), für eine Einleitung (**Introduction** bzw. **Einleitung**) und für ein Literaturverzeichnis (**Bibliography**, **References** bzw. **Literatur**, **Literaturverzeichnis** oder **Referenzen**).

Bei erfolgreicher Klassifikation eines Fileobjekts zu **Letter** oder **Paper** werden die Felder zu den dort definierten Elementen (**address**, **place**, **date** und **subject** bzw. **title**, **abstract** und **reference**) mittels Extraktoren extrahiert, die auf Mustersuche basieren (z.B. *PatternExtractor*).

Diskussion

Die Filter zur Klassifikation von Texten zu Briefen oder wissenschaftlichen Arbeiten können so konfiguriert werden, dass sie mit hoher Wahrscheinlichkeit korrekte Resultate liefern. Das bedeutet, dass sie beispielsweise in der Lage sind, einen Brief erfolgreich als **Brief** zu erkennen und gleichzeitig einen Text, der keinen Brief darstellt, nicht als **Brief** zu klassifizieren.

Anders sieht die Situation bei der Extraktion der Felder zur Beschreibung von Briefen oder wissenschaftlichen Arbeiten aus. Die Extraktion basiert zumeist auf konkreten Annahmen zum Aufbau des Textes. Eine Abweichung zu diesen Annahmen führt dann unweigerlich zu falschen Resultaten. Ein Beispiel soll dies verdeutlichen: Der Objekttyp **Letter** definiert ein Element **subject**. Ein entsprechendes Feld in einem Fileobjekt der Klasse **Letter** soll den Betreff eines Briefes darstellen. Die Extraktion des Feldes geschieht über einen für das Element definierten *PatternExtractor*. Der Extraktor ist so konfiguriert, dass er die letzte nicht leere Zeile vor der Anrede (bzw. dem, was vom System als Anrede angeschaut wird) extrahiert. Für einen Brief, dessen Aufbau nicht den getroffenen Annahmen entspricht, könnte diese Zeile einen beliebigen Text enthalten, der

nicht dem Betreff des Briefes entspricht. Das Fileobjekt zur Beschreibung dieses Briefes würde dennoch den dort vorgefunden Text als Betreff interpretieren und extrahieren.

Ähnliches gilt für ein Feld zum Element `title` aus dem Objekttyp `Paper`. Der definierte Extraktor geht davon aus, dass der Titel einer wissenschaftlichen Arbeit immer in der ersten Zeile des entsprechenden Textes zu finden ist. Bei einer wissenschaftlichen Arbeit mit einem von dieser Annahme abweichenden Aufbau wird daher der Titel nicht korrekt extrahiert.

Ein letztes Beispiel zeigt anhand von Literaturverweisen, wie schwierig es ist, bestimmte Daten aus einem Text zu extrahieren, dessen Aufbau nicht bekannt ist.

Ein Literaturverweis besteht in seiner Grundform aus Angaben zu Autor(en), Titel und Erscheinungsjahr eines Werkes. Weitere Angaben können zum Beispiel einen Verlag, die Art des Werkes oder einen Kommentar umfassen. Leider gibt es keinen allgemein gültigen Standard zur Anordnung und zur Art der Trennung dieser Elemente. Verschiedene Institutionen stellen ihre eigenen Konventionen auf. Ist es noch vergleichsweise einfach, das Erscheinungsjahr aus einem beliebig strukturierten Literaturverweis zu extrahieren, so ist es schon schwieriger, einen Titel von einer Angabe zum Autor zu unterscheiden⁷.

Mit den in `OBJECTSFS` implementierten Filtern und Extraktoren ist diese Aufgabe kaum zu bewerkstelligen. Aus diesem Grund werden Literaturverweise in einem Fileobjekt der Klasse `Paper` über mehrere Felder dargestellt, welche den einfachen Typ `string` haben (multiples Element `reference` in `Paper`). Dies verunmöglicht eine typbasierte Suche nach Werten in Literaturverweisen (Eine Suche nach einem Literaturverweis auf ein Werk mit einem bestimmten Titel ist beispielsweise nicht möglich. Die Suche muss über Angabe eines `string`-Wertes, der im `reference`-Feld eines `Paper`-Fileobjekts auftritt, formuliert werden. Dass der angegebene Wert den Titel eines Werkes darstellen soll, kann von der Suche dabei nicht berücksichtigt werden.).

⁷Zumindest gilt dies für eine Maschine. Ein Mensch hat damit im Allgemeinen keine Mühe.

8. Konklusion

Dieses Kapitel überprüft in einer Diskussion, inwiefern die in Abschnitt 1.2 aufgestellten Ziele mit OBJECTSFS erreicht wurden (Plattformunabhängigkeit, einfacher Aufbau, Generizität, flexible Schnittstelle, Integrationsmöglichkeit für benutzerspezifizierte semantische Informationen, Konfigurierbarkeit und Erweiterbarkeit). Es zeigt, wie schwierig es ist, ein möglichst allgemeines und trotzdem leistungsfähiges Semantisches Dateisystem zu erstellen und bringt die in diesem Zusammenhang noch offenen Fragen zur Sprache.

Zum Abschluss dieses Kapitels werden einige Möglichkeiten zur Erweiterung von OBJECTSFS vorgestellt.

8.1. Diskussion und offene Fragen

Die Forderungen nach Konfigurierbarkeit und Erweiterbarkeit werden in OBJECTSFS über die Möglichkeit, Konfigurationen zu erstellen (siehe Abschnitt 4.4) und Plugins einzubinden (Abschnitt 6.1), erfüllt. Die Erweiterbarkeit von OBJECTSFS wird durch das Verwenden von Java als Implementationssprache weiter unterstützt. Die Architektur nach dem erweiterten Ansatz (siehe Abschnitt 3.2), sowie die Implementation in Java garantieren eine weitgehende Plattformunabhängigkeit des Systems¹.

OBJECTSFS definiert ein einfaches Objektmodell (siehe Kapitel 4) und bietet über Fileobjekte (Abschnitt 4.2) einen flexiblen Zugang zu semantischen Informationen über Dateien.

Der Einsatz von XML-Technologien für die Darstellung zentraler Systemkonzepte (Typsystem, Fileobjekte, Konfigurationen) und für die unkomplizierte Integration von extern spezifizierten Datei-Metadaten erhöht die Flexibilität von OBJECTSFS zusätzlich (siehe Kapitel 4). Dieser Punkt, sowie die einfache Konfigurierbarkeit des Systems unterscheiden OBJECTSFS von den in Kapitel 3 vorgestellten Semantischen Dateisystemen.

Als eigentlicher Knackpunkt stellt sich das Erstellen eines generischen Systems heraus, das semantische Informationen zu Dateien unabhängig von ihrem Typ extrahieren kann und daher auch mit neuen und unbekannten Dateitypen zurecht kommt. Dies ist mit den vorgestellten Mitteln nur möglich, falls sich ein einheitliches Format zur Darstellung von Metadaten durchsetzen kann und zur Integration von semantischen Informationen in

¹Eine Konfiguration kann das System allerdings an eine gewisse Umgebung bzw. Plattform binden (z.B. über das Verwenden von *Executer*-Extraktoren zur Ausführung externer Programme).

den Dateidaten verwendet wird. Das selbständige Extrahieren von semantischen Informationen aus unstrukturierten Daten ist für ein Semantisches Dateisystem aussichtslos, da eine Applikation diese Informationen beliebig codieren kann.

Ein generisches (d.h. dateitypunabhängiges) Semantisches Dateisystem ist also auf das Vorhandensein von Metadaten in einer einheitlichen Form angewiesen, welche die semantischen Informationen zu den Dateien aufnehmen. Das in Semantic Web (siehe Abschnitt 3.3.1) verwendete RDF [24] würde sich zur Darstellung solcher Metadaten eignen². Es fragt sich, welche Rolle RDF in diesem Zusammenhang in Zukunft einnehmen wird.

Die Möglichkeit zur Angabe von Metadaten in einer Datei bedeutet nicht, dass die entsprechenden Daten auch vorhanden oder aktuell sind. Die Verantwortung zur Angabe von Metadaten liegt heute in den meisten Fällen in der Hand des Benutzers. Würde diese Verantwortung auf die dateierzeugenden und dateimanipulierenden Applikationen übertragen, so könnten diese eine weitere Verbreitung von Metadaten-Angaben sicherstellen. Es ist jedoch zu bezweifeln, dass damit auch eine hohe Qualität dieser Angaben erreicht werden kann.

Da sich bis heute noch kein Format zur Darstellung von Metadaten in den Dateidaten als Standard etabliert hat (und viele Dateitypen die Angabe von Metadaten gar nicht unterstützen), verwendet OBJECTSFS eine Konfiguration zur Anpassung an die Umgebung. Eine gut abgestimmte Konfiguration garantiert maximale Leistungsfähigkeit des Systems. Trotz einer dateitypabhängigen Konfiguration erlaubt eine geschickte Strukturierung des Typsystems über das Zusammenfassen von gemeinsamen Elementen in Supertypen (siehe auch Kapitel 7) einen dateitypunabhängigen Zugriff auf semantische Informationen (Fileobjekte).

OBJECTSFS setzt das Konzept der einheitlichen Darstellung von Datei-Metadaten um. Die angesprochenen Metadaten werden so abgelegt, dass sie vom Benutzer oder von einer Applikation leicht erzeugt oder angepasst werden können (siehe Abschnitt 4.3.2). Durch die automatische Integration dieser Daten kann OBJECTSFS effizient und flexibel mit zusätzlichen semantischen Informationen versorgt werden.

8.2. Erweiterungsmöglichkeiten

OBJECTSFS ist ein Prototyp eines semantischen Systems und lässt sich in diversen Bereichen weiterentwickeln. Nachfolgend eine Aufzählung von Punkten zu möglichen Erweiterungen von OBJECTSFS:

- **Konfigurations-Evolution:** Die Konfigurations-Evolution beschreibt die Unterschiede zwischen einer Ausgangs-Konfiguration und einer darauf aufbauenden, veränderten Konfiguration. Das Erkennen solcher Veränderungen erlaubt beim

²und wird zum Teil auch schon verwendet (z.B. in HTML oder PDF).

Wechsel der Systemkonfiguration eine selektive Neuverarbeitung von Dateien (siehe Abschnitt 6.3.1). Unter Umständen ist nur eine einfache Anpassung von Fileobjekten nötig (wenn zum Beispiel ein Element eines Objekttyps gelöscht wurde, müssen nur die entsprechenden Felder in den Fileobjekten gelöscht werden). Im besten Fall erfordert ein Wechsel der Systemkonfiguration keine Reaktion des Systems (wenn sich zum Beispiel nur der Name eines Elements geändert hat).

- **Erweiterung des Objektmodells:** Eine Erweiterung des Objektmodells könnte die Unterstützung zusätzlicher einfacher Typen oder eine Integration von weiteren XML-Schema Konzepten [24] vorsehen.
- **Erweiterung der Konfigurations- und Verarbeitungsmöglichkeiten:** Die Möglichkeiten der Konfiguration und der Verarbeitung können durch zusätzliche Filter- und Extraktor-Konzepte erweitert werden. Die Anwendung eines Extraktors könnte beispielsweise an eine Bedingung geknüpft werden, die der Eingabekontext (oder ein bestimmter, bereits extrahierter Feld-Wert) erfüllen muss. Dies würde Möglichkeiten zu einer dynamischen Verarbeitung auf Grund von Kontext-Daten (bzw. Feld-Werten) eröffnen (ohne das Erstellen zusätzlicher Typen und die Verwendung von Filtern). Denkbar wäre zum Beispiel eine Unterstützung von sprachspezifischer Verarbeitung.
- **Entwickeln von Plugins:** Zusätzliche Plugins (Filter und Extraktoren) stellen weitere Funktionalität für die Verarbeitung von Dateien zur Verfügung. Ein nützlicher Extraktor wäre beispielsweise ein *KeywordExtractor*, der die Schlüsselwörter aus einem Text extrahieren kann. Ein solcher Extraktor wäre universell in Konfigurationen einsetzbar und könnte für den Benutzer wertvolle semantische Informationen zu textbasierten Dateien bereitstellen.
- **Effizienz und Skalierbarkeit:** Es können zusätzliche Massnahmen zur Steigerung der Geschwindigkeit bei der Verarbeitung von Dateien und bei der Bearbeitung von Suchanfragen getroffen werden (z.B. Indexieren von Fileobjekten, Caching von Anfrage-Resultaten usw.). Die Skalierbarkeit des Systems kann untersucht werden, um eventuelle Anpassungen für die Aufnahme und Verwaltung grosser Mengen von Dateien und Fileobjekten vorzunehmen.
- **Unterstützung von RDF-Konzepten:** Wünschenswert wäre eine global definierte Semantik für Fileobjekte und Felder von Fileobjekten analog zu den standardisierten Prädikaten in RDF. Zusätzlich kann die Darstellung von Datei-Metadaten in RDF-Format unterstützt werden.
- **Erweiterung der Query-Möglichkeiten:** Eine Möglichkeit zur flexibleren Verknüpfung von Anfrage-Elementen (siehe Abschnitt 4.2.1) kann geschaffen werden (z.B. *oder*-Verknüpfungen statt nur *und*-Verknüpfungen der Anfrage-Elemente).

Ausserdem könnte die Unterstützung einer standardisierten Query-Sprache (z.B. OQL) für den Zugriff auf Fileobjekte in Betracht gezogen werden.

- **Internet-Integration:** Die Implementation einer Browser-Schnittstelle zum Objektsystem und Filesystem von OBJECTSFS würde einen Zugriff über das Internet erlauben.

A. Installation

Tabelle A.1 listet die verschiedenen Komponenten auf, die für die Installation von OBJECTSFS benötigt werden.

J2RE ab 1.4.0	<i>Java Runtime Environment</i> (oder <i>Java Development Kit</i>). OBJECTSFS verwendet das mit Java Version 1.4.0 eingeführte Logging-API (<code>java.util.logging</code>) und das Regular-Expressions Paket (<code>java.util.regex</code>). Zusätzlich macht das System von der mitgelieferten JAXP (<i>Java API for XML Processing</i>) Implementation (<code>javax.xml</code>) und dem neuen I/O-API (<code>java.nio</code>) Gebrauch.
Ozone Database 1.0.1	Objekt-orientierte Java-Datenbank. Ozone wird für die persistente Speicherung zentraler Java-Objekte von OBJECTSFS (Filesystem, Objektsystem, Systemkonfiguration) verwendet.
Etymon Pj 1.10	PDF-Library für die Extraktion von PDF-Objekten aus PDF-Dateien. Pj findet Verwendung in <i>PDFInfoExtractor</i> , einem Plugin für die Extraktion des Info-Dictionary einer PDF-Datei.
OBJECTSFS	Klassen des OBJECTSFS mit den verwendeten Icons (<i>Toolbar Button Graphics</i> aus dem <i>Java Look and Feel Graphics Repository</i>).
(Optional) Jakarta Ant 1.4.1	Java-Build-Tool für die Kompilation der OBJECTSFS-Klassen.

Tabelle A.1.: Benötigte Komponenten zur Installation von OBJECTSFS

Die in Tabelle A.1 aufgeführten Komponenten werden gemäss den jeweiligen Installationsanweisungen installiert und die Umgebungsvariablen wie beschrieben angepasst.

Ein Datenbank-Verzeichnis für Ozone muss angelegt werden. Dies geschieht mit dem Ozone Installations-Tool: `ozoneInst -d<dir>`, wobei `<dir>` den Namen des zu erstellenden Verzeichnisses angibt.

Ein mit Ozone mitgeliefertes Skript startet den Ozone-Server über `ozone -d<dir>`. Das Datenbank-Verzeichnis wird in `<dir>` übergeben. Um Ozone mit OBJECTSFS verwenden zu können, muss Ozone Zugriff auf alle von OBJECTSFS benötigten Klassen haben (Dies umfasst die Klassen von OBJECTSFS, die von zusätzlichen Plugins und die von diesen Plugins verwendeten Klassen.). Dazu ist der Klassenpfad (über die Umgebungsvariable `CLASSPATH`) vor dem Start des Ozone-Servers entsprechend anzupassen.

Nach dem Start des Ozone-Servers kann auch OBJECTSFS (bzw. die Test-Applikation) gestartet werden. Dies geschieht über den Aufruf `java ch.ethz.sfs.test.Test`. Über den Klassenpfad müssen die Klassen von OBJECTSFS, die verwendeten Icons, die Klassen von Ozone und die Klassen der implementierten Plugins mitsamt den von den Plugins verwendeten Klassen erreichbar sein. Der Klassenpfad wird global über die Umgebungsvariable `CLASSPATH` oder über einen Parameter `-cp` beim Aufruf von OBJECTSFS spezifiziert.

Die OBJECTSFS-Distribution stellt einige Skripts zur Verfügung, über die sich Ozone und OBJECTSFS einfach starten lassen. Ausführlichere Informationen dazu sind der mitgelieferten README-Datei zu entnehmen.

B. Klassendiagramm

Abbildung [B.1](#) zeigt die wichtigsten Klassen von OBJECTSFS. Die Klassen für das Event-Handling, GUI-Komponenten (Paket `ch.ethz.sfs.gui`) und das Logging (Paket `ch.ethz.sfs.logging`) sind nicht dargestellt. Es fehlen auch die Klassen der Plugin-Implementationen (Paket `ch.ethz.sfs.plugins`).

Auf die Angabe von Methoden und teilweise auch von Feldern wurde aus Gründen der Übersichtlichkeit verzichtet. Das Klassendiagramm soll das Design von OBJECTSFS veranschaulichen. Es beschreibt das System aus einer konzeptuellen Perspektive und nicht aus einer Perspektive der Spezifikation oder der Implementation. Details zur Implementation von OBJECTSFS sind direkt dem Sourcecode zu entnehmen.

B. Klassendiagramm

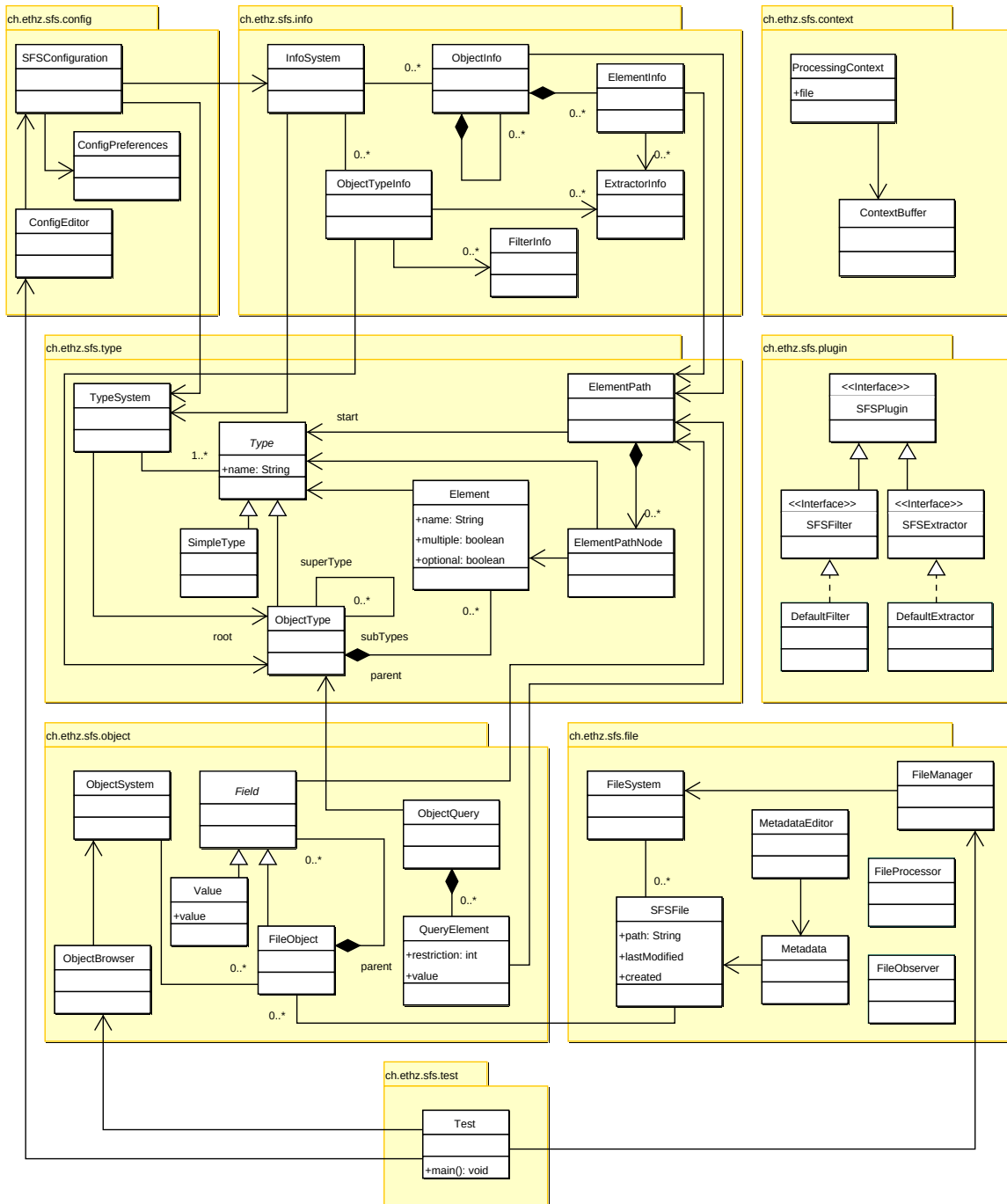


Abbildung B.1.: Die wichtigsten Klassen von OBJECTSFS

C. Beispielkonfigurationen

C.1. Beispielkonfiguration meta

```
<?xml version="1.0" encoding="iso-8859-1"?>
<configuration name="meta">
  <schema xmlns="http://www.w3.org/2001/XMLSchema">
    <element name="sfsFile">
      <element name="object" type="Root" minOccurs="0" maxOccurs="unbounded" />
    </element>
    <complexType name="Root" />
    <complexType name="File">
      <extension base="Root">
        <sequence>
          <element name="name" type="string" />
          <element name="lastModified" type="date" />
          <element name="size" type="integer" />
        </sequence>
      </extension>
    </complexType>
    <complexType name="NoFile">
      <extension base="Root" />
    </complexType>
    <complexType name="Document">
      <extension base="File">
        <sequence>
          <element name="author" type="Person" minOccurs="0" maxOccurs="unbounded" />
          <element name="title" type="string" minOccurs="0" />
          <element name="subject" type="string" minOccurs="0" />
          <element name="keyword" type="string" minOccurs="0"
            maxOccurs="unbounded" />
          <element name="date" type="date" minOccurs="0" />
        </sequence>
      </extension>
    </complexType>
    <complexType name="RTF">
      <extension base="Document" />
    </complexType>
    <complexType name="PDF">
      <extension base="Document" />
    </complexType>
  </schema>
</configuration>
```

```
<complexType name="LaTeX">
  <extension base="Document" />
</complexType>
<complexType name="Person">
  <extension base="NoFile">
    <sequence>
      <element name="name" type="string" />
    </sequence>
  </extension>
</complexType>
<complexType name="Song">
  <extension base="File">
    <sequence>
      <element name="artist" type="Person" minOccurs="0" />
      <element name="title" type="string" minOccurs="0" />
      <element name="album" type="string" minOccurs="0" />
    </sequence>
  </extension>
</complexType>
<complexType name="MP3">
  <extension base="Song" />
</complexType>
<complexType name="HTML">
  <extension base="Document" />
</complexType>
<complexType name="Mail">
  <extension base="File">
    <sequence>
      <element name="message" type="Message" minOccurs="0"
        maxOccurs="unbounded" />
    </sequence>
  </extension>
</complexType>
<complexType name="Message">
  <extension base="NoFile">
    <sequence>
      <element name="sender" type="Person" />
      <element name="receiver" type="Person" maxOccurs="unbounded" />
      <element name="subject" type="string" />
      <element name="date" type="date" />
    </sequence>
  </extension>
</complexType>
<complexType name="Image">
  <extension base="File">
    <sequence>
      <element name="width" type="integer" />
      <element name="height" type="integer" />
    </sequence>
  </extension>
</complexType>
```

```
</complexType>
<complexType name="GIF">
  <extension base="Image" />
</complexType>
</schema>
<infoSystem>
  <objectTypeInfo about="HTML">
    <filter name="html" class="ch.ethz.sfs.plugins.FileNameFilter"
      configuration="(?.i)\.html?" />
    <extractor name="content" class="ch.ethz.sfs.plugins.ContentExtractor"
      configuration="sof:eof" />
    <extractor name="head" class="ch.ethz.sfs.plugins.PatternExtractor"
      base="content" configuration="(?.s)&lt;html&gt;((.?.*))&lt;/head&gt;" />
  </objectTypeInfo>
  <objectTypeInfo about="MP3">
    <filter name="mp3" class="ch.ethz.sfs.plugins.FileNameFilter"
      configuration="(?.i)\.mp3" />
    <extractor name="last128b" class="ch.ethz.sfs.plugins.ContentExtractor"
      configuration="so128:eof" />
    <extractor name="id3v1" class="ch.ethz.sfs.plugins.PatternExtractor"
      base="last128b" configuration="TAG((.?.*))" />
  </objectTypeInfo>
  <objectTypeInfo about="LaTeX">
    <filter name="tex" class="ch.ethz.sfs.plugins.FileNameFilter"
      configuration="(?.i)\.tex" />
    <extractor name="content" class="ch.ethz.sfs.plugins.ContentExtractor"
      configuration="sof:eof" />
  </objectTypeInfo>
  <objectTypeInfo about="PDF">
    <filter name="pdf" class="ch.ethz.sfs.plugins.FileNameFilter"
      configuration="(?.i)\.pdf" />
    <extractor name="infodic" class="ch.ethz.sfs.plugins.PDFInfoExtractor"
      configuration="none" />
  </objectTypeInfo>
  <objectTypeInfo about="GIF">
    <filter name="gif" class="ch.ethz.sfs.plugins.FileNameFilter"
      configuration="(?.i)\.gif" />
  </objectTypeInfo>
  <objectTypeInfo about="RTF">
    <filter name="rtf" class="ch.ethz.sfs.plugins.FileNameFilter"
      configuration="(?.i)\.rtf" />
    <extractor name="content" class="ch.ethz.sfs.plugins.ContentExtractor"
      configuration="sof:eof" />
    <extractor name="info" class="ch.ethz.sfs.plugins.PatternExtractor"
      base="content" configuration="\{\\info(((\s*{.?.*})*)\s*)}" />
  </objectTypeInfo>
  <objectTypeInfo about="NoFile">
    <filter name="block" class="ch.ethz.sfs.plugin.DefaultFilter"
      configuration="none" />
  </objectTypeInfo>
```

```

<objectTypeInfo about="Message">
  <extractor name="header" class="ch.ethz.sfs.plugins.PatternExtractor"
    configuration="(m)((^p{Blank}*S.*$+))" />
</objectTypeInfo>
<objectTypeInfo about="File">
  <filter name="all" class="ch.ethz.sfs.plugins.FileNameFilter"
    configuration=".*" />
</objectTypeInfo>
<objectTypeInfo about="Mail">
  <filter name="msgID" class="ch.ethz.sfs.plugins.PatternFilter" base="content"
    configuration="(m)^Message-ID:\s<.*?>$" />
  <extractor name="content" class="ch.ethz.sfs.plugins.ContentExtractor"
    configuration="sof:eof" />
</objectTypeInfo>
<objectInfo path="LaTeX">
  <elementInfo path="LaTeX/subject:string">
    <extractor name="abstract" class="ch.ethz.sfs.plugins.PatternExtractor"
      base="content"
      configuration="(s)\\begin{abstract}\\((.*)\\)\\end{abstract}" />
  </elementInfo>
  <elementInfo path="LaTeX/title:string">
    <extractor name="title" class="ch.ethz.sfs.plugins.PatternExtractor"
      base="content" configuration="\\title{((.*)\\)}" />
  </elementInfo>
  <elementInfo path="LaTeX/author:Person">
    <extractor name="authorlist" class="ch.ethz.sfs.plugins.PatternExtractor"
      base="content" configuration="\\author{((.*)\\)}" />
    <extractor name="author" class="ch.ethz.sfs.plugins.Splitter"
      base="authorlist" configuration="(s\\and\\s|\\sand\\s|,s)" />
  </elementInfo>
  <elementInfo path="LaTeX/date:date">
    <extractor name="date" class="ch.ethz.sfs.plugins.PatternExtractor"
      base="content" configuration="\\date{((.*)\\)}" />
  </elementInfo>
  <elementInfo path="LaTeX/keyword:string">
    <extractor name="section" class="ch.ethz.sfs.plugins.PatternExtractor"
      base="content" configuration="\\section{((.*)\\)}" />
  </elementInfo>
</objectInfo>
<objectInfo path="GIF">
  <elementInfo path="GIF/width:integer">
    <extractor name="widthbytes" class="ch.ethz.sfs.plugins.ContentExtractor"
      configuration="sp6:eo2" />
    <extractor name="width" class="ch.ethz.sfs.plugins.NumberExtractor"
      base="widthbytes" configuration="le" />
  </elementInfo>
  <elementInfo path="GIF/height:integer">
    <extractor name="heightbytes" class="ch.ethz.sfs.plugins.ContentExtractor"
      configuration="sp8:eo2" />
    <extractor name="height" class="ch.ethz.sfs.plugins.NumberExtractor"

```

```
        base="heightbytes" configuration="le" />
    </elementInfo>
</objectInfo>
<objectInfo path="PDF">
    <elementInfo path="PDF/subject:string">
        <extractor name="subject" class="ch.ethz.sfs.plugins.PatternExtractor"
            base="infodic" configuration="/Subject\s*(((.*?))\s)" />
    </elementInfo>
    <elementInfo path="PDF/title:string">
        <extractor name="title" class="ch.ethz.sfs.plugins.PatternExtractor"
            base="infodic" configuration="/Title\s*(((.*?))\s)" />
    </elementInfo>
    <elementInfo path="PDF/author:Person">
        <extractor name="authorlist" class="ch.ethz.sfs.plugins.PatternExtractor"
            base="infodic" configuration="/Author\s*(((.*?))\s)" />
        <extractor name="author" class="ch.ethz.sfs.plugins.Splitter"
            base="authorlist" configuration="(\s|\sand\s)" />
    </elementInfo>
    <elementInfo path="PDF/date:date">
        <extractor name="creation" class="ch.ethz.sfs.plugins.PatternExtractor"
            base="infodic" configuration="/CreationDate\s*\s*(D:((\d{8})))" />
        <extractor name="date" class="ch.ethz.sfs.plugins.DateExtractor"
            base="creation" configuration="yyyyMMdd" />
    </elementInfo>
    <elementInfo path="PDF/keyword:string">
        <extractor name="keywordlist" class="ch.ethz.sfs.plugins.PatternExtractor"
            base="infodic" configuration="/Keywords\s*(((.*?))\s)" />
        <extractor name="keyword" class="ch.ethz.sfs.plugins.Splitter"
            base="keywordlist" configuration=",\s" />
    </elementInfo>
</objectInfo>
<objectInfo path="RTF">
    <elementInfo path="RTF/subject:string">
        <extractor name="subject" class="ch.ethz.sfs.plugins.PatternExtractor"
            base="info" configuration="\{\s*\subject\s*(((.*?))\s)\}" />
    </elementInfo>
    <elementInfo path="RTF/title:string">
        <extractor name="title" class="ch.ethz.sfs.plugins.PatternExtractor"
            base="info" configuration="\{\s*\title\s*(((.*?))\s)\}" />
    </elementInfo>
    <elementInfo path="RTF/author:Person">
        <extractor name="authorlist" class="ch.ethz.sfs.plugins.PatternExtractor"
            base="info" configuration="\{\s*\author\s*(((.*?))\s)\}" />
        <extractor name="author" class="ch.ethz.sfs.plugins.Splitter"
            base="authorlist" configuration="(\s|\sand\s)" />
    </elementInfo>
    <elementInfo path="RTF/date:date">
        <extractor name="creatim" class="ch.ethz.sfs.plugins.PatternExtractor"
            base="info" configuration="\{\s*\creatim\\yr((\d+\s*\mo\d+\s*\dy\d+))\}" />
        <extractor name="replace" class="ch.ethz.sfs.plugins.Replacer" base="creatim"
```

```

    configuration="\\w\w-&gt;-" />
    <extractor name="date" class="ch.ethz.sfs.plugins.DateExtractor"
    base="replace" configuration="yyyy-MM-dd" />
  </elementInfo>
  <elementInfo path="RTF/keyword:string">
    <extractor name="keywordlist" class="ch.ethz.sfs.plugins.PatternExtractor"
    base="info" configuration="\{\{\keywords\s*((.*?))\}" />
    <extractor name="keyword" class="ch.ethz.sfs.plugins.Splitter"
    base="keywordlist" configuration=",\s" />
  </elementInfo>
</objectInfo>
<objectInfo path="Message">
  <elementInfo path="Message/subject:string">
    <extractor name="subject" class="ch.ethz.sfs.plugins.PatternExtractor"
    base="header" configuration="(m)^Subject:\s((.*?))$" />
  </elementInfo>
  <elementInfo path="Message/receiver:Person">
    <extractor name="to" class="ch.ethz.sfs.plugins.PatternExtractor"
    base="header" configuration="(ms)^To:\s((.*?))^\s+?:" />
    <extractor name="split" class="ch.ethz.sfs.plugins.Splitter" base="to"
    configuration=",\s+" />
  </elementInfo>
  <elementInfo path="Message/sender:Person">
    <extractor name="from" class="ch.ethz.sfs.plugins.PatternExtractor"
    base="header" configuration="(m)^From:\s((.*?))$" />
  </elementInfo>
  <elementInfo path="Message/date:date">
    <extractor name="date" class="ch.ethz.sfs.plugins.PatternExtractor"
    base="header" configuration="(m)^Date:\s\D+(\d\s\w+\s\d+)" />
  </elementInfo>
</objectInfo>
<objectInfo path="Mail">
  <elementInfo path="Mail/message:Message">
    <extractor name="header" class="ch.ethz.sfs.plugins.PatternExtractor"
    base="content"
    configuration="(m)((^(.+){0,100}^Message-ID:\s<.*?&gt;$(^.+)*))" />
  </elementInfo>
</objectInfo>
<objectInfo path="File">
  <elementInfo path="File/name:string">
    <extractor name="name" class="ch.ethz.sfs.plugins.FileAttributeExtractor"
    configuration="name" />
  </elementInfo>
  <elementInfo path="File/size:integer">
    <extractor name="size" class="ch.ethz.sfs.plugins.FileAttributeExtractor"
    configuration="size" />
  </elementInfo>
  <elementInfo path="File/lastModified:date">
    <extractor name="lastModified"
    class="ch.ethz.sfs.plugins.FileAttributeExtractor"

```

```
        configuration="lastModified" />
    </elementInfo>
</objectInfo>
<objectInfo path="HTML">
    <elementInfo path="HTML/subject:string">
        <extractor name="description" class="ch.ethz.sfs.plugins.PatternExtractor"
            base="head"
            configuration="\&lt;meta\sname=\&quot;description\&quot;\scontent=\&quot;((.*?))\&quot;\&gt;"/>
    </elementInfo>
    <elementInfo path="HTML/title:string">
        <extractor name="title" class="ch.ethz.sfs.plugins.PatternExtractor"
            base="head" configuration="\&lt;title\&gt;((.*?))\&lt;/title\&gt;"/>
    </elementInfo>
    <elementInfo path="HTML/author:Person">
        <extractor name="author" class="ch.ethz.sfs.plugins.PatternExtractor"
            base="head"
            configuration="\&lt;meta\sname=\&quot;author\&quot;\scontent=\&quot;((.*?))\&quot;\&gt;"/>
    </elementInfo>
    <elementInfo path="HTML/date:date">
        <extractor name="datestring" class="ch.ethz.sfs.plugins.PatternExtractor"
            base="head"
            configuration="\&lt;meta\sname=\&quot;date\&quot;\scontent=\&quot;((.*?))\&quot;\&gt;"/>
        <extractor name="date" class="ch.ethz.sfs.plugins.DateExtractor"
            base="datestring" configuration="yyyy-MM-dd"/>
    </elementInfo>
    <elementInfo path="HTML/keyword:string">
        <extractor name="keywordlist" class="ch.ethz.sfs.plugins.PatternExtractor"
            base="head"
            configuration="\&lt;meta\sname=\&quot;keywords\&quot;\scontent=\&quot;((.*?))\&quot;\&gt;"/>
        <extractor name="keyword" class="ch.ethz.sfs.plugins.Splitter"
            base="keywordlist" configuration=",\s"/>
    </elementInfo>
</objectInfo>
<objectInfo path="MP3">
    <elementInfo path="MP3/title:string">
        <extractor name="title" class="ch.ethz.sfs.plugins.PatternExtractor"
            base="id3v1" configuration="((.*?))\x00"/>
    </elementInfo>
    <elementInfo path="MP3/artist:Person">
        <extractor name="artist" class="ch.ethz.sfs.plugins.PatternExtractor"
            base="id3v1" configuration=".{30}((.*?))\x00"/>
    </elementInfo>
    <elementInfo path="MP3/album:string">
        <extractor name="album" class="ch.ethz.sfs.plugins.PatternExtractor"
            base="id3v1" configuration=".{60}((.*?))\x00"/>
    </elementInfo>
```

```
</objectInfo>
</infoSystem>
<preferences fileObserverSleepTime="3600" fileProcessorPriority="5" />
</configuration>
```

C.2. Beispielkonfiguration text

```
<?xml version="1.0" encoding="iso-8859-1"?>
<configuration name="text">
  <schema xmlns="http://www.w3.org/2001/XMLSchema">
    <element name="sfsFile">
      <element name="object" type="Root" minOccurs="0" maxOccurs="unbounded" />
    </element>
    <complexType name="Root" />
    <complexType name="File">
      <extension base="Root">
        <sequence>
          <element name="name" type="string" />
          <element name="lastModified" type="date" />
          <element name="size" type="integer" />
        </sequence>
      </extension>
    </complexType>
    <complexType name="TextFile">
      <extension base="File">
        <sequence>
          <element name="content" type="Text" />
        </sequence>
      </extension>
    </complexType>
    <complexType name="TXT">
      <extension base="TextFile" />
    </complexType>
    <complexType name="RTF">
      <extension base="TextFile" />
    </complexType>
    <complexType name="NoFile">
      <extension base="Root" />
    </complexType>
    <complexType name="Text">
      <extension base="NoFile" />
    </complexType>
    <complexType name="Letter">
      <extension base="Text">
        <sequence>
          <element name="address" type="Address" minOccurs="0"
            maxOccurs="unbounded" />
          <element name="place" type="string" minOccurs="0" />
          <element name="date" type="date" minOccurs="0" />
        </sequence>
      </extension>
    </complexType>
  </schema>

```

```

        <element name="subject" type="string" minOccurs="0" />
    </sequence>
</extension>
</complexType>
<complexType name="Address">
    <extension base="NoFile">
        <sequence>
            <element name="name" type="string" maxOccurs="unbounded" />
            <element name="street" type="string" />
            <element name="zipcode" type="string" />
            <element name="city" type="string" />
        </sequence>
    </extension>
</complexType>
<complexType name="PS">
    <extension base="TextFile" />
</complexType>
<complexType name="PDF">
    <extension base="TextFile" />
</complexType>
<complexType name="Paper">
    <extension base="Text">
        <sequence>
            <element name="title" type="string" />
            <element name="abstract" type="string" minOccurs="0" />
            <element name="reference" type="string" minOccurs="0"
                maxOccurs="unbounded" />
        </sequence>
    </extension>
</complexType>
</schema>
<infoSystem>
    <objectTypeInfo about="NoFile">
        <filter name="block" class="ch.ethz.sfs.plugin.DefaultFilter"
            configuration="none" />
    </objectTypeInfo>
    <objectTypeInfo about="RTF">
        <filter name="rtf" class="ch.ethz.sfs.plugins.FileNameFilter"
            configuration="(i)\.rtf" />
    </objectTypeInfo>
    <objectTypeInfo about="TXT">
        <filter name="txt" class="ch.ethz.sfs.plugins.FileNameFilter"
            configuration="(i)\.txt" />
    </objectTypeInfo>
    <objectTypeInfo about="Paper">
        <filter name="paper" class="ch.ethz.sfs.plugins.PatternFilter"
            configuration="(ms) (^(\d+\.?)? \p{Blank} *(((Abstract) |
            ((Kurz)?(Z|z)usammenfassung)) | ((Einleitung) | (Introduction)) |
            ((Literatur(verzeichnis)? | (Referenzen) | (Bibliography) | (References)))
            \p{Blank} *$.*){2,})" />

```

```

</objectTypeInfo>
<objectTypeInfo about="PDF">
  <filter name="pdf" class="ch.ethz.sfs.plugins.FileNameFilter"
    configuration="(?.i)\.pdf" />
</objectTypeInfo>
<objectTypeInfo about="File">
  <filter name="all" class="ch.ethz.sfs.plugins.FileNameFilter"
    configuration=".*" />
</objectTypeInfo>
<objectTypeInfo about="PS">
  <filter name="ps" class="ch.ethz.sfs.plugins.FileNameFilter"
    configuration="(?.i)\.ps" />
</objectTypeInfo>
<objectTypeInfo about="Letter">
  <filter name="letter" class="ch.ethz.sfs.plugins.PatternFilter"
    configuration="(Sehr\sgeehrte|Guten\sTag)" />
  <extractor name="header" class="ch.ethz.sfs.plugins.PatternExtractor"
    configuration="(?.s)((.?.)) (Sehr\sgeehrte|Guten\sTag)" />
</objectTypeInfo>
<objectInfo path="RTF">
  <elementInfo path="RTF/content:Text">
    <extractor name="rtf2txt" class="ch.ethz.sfs.plugins.RTFTextExtractor"
      configuration="none" />
  </elementInfo>
</objectInfo>
<objectInfo path="TXT">
  <elementInfo path="TXT/content:Text">
    <extractor name="content" class="ch.ethz.sfs.plugins.ContentExtractor"
      configuration="sof:eof" />
  </elementInfo>
</objectInfo>
<objectInfo path="Paper">
  <elementInfo path="Paper/reference:string">
    <extractor name="references" class="ch.ethz.sfs.plugins.PatternExtractor"
      configuration="(?.ms)^(?.:\d+\.?.)?\p{Blank}*(?.:(?.:Literatur(?.:verzeichnis)?|
(?.:Referenzen)|(?.:Bibliography)|(?.:References))\p{Blank}*$(?.?.))" />
    <extractor name="reference" class="ch.ethz.sfs.plugins.PatternExtractor"
      base="references" configuration="(?.m)^(?.?.))$" />
  </elementInfo>
  <elementInfo path="Paper/abstract:string">
    <extractor name="abstract" class="ch.ethz.sfs.plugins.PatternExtractor"
      configuration="(?.ms)^(?.:\d+\.?.)?\p{Blank}*(?.:(?.:Abstract)|
(?.:(?.:Kurz)?(?.:Z|z)usammenfassung))\p{Blank}*$(?.?.))$"
      ^(\d+\.?.)?\p{Blank}*\S*\p{Blank}*$" />
  </elementInfo>
  <elementInfo path="Paper/title:string">
    <extractor name="title" class="ch.ethz.sfs.plugins.PatternExtractor"
      configuration="(?.m)\A\p{Space}*$(?.?.))\p{Blank}*$" />
  </elementInfo>
</objectInfo>

```

```
<objectInfo path="File">
  <elementInfo path="File/size:integer">
    <extractor name="size" class="ch.ethz.sfs.plugins.FileAttributeExtractor"
      configuration="size" />
  </elementInfo>
  <elementInfo path="File/lastModified:date">
    <extractor name="lastModified"
      class="ch.ethz.sfs.plugins.FileAttributeExtractor"
      configuration="lastModified" />
  </elementInfo>
  <elementInfo path="File/name:string">
    <extractor name="name" class="ch.ethz.sfs.plugins.FileAttributeExtractor"
      configuration="name" />
  </elementInfo>
</objectInfo>
<objectInfo path="PDF">
  <elementInfo path="PDF/content:Text">
    <extractor name="pstotxt" class="ch.ethz.sfs.plugins.Executer"
      configuration="pdf2txt(File)" />
  </elementInfo>
</objectInfo>
<objectInfo path="PS">
  <elementInfo path="PS/content:Text">
    <extractor name="pstotxt" class="ch.ethz.sfs.plugins.Executer"
      configuration="ps2txt(File)" />
  </elementInfo>
</objectInfo>
<objectInfo path="Address">
  <elementInfo path="Address/city:string">
    <extractor name="city" class="ch.ethz.sfs.plugins.PatternExtractor"
      configuration="(m)^\p{Blank}*\d{4,}\s+((.*?))\p{Blank}*$" />
  </elementInfo>
  <elementInfo path="Address/zipcode:string">
    <extractor name="zipcode" class="ch.ethz.sfs.plugins.PatternExtractor"
      configuration="(m)^\p{Blank}*((\d{4,})).*$" />
  </elementInfo>
  <elementInfo path="Address/street:string">
    <extractor name="street" class="ch.ethz.sfs.plugins.PatternExtractor"
      configuration="(m)^\p{Blank}*((.*\d+))\p{Blank}*$" />
  </elementInfo>
  <elementInfo path="Address/name:string">
    <extractor name="name" class="ch.ethz.sfs.plugins.PatternExtractor"
      configuration="(m)^\p{Blank}*((\D{2,40}?))\p{Blank}*$" />
  </elementInfo>
</objectInfo>
<objectInfo path="Letter">
  <elementInfo path="Letter/subject:string">
    <extractor name="subject" class="ch.ethz.sfs.plugins.PatternExtractor"
      base="header" configuration="(m)((^\S+.*$))\p{Space}*\z" />
  </elementInfo>
</objectInfo>
```

```
<elementInfo path="Letter/date:date">
  <extractor name="date" class="ch.ethz.sfs.plugins.PatternExtractor"
    base="header"
    configuration="(m)((\d{1,2}\.(\d{1,2}\.|\s\S*\.?s)\d{2,4}))\p{Blank}*$" />
</elementInfo>
<elementInfo path="Letter/place:string">
  <extractor name="place" class="ch.ethz.sfs.plugins.PatternExtractor"
    base="header" configuration="(m)^\p{Blank}*((.*)),\s\d{1,2}\." />
</elementInfo>
<elementInfo path="Letter/address:Address">
  <extractor name="address" class="ch.ethz.sfs.plugins.PatternExtractor"
    base="header"
    configuration="(m)((^\p{Blank}*D{2,40}$){1,3}^\p{Blank}*. *d+\p{Blank}*$
    ^\p{Blank}*\d{4}.*$))" />
</elementInfo>
</objectInfo>
</infoSystem>
<preferences fileObserverSleepTime="3600" fileProcessorPriority="5" />
</configuration>
```

D. Aufgabenstellung

D.1. Introduction

The goal of the *All4u* interdisciplinary project is to develop a tool and a framework that allows its users a new approach to their personal information space.

Within *All4u*, this diploma project is a system sided foundation stone. It aims at the development of a semantic file system (SFS), possibly designed as a layer on top of an existing hierarchical file system (HFS).

The goals of this diploma project are hence the design and development of functionality that allows for

- generic, not file-format dependent classification of documents
- key-word based access to document content

As a proof of concept, a simple file browser shall be developed, which provides an easy to use interface to the SFS functionality.

D.2. Brief characterization of a SFS

A SFS allows its users a flexible, content sensitive access to documents. Generally spoken, a SFS comprises:

- The analysis and classification of files and extraction of file information,
- Information storage in adequate index structures,
- Access to the data/documents through a simple query language.

In order to be able to cope with the many existing file formats, the SFS has to be designed as generally as possible keeping easy extensibility and configurability in mind.

D.3. Project plan

The project consists of 5 major parts:

- Design of a description language for file classifiers and class attributes.
- Implementation of several file classifiers.
- Design and implementation of extraction and storage mechanisms
- Design and implementation of a query language.
- Implementation of a file browser with its query conversion and result presentation functionality.

Open-ended part:

- Design and implement an easy, GUI based approach to the definition of new file classes and class attributes.
- Design and implement an approach to generic attributes extraction for unknown formats in collaboration with the user.

Literaturverzeichnis

- [1] CAREY, MICHAEL J., DAVID J. DEWITT, MICHAEL J. FRANKLIN, NANCY E. HALL, MARK L. MCAULIFFE, JEFFREY F. NAUGHTON, DANIEL T. SCHUH, MARVIN H. SOLOMON, C.K. TAN, ODYSSEAS G. TSATALOS, SETH J. WHITE und MICHAEL J. ZWILLING: *Shoring Up Persistent Applications*. Technischer Bericht, Computer Sciences Department, University of Wisconsin - Madison, 1994.
<http://www.cs.wisc.edu/shore/>.
- [2] DCMI, *Dublin Core Metadata Initiative*.
<http://www.dublincore.org/>.
- [3] GARCIA-MOLINA, HECTOR, JOACHIM HAMMER, KELLY IRELAND, YANNIS PAPA-KONSTANTINOU, JEFFREY ULLMAN und JENNIFER WIDOM: *Integrating and Accessing Heterogenous Information Sources in TSIMMIS*. Technischer Bericht, Department of Computer Science, Stanford University, 1995.
- [4] *Ghostscript*.
<http://www.cs.wisc.edu/~ghost/>.
- [5] GIFFORD, DAVID K., PIERRE JOUVELOT, MARK A. SHELDON und JAMES W. O'TOOLE JR: *Semantic File Systems*. Technischer Bericht, Programming Systems Research Group MIT Laboratory for Computer Science, 1991.
- [6] HARDY, DARREN R., MICHAEL F. SCHWARTZ und DUANE WESSELS: *Harvest User's Manual*, 1996.
- [7] *HFS, Macintosh Filesystem*.
<http://developer.apple.com/techpubs/mac/Files/Files-2.html>.
- [8] *ID3-Tags*.
<http://www.id3.org/>.
- [9] *Java*.
<http://java.sun.com/>.
- [10] *Lambda-DB*.
<http://lambda.uta.edu/lambda-DB/manual/>.

- [11] *MP3, MPEG Audio Layer 3.*
<http://www.mpeg.org/MPEG/mp3.html>.
- [12] *NTFS, New Technology Filesystem.*
<http://www.digit-life.com/articles/ntfs/>.
- [13] *ODMG, Object Data Management Group.*
<http://www.odmg.org/>.
- [14] *Ozone Database.*
<http://www.ozone-db.org/>.
- [15] *Paradise, A Parallel Database System for GIS Applications.*
<http://www.cs.wisc.edu/paradise/>.
- [16] *PDF Reference. File Format Specification.*
<http://partners.adobe.com/asn/developer/technotes/acrobatpdf.html>.
- [17] *Predator, Enhanced Data Type Object-relational DBMS.*
<http://www.cs.cornell.edu/database/predator/predator.html>.
- [18] *RTF File Format Specification.*
<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnrtfspec/html/rftspec.asp>.
- [19] *SELFHTML, HTML Dateien selbst erstellen.*
<http://selfhtml.teamone.de/>.
- [20] SHKLAR, LEON, AMIT SHETH, VIPUL KASHYAP und KSHITIJ SHAH: *InfoHarness: Use of Automatically Generated Metadata for Search and Retrieval of Heterogenous Information*, 1995.
- [21] TANENBAUM, ANDREW S.: *Moderne Betriebssysteme*. Prentice Hall International, Hanser, 1995.
- [22] *UDA, Microsoft Universal Data Access.*
<http://www.microsoft.com/data/>.
- [23] VASUDEVAN, VENU und PAUL PAZANDAK: *Semantic File Systems*, 1997.
<http://www.objs.com/survey/OFSExt.htm>.
- [24] *W3C, World Wide Web Consortium, Interoperable Web Technologies.*
<http://www.w3c.org/>.
- [25] *Wotsit, The Programmer's File Format Collection.*
<http://www.wotsit.org/>.

Index (Typen, Java-Klassen und -Interfaces)

Address, 82

Brief, 83

ConfigEditor, 54

ConfigManager, 54

ConfigPreferences, 36

ContentExtractor, 68, 79, 82

ContextBuffer, 39

date, 22, 23, 26, 30, 31

DateExtractor, 68

DefaultExtractor, 65, 68

DefaultFilter, 65, 67

Document, 23–25, 29–32, 40–42, 44, 46, 48, 55, 56, 61, 62, 75, 77–80

ElementInfo, 36, 43, 46, 47, 72

ElementPath, 24

ElementPathNode, 24

Executer, 67, 68, 82, 85

ExtractorInfo, 36, 37, 43, 46, 47, 58

Field, 29

File, 23, 24, 42, 48, 55, 56, 78, 79, 81, 82

FileAttributeExtractor, 68, 79

FileNameFilter, 57, 58, 67, 78, 82

FileObject, 27

FileObserver, 71, 74

FileProcessor, 69, 71, 72, 74

FilterInfo, 36, 37, 41, 45, 56, 72

GIF, 78, 79

HTML, 77–79

Image, 77–79

InfoSystem, 36, 72

integer, 22, 23, 26, 30, 31

KeywordExtractor, 87

LaTeX, 77–79

Letter, 24, 25, 29–32, 42, 55, 82, 83

Log, 53

Mail, 77–79

Message, 78

MP3, 77–79

NoFile, 78, 81, 82

NumberExtractor, 68

ObjectInfo, 36, 72

ObjectQuery, 31

ObjectSystem, 28, 31

ObjectType, 22, 23, 25

ObjectTypeInfo, 36, 41, 43, 46

Paper, 82–84

PatternExtractor, 67, 68, 79, 83

PatternFilter, 67, 79, 83

PDF, 77–79, 81, 82

PDFInfoExtractor, 67, 68, 79, 89

Person, 23, 24, 30, 32, 55, 78

PluginLoader, 68

PluginManager, 68

ProcessingContext, 39

PS, 81, 82

QueryElement, 31

Replacer, 68
Root, 23, 25, 43, 55, 78, 81
RTF, 55, 56, 59, 75, 77–79, 81, 82
RTFTextExtractor, 67, 68, 82

SFSConfiguration, 36
SFSExtractor, 65, 69
SFSFile, 33
SFSFilter, 65, 68
SFSPlugin, 65, 68
SimpleType, 22, 25
Song, 77–79
Splitter, 68
string, 22–24, 26, 30, 31, 78, 84

Text, 23–25, 30, 32, 48, 55, 82, 83
TextFile, 81, 82
TXT, 81, 82
Type, 22, 24
Type, 56
TypeSystem, 22, 23, 36

Value, 30