jMeta 0.5
Design

Version: 0.1
Status: - Draft -

May 24, 2016

Contents

List of Figures

List of Tables

1. Einleitung

Dieses Dokument spezifiziert das technische Design von `jMeta 0.5`.

Part I.

Analyse

Dieses Kapitel definiert die wesentlichen funktionalen und nicht-funktionalen Anforderungen an `jMeta`. Wesentlicher Input ist das Dokument [MetaComp].

2. Umfang

`jMeta` ist eine Java-Library zum Lesen und Schreiben von beschreibenden Daten (Metadaten). `jMeta` hat folgende Ziele:

- Eine generische, erweiterbare Schnittstelle zum Lesen und Schreiben von Metadaten zu definieren
- Zu einem Standard in Sachen Metadatenverarbeitung für Audio-, Video- und Bildformate zu werden

Damit visiert `jMeta` als Verwender Applikationen im Multimedia-Editing-Bereich an, beispielsweise Software zur Verwaltung einer Audio- und Videosammlung.

Besondere Stärke der Library soll ihre Vielseitigkeit (im Sinne unterstützter Formate) und Erweiterbarkeit sein. Gleichzeitig soll sie Zugriff auf alle Features der unterstützten Formate gewähren.

Eine Applikation soll wahlweise sehr generisch auf Metadaten zugreifen können, oder aber die Spezifika eines speziellen Formates sehr konkret nutzen können.

Andere Arten von Metadaten, die nicht für Audio-, Video- oder Bildformate gedacht sind, sind nicht Ziel von `jMeta`, insbesondere gilt dies für spezielle XML-Metadatenformate.

2.1. Features der Version `0.5`

Neben den noch zu besprechenden Ausschlüssen ("4 Anforderungen und Ausschlüsse"), die ganz klar sagen, was NICHT unterstützt wird, geben wir hier einen Überblick, welche Features

2.1.1. Unterstützte Metadatenformate

Die aktuelle Version unterstützt folgende Metadatenformate:

- ID3v1 und ID3v1.1
- ID3v2.3
- APEv2
- Lyrics3v2

2.1.2. Unterstützte Containerformate

- MPEG-1 Audio (MP3)

2.1.3. Unterstützte Eingabemedien

Die aktuelle Version unterstützt folgende Eingabemedien:

- Datei
- Java `InputStream`
- Java byte-Array

2.1.4. Unterstützte Ausgabemedien

Die aktuelle Version unterstützt folgende Eingabemedien:

- Datei
- Java byte-Array

2.2. Features für spätere Versionen

Die folgenden Features werden für Version `0.5` der Library noch nicht umgesetzt, sondern in späteren Versionen hinzugefügt:

- Die format-spezifischen High-Level APIS
- ...

2.3. Multimedia-Lizenzierung

Open Issue 2.1: – **Intro section MultimediaLicensing**
Intro section MultimediaLicensing

2.4. Verwandte Libraries

Hier werden Java libraries behandelt, die verwandt zu `jMeta` sind. Hierunter fallen existierende Java-Libraries im Umfeld Multimedia, die einerseits Konkurrenz darstellen, andererseits Anregungen liefern und ggf. Möglichkeiten für Integration und Adaption bieten.

2.4.1. Metadaten-Libraries

Eine Auswahl von Libraries die Zugriff auf Multimedia-Metadaten ermöglichen - Eine Quelle dafür ist beispielsweise `http://id3.org/Implementations`:

- **mp3agic (Java):** Lesen und Schreiben von ID3 Tags (1.0, 1.1, 2.3, 2.4), Lesen von ID3 v2.2, low-level Lesen von MP3-Dateien (inkl. VBR) - `https://github.com/mpatric/mp3agic`

- **BeagleBuddy (Java):** Lesen und Schreiben von ID3 Tags (1.0, 1.1, 2.3, 2.4), Lyrics3v2, Lyrics3v1, APEv1, APEv2, Lesen von MP§-Dateien CBR und VBR, Xing, LAME, und VBRI Header - http://www.beaglebuddy.com/
- **jaudiotagger (Java):** Audio-Metadaten - http://www.jthink.net/jaudiotagger/
- **Entagged (Java):** Audio-Metadaten - http://entagged.sourceforge.net/
- **MyID3 (Java):** Audio-Metadaten (ID3) - http://www.fightingquaker.com/myid3/
- **JID3 (Java):** Audio-Metadaten (ID3) - https://blinkenlights.org/jid3/
- **Javamusictag (Java):** Manchmal auch **jid3lib** für Audio-Metadaten (ID3) - http://javamusictag.sourceforge.net/, https://java.net/projects/jid3lib
- **id3lib (C/C++):** Audio-Metadaten (ID3) - http://id3lib.sourceforge.net/
- **Mutagen (Python):** Audio-Metadaten und -Container-Daten (ID3) - http://pypi.python.org/pypi/mutagen/1.12
- **jFlac (Java):** Audiodaten und Metadaten des Flac-Formates - http://jflac.sourceforge.net/apidocs/index.html?org/kc7bfi/jflac/metadata/VorbisComment.html
- **MPEG-7 Audio Encoder (Java):** Erzeugen von MPEG-7 Metadaten - http://mpeg7audioenc.sourceforge.net/
- **JAI Image I/O (Java):** Kann EXIF-Tags aus unterschiedlichsten Formaten lesen und schreiben - https://jai-imageio.dev.java.net/binary-builds.html
- **jmac (Java):** Library, die monkey audio codecs encoden und decoden kann, ebenso APE-Tags lesen unds schreiben - http://sourceforge.net/projects/jmac/
- ... und einige andere

Als größte Konkurrenz werden derzeit mp3agic und BeagleBuddy betrachtet.

Nachteile existierender Metadaten-Libraries

Jede der genannten Libraries spezialisiert sich auf nur wenige Formate. D.h. Applikationen wird das eben nur reichen, wenn sie auch nur auf diese Formate beschränkt sind. Ansonsten kann es sein, dass man mehrere völlig verschiedene Libraries nebeneinander nutzen muss.

Die Architektur und Erweiterbarkeit einiger der existierenden Libraries ist nicht überzeugend.

Die ID3 Libraries z.B. zeigen ihre Internas sehr offenherzig, was es für den Anwender unklar macht, was wirklich "public API" ist, und was nicht. Verwender der Library könnten daher versehentlich oder auch bewusst (um einen Fehler "umschiffen" oder eine Funktionalität besser nutzen zu können) Implementierungsklassen nutzen. Dies wiederum macht es für die Library-Entwickler potentiell gefährlich, die Implementierung der Libraries zu refaktorisieren oder gar auszutauschen, ohne die Anwender der Libraries mit solchen Migration zu belasten.

Anwendungen, die viele unterschiedliche Formate benötigen, müssen eine generische Erweiterbarkeit selbst herstellen, da die Libraries dies nicht von Haus aus bieten.

Vorteile von `jMeta`

Es gibt bereits so viele Libraries, die alle mehr oder weniger zu gebrauchen sind. Warum also `jMeta`?

Hierfür gibt es mehrere Gründe:

- Anwendungen, deren Kern-Eigenschaft die Erweiterbarkeit und Formatvielfalt ist, müssen nicht dutzende völlig verschiedenartige Libraries mit unterschiedlichen Programmiermodellen nutzen.
- Zudem müssen solche Anwendungen kein eigenes Erweiterbarkeitsframework bauen, um die Formate zu unterstützen, sondern können sich auf das Framework von `jMeta` stützen.
- `jMeta` bietet von Haus aus Implementierungen für eine Vielfalt an Multimedia-Formaten an, und ist nicht nur auf Audio-Formate wie MP3, oder Ogg oder FLAC oder WAV (Audio) beschränkt.
- Dennoch ist es modular, es erfordert es nicht, dass Anwendungen, die lediglich Audio-Formate unterstützen wollen, Video- oder Bild-Formate der Library mitnutzen, sondern eine selektive AUswahl der benötigten Formate ist möglich.
- `jMeta` bietet eine leicht erlernbare, bequeme aber gleichzeitig überschaubare Schnittstelle für verwendende Anwendungen an.

2.4.2. Multimedia Libraries

Kann `jMeta` mit einigen üblichen Java-Multimedia-Libraries kooperieren? Dies wird in den folgenden Abschnitten geklärt.

JMF

Das Java Media Framework ist eine offizielle Java library, die mit J2SE *desktop* technology ausgeliefert wird. Die aktuelle Version 2.1.1e wurde 2001 veröffentlicht. JMF kann von Client- und Serveranwendungen verwendet werden. JMF wurde

mit MP3 Decoder und Encoder bis 2002 geliefert, aber wegen Lizenzierungsproblemen wieder entfernt. Seit 2004 gibt es nur noch ein MP3 Playback-only Plug-in.[1]

Inzwischen ist JMF aber arg in die Jahre gekommen und wird wegen der großen Konkurrenz (genannt werden Adobe Flex, Xuggler etc.) wohl nur noch selten verwendet. Allerdings gibt es FMJ als Open-Source-ALternative, die API kompatibel ist: http://www.fmj-sf.net/.

JMF kommt in vier JAR-Dateien:[2]

- JMStudio: Simple Multimedia-Player-Applikation
- JMFRegistry: Eine Anwendung die das Verwalten verschiedenster JMF-Einstellungen und Plug-ins ermöglicht
- JMFCustomizer: Erlaubt das Erzeugen einer einfacheren JMF-Jar-Datei, die nur diejenigen JMF-Klassen enthält, welche die Client-Applikation wirklich benötigt, deswegen die Auslieferungsgröße verringert
- JMFInit: Inititialisiert eine JMF-Applikation

JMF enthält platform-spezifische *performance packs*, d.h. optimierte Pakete füe Betriebssysteme wie Linux, Solaris oder Windows.

Features: JMF kümmert sich um Zeit-basierte Medien. Die JMF features kann man wie folgt zusammenfassen:[3]

- Capture: Multimedia frame-Daten eines gegebenen Audio- oder Video-Signals lesen und es in einen spezifischen Codec in Echtzeit codieren.
- Playback: Multimedia-Daten abspielen, d.h. deren Bytes auf dem Bildschirm darstellen oder an die Audio-Ausgabegeräte weitergeben.
- Stream: Zugriff auf Stream-Inhalte
- Transcode: Konvertieren von Medien-Daten eines gegebenen digitalen Codecs in einen anderen, ohne zuerst decodieren zu müssen.

Kritik: [WikJMF] fasst einiges negatives Feedback zur JMF library zusammen:

- Eine Menge Codecs wie MPEG-4, MPEG-2, RealAudio und WindowsMedia werden nicht unterstützt, MP3 nur über ein Plug-in
- Keine Wartung und Weiterentwicklung der Library durch Sun oder Oracle
- Keine Editing-Funktionalität[4]
- Performance packs nur für einige wenige Plattformen

[1] Siehe [WikJMF].
[2] Siehe [WikJMF].
[3] Siehe [JMFWeb].
[4] D.h. das Bearbeiten von Multimedia-Content.

Basiskonzepte der API Lesen der Multimediadaten wird in Form von `DataSources` abstrahiert, während Ausgaben in `DataSinks` geschrieben werden. Keine Spezifika unterstützter Formate werden bereitgestellt, vielmehr können unterstützte Formate abgespielt, verarbeitet und exportiert werden, wobei nicht alle Codecs Support für Verarbeitung und transcoding bieten. Eine `Manager`-Klasse ist die primäre API für JMF-Anwender.[5]

Die API-Dokumentation zeigt, dass JMF sehr komplex und im Wesentlichen zeit- und event-basiert ist.[6] Es gibt Möglichkeiten, rohe Bytedaten über die Methode `read` des interfaces `PullInputStream` zu lesen. Jedoch kontrolliert JMF die Verarbeitung ausgehend von der Quelle, z.B. entweder einer Datei oder einen Stream.

Vergleich mit `jMeta`: Es scheint als gäbe es keine sinnvolle Kooperation zwischen `jMeta` und JMF. Es gibt Ähnlichkeiten und Unterschiede. Beide können Datenquellen und -Senken handhaben, also Mediendaten in verschiedensten Formaten lesen und schreiben. `jMeta` bringt zusätzlich weitgreifende Unterstützung für Metadaten-Formate und für nicht-audio-video Container-Formate mit. Jedoch ist sie im Vergleich zu JMF eher eine primitve Library, in dem SInne dass sie nur Zugriff auf die Daten liefert, ohne weiteres Framework für dessen Verarbeitung. `jMeta` liefert die raw bytes und Metadatn, die Anwendung kann sie verwenden wie gewünscht, z.B. diese abspielen, transcodieren oder was auch immer. JMF bietet diese Zusatzschritte und mag dafür holistischer erscheinen. `jMeta` ist mehr als Basisbilbiothek zu betrachten, die sich auf Metadaten spezialisiert, und man hätte `jMeta` zum Implementieren von JMF nutzen können.

JavaSound

JavaSound ist Oracle's Sound-Verarbeitungs-Library. Sie hat einige Gemeinsamkeiten mit JMF, kann aber als low-level betrachtet werden, da sie auch mehr Manipulations-Funktionalitäten für die Audio-Daten bietet. Sie unterstützt auch MIDI-Geräte.[7]

Basiskonzepte: JavaSound bietet im Wesentlichen die Klassen `Line`, die ein Element in der Audio-Pipeline repräsentiert, die abgeleiteten Klassen `Clip` für das Abspielen von Audio-Daten sowie `Mixer` für das Manipulieren der Audio-Daten an. Sie kann aus Streams ebenso wie aus Dateien oder rohen Bytes lesen. Sie unterstützt desgleichen Konvertierung zwischen verschiedenen Datei-Formaten.

Vergleich mit `jMeta`: Wie JMF kümmert sich JavaSound um das Abspielen und Verarbeiten der Audio-Daten statt um das Lesen der verfügbaren Detailinformationen. Hier kommt `jMeta` ins Spiel. `jMeta` liest nahezu alle spetifischen Informationen, inklusive Metadaten. Jedoch liefert es lediglich rohe Audio-Nutzedaten, und überlässt es dem Nutzer, diese zu verarbeiten und abzuspielen. `jMeta` und

[5] Siehe [WikJMF].
[6] Siehe [JMFDoc].
[7] Siehe [WikJavaSound].

JavaSound sind daher Wettbewerber im Sinne des Lesens und Schreibens von Audio-Daten. `jMeta` liefert deutlich mehr Detailinformationen aus den gelesenen Daten, inklusive jedweder eingebetteter Metadaten, während JavaSound die Medien abspielt - d.h. es fokussiert sich auf dessen eigentlichen Zweck. Jedoch gibt es einen Weg, über den beide kooperieren können - Das modell könnte so aussehen:

- `jMeta` liest alle Informationen inklusive Audio-Frames und Metadaten.
- Die Audio-Frames werden - wenn unterstützt - an JavaSound als Raw-Bytes gesendet, wie in Folgendem-Beispiel:[8]

```
<Read all tags here using jMeta>

SourceDataLine  line = null;
DataLine.Info   info =
   new DataLine.Info(SourceDataLine.class, audioFormat);

try
{
   line = (SourceDataLine) AudioSystem.getLine(info);

   /*
     The line is there, but it is not yet ready to
     receive audio data. We have to open the line.
   */
   line.open(audioFormat);
}
catch (LineUnavailableException e)
{
   e.printStackTrace();
   System.exit(1);
}
catch (Exception e)
{
   e.printStackTrace();
   System.exit(1);
}

/*
  Still not enough. The line now can receive data,
  but will not pass them on to the audio output device
  (which means to your sound card). This has to be
  activated.
*/
line.start();

while (<Read frames using jMeta>)
{
   int nBytesWritten = line.write(abData, 0, nBytesRead);
}
```

[8] Das Beispiel ist aus [JavaSoundSample] entnommen und leicht verkürzt worden.

Dieser Ansatz ist jedoch nicht sehr befriedigend, da er einen Performance-Overhead erzeugt, wenn es um das Abspielen oder transcoding von Multimedia-Inhalten geht. Daher scheinen JavaSound und `jMeta` nur entweder-oder einsetzbar zu sein.

3. Grundlegende Begriffe

Hier definieren wir einige grundlegende Begriffe, die im gesamten Design-Konzept verwendet werden. Viele davon stammen aus [MetaComp], Seiten 19 bis 29, wo noch deutlich mehr Begriffe definiert werden.

Alle Begriffe basieren im Wesentlichen auf einem Domänenmodell für Container- und Metadaten, wie es in [MetaComp] definiert ist. Ein für unsere Zwecke erweitertes Domänenmodell ist in folgender Abbildung dargestellt - sie stellt alle relevanten Begriffe und ihre Beziehungen zueinander als Überblick dar:

Open Issue 3.1: **– Add domain model figure**

3.1. Metadaten

In diesem Dokumen verstehen wir mit dem Begriff *Metadaten* in erster Linie beschreibende Daten, die aber nicht für das Parsen notwendig sind. Metadaten beschreiben andere Daten semantisch und strukturell. Das Ziel von `jMeta` ist insbesondere das Auslesen von Metadaten zu Audio- und Video-Datensätzen, beispielsweise Titel, Komponist etc. Die Struktur solcher Metadatenformate wird durch Metadaten-Formate definiert.

Geht es um speziell um (technische) Metadaten, die zum Parsen einer Datenstruktur beispielsweise in einem Container-Header notwendig sind, reden wir von *Parsing-Metadaten*.

3.2. Daten-Formate, Metadaten-Formate und Container-Formate

Ein Daten-Format definiert Struktur und Interpretation von binären oder textuellen Daten: Welche Zeichen oder Bytes mit welchen Werten in welcher Reihenfolge haben welche Bedeutung? Üblicherweise beschreiben diese Formate in ihren Spezifikationen wie anhand von Blöcken aufeinanderfolgender Bits und Bytes das Datenformat erkannt werden kann, und wie diese Blöcke in einzelne Felder mit bestimmter Bedeutung und Wertemenge zerfallen. Dabei besteht ein solcher Block von Bytes für uns aus (später definierten) Datenblöcke und Felder.

Metadaten-Formate sind Datenformate, die die Struktur digitaler Metadaten definieren. Beispiele sind:

- ID3v1
- ID3v2.3

- APEv1
- MPEG-7
- RDF/XML
- VorbisComment
- und andere ...

CONTAINER-FORMATe sind DATEN-FORMATe, die für das Speichern, Transportieren, Editieren, Suchen und anderweitige Verarbeiten von PAYLOAD Daten (zumeist Multimedia-Inhalte, also Audio, Video, Bilder oder Text) optimiert sind. Beispiele sind:

- MP3
- Ogg
- TIFF
- QuickTime
- JPEG 2000
- PDF
- und andere ...

Ein Beispiel für ein DATEN-FORMAT, das weder als METADATEN-FORMAT noch als CONTAINER-FORMAT betrachtet werden kann, ist HTML. XML ist ein DATEN-FORMAT das wiederum selbst benutzt werden kann, um andere XML DATEN-FORMATe zu definieren. Einige XML DATEN-FORMATe sind auch METADATEN-FORMATe, z.B. MPEG-7, MPEG-21 oder P_Meta.

3.3. Transformationen

Ein DATEN-FORMAT kann sogenannte TRANSFORMATIONen definieren. Eine TRANSFORMATION beschreibt eine Methode, wie gelesene oder zu schreibende Daten transformiert werden müssen, um bestimmte Ziele zu erfüllen. Man kann sich dies also als eine Art Codierung der Daten vorstellen. Im Gegensatz zur festen Datenformat-Spezifikation, die genau beschreibt, wie binäre Daten codiert und zu interpretieren sind, sind TRANSFORMATIONen optionale Features, die dynamisch für bestimmte Bereiche der Daten angewendet werden können oder auch nicht. Teilweise können TRANSFORMATIONen auch durch Anwender definiert werden. Beispiele sind die durch ID3v2 definierten Transformationen: Unsynchronization, Verschlüsselung und Kompression.

3.4. Datenblöcke

Als DATENBLOCK wird hier eine Folge von Bytes verstanden, die gemeinsam eine logische Einheit im Sinne des unterliegenden DATEN-FORMATs bilden. Jeder DATENBLOCK gehört also zu genau einem DATEN-FORMAT. Er hat eine Länge in Bytes. Wir unterscheiden verschiedenste konkrete Typen von DATENBLÖCKE die in den folgenden Abschnitten beschrieben werden.

3.4.1. Container: Payload, Header, Footer

Der wichtigste Typ von DATENBLOCK ist der CONTAINER: Er besteht aus einem oder mehreren optionalen HEADERn, genau einer PAYLOAD und einem oder mehreren optionalen FOOTERn. HEADER, PAYLOAD und FOOTER sind ebenso Typen von DATENBLÖCKEn, in dem Fall also Kind-DATENBLÖCKE eines CONTAINER-DATENBLOCKs.

CONTAINER sind ein verbreitetes Konzept zum Speichern von Multimedia- und Metadaten: Der HEADER beschreibt wichtige Eigenschaften des CONTAINERs, wie dessen Typ, Größe und viele andere Eigenschaften (die sogenannten Parsing-Metadaten). Die PAYLOAD (dt. Nutzdaten) enthält die interessanten Daten, beispielsweise Multimedia-Daten, die abgespielt werden können. Ein FOOTER erlaubt Rückwärts-Lesen. Die meisten CONTAINER-FORMATs spezifizieren die allgemeine Struktur eines CONTAINERs. Manche Formate erlauben dann auch das Hinzufügen benutzerdefinierter CONTAINER, die dem definierten Grundformat entsprechen, d.h. das Format ist dann erweiterbar.

3.4.2. Tag

Ein TAG ist ein spezieller CONTAINER, dessen Zweck darin besteht, beschreibende digitale Metadaten zu speichern. Das TAG kann entweder zu einem eigens definierten METADATEN-FORMAT gehören, oder aber im Rahmen eines CONTAINER-FORMATs definiert sein. Speziell bei Audio-Metadaten wird dieser Begriff häufig benutzt, Beispiele sind hier die ID3 oder APE-TAGs.

Die folgende Abbildung zeigt die Basisstruktur eines TAGs, inklusive weiterer Begriffe:

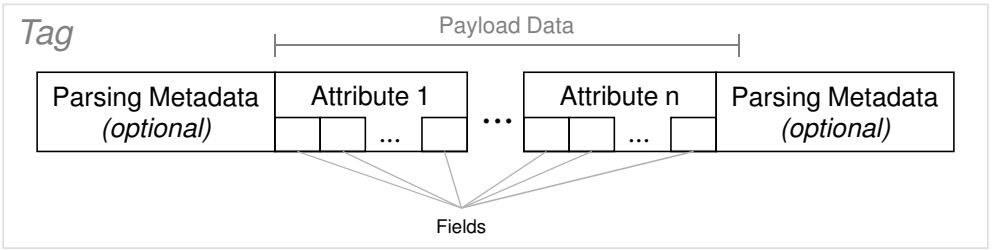

Figure 3.1.: Struktur eines TAGs

Die wichtigsten Teile eines TAGs bilden die ATTRIBUTe.

3.4.3. Attribut

Ein ATTRIBUT ist ein Teil eines TAGs, das die wertvollen Metadaten enthält als key-value-Paar enthält. Bekannte Beispiele sind Künstler, Titel, Album, Komponist etc. eines Musikstücks. Häufig ist ein ATTRIBUT auch ein CONTAINER, hat also ggf. einen HEADER, FOOTER und Nutzdaten. Der HEADER hilft meist dabei, den Typen (Künstler, Title, oder Album) des ATTRIBUT sowie die Größe von dessen Werte zu speichern. Die PAYLOAD enthält die jeweilige Information in einer kodierten Form, z.B. den Namen des Künstlers oder Titels des Musikstücks.

Die meisten ATTRIBUTe haben nur einen unstrukturierten einfachen Wert. In manchen Fällen gibt es aber auch komplexer strukturierte ATTRIBUT-Werte die aus mehreren Teilen in Form von Kind-FELDern oder gar DATENBLÖCKE bestehen.

In jedem Metadaten-Format hat ein ATTRIBUT einen speziellen Namen, hier einige Beispiele:

- ID3v1, Lyrics3: Field
- ID3v2: Frame
- APE: Item
- Matroska: SimpleTag
- VorbisComment: User Comment

In CONTAINER-FORMATen sind die ATTRIBUTe häufig CONTAINER die im jeweiligen CONTAINER-FORMAT definiert sind.

3.4.4. Felder

Ein FELD ist eine Sequenz von Bits, die zusammen eine spezielle Bedeutung in einem gegebenen DATEN-FORMAT haben. Das DATEN-FORMAT beschreibt, wie ein spezieller DATENBLOCK durch eine Folge von FELDern aufgebaut ist. Ein FELD hat einen Wertebereich und es wird auch definiert, wie diese Werte jeweils zu interpretieren sind. Oft wird ein Teil des Wertebereiches als "reserviert" definiert, um eine gewisse Erweiterbarkeit des Datenformats sicherzustellen.

3.4.5. Subjekt

Ein SUBJEKT bezeichnet das Ding, das durch ein TAG beschrieben wird, d.h. einen Teil einer Datei, oder eines Musikstückes, oder einer Web-Ressource or sogar eines real existierenden Objektes. Häufig enthält ein TAG Metadaten, die sich auf das aktuelle MEDIUM als SUBJEKT beziehen, es wird nicht explizit auf ein spezielleres SUBJEKT referenziert.

3.5. Medium

Ein MEDIUM bezeichnet Speichermedium der DATENBLÖCKE. Es kann sich dabei beispielsweise um eine Datei oder einen MEDIEN-STREAM, oder gar den Hauptspeicher selbst handeln.

4. Anforderungen und Ausschlüsse

Hier werden all expliziten Anforderungen an die Bibliothek `jMeta` in Version `0.5` sowie auch explizite Ausschlüsse dargestellt. Ausschlüsse dienen dafür, explizit klarzumachen, welche Themen (auf potentiell bliebig lange Zeit) nicht unterstützt werden. Davon abzugrenzen sind bereits bekannte (ggf. sogar notwendige) Erweiterungen, die aber nicht in dieser Version verfügbar sind, sondern absehbar in einer folgenden Version umgesetzt werden sollen. Das sind (potentielle) Features, die auf eine spätere Version verschoben worden sind. Diese sind im Abschnitt "2.2 Features für spätere Versionen" beschrieben.

4.1. ANF 001: Metadaten Menschenlesbar lesen und schreiben

Metadaten sollen in *menschenlesbarer Form* gelesen und geschrieben werden können. D.h. der Anwender der Library wird nicht genötigt, binäre Repräsentationen der Daten zu erzeugen, um sie schreiben zu können, bzw. binäre Daten zu interpretieren.

Begründung: Dies ist eine generelle Kernfunktionalität der Library.

Eine Liste der unterstützten Formate findet sich in "2.4.2 Features:".

4.2. ANF 002: Containerformate lesen

Populäre bzw. verbreitete Containerformate müssen gelesen werden können.

Begründung: Metadaten sind oft in Containerformaten eingebettet bzw. fest in deren Spezifikation verankert. Zudem müssen Container-Segmente erkannt werden können, um sie zu überspringen und den eigentlichen Anfang der Metadaten finden zu können.

4.3. ANF 003: Spezifikation unterstützter Metadaten- und Containerformate erfüllen

Sofern eine Spezifikation eines unterstützten Metadaten- bzw. Containerformates vorliegt, muss diese vollständig unterstützt werden. Es muss vollständigen Zugriff auf alle unterstützten Features des Datenformates möglich sein.

Begründung: So wird sichergestellt, dass spezifikationskonforme Metadaten geschrieben werden, die auch von anderen Libraries bzw. Anwendungen wieder gelesen werden können. Zudem kann der Nutzer der Library alle Features des jeweiligen Formates ausnutzen, ohne wiederum allzu viel Eigenimplementierung

leisten zu müssen. Dies ist auch eine Differenzierungsmöglichkeit gegenüber anderen Libraries.

4.4. ANF 004: Zugriff auf alle Rohdaten über die Library

Zusätzlich zum Zugriff auf menschenlesbare Metadaten ("4.1 ANF 001: Metadaten Menschenlesbar lesen und schreiben") soll es ebenso möglich sein, alle Rohdaten aub Byteebene zu lesen. Es soll feingranularer Zugriff auf alle Felder der Binärdaten möglich sein.

Begründung: So können Anwender selbst ein Parsing implementieren, ohne die high-level-Funktionen nutzen zu müssen. Sie können selbst auf Byte- und Bitebene Daten manipulieren und auslesen. Ein Zugriff auf die Binärdaten ist möglich (wenn nötig), ohne wiederum einen eigenen Umweg gehen zu müssen, z.B. durch erneutes Lesen und Parsen der Daten.

4.5. ANF 005: Performance vergleichbar mit anderen Java-Metadaten-Libraries

Die Performance der Library soll gleichwertig oder besser als die anderen Java-Metadaten-Libraries sein. Hierfür müssen die Vergleichslibraries benannt und ein entsprechender Benchmark definiert und durchgeführt werden.

Begründung: Die Library soll ähnlich performant wie bestehende Lösungen der idealerweise performanter sein, um hier kein Argument gegen ihren Einsatz zu liefern.

4.6. ANF 006: Fehlererkennung, Fehlertoleranz, Fehlerkorrektur

Ergänzend zur "4.3 ANF 003: Spezifikation unterstützter Metadaten- und Containerformate erfüllen" muss die Library aber auch *fehlertolerant* sein, so weit möglich. D.h. u.a., das Spezifikationsverstöße und fehlerhafte Parsing-Metadaten erkannt werden, und dies - soweit nicht unumgänglich - nicht zum Abbruch des Parsens mit einem Fehler endet. Verstöße werden protokolliert und wenn möglich automatisch korrigiert (optional, wenn es der Library-Anwender wünscht).

Begründung: Altanwendungen oder andere Libraries schreiben Datenformate manchmal nicht 100% spezifikationskonform. Zudem sind nicht alle Spezifikationen eindeutig oder genau genug, sodass Varianten entstehen könnten. Trotz Vorliegen fehlerhafter Daten soll der Anwender der Library in die Lage versetzt werden, Daten dennoch auslesen und ggf. sogar korrigieren zu können.

4.7. ANF 007: Erweiterbarkeit um neue Metadaten- und Containerformate

Die Library muss auf komfortable Art und Weise um neue Metadaten- und Containerformate erweitert werden können. In der Mindestausbaustufe der Anforderung muss eine einfache Erweiterbarkeit durch die Library-Entwickler möglich sein, in der Maximalausbaustufe ist eine einfache Erweiterung durch jeden Anwender der Library mit Programmiererfahrung möglich.

Begründung: Es werden immer wieder neue Formate entwickelt und verfügbar. Die Erweiterbarkeit stellt eine lange Lebensdauer der Library sicher und ermöglicht zudem eine einfachere Pflege durch die Library-Entwickler. In der Maximalausbaustufe "Erweiterbar durch Endanwender" ist dies ein klares Differenzierungskriterium gegenüber Konkurrenzlibraries, die solche Erweiterbarkeit nicht bieten.

4.8. ANF 008: Lesen und Schreiben großer Datenblöcke

`jMeta` muss das Lesen und Schreiben sehr großer Datenmengen effizient unterstützen, und dabei Mechanismen verwenden, um OutOfMemoryErrors zu vermeiden.

Begründung: Besonders im Video-Bereich können teilweise gigabyte-große Nutzdaten vorkommen. Die Länge der Nutztdaten muss korrekt interpretiert werden können. Wegen "4.4 ANF 004: Zugriff auf alle Rohdaten über die Library" muss auch das Lesen und Schreiben der Nutzdaten unterstützt werden, ohne dass Speicher knapp wird, d.h. ein etappenweises Lesen und Schreiben o.ä. muss möglich sein. Dies ist auch ein Differenzierungsmerkmal zu anderen Libraries, die dies ggf. gar nicht unterstützen können.

4.9. ANF 009: Selektive Formatauswahl

Eine Anwendung, die `jMeta` verwendet, muss selektiv diejenigen Formate auswählen können, die sie unterstützen möchte und die zur Laufzeit geladen werden sollen. Dies gilt aber nicht nur für die Laufzeit, sondern auch für die Library-Pakete selbst.

Begründung: Audio-Anwendungen brauchen keine Erweiterungen für Video- oder Bildformate. Anwendungen können den Laufzeit- ebenso wie den Speicher-Overhead minimieren, indem sie nur genau die Formate wählen, die sie wirklich benötigen.

4.10. AUS 001: Lesen aus Media Streams

Ein Lesen von Metadaten oder Containerdaten aus Streams (streaming media) wird nicht explizit unterstützt. Es können im Design Möglichkeiten zur aktiven

Unterstützung vorgesehen werden, dies ist aber nicht zwingend erforderlich.

Begründung: Kombinierte Anwendungen (Recorder bzw. Player) machen für diesen Fall mehr Sinn und sind auch weitgehend verfügbar. Die zusätzliche Unterstützung für Streaming könnte das Design verkomplizieren. Es ist aktuell unklar, wie dies umzusetzen wäre.

4.11. AUS 002: Lesen von XML-Metadaten

Es gibt auch XML-Metadatenformate. Üblicherweise werden diese aber nicht in Multimedia-Daten eingesetzt, da sie sehr "verbos" sein können. Hier sind weiterhin die binären Metadatenformate die Platzhirsche. Die Unterstützung von XML wird nicht im Kern der Library vorgesehen.

Begründung: Der Versuch, sowohl binäre als auch XML-Formate über die gleiche API oder gar Implementierung zu unterstützen, kann zu einem sehr komplizierten Design führen. Java bietet viele sinnvolle Standard-Möglichkeiten zum Parsen und zum Schreiben von XML-Metadaten. Evtl. könnten diese in Ausbaustufen in einer Implementierung (hinter der gleichen API-Schnittstelle) in einer späteren Ausbaustufe unterstützt werden.

4.12. AUS 003: Keine Anwender-Erweiterung der `jMeta`-Medien

Das Erweitern von `jMeta` um neue Medien wird in der aktuellen Version nicht unterstützt.

Begründung: Die zur Verfügung stehenden Mechanismen decken bereits viele Anwendungsfälle ab. Eine Erweiterbarkeit auch um neue Medien macht `jMeta` selbst ebenso wie den Erweiterungsmechanismus nur komplexer, ohne das ein wesentlicher Mehrwert erkennbar ist. Es ist z.B. nicht klar, um welche Medien `jMeta` überhaupt erweitert werden soll. Sollten neue Medien sinnvoll sein, dann kann es einen neuen Core-Release geben, der diese Medien unterstützt.

5. Referenzbeispiele

Um die Erfüllung der `jMeta`-Anforderungen sicherzustellen, wird eine Menge von (größtenteils) real-life Beispielen für alle unterstützten Formate definiert. Diese Beispiele werden benutzt, um Designentscheidungen zu illustieren, aber auch um sie zu verifizieren. In diesem Kapitel werden diese Beispiele in Kürze definiert. Die detaillierte Struktur jedes verwendeten Datenformats wird in [MetaComp] behandelt.

Wichtig: Die Größen der Datenblöcke in den folgenden Abbildungen haben keine Bedeutung.

5.1. Beispiel 1: MP3 File mit ID3v2.3, ID3v1.1 und Lyrics3

Die folgende Abbildung zeigt das erste Beispiel, eine MP3-Datei mit drei TAGs, ID3v2.3, Lyrics3v2 und ID3v1.1. Alle befinden sich am Ende der Datei:

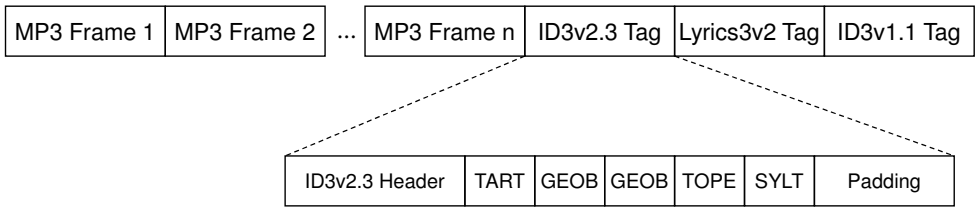

Figure 5.1.: Beispiel 1: MP3 File mit ID3v2.3, ID3v1.1 und Lyrics3

Das ID3v2.3-TAG hat mehrere Frames, einen `GEOB`-Frame inbegriffen. Weiterhin hat es ein wenig Padding am Ende. Jeder der MP3-Frames korrespondiert zu einem MPEG-1 "elementary stream audio format".

5.2. Beispiel 2: MP3 File mit zwei ID3v2.4 Tags

Die folgende Abbildung zeigt das zweite Beispiel, eine MP3-Datei mit zwei ID3v2.4 TAGs, eines am Anfang, das andere am Ende der Datei:

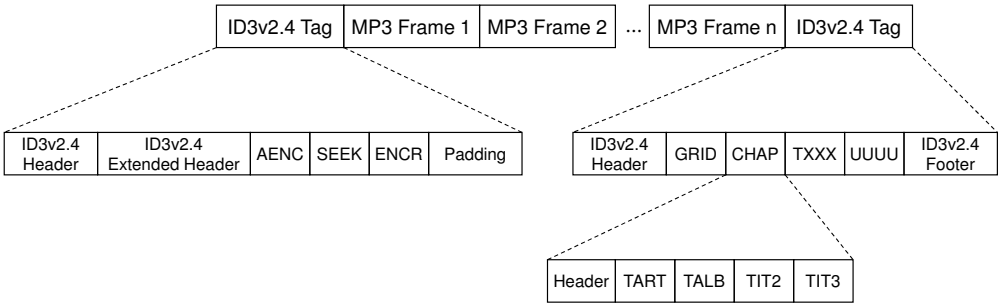

Figure 5.2.: Beispiel 2: MP3 File mit zwei ID3v2.4 Tags

Die zwei ID3v2.4 TAGs sind virtuell über einen `SEEK`-Frame verbunden. Beide haben verschiedene Spezialitäten, die in [MetaComp] beschrieben sind.

5.3. Beispiel 3: Ogg Bitstream mit Theora und VorbisComment

Die folgende Abbildung zeigt das dritte Beispiel, einen Ogg bitstream der Theora als Nutzdaten enthält, mit einem Vorbis comment:

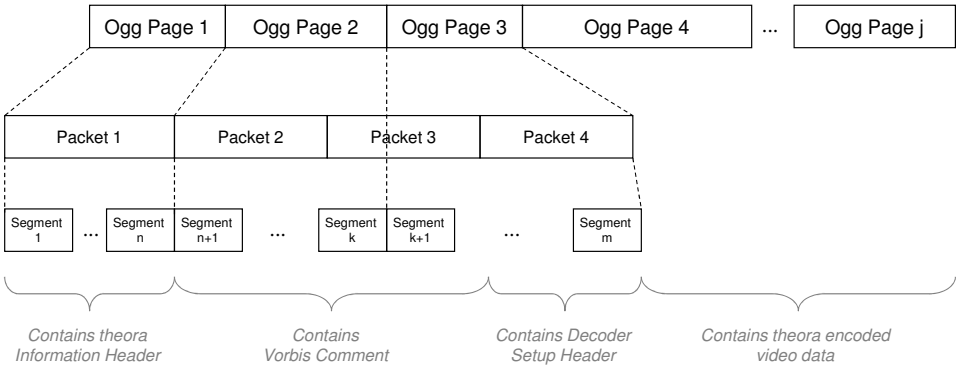

Figure 5.3.: Beispiel 4: Ogg Bitstream mit Theora und VorbisComment

Das Beispiel scheint auf den ersten Blick komplex zu sein. In einem Ogg bitstream sind physikalische und logische Struktur nicht notwendigerweise das gleiche. Die physikalische Strukture wird durch pages, packets und segments gebildet, während die logische Struktur die struktur der gewrappten Daten ist. Wir haben hier theora als Video-Daten-Beispiel verwendet, aber dies ist prinzipiell beliebig, weil der Codec für `jMeta` keine Rolle spielt. Was aber eine Rolle spielt ist die Position des Vorbis Comment, eines der unterstützten Datenformate. Dies hängt jedoch unglücklicherweise vom gespeicherten Codec ab. In diesem Beispiel startet

der vorbis comment in der zweiten Page und überspannt zwei packets. Das zweite dieser Packets überspannt zwei Ogg pages.

Part II.

Architektur

In this part, the high-level structure of `jMeta` is defined. This is done in two refinement levels and an additional perspective:

- The technical architecture shows the technical environment and parts of `jMeta`
- The functional architecture shows and describes the functional components of `jMeta`
- The development view shows the structure of the `jMeta` development artifacts
- The deployment view shows the structure of the `jMeta` deployment artifacts

6. Allgemeine Designentscheidungen

In diesem Kapitel werden allgemeine Designentscheidungen getroffen, die einen unumkehrbaren Einfluss auf die Architektur der Gesamt-Library haben. Sie definieren die Rahmenbedingungen für die Library, und dienen als Basis zur Definition ihrer technischen und vorallem fachlichen Architektur. Sie bilden auch generell und übergreifend die Basis zur Erfüllung der Anforderungen, haben daher noch keinen Bezug zu einer spezifischen Anforderung.

Die Erfüllung der spezifischen Anforderungen wird durch die detaillierten Designentscheidungen ermöglicht, die im Kapitel "8 Fachliche Architektur" und im Teil "III `jMeta` Design" definiert sind.

6.1. Verwendung von Java

DES 001: **`jMeta` basiert auf Java SE 8**
`jMeta` wird basierend auf dem "latest update" von Java SE 8 entwickelt. Ein Umstieg auf neuere Java-Versionen wird im Rahmen des Lebenszyklus der Library wiederholt in Erwägung gezogen.

Begründung: Java als Programmiersprache ist etabliert und weit verbreitet sowie plattformunabhängig. Prinzipiell ist der Portierungsaufwand zu anderen Betriebssystemen, Java ME sowie Java für Smartphones (z.B. Android) damit weitaus geringer als bei Verwendung von beispielsweise C/C++. Da der Autor langjährige Erfahrung mit Java hat, stellt deren Verwendung eine höchstmögliche Produktivität sicher. Die Konkurrenzlibraries im Java-Umfeld sind überschaubar. Die aktuell (Stand 15. März 2016) neueste Version Java 8 wird ganz klar deshalb genutzt, weil die neuesten verfügbaren Features von Sprache und Library genutzt werden sollen. Java 7 hingegen wird ab voraussichtlich April 2015 von Oracle nicht mehr mit öffentlichen Updates versorft, Support ist aber weiterhin einkaufbar. Somit kann es sein, dass öffentliche Fixes für bekannte Bugs nicht mehr für Java 7 erscheinen werden.

Nachteile: Anwendungen, die auf Java 7 oder älter basieren, werden von `jMeta` nicht mehr unterstützt. Das träfe dann insbesondere auf Java-EE-Anwendungen zu, die vielfach noch auf so innovative Produkte wie Websphere 8.0 (oder älter!) aufsetzen.

6.2. Verwendung anderer Libraries

DES 002: **`jMeta` setzt so wenig wie möglich Dritt-Libraries ein**
`jMeta` setzt weitestgehend allein auf die Libraries von Java SE auf. Es werden - im produktiven Code - keine Abhängigkeiten zu dritten Libraries genutzt.

Begründung: Zusätzliche Abhängigkeiten können zu Mehraufwänden bei der Verwaltung (Build, Deployment, Versionsmanagement) führen. Letztlich ist `jMeta` dann auch von der Lizenzierung, vom Release- und Bug-Management und den "Launen" der Library-Entwickler abhängig, was hiermit vermieden wird. Zudem wird die Anwendung insgesamt leichtgewichtiger, sowohl zur Laufzeit als auch im Hinblick auf die Auslieferungsgröße.

Nachteile: Evtl. höherer Entwicklungsaufwand, weil das Rad ab und an "neu erfunden" wird.

6.3. Komponentenbasierte Library

DES 003: **`jMeta` ist komponentenbasiert**
`jMeta` besteht aus sogenannten Komponenten (Definition siehe nächsten Abschnitt), die sich gegenseitig nur über klar definierte Schnittstellen verwenden.

Begründung: Die Untergliederung in Komponenten ermöglicht es, die Komplexität der Library über ihren gesamten Lebenszyklus hinweg besser zu beherrschen. Durch eine sinnvolle Komponentengliederung wird eine klare Aufgabentrennung, Entkopplung und eine bessere Erweiterbarkeit sichergestellt. Änderungen an einer Implementierung einer Komponente haben ein deutlich geringeres Risiko, sich auf weite Teile der Library auszuwirken, sondern werden lokal auf die Komponentenimplementierung beschränkt bleiben.

Nachteile: Ggf. etwas mehr overhead und mehr Komplexität, da Mechanismen zur Entkopplung eingesetzt werden müssen.

6.3.1. Definition des Komponentenbegriffs

Eine *Komponente* in `jMeta` ist *eine abgeschlossene Software-Einheit mit klar definierter Aufgabe*. Sie bietet *Services* an, die über eine klar *definierte Schnittstelle* genutzt werden können. Diese Services werden sowohl von den Anwendern der Library als auch von anderen Komponenten der Library genutzt. Eine Komponente hat ggf. *Datenhoheit* über bestimmte Daten, d.h. nur die Komponente selbst darf diese Daten lesen und modifizieren. Andere Komponenten müssen diese spezielle Komponente nutzen, um diese Daten zu verwenden.

Die folgende Abbildung zeigt schmeatisch eine `jMeta` Komponente:

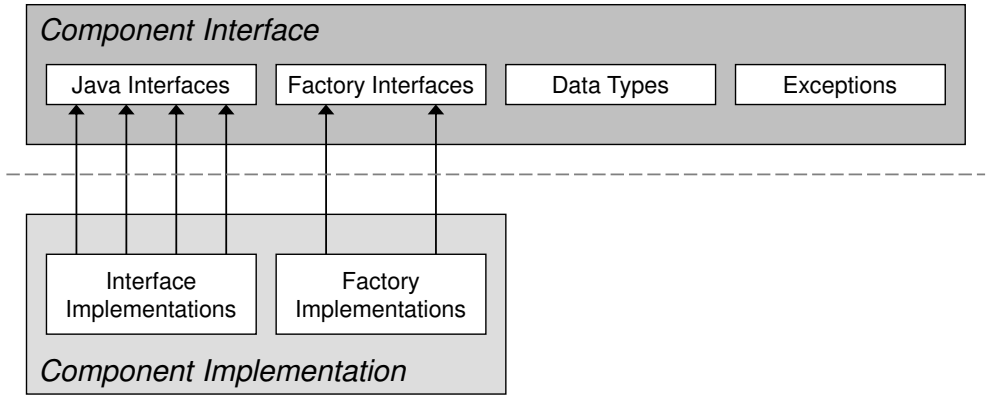

Figure 6.1.: Struktur einer Komponente

Die Komponentenschnittstelle besteht aus einem oder mehreren Java interfaces, exceptions und Datentypen.

- *Java interfaces* stellen die funktionalen API der Komponente dar, jede Methode entspricht einem Service der Komponente. Ein Java interface wird für eine klar definierte Unteraufgabe der Komponente genutzt. Manche Interfaces haben Erzeugungscharakter, geben also Zugriff auf andere Interfaces der Komponente.

- *Datentypen* sind konstante Java-Klassen, die direkt in implementierungsform zur Verfügung gestellt werden. Meist dienen sie dazu, Daten zu halten, die als Eingabe oder Rückgabe verwendet werden.

- *Exceptions* sind Fehler die auf funktionelle Fehler hindeuten. Es handelt sich um geprüfte Exceptions, die an der Service-Schnittstelle definiert werden und vom Anwender behandelt werden müssen.

6.3.2. Designentscheidungen zur Verwendung von Komponenten

DES 004: **Entkoppeln von Komponenten über ein leichtgewichtiges Service-Locator-Pattern**
Um geringe Kopplung zwischen den Komponenten zu erreichen, dürfen sich diese nur über ihre Komponentenschnittstellen kennen. Dies trifft auch für das Erzeugen anderer Komponenten bzw. das Erlangen einer Referenz auf ein anderes Komponenteninterface zu. Um dies zu erreichen, wird ein leichtgewichtiges Service-Locator-Pattern in Form einer eigenen Utility verwendet.

Begründung: Wir wollen vernünftige Entkopplung zwischen Komponenten, und daher brauchen wir eine entsprechende Utility. Diese soll leichtgewichtig sein, damit scheiden Java EE und Spring aus, Drittlibraries wie Google Guice ebenso wegen DES 004.

Nachteile: Keine erkennbar

DES 005: **Singleton-Komponenten**
Jede Komponente hat eine einzelnes *Zugriffs-Java-interface*, das wiederum Zugriff auf all Services der Komponente gewährt. Dazu kann das Zugriffsinterface Instanzen vefschiedener anderer Klassen oder Interfaces zurückliefern, mit denen der Aufrufer dann arbeiten kann.
Aus einer Laufzeit- und Implementierungsperspetive hat jedes der Zugriffs-Java-Interfaces eine Art "singleton"-Implementierung. Es darf also nur eine Instanz der Zugriffs-Java-Interfaces einer Komponente geben. Natürlich darf es im Kontrast dazu mehrere Instanzen jedes anderen Interfaces geben, dass die Komponente definiert.

Begründung: Die Zugriffs-Java-Interfaces sind nur funktional, und da es nur genau eine Komponentenimplementierung in `jMeta` je Komponente gibt, hat man immer nur eine implementierende Klasse. Für diese ist zur Laufzeit nur eine Instanz erforderlich, da sie keine Zustände hält. Dies spart Speicherplatz und Initialisierungsaufwand.

Nachteile: Keine erkennbar

DES 006: **Unterteilung in Subsysteme**
Eine Ebene über den Komponenten untergliedern wir `jMeta` noch in sogenannten *Subsysteme*. Ein Subsystem zerfällt in Komponenten, und hat sonst keine anderen Inhalte. Es ist also nur eine weitere Gliederungsebene. Generell ist es Komponenten innerhalb desselben Subsystems erlaubt, stärker an andere Komponenten gekoppelt zu sein, während Subsysteme untereinander eher über eine geringere Kopplung verfügen sollten. Um dies zu gewährleisten, können Subsysteme sogenannte Fassadenkomponenten anbieten.

Begründung: Anhand dieser Untergliederung können wir bereits eine grobe Architektursicht mit den wichtigsten Elementen und Abhängigkeiten definieren und auf dieser Basis die Library schrittweise weiter verfeinern.

Nachteile: Keine erkennbar

DES 007: **API- und Implementierungs-Layer**
Jede Komponente bietet ihre Dienste, Exceptions und Datentypen über einen API-Layer an. Dieser stellt die öffentliche Schnittstelle der Komponente dar. Andere Komponenten ebenso wie der `jMeta`-Anwender dürfen die Komponente nur über diese API-Klassen verwenden. Der Implementierungs-Layer der Komponente implementiert den API-Layer und ist privat. Insbesondere sind compile-Zeit-Abhängigkeiten zu dessen Klassen von anderen Komponenten oder Anwender-Klassen aus verboten.

Begründung: Es gibt eine Klare Trennung zwischen privaten und öffentlichen Anteilen, was die Kopplung und das Risiko von Fehlerpropagation sowie Inkompatibilitäten verringert, weil sich interne Änderungen an der Komponente idealerweise gar nicht auf nutzende Komponenten auswirken.

Nachteile: Keine erkennbar

***DES 008:* Keine Schichtenarchitektur in der Komponenten-Implementierung**
Die Implementierungs-Schicht einer Komponente in `jMeta` wird nicht weiter in Unter-Schichten gegliedert (wie dies in EE-Anwendungen häufig der Fall ist). Die innere Struktur einer Komponente wird durch keine Architekturvorgaben standardisiert, sondern zweckdienlich implementiert. Eine Schichtenarchitektur wäre beispielsweise: Eine Schicht kümmert sich um das Prüfen von Vorbedingungen, eine zweite implemeniert die Funktionalität, eine dritte kümmert sich um den Zugriff auf externe Daten.

Begründung: Eine Schichtenarchitektur in der Komponentenimplementierung ist für `jMeta` nicht notwendig und verkompliziert dessen Architektur. Es handelt sich um keine klassische "3-tier"-Anwendung, sondern eine Hilfslibrary, in welcher nur wenige Komponenten Datenzugriffe durchführen. Eine Schichtenarchitektur würde zu einer Verringerung der Übersichtlichkeit und mehr Redundanz führen, ohne nennenswerte Vorteile bei "separation of concerns" oder Entkopplung zu bringen.

Nachteile: Keine erkennbar

6.4. Entwicklungsumgebung

***DES 009:* Entwicklungsumgebung**
Als Entwicklungsumgebung wird eine Kombination aus Eclipse, Maven und Subversion genutzt.

Begründung: Bekannte und kostenloste Toolsuite.

Nachteile: Keine erkennbar

6.5. Multithreading

***DES 010:* `jMeta` ist nicht thread-safe**
`jMeta` ist keine thread-safe Library und verwendet keine Java-APIs, die thread-safe sind.

Begründung: Thread-Sicherheit bedeutet ggf. Performance-Verringerung durch Erzeugung von Synchronisationspunkten und Erhöhung der Gesamtkomplexität. Single-thread-Anwendungen werden benachteiligt. Es ist schwierig, thread-safety *korrekt* umzusetzen. Anwender können selbst dafür Sorge tragen, dass ihre multi-threaded-Anwendung thread-safe ist.

Nachteile: Keine erkennbar

6.6. Architektur

DES 011: **Architektur von `jMeta`**
`jMeta` basiert auf der technischen und fachlichen Architektur, wie sie in den Kapiteln "7 Technische Architektur" und "8 Fachliche Architektur" definiert wird.

Begründung: Siehe Diskussion der Architektur im Detail in den nächsten Abschnitten.

Nachteile: Siehe Diskussion der Architektur im Detail in den nächsten Abschnitten.

7. Technische Architektur

7.1. Technische Infrastruktur

Die technische Infrastruktur beschreibt die Umgebung, die für das Arbeiten mit `jMeta` benötigt wird, wie in der folgenden Abbildung gezeigt:

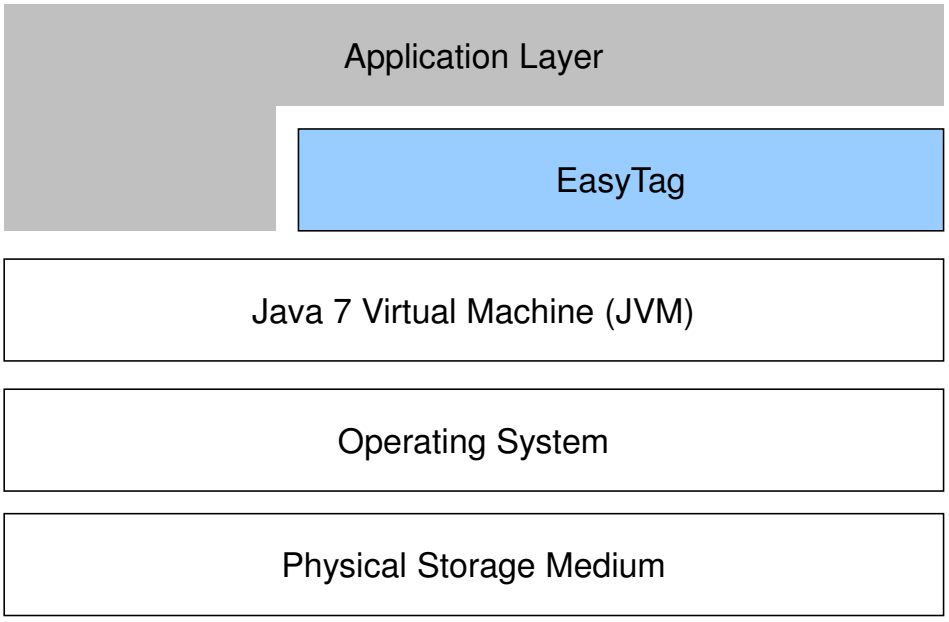

Figure 7.1.: Technische Infrastruktur von `jMeta`

Die technische Schichtenstruktur kann als Abhängigkeits- und Kommunikationsstruktur interpretiert werden. Der Applikations-Layer basiert auf `jMeta` während sowohl `jMeta` als auch der Applikations-Layer die Java 8 benutzen. Java 8 greift auf das Betriebssystem zu, welches Dienste zum Zugriff auf das physische Medium bietet.

Die Schichten sollten dabei nur auf benachbarte Schichten zugreifen.

7.1.1. Application Layer

Der Application Layer ist die Software, die `jMeta` zum Extrahieren und Schreiben von Metadaten nutzt. Es handelt sich um eine Java-Anwendung, mindestens für

Java 8 entwickelt worden ist. Sie nutzt natürlich darüber hinaus andere Java-Funktionalität und Libraries. Es kann sich um eine Java-SE-Desktop-Applikation oder auch eine Java-EE-Server-Applikation handeln.

7.1.2. `jMeta`

In der Abbildung bezeichnet `jMeta` alle Laufzeitkomponenten der Library `jMeta`. Diese Komponenten werden vom Application Layer aufgerufen und genutzt. `jMeta` selbst ist eine reine Java Library, die auf der Java 8 Java Virtual Machine (JVM) läuft. Sie kann somit nicht mit älteren Java-Versionen eingesetzt werden.

7.1.3. Java Virtual Machine (JVM)

`jMeta` benötigt eine Java Virtual Machine (JVM). `jMeta` wird in Java 8 entwickelt und ist daher nicht auf früheren Versionen nutzbar.

7.1.4. Betriebssystem

Die JVM läuft auf jedem Betriebssystem, das Java 8 unterstützt. Daher entkoppelt die JVM `jMeta` vom Betriebssystem. Es gibt jedoch einige Systemfunktionalitäten wie Prozess- und Threadmanagement ebenso wie Dateisystemzugriff, die teilweise vom Betriebssystem abhängen.

7.1.5. Physisches Speichermedium

`jMeta` greift auf Daten zu, die auf einem physischen Speichermedium gespeichert sind. Seine Lage oder Art ist unterschiedlich. In vielen Fällen handelt es sich um eine Datei auf einer Festplatte, es könnte sich aber auch um eine entfernt gespeicherte Ressource, Hauptspeicher oder eine Datenbanktabelle in einer entfernten Datenbank handeln.

Der Application Layer muss bei Verwendung von `jMeta` niemals selbst auf das physische Medium zugreifen. `jMeta`seinerseits nutzt die Java 8, diese das Betriebssystem, um auf die Daten des Mediums zuzugreifen.

7.2. Technische Basiskomponenten

Technische Basiskomponenten dienen nur als Hilfsmittel oder Rahmen der Umsetzung der fachlichen Inhalte von `jMeta`. Unter "fachliche" Inhalte wird das Lesen und Schreiben von Metadaten und das Lesen von Container-Daten verstanden. Die dafür notwendigen technischen Basiskomponenten werden hier kurz aufgeführt:

- Logging
- Service-Locator
- Utility

- Verwaltung von Erweiterungen

Details zu diesen Komponenten findet sich in “III `jMeta` Design”.

8. Fachliche Architektur

Die fachliche Architektur umfasst die Gliederung der Library-Funktionalität in fachliche Einheiten. Auf detaillierter Ebene sind dies die bereits definierten Komponenten. Auch wenn diese rein technische Funktionen umsetzen, beispielsweise Logging, werden sie in der fachlichen Architektur aufgeführt.

Hier noch einige detaillierte Designentscheidungen, die sich auf die fachliche Architektur beziehen.

8.1. Grundlegende Designentscheidungen zur fachlichen Architektur

Die folgenden grundlegenden Designentscheidungen haben einen maßgeblichen Einfluss auf die fachliche Architektur der Library, und sie haben einen übergreifenden Effekt, sind also nicht auf einzelne Subsysteme oder Komponenten beschränkt. Daher werden sie hier definiert. Sie liefern eine generelle Begründung des später entwickelten fachlichen Designs.

DES 012: **High-Level- und Low-Level-API**
Wir untergliedern `jMeta` in einen High-Level-Anteil, der bequeme User-Funktionalität zum Zugriff auf Metadaten bietet, und einen Low-Level-Anteil, der generische Expertenfunktionalität auf Bit- und Byte-Ebene bietet.

Begründung: Zunächst muss eine Low-Level-Zugriffsmöglichkeit gemäßt "4.4 ANF 004: Zugriff auf alle Rohdaten über die Library" zur Vergfügung gestellt werden. Statt Low-Level- und High-Level-Zugriff in einer unübersichtlichen API gemeinsam bereitzustellen, separieren wir sowohl API als auch die Implementierung dieser Belange. Aus Anwendersicht ist dann klar, welche API für ihn als "bequem" gedacht ist, und welche nur für detaillierten feingranularen Zugriff verwendet werden soll. Die low-level-API kann von der High-level-API aufgerufen werden, um diese zu implementieren. Dies schafft auch eine saubere Trennung in der Implementierung.

Nachteile: Ggf. höhere Komplexität der Gesamtlösung

***DES 013:* Generisches Parsen und Schreiben anhand einer Format-Spezifikation**
Das Parsen und Schreiben sämtlicher Metadaten- und Container-Formate wird anhand einer generellen Format-Spezifikation durch eine zentrale Komponente durchgeführt. Die Format-Spezifikation beschreibt, welche Features und Teile ein binäres Datenformat enthält, insbesondere, wie ein Datenblock dieses Formates aufgebaut ist und interpretiert werden muss. Es handelt sich also um eine Art generische Anleitung für das Parsen (und auch das Schreiben und Validieren) dieses Datenformates.
Weitere Designentscheidungen in späteren Abschnitten werden diese Designentscheidung vertiefen.

Begründung: Gemäß dem Dokument [MetaComp] haben zumindest binäre Container- und Metadatenformate viele Gemeinsamkeiten, die unter anderem die Definition eines generellen Domänenmodells ermöglichen. Diese Gemeinsamkeiten lassen sich auch durch generelle Formatspezifikationen beschreiben. Statt für jedes neu zu unterstützende Datenformat komplett neuen Parse-Code schreiben zu müssen, können viele Formate durch einheitlichen (nur einmal zu testenden) generischen Parse-Code unterstützt werden. Es ist eine Entkopplung von Format-Beschreibung und Lesen/Schreiben möglich. Die Formatbeschreibung kann als Textdokument abgelegt werden. Eine Erweiterung um ein neues Format ist daher im Idealfall einzig und allein durch Erzeugen einer solchen konformen Textdatei umsetzbar. Somit ermöglicht diese Designentscheidung die Umsetzung der Anforderung "4.7 ANF 007: Erweiterbarkeit um neue Metadaten- und Containerformate".

Nachteile: Es kann nicht für jedes denkbare zukünftige Format sichergestellt werden, dass die Möglichkeiten der Format-Spezifikation ausreichen, um alle Features des jeweiligen Formates wirklich abzudecken. Dies kann zur Notwendigkeit führen, die Format-Spezifikation zu erweitern und damit auch das generische Parsen. Alternativ kann dies durch Möglichkeiten ausgeglichen werden, das Parsen doch selbst umzusetzen (und eine entsprechende Implementierung statt der generischen zu verwenden). Weiterer Nachteil: Evtl. leichter Performance-Verlust, da das generische Parsen natürlich viele verschiedene Fälle unterstützen muss.

***DES 014:* Überschreiben des generischen Parsens und und Schreibens**
Erweiterungen können für ihre Datenformate den generischen Lese- und Schreibvorgang (siehe DES 014) überschreiben und erweitern, um sie an spezielle Gegebenheiten ihres Datenformates besser anzupassen.

Begründung: Dies minimiert die Nachteile von DES 014 und ermöglicht in Einzelfällen einfacherer oder performantere Implementierungen.

Nachteile: Keine erkennbar

DES 015: **Format-Spezifika werden nur in Erweiterungen definiert**
Jegliche Spezifika eines Metadaten- oder Containerformates werden ausschließlich über Erweiterungen implementiert, das gilt selbst für Datenformate, die direkt mit der Kernversion von `jMeta` unterstützt werden.

Begründung: So wird bereits mit der Kernlibrary selbst das Erweiterungskonzept genutzt und erprobt. Es findet eine strikte Trennung zwischen Kernimplementierung und Format-Spezifika statt, was eine bessere Beherrschung der Gesamt-Komplexität ermöglicht.

Nachteile: Keine erkennbar

DES 016: **Fassadenkomponenten für High-Level- und Low-Level-Anteile**
Die Subsysteme `Metadata API` und `Container API` verfügen über je eine Fassadenkomponente, die Zugriff auf die anderen *öffentlichen* Komponenten des jeweiligen Subsystems gewähren. "Öffentlich" sind diejenigen Komponenten, die vom Anwender oder von `Bootstrap` direkt zugegriffen werden müssen.

Begründung: Das Subsystem `Bootstrap` muss keine direkte Abhängkeit zu den Komponenten der Subsysteme der High-Level- und Low-Level-Anteile eingehen, sondern gibt nur eine Instanz der Fassadenkomponenten zurück, dies verringert die Kopplung. Spezielle Methoden zum Zugriff auf die anderen Komponenten des Subsystems können in den Fassadenkomponenten bereitgestellt werden und müssen nicht im Subsystem `Bootstrap` bereitgestellt werden (was auch nicht der Aufgabe von `Bootstrap` entsprechen würde).

Nachteile: Keine erkennbar

***DES 017:* Technische Basiskomponenten bilden ein eigenes Subsystem ohne Fassade**
Alle technischen Basiskomponenten bilden ein eigenes Subsystem und werden nicht zusammen mit fachlichen Komponenten in ein Subsystem aufgenommen. Es wird keine Fassadenkomponente zum Zugriff auf die Basiskomponenten bereitgestellt.

Begründung: Um die Kohärenz der Subsysteme zu erhalten, werden die technischen Komponenten in ein eigenes Subsystem ausgelagert. Die technischen Basiskomponenten können als sogenannte "0-Software", d.h. perfekt wiederverwendbare Software betrachtet werden. Sie haben keine inhaltlich-fachliche Funktionen und sollten (in den meisten Fällen) keine weiteren Abhängigkeiten zu anderen Komponenten haben. Da sie alle recht spezifische und umfangreiche Funktionalität anbieten, macht ein Verwenden einer Fassadenkomponenten keinerlei Sinn. Diese würde einerseits viele unterschiedliche Belange, die nicht verwandt sind, in ein Interface zwängen, und andererseits keineswegs zu geringerer Kopplung führen.

Nachteile: Keine erkennbar

8.2. Subsysteme

Die Unterteilung in Subsysteme zeigt bereits grob die wichtigsten Teile der Library und erste Abhängigkeiten zwischen ihnen. Über Subsysteme verorten wir auch den Begriff der *Erweiterung*. Zudem bildet sich hier direkt die Designentscheidung DES 017 ab.

Das folgende Architekturbild zeigt die Subsysteme von `jMeta` und ihre Beziehungen zueinander. Ein Pfeil bedeutet dabei eine hier noch nicht näher konkretisierte Abhängigkeit, die sich entweder als Compile-Zeit- oder als Laufzeit-Abhängigkeit oder beides manifestieren kann.

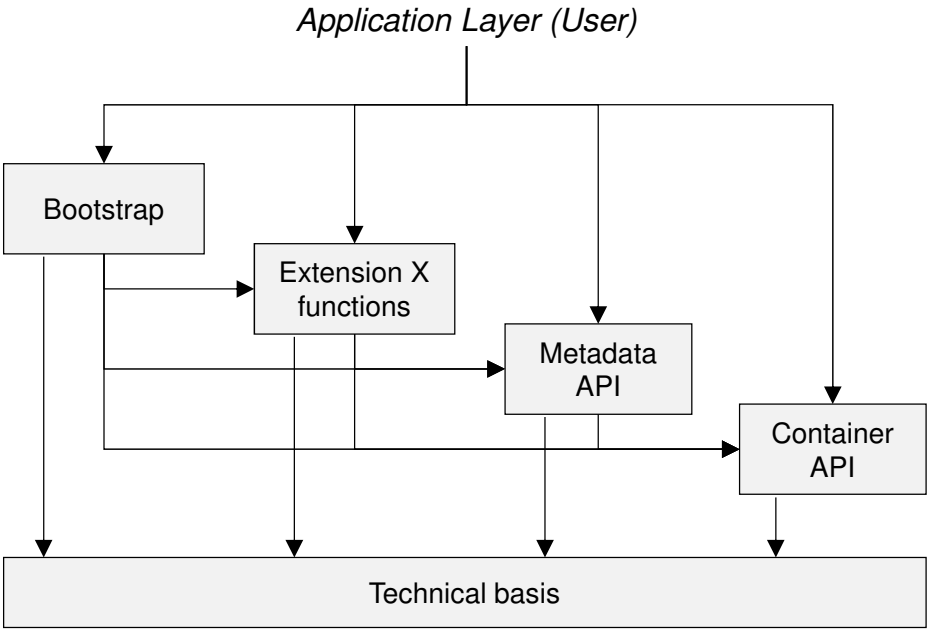

Figure 8.1.: Subsysteme von `jMeta`

Der Anwender der Library kann direkt auf die Subsysteme `Bootstrap`, `Extension`, `Metadata API` und `Container API` zugreifen, während die `Technical Base` nicht zugreifbar ist.

Die Subsysteme werden im Folgenden weiter konkretisiert.

8.2.1. `Bootstrap`

Dieses Subsystem kapselt alle Initialisierungen von `jMeta`. Das Subsystem ist der Eintrittspunkt der Benutzung von `jMeta` für den Anwender. Es nutzt daher alle anderen Subsysteme, um diese zu initialisieren.

Die Komponenten des Subsystems sind im Abschnitt "8.4 Komponenten des Subsystems `Bootstrap`" aufgeführt.

8.2.2. `Metadata API`

Die High-Level-Anteile der Library gemäß DES 017. Das Subsystem greift auf `Technical Base` und `Container API` zu. Letzteres deshalb, weil gemäß DES 017 die Implementierung der High-Level-Anteile durch Verwendung der low-level-Anteile erfolgt.

Die Komponenten des Subsystems sind im Abschnitt "8.5 Komponenten des Subsystems `Metadata API`" aufgeführt.

8.2.3. Container API

Die Low-Level-Anteile der Library gemäß DES 017. Greift nur auf die technische Basis zu.

Die Komponenten des Subsystems sind im Abschnitt "8.6 Komponenten des Subsystems Container API" aufgeführt.

8.2.4. Technical Base

Eine Sammlung von Komponenten, die als technische Rahmenkomponenten betrachtet werden können und keine fachlich-inhaltlichen Beiträge zum Thema "Metadaten/Container" liefern. Sie werden von so gut wie allen anderen Subsystemen benötigt.

Die Komponenten des Subsystems sind im Abschnitt "8.7 Komponenten des Subsystems Technical Base" aufgeführt.

8.2.5. Extension

jMeta erlaubt eine beliebige Anzahl an Erweiterungen, jede davon entspricht in der fachlichen Architektur einem Subsystem.

Details finden sich im Abschnitt "8.8 Erweiterungen (Subsystem Extension)".

8.3. Komponenten-Steckbrief

Da in den folgenden Abschnitten Komponenten einführend beschrieben werden, wird hier ein Steckbrief, d.h. eine grundlegende Beschreibungsstruktur für eine Komponentengrobbeschreibung definiert. Dieser Steckbrief wird dann in den folgenden Abschnitten für jede Komponente ausgefüllt.

Komponenten-Name: Der Name der Komponente.

Aufgabe: Die Aufgabe der Komponente.

Kontrollierte Daten: Die Daten, die durch die Komponente kontrolliert werden, d.h. gelesen und geschrieben werden. Nutzt die Komponente Daten anderer Komponenten, wird dies hier nicht erwähnt.

Abhängig von <Komponenten-Name>: Dieses Element kommt mehrfach je Komponente vor, von der diese Komponente abhängt. Der Grund für die Abhängigkeit wird kurz erläutert.

8.4. Komponenten des Subsystems Bootstrap

Die folgende Abbildung zeigt die Komponenten des Subsystems Bootstrap.

Open Issue 8.1: **– Bootstrap subsystem Componenten Abbildung**

8.4.1. Komponente `EasyTag`

Komponenten-Name: `EasyTag`.

Aufgabe: Der Einstiegspunkt für jeden User von `jMeta`. Es ermöglicht Zugriff auf alle anderen Komponenten der Library.

Kontrollierte Daten: Keine.

Abhängig von `Container API`: `EasyTag` gibt Zugriff auf die Komponente `Container API`.

Abhängig von `Metadata API`: `EasyTag` gibt Zugriff auf die Komponente `Metadata-API`.

Abhängig von `ExtensionManagement`: `EasyTag` lädt alle Erweiterungen unter Nutzung dieser Komponente.

Abhängig von `Logging`: `EasyTag` nutzt diese Komponente zur Protokollierung des Startup-Prozesses.

Abhängig von `SimpleComponentRegistry`: `EasyTag` nutzt diese Komponente zum Instantiieren bzw. Abfragen von Implementierungen anderer verwendeter Komponenten-Interfaces.

Abhängig von `Utility`: `EasyTag` nutzt diverse querschnittliche Funktionen von `Utility`.

8.5. Komponenten des Subsystems `Metadata API`

Die Komponenten des Subsystems `Metadata API` werden in dieser Version der Library noch nicht definiert.

8.6. Komponenten des Subsystems `Container API`

Die folgende Abbildung zeigt die Komponenten des Subsystems `Container API`.

Open Issue 8.2: **– Container API subsystem Abbildung**

8.6.1. `Container API`

Komponenten-Name: `Container API`.

Aufgabe: Fassadenkomponente. Gewährt allen Anwendern Zugriff auf die anderen öffentlichen Komponenten dieses Subsystems.

Kontrollierte Daten: Keine.

Abhängig von `DataBlocks`: Gewährt Zugriff auf diese Komponente.

Abhängig von `DataFormats`: Gewährt Zugriff auf diese Komponente.

8.6.2. `DataBlocks`

Komponenten-Name: `DataBlocks`.

Aufgabe: Gewährt lesenden Zugriff auf Metadaten und Containerdaten und schreibenden Zugriff auf Metadaten auf Bit- und Byte-Ebene, allerdings werden die Daten in handlichen Portionen gemäß Datenformat-Spezifikation geliefert.

Kontrollierte Daten: Ein Zugriff auf CONTAINER- und TAG-Daten ist nur über diese Komponente erlaubt. Somit hat sie die Kontrolle über diese Daten. Lediglich `Media` darf auf noch generischerer Ebene auf Daten externer Medien zugreifen. Tatsächlich nutzt `DataBlocks` `Media` für das Lesen und Schreiben.

Abhängig von `Media`: `DataBlocks` muss Datenpakete von Medien lesen oder auf diese Schreiben. Dazu nutzt es `Media`.

Abhängig von `DataFormats`: Das Lesen und Schreiben der Daten erfolgt anhand von Format-Spezifikationen, die durch `DataFormats` geliefert werden.

8.6.3. `DataFormats`

Komponenten-Name: `DataFormats`.

Aufgabe: Verwaltet die konkreten Datenformat-Definitionen aller unterstützten Metadaten- und Container-Formate.

Kontrollierte Daten: Hat Kontrolle über die Datenformat-Definitions-Daten.

Abhängig von: Keinen anderen Komponenten.

8.6.4. `Media`

Komponenten-Name: `Media`.

Aufgabe: Bietet Primitive für den Zugriff auf physische Medien an.

Kontrollierte Daten: Daten von externen Medien dürfen nur über diese Komponente zugegriffen und manipuliert werden.

Abhängig von: Keinen anderen Komponenten.

8.7. Komponenten des Subsystems `Technical Base`

Die folgende Abbildung zeigt die Komponenten des Subsystems `Technical Base`.

Open Issue 8.3: **– Tech Base subsystem Abbildung**

8.7.1. `ExtensionManagement`

Komponenten-Name: `ExtensionManagement`.

Aufgabe: Verwaltet alle Erweiterungen von `jMeta`, d.h. laden, verifizieren und Auslesen von Informationen zu jeder Erweiterung.

Kontrollierte Daten: Beschreibungsdaten der Erweiterungen können nur über diese Komponente geladen werden.

8.7.2. `Logging`

Komponenten-Name: `Logging`.

Aufgabe: Bietet Primitive zum Ausgeben von Logging-Informationen.

Kontrollierte Daten: Logging-Konfiguration.

Abhängig von: Keiner anderen Komponente.

8.7.3. `Utility`

Komponenten-Name: `Utility`.

Aufgabe: Bietet diverse (technische) Querschnittsfunktionen, die von allen anderen Komponenten regelmäßig benötigt werden, z.B. Hilfsfunktionen zur Umsetzung von design-by-contract.

Kontrollierte Daten: Keine.

Abhängig von: Keiner anderen Komponente.

8.7.4. `SimpleComponentRegistry`

Komponenten-Name: `SimpleComponentRegistry`.

Aufgabe: Technische Komponente mit Service-Locator-Funktionalität zum Abfragen der Implementierungen von Interfaces anderer Komponenten.

Kontrollierte Daten: Konfiguration von Komponenten, Interfaces und ihren Implementierungen.

Abhängig von: Keiner anderen Komponente.

8.8. Erweiterungen (Subsystem `Extension`)

Eine *Erweiterung* fasst formatspezifische Inhalte zu einem oder mehreren Datenformaten zusammen. Je Datenformat, dass die Erweiterung definiert, sind dies (vergleiche DES 017, DES 017 und DES 017):

- Eine Datenformat-Spezifikation, welche die Datenblöcke des Datenformats und deren Aufbau und Zusammenhang untereinander definiert
- Eine API, welche Komfortfunktionen und Konstanten zum Arbeiten mit dem Datenformat (z.B. Erzeugen von Datenblöcken usw.) bietet
- Die API enthält insbesondere die Datenformat-Kennung, die der Anwender von `jMeta` auch beim Arbeiten mit `Metadata API` und `Container-APIverwenden` kann.
- Implementierungs-Erweiterungen für `Container API`, welche den Parse- und Schreibevorgang beeinflussen. Mit diesem Mechanismus kann ein Datenformat den in `Container API` definierten Standard-Parse- und Schreibe-Algorithmus erweitern oder überschreiben.

Die Frage ist nun: Wie korrespondiert der Begriff der Erweiterung mit dem eines Subsystems und einer Komponente? Eine einzelne konkrete Erweiterung wurde oben bereits als Subsystem eingeführt. Sie besteht also aus mehreren Komponenten. Genauer sollte je enthaltenem Datenformat je eine Komponente im Sinne von DES 017 enthalten sein. Jede Komponente enthält die oben definierten API- und Implementierungsanteile. Für die interne Gestaltung der Erweiterung ist freilich der Implementierer der Erweiterung verantwortlich. Jedoch lassen sich Richtlinien für die innere Strukturierung einer Erweiterung definieren, die sich im Wesentlichen an den Richtlinien der `jMeta`-Kernimplementierung orientieren.

Unterschiedliche Erweiterungen haben i.d.R. nichts miteinander zu tun und sollten sich wie für Komponenten üblich maximal über ihre Schnittstellen gegenseitig aufrufen.

Part III.

jMeta Design

Dieser Teil definiert das Design der Komponenten je Subsystem, basierend auf den vorangegangenen Kapiteln.

9. Übergreigende Aspekte

Als Design wird hier die Fortsetzung der skizzierten Architektur im Detail verstanden.

In diesem Abschnitt werden übergreifende Aspekte des Designs von `jMeta` behandelt, die keinen Bezug zu nur einer Komponente oder nur einem Subsystem haben. Es handelt sich meist um die bekannten *cross-cutting concerns*.

9.1. Generelle Fehlerbehandlung

Hier wird der generelle komponenten-übergreifende Ansatz der Fehlerbehandlung in `jMeta` behandelt. Es werden also keine konkreten Fehler bestimmter Komponenten behandelt.

9.1.1. Abnormale Ereignisse vs. Fehler einer Operation

Gemäß [Sied06] können wir Fehler wie folgt kategorisieren - die Kategorien werden hier zusätzlich untergliederd und benannt:

- *Kategorie 1: Abnormale Ereignisse*: Ereignisse, die nur selten auftreten sollten und spezielle Behandlung erfordern.
  - Verbindung zu einem Server ist abgebrochen
  - Eine Dateioperation schlägt fehl
  - Eine Konfigurations-Datei oder Tabelle, deren Existenz vorausgesetzt wird, ist nicht vorhand
  - Ein externer Speicher- oder der Hauptspeicherplatz ist erschöpft
- *Kategorie 2: Fehler einer Operation*: Eine Operation kann mit einem Fehler oder mit einem Erfolg beendet werden. Fehler einer Operation kann man von abnormalen Ereignissen dadurch unterscheiden, dass sie eine höhere Wahrscheinlichkeit haben, aufzutreten, und dass sie in der Regel direkt vom Aufrufer der Operation behandelt werden können.
  - Kategorie 2a: Die Operation kann aus bestimmten inhaltlichen Gründen nicht korrekt durchgeführt werden, und muss daher abgebrochen werden. Z.B. hat ein Konto nicht die notwendige Deckung für die Durchführung einer Überweisung.
  - Kategorie 2b: Ungültiger User-Input, z.B. ist ein Eingabewert außerhalb des zulässigen Bereiches oder ein Objekt, auf dass sich die Eingaben beziehen, existiert nicht (mehr).

  - Kategorie 2c: Das aufgerufene Objekt hat nicht den notwendigen Zustand, der zum Aufruf der Operation gegeben sein muss.

9.1.2. Fehlerbehandlungs-Ansätze

Fehlerbehandlungsmechanismen, die manchmal auch in Kombination eingesetzt werden:

- *Error codes*: In prozeduralen System-Programmierungs-APIs, wie bei der Linux- oder Windows-API begegnen einem häufig noch error codes, d.h. Operationen liefern üblicherweise error codes als Rückgabewert. Einer der Codes ist häufig mit der Semantik "kein Fehler aufgetreten" belegt. Andere definieren spezielle Fehlersemantiken, die bei Ausführung der Operation aufgetreten sind.

- *Error-Handler*: Manche APIs ermöglichen es, Fehler-Behandlungsroutinen, sogenannte error handler anzugeben, die in Form von Call-Backs von der aufgerufenen Operation gerufen werden, wenn Fehler aufgetreten sind. Ein solcher error handler kann den Fehler dann behandeln.

- *Exceptions*: In objektorientierten Programmen sind Exceptions das Mittel der Wahl für die Fehlerbehandlung. Sie werden von einer Operation geworfen, was die Reihenfolge der Code-Ausführung ändert. Sie können in der call hierarchy gefangen werden. Geschieht dies nicht, beenden sie üblicherweise den Prozess, in dem die Operation ausgeführt werden ist. Fangen entspricht meist der Behandlung des Fehlers. Einge objekt-orientierte Sprachen wie Java und C++ unterscheiden zwischen checked und unchecked Exceptions.

***DES 018:* Fehlersignalisierung durch Exceptions**
`jMeta` nutzt ausschließlich Exceptions als Mechanismus zur Fehlersignalisierung.

Begründung: Exceptions sind in Java gut unterstützt und wohlbekannt. Die anderen oben genannten Mechanismen sind in Java-APIs so gut wie nicht zu finden. Entsprechend ist die Verwendung des De-factor-Standardmechanismus auch für `jMeta` sinnvoll und gut geeignet.

Nachteile: Keine erkennbar

9.1.3. Allgemeine Designentscheidungen zur Fehlerbehandlung

Zunächst eine Designentscheidung mit sehr allgemeinen Richtlinien zu Exception-Klassen:

***DES 019:* Richtlinien für die allgemeine Fehlerbehandlung in `jMeta`**
Es gelten folgende Richtlinien in `jMeta`:

- Für jede Fehlerkategorie wird eine separate Exception-Klasse definiert, diese hat einen sinnvollen Namen, der die Fehlerkategorie treffend beschreibt. Dieser Name endet mit "Exception". Die Klasse speichert notwendige Kontextinformationen zur Fehlerursache, die über getter im Rahmen der Fehlerbehandlung abgefragt werden kann.
- `jMeta` wirft keine Exceptions der Java-Standard-Library. Stattdessen werden solche Fehler ggf. in eigene `jMeta` Exceptions als cause gewrappt.
- Generell muss eine `jMeta`-Exception eine verursachende Exception als cause setzen.
- `jMeta`-Exceptions können einen erläuternden Text zur Ursache des Fehlers enthalten. Dieser muss in U.S. Englisch formuliert werden.

Begründung: Fehleranalyse wird somit nicht unnötig erschwert, Exceptions haben eine erkennbare Bedeutung und werden nicht zu generisch.

Nachteile: Keine erkennbar

Die folgende Design-Entscheidung schließt eine Fehlerfassade aus:

***DES 020:* Keine Fehlerfassade in `jMeta`**
`jMeta` wird nicht durch eine Fehlerfassade umgeben, die alle unchecked Exceptions abfängt, bevor sie zum Anwender der Library gelangen können.

Begründung: Eine solche Fehlerfassade bedeutet einen zusätzlichen Overhead. Die breite Schnittstelle der Library müsste so an allen "Ausgängen" mit der Fehlerfassade umgeben werden, was die Implementierung unnötig verkompliziert. `jMeta` kann ohnehin nicht alle unchecked Exceptions, die auftreten können, sinnvoll behandeln. Eine Weitergabe an den Anwender ist damit sinnvoll.

Nachteile: Keine erkennbar

Die folgenden Designentscheidungen geben ganz grundlegende an, welche Exception-Arten für welche Fehlerkategorien eingesetzt werden:

DES 021: **Unchecked exceptions für abnormale Ereignisse (Kategorie 1)**
Im Falle von abnormalen Ereignissen wird in `jMeta` entweder eine spezielle `jMeta`-Exception als unchecked Exception (d.h. Exception, die von java.lang.RuntimeException ableitet) geworfen, oder es wird eine durch eine Java-Standard-Library-Methode erzeugte Exception geworfen.
Es wird je Komponente entschieden, welche Fehler als abnormal gelten.

Begründung: Abnormale Ereignisse können vom Aufrufer meist nicht sinnvoll behandelt werden. Durch checked exception würde jedoch zumindest ein "catch" erzwungen. Es macht darüber hinaus außer in Einzelfällen häufig wenig Sinn, runtime exceptions der Java-Standard-Library abzufangen und in `jMeta`-Exceptions zu konvertieren. Dies bringt nicht nur overhead mit sich, sondern gefährdet auch die Portabilität, da unter Umständen unspezifizierte Exceptions gefangen werden.

Nachteile: Keine erkennbar

DES 022: **Checked exceptions für inhaltliche Fehler der Operation (Kategorie 2a)**
Im Falle von inhaltlichen Fehlern einer Operation wird in `jMeta` eine spezielle `jMeta`-Exception als checked Exception (d.h. Exception, die von java.lang.Exception ableitet) geworfen.
Es wird je Komponente und Operation entschieden, welche Fehler als inhaltliche Fehler der Operation gelten.

Begründung: Inhaltliche Fehler einer Operation können erwartet werden. Sie treten häufiger auf als abnormale Ereignisse. Aufrufer wissen i.d.R., wie sie diese behandeln müssen.

Nachteile: Keine erkennbar

DES 023: **Design-by-Contract für fehlerhafte Verwendung einer öffentlichen Operation (Kategorien 2b und 2c)**
Erfolgen Aufrufe auf öffentliche API-Operationen einer Komponente im falschen Objektzustand (d.h. eine Vorbedingung ist nicht erfüllt) oder werden dort Parameterwerte angegeben, die nicht dem gültigen Wertebereich entsprechen, dann verfährt `jMeta` gemäß design-by-contract rigoros, indem eine spezielle `jMeta` Unchecked Exception geworfen wird und die Verarbeitung der Operation somit ohne Effekt beendet wird. Dies signalisiert, dass es sich um einen fehlerhaften Aufruf der Operation handelt. Dieses Verhalten wird in der Schnittstellenbeschreiung der Methode definiert.

Begründung: Der Vertrag ist klar definiert, dem Aufrufe ist klar, was er erfüllen muss, um die Methode verwenden zu dürfen. Falscher Aufruf wird als Programmierfehler gewertet und entsprechend quittiert. Die `jMeta`-Schnittstelle verhindert so, dass fehlerhafte Eingabe zu inkonsistenten Zuständen oder Daten oder zum Propagieren von Fehlern in untere Schichten führen und dann erst später zu, Vorschein kommen, was die Analyse solcher Fehler sehr erschweren kann. Hier wird gemäß "fail fast" gehandelt und der Fehler sofort bei der ersten Möglichkeit erkannt.

Nachteile: Keine erkennbar

9.2. Logging in `jMeta`

Logging wird in `jMeta` ebenso verwendet, wie folgende Designentscheidung verrät:

DES 024: **Verwendung von Logging in `jMeta`**
Logging wird in `jMeta` zumindest in den Subsysteme `Bootstrap` und `Technical Base` verwendet, um Startup der Library zu protokollieren. In anderen Subsystemen wird logging nur in Ausnahmefällen, ggf. bei Fehlerbehandlung eingesetzt. Das Logging kann auf Klassengranularität im Feinheitsgrad vom Anwender konfiguriert oder auch (komplett Klassenübergreifend) deaktiviert werden.

Begründung: In hinreichend komplexen Systemen kann Logging zur Fehleranalyse nicht ersetzt werden. Logging ist zumindest für komplexe, fehleranfällige Abläufe unerlässlich. Deaktivierbarkeit verringert die Gefahr von Performance-Problemen.

Nachteile: Keine erkennbar

9.3. Konfiguration

Unter dem Begriff *Konfiguration* wird bei `jMeta` das Anpassen von bestimmten Parametern bezeichnet, die zur Laufzeit Einfluss auf die Funktionalität von `jMeta`

haben. Dies ist sehr schwammig und die Abgrenzung zwischen settern und Konfiguration fällt mit dieser Definition erstmal schwer.

Die wesentlichen Merkmale von Konfiguration sind jedoch:

- Konfiguration ist Bestandteil der öffentlichen Schnittstelle von `jMeta`, kann also vom Anwender beliebig angepasst werden.

- Sie ist generisch und damit für weitere Releases erweiterbar, d.h. neue Konfigurationsparameter hinzufügen bedeutet in erster Linie tatsächlich für die API nur, dass eine neue Konstante hinzukommt; die API für das Setzen und Abfragen bleibt unverändert.

- Konfigurationsparameter werden i.d.R. einmalig beim ersten Laden der Library aktiv, sollen aber bei `jMeta` auch dynamisch geändert werden können, teilweise mit sofortiger Wirksamkeit

Wir halten folgende Designentscheidungen fest:

DES 025: **`jMeta` bietet einen generischen und erweiterbaren Konfigurationsmechanismus über die öffentliche API an**
Die Library über ihre öffentliche API Mittel zum Setzen und Abfragen von Konfigurationsparametern an. Die verfügbaren Konfigurationsparameter werden über die API als Konstanten repräsentiert und in der Dokumentation aufgeführt. In `jMeta` kann prinzipiell jede Klasse einzeln konfigurierbar sein, d.h. der Sichtbarkeitsbereich der Konfigurationen muss in keinster Weise global sein.

Begründung: Dies gewährt Flexibilität in vielerlei Hinsicht:

- Zukünftige Versionen von `jMeta` können einfach um weitere Konfigurationsmöglichkeiten erweitert werden, ohne die API anpassen zu müssen (bis auf die neue Konfigurationskonstante)

- Konfigurationen mit globaler Gültigkeit machen meist wenig Sinn, sondern sind eher hinderlich, häufig erzeugt die Library einzelne Objekte (z.B. pro Sitzung oder pro Medium), die unabhängig voneinander konfiguriert werden müssen.

Nachteile: Keine bekannten Nachteile

Darüber hinaus gilt:

DES 026: **Konfiguration zur Laufzeit, keine properties-Dateien**
`jMeta` kann über Laufzeitaufrufe konfiguriert werden, und nicht über properties-Dateien

Begründung: Properties-Dateien mögen in einigen Anwendungsfällen sinnvoll sein, aber nicht für eine dynamisch anpassbare Konfiguration. Sie sind eben nur statisch, und Änderungen erfordern i.d.R. den Neustart der Applikation. Natürlich können properties in Zukunft als Initial-Konfiguration dienen, die dann dynamisch nachjustiert werden kann. In der aktuellen `jMeta`-Version wird allerdings keine Notwendigkeit für statische Konfiguration gesehen.

Nachteile: Keine bekannten Nachteile

Für die Konfigurationsparameter selbst gilt:

DES 027: **Jeder Konfigurationsparamater hat einen Gültigkeitsbereich und einen Standardwert**
Jeder Konfigurationsparameter bei `jMeta` ist in dem Sinne verpflichtend, dass er immer einen Wert (in seinem Scope) hat. Dazu wird für jeden Parameter ein sinnvoller Standardwert definiert, der gilt, wenn kein explizites Setzen durchgeführt worden ist.
Zudem hat jeder Konfigurationsparameter einen gültigen Wertebereich, z.B. Minimum und Maximum bzw. gültige Werte.

Begründung: Standardwerte sind immer wichtig, denn keiner will den Anwender zwingen, explizit zu konfigurieren. Die Standardwerte sollten so gewählt sein, dass das Verhalten der Library dem 80%-Fall genügt und stabil funktioniert.
Wertebereiche kommunizieren dem Anwender bereits klar, welche Werte gültig sind und können für Plausibilitätsprüfungen genutzt werden.

Nachteile: Keine bekannten Nachteile

Schließlich legen wir noch fest:

***DES 028:* Für Konfigurationen wird die Utility-API "10.1.1 Configuration API" genutzt.**
`jMeta` nutzt eine API der Komponente `Utility`, die auch prinzipiell von beliebigen anderen Projekten verwendet werden kann, um Konfigurationen anzubieten.

Begründung: Konfiguration kann über eine generische API erfolgen. Die Notwendigkeit von Konfiguration ist nichts `jMeta`-spezifisches, sondern generell von Bedeutung. Daher macht das Aufbauen von wiederverwendbaren Komponenten in einem Utility-Anteil Sinn. Dies fördert auch die Einheitlichkeit: Die API für Konfiguration sieht für alle Komponenten von `jMeta` gleich aus.

Nachteile:

10. Technical Base Design

10.1. Utility Design

In diesem Abschnitt wird das Design der Komponente `Utility` beschrieben. Grundaufgabe der Komponente ist das Anbieten genereller Querschnittsfunktionalität, die unabhängig von der Fachlichkeit ist, und so potentiell in mehreren Projekten Verwendung finden kann. Hier werden allerdings nur diejenigen Aspekte beschrieben, die für `jMeta` relevant sind.

10.1.1. Configuration API

Die Configuration API bietet allgemeine Funktionen für Laufzeit-Konfiguration von Software-Komponenten an. Wir entwickeln hier das Design der API.

***DES 029:* Ein Konfigurationsparameter wird durch eine generische Klasse `AbstractConfigParam` repräsentiert**
Die Klasse `AbstractConfigParam` repräsentiert einen konkreten Konfigurationsparameter, nicht jedoch dessen Wert an sich. Der Typparameter T gibt die Klasse des Wertes der Property an, dabei gilt: `T extends Comparable`. Instanzen der Klasse `AbstractConfigParam` werden als Konstanten in konfigurierbaren Klassen definiert. Die Klasse hat folgende Eigenschaften und Funktionen:

- `getName()`: Der Name des Konfigurationsparameters
- `getDefaultValue()`: Der Default-Wert des Konfigurationsparameters
- `getMaximumValue()`: Den Maximal-Wert des Konfigurationsparameters oder null, falls er keinen hat
- `getMinimumValue()`: Den Minimal-Wert des Konfigurationsparameters oder null, falls er keinen hat
- `getPossibleValues()`: Eine Auflistung der möglichen Werte, oder null falls er keine Auflistung fester Werte hat
- `stringToValue()`: Konvertiert eine String-Repräsentation eines Konfigurationsparameterwertes in den eigentlichen Datentyp des Wertes
- `valueToString()`: Konvertiert den Wert eines Konfigurationsparameterwertes in eine Stringrepräsentation

Begründung: Eine solche Repräsentation garantiert Typ-Sicherheit und eine bequeme Verwendung der API. Das Einschränken auf `Comparable` ist keine wirkliche Einschränkung, da so gut wie alle Wert-Klassen aus Java-SE, u.a. die numerischen Typen, Strings, Boolean, Character, Charset, Date, Calendar, diverse Buffer-Implementierungen usw. `Comparable` implementieren.

Nachteile: Keine bekannten Nachteile

Jede konfigurierbare Klasse soll möglichst eine einheitliche Schnittstelle bereitstellen, um Konfigurationsparameter zu setzen und abzufragen:

***DES 030:* Schnittstelle für das Handhaben von Konfigurationsparametern**
Jede konfigurierbare Klasse muss die Schnittstelle `IConfigurable` implementieren, mit folgenden Methoden:

- `setConfigParam()`: Setzt den Wert eines Konfigurationsparameters
- `getConfigParam()`: Liefert den aktuellen Wert eines Konfigurationsparameters
- `getAllConfigParams()`: Liefert alle Konfigurationsparameter mit ihren aktuellen Werten
- `getSupportedConfigParams()`: Liefert ein Set aller von dieser Klasse unterstützten Konfigurationsparameter
- `getAllConfigParamsAsProperties()`: Liefert alle Konfigurationsparameter als eine `Properties`-Instanz.
- `configureFromProperties()`: Setzt die Werte aller Konfigurationsparameter basierend auf einer `Properties`-Instanz
- `resetConfigToDefault()`: Setzt die Werte aller Konfigurationsparameter auf ihre Default-Werte zurück

Dabei müssen alle unterstützten Konfigurationsparameter der Klasse unterschiedliche Namen haben.

Begründung: Damit wird eine einheitliche Schnittstelle für jede konfigurierbare Klasse ermöglicht. Die unterschiedlichen Namen sind zur eindeutigen Identifikation notwendig.

Nachteile: Keine bekannten Nachteile

Damit nun nicht jede Klasse selbst die Handhabung und Verifikation der Konfiguration implementieren muss, definieren wir:

***DES 031:* `ConfigHandler` implementiert `IConfigurable` und kann von jeder konfigurierbaren Klasse verwendet werden**
Die nicht-abstrakte Klasse `ConfigHandler` implementiert `IConfigurable` und übernimmt die gesamte Aufgabe der Konfiguration. Sie kann von konfigurierbaren Klassen entweder als Basisklasse oder aber als aggregierte Instanz verwendet werden, an die alle Aufrufe weitergeleitet werden.

Begründung: Keine Klasse muss die Konfigurationsverwaltung selbst implementieren

Nachteile: Keine bekannten Nachteile

Der Umgang mit fehlerhaften Konfiguration ist Teil der folgenden Designentscheidung:

DES 032: **Fehlerhafte Konfigurationsparameterwerte führen zu einem Laufzeitfehler**
Wird ein fehlerhafter Wert für einen Konfigurationsparameter übergeben, reagiert die API mit einem Laufzeitfehler, einer `InvalidConfigParamException`. Hierfür wird eine öffentliche Methode `checkValue()` in `AbstractConfigParam` bereitgestellt.

Begründung: Die Wertebereiche der Parameter sind wohldefiniert und beschrieben, es handelt sich um einen Programmierfehler, wenn ein falscher Wert übergeben wird.

Nachteile: Keine bekannten Nachteile

Änderungen von Konfigurationsparametern müssen u.U. sofort wirksam werden, daher definieren wir:

DES 033: **Observer-Mechanismus für Konfigurationsänderungen**
Es wird ein Observer-Mechanismus über `IConfigChangeListener` bereitgestellt, sodass Klassen über dynamische Konfigurationsänderungen informiert werden. Dieses Interface hat lediglich eine Methode `configurationParameterValueChanged()`.
`IConfigurable` erhält damit zwei weitere Methoden: `registerConfigurationChangeListener()` und `unregisterConfigurationChangeListener()`.

Begründung: Die konfigurierbaren Klassen müssen nicht immer diejenigen Klassen sein, welche mit den Konfigurationsänderungen umgehen müssen und die Konfigurationsparameter direkt verwenden. Stattdessen kann es sich um ein kompliziertes Klassengeflecht handeln, das zur Laufzeit keine direkte Beziehung hat. Daher ist ein entkoppelnder Listener-Mechanismus nötig.

Nachteile: Keine bekannten Nachteile

10.2. `SimpleComponentRegistry` Design

In diesem Abschnitt wird das Design der Komponente `SimpleComponentRegistry` beschrieben. Grundaufgabe der Komponente ist die Implementierung der Designentscheidungen DES 033 und DES 033.

Wir geben hier lediglich einen kurzen Abriss des Basisdesigns in Form mehrere aufeinanderfolgender Designentscheidungen.

***DES 034:* Eine Komponente bietet genau ein Komponenten-Java-Interface**
Jede Komponente im Sinne von `SimpleComponentRegistry` bietet genau ein Java-Interface, das `IComponentInterface` implementiert.

Begründung: Damit ist der Implementierung klar, wie eine Komponente seine Funktionalität nach Außen anbieten muss, und es ist dem Anwender klar, wie die Funktionalität der Komponente insgesamt aussieht. Zudem werden Komponenteninterfaces klar dadurch markiert, dass sie `IComponentInterface` implementieren. Dies bringt nicht etwa den Nachteil mit sich, dass die Schnittstelle nicht frei nach OO-Aspekten designt werden könnte. Denn jede Methode des Komponenten-Interfaces kann wiederum beliebige Datentypen und weitere Interfaces liefern, so dass dem Design dadurch keine Grenzen gesetzt sind.

Nachteile: Keine bekannten Nachteile

Gemäß DES 034 ist eine Komponente ein "Singleton". Dennoch muss zusätzlich klar definiert werden, wie der Lebenszyklus einer Komponente aussieht:

***DES 035:* Der Komponentenlebenszyklus besteht aus Initialisierung und Nutzung**
Eine Komponente (und damit die Einzel-Implementierung des Komponenteninterfaces gemäß DES 035) hat folgende Lebensabschnitte und -ereignisse:

- *Initialisierung:* Die Implementierung registriert sich als Implementierung eines Komponenteninterfaces bei `SimpleComponentRegistry`, holt sich Instanzen anderer Komponenten nach bedarf, und initialisiert sich. Dieser letztere Teilschritt ist komponentenspezifisch und kann beispielsweise das Laden von komponenten-spezifischen Konfigurationsdateien o.ä. beinhalten. Die Initialisierung wird direkt im Konstruktor der Implementierung des Komponenteninterfaces durchgeführt.
- *Nutzung:* Die Komponente kann danach über ihr Interface von anderen Komponenten oder Anwendern der Library genutzt werden.

Diese simple Zweiteilung impliziert direkt, dass die Initialisierung in der notwendigen Reihenfolge erfolgen muss, erst die benötigten Komponenten, dann die abhängigen Komponenten.

Begründung: Die Komponenten können so einfach wie möglich gestaltet werden, es ist nicht nötig, manuell bestimmte Methoden in der richtigen Reihenfolge aufzurufen, bis auf die Initialisierung, die in der richtigen Reihenfolge erfolgen muss. Auf diese simple Art wird auch direkt jede Art der zyklischen Abhängigkeiten unterbunden, denn eine Komponente muss erst vollständig und erfolgreich registriert worden sein, bevor eine abhängige Komponente ihre Implementierung abfragen kann.

Nachteile: Keine Nachteile bekannt

***DES 036:* Komponentenbeschreibung**
Jede Komponente besitzt eine Komponentenbeschreibung als Instanz der Klasse ComponentDescription. Diese beinhaltet: Eine id, das Komponenten-Interface, die Autoren, die Version sowie eine Beschreibung der Komponente. Die ComponentDescription kann sowohl über das Interface `ISimpleComponentRegistry` abgefragt werden also auch über eine Methode des implementierten Interfaces `IComponentInterface`.

Begründung: Metadaten über eine Komponente lassen sich einfach angeben und zur Laufzeit loggen. Damit stehen alle Informationen für Fehleranalysen auch zur Laufzeit zur Verfügung.

Nachteile: Keine bekannten Nachteile

Nun zu einigen notwendigen Limitierungen bezüglich der Eindeutigkeit von

Komponenten:

***DES 037:* Ids und Interfaces müssen eindeutig sein, parallele Versionen derselben Komponente sind nicht zulässig**
Im Rahmen derselben `ISimpleComponentRegistry`-Instanz darf dieselbe Komponenten-Id gemäß DES 037 nur einmalig verwendet werden. Auch dasselbe Komponenten-Interface darf nur einmalig in einer solchen Instanz verwendet werden. Auch wenn die Komponentenbeschreibung gemäß DES 037 ein Versionsattribut enthält, heißt dies nicht, dass mehrere unterschiedliche Versionen derselben Komponente zur gleichen Zeit aktiv sein können.

Begründung: Alles andere würde einerseits DES 037 verletzen, hätte andererseits i.d.R. gar keinen sinnvollen Anwendungsfall und würde somit nur die Komplexität unnötig vergrößern.

Nachteile: Keine bekannten Nachteile

Nun zusammenfassend noch die Schnittstelle des Interfaces `ISimpleComponentRegistry`:

***DES 038:* `ISimpleComponentRegistry` bietet folgende Methoden**

- `getAllRegisteredComponentInterfaces()`: Liefert alle aktuell registrierten Komponenten-Interfaces
- `getAllRegisteredComponents()`: Liefert alle aktuell registrierten Komponenten in Form von ComponentDescriptions
- `getComponentDescription()`: Liefert zu einer gegebenen id die zugehörige ComponentDescription, für unbekannte Id eine Exception
- `hasComponent()`: Liefert zu einer gegebenen id, ob für diese aktuell eine Komponente registriert ist, oder nicht
- `hasComponentInterface()`: Liefert zu einem gegebenen Interface, ob für dieses aktuell eine Komponente registriert ist, oder nicht
- `registerComponent()`: Registriert die angegebene Komponentenimplementierung mit der gegebenen ComponentDescription
- `getComponentImplementation()`: Liefert für ein verwaltetes Interface die Implementierungs-Instanz zurück

Begründung: Die Registry kann nach ihrer aktuellen Komponentenbestückung ausgelesen werden, so kann z.B. durch vorherige Prüfung eine Exception bei nicht vorhandener oder doppelter Id umgangen werden.

Nachteile: Keine bekannten Nachteile

Schließlich noch ein Schlusswort zum Interface `IComponentInterface`:

DES 039: `IComponentInterface` **ist im Wesentlichen ein Tagging-Interface mit einer abstrakten Implementierung, von der alle Komponentenimplementierungen ableiten sollten**
`IComponentInterface` hat lediglich eine Methode zum Abfragen der ComponentDescription gemäß DES 039. Die abstrakte Implementierung übernimmt automatisch die Registrierung seiner selbst bei der übergebenen `ISimpleComponentRegistry`-Instanz.

Begründung: Weitere Methoden für Komponenten sind nicht notwendig bzw. liefern keinen Mehrwert. Die Selbstregistrierung erspart es ableitenden Klassen, diese selbst vorzunehmen und eliminiert dabei das Risiko, dies zu vergessen. Sie muss ohnehin im Rahmen der Initialisierung im Konstruktor erfolgen, sodass ein separater Aufruf auch nicht notwendig ist.

Nachteile: Keine bekannten Nachteile

11. Container API Design

11.1. Media Design

In diesem Abschnitt wird das Design der Komponente `Media` beschrieben. Grundaufgabe der Komponente ist der Zugriff auf Speicherbereiche, die Multimedia-Daten enthalten. Primär sind dies Dateien.

Den Begriff MEDIUM wollen wir hier zuerst noch etwas schärfer fassen: In "3.5 MEDIUM" hatten wir definiert: "Ein MEDIUM bezeichnet das Speichermedium der DATENBLÖCKE. Es kann sich dabei beispielsweise um eine Datei oder einen MEDIEN-STREAM, oder gar den Hauptspeicher selbst handeln."

Im Speziellen subsummiert der Begriff die Aspekte "physikalische Speicherung" und "Zugriffsmechanismus" (z.B. Dateibasiert random-access, oder byte stream). Es kann also durchaus zwei verschiedene Medien geben, die auf dieselbe physikalische Speicherung zugreifen, dies jedoch über einen anderen Zugriffsmechanismus tun. Der Begriff des Mediums ist eine Abstraktion und lässt potentiell auch ganz andere Möglichkeiten zu, wie beispielsweise media streams, Datenbanken und mehr.

11.1.1. Grundlegende Designentscheidungen `Media`

Hier werden die wesentlichen Designentscheidungen zur Komponente `Media` beschrieben.

Unterstützte Medien

Hier werden Designentscheidungen zu unterstützten MEDIEN entwickelt. Zunächst ist klar, dass `jMeta` Dateien als Grundmedium unterstützen muss.

DES 040: **Unterstützung von Random-Access-Zugriff auf Dateien**
`jMeta` ermöglicht über die Komponente `Media` die Angabe von Dateien als Ein- und Ausgabemedien mit dem Zugriffsmechanismus "Random Access".

Begründung: Dateien sind die fundamentalsten und am häufigsten anzutreffenden digitalen Medien-Behälter. Natürlich sind MP3-Dateien, AVI-Dateien etc. mit Multimedia-Inhalten weitverbreitet. Eine Library wie `jMeta` muss daher Dateien als Kernelement akzeptieren und verarbeiten können. Um Dateien effizient zu verarbeiten, ist ein Random-Access-Zugriff unerlässlich. Insbesondere ist dadurch das Lesen an beliebigen Stellen - beispielsweise Tags am Ende einer Datei - sowie das Überspringen unwichtiger Inhalte effizient umsetzbar.

Nachteile: Keine bekannten Nachteile

Aber auch lesende Streams sollen unterstützt werden, um die Anwendungsvielfalt der Library zu erhalten:

DES 041: **Unterstützung von sequentiellen, lesenden byte streams**
`jMeta` ermöglicht über die Komponente `Media` die Verwendung von lesenden byte streams, also `InputStream`s für Eingabe mit dem Modus "sequentieller Zugriff".

Begründung: `InputStream` repräsentiert aus Java-Sicht die generellste Variante eines MEDIUMS, sodass potentiell eine höhere Flexibilität für den Einsatz von `jMeta` erreicht wird. Beispielsweise lassen sich Multimedia-Dateien aus ZIP- und JAR-Dateien als Streams lesen, oder es wird auch in späteren Versionen ggf. dadurch die Unterstützung von media streams einfacher möglich - Wobei hier klar gesagt wird: media streams haben mit dieser Designentscheidung nichts zu tun. Sie könnten später auch völlig anders (beispielsweise durch ein anderes MEDIUM) umgesetzt werden.

Nachteile: Ein `InputStream` unterstützt per Definition nur sequentiellen Zugriff und keinen random-access-Zugriff (etwa via `FileInputStream`). Ggf. entsteht bei der Implementierung eine höhere Komplexität, zudem kann es zu deutlichen Performance-Einbußen bei Streams kommen, wegen mangelnder Random-Access-Fähigkeit.

Zuletzt bietet die Library noch aus Gründen der Flexibilität Zugriff auf bereits in den Hauptspeicher geladene Inhalte an:

DES 042: **Unterstützung von Random-Access-Zugriff auf byte-Arrays**
`jMeta` ermöglicht über die Komponente `Media` die Verwendung von byte-Arrays als Eingabemedium und Ausgabemedium, mit der Zugriffsmethode "Random-Access".

Begründung: Bereits durch andere Libraries oder aus bestimmten Notwendigkeiten in den Hauptspeicher geladene Inhalte können ohne Umwege direkt mit `jMeta` verarbeitet werden, was wiederum die Einsatzflexibilität der Library erhöht.

Nachteile: Keine bekannten Nachteile

Was ist nun aber mit `OutputStream`s? Dies wird im Folgenden erklärt:

DES 043: **Keine Unterstützung von schreibenden byte streams**
`jMeta` unterstützt keine schreibenden byte streams, also `OutputStream`s.

Begründung: `OutputStream`s sind nur-schreibend, aber dennoch nicht random-access. Damit ist - unter der Annahme, wir wollen auf random-access-fähige Medien auch so zugreifen - eine Zweitimplementierung neben schreibenden random-access Zugriffen zwingend erforderlich. Eine gemeinsame Verwendung von `InputStream`s und `OutputStream`s zum Lese-/Schreibzugriff auf dasselbe Medium ist von der Java-API nicht vorgesehen und führt zu diversen Problemen. Da `jMeta` bereits das Schreiben in Ausgabedateien sowie byte-Arrays unterstützt, wird aus Aufwandsgründen auf die Unterstützung von `OutputStream`s als Ausgabemedien verzichtet. Der Anwender kann die Ausgabe auf `OutputStream`s leicht selbst nachimplementieren, indem beispielsweise zuerst in ein byte-Array geschrieben wird, um dann in einen `OutputStream` zu schreiben.

Nachteile: Keine erkennbar.

Konsistenz des Medium-Zugriffs

Paralleler Zugriff auf dasselbe Medium aus verschiedenen Prozessen oder Threads, lesend durch den einen und schreibend durch den anderen, kann zu nicht unerheblichen Schwierigkeiten führen - auch ohne Einsatz von Caches. Hat man beispielsweise bestimmte Parsing-Metadaten wie Längenangaben gelesen, und wird die Datei danach durch einen parallelen Prozess verkürzt, dann kann der Versuch, die entsprechende Anzahl von Bytes zu lesen, zu einem Fehler führen.

Um dieses Problem zu umgehen, gibt es prinzipiell Sperrmechanismen für den exklusiven Zugriff auf das MEDIUM, zumindest bei Dateien. Wir legen fest:

DES 044: **Sperren von Dateien bei Zugriff durch `jMeta`**
Dateien werden *immer* während des Zugriffs durch `jMeta` explizit gesperrt, Dateiinhalte werden durch ein exklusives Lock vor dem Schreiben durch andere Prozesse und Threads geschützt. Es sei auf [PWikIO] hingewiesen, wo wir darlegen, dass man eine Datei in Java explizit zum Schreiben öffnen muss, um sie sperren zu können. "Während des Zugriffs" heißt: Nach dem ersten Öffnen der Datei und vor dem finalen Schließen. Das Lock bleibt also unter Umständen lange bestehen. Wir öffnen eine Datei auch dann zum Schreiben, wenn der Anwender explizit nur-lesend zugreifen möchte. Diese eben daher, weil wir sie sperren wollen.

Begründung: Andere Prozesse ebenso wie andere Threads in der JVM können dann nicht auf die Datei zugreifen und Daten ändern. Damit verhindern wir Konsistenzprobleme, die zu Fehlersituationen in `jMeta` führen können. Dass wir das beim (Java-seitigen) nur-lesenden Zugriff nicht tun können, ist ein Wehrmutstropfen, aber wir öffnen dennoch immer zum Schreiben.

Nachteile: Andere Prozesse und Threads können nicht schreibend zugreifen - der Anwendungsfall, auf dieselbe Datei über mehrere Threads mittels`jMeta` zuzugreifen, wird damit explizit nicht unterstützt. In einigen Unix-Varianten signalisiert das Locken einer Datei nur eine Absicht, und schreibende Zugriffe durch parallele Prozesse sind weiterhin möglich.

Es sei allerdings darauf hingedeutet, dass das Sperren sowohl von Hauptspeicherinhalten als auch von Byte-Streams keinen Sinn macht bzw. nicht möglich ist, wie in den folgenden Designentscheidungen erklärt:

DES 045: **Kein Sperren von byte streams**
Rohe byte streams werden nicht gesperrt

Begründung: Das Interface `InputStream` bietet keine Sperrmethoden an. `jMeta` wird nicht auf die Art des Streams schließen und diesen dann ggf. doch sperren, z.B. indem ein `FileInputStream` erkannt wird.

Nachteile: Keine bekannt

DES 046: **Kein Sperren von byte-Arrays**
Byte-Arrays werden nicht gesperrt

Begründung: Speicherbereiche sind zunächst einmal durch Betriebssystemmechanismen vor dem Überschreiben durch andere Prozesse geschützt (in der Regel). Es stellt sich evtl. die Frage, wie man sie vor Manipulationen anderer Threads schützen will. Die Antwort: Gar nicht. Einerseits bekommt der Anwender ohnehin eine Referenz auf die rohen Bytes, wenn er dies wünscht (siehe "4.3 ANF 003: Spezifikation unterstützter Metadaten- und Containerformate erfüllen"). Und diese kann er damit auch im gleichen Thread - unter Umgehung von `jMeta` beliebig manipulieren. Andere Threads können dies genauso tun, oder aber direkt durch schreibende Aktionen auf `jMeta`. Dies kann zwar ggf. durch Thread-Sperrmechanismen verhindert werden (z.B. bei jedem Herausreichen der Referenz eine Semaphore setzen, so dass andere schreibende und lesende Aufrufe nicht mehr zugreifen können - und dann blockieren?), widerspricht aber DES 046. Die Komplexität dieser Mühe rechtfertigt ihren Nutzen in keinster Weise, zumal - wie oben dargelegt - Zugriffe auf das Referenz-Array durch denselben Thread dann immernoch möglich sind. Der Anwender von `jMeta` muss Threadsicherheit selbst sicherstellen.

Nachteile: Keine bekannt

Vereinheitlichte API für Medienzugriff

Wir haben im Wiki-Artikel [PWikIO] klar die Unterschiede zwischen byte streams und random-access-Dateizugriffen herausgestellt. Bei so vielen Unterschieden stellt sich die Frage: Kann man dies überhaupt vereinheitlichen bzw. macht der Aufwand hier Sinn? Der kleinste gemeinsame Nenner zwischen random-access-Dateizugriff und `InputStreams` ist das lineare Lesen aller Bytes im Medium. Dies ist eindeutig zu wenig. Es verleugnet alle Vorteile des random-access-Zugriffs. Eine Schnittmenge der Features für eine Vereinheitlichung kann daher nicht sinnvoll sein.

Vielmehr streben wir eine vereinheitlichende Kombination an:

***DES 047:* Vereinheitlichter Zugriff auf alle unterstützten Medien in einer API**
`Media` bietet eine gemeinsame Abstraktion zum Zugriff auf random-access-Dateien, `InputStream`s sowie byte-Arrays. Diese API bietet die Vorteile beider Zugriffsarten über ein einheitliches Interface an. Die Implementierung wirft in manchen Fällen Ausnahmen der Art "Operation not supported", wenn ein Feature vom Medium nicht unterstützt wird, in anderen Fällen wird ein sinnvolles Alterantivverhalten implementiert. Der anwendende Code muss an einigen Stellen Fallunterscheidungen machen.
Während byte-Arrays kein Problem für die Abstraktion darstellen, haben jedoch auch random-access-Dateien und `InputStream`s mehr gemeinsam, als man gemeinhin denkt:

- Die Operationen Öffnen, (sequentielles) Lesen, Schließen.
- Auch `InputStream`s kann man (zumindest technisch) einen Anfang, Offsets und ein Ende zuordnen.
- Auch Dateien können read-only sein, was `InputStream`s perf Definition immer sind.

Schreibende Zugriffe auf ein read-only-Medium (damit insbesondere auch einen `InputStream`) werden mit einem Laufzeitfehler quittiert.
Der wesentliche Unterschied zwischen Dateien und `InputStream`s ist eben: Wahlfreier Zugriff ist auf Dateien möglich, während `InputStream`s nur sequentiell ausgelesen werden können. Dieser Unterschied ließe sich aber prinzipiell beispielsweise durch Mechanismen wie Zwischenspeicherung o.ä. verringern.

Begründung: Die API der Komponente `Media` wird nach Außen insgesamt einfacher, die Verwendung komfortabler. Verwendende Komponenten der Komponente `Media` können ihren Verwendern eine einfachere Schnittstelle zur Verfügung stellen. Gleichzeitig bleiben die Vorteile der beiden Ansätze random-access vs. byte stream weiter verfügbar.

Nachteile: Einige wenige Operationen der API können notwendigerweise nicht von beiden Medienarten gleichartig umgesetzt werden, so dass in manchen Fällen Fallunterscheidungen um Client-Code nötig werden.

Zweistufiges Schreibprotokoll

Beim Schreiben geht es um die Bündelung von Zugriffen, auch eine Art von Pufferung. Wir wollen für optimale Schreibperformance tatsächlich eine solche Bündelung umsetzen. Daher halten wir in folgender Designentscheidung fest, wie das Schreiben in `jMeta` generell umgesetzt werden soll:

DES 048: `Media` **nutzt ein zweistufiges, durch den Anwender gesteuertes Protokoll für das Schreiben von Daten**
Die erste Stufe ist das Anmelden von Änderungen, die auf das externe Medium geschrieben werden sollen. Hier wird also noch nicht auf das externe Medium zugegriffen. Die zweite Stufe ist die Operation *flush*, das Schreiben aller Änderungen. Die unterliegende Implementierung bündelt die Schreibaktionen je nach Bedarf in ein oder mehrere Pakete und führt das Schreiben auf das externe Medium erst in der zweiten Stufe durch.

Begründung: Es ist eine effiziente Implementierung des Schreibens möglich, intern können Schreibaktionen je nach Bedarf gebündelt werden. Und dies, ohne den Anwender der Komponente dazu zu zwingen, dies selbst zu tun, er kann sie zu den Zeitpunkten und an den Stellen absetzen, an denen dies notwendig ist. Damit wird die Struktur der Schreibaktionen des Anwendercodes nicht unnötig mit Auflagen belastet. Zudem entsteht dadurch auch die potentielle Möglichkeit eines "undo", also des Rückgängigmachens der Änderungen.

Nachteile: Fehler, die beim tatsächlichen Schreiben auf das externe Medium erfolgen, werden ggf. erst spät erkannt. Die anwendende Komponente muss sich bewusst sein, dass das die Schreib-Aufrufe selbst (Schritt 1) sehr performant ist, das eigentliche physikalische Schreiben (Schritt 2) der Änderungen mit *flush* dann aber sehr lange dauern kann. Zudem muss der Anwender daran denken, dass Schritt 2 explizit notwendig ist.

Es soll aber klar darauf hingewiesen werden, dass dies nicht analog zu O/R-Mappern Hibernate oder OpenJPA ein Transaktionsprotokoll darstellt. Das oben eingeführte Protokoll ist ganz klar nicht ACID-fähig! Dazu die folgende Designentscheidung, die eher eine Art Ausschluss darstellt:

***DES 049:* Schreiben in `Media` garantiert keinesfalls ACID, bei Fehlern in *flush* gibt es kein rollback**
ACID (atomicity, consistency, isolation und durability) werden durch die Implementierung von `Media` und allgemein durch Java File I/O nicht sichergestellt. Erfolgt beispielsweise beim Schreiben der zweiten Änderung in *flush* ein Fehler, so werden die darauffolgenden Daten nicht mehr geschrieben, sondern das flushing bricht ab. Zudem wurde die erste Änderung bereits auf das externe Medium dauerhaft geschrieben, dies wird nicht rückgängig gemacht. Die Operation *undo* darf also nicht mit Rollback verwechselt werden und ist auch keine automatisch durchgeführte Aktion. Während Isolation und Durability noch mehr oder weniger gewährleistet werden können, ist der Anwender selbst zuständig, Konsistenz herzustellen.

Begründung: Ein Transaktionsmanager, der ACID sicherstellt, und das für Dateien, ist sehr schwierig korrekt umsetzbar. Diese Anforderung ist übertrieben, keine andere Library setzt diese so um. `jMeta` soll nicht als Datenbank verwendbar sein.

Nachteile: Keine erkennbar

Welche schreibenden Operationen sollen angeboten werden? Eine Methode write() - die einzige echt schreibende Methode der Java File I/O - ist nicht ausreichend. Denn wie löscht man mit dieser Methode? Überschreiben ist hier das Stichwort. `Media` muss eine bessere API zur Verfügung stellen, die dem Anwender einige Arbeit abnimmt. Wir werden hier nur die notwendigen Operationen angeben, ohne auf deren Implementierung einzugehen - dies wird später nachgeholt.

DES 050: **`Media` bietet die schreibenden Operationen *insert*, *replace* und *remove***
Der Anwender kann:

- N Bytes an einer bestimmten Stelle einfügen (*insert*)
- N Bytes an einer bestimmten Stelle durch ebenso genau N neue Bytes ersetzen (*replace*)
- N Bytes an einer bestimmten Stelle löschen (*remove*)

Begründung: Dies sind die Operationen, die für eine Metadaten-Library prinzipiell immer notwendig sind: replace ist für Formate mit statischer Länge (wie ID3v1) und solche, die einen Padding-Mechanismus oder etwas vergleichbares bieten (wie ID3v2) perfekt einsetzbar. Bei dynamisch erweiterbaren Formaten wie ID3v2 benötigt man Möglichkeiten, neue Datenblöcke einzufügen (insert) und zu löschen (delete).
Die Bürde, diese bequemen Operationen mittels der Java File I/O zu implementieren, die letztlich nur write() und truncate() bietet, wird von `Media` übernommen, sodass sich der Anwender nicht darum kümmern muss.

Nachteile: Keine bekannten Nachteile

Da wir einen zweistufigen Schreibprozess verwenden, ist das "Rückgängig-Machen" noch nicht geflushter Änderungen möglich. Ist es auch sinnvoll?

DES 051: **Schreibende Operationen auf einem Medium vor einem *flush* können via *undo* rückgängig gemacht werden**
Schreibende Operationen führen zu schwebenden Änderungen gemäß DES 051. Diese können vor einem *flush* rückgängig gemacht werden.

Begründung: Die Anwendungslogik kann das *undo* von Änderungen in manchen Fällen erforderlich machen. Statt dann die entgegensetzte Operation aufrufen zu müssen, ist es für den Anwender einfacher, eher die Operation selbst direkt ungeschehen zu machen. Umgekehrt können sich einander genau aufhebende Operationen wie das Einfügen von n Bytes an Position x und das nachgelagerte Löschen von n Bytes an Position x durch undo umgesetzt werden, und führen so nicht zu unnötigen Einträgen in den internen Datenstrukturen. Somit muss der Anwendungscode nicht notwendigerweise selbst über Änderungen buchführen.

Nachteile: Keine erkennbar

Caching

Die Komponente `Media` übernimmt die Aufgabe der I/O mit potentiell langsamen Eingabe- und Ausgabe-Medien. Daher sind hier die wesentlichen Performance-Probleme der Gesamtlibrary zu lösen. Wir werden die Themen mit einigen Be-

gründungen und Herleitungen angehen.

In [PWikIO] wird Grundlegendes zur Performance bei Dateizugriffen vermittelt. Die oberste Maxime für performantes I/O ist die Minimierung von Zugriffen auf das externe, potentiell langsame Medium. Wir haben für das Schreiben bereits DES 051 eingeführt. Die Frage ist, wie man das Lesen performant gestalten kann.

Zunächst relativ klar ist, dass man sich beim Lesen im Sinne einer verbesserten Performance mit Pufferung behelfen sollte:

***DES 052:* Lesende Zugriffe können durch einen Puffer-Mechanismus durchgeführt werden, gesteuert durch den Anwender**
Bei jedem Lesezugriff kann der anwendende Code die Anzahl zu lesender Bytes vorgeben, dies entspricht einem Buffering. Der Code steuert dabei selbst, wie groß der Buffer ist. Potentiell kann er also auch 1 Byte lesen. Es liegt in der Verantwortung des anwendenden Codes, hier eine sinnvolle Menge an Bytes zu lesen.

Begründung: Eine pauschal eingebaute Pufferung mit konstanter Byte-Anzahl würde eher zu Performancenachteilen führen, weil unter Umständen bei vielen Zugriffen zu viele Bytes gelesen werden, viel mehr als notwendig. Zudem müsste man unter Umständen bei einer festen Puffergröße zweimal statt nur einmal lesen. Gemäß [PWikIO] gibt es eben kein Patentrezept für Puffergrößen bei File I/O. Daher steuert der anwendende Code.

Nachteile: Keine bekannt.

Ein weiterer wichtiger Aspekt ist Caching: Unter *Caching* verstehen wir hier die unter Umständen langdauernde Zwischenspeicherung von Inhalten eines MEDIUM im RAM, um einen schnelleren Zugriff zu erreichen. Pufferung unterscheidet sich von Caching, da Pufferung nur kurzlebige Zwischenspeicher ohne Synchronisierungsbedarf benötigt.

Zunächst einmal kann man abgesehen von der bereits behandelten Pufferung unterschiedliche Arten von "Caching" in einer auf `jMeta` basierenden Anwendung erkennen:

- Beim Dateizugriff gibt es Caches auf HW-Ebene, im Betriebsystem bzw. Dateisystem.
- Java unterstützt temporäre Pufferung durch den `BufferedInputStream`, und Caching explizit über `MappedByteBuffer`.
- Anwendungen, die lediglich an menschenlesbaren Metadaten interessiert sind, lesen diese, konvertieren diese über `jMeta` in ihre Klartext-Darstellung und zeigen diese in ihrer GUI an. In solchen Fällen stellt die Speicherung in der GUI eine Art Caching dar, da i.d.R. bei Änderungen der Anzeige kein erneutes Lesen auf das Medium notwendig wird, ein einmaliges Lesen reicht völlig. Hierbei handelt es sich freilich nicht um ein hardware-nahes Cachen von Rohdaten, sondern um ein Caching von menschenlesbaren Daten.

Wenn wir all diese Varianten betrachten, wieso benötigt man noch ein eingebautes Caching in `jMeta`? Betrachten wir zur Beantwortung dieser Frage noch zunächst ein paar Anwendungsszenarien der Library: Ein Anwendungsszenario ist das bereits erwähnte Auslesen von Metadaten aus einer Datei, um diese an einer GUI anzuzeigen. Für diesen Fall wäre ein Caching übertrieben. Man liest einmal, und ggf. nochmals wenn es der Anwender ein weiteres Mal explizit wünscht. Dies würde vermutlich auch ohne Caching zu einer akzeptablen Performance führen. Ein anderer Anwendungsfall ist das beliebige Hin- und Herspringen in einer Container-Datei mit der low-level-API, um bestimmte Inhalte zu verarbeiten. Dies ist an sich erstmal "random access". Will man aber dieselbe Stelle mehrfach lesen? Unter Umständen ja. Will man dann einen expliziten Mediums-Zugriff oder nicht? Evtl. ja, oder auch eben nicht.

Die letzte Frage wirft ein grundlegendes Problem mit dem Caching auf: Die der Synchronizität mit dem externen Medium. Falls das Medium zwischenzeitlich verändert wurde, ist der Cache-Inhalt u.U. veraltet und ungültig. Dies kann Code, der auf den Cache zugreift, in aller Regel nicht erkennen.

Wir fassen hier zunächst die bereits erkannten Vor- und Nachteile von Caches für unsere Zwecke zusammen:

Vorteile	**Nachteile**
+ Performance-Verbesserung bei mehrfachen lesenden Zugriffen, da der Cache-Zugriff ggf. deutlich schneller ist als der externe Speicherzugriff	− Bei einmaligem Lesezugriff keine Performance-Vorteile
+ In einem Cache ist man prinzipiell deutlich flexibler als auf langsamen externen Medien, Umordnungen, Korrekturen und Bündelungen sind (einfacher) möglich.	− Änderungen am MEDIUM können nicht erkannt werden und führen zu ungültigen Cache-Inhalten, die wiederum bei nachfolgenden Aktionen zu Fehlverhalten und Korruption von Daten führen können.
	− Für Caching ist in aller Regel zusätzlicher Code notwendig, beispielsweise müssen Fragen wie "wann werden die Daten freigegeben?" beantwortet werden. Für den Umgang mit Konsistenzthemen wird Code i.d.R. komplexer als ohne Cache.
	− Mehr Heap-Speicher notwendig.

Table 11.1.: Vor- und Nachteile von Caching in `jMeta`

Die Nachteile in der Tabelle überwiegen. Warum sollte man also Caching in `jMeta` einsetzen? Die Designentscheidungen DES 052, DES 052 und DES 052 haben gemeinsam, dass sie durch Caching einfacher oder besser umgesetzt werden können:

DES 053: **`Media` nutzt dauerhafte Zwischenspeicherung (Caching)**
`Media` nutzt einen RAM-basierten, dauerhaften Zwischenspeicher (Cache) zum Speichern der bereits aus dem MEDIUM gelesenen Daten. Darauffolgende Lese-Zugriffe greifen auf die Cache-Inhalte (falls vorhanden) zu. Schreibende Aktionen schreiben erst in den Cache, um dann durch ein *flush* final auf das Ausgabe-MEDIUM geschrieben zu werden, wie in DES 053 angegeben.

Begründung:

- Dies kann zur direkten Umsetzung von DES 053 im Sinne einer Pufferung beim Lesen verwendet werden. Der Cache fungiert hierbei als der in DES 053 genannte Zwischenspeicher. Allerdings war er in DES 053 nicht notwendigerweise als dauerhaft vorausgesetzt worden.
- Zudem hilft dies bei der Umsetzung von DES 053: Die angemeldeten Schreiboperationen werden im ersten Schritt des zweistufigen Leseprotokolls zunächst nur in den Cache oder eine verwandte Datenstruktur geschrieben.
- Das Abbilden der in DES 053 geforderten Operationen auf Medium-Zugriffe kann über einen Cache optimiert implementiert werden. Beispielsweise wird ggf. erneutes Lesen von Medium-Daten erforderlich sein, bevor Schreioperationen durchgeführt werden können. Die damit verbundenen Umorganisationen von Daten können nur in einem Zwischenspeicher erfolgen, da noch nichts festgeschrieben worden ist.

Nachteile: Wurden Tabelle 11.1 bereits detailliert dargelegt. Die Alternative ist ja immer der direkte Mediums-Zugriff ohne zwischengeschalteten Cache. Die folgenden Nachteile gibt es gegenüber dieser Direktvariante:

- Höhere Code-Komplexität
- Mehr Heap-Speicher wird benötigt, da der Cache dauerhaft Speicher belegt
- Das Medium kann sich durch externe Prozesse verändern, sodass es nicht mehr mit den Cache-Inhalten synchron sind.

Man kann sogar noch einen Schritt weiter gehen, und das Caching auch zur besseren Umsetzung von DES 053 nutzen, wie in folgender Designentscheidung dargelegt:

***DES 054:* Caching wird zur besseren Angleichung von `InputStream`s und Dateien gemäß DES 054 genutzt.**
Die aus einem `InputStream` gelesenen Daten werden im Cache hinterlegt. Leseaktionen können daher auf vorhergende Bytes "zurückgehen", indem statt des Direktzugriffs auf das Medium die Bytes aus dem Cache gelesen werden. "Vorgriffe" auf Bereiche, die erst gelesen werden müssen, führen dazu, dass bis zur entsprechenden Stelle gelesen wird.

Begründung: Dies setzt DES 054 nahezu vollständig um, "transparent" für den Anwender.

Nachteile: Noch mehr Heap-Speicher bei `InputStream`s, was zu `OutOfMemoryError`s führen kann.

Natürlich ist der in DES 054 genannte Nachteil schwerwiegend. Wenn man nichts tun würde, um diesen Nachteil zu verringern oder zu vermeiden, dann wäre DES 054 unsinnig, denn die damit erzielten Vorteile werden durch die Nachteile dramatisch überschattet.

Als erster Schritt sind die folgenden drei Designentscheidungen notwendig:

***DES 055:* Der Anwender kann Cache-Inhalte freigeben, zudem kann er das Caching deaktivieren.**
Das Freigeben der Cache-Inhalte kann feingranular für einen angegebenen Offset-Bereich erfolgen.
Damit kann der Anwender selbst die aktuelle Größe des Caches steuern, wobei klar sein muss, dass nachfolgende random-Access-Leseoperationen bei random-access-Medien zu erneuten langsamen Zugriffen auf das Medium führen, während bei `InputStream`s eine entsprechende Exception geworfen werden muss, da keine Daten für vorherige Offsets vorhanden sind und über den `InputStream` kein "Zurückgehen" möglich ist.
Darüber hinaus kann der Anwender das Caching vollständig deaktivieren. Auch hier ist der Zugriff auf Daten vorheriger Offsets bei `InputStream`s nicht möglich und wird mit einer Exception quittiert.

Begründung: Der Anwender steht in der Verantwortung, über die Speichermenge zu entscheiden: Ist das Medium beispielsweise klein, kann er das Caching in Kauf nehmen. Ist er groß, hat er zwei Möglichkeiten: Den Cache regelmäßig aufräumen (z.B. die zuerst gelesenen Daten ab einer bestimmten Größe), oder aber das Caching komplett zu deaktieren, was dann aber eine schritthaltende Verarbeitung notwendig macht.

Nachteile: Die Implementierung von `Media` wird durch entsprechende Fallunterscheidungen komplexer.

***DES 056:* Die `Media` API ermöglicht für `InputStreams` das explizite Überspringen von Bytes**
Bytes müssen nicht notwendigerweise gelesen und an den Anweder geliefert werden. Stattdessen ist auch ein Überspringen (`skip`) möglicht. Bei `InputStreams` ist das Überspringen eine eingebaute Funktionalität. Bei random-Access-Zugriff ist das Überspringen eher nicht bedeutsam.

Begründung: Der Anwender kann explizit Daten, die nicht bedeutsam sind (z.B. nicht verarbeitet werden sollen), explizit überspringen.
Zweiter Grund ist, dass DES 056 es zunächst erfordert, dass beim Lesen von höheren Offsets (also Bereichen, die bisher noch nicht gelesen worden sind) im Falle von `InputStreams` ggf. eine große Anzahl von Bytes gelesen werden muss, um an die angegebene Stelle zu gelanden. Dies lässt den Cache anschwillen. Hier ist das Überspringen eine gute Alternative, da es den Cache nicht wachsen lässt.

Nachteile: Keine Nachteile erkennbar

***DES 057:* Performance-Nachteile von lesenden byte streams explizit dokumentieren**
Verringert man die Unterschiede zwischen `InputStreams` und Dateien durch Caching gemäß DES 057: Dann muss man damit umgehen, dass man niemals potentiell unendlich lange `InputStreams` im Cache speichern kann. Der Anwender kann bei Dateizugriff ja z.B. auch zwischen `FileInputStream` und dem direkten Zugriff über Dateien wählen. Die dann eventuell durch die Verwendung von `FileInputStream` begründeten Performance-Nachteile beim Lesen gegenüber direktem random-access-Dateizugriff - die durch Umsetzung einer einheitlichen API gemäß DES 057 entstehen - werden in der `jMeta`-Dokumentation explizit erwähnt und dargestellt. Die Ausweichmechanismen (Skippen von Bytes, Speicherfreigabe, Deaktivieren des Cachings) werden mit ihren Folgen explizit dargestellt.

Begründung: Es werden keine falschen Erwartungshaltungen geweckt und zukünftige Bugs vermieden. Der Vertrag ist klar gegenüber dem Anwender dargestellt. Er muss das für seine Anwendungszwecke geeignete Medium wählen.

Nachteile: Keine bekannten Nachteile

Nach diesen Ausführungen verdienen die in Tabelle 11.1 und in DES 057 dargestellen Nachteile des Caching hier noch eine abschließende Betrachtung:

- Code-Komplexität: Die höhere Code-Komplexität kann man nicht wegdiskutieren. Diese muss man bei der Verwendung eines dauerhaften Cachings in Kauf nehmen, wenn man überzeugt ist, dass ansonsten die Vorteile überwiegen. Man muss dann aber gewissenhafte und extensive Integrationstests erstellen.

- Mehr Heap-Speicher: Es ist üblich, dass man eine bessere Laufzeit manchmal durch eine schlechtere Speicherkomplexität erreichen kann. So auch hier. Um dennoch `OutOfMemoryError`s zu vermeiden, haben wir mit DES 057, DES 057 und DES 057 genügend Ausweich-Möglichkeiten dargestellt.

- Mangelnde Synchronität: Dass der Cache ggf. Updates enthält, die sich noch nicht auf dem externen Medium befinden, ist gemäß DES 057 erlaubt und darüber muss man sich keine Sorgen machen. Das Problem stellen Änderungen an der Datei durch anderen Prozesse und Threads dar. Diese können durch `jMeta` nicht sinnvoll allgemeingültig gehandhabt werden. Daher wurde das Sperren von Medien in DES 057 dargestellt. Auch dieses kann bei manchen Betriebssystemen keinen vollständigen Schutz gegen externe Manipulationen bieten. Führen diese externen Manipulationen zu nicht auflösbaren Inkonsistenzen oder unerwarteten Fehlern, dann wird dies dem Anwender entsprechend über einen Laufzeitfehler signalisiert.

Die Umsetzung des dargestellen Caching-Mechanismus stellt eine große Herausforderung dar. Sie wird später im Implementierungsabschnitt besprochen. Hier können wir zunächst nur einen Weg ausschließen:

DES 058: **`MappedByteBuffer` wird für Zwischenspeicherung nicht verwendet**
Man könnte auf die Idee kommen, nun `MappedByteBuffer` für das in DES 058 genannte Caching zu verwenden. Dies wird aber explizit nicht getan, stattdessen wird eine per Hand implementierte Zwischenspeicherung umgesetzt.

Begründung: Es ist nicht garantiert, dass das jeweilige Betriebssystem einen `MappedByteBuffer` überhaupt unterstützt, und auch nicht, dass die Daten wirklich aktuell im Hauptspeicher sind. Für jede zusammenhängende Region muss man einen neuen `MappedByteBuffer` inklusive erneutem Betriebssystemaufruf erzeugen.

Nachteile: Evtl. ist das per Hand umgesetzte Caching schwieriger umsetzbar.

Halten wir zuletzt noch folgende sinnvolle Sonderbehandlung des Cachings bei byte-Array-Medien fest:

DES 059: **Für byte-Array-Medien ist das Caching immer deaktiviert**
Für byte-Array-Medien ist das Caching immer deaktiviert

Begründung: byte-Arrays befinden sich bereits im RAM, ein Caching würde nur unnötigen Overhead darstellen

Nachteile: Keine bekannten Nachteile

Lesender Zugriff auf das Medium

Das zweistufige Protokoll für das Schreiben, das in "11.1.1 Zweistufiges Schreibprotokoll" vorgestellt worden ist, wirft einige Fragen bezüglich des Lesens von Daten auf. Die wichtigste darunter ist: Was muss der Anwender nach dem Aufruf von schreibenden Operationen (Schritt 1) und vor deren *flush* (Schritt 2) beachten? Insbesondere: Was liefern lesende Aufrufe nach bereits erfolgten Änderungen, aber vor deren *flush* zurück? Die Antwortmöglichkeiten:

1. Entweder den letzten festgeschriebenen Stand, also den Stand des externen Mediums nach dem letzten *flush*,

2. Oder aber bereits einen Stand, der die "schwebenden" Änderungen durch den Aufruf der schreibenden Operationen mit enthält?

Man könnte sich nun auf den Standpunkt stellen: Natürlich Variante (2), denn so funktionieren ja auch (meist) Transaktionen auf Datenbanken: Was man bereits in derselben Transaktion geschrieben hat, wird auch wieder gelesen, auch wenn es noch nicht auf der DB festgeschrieben ist. Aus Sicht von `jMeta` gilt:

> ***DES 060:* Der Anwender kann nur den aktuell auf dem Medium festgeschriebenen Datenstand lesen**
> Auch bei vorhandenen, vorgemerkten Änderungen, die noch nicht durch ein *flush* festgeschrieben sind (z.B. Einfügungen, Löschungen), kann der Anwender mit `Media` nur den zuletzt festgeschriebenen Stand lesen.
>
> **Begründung:** Die Änderungen hat der Anwender angemeldet und daher auch die entsprechenden neu zu schreibenden bzw. zu überschreibenden Daten in der Hand. Daher ist deren alleinige Verwaltung durch `Media` zu diesem Zeitpunkt noch nicht zwingend erforderlich. Zudem muss es weiterhin möglich sein, die binären Daten eines zur Löschung angemeldeten Datenblocks lesen zu können. Die Lese-Operation wird dadurch wesentlich weniger komplex, da sie nicht zusätzlich die bereits angemeldeten, aber nicht festgeschriebenen Änderungen behandeln muss. Weiterer Vorteil ist, dass Code, der Daten liest, immer nur auf einem festgeschriebenen Stand arbeitet. Daher kann es nicht zu der Situation kommen, das Logik auf Basis nicht festgeschriebener Daten durchgeführt wird.
>
> **Nachteile:** Die Erwartungshaltung "was vorher (wenn auch nur schwebend) geschrieben worden ist, das liest man auch wieder" wird nicht erfüllt. Der Anwender muss für geänderte oder neu eingefügte Daten die Bytes selbst (temporär) verwalten.

In "11.1.1 Caching" wurde das Caching ausführlich diskutiert. Zunächst benötigen wir eine explizite Operation zum Puffern von Daten, ohne diese direkt zurückzuliefern:

***DES 061:* Explizite Operation zum Puffern von Medium-Daten**
Es wird eine Operation *cache* bereitgestellt, die n Datenbytes ab einem bestimmen Offset puffert, ohne diese direkt zu liefern. Darüber hinaus wird eine Abfrageoperation bereitgestellt, welche die Anzahl der an einem bestimmten Offset aktuell gecachten Bytes (ohne Lücken) zurückliefert.

Begründung: Für die Umsetzung von DES 061 notwendig

Nachteile: Keine bekannten Nachteile

Darüber hinaus muss man natürlich noch an die Daten selbst herankommen:

***DES 062:* Explizite Operation zum Abholen von Daten**
Es wird eine Operation *getData* bereitgestellt, die n Datenbytes ab einem bestimmen Offset zurückliefert

Begründung: Ohne diese könnten keine Daten vom Medium gelesen werden

Nachteile: Keine bekannten Nachteile

Nun stellt sich die Frage, wie die Operation *getData* mit dem Cache zusammenspielt, die hier beantwortet wird:

***DES 063: getData* kombiniert Daten aus Cache und Medium und aktualisiert dabei den Cache, falls nötig**
getData liest die Daten aus dem Cache, falls vollständig darin vorhanden. Sind die Daten nicht vollständig im Cache vorhanden, liest *getData* die vorhanden Bytes aus dem Cache, sowie die nicht vorhandenen Bytes aus dem Medium und hinterlegt sie im Cache. Die Daten werden dann aus den beiden Quellen zusammengefügt. Der Anwender kann bei jedem Aufruf steuern, ob sich *getData* wie in den letzten Sätzen beschrieben verhält, oder in jedem Fall direkt auf das Medium zugreift, den Cache also ignoriert, diesen dann aber aktualisiert.

Begründung: Um effizientes Lesen zu ermöglichen, kann *getData* so verwendet werden, dass Daten - wenn möglich - aus dem Cache entnommen werden, und nur im Zweifelsfall auf das Medium zugegriffen werden muss. Der erzwungene Direktzugriff auf das Medium wird aber dennoch als zusätzliche Möglichkeit angeboten, um beispielsweise Synchronisierungsprobleme ausgleichen zu können (die aber wegen Sperrung niemals auftreten sollten!). Aktualisierung des Cache durch neu gelesene Daten ist zur Performancesteigerung sinnvoll.

Nachteile: Keine bekannten Nachteile

Wir wollen noch definieren, wie die gelesenen Mediendaten repräsentiert werden:

***DES 064:* Gelesene Mediendaten werden als read-only `ByteBuffer` repräsentiert**
Gelesene Daten werden nicht als `byte`-Arrays zurückgeliefert, sondern als `ByteBuffer`-Instanzen.

Begründung: Einerseits können Anwender direkt von den Konvertierungs-Funktionen von `ByteBuffer` profitieren, zum anderen ist die Implementierung bezüglich der Inhalte der `ByteBuffer` flexibler, da nur die Bytes zwischen `position()` und `limit()` gelesen werden können. Damit kann z.B. ein intern verwalteter, viel größerer `ByteBuffer` verwendet werden, auf den nur eine kleine Sicht herausgegeben wird.

Nachteile: Keine bekannten Nachteile

Kommen wir nun zum Thema der timeouts. Bei Java kann jeder lesende und schreibende Aufruf blockieren. Wie wollen wir damit umgehen? Die folgende Designentscheidung legt dies klar fest.

***DES 065:* `jMeta` unterstützt nur Lese-Timeouts, und dies nur für byte streams**
`Media` bietet nur die Möglichkeit, Lese-Timeouts in Millisekunden für byte streams direkt auf dem jeweiligen `InputStreamMedium` zu konfigurieren.

Begründung: Wir gehen davon aus, dass das Lesen von Daten aus einer Datei eben eine gewisse Weile dauert, und nehmen in Kauf, dass dies auch blockiert. Das Lesen aus Dateien ist im Allgemeinen aber kein Kandidat für lange (oder überhaupt) Blockierungen. Da die Implementierung eines Timeout-Verfahrens komplex ist, verzichten wir für Dateien daher darauf. Für `InputStream`s hingegen wie media streams oder Socket-Verbindungen sind Blockierungen ein häufig zu erwartender Sachverhalt. Daher sind timeouts beim Lesen aus byte streams sehr sinnvoll und `jMeta` bietet die Möglichkeit einer Konfiguration für solche timeouts. Schreibende Operationen für byte streams werden gemäß DES 065 nicht unterstützt.

Nachteile: Höhere Komplexität bei der Umsetzung des Zugriffs auf ein `InputStreamMedium`.

11.1.2. API Design

Auf Basis der im letzten Abschnitt festgelegten grundlegenden Designentscheidungen können wir nun ein Design der API der Komponente `Media` entwickeln. Die API bezeichnet die öffentliche Schnittstelle der Komponente, also alle Klassen, die andere Komponenten verwenden können, um die `Media`-Funktionalität zu nutzen.

Repräsentation eines Medium

Da das Medium bereits sehr oft als Begriff aufgetaucht ist, macht eine Repräsentation durch eine Klasse Sinn.

***DES 066:* Medien werden als Interface und Implementierungsklassen repräsentiert, mit folgenden Eigenschaften**
Ein Medium wird also als Java-Interface `IMedium` repräsentiert und ermöglicht es so den Anwendern von `jMeta`, ein konkret zu verwendendes Medium (das sind dann die Implementierungen dieses Interfaces) anzugeben. Als Implementierungen unterstützen wir gemäß DES 066 ein `FileMedium`, gemäß DES 066 ein `InputStreamMedium` sowie gemäß DES 066 ein `InMemoryMedium`. Ein Medium hat nur die folgenden Eigenschaften:

- Ist random-access: Ja/Nein - **Begründung:** Diese Eigenschaft hat starken Einfluss auf den Schreib- und Lesevorgang, ist aber gleichzeitig eine intrinsische Eigenschaft des MEDIUM selbst und nicht des Schreib- und Lesevorgangs. Also wird die Eigenschaft direkt beim MEDIUM hinterlegt.
- Existiert aktuell: Ja/Nein - **Begründung:** Existenzprüfung kann in Java am MEDIUM selbst vorgenommen werden.
- Ist nur lesbar: Ja/Nein - **Begründung:** Diese Eigenschaft deaktiviert praktisch das Schreiben, wenn auf "Ja" gesetzt. Einge MEDIEN können niemals geschrieben werden (beispielsweise `InputStream`s), für andere ist dies aber möglich. Dieses Flag soll auch genutzt werden, um dem `jMeta`-Anwender eine Möglichkeit zu geben, zu signalisieren, nur-lesend zugreifen zu wollen.
- Aktuelle Länge in Bytes (nur für random-access relevant) - **Begründung:** Java bietet für jede unterstütze Art von `IMedium` außer für `InputStream` Abfragen ab, so dass dies direkt in der `IMedium`-Implementierung umgesetzt werden sollte. Für `InputStream` und generell nicht-random-access-Medien spielt der Begriff der Länge allerdings keine Rolle. Somit wird hier ein Wert geliefert, der eine unbekannte Länge widerspiegelt. Im Sinne der Designentscheidung DES 066 handelt es sich hierbei um die aktuell festgeschriebene Länge, und nicht die durch ggf. aktuell schwebende, aber nicht festgeschriebene Schreiboperationen geänderte Länge.
- Einen Klartextnamen, des MEDIUMS - **Begründung:** Dieser ist sinnvoll und hilfreich zur (informellen) Identifikation des `IMedium` z.B. in Logausgaben und kann z.B. aus dem Dateinamen abgeleitet sein.
- Das "gewrappte" eigentliche Objekt zum Zugriff auf das Medium, z.B. die Datei, der `InputStream` oder das byte-Array.

Begründung: Es kann genau gesteuert werden, welche Medien unterstützt werden. Der Anwender kann bequem das zu verwendende Medium angeben. Die weitere API wird dadurch vereinfacht, da in den Schnittstellen nicht mehr zwischen verschiedenen Medien unterschieden werden muss, stattdessen wird `IMedium` verwendet. Begründung für die einzelnen Eigenschaften siehe oben.

Nachteile: Keine erkennbar

Aus Konsistenzgründen gibt es allerdings bestimmte Einschränkungen was das Setzen der Eigenschaften eines Mediums angeht:

***DES 067:* Ist eine `IMedium`-Implementierung schreibbar, dann muss sie auch random-access sein**
Jede (prinzipiell) schreibbare `IMedium`-Implementierung muss auch random-access sein.

Begründung: Die `jMeta`-APIs zum Schreiben von Inhalten können sich somit auf random-access-Ausgabe-Medien konzentrieren. Es ist kein separates Design einer API und Implementierung für sowohl random-access- als auch nicht-random-access-Ausgabemedien notwendig. Die API wird einfacher für den Anwender. Der Verzicht auf nicht-random-access-Ausgabemedien wie `OutputStream`s kann wie in dem in DES 067 angegebenen Beispiel ausgeglichen werden.

Nachteile: Keine erkennbar

Hier noch eine wichtige Notiz für byte-Array-Medien:

***DES 068:* Für byte-Array-Medien wird eine schreibende Methode zum Neu-Setzen des Byte-Arrays benötigt**
Man kann über die öffentliche Methode `setBytes()` der Klasse `InMemoryMedium` die Bytes des Mediums neu setzen.

Begründung: Es ist größtenteils unkritisch, die Methode öffentlich zu machen, stattdessen ist es sogar ein Vorteil für den Anweder, die Bytes auch selbst neu setzen zu können. Einzig zwischen Anmelden von Schreiboperationen und einem flush führt der Aufruf der Methode zu unerwartetem Verhalten.
Wichtig ist die Methode für die Implementierung: Durch Schreibaktionen muss das byte-Array in manchen Situationen nach hinten vergrößert oder auch verkleinert werden. Dies entspricht einem Neu-Anlegen und Kopieren des Arrays. Die Methode muss deswegen auch öffentlich sein, da die entsprechende Implementierungsfunktionalität sich in einem anderen Package befindet.

Nachteile: Keine bekannten Nachteile

Positionen in einem Medium

Immer dann, wenn Daten aus einem MEDIUM gelesen oder in das MEDIUM geschrieben werden sollen, stellt sich die Frage des "Wo?". Üblicherweise nutzen Libraries dann Integer- oder Longangaben als Offsets. Es sei allerdings noch gesagt: Offsets machen nicht nur bei random-access-Medien Sinn. Man kann sie auch als Offset seit Start des Lesens eines `InputStream`s interpretieren, was in `jMeta` auch der Fall ist. Wir legen insgesamt grundlegend fest:

***DES 069:* Byte-Offsets werden für alle Arten von MEDIEN verwendet**
Byte-Offsets, die sich auf eine Position in einem MEDIUM beziehen, werden für alle Medien, random-access und nicht-random-access, verwendet, um die Position eines Datenbytes anzugeben. Bei byte streams beziehen sie sich auf die Position des aktuellen Bytes seit dem Lesen des ersten Bytes, das Offset 0 hat. Das offset-basierte Lesen wird wie in DES 069 und DES 069 dargestellt simuliert, denn beim direkten Lesen aus einem `InputStream` via Java API spielen Offets keine Rolle, es wird immer von der aktuellen Position gelesen.

Begründung: Wir müssen somit in der Implementierung nur an wenigen Stellen zwischen random-access und nicht-random-access unterscheiden. Anwender können die API einheitlich und unabhängig vom tatsächlichen Medium (mit Einschränkungen: siehe DES 069) nutzen.

Nachteile: Keine erkennbar

Es ist nur bedingt sinnvoll, Offsets gestaltlos als primitive Datentypen zu belassen. Stattdessen bietet sich die Repräsentation als benutzerdefinierten Datentyp an:

***DES 070:* `jMeta` verwendet das Interface `IMediumReference` zur Bezugnahme auf Offsets in einem MEDIUM.**
Das Interface vereint das MEDIUM und den Offset im MEDIUM miteinander, und stellt daher eine Art "globale" Adresse eines Datensegments dar. Neben dem Abfragen des Mediums und des Offsets gibt es einige Hilfsmethoden in dieser Klasse:

- `behindOrEqual()`: Liefert true, falls eine andere `IMediumReference` sich auf dem selben Medium hinter oder an der gleichen Position wie die aktuelle Instanz befindet.
- `before()`: Liefert true, falls eine andere `IMediumReference` sich auf dem selben Medium vor der gleichen Position wie die aktuelle Instanz befindet.
- `advance()`: Erzeugt eine neue `IMediumReference`, die sich um die angebene Anzahl an Bytes vor (negativ) oder hinter (positiv) der aktuellen Instanz befindet.

Begründung: Wir legen klar fest, wie mit Offsets umgegangen wird. Implementierung kann z.B. ein interner Datentyp sein. Zudem können wir in den Datentyp einige Hilfsfunktionen (z.B. Validierung, Offsetvergleiche, weitersetzen etc.) einbauen, die die Implementierung und auch die Verwendung an vielen Stellen erleichtern.

Nachteile: Keine erkennbar

Kommen wir nun noch zu einer zentralen Entscheidung für den Umgang mit

Datenlängen und Offsets:

DES 071: **`jMeta` verwendet long für Längen- und Offset-Angaben, Einheit ist immer Bytes**
In `jMeta` werden Längen- und Offset-Angaben grundsätzlich mittels des Java-Datentyps long repräsentiert. Die Länge wird in allen Fällen immer in Bytes angegeben, Offsets sind nullbasierte Byte-Offsets.

Begründung: Diese Festlegung garantiert zunächst Einheitlichkeit. Wir möchten aber auch Anforderung "4.8 ANF 008: Lesen und Schreiben großer Datenblöcke" umsetzen können. Integer mit maximal nur 4,3 GB ist bereits allzu begrenzt, bleibt also nur noch long. Der Datentyp long ermöglicht positive Zahlen bis $2^{63} - 1 = 9223372036854775807$, also rund $9 \cdot 10^{18}$ Bytes, das sind 9 Exabytes oder 9 Milliarden Gigabyte. Aus aktueller Sicht werden wir solche Längen und Offsets bei Eingabemedien, selbst bei Streams, noch nicht allzu bald erreichen. Zudem werden große Datenmengen so gut wie immer in kleine Einheiten unterteilt, die wiederum keine exorbitanten Längen aufweisen. Auch die Java File I/O verwendet vielerorts long für Offset- und Längenangaben.

Nachteile: Mehr Speicherplatz für die Speicherung von Offsets und Längen. Bei der Entwicklung von Speichermedien, Speicherbedarf und Rechengeschwindigkeit kann es durchaus sein, dass die genannte maximale Datenmenge doch in 100 Jahren vereinzelt erreicht wird. Wenn es `jMeta` dann noch im Einsatz geben sollte, wäre dies einen change request wert. Aber natürlich ist klar, dass in einem solchen Fall noch viele weitere der aktuellen Basiskonzepte und -frameworks, so auch die Java-Standardlibrary, nicht mehr tragfähig sind.

Einen eher seltenen Spezialfall behandelt folgende Designentscheidung:

DES 072: **Keine spezielle Behandlung für den Fall eines long-Überlaufs**
Bei außerordentlich langem ununterbrochenen Lesen aus einem `InputStream` könnte man meinen, dass es auch trotz Verwendung von long einmal zu einem Überlauf kommen könnte. Dies ist jedoch unwahrscheinlich, daher wird dieser Fall nicht behandelt, es wird immer davon ausgegangen, dass das aktuelle Offset positiv ist und inkrementiert werden kann, ohne die Grenze des Datentyps long zu erreichen.

Begründung: Auch hier gilt nämlich: Der Datentyp long ermöglicht positive Zahlen bis $2^{63} - 1 = 9223372036854775807$, also rund $9 \cdot 10^{18}$ Bytes, das sind 9 Exabytes oder 9 Milliarden Gigabyte. Nehmen wir also an, eine Implementierung könnte es schaffen, 10 GB pro Sekunde zu lesen, dann bräuchte es dennoch 9 Milliarden Sekunden, das sind umgerechnet fast 30 Jahre, um die Offset-Grenze zu erreichen und einen Überlauf zu erzeugen.

Nachteile: Keine erkennbaren

Nun ergibt sich die Problematik, dass das Medium durch schreibende Zugriffe geändert wird. Wie ändern sich dann die Offsets? Muss man bereits erzeugte `IMediumReference`-Instanzen gemäß den Änderungen des Mediums aktualisieren, oder nicht? Wenn ja, wann muss dies geschehen? Prinzipiell gäbe es folgende Varianten:

1. Aktualisiere bereits erzeugte `IMediumReference`-Instanzen nie

2. Aktualisiere bereits erzeugte `IMediumReference`-Instanzen bereits direkt bei jeder schwebenden Änderung (siehe DES 072)

3. Aktualisiere bereits erzeugte `IMediumReference`-Instanzen erst bei einem expliziten *flush* gemäß DES 072

Angenommen, `IMediumReference`-Instanzen werden nicht aktualisiert. D.h. also der anwendende Code merkt sich z.B. selbst eine Position eines Elementes via `IMediumReference`-Instanz, und verwendet diese, um Daten zu lesen oder zu schreiben. Findet beispielsweise eine Einfügeoperation auf dem Medium vor dem Offset der `IMediumReference`-Instanz statt, dann verweist diese nun auf andere Daten als zuvor, und somit nicht mehr auf die vorherigen Daten. Denken wir uns nicht nur Roh-Bytes, sondern - wie bei Datenformaten notwendig - *Objekte*, also Teile der Binärdaten mit einer bestimmten Bedeutung, die etwas repräsentieren. Dann ist das Fehlen einer Anpassung des Offsets fatal. Der Code, der `Media` verwendet, verortet ein Objekt dann ggf. an einer falschen Position.

Damit steht fest:

DES 073: **`Media` muss `IMediumReference`-Instanzen nach Änderungen am Medium automatisch aktualisieren**
Alle jemals für ein Medium erzeugten `IMediumReference`-Instanzen müssen bei Veränderungen des Mediums automatisch aktualisiert werden. Die Art der Aktualisierung ist komplizierter als man zunächst denkt. Um dies zu vereinfachen, wird der oben diffus angedeutete Begriff eines "Objektes", auf das sich ein Offset bezieht, so definiert: Ein Objekt beginnt an einem bestimmten Offset x und hat eine Länge von n Bytes. Löschungen und Einfügungen im Offset-Intervall $[x, x+n]$ ändern das Objekt, was aber von `Media` in keiner Form speziell behandelt werden kann.
Sei nun y das Einfüge- bzw. Löschoffset und k Anzahl der einzufügenden bzw. zu löschenden Bytes. Sei $\overline{x}$ das neue Offset der `IMediumReference`-Instanz nach Aktualisierung. Dann gelten im Detail folgende Regeln:

- *Einfügen vor der Instanz:* Ist $y \leq x$, dann ist $\overline{x} := x + k$. Dies schließt also den Fall ein, dass Bytes genau am Offset x eingefügt werden.
- *Einfügen hinter der Instanz:* Ist $y > x$, dann ist $\overline{x} := x$, also ändert sich nichts an der Instanz, natürlich auch dann nicht, wenn $y < x + n$ ist. Wie dies semantisch das "Objekt" ändert, liegt in der Hand des Anwenders.
- *Löschen vor der Instanz ohne Überlappung:* Ist $y + k \leq x$, dann ist $\overline{x} := x - k$. Hier werden also k Bytes vor der Instanz gelöscht, jedoch existiert keine Überlappung mit dem Objekt, auf das die Instanz verweist.
- *Löschen vor der Instanz mit Überlappung:* Ist $y \leq x$, jedoch $y + k > x$, dann handelt es sich um eine Überlappung des Löschbereichs mit dem Objekt. Inhaltlich kann man also von einer "Verkürzung" des Objektes (unter Umständen auf Länge 0) von vorne sprechen. Der Beginn des Objektes rutscht damit um $x - y$ Bytes nach vorne auf y, damit gilt $\overline{x} := y$.
- *Löschen hinter der Instanz:* Ist $y > x$, dann ist $\overline{x} := x$, also ändert sich nichts an der Instanz, natürlich auch dann nicht, wenn $y < x + n$ ist. Wie dies semantisch das "Objekt" ändert, liegt in der Hand des Anwenders.

Begründung: Die Verbindung zwischen einer `IMediumReference`-Instanz und einem "Objekt" aus Sicht des Anwenders bleibt auch bei schreibenden Änderungen des Mediums erhalten. Man kann das Ändern der Offsets nicht dem Anwender aufbürden, weil dieser sonst wiederum detailliert buchführen müsste, welche Aktionen er durchgeführt hat. Die Umsetzung dieser komplexen Funktionalität würde man aber gerade von einer Komponente `Media` erwarten. Werden alle Bytes des Objektes gelöscht, dann verweist die `IMediumReference`-Instanz immerhin noch auf den "ehemaligen" Ort des Objektes. Der Anwender muss nicht sicherstellen, dass sich seine `IMediumReference`-Instanz noch immer auf das korrekte Objekt bezieht, weder durch Buchführung noch durch nochmaliges "Parsen".

Nachteile: Es ist eine zentrale Verwaltung der `IMediumReference`-Instanzen nötig (siehe DES 073).

Wie bereits in den Nachteilen von DES 073 angedeutet, impliziert die automatische Aktualisierung aller jemals erzeugten `IMediumReference`-Instanzen direkt, dass nur `Media` `IMediumReference`-Instanzen erzeugen darf. Diese müssen in einem Pool verwaltet werden, um sie bei schreibenden Operationen automatisch aktualisieren zu können.

DES 074: **`IMediumReference`-Instanzen werden zentral von `Media` verwaltet und können nicht von Anwendern direkt erzeugt werden**
Der Lebenszyklus von `IMediumReference`-Instanzen wird von `Media` gesteuert, sie werden über eine andere Klasse per factory-Methode erzeugt und herausgegeben.

Begründung: Dies ist zwingend notwendig, um DES 074 umsetzen zu können. Instanzen, die von Anwendern erzeugt worden sind, können nicht aktualisiert werden, damit muss deren "manuelle" Erzeugung durch den Anwender unterbunden werden.

Nachteile: Komplexere Instanzierung von `IMediumReference`-Instanzen.

Die Frage, *wann* die `IMediumReference`-Instanzen aktualisiert werden sollen, ist weiterhin ungeklärt. Die folgende Designentscheidung legt dies klar fest:

***DES 075:* `IMediumReference`-Instanzen werden erst nach einem *flush* aktualisiert**
Gemäß DES 075 werden `IMediumReference`-Instanzen automatisch bei Änderungen des Mediums aktualisiert. Diese automatische Aktualisierung erfolgt erst beim *flush*.

Begründung: Angenommen, die `IMediumReference`-Instanzen würden bereits bei jeder einzeln geschriebenen schwebenden Änderung (insert, replace, remove) angepasst werden. Dann wäre folgendes notwendig:

- Beim Lesen von Daten befinden sich diese nicht notwendigerweise im Cache. Damit ist unter Umständen ein Lesen vom externen Medium nötig. Da alle `IMediumReference`-Instanzen nach einer schwebend geschriebenen Schreiboperation bereits den neuen Stand widerspiegeln, der jedoch noch nicht dem externen Medium entspricht, müsste anhand der bisher erfolgten Änderungen der tatsächliche Offset der Daten auf dem Medium berechnet oder rekonstruiert werden.
- Die Operation `undo` gemäß DES 075 erfordert, dass die Offsets wieder "rück-angepasst" werden, wenn eine Operation rückgängig gemacht wird.

Werden `IMediumReference`-Instanzen hingegen erst nach einem *flush* aktualisiert, dann wird beim Lesen direkt auf den Cache oder das externe Medium zugegriffen, die `IMediumReference`-Instanzen müssen dafür nicht umgerechnet werden.

Nachteile: Keine bekannten Nachteile

Daraus ergibt sich direkt die folgende Designentscheidung:

***DES 076:* Offsetangaben beim Arbeiten mit `Media` beziehen sich auf Offsets nach letztem *flush* bzw. Öffnen**
Für Operationen von `Media`, die jeweils einen Offset übernehmen, an dem die Operation ausgeführt werden soll, beziehen sich die übergebenen Offsets auf Positionen auf dem MEDIUM nach dem letzten *flush* bzw. dem Öffnen - falls noch kein *flush* erfolgt ist. Insbesondere müssen die angegebenen Offsets innerhalb des Intervalls $[0, \text{length}]$ liegen, wobei "length" die aktuelle Länge des Mediums ist.

Begründung: Ergibt sich notwendigerweise aus DES 076. Für den Anwender bleibt die Offset-Situation damit stabil, er muss nicht darauf achten, welche Einfüge- oder Löschoperationen welcher Länge bereits erfolgt sind. Genauso bleibt die Offset-Situation für die Implementierung selbst stabil: Prüfungen von Offsets und die Organisiation von internen Datenstrukturen kann sich auf diese Invariante verlassen.

Nachteile: Keine erkennbar

Semantik der schreibenden Operationen

In DES 076 haben wir die primitiven schreibenden Operationen *insert*, *replace* und *remove* definiert, die für `Media` notwendig sind. Zur API von `Media` gehört dann aber auch deren garantiertes Verhalten. Wichtig ist insbesondere der Umgang mit Wechselwirkungen zwischen diesen Operationen, das sind Aufrufe, die sich auf dieselben oder überlappende Offset-Bereiche des Mediums beziehen. Diese klären die folgenden Designentscheidungen. Es handelt sich um Bestandteile des Schnittstellenvertrages, d.h. dieses Verhalten gehört zur API und ist entsprechend für die Anwender der Operationen so zu dokumentieren.

Die folgenden Designentscheidungen haben ein Prinzip gemeinsam, das hier herausgearbeitet wird:

***DES 077:* Die zeitlich neuesten Aufrufe machen bei Überlappungen zeitlich vorangehende Aufrufe ungültig bzw. passen diese an**
Erfolgen mehrfache Aufrufe der Operationen *insert*, *replace* oder *remove*, so passen zeitlich spätere Aufrufe die Auswirkungen zeitlich früherer Aufrufe an bzw. machen diese ungültig und damit quasi rückgängig, sofern sich die Wirkbereiche der Operationen überlappen oder überdecken.

Begründung: Zeitlich spätere Aufrufe sind als mit "aktuellstem Wissensstand erfolgt" anzusehen. Es sollen keine widersprüchlichen Aktionsabläufe erstellt werden können. Diese führen bei naiver Implementierung ggf. zu unerwünschten Inkonsistenzen und damit spätrer zu behandelnden Fehlersituationen oder auch unnötigen oder gar fehlerhaften Löschungen und Ersetzungen.

Nachteile: Keine erkennbar

Wir beginnen mit dem Einfügen von Daten über die Operation *insert*:

***DES 078: insert* konkateniert Einfügungen in Aufrufreihenfolge, die Operation wird von anderen schreibenden Aufrufen nicht beeinflusst**
Die folgenden zeitlichen Aufrufreihenfolgen haben die beschriebene Semantik:

- Schritt 1: *insert*, Schritt 2: *insert* am gleichen Offset: *insert* ist konkatenierend, jeder Aufruf mit demselben Offset bezeichnet ein neues, nachgelagertes Einfügen, d.h. Einfügungen an dieselbe Stelle im Medium werden auch mit aufsteigenden Offsets durchgeführt. Um dies unmissverständlich darzulegen, nehmen wir zwei Aufrufe von *insert* mit gleichem Offset x an. "Aufruf 1" sei der zeitlich erste mit Datenlänge n_1, "Aufruf 2" der zeitlich darauffolgende Aufruf für Offste x mit Datenlänge n_2. Das Endergergebnis auf dem Medium nach *flush*:
  - Ab Offset x befinden sich die Daten von "Aufruf 1"
  - Ab Offset $x + n_2$ befinden sich die Daten von "Aufruf 2"
- Schritt 1: *insert*, Schritt 2: *remove* am gleichen Offset: Da sich die Offsets gemäß DES 078 auf Offsets des externen Mediums beziehen, betrifft das *remove* nur Bytes auf dem externen Medium, kann also keinerlei Inhalte löschen, die vorher mit insert (ohne zwischenzeitliches *flush*) eingefügt worden sind. Sprich: Die Operationen beeinflussen sich nicht.
- Schritt 1: *insert*, Schritt 2: *replace* am gleichen Offset: Die Operationen beeinflussen sich nicht, mit gleicher Begründung wie bei *remove*.

Begründung: Zum konkatenierten Einfügen: Man könnte "insert" auch so interpretieren, dass der zweite Aufruf die Daten vor dem ersten einfügen müsste. Stattdessen heißt "insert" jedoch: Einfügen vor das jeweilige aktuell auf dem Medium persistierte Byte. Die Semantik des Konkatenierens ist die logischste, weil der Anwendercode dann Einfügungen linear mit wachsenden Offsets vornehmen kann, was in den meisten Fällen das übliche Vorgehen ist.

Nachteile: Keine erkennbar

Die Operation *remove* verhält sich folgendermaßen:

***DES 079:* *remove* erlaubt Überlappungen, die jedoch korrigiert werden, Mehrfachaufrufe mit gleichen Eingabeparametern werden ignoriert**
Die folgenden zeitlichen Aufrufreihenfolgen haben die beschriebene Semantik:

- Schritt 1: *remove* n Bytes ab Offset x, Schritt 2: *remove* oder *replace* n Bytes ab Offset x: Bei zweimaligem, exakt identischen Aufruf von *remove* bzw. Aufruf von *remove* und danach *replace* für den gleichen Offsetbereich wird der zeitlich erste Aufruf für ungültig erklärt und bei *flush* nicht verarbeitet.

- Schritt 1: *remove* n Bytes ab Offset x, Schritt 2: *remove* m Bytes ab Offset $y \in (x, x+n)$ mit $y+m < x+n$: Es handelt sich also beim zweiten Aufruf um einen Aufruf, bei dem das Löschintervall vollständig im Inneren des Löschintervalls des ersten Aufrufs liegt. Da die zweite Löschung vollständig in der ersten enthalten ist, bleibt die erste bestehen, und die zweite Löschung wird ignoriert. Dies ist insofern kein Widerspruch zu DES 079, als es sich hier um keine echte Überlappung oder Überdeckung des ersten Aufrufs durch den zweiten handelt, sondern vielmehr um eine Überdeckung des zweiten Aufrufs durch den ersten.

- Schritt 1: *remove* n Bytes ab Offset x, Schritt 2: *replace* m Bytes ab Offset $y \in (x, x+n)$ mit $y+m < x+n$: Es handelt sich also beim zweiten Aufruf um einen Aufruf, bei dem das Ersetzungsintervall vollständig im Inneren des Löschintervalls des ersten Aufrufs liegt. Leider lässt sich dieser Fall nicht wie der vorherige behandeln. Die nachfolgende Ersetzung muss die vorherige Löschung teilweise rückgängig machen: Das Löschintervall des ersten Aufrufs wird dann zweigeteilt: Teil 1 ist eine Löschung des Intervalls $[x, y)$, Teil 2 ist eine Löschung des Intervalls $[y+m, x+n)$. Die Ersetzung wird ebenfalls wie angegeben durchgeführt für das Intervall $(y, y+m)$

- Schritt 1: *remove* n Bytes ab Offset x, Schritt 2: *remove* oder *replace* m Bytes ab Offset $y \in (x, x+n)$ mit $y+m \geq x+n$: Es handelt sich also beim zweiten Aufruf um einen überlappenden Aufruf, der ggf. noch weitere Bytes hinter dem ersten Löschaufruf entfernt, aber doch mindestens das Ende des ersten Löschaufrufs mit einschließt. Dann wird das Löschintervall des ersten Aufrufs auf $[x, y)$ verkürzt mit der neuen Löschlänge $\overline{n} := y - x$.

- Schritt 1: *remove* n Bytes ab Offset x, Schritt 2: *remove* oder *replace* m Bytes ab Offset $y < x$ mit $y+m < x+n$: In diesem Fall findet durch den zweiten Aufruf eine vollständige "Überdeckung" von Vorne statt, ohne dass der Löschbereich des ersten Aufrufs vollständig überlagert wird. Auch hier wird der zeitlich erste Aufruf angepasst: Das Löschintervall des ersten Aufrufs wird auf $[y+m, x+n)$ verkürzt mit der neuen Löschlänge $\overline{n} := x + n - y - m$.

- Schritt 1: *remove* n Bytes ab Offset x, Schritt 2: *remove* oder *replace* m Bytes ab Offset $y < x$ mit $y+m \geq x+n$: In diesem Fall findet durch den zweiten Aufruf eine "Überlagerung" von Vorne statt. Analog zum Aufruf mit identischen Parametern wird der zeitlich erste Aufruf für ungültig erklärt und bei *flush* nicht verarbeitet.

- Schritt 1: *remove* n Bytes ab Offset x, Schritt 2: *insert* ab Offset $y \in [x, x+n)$: Da sich die Aufrufe gemäß DES 079 auf Offsets des externen Mediums beziehen, werden hier zuerst n bereits auf dem Medium vorhandene Bytes gelöscht, dann direkt nach dem letzten vor Offset x noch

Die Operation *replace* verhält sich folgendermaßen:

***DES 080: replace* erlaubt Überlappungen, die jedoch korrigiert werden, Mehrfachaufrufe mit gleichen Eingabeparametern werden ignoriert**
Die folgenden zeitlichen Aufrufreihenfolgen haben die beschriebene Semantik:

- Schritt 1: *replace* n Bytes ab Offset x, Schritt 2: *replace* oder *remove* n Bytes ab Offset x: Bei zweimaligem, exakt identischen Aufruf von *replace* bzw. Aufruf von *replace* und danach *remove* für den gleichen Offsetbereich wird der zeitlich erste Aufruf für ungültig erklärt und bei *flush* nicht verarbeitet. Auch alle Ersetzungsbytes des ersten Aufrufs werden damit ignoriert.
- Schritt 1: *replace* n Bytes ab Offset x, Schritt 2: *replace* oder *remove* m Bytes ab Offset $y \in [x, x+n)$: Bei einem nur überlappenden Aufruf wird der zeitlich erste Aufruf angepasst: Das Ersetzungsintervall des ersten Aufrufs wird auf $[x, y)$ verkürzt mit der neuen Länge $\overline{n} := y - x$. Die nachgelagerten Ersetzungsbytes des ersten Aufrufs von *replace* werden ignoriert.
- Schritt 1: *replace* n Bytes ab Offset x, Schritt 2: *replace* oder *remove* m Bytes ab Offset $y < x$ mit $y + m < x + n$: In diesem Fall findet durch den zweiten Aufruf eine "Überlappung" von Vorne statt, ohne dass der Ersetzungsbereich des ersten Aufrufs vollständig überlagert wird. Auch hier wird der zeitlich erste Aufruf angepasst: Das Ersetzungsintervall des ersten Aufrufs wird auf $[y + m, x + n)$ verkürzt mit der neuen Länge $\overline{n} := x + n - y - m$. Die vorgelagerten Ersetzungsbytes des ersten Aufrufs von *replace* werden ignoriert.
- Schritt 1: *replace* n Bytes ab Offset x, Schritt 2: *replace* oder *remove* m Bytes ab Offset $y < x$ mit $y + m \geq x + n$: In diesem Fall findet durch den zweiten Aufruf eine vollständige "Überlagerung" von Vorne statt. Analog zum Aufruf mit identischen Parametern wird der zeitlich erste Aufruf für ungültig erklärt und bei *flush* nicht verarbeitet. Auch alle Ersetzungsbytes des ersten Aufrufs werden damit ignoriert.
- Schritt 1: *replace* n Bytes ab Offset x, Schritt 2: *insert* ab Offset $y \in [x, x+n]$: Da sich die Aufrufe gemäß DES 080 auf Offsets des externen Mediums beziehen, werden hier zuerst die n Bytes des Mediums überschrieben, dann direkt am angegebenen Offset die neuen Bytes eingefügt.

Begründung: Es werden keine (versehentlichen) Mehrfachlöschungen oder Überschreibungen durchgeführt.

Nachteile: Keine erkennbar

Das Problem: Wie ordnet man nun Datenblöcken ihre tatsächlichen Daten vor einem *flush* zu? Das Offset ist bei Mehrfacheinfügungen via *insert* vor einem *flush* nicht mehr eindeutig. Will man nun den Bezug zwischen einem neuen Datenobjekt und seinen Daten herstellen, reicht das Offset nicht aus.

Die Antwort wird in folgender Designentscheidung gegeben:

***DES 081:* Jede schreibende Operation liefert Informationen zu den Daten als Instanz der Klasse `MediumAction` zurück**
Diese Klasse enthält folgende Daten:

- Art der Änderung (*insert*, *replace*, *remove*); **Begründung:** Anwender muss die Änderung einordnen können
- `IMediumReference` der Änderung; **Begründung:** Es muss klar sein, wo etwas geändert werden soll
- Daten der Änderung (Länge bzw. zu schreibende Daten); **Begründung:** Es muss klar sein, was geändert werden soll.
- Gültigkeit: Handle ist bereits durch *flush* festgeschrieben oder noch schwebend; **Begründung:** Damit kann von Außen festgestellt werden, ob die Daten in der Instanz noch nicht festgeschrieben (schwebend) sind und daher der `MediumAction` entnommen werden müssen, oder ob die Daten bereits auf das Medium geschrieben worden sind, und damit von diesem gelesen werden können.

Begründung: Gemäß DES 081 kann der Anwender durch lesenden Zugriff via `Media` nur den aktuell festgeschriebenden Stand des Mediums und damit keine schwebenden Änderungen lesen.
Dennoch ist es notwendig, dass der Anwendungscode auch vor einem *flush* die bisher nur schwebend geschriebenden Daten von *insert*s oder *replace*s erneut auslesen kann. Dies ist über eine `MediumAction`-Instanz möglich, die der Anwender durch einen Aufruf von *insert* oder *replace* als Rückgabe erhält. Ist die `MediumAction`-Instanz noch schwebend, liefert der Anwendungscode die schwebenden Bytes, sonst durch Zugriff auf `Media` die persistierten Bytes. Handelt es sich um ein *remove* oder wurden für einen Datenblock keine Änderungen vorgenommen, dann kann der anwendende Code Daten wie gewohnt von `Media` lesen.
Instanzen von `MediumAction` können zudem zur optimalen internen Datenverwaltung verwendet werden.

Nachteile: Keine erkennbar

Medium-Zugriff beenden

Wir benötigen noch eine Operation zum Beenden des Mediumzugriffs. Daher definieren wir:

***DES 082:* Operation *close* beendet den Medienzugriff und leert den Cache, nachfolgende Operationen sind nicht möglich**
Eine vom Anwender manuell aufzurufende Operation *close* beendet den Zugriff auf eine Medium. Es sind keinerlei nachfolgende Operationen auf dem Medium über den geschlossenen Zugriffsweg mehr möglich. Der Anwender muss explizit das Medium erneut öffnen.

Begründung: Die Implementierung arbeitet (teilweise) mit Betriebssystemressourcen wie Dateien, die geschlossen werden müssen. Zudem werden Cache-Inhalte und andere Speicherinhalte so unter Umständen nie freigegeben.
Das Schließen kann nicht automatisch erfolgen, sondern der Anwender muss dies explizit steuern.

Nachteile: Keine bekannten Nachteile

Die öffentliche API des Medium-Zugriffs

Aus den vorangehenden Abschnitten und Designentscheidungen entwerfen wir nun die öffentliche Schnittstelle der Komponente `Media`. Bisher wurden nur die Klassen `IMediumReference`, `IMedium` und `MediumAction` eingeführt, sowie einige Operationen für den Umgang mit Medien genannt. Wie werden die Operationen zum Zugriff auf ein Medium öffentlich angeboten? Das klärt die folgende Designentscheidung.

***DES 083:* Zugriff auf ein Medium über ein Interface `IMediumStore`**
Der lesende und schreibende Zugriff auf ein Medium (sowohl random-access als auch byte stream gemäß DES 083) erfolgt über ein Interface `IMediumStore` mit den in Tabelle 11.2 aufgelisteten Operationen.

Begründung: Eine weitere Zerlegung der Funktionalität in mehrere Interfaces ist weder notwendig noch sinnvoll. Ein Zerlegung würde dem Anwender eine unnötig komplexe Schnittstelle ohne Mehrwert bieten. Die Implementierung kann dank der Verwendung eines Interfaces dennoch beliebig nach Notwendigkeit modularisiert werden. Begründungen für die einzelnen Operationen siehe Tabelle.

Nachteile: Keine erkennbar

Operation	**Beschreibung**	**Funktion und Begründung random-access**	**Funktion und Begründung `InputStream`**
`cache()`	Puffert n Bytes gemäß DES 083 und DES 083 dauerhaft in den internen Cache, startend vom angegebenen Offset x. Hat keine Wirkung, wenn Caching deaktiviert ist.	Der Anwendungscode kann Daten vorladen, dann aber erst später verarbeiten, es wird ihm also selbst die Möglichkeit gegeben, Daten effizient zu lesen.	Ist der Offset x kleiner dem aktuell höchsten Lese-Offset, folgt eine `InvalidMediumReferenceException`. Zusätzlich gilt hier: Ist er hingegen größer als der gemerkte höchste Lese-Offset, dann werden auch die Differenzbytes bis dorthin gelesen oder (je nach Konfiguration) auch durch `skip()` übersprungen. Danach werden die eigentlichen Bytes gelesen. Dies stellt sicher, dass alle Bytes prinzipiell verfügbar sind. Der letzte Lese-Offset wird entsprechend weitergesetzt.
`getCachedByteCountAt()`	Liefert gemäß DES 083 und DES 083 die Anzahl der am Offset x im internen Cache lückenlos vorhandenen Bytes zurück.	Siehe `cache()`. Der Anwendungscode muss zusätzlich prüfen können, ob bereits genügend Daten vorgeladen worden sind.	Siehe bei random-access.

Operation	**Beschreibung**	**Funktion und Begründung random-access**	**Funktion und Begründung `InputStream`**
`getData()`	Liest n Bytes ab Offset x mit dem angegebenen Modus (`forceMediumAccess` = `true` oder `false`) gemäß DES 083, DES 083, DES 083	Bei `forceMediumAccess` = `false`: Es wird zunächst versucht, die Daten aus dem Cache zu lesen, sind diese (teilweise) nicht vorhanden, werden die fehlenden Daten direkt vom externen Medium gelesen. Bei `forceMediumAccess` = `true`: Die Daten werden immer direkt vom externen Medium gelesen, der Cache wird ignoriert. In beiden Fällen wird der Cache jedoch durch neu vom externen Medium gelesene Daten aktualisiert. Zudem wird in beiden Fällen nur auf den Cache zugegriffen, wenn Caching aktiviert ist.	`forceMediumAccess` wird ignoriert. Sind Daten für das angegebene Offset im Cache vorhanden, so werden sie zurückgeliefert. Ist dies nicht der Fall, folgt eine `InvalidMediumReferenceException`. Für `InputStream`s ist daher ein Aufruf von `cache()` unerlässlich, bevor wirklich gelesen werden kann.
`isAtEndOfMedium()`	Prüft, ob Offset x das Ende des MEDIUMS darstellt.	Auf Basis dieser Erkenntnis kann der anwendende Code das weitere Lesen abbrechen und muss nicht auf eine Exception oder andere Signale achten.	Das übergebene Offset wird ignoriert, und es wird versucht, an der aktuellen Stelle ein Byte zu lesen. Resultiert -1, so sind wir am Ende des Streams, ansonsten wird das Byte in den Cache übernommen.

Operation	**Beschreibung**	**Funktion und Begründung random-access**	**Funktion und Begründung `InputStream`**
`skip()`	Überspringt n Bytes gemäß DES 083	Hat keine Funktion, tut einfach nichts, weil Skippen hier keine Vorteile bringt.	Verwirft n Bytes durch Aufruf von `InputStream.skip(n)`. Somit ermöglicht dies Anwendern, unwichtige Bytes bewusst zu überspringen, um beispielsweise bei Aktiviertem Caching Speicherplatz zu sparen. Der letzte Lese-Offset wird entsprechend weitergesetzt.
`discard()`	Löscht bis zu n Bytes im Cache ab Offset x gemäß DES 083. Ermöglicht dem anwendenden Code, ggf. vorhandene Speicherinhalte freizugeben, um unnötig allokierten Heap-Speicher freizumachen.	Erneutes Puffern und Lesen der Daten mit `getData()` ist jederzeit möglich.	Nach dem Freigeben von Daten mit `discard()` führt der Versuch, bei einem `InputStream` mittels `getData()` auf bereits freigegebene Offset-Bereiche zuzugreifen, zu einer `InvalidMediumReferenceException`.
`insertData()`	Setzt die schreibende Operation *insert* gemäß DES 083, DES 083 und DES 083 um: Fügt Daten an einer angegebenen Stelle ein, darauffolgende Daten werden dadurch "nach hinten" geschoben, die Änderungen werden erst mit `flush()` geschrieben.	Das Einfügen neuer Metadaten ist ein Standardfall in `jMeta` und muss unterstützt werden.	`ReadOnlyMediumException`

Operation	**Beschreibung**	**Funktion und Begründung random-access**	**Funktion und Begründung `InputStream`**
`removeData()`	Setzt die schreibende Operation *remove* gemäß DES 083, DES 083 und DES 083 um: Entfernt n Bytes ab Offset x. Darauffolgende Daten werden dadurch "nach vorne" geschoben, die Änderungen werden erst mit `flush()` geschrieben	Das Löschen vorhandener Metadaten ist ein Standardfall in `jMeta` und muss unterstützt werden.	`ReadOnlyMediumException`
`replaceData()`	Setzt die schreibende Operation *replace* gemäß DES 083, DES 083 und DES 083 um: Entfernt n Bytes ab Offset x durch neue Bytes der Länge n. Die Änderungen werden erst mit `flush()` geschrieben	Es kommt häufig vor, dass statt komplettem Entfernen oder komplett neuem Einfügen ein Überschreiben existierender Daten erfolgen muss. Dies ist - insbesondere am Beginn einer Datei - eine sehr viel effizientere Operation als Einfügen und Löschen und muss daher direkt unterstützt werden.	`ReadOnlyMediumException`
`flush()`	Setzt die schreibende Operation *flush* gemäß DES 083 um: Alle aktuell im Zwischenspeicher vorhandenen *geänderten* Daten werden in geeigneter Form tatsächlich auf das externe Medium geschrieben. Es kann nicht garantiert werden, dass diese Operation atomar (ganz oder gar nicht) erfolgt.	Dies ist die praktische Umsetzung von DES 083: Während `insertData()`, `removeData()` und `replaceData()` nur in den Zwischenspeicher schreiben, wird über diesen Aufruf explizit auf das externe Medium geschrieben.	`ReadOnlyMediumException`

Operation	**Beschreibung**	**Funktion und Begründung random-access**	**Funktion und Begründung `InputStream`**
`createMediumReference()`	Erzeugt eine `IMediumReference`-Instanz für das gegebene Offset x gemäß DES 083	Wird für random-access benötigt	Wird für `InputStream`s benötigt
`undo()`	Macht die Änderung der gegebenen `MediumAction` gemäß DES 083 rückgängig, sofern die Änderung noch schwebend ist.	Notwendig für random-access	`InvalidMediumActionException`
`close()`	Schließt alle intern gehaltenen Ressourcen gemäß DES 083, leert den kompletten Cache-Inhalt und weitere interne Datenstrukturen	Siehe DES 083	Siehe DES 083

Table 11.2.: Operationen der `Media` API

Das Komponenten-Interface der API

Wie werden die genannten Funktionalitäten nach Außen zur Verfügung gestellt? Es muss eine Funktionalität geben, welche einen `IMediumStore` für ein gegebenens `IMedium` erzeugt. Die API zur Erzeugung eines `IMediumStore` sieht daher prinzipiell wie folgt aus:

DES 084: **`IMediaAPI` als zentraler Zugriffspunkt mit Erzeugungsfunktionen für `IMediumStores`**
Die Schnittstelle `IMediaAPI` stellt den zentralen Zugriffspunkt zur Komponente `Media` dar. Mit der Methode `createMediumStore()` kann man einen `IMediumStore` erzeugen.

Begründung: Die Notwendigkeit einer weiteren Schnittstelle zusätlich zu `IMediumStore` ist klar: Ein `IMediumStore` bezieht sich genau auf eine "Zugriffssession" eines Mediums, und man will natürlich ggf. mehrere gleichzeitig haben. In einem solchen Interface noch Erzeugungsfunktionen für sich selbst unterzubringen ist schlechter Stil und behindert das klare Verständnis der Funktion des Interfaces durch Anwender.

Nachteile: Keine bekannt

Fehlerbehandlung

Generell werden Verletzungen des Schnittstellenvertrags wie durch DES 084 angegeben mit einer runtime exception quittiert.

Die folgende Tabelle fasst alle weiteren Fehlersituationen bei der Arbeit mit `Media` zusammen:

Fehlersituation	**Beschreibung**	**Reaktion** `jMeta`	**API Methode**
Medium existiert nicht	Das Medium existiert nicht, in `jMeta` muss es sich aber um ein vorhandenes Medium handeln.	Es handelt sich um eine abnorme Situation, daher wird eine `MediumAccessException`, eine runtime exception, geworfen.	`IMediaAPI .createMediumStore()`
Medium ist bereits gesperrt	Das Medium ist bereits durch einen anderen Prozess gesperrt. `jMeta` kann daher nicht auf dem Medium arbeiten. Es ist die Pflicht das Anwenders, sicherzustellen, dass das Medium nicht bereits verwendet wird.	Es handelt sich um eine abnorme Situation, daher wird eine `MediumAccessException`, eine runtime exception, geworfen.	`IMediaAPI .createMediumStore()`
Unbekannter Medium-Typ	Es wird eine `IMedium`-Implementierung durch den Anwender angegeben, die nicht unterstützt wird.	Dies wird ebenso als abnorme Situation eingestuft, die den Schnittstellenvertrag verletzt, daher wird die gleiche Exception wie bei Verletzungen des Schnittstellenvertrages üblich geworfen.	`IMediaAPI .createMediumStore()`

Fehlersituation	**Beschreibung**	**Reaktion** `jMeta`	**API Methode**
Ende des Mediums beim Lesen erreicht	Natürlich hat jedes Medium einmal ein Ende. Beim Lesen kann das Ende erreicht werden. Beim Schreiben ist dies nicht möglich - hier gehen wir davon aus, dass alle Ausgabemedien virtuell unendlich sind. Im Falle nicht vorhandenen Speicherplatzes wird z.B. mit einer `MediumAccessException` in Folge einer `IOException` reagiert. Es kann vom anwendenden Code nicht verlangt werden, dass er weiß, wann das Ende des Eingabemediums erreicht ist. Das Erreichen des Endes bei einer Leseaktion kann ein Fehler sein oder auch nicht - das hängt von der Anwendungssituation ab. Der Anwendungscode muss dies daher situationsbedingt behandeln.	Da es sich also nicht notwendigerweise um eine abnorme Siutation handelt, wird hier eine `EndOfMediumException`, eine checked exception, geworfen.	`IMediumStore .cache()`, `IMediumStore .getData()`
Timeout beim Lesen	Ein lesender Aufruf auf dem Medium kehrte nach einem ggf. konfigurierten Timeout nicht zurück.	Es handelt sich um eine abnorme Situation, daher wird eine `ReadTimedOutException`, eine runtime exception, geworfen.	`IMediumStore .cache()`, `IMediumStore .getData()`

Fehlersituation	**Beschreibung**	**Reaktion** `jMeta`	**API Methode**
Nur lesendes Medium	Die durch den Anwender angegebene `IMedium`-Implementierung ist read-only, daher ist nur lesender Zugriff möglich.	Versucht der Anwender dennoch, zu schreiben, ist dies eine abnorme Sitation und wird mit einer `ReadOnlyMediumException`, einer runtime exception, quittiert.	`IMediumStore .flush()`, `IMediumStore .insertData()`, `IMediumStore .removeData()`, `IMediumStore .replaceData()`
Ungültiges Cache-Offset	Mit `cache()` wird bei einem `InputStream` versucht, an einem Offset zu Cachen, der kleiner ist als der letzte Lese-Offset.	Da es sich um eine Fehlverwendung handelt, resultiert in diesem Fall eine `InvalidMediumReferenceException`, eine runtime exception.	`IMediumStore .cache()`
Stream-Daten nicht vorhanden	Mit `getData()` abgefragte Daten des Caches sind nicht im Cache für den angegebenen Offset vorhanden und das darunterliegende Medium ist ein `InputStream`.	Da es sich um eine Fehlverwendung handelt, resultiert in diesem Fall eine `InvalidMediumReferenceException`, eine runtime exception.	`IMediumStore .getData()`
Unbekannte oder ungültige `MediumAction`	Der Anwender übergibt einer Operation eine ungültige oder auch unbekannte `MediumAction`	Dies ist eine abnorme Situation und wird mit der runtime exception `InvalidMediumActionException` quittiert	`IMediumStore.undo()`

Fehlersituation	**Beschreibung**	**Reaktion** `jMeta`	**API Methode**
`IOException` in der Implementierung	Die verwendete Java-Implementierung wirft in einer beliebigen anderen als den bereits dargestellten Fehlersitationen bei einem Aufruf eine `IOException`.	Dies wird ebenso als abnorme Situation eingestuft, daher wird eine `MediumAccessException`, eine runtime exception, geworfen.	`IMediumStore .cache()`, `IMediumStore .getData()`, `IMediumStore .isAtEndOfMedium()`, `IMediumStore .flush()`
Der `IMediumStore` wurde bereits durch `close()` geschlossen	`MediumStoreClosedException`, eine runtime exception	`MediumStoreClosedException`, eine runtime exception	Alle

Table 11.3.: Fehlerbehandling in der Komponente `Media`

Zusammenfassend halten wir fest:

***DES 085:* Erreichen des Endes des Mediums ist nicht notwendig ein Fehler, andere Probleme bei I/O werden als abnorme Ereignisse eingestuft.**
`Media` betrachtet das Erreichen des Endes des Mediums nicht als abnormes Ereignis, sondern dies muss situationsbedingt durch den Anwendungscode behandelt werden. `jMeta` betrachtet Ausgabemedien als virtuell unbegrenzt und sieht daher keine Behandlung von Endebedingungen für das Schreiben (im Übrigen analog zur Java-API) vor.
Alle anderen Fehlersiutationen in `Media` sind gemäß Tabelle 11.3 abnorme Ereignisse.

Begründung: Siehe Tabelle

Nachteile: Keine bekannt

11.1.3. Desing der Implementierung

Zugriff auf das Medium

Der Zugriff auf ein MEDIUM wird in jeweils einer eigenen Klasse implementiert, wie in folgender Designentscheidung dargelegt:

***DES 086:* Interface `IMediumAccessor` mit Implementierungen für Zugriff auf MEDIEN**
Der Zugriff auf ein MEDIUM wird über ein Interface `IMediumAccessor` mit den folgenden Primitiven implementiert:

- Öffnen: Beim Öffnen wird ein exklusives Lock auf das Medium erstellt, falls das Medium dies unterstützt (siehe DES 086)
- Prüfung, ob aktuell geöffnet
- Schließen
- Lesen ab Offset x: Liest n Bytes vom externen Medium durch einen expliziten Zugriff, liefert die Bytes in einem `ByteBuffer`, das Offset x wird für nicht random-access-Medien ignoriert
- Schreiben ab Offset x: Schreibt n bytes, die in einem `ByteBuffer` übergeben werden, nicht implementiert bei byte-Stream-Medien (der Aufruf wird ignoriert)
- Verkürzen auf Länge n: Kürzt das Medium auf die neue Länge n, nicht implementiert bei byte-Stream-Medien (der Aufruf wird ignoriert)
- Prüfung, ob das angegebene Offset x das Ende des MEDIUMS darstellt, das Offset x wird für nicht random-access-Medien ignoriert
- Überspringen von Bytes: Überspringt n Bytes, für random-access-Medien tut diese Funktionalität nichts

`IMediumStore` greift ausschließlich über dieses Interface auf das MEDIUM zu.

Begründung: `IMediumStore` selbst kann sich um das Caching und die komplexe Umsetzung der Schreibfunktionalität kümmern, während der eigentliche Mediums-Zugriff von konkreten Implementierungen generisch umgesetzt wird, unabhängig davon, welches MEDIUM dahinter steckt. Klassisches Separation of concerns zur Erhöhung der Flexibilität und Verständlichkeit der Lösung.

Nachteile: Keine bekannt

Verwaltung der `IMediumReference`-Instanzen

Gemäß DES 086 müssen `IMediumReference`-Instanzen zentral von `IMediumStore` verwaltet werden. Gleichzeitig können wir mehrfache `IMediumReference`-Instanzen haben, die sich auf dasselbe Offset, aber inhaltlich unterschiedliche Daten beziehen - und zwar bei der Methode `insertData`. Neue Daten werden vor dem `flush()` am gleichen Offset nacheinander eingfügt, siehe DES 086. Letztlich müssen die `IMediumReference`-Instanzen dann bei einem `flush()` automatisch aktualisiert werden, wie in DES 086 und DES 086 geschildert.

Damit wird folgendes klar:

***DES 087:* Es ist kein Pooling von `IMediumReference`-Instanzen für gleiche Offsets möglich**
Entegegen Java-Strings ist die Wiederverwendung von `IMediumReference`-Instanzen in `createMediumReference()` nicht möglich. Damit ist gemeint: Wurde bereits eine `IMediumReference`-Instanz für Offset x erzeugt, kann diese Instanz beim nächsten Aufruf der Methode für das gleiche Offset x nicht erneut herausgegeben werden. Stattdessen muss eine neue `IMediumReference`-Instanz erzeugt werden.

Begründung: Angenommen, wir würden Pooling verwenden. Weiter angenommen, es gibt zwei mittels `insertData()` angemeldete Einfügungen der Längen n_1 und n_2 am gleichen Offset x. Dann würde die interne Implementierung lediglich eine Instanz von `IMediumReference` für das Offset x vorhalten. `flush()` muss das Offset der ersten Einfügung unverändert lassen, denn die Daten der ersten Einfügung werden ja an dieser Stelle eingefügt und bleiben dort. Allerdings muss das Offset der zweiten Einfügung angepasst werden, weil sich diese Daten dann nach dem `flush()` am Offset $x + n_1$ befinden. Damit kann nicht dasselbe Objekt für beide `IMediumReference`-Instanzen verwendet werden.

Nachteile: Bei vielen erzeugten `IMediumReference`-Objekten kann es zu hohem Speicherverbrauch kommen.

Darüber hinaus müssen die `IMediumReference`-Objekte in einer dedizierten Datenstruktur verwaltet werden:

***DES 088:* Alle jemals mit `createMediumReference()` erzeugten Instanzen werden in einer dedizierten Datenstruktur verwaltet, die Duplikate zulässt**
Alle Instanzen werden in einer Datenstruktur gehalten, die Duplikate zulässt. Diese muss dediziert sein, also einzig für den Zweck der Verwaltung aller jemals erzeugten `IMediumReference`-Objekte dienen.

Begründung: Dupliakte müssen wegen DES 088 möglich sein. Man kann diese Datenstruktur weder mit den Cache- noch den Datenstrukturen für schwebende Änderungen zusammenwerfen, da es einerseits natürlich immer mehr `IMediumReference`-Instanzen geben kann, als es Cache-Einträge oder schwebende Änderungen gibt, und andererseits Cache und Änderungsliste gelöscht werden können, während die `IMediumReference`-Instanzen bis zum expliziten Schließen des Mediums überdauern müssen.

Nachteile: Keine bekannten Nachteile

Sind die Einträge der durch DES 088 induzierten Datenstruktur in spezieller

Form sortiert? Das klärt folgende Designentscheidung:

DES 089: **Unsortierte ArrayList zur Verwaltung aller `IMediumReference`-Instanzen**
Es wird eine `ArrayList` als Datenstruktur für die Verwaltung aller `IMediumReference`-Instanzen genutzt. Das Einfügen der `IMediumReference`-Instanzen erfolgt in Erzeugungsreihenfolge. Die Liste wird nicht sortiert.

Begründung: Die Liste erlaubt Duplikate. Ein Sortieren der Liste bei jedem Einfügen würde zu durchschnittlich $O(n \log n)$ Laufzeitkomplexität beim Erzeugen einer `IMediumReference`-Instanz führen, wenn n die Listengröße ist. Eine aufsteigende Sortierung nach Offset wäre für `flush()` etwas effizienter, rechtfertigt den Aufwand beim Einfügen aber kaum, denn das Erzeugen einer `IMediumReference` ist eine deutlich häufiger verwendete Operation als `flush()`. Folgende Alternativen werden insbesondere nicht genutzt:

- Eine `Map<Long, List<IMediumReference>>` mit Offsets als Key und allen `IMediumReference`-Instanten für das Offset als Wert funktioniert nicht, weil sich Offsets ja durch Einfügungen verschieben, also müssten die Keys der Map entweder `IMediumReference`-Instanzen sein, oder aufwändig aktualisiert werden.
- Ein `TreeSet<IMediumReference>` scheidet zunächst aus, weil es keine Duplikate zulässt.
- Die Kombination eines `TreeSet` mit einem speziellen `Comparator`, der `IMediumReference`-Instanzen nur dann als gleich ansieht, wenn es sich um das gleiche Objekt handelt, und als größer, wenn das Offset gleich, aber das Objekt unterschiedlich, würde solche "Duplikate" mit gleichen Offsets akzeptieren und die korrekte Sortierung bei jedem Einfügen sicherstellen. Jedoch ist das `Set` dann nicht kompatibel mit `equals`, was in den javadocs nicht empfohlen wird. Zweitens gilt wieder: Der Mehraufwand beim Einfügen ist nicht gerechtfertigt.

Nachteile: Das Auffinden aller `IMediumReference`-Instanzen in der `flush()`-Implementierung, die größer als ein gegebenes Offset sind, hat $O(n)$-Komplexität, was aber durchaus tolerierbar ist.

Um die Komplexität von `IMediumStore` in handhabbare Teile zu zerlegen, wird folgendes beschlossen:

***DES 090:* Die Klasse `MediumReferenceFactory` wird für Verwaltung der `IMediumReference`-Instanzen vorgesehen**
`MediumReferenceFactory` implementiert die Designentscheidungen DES 090, DES 090 sowie DES 090. Die Klasse stellt dafür folgende Methoden bereit:

- `createMediumReference()` zum Erzeugen von `IMediumReference`-Instanzen
- `updateReferences()` zur Implementierung von DES 090, dabei wird eine `MediumAction`-Instanz übergeben
- `getAllReferences()` liefert alle verwalteten Instanzen
- `getAllReferencesInRegion()` liefert alle verwalteten Instanzen mit bestimmtem Offset-Bereich
- `getAllReferencesBehindOrEqual()` liefert alle verwalteten Instanzen mit Offsets größer als dem angegebenen Offset
- `clearAll()` löscht alle verwalteten Referenzen

Begründung: Verringerung der Gesamtkomplexität von `IMediumStore`.

Nachteile: Keine bekannten Nachteile

Als letzte Frage ergibt sich nun noch, wie man mit der in DES 090 angegebenen Methode `advance()` umzugehen hat. Diese erzeugt ebenso neue `IMediumReference`-Instanzen. Müssen diese auch der Offset-Verwaltung von `MediumReferenceFactory` unterliegen?

***DES 091:* Auch die mit `IMediumReference.advance()` erzeugten `IMediumReference` müssen von `MediumReferenceFactory` verwaltet werden.**
Dafür muss der neuen `IMediumReference`-Instanz bei Erzeugung ein Verweis auf ihre erzeugende `MediumReferenceFactory` mitgegeben werden. Um dennoch das einfache Erzeugen von Instanzen der `IMediumReference`-Implementierungsklasse (z.B. in Unittests) zu ermöglichen, wird der Konstruktor öffentlich gemacht.

Begründung: Der Anwendungscode kann beliebig `IMediumReference.advance()` aufrufen, und die zurückgegebenen Referenzen ebenso beliebig verwenden, beispielsweise für neu erzeugte Datenblöcke. Damit ist klar, dass auch diese Referenzen durch die automatische Korrektur von `MediumReferenceFactory.updateReferences()` angepasst werden müssen.

Nachteile: Sehr enge Kopplung zwischen `IMediumReference` und `MediumReferenceFactory`, da sich beide gegenseitig kennen müssen.

Open Issue 11.1: **– Prüfung: Weak References benutzen?**
Prüfung: Weak References benutzen?

Interne Datenstrukturen für das Caching

Der Cache verwaltet zunächst einmal Datenbytes pro Offset, immer unter der Annahme, dass die im Cache gehalteten Daten exakt identisch mit den Medien-Daten sind, gemäß DES 091. Es wurde bereits definiert, dass es eine explizite Methode `cache()` zum Einlesen von Cache-Inhalten und eine Methode `getCachedByteCount()` zum Abfragen zusammenhängender Byte-Zahlen gibt (siehe Tabelle 11.2 sowie DES 091 und DES 091).

Das Abfragen der Daten erfolgt dann über `getData()`, welche den Cache einerseits überspringen kann, andererseits jedoch diesen auch automatisch mit fehlenden Daten aktualisiert (DES 091 und DES 091).

Schließlich kann man Cache-Inhalte mit `discard()` gezielt freigeben oder das Caching komplett deaktivieren (siehe DES 091).

In vielen Fällen wollen wir abfragen, welche Anteile von zu lesenden, zu schreibenden oder auch zu löschenden Daten im Cache sind. Der Cache kann durch mehrfache Fülloperationen beliebig fragmentiert sein. Daher benötigen wir eine Klasse dafür:

DES 092: **Klasse `MediumRegion` für zusammenhängende Abschnitte eines Mediums, insbesondere für Cache-Abschnitte**
Die Klasse `MediumRegion` repräsentiert Bereiche eines Mediums, die unter Umständen gecacht sind. Dafür bietet sie folgende Methoden:

- `getStartReference()`: Start-`IMediumReference` des Bereiches abfragen
- `getSize()`: Länge des Bereiches abfragen
- `getBytes()`: Liefert null, falls nicht im Cache, sonst den `ByteBuffer` mit den gecachten Daten der Region
- `isCached()`: Liefert true, falls der Bereich gecacht ist, sonst false
- `isContained()`: Liefert true, falls die angegebene `IMediumReference` im Bereich enthalten ist, sonst false
- `discardBytesAtFront()`: Ermöglich Verkleinerung des Bereiches durch Verwerfen von Bytes an dessen Anfang
- `discardBytesAtEnd()`: Ermöglich Verkleinerung des Bereiches durch Verwerfen von Bytes an dessen Ende

Begründung: Die Cache-Bereiche und die nicht-gecachten Bereiche können einheitlich behandelt werden.

Nachteile: Keine bekannten Nachteile

Die Verwaltung der Cache-Inhalte wird von einer Hilfsklasse erledigt:

DES 093: **Cache-Verwaltung durch die Klasse** `MediumCache`
Die Cache-Verwaltung erfolgt durch die Klasse `MediumCache`, mit folgenden Methoden:

- `getCachedByteCountAt()`: Liefert die Anzahl der am Offset x zusammenhängend gecachten Daten zurück
- `getData()`: Liefert für den Offsetbereich $[x, x+n]$ eine Liste von `MediumRegion`-Instanzen, zurück, welche die gecachten und nicht gecachten Bereiche im Offset-Bereich angeben.
- `getCachedRegions()`: Liefert eine Liste aller aktuell gecachten `MediumRegion`-Instanzen zurück.
- `addData()`: Fügt am Offset x Daten in den Cache ein. Vorher dort gecachte Daten werden überschrieben.
- `discardData()`: Gibt am Offset x bis zu n Bytes aus dem Cache frei
- `clearAll()`: Gibt alle Daten des Caches frei
- `setMaxRegionSize()`: Setzt die maximal verwendete Größe einer Region. Default ist `Integer.MAX_VALUE`. Die erzeugten Regionen werden nach Änderung maximal diese Größe erreichen. Die bestehenden Regionen bleiben unverändert.
- `getMaxRegionSize()`: liefert die maximal verwendete Größe einer Region
- `enableCaching()`: Aktiviert oder deaktiviert das Caching
- `isCachingEnabled()`: Frag ab, ob das Caching aktiviert oder deaktiviert ist

Begründung: Verringerung der Komplexität von `IMediumStore`

Nachteile: Keine bekannten Nachteile

Wie werden die Cache-Regionen nun intern verwaltet?

DES 094: **Es wird eine `TreeMap` für die Verwaltung der Cache-Inhalte verwendet**
Es wird eine `TreeMap<IMediumReference, MediumRegion>` für die Verwaltung der Cache-Inhalte verwendet.

Begründung: Die Inhalte müssen basierend of Offsets ausgelesen werden. Daher ist eine nach Offset sortierte Datenstruktur notwendig. Sie ermöglicht das effiziente Auslesen aller `MediumRegion`s größer oder kleiner einem bestimmten Offset. Diese Operation sollte statt $O(n)$ (vermutlich) sogar eine Laufzeitkomplexität von $O(log(n))$ haben.

Nachteile: Keine bekannten Nachteile

Aus Performance-Gründen haben wir bereits die Möglichkeit für das Freigeben von Daten vorgesehen. Es zeichnet sich allerdings ab, dass man auch die Maximalgröße des Caches durch einen Automatismus regeln können muss:

DES 095: **Der Anwender kann die Maximalgröße des Caches einstellen**
Die Maximalgröße kann für `MediumCache` über `setMaxCacheSize()` gesetzt und mit `getMaxCacheSize()` abgefragt werden. Darüber hinaus kann die aktuelle Cache-Größe über `getCurrentCacheSize()` abgefragt werden. Standardmäßig ist die maximale Cache-Größe nicht limitiert, was durch eine Konstante `UNLIMITED` repräsentiert wird. Das Setzen der maximalen Cache-Größe führt zum Freigeben bzw. Verkleinern von vorhandenen Cache-Regionen, wird also sofort angewandt. Welche Cache-Regionen freigegeben werden, ist undefiniert. `addData()` prüft, ob die aktuelle Cache-Größe plus die neu hinzuzufügenden Bytes die maximale Cache-Größe überschreiten. Falls dem so ist, werden nur soviele Bytes ab Beginn des zu cachenden `ByteBuffer`s hinzugefügt, bis die maximale Cache-Größe erreicht ist, unter Umständen also keine Bytes. `addData()` liefert dann eine `MediumRegion` zurück, welche die tatsächliche Regions-Größe angibt.

Begründung: Dies ermöglicht dem Anwender aktiv, die Cache-Größe zu beeinflussen und `OutOfMemoryError`s vorzubeugen

Nachteile: Keine bekannten Nachteile

Wir sollten uns noch über Fragementierungen des Caches Gedanken machen. Angenommen, durch aufeinanderfolgende Aufrufe von `addData()` würde 20 mal jeweils genau 1 Byte in einem zusammenhängenden Offset-Bereich der Länge 20 gecached werden. Entstehen dadurch 20 verschiedene `MediumRegion`s mit einer Länge von 1? Die analoge Fragestellung ergibt sich für einen Aufruf von `getData()`, der sich über einen Offsetbereich mit Lücken in der Cache-Abdeckung ergibt. Nehmen wir beispielsweise an, der Cache enthält ab Offset x 20 Bytes, ab Offset $y := x + 30$ weitere 50 Bytes, hat also eine Lücke von 10 Bytes. Erfolgt

nun ein Aufruf von `getData()` für den Bereich $x - 10$ mit Länge 100, dann haben wir fünf Regionen:

- Region 1: $[x - 10, x)$ befindet sich nicht im Cache
- Region 2: $[x, x + 20)$ befindet sich im Cache
- Region 3: $[x + 20, y)$ befindet sich nicht im Cache
- Region 4: $[y, y + 50)$ befindet sich im Cache
- Region 5: $[y + 50, y + 60)$ befindet sich nicht im Cache

`getData()` wird nun die Regionen 1, 3 und 5 gemäß DES 095 im Cache ergänzen. Haben wir nach dem Aufruf dann also 5 `MediumRegion`s im Cache?

Wir legen fest:

***DES 096:* Fragmentierung zusammenhängender Cache-Bereiche wird vermieden**
Bei aufeinanderfolgenden Aufrufen von `addData()`, bei denen Aufruf 1 den Offsetbereich $[x, x + n)$ und Aufruf 2 den Offsetbereich $[x + n, x + n + m)$ abdeckt - die beiden Bereiche also direkt ohne Lücke aneinander grenzen - ist das Endergebnis eine `MediumRegion`, die den Offsetbereich von $[x, x + n + m)$ abdeckt.
Bei einem Aufruf von `getData()`, dessen Offset-Bereich sich über mehrere durch Lücken getrennte gecachte `MediumRegion`s erstreckt, werden die vom Medium gelesenen Lücken mit den bereits im Cache befindlichen, bisher getrennten `MediumRegion`s zu einer `MediumRegion` verschmolzen.
In beiden Fällen setzen wir voraus, dass die Gesamtlänge des neuen Bereiches kleiner als die konfigurierte maximale Regionsgröße gemäß DES 096 ist. Wird diese überschritten, gilt: Sei die Gesamtlänge n mit konfigurierter maximaler Blockgröße m, also $n > m$. Ist dann n durch m teilbar, so entstehen $\frac{n}{m}$ verschiedene `MediumRegion`s. Ist n nicht durch m teilbar, so entstehen $\lfloor \frac{n}{m} \rfloor + 1$ `MediumRegion`s.
Eine Fragmentierung von zusammenhängenden `MediumRegion`s im Cache passiert also nur beim Überschreiten der maximalen Regionsgröße.

Begründung: Mehr Objekte benötigen mehr Speicher und erfordern einen höheren Verwaltungsaufwand. Die Fragmentierung bietet keinerlei Vorteile.

Nachteile: Keine bekannten Nachteile

Eine verwandte Frage ist der Umgang mit Lücken, auch bereits bei aufeinanderfolgenden Aufrufen von `addData()`: Wie im eingehenden Beispiel werden 20 Bytes durch 20 Aufrufe von `addData()` in den Cache geladen. Jedoch befinde sich zwischen je zwei aufeinanderfolgenden Bytes eine "Lücke" von einem Byte, das nicht im Cache enthalten ist. Auch hier darf man die Frage stellen: Entstehen dennoch 20 `MediumRegion`s?

DES 097: **Fragmentierung unzusammenhängender Cache-Bereiche wird nicht unterbunden**
Die wie im oberen Abschnitt beschreibene Situation "eng benachbarter", aber unzusammenhängender Cache-Bereiche geringer Länge wird nicht vermieden.

Begründung: Dies würde zu noch höherer Komplexität in der Implementierung von `addData()` führen. Heuristiken, nach denen Zusammenfassungen erfolgen, müssten erst definiert werden. Denkbar wäre beispielsweise, dass ein Aufruf von `addData()` erkennt, dass es andere Offset-Bereiche gibt, die sich im Radius $[x-k, x+k]$ befinden, mit einem zu definierenden k. In diesem Fall könnten auch die "Lückenbytes" zusätzlich gecacht werden. Hier muss allerdings einerseits sichergestellt werden, dass ebenso die maximale Größe einer Region wie die maximale Größe des Caches nicht überschritten wird. Für `InputStream`s ist dieses "Lückenfüllen" nicht möglich und muss daher ohnehin unterbleiben. Zudem muss dieser Weg den Sonderfall des Endes des Mediums berücksichtigen. Stattdessen ist der Anwender selbst in der Pflicht, für sinnvolle, d.h. zusammenhängende Cache-Bereiche durch entsprechende Verwendung der Methode zu sorgen.

Nachteile: Keine bekannten Nachteile

Interne Datenstrukturen für die Verwaltung schwebender Änderungen

Für das zweistufige Schreiben müssen die durch `insertData()`, `removeData()` und `replaceData()` angemeldeten Änderungen sinnvoll verwaltet werden, damit sie später durch `flush()` verarbeitet werden können. Eine Änderung wird dabei durch eine `MediumAction` repräsentiert.

Halten wir zunächst folgendes fest:

DES 098: `MediumAction` **hat eine Sequenznummer zur Unterscheidung von Einfügungen am gleichen Offset**
`MediumAction` definiert eine Sequenznummer (beginnend bei 1) zur Unterscheidung von Einfügungen am gleichen Offset. Diese wird bei darauffolgenden Einfügungen am gleichen Offset um 1 inkrementiert. Zwei `MediumAction`-Instanzen sind per `equals()` gleich wenn:

- Sie sich auf die gleiche `IMediumReference` beziehen
- Sie denselben Änderungstyp haben (insert, remove, replace)
- Sie dieselbe Sequenznummer haben

Begründung: Zwei unterschiedliche `MediumAction`-Instanzen, die sich auf dasselbe Offset beziehen, können als solche sonst nicht in eine Reihenfolge gebracht werden. Die Reihenfolge müsste lediglich in externen Strukturen gehalten werden. Eine Unterscheidung und Sortierung ist dann schwieriger möglich.

Nachteile: Keine bekannten Nachteile

Nun können wir festlegen, in welcher Form die `MediumActions` gespeichert werden:

DES 099: `MediumActions` **werden in einer sortierten Datenstruktur ohne Duplikate vorgehalten**
Die vor einem `flush()` erzeugten `MediumActions` werden in einer nach Offset und Sequenznummer (gemäß DES 099) sortierten Datenstruktur vorgehalten, die keine Duplikate zulässt (z.B. `TreeSet`).

Begründung: Die Sortierung ist für die Abarbeitung in Reihenfolge durch `flush()` nötig, zwei `MediumActions` mit gleichem Offset, gleicher Sequenznummer und desselben Typs sollen nicht als Duplikat auftauchen können.

Nachteile: Keine bekannten Nachteile

Nun zur Verwaltung der `MediumActions`:

***DES 100:* Zur Verwaltung dient die Klasse `MediumChangeManager`**
Zur Verwaltung dient die Klasse `MediumChangeManager`, die intern DES 100 implementiert, mit folgenden Methoden:

- `scheduleInsert()` zum Anmelden eines *insert*, implementiert DES 100, erhält als Eingabe eine `MediumRegion` und liefert eine `MediumAction` des entsprechenden Typs und mit passender Sequenznummer zurück
- `scheduleRemove()` zum Anmelden eines *remove*, implementiert DES 100, erhält als Eingabe eine `MediumRegion` und liefert eine `MediumAction` des entsprechenden Typs und mit passender Sequenznummer zurück
- `scheduleReplace()` zum Anmelden eines *replace*, implementiert DES 100, erhält als Eingabe eine `MediumRegion` und liefert eine `MediumAction` des entsprechenden Typs und mit passender Sequenznummer zurück
- `undo()` zum Rückgängigmachen, implementiert DES 100
- `iterator()` liefert einen `Iterator<MediumAction>` zum lesenden Iterieren der Änderungen in der korrekten Reihenfolge, `remove()` wird nicht implementiert
- `clearAll()` löscht alle Änderungen

Dabei erzeugen die drei `schedule`-Methoden `MediumActions` gemäß DES 100 und DES 100.

Begründung: Verringerung der Gesamtkomplexität von `IMediumStore`.
Der Iterator erlaubt das Lesen der Änderungen in Reihenfolge, wird aber für die Abarbeitung nicht benötigt, wie wir später sehen werden. `remove` auf dem Iterator ist unnötig, da `undo()` eine Aktion rückgängig macht.

Nachteile: Keine bekannten Nachteile

Zuletzt stellen wir noch einige Gemeinsamkeiten zwischen `MediumActions` und `MediumRegions` fest, und definieren daher:

***DES 101:* `MediumAction` aggregiert eine `MediumRegion`-Instanz**
`MediumAction` aggregiert eine `MediumRegion`-Instanz, welche Start, Länge und Inhalt der mit der Aktion verknüpften Bytes enthält.

Begründung: `MediumAction` benötigt eine Start-`IMediumReference`, eine Länge der Änderung sowie ggf. die Änderungsbytes, falls vorhanden. Man kann sie daher als Klasse interpretieren, die sich auf eine `MediumRegion` bezieht. Dies ist eher eine "hat ein"- statt eine "ist ein"-Beziehung. Daher ist Aggregation hier angebrachter als Vererbung.

Nachteile: Keine bekannten Nachteile

Implementierung von flush

Die wohl komplexeste Funktionalität von `IMediumStore` ist `flush()`, denn:

- `flush()` muss alle bisher angemeldeten Änderungen durchgehen,
- mit den aktuellen Cache-Inhalten abgleichen,
- sinnvolle Blöcke der zu schreibenden Daten bilden,
- bei Einfügungen oder Löschungen Daten hinter der Einfügestelle lesen und erneut schreiben,
- die `IMediumReference`- und `MediumAction`-Instanzen aktualisieren,
- den Cache aktualisieren

Dann kommt noch hinzu, dass die schreibenden Operationen einige Eigenarten haben:

- `replaceData()` ist zunächst harmlos, denn es werden lediglich existierende Bytes überschrieben, ggf. wird das Medium nach hinten verlängert. Hierfür kann man die bereits in der Java File I/O vorhandenen Schreiboperationen einsetzen. Gäbe es nur diesen Fall, wäre flush() beinahe "trivial"
- `insertData()` und `removeData()` erfordern, die Daten hinter der Einfügung oder Löschung zu lesen, und dann wieder zu schreiben
- Erfolgen diese Operationen bei großen Dateien am Beginn, dann ist das Lesen einer großen Speichermenge bis zum Ende der Datei möglicherweise nicht möglich, ohne einen `OutOfMemoryError` zu erzeugen
- Damit muss blockweise gelesen und geschreiben werden
- Hierbei unterscheiden sich die beiden Operationen:
  - Bei `insertData()` von n Bytes ab Offset x müssen die Daten vom Ende des Mediums blockweise bis zum Offset x gelesen und geschrieben werden. Zuerst werden also die letzten k Bytes ab Offset r gelesen, und dann ab Offset $r + k$ geschrieben, dann wiederum k Bytes ab Offset $r - k$ gelesen, um dann bei r geschrieben zu werden usw. bis zum Offset x. Ein anderer Weg funktioniert nicht, wenn man keine Bytes der Datei verlieren möchte.
  - Hingegen bei `removeData()` von n Bytes ab Offset x müssen die Daten beginnend vom Offset $x+n$ blockweise bis zum Ende des Mediums gelesen und geschrieben werden. Zuerst werden also die letzten k Bytes ab Offset $x+n$ gelesen, und dann ab Offset x geschrieben, dann wiederum k Bytes ab Offset $x + n + k$ gelesen, um dann bei $x + n$ geschrieben zu werden usw. bis zum letzten Offset im Medium. Ein anderer Weg funktioniert nicht, wenn man keine Bytes der Datei verlieren möchte.

Fest steht damit zunächst, dass wir die Konfiguration einer maximalen Block-Größe für das Schreiben benötigen:

DES 102: **Eine maximale Schreibe-Block-Größe muss für den Anwender konfigurierbar sein**
Diese muss zwischen 1 und N Bytes liegen

Begründung: Benötigt aufgrund der notwendigen Operationen beim Einfügen und Entfernen. Durch die Konfigurierbarkeit kann der Anwender selbst entscheiden, wie viele Bytes in einem Rutsch gelesen und geschrieben werden sollen.

Nachteile: Keine bekannten Nachteile

Das Herausfinden, welche Operationen durchgeführt werden müssen, ist also eine komplexe Aufgabe. Diese komplexe Aufgabe können wir in eine Methode gießen:

DES 103: **`createFlushPlan()` erzeugt einen Schreib-Lese-Plan für ein flush in Form einer `List<MediumAction>`**
`createFlushPlan()` erzeugt einen Schreib-Lese-Plan für ein flush und liefert eine `List<MediumAction>`. Der Schreib-Lese-Plan enthält die in gegebener Reihenfolge auszuführenden Aktionen. Die Liste der möglichen Aktionen wird um `READ`, `WRITE` und `TRUNCATE` erweitert. Dabei ist:

- READ primitives Lesen von n Bytes ab einem Offset
- WRITE primitives Schreiben von n Bytes ab einem Offset
- TRUNCATE das explizite Kürzen der Datei, was insbesondere bei Löschungen notwendig ist

Jeder gelieferten `READ`-Aktion muss eine `WRITE`-Aktion folgen, sonst ist der Plan ungültig. Die Operation liefert auch `REMOVE`-, `REPLACE`- und `INSERT`-Aktionen explizit, obwohl diese implizit durch `READ-WRITE`-Aktionen implementiert werden könnten.
Der Schreib-Lese-Plan enthält damit zusätzlich zu den bereits durch den Anwender hinzugefügten Aktionen neue `MediumAction`s. Diese werden allerdings nicht in der internen Datenstruktur des `MediumChangeManager` eingefügt.
Der erzeugte Plan kann dann im Nachgang abgearbeitet werden.

Begründung: Die Ermittelung der notwendigen Operationen ist ein komplexer Vorgang, der separat erfolgen sollte. Ein sofortiges Ausführen einer Aktion wäre alternativ möglich, aber würde das Testen des entsprechenden Codes erschweren. `MediumChangeManager` ist der richtige Ort für diese Operation, da hier ohnehin alle `MediumAction`s verwaltet werden. Die im Plan zusätzlich enthaltenen `MediumAction`s werden nicht in die interne Datenstruktur eingefügt, weil die Operation so ideal wiederholbar und "zustandslos" ist.
Die Erweiterung um `WRITE` scheint unnötig, da es bereits eine Operation `REPLACE` gibt. Jedoch unterscheidet sich `WRITE` insofern, dass die zu schreibenden Bytes beim Erzeugen der Aktion noch nicht bekannt sind, im Gegensatz zu `REPLACE`.

Nachteile: Keine bekannten Nachteile

Nun hätten wir alle Zutaten beisammen, um das Schreiben via `flush()` zu implementieren:

1. Erzeuge den Schreib-Lese-Plan gemäß DES 103
2. Iteriere alle Einträge des Schreib-Lese-Plans, die folgenden Schritte gelten für jeden einzelnen Eintrag:
3. Führe die Aktion an der durch die `MediumAction` angegebenen Stelle durch:
   - Falls Aktion = `READ`: Lies n Bytes, die in der darauffolgenden Aktion geschrieben werden. Dazu werden zuerst via `MediumCache.getData()` die Regionen ermittelt. Diejenigen, die nicht gecached sind, werden

durch direkten Zugriff auf das Medium via `IMediumAccessor` gelesen. Dann wird ein entsprechender `ByteBuffer` schrittweise aufgebaut.

- Falls Aktion = `WRITE`: Schreibe die Bytes, welche die vorherige `READ`-Operation geliefert hat, durch direkten Zugriff auf das Medium via `IMediumAccessor`
- Falls Aktion = `REPLACE`: Schreibe die Bytes in der `MediumAction` durch direkten Zugriff auf das Medium via `IMediumAccessor`
- Falls Aktion = `INSERT`: Schreibe die Bytes in der `MediumAction` durch direkten Zugriff auf das Medium via `IMediumAccessor`
- Falls Aktion = `REMOVE`: Ignoriere die Aktion, weil sie implizit durch `READ`, `WRITE` und `TRUNCATE` durchgeführt wird
- Falls Aktion = `TRUNCATE`: Führe ein explizites Verkürzen des Mediums durch.

4. Aktualisiere den Cache (Teil 1): Falls Aktion = `REMOVE`, dann rufe `MediumCache.discardData()` auf, um die Bytes aus dem Cache zu entfernen.

5. Nur wenn die Aktion = `INSERT` oder Aktion = `REMOVE`: Rufe `MediumReferenceFactory.updateReferences()` für den Bereich auf, sodass alle dahinter liegenden `IMediumReference`-Instanzen aktualisiert werden

6. Aktualisiere den Cache (Teil 2): Falls Aktion = `WRITE`, `INSERT` oder `REPLACE`, dann rufe `MediumCache.addData()` auf, um die Bytes in den Cache aufzunehmen.

7. Falls Aktion = `INSERT`, `REMOVE` oder `REPLACE`: Rufe `MediumChangeManager.undo()` auf, um die Aktion zu entfernen

DES 104: **`flush()` wird gemäß des oben angegebenen Ablaufs implementiert**
`flush()` wird gemäß des oben angegebenen Ablaufs implementiert

Begründung: Der Schreib-Lese-Plan muss explizit alle Operationen enthalten, also auch die vom User ausgelösten Operationen `REPLACE`, `INSERT` und `REMOVE`, selbst wenn eigentlich `READ` und `WRITE` für deren Implementierung ausreichen würden. Grund dafür ist einerseits, dass bei einem `WRITE` nicht klar ist, ob die Bytes der vorherigen `READ`-Aktion verwendet werden sollen, oder vorgegebene Bytes. Darüber hinaus müssen die Aktionen des Anwenders explizit aus dem `MediumChangeManager` entfernt, und deren Einfluss auf `IMediumReference`-Instanzen explizit ausgeführt werden. Dazu benötigt man ihre konkreten Typen, `WRITE` reicht nicht aus.
`undo()` wird nur bei den Anwender-Operationen ausgeführt, weil nur diese gemäß DES 104 in den internen Datenstrukturen von `MediumChangeManager` verwaltet werden.
Die Cache-Aktualisierung ist zweigeteilt: Im Falle von `REMOVE` wird *zuerst* der Cache aktualisiert, um *danach* die `IMediumReference`-Instanzen anzupassen. Dies liegt darin begründet, dass ansonsten am gleichen Offset unter Umständen mehrere Cache-Regionen liegen würden. Dies würde den Cache also in einen inkonsistenten und fehlerhaften Zustand bringen. Mit der gleichen Begründung werden die anderen schreibenden Operationen erst *nach* dem Aktualisieren der `IMediumReference`-Instanzen angepasst.
Die Cache-Verwaltung wird komplett `MediumCache.addData()` überlassen, sodass darin Maximalgröße, zusammenhängende Bereiche etc. optimiert werden können.

Nachteile: Keine bekannten Nachteile

Denken wir über die Fehlerbehandlung dieser Abfolge nach:

- Geht beim Erzeugen des Schreib-Lese-Plans (Schritt 1) etwas schief, dann kann der Anwender es nochmal versuchen, da alle Änderungen noch vorhanden und auf dem externen Medium noch nichts geändert worden ist
- Geht beim Zugriff auf das externe Medium (Schritt 3) etwas schief, dann sind ggf. vorher bereits Aktionen des Schreib-Lese-Plans erfolgreich durchgeführt worden, das externe Medium also bereits geändert worden. Dies entspricht den Aussagen in DES 104. Da die vorherigen Operationen bereits entfernt wurden, haben wir aber einen sauberen "Wieder-Aufsetz-Stand", d.h. der Anwender könnte das Ganze auch hier erneut versuchen.
- Geht beim Aktualisieren des Caches in Schritt 4 oder 6 etwas schief, dann wird weder die Aktion entfernt noch werden die Medium-Referenzen aktualisiert. Analog, wenn ewas im Schritt 5 schiefgeht. In diesen Fällen ist entweder `MediumCache` oder `MediumReferenceFactory` in einem inkonsistenten Zustand. Die Frage ist, ob dann noch etwas zu retten ist. Das klärt die folgende Designentscheidung

***DES 105:* Rückgängig-Machen der Änderungen erfolgt immer, Cache wird bei Update-Fehlern geleert**
Sollte beim `flush()` in den Schritten 4 bis 6 eine `Exception` geworfen werden, dann soll die Aktion dennoch rückgängig gemacht werden.
Weitere Maßnahmen werden nicht eingeleitet, denn das Fehlschlagen der Schritte 4 bis 6 tritt nur im Falle von Programmierfehlern oder schweren Systemfehlern ein. In diesen Situationen ist Recovery ohne schwierig bis unmöglich.

Begründung: `flush()` ist eine sensible Operation, die zwar kein ACID erfüllen mag, aber zumindest im Fehlerfall das Gesamtsystem in einen halbwegs konsistenten Zustand belassen soll. Dazu gehört, dass bereits auf dem Medium erfolgte fehlerfreie Operationen auch nicht nochmals ausgeführt werden dürfen. Damit muss die entsprechende Aktion auch entfernt werden.
Der Versuch, auch Fehler bei Cache-Verwaltung oder `MediumReferenceFactory`-Zugriffen zu behandeln, führt zu einer hohen Komplexität, die im Sinne der Wahrscheinlichkeit dieser Ereignisse unnötig ist. Zudem ist auch dann das System i.d.R. nach wie vor in einem inkonsistenten Zustand. Solche Fehlerbehandlungen, die glauben etwas retten zu können, machen die Dinge dann ggf. auch nur noch schlimmer.

Nachteile: Keine bekannten Nachteile

Auf den zweiten Blick betrachtet ergibt sich aus der Reihenfolge der Operationen noch folgendes:

***DES 106:* Die Operation `REMOVE` muss im Schreib-Lese-Plan hinter derjenigen `WRITE`- oder `TRUNCATE`-Operation folgen, die den letzten Teil der gelöschten Bytes überschreibt**
`REMOVE` soll erst dann rückgängig gemacht werden, wenn auch alle Bytes, die gelöscht werden, durch die Bytes auch überschrieben werden.

Begründung: Sonst würde die Aktion `REMOVE` schon vor den eigentlich ausführenden Operationen erfolgen. Damit ist kein ein erneuter Versuch des Löschens durch den Anwender im Falle eines Fehlschlagens der entsprechenden `WRITE`- oder `TRUNCATE`-Operation möglich.

Nachteile: Keine bekannten Nachteile

Konfigurationsparameter

Auch wenn die durch den Anwender konfigurierbaren Parameter zur öffentlichen Schnittstelle gehören, so können sie erst hier aufgeführt werden, da erst nach Design der Implementierung klar geworden ist, was von Außen konfigurierbar sein muss.

Zunächst legen wir folgendes fest:

***DES 107:* Die Konfiguration von `Media` erfolgt auf einer `IMedium`-Instanz**
Die Konfiguration von `Media` erfolgt auf einer `IMedium`-Instanz. Damit haben alle Konfigurations-Parameter den Scope eines `IMedium`s, beziehen sich also nur auf dieses. Es gibt entsprechend eine `setProperty()`- und eine `getProperty()`-Methode auf einem `IMedium`.

Begründung: Die gesamte interne Implementierung der wesentlichen Klassen, also `IMediumStore`, `IMediumAccessor`, `MediumCache` usw. arbeitet auf genau einem Medium. Der Anwender kann `IMedium`-Instanzen direkt erzeugen und nach Belieben konfigurieren, unabhängig von anderen `IMedium`-Instanzen.

Nachteile: Keine bekannten Nachteile

Nun ist die Frage, welche Konfigurationsparameter wir benötigen. Diese sind in Tabelle 11.4 aufgeführt.

Medium	Parameter-Name	Typ	Default-Wert	Beschreibung
`File`, `InputStream`	`ENABLE_CACHING`	boolean	`true`	Aktiviert oder deaktiviert das Caching für das Medium gemäß DES 107. Der Cacheinhalt wird beim Setzen auf `false` sofort geleert.
`File`, `InputStream`	`MAX_CACHE_REGION_SIZE`	int > 0	`Integer .MAX_VALUE`	Setzt die maximale Größe einer Cache-Region gemäß DES 107.
`File`, `InputStream`	`MAX_CACHE_SIZE`	long > 0	Long `.MAX_VALUE`	Setzt die maximale Cache-Größe gemäß DES 107.
`File`, byte-Array	`MAX_WRITE_BLOCK_SIZE`	int > 0	8192	Die maximale Größe einer Lese-Schreib-Aktion in Bytes, die durch `INSERT`s oder `REMOVE`s beim `flush()` veranlasst wird, siehe DES 107.
`InputStream`	`SKIP_ON_FORWARD_READ`	boolean	`false`	Bei einem Aufruf von `IMediumStore.cache()` für einen `InputStream` mit einem bisher noch nicht gelesenen Offset werden bei einem Wert von `false` immer alle Daten zwischen aktuellem und angegebenem Offset in den Cache gelesene. Bei Angabe von `true` wird zukünftig stattdessen `IMediumStore.skip()` verwendet. Vergleiche DES 107 und die Beschreibung von cache() in 11.2.

Medium	Parameter-Name	Typ	Default-Wert	Beschreibung
`InputStream`	`READ_TIMEOUT_MILLIS`	int > 0	Integer `.MAX_VALUE`	Der maximale Lese-Timeout für `InputStream`s gemäß DES 107, in Millisekunden.

Table 11.4.: Konfigurationsparameter der Komponente `Media`

Bibliography

[JavaSoundSample] Java Sound Resources: Examples - SimpleAudioPlayer.java
Matthias Pfisterer, Florian Bomers, 2005
http://www.jsresources.org/examples/SimpleAudioPlayer.java.html
8

[JMFDoc] JMF API Documentation (Javadoc)
Sun Microsystems Inc., 2004
http://download.oracle.com/docs/cd/E17802_01/j2se/javase/technologies/desktop/media/jmf/2.1.1/apidocs/index.html
6

[JMFWeb] JMF Features
Oracle, 2016
http://www.oracle.com/technetwork/java/javase/features-140218.html
3

[MetaComp] Metadata Compendium - Overview of Popular Digital Metadata Formats
Jens Ebert, August 2010
I, 3, 5, 5.2, 8.1

[PWikIO] Personal Wiki, article "File I/O mit Java"
Jens Ebert, March 2016
http://localhost:8080/Start/IT/SE/Programming/Java/JavaIO
11.1.1, 11.1.1, 11.1.1, 11.1.1

[Sied06] Moderne Software-Architektur: Umsichtig planen, robust bauen mit Quasar
Johannes Siedersleben, 2004
ISBN: 978-3898642927
9.1.1

[WikJavaSound] Wikipedia article Java Sound, October 3rd, 2010
http://de.wikipedia.org/wiki/Java_Sound
7

[WikJMF] Wikipedia article Java Media Framework, October 3rd, 2010
http://en.wikipedia.org/wiki/Java_Media_Framework
2.4.2, 1, 2, 5