jMeta 0.5
Design

Version: 0.1
Status: - Draft -

May 28, 2017

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:

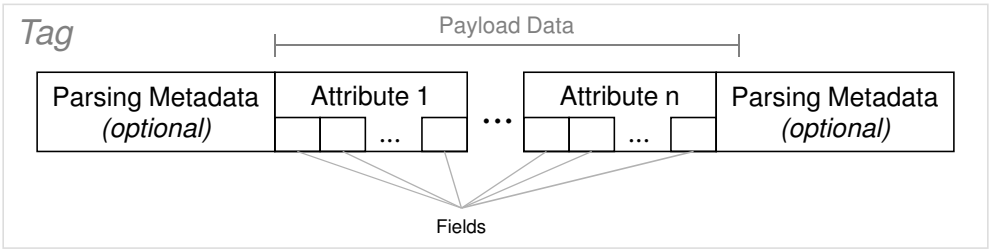

Figure 3.1.: Struktur eines TAGs

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:

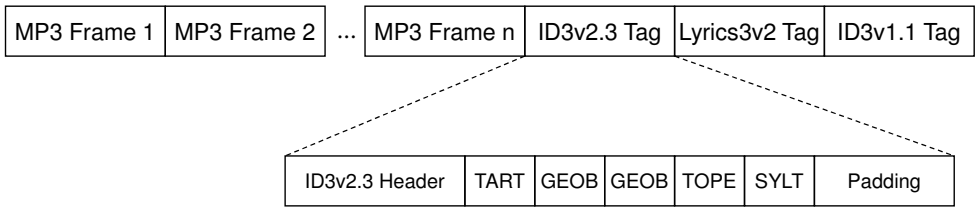

Figure 5.1.: Beispiel 1: MP3 File mit ID3v2.3, ID3v1.1 und Lyrics3

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:

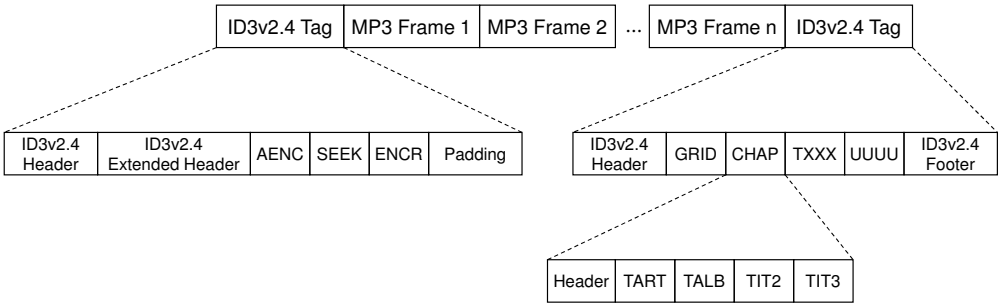

Figure 5.2.: Beispiel 2: MP3 File mit zwei ID3v2.4 Tags

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:

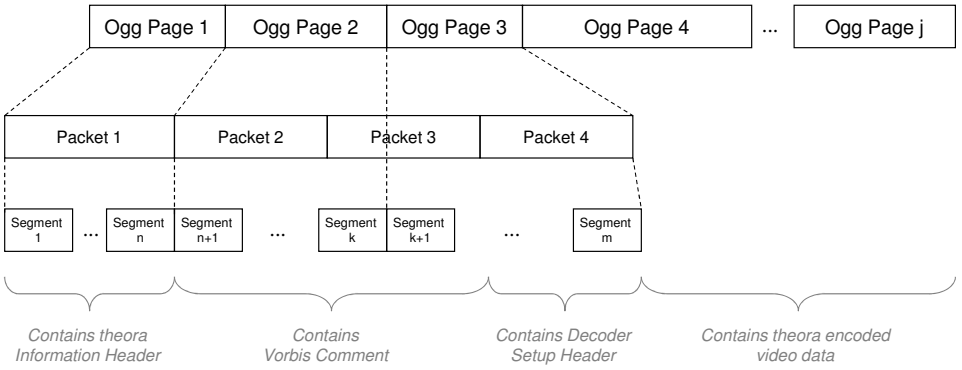

Figure 5.3.: Beispiel 4: Ogg Bitstream mit Theora und VorbisComment

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:

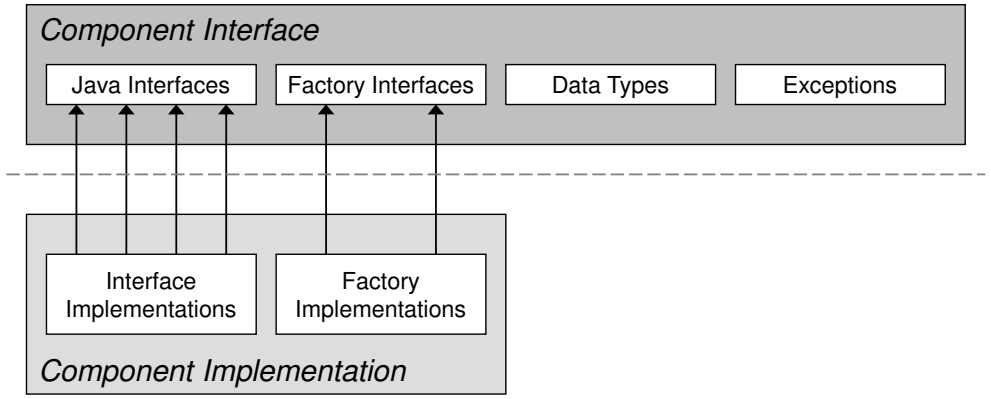

Figure 6.1.: Struktur einer Komponente

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:

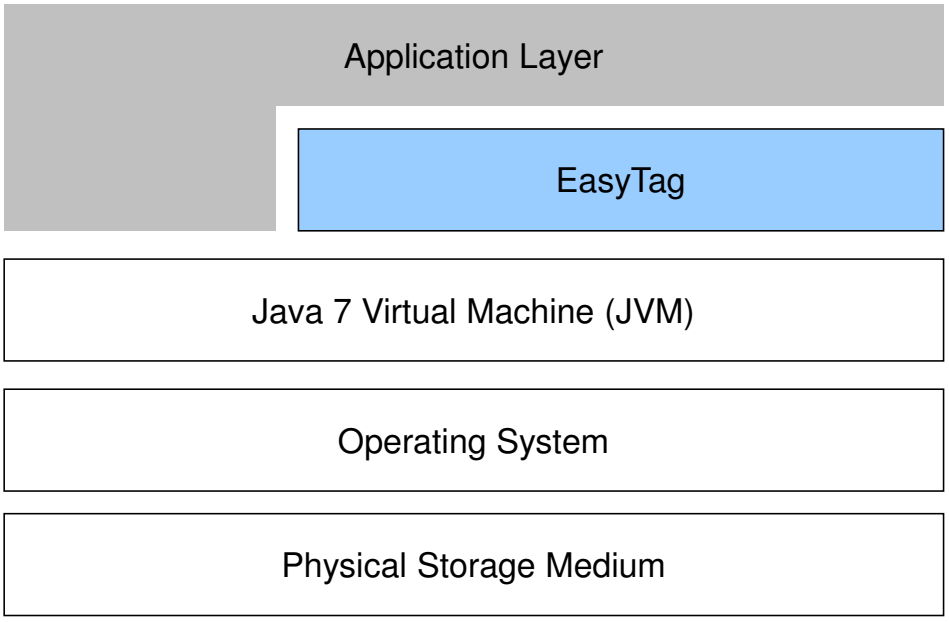

Figure 7.1.: Technische Infrastruktur von `jMeta`

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.

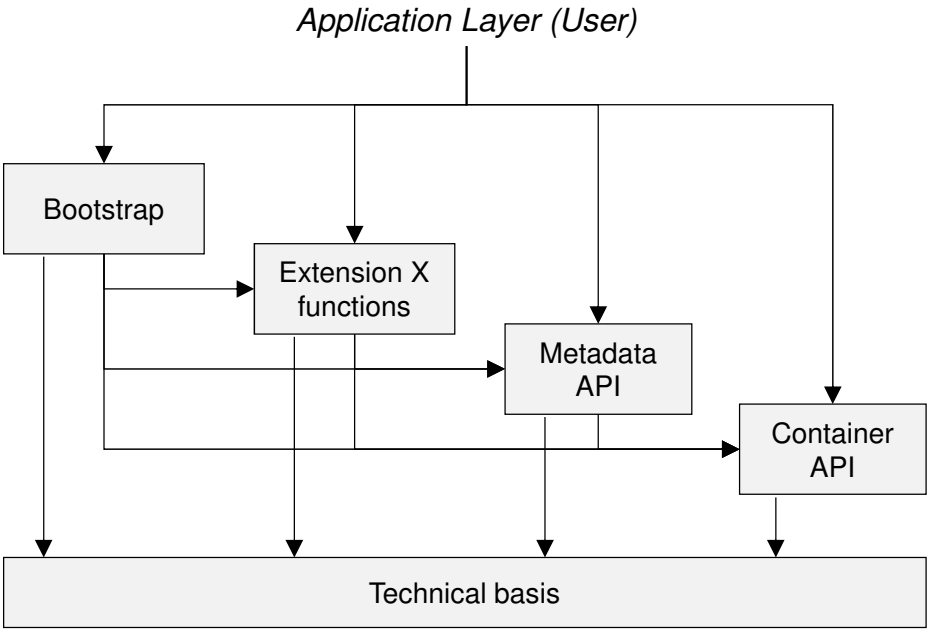

Figure 8.1.: Subsysteme von `jMeta`

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, z.B. 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

Die Frage ist natürlich, wann und auf welchen Levels genau geloggt wird:

***DES 025:* Informative Ausgaben der Library auf INFO-Level, Details auf DEBUG, und Fehler auf ERROR**
Wir loggen folgende Ausgaben auf dem angegeben Level:

- `INFO`: Jegliche Ausgaben, die sich auf die Systemumgebung und die Version der Library beziehen werden einmalig beim ersten Verwenden der Library in der aktuellen JVM geloggt.
- `DEBUG`: Detailausgaben zu Fortschritten bestimmter Startup-Aktivitäten oder auch komplexer Operationen werden bei jedem Aufruf der komplexen Operation geloggt
- `ERROR`: Im Falle von Laufzeitfehlern, die die Library selbst wirft, werden zusätzlich zur Exception Fehlertexte in Englisch auf dem Level ERROR geloggt. Ausnahme: Fehler beim Prüfen von Vorbedingungen, insb. Eingabeparametern führen niemals zu zusätzlichen Logausgaben.

Begründung: Beliebiges unnötiges Loggen wird eingedämmt. Komplexe, ggf. langlaufende Hintergrund-Operationen benötigen detaillierte Informationen für evtl. Fehleranalysen. Bei Laufzeitfehlern der Library selbst soll der Anwender bzw. der Analyst durch entsprechende Details im Logfile darauf hingewiesen werden, dass es an einer bestimmten Stelle ein Problem gegeben hat.

Nachteile: Keine bekannten Nachteile

Zusätzlich muss noch festgelegt werden, ob es eine zentrale Instanz für das Logging gibt, oder ob stattdessen einfach eine Library genutzt wird.

DES 026: **Es wird keine Logging-Komponente erstellt, stattdessen wird slf4j direkt an allen notwendigen Stellen genutzt**
Statt einer dedizierten, eigen-implementierten Logging-Komponente, die jede andere Komponente kennt, für das Logging genutzt wird und intern ein Logging-Framework kapselt, wird slf4j an allen notwendigen Stellen direkt genutzt.

Begründung: Zuerst wurde eine zentrale Logging-Komponente implementiert, die letztlich nur java util Logging verwendet und festdefinierte Formatierungen nutzte. Grundidee war es hierbei, das Logging "wegzukapseln", um Logframeworks austauschbar zu gestalten sowie einige Konfigurationsaufgaben bezüglich des Loggings zu übernehmen. Diese Variante hat sich als wenig sinnvoll erwiesen, aus folgenden Gründen:

- Es muss jeder Komponente auf irgendeinem Wege eine Instanz der Logging Komponente mitgegeben werden bzw. diese muss sich eine Instanz dieser Komponente besorgen, damit können alle Komponenten nur gemeinsam mit der Logging-Komponente wiederverwendet werden
- Soll das Logging für wichtige zentrale Library-Elemente wie `ISimpleComponentRegistry` oder das Verwalten von Erweiterungen genutzt werden, muss es vor deren Initialisierung initialisiert werden, da auch und gerade diese Bestandteile extensiv loggen müssen. Allerdings kann die Logging-Komponente nicht erstellt werden, wenn es noch kein Komponentenframework gibt. Ein Henne-Ei-Problem, was in der Vorversion zu seltsamen und unnötigen Konstrukten geführt hat
- Innovationen im gekapselten Logging-Framework bei Versions-Upgrades erfordern neuen Aufwand in der Logging-Komponente; insgesamt muss die Logging-Komponente entweder alle Funktionen des Logging-Frameworks weitergeben oder diese stark beschnitten anbieten

Die "Kapselung" des Loggings ist ein nur auf den ersten Blick gutes Argument. Erstens sollte man sich fragen: Wie oft tauscht man das Logging-Framework aus? slf4j bietet genau die Austauschbarkeit von Logging-Implementierungen direkt an. Als zusätzliches Plus kann der Anwender von `jMeta` selbst die slf4j-Implementierung seiner Wahl nutzen, was die Library u.a. ideal in bestehende Anwendungen integriert, ohne diesen die Verwendung einer weiteren neuen Log-Library aufzuzwingen. Ausgaben von `jMeta` können so beispielsweise direkt in die Hauptlogdatei der Anwendung mit aufgenommen werden. Zweitens ist Logging an sich "0-Software" ohne anwendungsspezifische Logik, eine eigene Komponente dafür wirkt übertrieben und verkompliziert neben den bereits erwähnten notwendigen Abhängigkeiten die Architektur.

Nachteile: Keine bekannten Nachteile

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 027: **`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 028: **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 029: **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 030:* 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 031:* 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 032:* 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 033:* `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 034: **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 035: **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 035 und DES 035.

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

***DES 036:* 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 036 ist eine Komponente ein "Singleton". Dennoch muss zusätzlich klar definiert werden, wie der Lebenszyklus einer Komponente aussieht:

***DES 037:* Der Komponentenlebenszyklus besteht aus Initialisierung und Nutzung**
Eine Komponente (und damit die Einzel-Implementierung des Komponenteninterfaces gemäß DES 037) 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 038:* 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 039:* Ids und Interfaces müssen eindeutig sein, parallele Versionen derselben Komponente sind nicht zulässig**
Im Rahmen derselben `ISimpleComponentRegistry`-Instanz darf dieselbe Komponenten-Id gemäß DES 039 nur einmalig verwendet werden. Auch dasselbe Komponenten-Interface darf nur einmalig in einer solchen Instanz verwendet werden. Auch wenn die Komponentenbeschreibung gemäß DES 039 ein Versionsattribut enthält, heißt dies nicht, dass mehrere unterschiedliche Versionen derselben Komponente zur gleichen Zeit aktiv sein können.

Begründung: Alles andere würde einerseits DES 039 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 040:* `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 041: **`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 041. 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 this section, the design of the component Media is described. Basic task of the component is to provide access to memory areas which contain multimedia data. Primarily these are files.

The term MEDIUM needs to be sharpened here: In "3.5 MEDIUM" we had defined: "A MEDIUM defines the storage medium of DATENBLÖCKE. It can be a file or a MEDIEN-STREAM, or the main memory itself."

In detail, the term summarizes the aspects "physical storage" and "access mechanism" (e.g. file-based random-access, or byte stream). Thus there might perfectly be two different media which access the same physical storage, but using different access mechanism. The term MEDIUM is an abstraction and potentially allows even more special possibilities, like media streams, databases etc.

11.1.1. Basic Design Decisions Media

Here, the fundamental design decisions of the component Media beschrieben.

Supported Medien

This section lists decisions about supported MEDIEN. To start with, it is clear that jMeta must support files as basic medium.

> ***DES 042:*** **Support for random-access file access**
> jMeta supports the use of files as input and output medium via Media with access mechanism "Random Access".
>
> **Begründung:** Files are *the* fundamental and most common digital media containers, even in 2016. Of course MP3 files, AVI files etc. with multimedia content are wide-spread. A library such as jMeta must support files as core element. To more efficiently process files, random-access is inevitable. Especially reading at arbitrary offsets - e.g. tags at end of file - as well as skipping of unimportant content is efficient to implement with random-access.
>
> **Nachteile:** No disadvantages known

But also reading streams shall be supported to increase the flexibility of the library:

DES 043: **Support for sequential, reading byte streams**
`jMeta` supports the use of reading byte streams, i.e. `InputStream`s for input in mode "sequential access".

Begründung: `InputStream` represents the most general alternative of a MEDIUM from Java perspective, which ensures a potentially higher flexibility for using `jMeta`. E.g. multimedia files can be read from ZIP or JAR archives using streams, and support for media streams might be easier to implement in later releases - However: To state clearly: media streams do have nothing to do with this design decision. They might be implemented completely different in upcoming releases.

Nachteile: An `InputStream` supports by definition only sequential access and no random-access (e.g. via `FileInputStream`). Thus there might be higher complexity for implementation, as well as signficant performance drawbacks because of lacking random-access.

Last but not least, the library offers access to RAM contained data, due to flexibility:

DES 044: **Support for random-access to byte arrays**
`jMeta` allows for random-access to byte arrays as input medium and output medium.

Begründung: Already loaded memory content can be parsed with `jMeta` without need for artistic climbs, increasing flexibility of the library.

Nachteile: No disadvantages known

What about `OutputStream`s? That is discussed in the following:

DES 045: **No support for writing byte streams**
`jMeta` does not support writing byte streams, i.e. `OutputStream`s.

Begründung: `OutputStream`s are write-only, but still not random-access. Thus we would need - provided we want to access random-access media in a random-access style - a second implementation next to writing random-access. A combined usage of `InputStream`s and `OutputStream`s for Read-/Write access on the same medium is not designed into the Java API and leads to diverse problems. As `jMeta` already implements writing to output files and byte arrays, for reasons of effort, `OutputStream`s are not supported as output media. The user might implement `OutputStream`s easily by him- or herself, e.g. by first writing into byte arrays, then into an `OutputStream`.

Nachteile: No disadvantages known

Consistency of Medium Accesses

Parallel access to the same medium from different processes or threads, reading by one and writing by the other, might lead to unpredictable difficulties - even without using any caching. If you e.g. have some parsing metadata like the length of a block in bytes at hand, but a parallel process shortens the block, your read access trying to fetch the whole block will run into unexpected end of file or read inconsistent data.

To avoid such problems, there are special locking mechanisms for exclusive access to the bottleneck ressource, at least for files. We define:

DES 046: **Locking of files during `jMeta` access**
Files are *always* locked during access by `jMeta` explicitly. File content are protected by exclusive locks from corruption by other processes and threads. See [PWikIO], where we show that a file in Java must be explicitly opened for writing to be able to lock it. "During access" means: After opening it and until closing it. The lock thus might be long-term. `jMeta` opens a file for writing (and locking) even if the user explicitly requested read-access only.

Begründung: Other processes and threads of the same JVM cannot access the files and corrupt any data, which avoids consistency problems.

Nachteile: It is not possible to access the same file in parallel threads when using `jMeta`. It seems rather unlikely that such parallel access to the same file (e.g. reading at different places) can speedup an application. But for future media this might indeed be a drawback.

The locking of byte streams or memory regions does not make sense, as discussed in the following desing decisions:

DES 047: **No locking of byte streams**
Byte streams are not locked

Begründung: The interface `InputStream` does not offer any locking mechanisms. `jMeta` will not try to guess the kind of stream and lock it (e.g. by checking if it is a `FileInputStream`).

Nachteile: No disadvantages known

For different processes, the os usually protects access of memory regions. The question is whether `jMeta` should protect access to byte arrays:

***DES 048:* No locking of byte arrays**
Byte arrays are not locked

Begründung: This makes not much sense as the user anyways gets a reference to the byte array by the API, and thus can access and manipulate the raw bytes arbitrarily in a multi- or single-threaded way. Protecting it by thread locking mechanisms increases complexity and does not seem to generate any benefits whatsoever.

Nachteile: No disadvantages known

Unified API for Media Access

In the wiki article [PWikIO], we have shown clearly the differences between byte streams and random-file-access. With so many difference the question arises: Can this be unified at all and does the effort make sense here? The least common demoninator for random-file-access and `InputStream`s is the linear reading of all bytes in the medium. This is clearly too less. It denies all advantages of random-access. The intersection of features for a unification is therefore not making sense.

Moreover, we want a unifying combination of both approaches:

***DES 049:* Unified access to all supported media types in one API**
`Media` offers a common abstraction for accessing files via random-access, `InputStream`s as well as byte arrays. This API provides the advantages of both access mechanisms via a common interface. The implementation throws exceptions of kind "Operation not supported" in some cases, if a feature is not supported by the medium. In other cases, a meaningful alternative behaviour is implemented. The using code must perform branche decisions at some places depending on the medium type.
While byte arrays are no problem for the abstraction, even random-access files and `InputStream`s have more in common as you might think at first glance:

- The operations Open, (sequential) Read, Close.
- `InputStream`s can also (at least technically) be assigned a beginning, offsets and an end.
- Files can be read-only, too, which `InputStream`s are always by definition.

Writing access to a read-only medium are acquitted with a runtime exception (especially for an `InputStream`).
The main difference between files and `InputStream`s is of course: Random access is possible for files, while `InputStream`s can only be read sequentially. This difference can be potentially decreased using mechanisms such as buffering.

Begründung: The API of the component `Media` gets easier for outside users, its usage feels more comfortable. Using components of `Media` can offer their users in turn an easier interface. At the same time, the advantages of both approaches (random-access and better performance for files, generality and flexibility for streams) are still available.

Nachteile: A few operations of the API cannot be implemted for both media types, which makes case decisions in the client code necessary in some cases.

Two-Stage Write Protocol

When Writing, it is all about bundeling accesses and buffering. We want optimum performance und thus want to implement these mechanisms. Therefore we commit to following design decisionfor implementing writing in `jMeta` in general:

DES 050: **`Media` uses a two-stage write protocol controlled by the user**
The first stageis the mere registration of changes, that need to written to the external medium. In this first stage, there is no access to the external medium yet. The second stage is the operation *flush*, the final writing and commiting of all changes to the external medium. The underlying implementation bundles the write actions according to its needs into one or several packets and execute the write only in the second stage.

Begründung: An efficient write implementation is possible. Internally, write actions can be bundled as needed to perform better. And this can be done without forcing the user to do it himself. The user can perform write (registration) actions whenever his code architecture needs it. Saying this, the user code is not burdened with too much restrictions or rules. Furthermore, the potential possibility of an "undo" of already registered actions comes into view.

Nachteile: Errors that occur when actually flushing changes to the external medium are recognizes potentially quite late. Thus the registration of changes is quite fast while the flush itself can be a long taking process. Bugs might be introduced by user code forgetting to implement the second step, the flush.

Even if we implement this, it must be clearly stated that this is not in any way a transaction protocol as implemented by some O/R mappers (e.g. hibernate) or application servers. The mentioned protocol is much simpler and not in the least capable to provide ACID! Thus the following exclusion:

DES 051: **Writing in `Media` does not guarantee ACID, in case of errors during *flush*, there is no rollback**
ACID (atomicity, consistency, isolation and durability) is not ensured neither by the implementation ofn `Media` nor in gerenal by the Java File I/O. If e.g. an error occurs during Writing in the *flush* stage, some data has been written already, while upcoming data will not get written anymore. There is no undo of already written data. The operation *undo* must not be mixed up with a rollback and it is no action that is done automatically. While isolation and durability can be more or less provided, the user is responsible for consistency and atomicity himself.

Begründung: A transaction manager that guarantees ACID, and this for files, is really hard to implement (correctly). This requirement is somehow out of scope, no other competing library is doing something similar. `jMeta` will not be a database!

Nachteile: No disadvantages known

Requirements for the Two-Stage Write Protocol

Which writing operations must be offered? One method write() - at the end it is the only really writing primitive of the Java File I/O - is not sufficient. How do you remove with this method? write() equals *overwriting*, which is not convenient at all. `Media` must offer a better API, taken some of the burdens of I/O from the user. Here, we only specify the necessary operations, without going into details with their implementation - this will be done later.

To develop a good design, however, you must first list down the user's requirements to `Media`. This especially includes the requirements for a two-stage write protocol. Main users of the component is definitely the component `DataBlocks`. It uses `Media` to extract and write metadata from and into tags. Without going into the design details of `DataBlocks`, here we nevertheless list detailed requirements that `DataBlocks` has for `Media` regarding two-stage writing, see table 11.1.

ID	Requirement	Motivation
AMed01	It must be possible to insert bytes	Formats such as ID3v2 can be dynamically extended and have a payload of flexible length. Before an already present data block, it must be possible to insert another one. There is especially a need for an insertion operation in the case when metadata with dynamic length need to be written at the beginning of a file.
AMed02	It must be possible to remove bytes	With the same motivation as for insertion. It must be especially possible to remove entire metadata tags.
AMed03	It must be possible to replace bytes and not only overwrite, but also grow or shrink an existing byte area with replacement bytes	In metadata formats, there are both static fields with fixed length as well as dynamic fields such as null-terminated strings. If bytes are already present, it must be possible to overwrite them to save costly remove and insert operations. The growing and shrinking is especially useful and represents a higher level of abstration. If this would not be possible, replacing a previous small string value by a new longer or shorte one would need to be implemented with two operations (overwrite and insert or remove, respectively).

ID	Requirement	Motivation
AMed04	Inserted data (Anforderung AMed01) must be changeable before a flush e.g. by extending, overwriting or removing of child fields inside the inserted data block	Based on the two-stage write protocol, an arbitrary number of writing changes can be made before a flush, and these might correct each other. E.g. a new ID3v2 tag footer is inserted, that stores the length of the tag. Assume that after this, a new frame is inserted into the tag, before the flush. This requires the size field in the firstly inserted footer to be changed afterwards again, before the flush.
AMed05	Replaced data (Anforderung AMed03) must be changeable before a flush e.g. by extending, overwriting or removing of child fields inside the replaced data block	A prominent example is insertion of and step-by-step extension of a frame into an ID3v2 tag: For the first creation as well as each extension, the size field of the tag must be changed, which induces a replace operation each time. E.g. it is allowed that users first only create and insert the new frame, and then insert new child fields afterwards, step by step.
AMed06	The padding feature of several data formats should be used by jMeta	Formats such as ID3v2 allow padding, i.e. using an overwrite buffer are to avoid newly writing the whole file. jMeta must use this feature when writing data, such that e.g. an insert only affects the file content until the padding area, effectively decreasing the padding, while a remove increases the padding, but the overall tag size remains the same. It is rather an indirect requirement which needs not necessarily be implemented by Media only.

ID	Requirement	Motivation
AMed07	The operations replace, remove and insert must be undoable before a `flush`	This allows to avoid unnecessary accesses to the medium and to undo mistakes by end users.

Table 11.1.: Requirements for the two-stage write protocol by `DataBlocks`

Based on these requirements, we can first define the following basic design decisions for writing:

DES 052: **`Media` offers the writing operations *insert*, *replace* and *remove***
The user can:

- *insert* N bytes at a given offset
- *replace* N bytes at a given offset by M new bytes
- *remove* N bytes at a given offset

Begründung: These are operations, that are in principle already necessary for a metadata library: `replace` is needed for formats with static length (such as ID3v1) and those allowing a padding mechanism or similar (such as ID3v2). For dynamically extensible formats such as ID3v2, additional possibilities to `insert` and `remove` data blocks are necessary. A dynamic replacement (replace N bytes by $M = N$ or $M \neq N$ bytes) is necessary to easily change fields with dynamic lengths, without the need to inconveniently call several different operations (e.g. first `replace`, then `remove` when decreasing the length of a string by setting a new value).
The burden to implemen these convenient operations using the Java File I/O, which essentially only offers write() and truncate(), is taken over by `Media`, such that the user need not care.

Nachteile: No disadvantages known

As we have a two-stage write protocol, the *undo* of not yet flushed changes is possible, and according to the requirements also necessary.

DES 053: **Writing operations on a medium can be undone with *undo* before a *flush***
Writing operations lead to pending changes according to DES 053. These can be undone according to the requirements defined above.

Begründung: The application logic can require *undo* in some cases, e.g. for corrections of mistakes done by an end user. Instead of requiring to call the inverse operation (if any at all), the user is much more convenient with undoing the operation itself directly. This also ensures that the using code does not need to trace changes to be able to find which is the inverse operation.

Nachteile: No disadvantages known

Caching

The component `Media` takes over I/O tasks with potentially slow input and output media. Thus, it is here where the basic performance problems of the whole library need to be solved. We will approach these topics with some motivation and deductions.

In [PWikIO], basic stuff regarding performance with file access is discussed. The ground rule for performant I/O is minimizing accesses to the external, potentially slow medium. For writing, we already introduced DES 053. The question is, how you can make reading perform better, too.

At first it is quite clear that for reading, you should help yourself with buffering to improve performance:

> ***DES 054:*** **Reading access can be done using a buffering mechanism, controlled by the component's user**
> For each reading access, the calling code can specify the number of bytes to read, which corresponds to a buffering. The code controls the size of the buffer by itself. It potentially can also read only 1 byte. It lies in the responsibility of the calling code to read an amount of bytes that makes sense and minimizes read accesses.
>
> **Begründung:** A hardcoded fixed length buffering would rather lead to performance disadvantages, as there might be read too much bytes, more than usually necessary in average. Furthermore, reading two or severeal times, depending on the size of this fixed length, would be necessary. According to [PWikIO], there is no "one size fits all" for buffer sizes in file I/O. The logical consequence is to let the user decide.
>
> **Nachteile:** No disadvantages known.

A further important aspect is caching: With *Caching*, we understand the possibly persistent storage of MEDIUM contents in RAM to support faster access to the data. Buffering differs from caching in a sense that buffering is only a short-lived temporary storage without the necessity of synchronisation.

To start with, we can see - besides the already mentioned buffering - different kinds of "Caching" in an application that is based on `jMeta`:

- During file access there are caches on hardware level, in the OS and file system.
- Java supports temporary buffering via the `BufferedInputStream`, and Caching explicitly via `MappedByteBuffer`s.
- Applications that mostly are interested in human-readaable metadata read these, convert them via `jMeta` in a clear-text representation and show this representation in their GUI. The GUI model represents in this cases a kind of caching, as in case of changes of the attributes via this GUI, it is not

necessary to re-read again from the medium. This is surely not a hardware-related caching of raw data.

If we look at all these alternatives, the question arises why at all an additional built-in caching in `jMeta` would be needed? For answering this question, we should look at some use case szenarios for the library: One scenario is the already mentioned reading of metadata from a file to display it in a GUI. For such a case, caching would usually not be very useful. You read once, and maybe twice, if the user wishes to update the screen. It would be an acceptable performance without caching. Another use case is the arbitrary jumping between parts of a container format file using a low-level API to process specific contents. This is true "random access". The question is: Do you want to read the same place twice? Possibly yes. Instead, would you want a direct medium access again? Possibly yes or no.

The last question brings up a general problem with caching: The problem of synchronizity with the external medium. If the medium has been changed in between, the cache content is probably aged and invalid. Code that accesses the cache can usually not recognize this.

We first sum up the determined advantages and disadvantages:

Advantages	**Disadvantages**
+ Performance improvement when reading the same offset multiple times, as cache access is much faster than the access to the external medium	− When only accessing once there is of course no performance improvement
+ In a cache you are - in principle - more flexible to reorganize data than on an external medium, making it easier to correct, undo or bundle changes.	− Changes on the MEDIUM cannot be recognized and lead to invalid cache content that might lead to erroneous behaviour or data corruption in follow-up write actions.
	− There is additional code necessary for caching, e.g. questions such as "when is the data freed?" must be answered. For consistency topics, even more complex code is necessary.
	− Move heap space required

Table 11.2.: Advantages and disadvantages of caching in `jMeta`

The disadvantages outweight the advantages. Why should you then use caching in `jMeta`? Some of the previous design decisions combine well with a caching

approach:

- DES 054 may or may not be easier to implement using a cache. In this case the cache would be used to store the registered changes before a flush. However, if it would only be this, a cache would be greatly too complex, Easier solutions are possible for holding the not-yet-flushed data.

- DES 054 can be merged with a cache, i.e. anything that has been buffered should directly go into the cache for upcoming read actions

- DES 054 can be achieved using a cache, as we see just a little later

Thus we decide:

DES 055: `Media` **uses permanent fast storage (Caching) for read medium data**
`Media` uses a RAM based, permanent fast storage (Cache) to store already read content of the MEDIUM. Upcoming read accesses access the cache content (if present) only.

Begründung:

- We provide faster repeated read access to already read data to the end-user
- This can be used for direct implementation of DES 055 in a sense of buffering when reading. The cache works as the buffer for DES 055.
- To achieve DES 055 is possible using a cache

Nachteile: Were given in table 11.2. The alternative is a direct medium access. To summarize the disadvantages against a direct medium access:

- Higher code complexity
- More heap required, the cache is durable
- The medium might change by external processes, such that the cache content is not in synch anymore.

Note that this last mentioned disadvantage is mostly mitigated by DES 055.

How to use caching to better achieve DES 055?

***DES 056:* Caching is used to better mitigate the differences between `InputStream`s and files according to DES 056.**
The data that has been read from an `InputStream` are always put into a cache. Reading actions are therefore allowed to "go back" to already read data, by not issuing another direct access (which is anyway not possible using an `InputStream`), but by taking the data from the cache. "Read ahead" for areas that have not yet been reached on the `InputStream` lead to the behaviour tha all data up to the future offset us read and cached.

Begründung: This implements DES 056 nearly entirely, "transparent" to the user.

Nachteile: Even more heap space is necessary for `InputStream`s, as in extreme cases the whole medium might end up in the cache, which might lead to `OutOfMemoryError`s.

Of course, the disadvantages mentioned in DES 056 are heavy-weigth. If you wouldn't do anything about it to mitigate this disadvantage, then DES 056 would be nonsense, as the advantages of this approach would be dramatically overshadowed by its disadvantages.

As a first step, the following three design decisions are necessary:

***DES 057:* The user can release cache content explicitly and can even disable caching entirely**
Releasing cache data can be done fine-grained for a specified offset range.
Thus the user himself can control the size of the cache, whereas it must be clear that any follow-up random-access reading will lead to repeated slow read access to the external medium (in case of a random-access medium) or to an exception (in case of an `InputStream`), respectively. The behaviour for `InputStream`s cannot be different as there is no data to return, and it cannot simple be read again, the stream has already progressed, there is no turning back in that case. Moreover the user can disable caching entirely. Here, too, access to previously cached offsets is not possible for `InputStream`s and will be acquitted by an exception.

Begründung: The user is responsible to decide about the memory footprint: E.g. if the medium is comparatively small, caching can be tolerated. If it is a big medium, the user has two possibilities: Cleaning up the cache regularly (e.g. data first read starting at a size threshold), or he can even disable caching, however demanding a step-wise processing of the data.

Nachteile: The implementation of `Media` gets more complex due to corresponding case decisions.

DES 058: **The `Media` API allows the skipping of bytes for `InputStreams`**
Bytes must not necessarily be read and delivered to the user. Instead, skipping is possible via (`skip`). For `InputStreams` skipping is a built-in functionality. For random-access skipping is not needed.

Begründung: The user can explicitly skip data that is not needed and can be ignored.
Second motivation is that DES 058 in case of `InputStreams` makes it necessary to read a big number of bytes in case of reading data from higher offsets (i.e. areas that were not read yet) to finally get to the demanded offset were actually reading is to start. That of course might bloat the cache: E.g. assume current offset is 0, and the user wants to read 100 bytes starting from offset 1000. What to be done with the bytes between offsets 0 and 1000? Accorind to DES 058, these bytes must be read into the cache. However, with skipping there would be a second possible alternative.

Nachteile: No disadvantages known

DES 059: **Performance drawbacks of `InputStreams` are explicitly documented**
Reducing differences between `InputStreams` and files by caching in accordance to DES 059 means: You must deal with the fact that you cannot store virtually unlimited `InputStreams` in a cache. For file access, the user can also choose between `FileInputStream` and more direct access via `RandomAccessFiles`. The performance drawbacks induced by using `FileInputStream` compared to random-access - which are introduced by a unified API according to DES 059 - are explicity described in the `jMeta` documentation. The mitigation mechanisms (skipping of bytes, releasing cache data, disabling caching) are explicitly described with their corresponding consequences.

Begründung: There are no wrong expectations by providing the unified API. The contract is described clearly enoughto the user. He must choose the medium best suited for his purpose.

Nachteile: No disadvantages known

After these results, the disadvantages previously listed in table 11.2 and in DES 059 need a closing look:

- Code Complexity: The higher code complexity cannot be disregarded. You have to accept it when implementing a permanent caching. It implies you need very good unit and integration tests to ensure it works as expected.
- More Heap Memory: It is usual to achieve a better runtime performance by increasing the memory footprint. So it it shere. To nevertheless avoid `OutOfMemoryErrors`, we have defined DES 059, DES 059 and DES 059 and

thus give enough room for the user to avoid these situations.

- Data Corruption due to Out-Of-Synch Medium: That the cache receives updates that are not yet persisted on the external medium is allowed according to DES 059. A problem might arise due to changes by other processes or threads. These cannot be handeled in a general way by `jMeta`. Thus we have introduced the locking of media in DES 059. Even this cannot give a full protection for some OSs. The user is in any case informed about irresolvabe inconsistencies by a runtime exception.

Implementing the discussed caching mechanisms is a big challenge. It will be more detailed in the implementation part of this component. Here, we can only exclude one way of implementing it:

DES 060: **`MappedByteBuffer` will not be used to implement caching**
You could come with the idea to use the Java NIO class `MappedByteBuffer` for implementing the caching of DES 060. However, we do not use it and implement another solution "by hand".

Begründung: It is not guaranteed, that the any OS supported by Java also supports a `MappedByteBuffer`. It is also not guaranteed that the data "cached" is really in RAM. For each concecutive region of a medium a new `MappedByteBuffer` instance including new OS call would need to be created. Thus this approach is unpredictable and might not in any case yield the wished for results.

Nachteile: The "by hand" caching is harder to implement.

At the end we shortly list a special case for of caching for byte array media:

DES 061: **For byte array media, caching is always disabled**
For byte array media, caching is always disabled

Begründung: byte arrays already are in RAM, caching would just be unncessary overhead

Nachteile: No disadvantages known

Reading Access to the Medium

The two-stage write protocol introduced in "11.1.1 Two-Stage Write Protocol" brings up some questions regarding reading the data. The most important among these: What does the user need to consider after calling a writing operation (stage 1) and before *flush*ing these changes (stage 2)? Especially: What do reading calls return after already having made changes, that are however not yet *flush*ed? The possibilities we have:

1. Either the last persisted state on the medium after opening it or the last successful *flush*, respectively,

2. Or already a state the includes any "pending" changes introduced by writing calls before the *flush*?

You could base your answer on the following: For sure alternative (2), as it is this way that transactions mostly work on databases: What you have already written during the transaction, you re-read later, too, even if the transaction is not yet persisted. However, the view of `jMeta` is as follows:

***DES 062:* The user can only read what is currently persisted on the medium**
Even if there are pending changes not yet *flush*ed (e.g. inserts, removes), with `Media` the user can only see the latest flushed state.

Begründung: The changes the user has registered are coming form the user, and he thus could potentially keep bookmarks of them. Therefore their sole management by `Media` is - at this point in time - not strictly necessary.
Furthermore it must still be possible to read the data of a datablock that is threatened by a pending remove.
Another good reason for this behaviour is that the reading operations are much less complex, as they do not need to consider any pending changes. The code that reads data can be sure to always only work on a persistent state. Thus it cannot occur that logic is basing on data that is not yet persisted.

Nachteile: The expectation that "what I have written before - even if pending - I can re-read afterwards" is not fulfilled. The user must manage this for changed data by himself, at least temporarily until the next flush.

In "11.1.1 Caching", caching has been discussed in detail. For buffering, we first need an operation to do buffering without actually returning the buffered data:

***DES 063:* Explicit operation for buffering of media data**
There is an operation *cache* which buffers n data bytes starting at a given offset, without returning this data. Additionally, there is an operation to query the number of bytes buffered concecutively starting at a given offset.

Begründung: Necessary for implementing DES 063.

Nachteile: No disadvantages known

Additionally, you must be able to get your hands at the buffered data:

***DES 064:* Explicit operation to get read data**
There is an operation *getData* which returns n data bytes starting at a given offset

Begründung: Without it there would not be any possibility to read data from a medium

Nachteile: No disadvantages known

Now the question areises, how the operation *getData* interacts with the cache, which is answered here:

***DES 065:* *getData* combines data from the cache with data form the medium and updates the cache thereby, if necessary**
getData reads data from the cache, if they are entirely contained in it. Are they not entirely contained, *getData* reads the bytes present in the cache, and reads the non-present ones from the medium, adding it to the cache afterwards. Thus data is combined from the two sources. The user can control for each call, if *getData* behaves as mentioned previously or in any case directly accesses the medium, i.e. ignoring the cache, while still updating it.

Begründung: To ensure efficient reading, *getData* can be used as such to fetch data from the cache, if anyhow possible, and only in other cases the medium must be accessed. The forced direct access is offered as additional possibility. Updating the cache by read data is useful, e.g. if the cache is very fragmented and the *getData* result would reduce fragmentation.

Nachteile: No disadvantages known

We want to define now how the read media data is represented:

***DES 066:* Read media data is represented as read-only `ByteBuffer`**
Read data is not returned in the form of `byte` arrays, but as `ByteBuffer` instances that are read-only.

Begründung: Firstly, users can directly gain profit from conversion functions offered by `ByteBuffer`, on the other hand the implementation is more flexible when it comes to the content of the `ByteBuffer`, as only the bytes between `position()` and `limit()` can be read. Using this e.g. an internally managed, much bigger `ByteBuffer` object can be returned as a read-only view instead of copying it.

Nachteile: No disadvantages known

Let's discuss the topic of timeouts. In Java, each reading and writing I/O call might block. How to deal with this? The following design decision clearly states

how.

DES 067: **`jMeta` supports only reading timeouts, and this only for byte streams**
`Media` only offers the possibility to configure timeouts in milliseconds for reading from byte streams.

Begründung: We assume that reading data from files takes a while and take into account that it might block. Reading data from files is, however, usually not a candidate for long or even any blocking. As the implementation of a timeout mechanism can be quite complex, we avoid to to this for files. For `InputStream`s, like media streams or socket connections, however, blocking is a quite possible case. Thus timeouts when reading from byte streams are really useful and `jMeta` offers to configure them. Writing operations forr byte streams are not supported according to DES 067.

Nachteile: Higher complexity for byte stream implementation.

11.1.2. API Design

On the basis of the design decisions made in the previous section, we can now develop an API design for the component `Media`. The API is the public interface of the component, i.e. all classes that can be used by other components to access the `Media` functionality.

Reprasentation of a Medium

The medium has appeared a lot of times already as a term, thus a representation as a class makes sense.

***DES 068:* Media are represented as interface and implementation class with following properties**
A medium is represented as Java interface `IMedium` and allows users of `jMeta` to specify a concrete physical medium (i.e. the implementations of the interface `IMedium`). As implementations we support a `FileMedium` according to DES 068, according to DES 068 aan `InputStreamMedium` and according to DES 068 a `InMemoryMedium`.
A medium has the following properties:

- Is random-access: Yes/No - **Motivation:** This property has strong impact on the read and write process, yet it is an intrinsic property of the MEDIUM itself and not of the access mechanism. Thus it is directly available for at a MEDIUM.
- Currently exists: Yes/No - **Motivation:** Checking existence in Java can be done at the MEDIUM level itself.
- Read-only: Yes/No - **Motivation:** This property disables writing in practice if set to "Yes". Some MEDIEN can never be written (e.g. `InputStream`s), for others it is possible. This flag shall be used to also give the `jMeta` user a possibility to signal he wants to only access read-only.
- Current lenght in bytes (only relevant for random-access) - **Motivation:** Java offers queries for each kind of `IMedium` except `InputStream`. Thus this should be implemented directly in the `IMedium` implementation. For `InputStream` and non-random-access media in general, terms like length to not make much sense. Thus here there is no value, but a constant indicating an unknown length. In spirit pf design decision DES 068, it is a currently persisted lenght and not a length including any not-yet persisted changes.
- A clear text name of the MEDIUM - **Motivation:** This is helpful for identification purposes of the `IMedium` e.g. in log output. It can be derived from e.g. a file name, depending on the medium type.
- The "wrapped" object representing the raw medium or its access mechanism, e.g. the file, the `InputStream` or the byte array.

Begründung: It can be controlled in detail which medium types are supported. The user can specify the medium to use in a comfortable way. Further API parts get more easier, as their interfaces must not distinguish between different media types, but rather only use the abstaction that `IMedium` offers. Motivation for each of the properties see the listing above.

Nachteile: No disadvantages known

Due to consistency reasons there are some restrictions regarding the manipulation of media properties:

***DES 069:* If a `IMedium` implementation is writable, it must also be random-access**
Every in principle writable `IMedium` implementation must be random-access, too.

Begründung: The `jMeta` APIs for writing content can thus concentrate on random-access output media. No separate API design and implementation for output media that are not random-access is necessary. The API gets easier for end-users. Lack of non-random-access output media such as `OutputStream`s can be mitigated via the examples in DES 069.

Nachteile: No disadvantages known

Here is a very importante note for byte array media:

***DES 070:* For byte array media, a writing method for resetting the whole byte array is necessary**
The user can reset the bytes of the medium via a public method `setBytes()` of class `InMemoryMedium`.

Begründung: It is mostly not harmful to offer the method as public, it is even an advantage for the users, as he can set the bytes himself. Only between registering write operations and a flush, this call leads to unexpected behaviour. This methode is very important for the implementation: Via writing actions, the byte array must be extended or shrinked in some situations. This basically means reacreating and copying the array. The method must thus be public, as the corresponding implementation funcitonality will be for sure placed in another package.

Nachteile: No disadvantages known

Positions in and Lengths of a Medium

In each case where dare is read from or written to a MEDIUM, the question "where?" arises. Usually libraries use integer or long variables to represent offsets. It must be said however: Offsets do not only make sense for random-access media. You could also interpret them as offset since start of reading from an `InputStream`, which is actually the way it is done in `jMeta`. We decide:

***DES 071:* Byte offsets are used for any kind of MEDIEN**
Byte offsets that refer to a position on a MEDIUM are used for all media types: random-access and non-random-access. For byte streams they refer to the position of the current byte since start of reading the first byte after opening the stream, which has offset 0. The offset-based reading is simulated as specified in DES 071 and DES 071, because when directly reading from an `InputStream` via Java API, offsets are not needed, it is always read from the current position of the stream.

Begründung: We must therefore distinguish between random-access and non-random-access only at a few places in the implementation. Users can use the API uniformly and irrespective of the actual medium type (with restrictions: see DES 071).

Nachteile: No disadvantages known

It makes not so much sense to represent offsets only via a primitve data type. Instead, the representation as a user-defined data type offers some advantages:

***DES 072:* `jMeta` uses the interface `IMediumReference` to represent offsets on a MEDIUM.**
The interface binds both the MEDIUM and the offset on this MEDIUM together, and thus is a kind of "global" address of a byte. Next to reading medium and of offset, it offers some helper methods:

- `behindOrEqual()`: Returns true if another `IMediumReference` is located on the same medium behind of at the same position as this instance.
- `before()`: Returns true if another `IMediumReference` is located on the same medium before the position of this instance.
- `advance()`: Creates a new `IMediumReference` that is located by the given byte number before (negative argument) or after (positive argument) this instance.

Begründung: We clearly state how the library deals with offsets. We can thus implement some helper functions into the datatype (e.g. validation, offset comparison, advance etc.) which ensure reuse and ease working with offsets in general.

Nachteile: No disadvantages known

Now we come to a central decision when it comes to dealing with lengths and offsets:

DES 073: **`jMeta` uses long for length and offset specifications, byte is always its unit**
In `jMeta`, lenghts and offsets are always specified using the Java datatyp long. The lenght is in any case the number of bytes, offsets are zero-based, linearly increasing byte offsets.

Begründung: This guarantees uniformity. However, we also want to meet the requirement "4.8 ANF 008: Lesen und Schreiben großer Datenblöcke". Integer with a maximum of 4.3 GB is already too limited, which leaves only long as a viable option. The datatype long allows for positive numbers up to $2^{63} - 1 = 9223372036854775807$, i.e. approximately $9 \cdot 10^{18}$ bytes, which is 9 exabytes or 9 billion gigabytes. From current point of view, such lenghts and offsets for input media, even for streams, should be sufficient for some decades to come. Furthermore, big data chunks are almost in any case subdivided in small units that can be easier handled, and these small units will not have big lengths. Even the Java file I/O uses long as offset and length datatype in most cases.

Nachteile: More memory for saving offsets and lengths is necessary. If we look at the development of storage media, storage needs and processing speed it might be that in 100 years the maximum data volume of long will be reached. If `jMeta` is still used in these future scenarios, a change request would be worth it!

A rather seldom special case is dealt with in the following design decision:

DES 074: **No special handling of long overflows**
For unusal long uninterrupted reading from `InputStream` you could think that even when using long, it could come to an overflow in some time. This is however very unlikely, thus this case is not treated. The implementation always assumes taht the current offset is positive and can be incremented without reaching the max long number.

Begründung: Even here it holds true: The datatype long allows positive numbers up to $2^{63} - 1 = 9223372036854775807$. Let us assume that an implementation could make it to process 10 GB per second, then it would still need 9 billion seconds, i.e. nearly 30 years, to reach the offset limit and create an overflow.

Nachteile: No disadvantages knownn

Now the problem arises that the medium changes due to writing access. How do the offsets change in this case? Is it necessary to update already created `IMediumReference` instances according to the changes on the mediums, or not? If yes, when this needs to happen? In principle, we could see following alternatives:

1. Never update already created `IMediumReference` instances
2. Update already created `IMediumReference` instances directly for each pend-

ing change registered (see DES 074)

3. Update already created `IMediumReference` instances only when an explicit *flush* according to DES 074 occurs

Assume that `IMediumReference` instances are not updated when writing. That means the user code remembers a position of an element in form of a `IMedium-Reference` instance, and uses it to read or write data. If e.g. an inseration operation takes place on the medium before the offset of the `IMediumReference` instance, then the instance refers to another data byte than before, and thus not anymore to the object it was referring to initially. We should not only think of raw bytes but - as necessary for data formats - *objects*, i.e. parts of the binary data that form a specific unit with which has a specific meaning, representing something. Then failure to update the offset is fatal. Code using `Media` locates an object at the wrong place if the medium changed before that offset meanwhile.

To formulate the following design decision a bit easier, the vague term of "Object" used above is now defined a bit sharper: An object is a consecutive byte unit starting at aspecific offset x and it has a length of n bytes. `remove`, `insert` and `replace` in the offset interval $[x, x + n]$ change these objects, which cannot be in any case specifically treated by `Media`.

DES 075: **`Media` needs to automatically update `IMediumReference` instances after medium changes**
All `IMediumReference` instances ever created for a medium must be updated automatically whenever this medium changes. The kind of update needed is more complex than you would think on first glance.
Let y be the insert or remove offset and k the number of bytes to insert or remove. Let $\overline{x}$ be the offset of the `IMediumReference` instance after updating. Then the following detailed rules apply:

- *`insert` before the object start offset:* Is $y \leq x$, then $\overline{x} := x + k$. I.e. this includes the case that new bytes are inserted exactly at offset x.
- *`insert` behind the object start offset:* Is $y > x$, then $\overline{x} := x$, i.e. it does not change the start offset of the object, and this even in the case that $y < x + n$, i.e. the action happens *within* the object. If and how this changes the object semantically is lying in the hands of the user and cannot be recognized or interpreted by `jMeta`.
- *`remove` before the object start offset without overlap:* Is $y + k \leq x$, then $\overline{x} := x - k$. Thus k bytes are removed before the object, however the removed region does not overlap with the object.
- *`remove` before the object start offset with overlap:* Is $y \leq x$, but $y + k > x$, then the removed region overlaps the object. It is thus a *truncation* of the object starting at front, and it might even reduce the object to length 0. Thus the start offset of the object shifts $x - y$ to be equal to y, thus it follows that $\overline{x} := y$. If and how this changes the object semantically is lying in the hands of the user and cannot be recognized or interpreted by `jMeta`.
- *`remove` behind the object start offset:* Is $y > x$, then $\overline{x} := x$, i.e. the object start offset remains unchanged, of course also in the case that $y < x + n$. In the latter case, however, the object is truncated at its end. If and how this changes the object semantically is lying in the hands of the user and cannot be recognized or interpreted by `jMeta`.
- *`replace`:* Replacing n bytes by $m > n$ bytes has the same effect to existing objects as an `insert`, with the very same case distinctions. Likewise, `replace` behaves like `remove` if $m < n$. The case $m = n$, i.e. an *overwrite* operation does not lead to any offset changes of existing objects.

Begründung: The connection between an `IMediumReference` instance and an object is a viable picture and quite illustrates the real use case behind manipulating raw binary data. Due to this design decision, this connection persists - from point of view of the caller - even in case of writing changes to the medium. We cannot burden the user with keeping track of these changes as he would need to manage `IMediumReference` himself in a complex way, listing which operations he has done. This complex book-keeping is what you would expec from a component such as `Media`.
If all bytes of the object are removed, then the `IMediumReference` instance still refers to the "previous" location of the object. The user must not ensure that the `IMediumReference` instance still refers to the same object.

Nachteile: A central management of `IMediumReference` instances must be implemented (see DES 075).

As already indicated by DES 075 the automatic updating of already created `IMediumReference` instances requires that only `Media` may create `IMediumReference` instances. These must be managed in a kind of pool to be able to automatically update them in case of writing operations.

DES 076: **`IMediumReference` instances are centrally managed by `Media` and cannot be directly created by users of the component**
The lifecycle of `IMediumReference` instances is controlled by `Media`. They are created and returned to the user via a factory method.

Begründung: It is strictly required to implement DES 076. Instances that have been created by the user cannot be update automatically, thus we have to ensure the manual creation by the user does not happen.

Nachteile: More complex instantiation of `IMediumReference` instances.

The question *when* to update `IMediumReference` instances has still not been answered yet. The following design decision clearly defines this:

DES 077: **`IMediumReference` instances are only updated after a *flush***
According to DES 077 `IMediumReference` instances are automatically updated in case of medium changes. This automatic update only happens at *flush* time.

Begründung: Assumed that `IMediumReference` instances would already be updated whenever a pending change is registered using *insert*, *replace* or *remove*. In this case the following would be necessary:

- When reading data, this data is not necessarily in a cache. This indicates that reading from external medium is necessary. If all `IMediumReference` instances would reflect a state including any pending changes, they would no longer correspond to the state of the external medium. If you would now want to read or write to the external medium, the real offset on the external medium would need to be "reconstructed" based on the changes made so far, everytime you want to know where current data resides on the medium. This implies a complex coding overhead that would not ease debugging errors or understanding the current state of instances.
- The operation `undo` according to DES 077 requires that offsets would need to be "re-adapted" if a pending changes is undone again. Again this is additional complexity.

If `IMediumReference` instances in contrast are first updated after a *flush*, then no reconstruction of original offsets based on already made changes is necessary.

Nachteile: No disadvantages known

This directly implies the following design decision:

***DES 078:* In `Media`, offset specifications always are offsets on the external medium as it was looking like after the last *flush* or after opening it initially**
For operations of `Media` that take an offset as argument, this offset refers to a location on the external MEDIUM after the last *flush* or the initial opening - in case no *flush* has occurred yet. These offsets must especially be located within the interval $[0, \text{length}]$, where "length" is the current length of the medium in bytes.

Begründung: Naturally follows from DES 078. For users, the offset situation remains stable and logical, he does not need to maintain a history of *insert*, *replace* and *remove* operations. Likewise, the offsets stay stable for the implementation, too: Checking offsets and organisation of internal data structures can be based on this invariant.

Nachteile: No disadvantages known

Semantic of Writing Operations

In DES 078, we have defined the primitive writing operations *insert*, *replace* and *remove* that are essential for `Media`. In the same section, we have listed basic requirements for writing that lead to these operations. The API of `Media` includes their guaranteed behavior. It is very important to define interrelations between these operations, especially how they behave for overlapping offset areas of the medium. These things are defined by the following design decisions. The behaviors are part of the API contract and must be documented as such for the API users.

We start with the operation *insert*:

***DES 079:* *insert* concatenates insertions in call order, the operation is not influenced by any other writing operations**
The temporal order of calls have the following semantics:

1. Step 1: *insert*, Step 2: *insert* at the same offset: *insert* concats, each call with the same offset determines a new, consecutive insertion, i.e. insertions at the same medium location are done with increasing offsets. Let's take two calls to *insert* at the same offset x. Let "Call 1" be the earlier call with insertion length n_1, "Call 2" the later call at x with insertion length n_2. The end result on the medium after *flush* is:
   - At offset x, the insertion data of "Call 1" is located
   - At offset $x + n_2$, the insertion data of "Call 2" is located
2. Step 1: *insert*, Step 2: *remove* at same offset: Do not influence each other.
3. Step 1: *insert*, Step 2: *replace* at same offset: Do not influence each other.
4. Step 1: *insert* at offset x, Step 2: *replace* m bytes by n, starting at offset y, where $x > y + m$, i.e. the insertion does not happen within the replacement region, but behind: These operations do not influence each other. This includes the case that $m > n$, i.e. an override operation with a removal, and $x \in [y, y + m)$, i.e. the insertion does not happen within the replacement region, but behind.
5. Step 1: *insert* at offset x, Step 2: *replace* m bytes by an arbitrary number of new bytes, starting at offset y, where $x \in (y, y + m)$: Put in other words, the bytes to replace contain the prior insertion. Here, the second call is invalid and rejected with an exception.
6. Step 1: *insert* at offset x, Step 2: *insert* at offset $y \neq x$: Do not influence each other.

Begründung: For (1): You could alternatively interpret "insert" as such that the second call inserts data *before* the previous earlier one. However, the design decision says that "insert" inserts before the currently persisted medium byte at the insertion offset. Concatenating allows user code to linearly insert stuff with increasing offsets, which is in most cases the convenient and expected behavior. For (2), (3) and (4): As offsets refer to offsets on the external medium according to DES 079, *remove* and *replace* only affect bytes that are currently persisted on the external medium, they thus cannot remove any pending content of a prior *insert* not yet *flush*ed. For (5): Otherwise complex logic is necessary to mix insertion bytes into replacement bytes. The user can achieve the same by simply including the insertion bytes in the replacement bytes starting at offset y.

Nachteile: For (1): You could alternatively interpret "insert" as such that the second call inserts data *before* the previous earlier one. However, the design decision says that "insert" inserts before the currently persisted medium byte at the insertion offset. With concatenating, user code can linearly insert stuff with increasing offsets, which is mostly the expected behavior.

The operation *remove* behaves as follows:

***DES 080: remove* allows no overlaps, only later calls with bitter region make earlier calls obsolete**
The following temporal order of calls have the described semantics:

1. Step 1: *remove* n bytes at offset x, Step 2: *remove* or *replace* m bytes at offset $y \leq x$ with $y + m \geq x + n$: When calling *remove* twice or first *remove*, then *replace*, where the second call fully contains the offset region of the first, the earlier call is declared as invalid and is not processed during a *flush*. Of course this is also true for all earlier calls, not only the last one.
2. Step 1: *remove* n bytes at offset x, Step 2: *remove* or *replace* m bytes at offset $y \in (x, x + n)$ with $y + m < x + n$: The first call fully contains the region of the second call. The second call is invalid and rejected with an exception.
3. Step 1: *remove* n bytes at offset x, Step 2: *remove* or *replace* m bytes at offset $y \in (x, x + n)$ with $y + m \geq x + n$: The second call is an overlapping call, that probably removes additional bytes behind the first remove call, but still includes the end of the first call. The second call is invalid and rejected with an exception.
4. Step 1: *remove* n bytes at offset x, Step 2: *remove* or *replace* m bytes at offset $y \leq x$ with $y + m < x + n$: In this case there is an overlap from the start. The second call is invalid and rejected with an exception.
5. Step 1: *remove* n bytes at offset x, Step 2: *insert* at offset $y \in [x, x + n)$: As the insert call according to DES 080 refers to offsets on the external medium, here first n bytes present on the medium are removed, then the new bytes are inserted after the last byte persisted, i.e. at offset x.
6. Step 1: *remove* n bytes at offset x, Step 2: *remove* or *replace* at offset y without any overlaps to $[x, x + n)$: Do not influence each other.

Begründung: There are no accidental multiple removals or replaces. For (1): It may happen that the user first removes a chield field, and additionally still before the flush `flush`, he removes the parent block. For (2) to (4): Would we allow the second calls, we would need to change the remove areas of the first calls. This leads to a more complex logic, that is not necessary according to the requirements stated in "11.1.1 Requirements for the Two-Stage Write Protocol". A very important reason to not allow this is implementation of `undo`: Would we allow later changes of already made calls, then `undo` would be quite complex, because it would need to also undo such changes to removed regions, what basically means keeping a history of all changes made for the same region.

Nachteile: No disadvantages known

The operation *replace* behaves like this:

***DES 081: replace* does not allow overlaps, only later calls with bigger reagions make earlier calls obsolete**
The behaviour of *replace* mostly matches the behavior of *remove*, as described in cases 1 to 4 of DES 081. You only have to replace *remove* by *replace* in the description of DES 081 :-). *replace* interacts with an insert the same way as described in DES 081 for the inverse order which are cases 5 to 7:

1. to 4. Exactly the same as described in DES 081 with *replace* as first step.
5. Step 1: *replace*, Step 2: *insert* at the same offset: These operations do not influence each other.
6. Step 1: *replace* m bytes by n, starting at offset y, Step 2: *insert* at offset x, where $x > y + m$, i.e. the insertion does not happen within the replacement region, but behind: These operations do not influence each other. This includes the case that $m > n$, i.e. an override operation with a removal, and $x \in [y, y + m)$, i.e. the insertion does not happen within the replacement region, but behind.
7. Step 1: *replace* m bytes by an arbitrary number of new bytes, starting at offset y, Step 2: *insert* at offset x, where $x \in (y, y + m)$: Put in other words, the bytes to replace contain the later insertion. Here, the second call is invalid and rejected with an exception.
8. Step 1: *replace* at offset x, Step 2: *remove* or *replace* at offset y without any overlaps: Do not influence each other.

Begründung: See DES 081, cases 1 to 4 and DES 081, cases 3 to 5

Nachteile: No disadvantages known

Another problem: How to map the actual data to a data block before a *flush*? The offset alone is not unique anymore in the face of multiple *insert*s at the same offset before their *flush*. If you now want to relate an action (*insert*, *remove*, *replace*) data to its (current or future) offset, the offset alone is not sufficient to state clearly which action happened first at the offset.

An answer is given in the following design decision that explains how action, data and offset are brought into close relation:

***DES 082:* Each of the writing operations returns an instance of a class `MediumAction` to describe the action in more detail**
This class contains following data:

- The kind of action (*insert*, *replace*, *remove*); **Motivation:** The user must be able to know the kind of change
- `IMediumReference` of the action; **Motivation:** It must be clear where the change happened or must happen
- Data of the action (length or bytes to write); **Motivation:** It must be clear what needs to be changed.
- Number of actually affected medium bytes (0 for *insert*, number of bytes to remove for *remove*, number of bytes to replace for *replace*; **Motivation:** For `replace` only one length is not enough as the number of bytes to replace may be different from the length of the replacement bytes.
- Validty: Handle is already persisted by a *flush* or still pending; **Motivation:** Thus it can be clearly state from outside whether the data must still be persisted and thus must be taken from the instance of the user requires to read them, or the data has already been written to the external medium.

Begründung: According to DES 082, the user can only read data that is currently persisted.
Nevertheless it is necessary that application code can also re-read data previously registered for writing, but not yet persisted by a *flush*. That now becomes possible with the `MediumAction` class that is returned by *insert*, *replace* and *remove*. Is the `MediumAction` still pending, the application code can return the pending bytes from the `MediumAction`, otherwise by directly accessing the `Media` read functionality to fetch the currently persisted bytes.
Instances of `MediumAction` can ideally be used for internal data management by `Media`.

Nachteile: No disadvantages known

End medium access

We still need an operation to end the medium access, thus we define:

***DES 083:* Operation *close* ends the medium access and empties the cache, consecutive operations are not possible on the medium**
A user can manually end medium access by calling *close*. It is then no longer possible to access the medium via the closed access way. The user must explicitly reopen the medium to access it again.

Begründung: The implementation works with OS resources such as files that must be closed. Furthermore cache content and other memory is not freed anytime if you could not close the medium.
Closing cannot be implemented automatically but must be explicity called by the user.

Nachteile: No disadvantages known

The public API of medium access

Based on the previous design decisions we now design the public API of the component `Media`. Until now, we only introduced the classes `IMediumReference`, `IMedium` and `MediumAction` as well as some abstract operations to deal with media. How do we offer these operations to users? This is explained by the following design decision.

***DES 084:* Access to a medium is done using the `IMediumStore`**
The reading and writing access to a medium (both random-access as well as byte stream according to DES 084) is offered via interface `IMediumStore` with the operations listed in table 11.3.

Begründung: A further subdivision of functionality into more than one interface is neither necessary nor helpful. It would just be an unneccessary complex API, and despite the single interface, the implementation can still be modularized as needed. The individual operations are motivated in the table itself.

Nachteile: No disadvantages known

Operation	**Description**	**Features and motivation random-access**	**Features and motivation `InputStream`**
`cache()`	Buffers n bytes according to DES 084 and DES 084 permanently in the internal cache, starting at the given offset x. Has effect if caching is enabled.	User code can prefetch data to work on it later, he himself gets the possibility to efficiently read and process data.	If x is smaller than the current highest read offset, a `InvalidMediumReferenceException` is thrown. Additionally: Is it bigger than the current highest read offset, the bytes up to the new offset are read or *skip*ped (depending on current configuration). Afterwards the actually requested bytes are read. This ensures that all bytes are available in principle. The last read offset is advanced correspondingly.
`getCachedByteCountAt()`	According to DES 084 and DES 084, it provides the nuimber of bytes that are consecutively cached at offset x.	See `cache()`. User code must be able to additionally check whether enough data is already buffered or not.	See random-access.

Operation	**Description**	**Features and motivation random-access**	**Features and motivation `InputStream`**
`getData()`	Reads n bytes at offset x with the given mode (`forceMediumAccess` = `true` or `false`) according to DES 084, DES 084, DES 084	With `forceMediumAccess` = `false`: It first tries to read the data from the cache, are they (partly) not available, the missing data is directly read from external medium. With `forceMediumAccess` = `true`: Data is always directly read from external medium, cache is ignored. In both cases, the cache will be updated with data newly read from external medium. Furthermore, only when caching is active, the method actually accesses the cache, in both cases.	`forceMediumAccess` is ignored. If there is data for the given offset in the cache, the data is returned. If it is not, a `InvalidMediumReferenceException` is thrown. Thus, for `InputStream`s, a call to `cache()` is mandatory before reading can actually start.
`isAtEndOfMedium()`	Checks if offset x is at the end of the MEDIUM.	Based on this knowledge, the user code can skip further reading and does not need to check for exceptions.	The provided offset is ignored, it is tried to read bytes from current offset. Is this resulting in return code -1, we are at the end of the stream, otherwise the byte is added to the cache.

Operation	Description	Features and motivation random-access	Features and motivation `InputStream`
`skip()`	Skips n bytes according to DES 084	Simply does nothing, as skipping does not give any advantages here.	Skips n bytes by calling `InputStream.skip(n)`. It allows users to ignore unimportant bytes explicitly, to e.g. save memory when caching is active. The last read offset is advanced correspondingly.
`discard()`	Removes up to n bytes from the cache starting at offset x according to DES 084. Allows user code to free up memory to avoid unnecessarily allocated heap.	Buffering and reading data again with `getData()` is always possible.	After freeing up memory with `discard()`, the attempt to access freed offset ranges for an `InputStream` using `getData()` leads to an `InvalidMediumReferenceException`.
`insertData()`	Implements the writing operation *insert* according to DES 084, DES 084 and DES 084: Adds data at the given offset, consecutive bytes are shifted "to the back", the changes first get written only with `flush()`.	The insertion of new metadata is a common case in `jMeta` and must be supported.	`ReadOnlyMediumException`

Operation	**Description**	**Features and motivation random-access**	**Features and motivation `InputStream`**
`removeData()`	Implements the writing operation *remove* according to DES 084, DES 084 and DES 084: Removes n bytes at offset x. Consecutive bytes are shifted "to front", the changes first get written only with `flush()`	Removing existing metadata is a standard case in `jMeta` and must be supported.	`ReadOnlyMediumException`
`replaceData()`	Implements the writing operation *replace* according to DES 084, DES 084 and DES 084: Replaces n bytes at offset x with new bytes of length m. The changes first get written only with `flush()`	It often happens that – instead of entirely removing or newly inserting data – existing data must be overwritten. This is - especially at the beginning of a file - a much more efficient operation than insertion and deletion and must thus directly be supported.	`ReadOnlyMediumException`
`flush()`	Implements the writing operation *flush* according to DES 084: All *changed* data currently in the temporary buffer are written in a suitable way to the external medium. It cannot be guaranteed that this operation is atomic.	This is a practical implementation of DES 084: While `insertData()`, `removeData()` and `replaceData()` only write into a temporary buffer, this call directly writes to the external medium.	`ReadOnlyMediumException`
`createMediumReference()`	Creates a new `IMediumReference` instance for the given offset x according to DES 084	Is needed for random-access	Is needed for `InputStreams`

Operation	**Description**	**Features and motivation random-access**	**Features and motivation `InputStream`**
`undo()`	Undoes the changes of the given `MediumAction` according to DES 084, as far as it is still pending.	Is needed for random-access	`InvalidMediumActionException`
`close()`	Closes all internal resources according to DES 084, clears the complete cache contents and other internal data structures	See DES 084	See DES 084

Table 11.3.: Operationen der `Media` API

The component interface

How are the public functions exposed to the outside world? There must be a functionality that creates a `IMediumStore` instance for a given `IMedium`. The API for this looks as follows:

DES 085: **`IMediaAPI` is the central entry point with creation functions for `IMediumStores`**
The interface `IMediaAPI` offers the central entry point for the component `Media`. Using the method `createMediumStore()`, users can create an `IMediumStore` instance.

Begründung: The necessity for a further interface in addition to `IMediumStore` is clear enough: A `IMediumStore` refers to just a single "medium access session" for a medium, and of course users want to be able to open multiple media at the same time using `Media`. Pushing creation functions into `IMediumStore` is not considered good practice as it would decrease comprehensibilty.

Nachteile: No disadvantages known

Error Handling

In general violations of the interface contract according to DES 085 are acquitted with a runtime exception.

The following table summarizes all further error situations when working with `Media`:

Error Scenario	**Description**	**Reaction** `jMeta`	**API method**
Medium does not exist	The medium does not exist, in `jMeta` however it must be an existing medium.	It is an abnormal situation, thus a `MediumAccessException`runtime exception is thrown.	`IMediaAPI .createMediumStore()`
Medium is already locked	The medium is already locked by another process. `jMeta` cannot work with the medium. It is the burden of the caller to ensure, that the medium is not used in parallel.	It is an abnormal situation, thus a`MediumAccessException`runtime exception is thrown.	`IMediaAPI .createMediumStore()`
Unknowm media type	A `IMedium` implementation is specified by the caller which is unsupported.	This is an abnormal situation violating the interface contract, thus the same exception as for contract violations is thrown.	`IMediaAPI .createMediumStore()`

Error Scenario	**Description**	**Reaction** `jMeta`	**API method**
End of medium during reading	Of course each medium has an end sometimes. When reading, this end can be reached. When writing, this is not actually possible - there, we assume that all output media virtually are unlimited. If there is no more memory for writing, `Media` usually reacts with a `MediumAccessException` following an `IOException`. It cannot be requested from the calling code during reading, that it knows where the end of the input medium is. Reaching its end during reading can be an error, but it needs not - this depends on the current usage situation. The calling code thus must handle this depending on current context.	Because it is not necessarily an abnormal situation, a `EndOfMediumException` checked exception is thrown.	`IMediumStore .cache()`, `IMediumStore .getData()`
Timeout during reading	A reading call fo the medium does not return after a configured timeout.	It is an abnormal situation, thus a `ReadTimedOutException` runtime exception is thrown.	`IMediumStore .cache()`, `IMediumStore .getData()`

Error Scenario	Description	Reaction jMeta	API method
Write to read-only medium	The user-provided `IMedium` implementation is read-only, thus it only allowes read access.	If the user nevertheless tries to write, this is an abnormal sitation and is signalled by a `ReadOnlyMediumException` runtime exception.	`IMediumStore .flush()`, `IMediumStore .insertData()`, `IMediumStore .removeData()`, `IMediumStore .replaceData()`
Consecutive write calls overlap	Concecutive calls to `removeData` or `replaceData` before a `flush` overlap in an invalid way (see DES 085 and DES 085)	Is acquitted with an `InvalidOverlappingWriteException` runtime exception.	`IMediumStore .removeData()`, `IMediumStore .replaceData()`
Invalid cache offset	With `cache()` it is tried for an `InputStream` to cache an offset that is smaller than the last read offset.	This is a wrong usage of the API, thus it results in an `InvalidMediumReferenceException` runtime exception.	`IMediumStore .cache()`
Stream data not available	Data requested with `getData()` for a given offset are not any more available in the cache, and the underlying medium is an `InputStream`.	This is a wrong usage of the API, thus it results in an `InvalidMediumReferenceException`, a runtime exception.	`IMediumStore .getData()`

Error Scenario	**Description**	**Reaction** `jMeta`	**API method**
Unknown or invalid `MediumAction`	The user passes an unknwon or invalid `MediumAction` to any operation	This is an abnormal situation and is acquitted with the runtime exception `InvalidMediumActionException`	`IMediumStore.undo()`
`IOException` in the implementation	The Java implementation use throws an `IOException`, at any place where none of the already presented error situations are involved.	It is an abnormal situation, thus a `MediumAccessException` runtime exception is thrown.	`IMediumStore .cache()`, `IMedium-Store .getData()`, `IMediumStore .isAtEndOfMedium()`, `IMediumStore .flush()`
The `IMediumStore` was already closed using `close()`	`MediumStoreClosedException`, a runtime exception	`MediumStoreClosedException`, a runtime exception	all

Table 11.4.: Error handling in the component `Media`

To summarize:

***DES 086:* Reaching the end of medium is not necessarily an error, other problems with I/O are seen as abnormal events.**
`Media` sees achieving at the end of a medium not as abnormal event, but it must be handled according to the current context by the user code. `jMeta` assumes output media to be virtually unlimited and thus does not implement any means of treating end of medium situations when writing (also in accordance to the Java API).
All other error sitations in `Media` are abnormal situations according to table 11.4.

Begründung: See table

Nachteile: No disadvantages known

11.1.3. Implementation Desing

Medium Access

Access to the MEDIUM is implemented in own classes, as the following design decision states:

DES 087: **Interface `IMediumAccessor` with implementationen for MEDIEN access**
Access to a MEDIUM is possible via an interface `IMediumAccessor` with following primitives:

- Open: Opening creates an exclusive lock on the medium, as far as the medium supports this (see DES 087)
- Check if opened
- Close
- Read from offset x: Read n bytes from external medium by explicit access, return the bytes in a `ByteBuffer`, the offset x is ignored for non-random-access media
- Write to offset x: Schreibt n bytes, die in einem `ByteBuffer` übergeben werden, nicht implementiert bei byte-Stream-media (der Aufruf wird ignoriert)
- Truncate to length n: Truncates the medium to a new length n; not implemented for byte-Stream media (call is ignored)
- Check if current offset is at end of MEDIUMS, x is ignored for non-random-access media
- Skipping bytes: Skips n bytes, for random-access media, it does not do anything

`IMediumStore` exclusively accesses the MEDIUM only via this interface.

Begründung: `IMediumStore` itself can deal with caching and the complex implementation of writing functionality independently from the concrete medium, while the actual medium access can be abstracted away by concrete implementations generically. Classical separation of concerns to increase maintainability and comprehensibility of the solution.

Nachteile: No disadvantages known

Management of `IMediumReference` instances

According to DES 087, `IMediumReference` instances must be maintained centrally by `IMediumStore`. At the same time, we can have multiple `IMediumReference` instances referring to the same offset of the same medium, but actually refer to different data - in the special case of the methode `insertData`. New data is inserted subsequently at the same offset with `flush()`, see DES 087. In the end, the `IMediumReference` instances must be automatically updated with `flush()`, as described in DES 087 and DES 087.

Thus, these things follow:

DES 088: **No pooling of `IMediumReference` instances for the same offsets is possible**
In contrast to Java strings, the reuse of `IMediumReference` instances in `createMediumReference()` for same offsets is not possible, i.e. if an `IMediumReference` instance for offset x has already been created, the same instance cannot be returned for the same offset on the next call. A new `IMediumReference` instance must be created instead.

Begründung: Assum we use pooling. Furthermore, assume there are two inserts of lengths n_1 and n_2 at the same offset x scheduled via `insertData()`. The internal implementation would only have a single instance of `IMediumReference` for offset x. `flush()` must keep the offset of the first insertion unchanged, as the data of this first insertion remain there. Howeder, the offset of the second insertion must be changed, as these inserted bytes are actually located at offset $x + n_1$ after `flush()`.

Nachteile: For many created `IMediumReference` objects, there could be a bigger memory footprint.

Furthermore, the `IMediumReference` objects must be maintained in a dedicated data structure:

DES 089: **All instances ever created with `createMediumReference()` are maintained in a dedicated data structure that allows duplicates**
All instances are held in a data structure that allows duplicates. It must be dedicated, i.e. only be used for storing all ever created `IMediumReference` objects.

Begründung: Dupllicates must be possible due to DES 089. We cannot mix that data structure with the caching or pending change data structures, as on the one hand side there will be always more `IMediumReference` instances, than cache entries or pending changes, and on the other hand cache and pending change list could be cleared, while the `IMediumReference` instances must be kept until the explicit close of the mediums.

Nachteile: No disadvantages known

Are these entries of the dedicated data structure indicated by DES 089 in any way sorted?

DES 090: **Unsorted ArrayList for keeping the `IMediumReference` instances**
We use an `ArrayList` as data structure for maintaining all `IMediumReference` instances. Insertion of `IMediumReference` instances is done in creation order. The list is unsorted.

Begründung: A list allows for duplicates. Sorting of the list after each addition would lead to $O(n \log n)$ runtime complexity on average when creating a new `IMediumReference` instance, if n is the current size of the list. Ascending sort order by offset would be a bit more efficient for `flush()`, but does hardly justify the effort during insertion, because the creation of a `IMediumReference` instance is usually much more commonly done than a `flush()`. Following approaches have been rejected especially:

- A `Map<Long, List<IMediumReference>>` with offsets as key and all `IMediumReference` instances for the offset as value won't work, as the offsets would need to be shifted on each insertion, i.e. the keys of the map would either need to be `IMediumReference` instances again, or would need to be updated expensively.
- A `TreeSet<IMediumReference>` can be excluded as it does not allow duplicates.
- A combination of a `TreeSet` with a special `Comparator`, which assumes `IMediumReference` instances only to be equal, if they are the same objects, and as bigger, if the offset is the same, but the object is different, would accept such "duplicates" with the same offsets and ensure correct sorting for every insert. However, the `Set` then is incompatible with `equals`, what is not a best practice according to the javadocs. Secondly: More time required for insertion as mentioned.

Nachteile: Finding all `IMediumReference` instances within the `flush()` implementation, that are bigger than a given offset has $O(n)$ complexity, which can however be tolerated.

To handle the complexity of `IMediumStore` we decide:

***DES 091:* The class `MediumReferenceFactory` is used for maintaining `IMediumReference` instances**
`MediumReferenceFactory` implements the design decisions DES 091, DES 091 as well as DES 091. It offers following methods:

- `createMediumReference()` to create `IMediumReference` instances
- `updateReferences()` for implementation of DES 091, where a `MediumAction` instance is passed
- `getAllReferences()` returns all maintained instances
- `getAllReferencesInRegion()` returns all maintained instances with given offset range
- `getAllReferencesBehindOrEqual()` returns all maintained instances with offsets bigger than the given offset
- `clearAll()` removes all maintained instances

Begründung: Reduction of total complexity of the `IMediumStore`.

Nachteile: No disadvantages known

The last question for `IMediumReference` instances that still has to be answered: How to deal with the method `advance()` mentioned in DES 091. It creates `IMediumReference` instances. Do they have to be maintained in `MediumReferenceFactory`, too?

***DES 092:* The `IMediumReference` instances created with `IMediumReference.advance()` need to be maintained with `MediumReferenceFactory`, too.**
Initially, the new `IMediumReference` instance must get a reference to its creator, i.e. `MediumReferenceFactory`, during construction. To still enable a simple creation of instances of `IMediumReference` implementation classes (e.g. in unit tests), the constructor is public.

Begründung: The client code can arbitrarily use `IMediumReference.advance()`, and the returned references can be used as usual, e.g. for newly created data blocks. Thus it is clear that even these references are auto-corrected with `MediumReferenceFactory.updateReferences()`.

Nachteile: Close coupling between `IMediumReference` and `MediumReferenceFactory`, as both are knowing each other.

Open Issue 11.1: **– Check: Use Weak References?**
Check: Use Weak References?

Internal Data Structures for Caching

The cache on first sight matintains data bytes per offset, always assuming that the data in cache is exactly identical to the data on the external medium, according to DES 092. It was already defined, that there is an explicit `cache()` method for eading data into the cache and a method `getCachedByteCount()` for querying connected bytes from cache (see table 11.3, as well as DES 092 and DES 092). Query data from the cache is done using `getData()`, which can on the one hand skip the cache, on the other hand it automatically updates the cache with missing data (DES 092 and DES 092). Finally, cache contents can be freed using `discard()`, or caching can even be entirely inactive (see DES 092).

In most cases we want to query which parts of data to read, write or remove is within the cache. The cache might be fragmented arbitrarily due to multiple fill operations. Thus we need a class for this:

DES 093: **Class `MediumRegion` for consecutive regions of a mediums, especially also for cache areas**
The class `MediumRegion` represents regions of a mediums, that are cached under some circumstances. For this, it offers the following methods:

- `getStartReference()`: Query start `IMediumReference` of the region
- `getSize()`: Query length of the region
- `isCached()`: Returns true if cached, false otherwise
- `getBytes()`: Returns null if the region is not cached, otherwise the `ByteBuffer` with the cached region data
- `isContained()`: Returns true if the given `IMediumReference` is contained within the region, false otherwise
- `discardbytesAtFront()`: Allows to trim the region at its beginning by discarding a given number of bytes
- `discardbytesAtEnd()`: Allows to trim the region at its back by discarding a given number of bytes

Begründung: The cache regions and the non-cached regions can be treated in the same way

Nachteile: No disadvantages known

Maintenance of cache content is done by a helper class:

***DES 094:* Cache maintenance by `MediumCache`**
Cache maintenance is done by the class `MediumCache`, with following methods:

- `getCachedByteCountAt()`: Returns the number of bytes cached consecutively starting from offset x
- `getData()`: Returns a list of `MediumRegion` instances for the offset range $[x, x+n]$, which represent the cached and non-cached ranges within the offset range. I.e. whenever there is a gap in the cache, this gap is also represented by a single `MediumRegion` instance without data
- `getCachedRegions()`: Returns a list of all cached `MediumRegion` instances
- `addData()`: Adds data to the cache at offset x. Previously cached data is overridden.
- `discardData()`: Frees up to n bytes at offset x from the cache
- `clearAll()`: Frees all data in the cache
- `setMaxRegionSize()`: Stes the maximum region size. Default is `Integer.MAX_VALUE`. The created regions will have at most the given size. Any already existing regions remain unchanged.
- `getMaxRegionSize()`: Returns the maximum size of a region
- `enableCaching()`: Enables or disables caching
- `isCachingEnabled()`: Queries if caching is enabled or disabled

Begründung: Reducing complexity of `IMediumStore`

Nachteile: No disadvantages known

How are these regions internally managed?

***DES 095:* A `TreeMap` is used for managing the cache contents**
A `TreeMap<IMediumReference, MediumRegion>` is used for managing the cache contents.

Begründung: Content must be read based on offsets. Thus a data structure sorted by offset is necessary. It allows the efficient retrieval of all `MediumRegions` bigger or smaller than a given offest. This operation should most probably need a runtime complexity of only $O(log(n))$ instead of $O(n)$.

Nachteile: No disadvantages known

Due to performance reasons, we already considered a possibility to free up data from the cache. It, however, shows, that the maximum size of the cache could

additionally be managed by an automatic mechanism:

***DES 096:* The user is able to set the maximum cache size**
The maximum size for the `MediumCache` can be set via `setMaxCacheSize()` and retrieved via `getMaxCacheSize()`. Furthermore the current cache size can be retrieved using `getCurrentCacheSize()`. By default, the maximum cache size is not limited, which is represented by the constant `UNLIMITED`. Setting a maximum cache size leads to freeing up or shrinking of existing cache regions, i.e. it will be directly applied. Which cache regions get freed, is undefined. `addData()` checks wheter the current cache size plus the bytes to add exceed the maximum cache size. If so, only as much bytes starting from begin of the `ByteBuffer`s to cache are added up to the maximum size, which might probably be 0 bytes in total. `addData()` retzrns a `MediumRegion` which reflects the actual region size cached.

Begründung: This allows the caller to actively influence the cache size and avoid `OutOfMemoryError`sn

Nachteile: No disadvantages known

We should also think about cache fragementation. Let us assume that, by subsequent calls to `addData()`, we would add a single byte to the cache 20 times for a consecutive offset range. Will this result in 20 different `MediumRegion`s with a length of one byte each? The same question arieses for calls to `getData()`, which spans over an offset range with gaps in the cache coverage. Assume we have a cache that contains 20 bytes starting at offset x, further 50 bytes at offset $y := x + 30$, i.e. it has a gap of 10 bytes between the two regions. If there is a now a call to `getData()` for range $x - 10$ with length of 100, we would result in five regions:

- Region 1: $[x - 10, x)$ not in the cache
- Region 2: $[x, x + 20)$ in the cache
- Region 3: $[x + 20, y)$ not in the cache
- Region 4: $[y, y + 50)$ in the cache
- Region 5: $[y + 50, y + 60)$ not in the cache

`getData()` will then add regions 1, 3 and 5 to the cache, according to DES 096. So, do we have 5 `MediumRegion`s in the cache after the call?

We decide:

***DES 097:* Fragmentation of subsequent cache regions is avoided**
For subsequent calls of `addData()`, for which the first call covers offset range $[x, x+n)$ and the second call covers offset range $[x+n, x+n+m)$ - i.e. both ranges are directly connected without gaps - the result is a single `MediumRegion`, covering offset range $[x, x+n+m)$.
A call to `getData()`, whose offset range is covering multiple cached `MediumRegion`s divided by gaps, creates a new single `MediumRegion`s, joining the previously separate `MediumRegion`.
In both cases we assume that the total length of the new region is smaller than the maximum configurable region size according to DES 097. If this size is exceeded: Let n be the total size with configurable maximum region size m, and let $n > m$. If n is divided by m without remained, then the result will be $\frac{n}{m}$ different `MediumRegion`s. If n cannot be divided by m without remaineder, we have $\lfloor \frac{n}{m} \rfloor + 1$ `MediumRegion`s.
A fragmentation of consecutive `MediumRegion`s thus only happens when the maximum region size is exceeded.

Begründung: More objects need more memory and cause a higher management effort. Fragmentation does not offer any benefits.

Nachteile: No disadvantages known

A related question is how to treat gaps in general, even for multiple subseqhent calls of `addData()`: AS shown in the initial example, we load 20 bytes with 20 calls to `addData()` into the cache. However, there is a gap of ine Byte between each of them, and this gap is not contained in the cache. Even here you have to ask: Will this create 20 `MediumRegion`s?

***DES 098:* Fragmentation of unconnected cache areas is not actively avoided**
`Media` does not try to mitigate or avoid the situation of "nearby" but unconnected cache regions of small sizes.

Begründung: It would lead to even higher complexity of the implementation of `addData()`. Heuristics that tell how to join those unconnected regions must be defined first. It could e.g. be possible for a call to `addData()` to detect that there are other offset ranges in a given radius $[x-k, x+k]$, with a to-be-defined constant k. In that case, the gap bytes could be additionally cached. However, it still must be ensured that the maximum region size and maximum cache size is not exceeded. For `InputStream`s, this "gap filling" is not possible and thus must anyway not be done. Last but not least, the functionality must respec the end of medium.
In contrast to that high complexity, we can ask the caller to ensure the he only adds data for reasonably connected offset ranges.

Nachteile: No disadvantages known

Internal Data Structures for Managing Pending Changes

For the two-stage write protocol, changes scheduled via `insertData()`, `removeData()` and `replaceData()` must be managed in a reasonable way, such that they later can be processed during `flush()`. Any change is represented as a `MediumAction`.

We first state the following:

DES 099: `MediumAction` **has a sequence number for distinguishing actions at the same offset**
`MediumAction` defines a sequence number (starting at 0) for distinguishing actions at the same offset. It is incremented by consecutive actions (no matter which type) at the same offset by 1. This sequence number is not thus only used for `insert`s at the same offset, but also for all other types of actions.

Begründung: Two distinct `MediumAction` instances referring to the same offset cannot be sorted without this mechanism. However, this is especially important for `insert`s that are allowed to refer to the same offset (see DES 099). Furthermore, it is also important for other types, as `insert` allows a later `remove` or `replace` at the same offset (see DES 099).

Nachteile: No disadvantages known

We now have to define comparisons between two `MediumAction`s in a reasonable way, as sorted data structures for efficient retrieval and iteration in order can be used correspondingly:

***DES 100:* Comparisons of `MediumActions` is done based on `IMediumReference` and the sequence number**
A `MediumAction a` is smaller than another `MediumAction b` according to Javas `compareTo`, if and only if one of the following criteria are met:

- The `IMediumReference` of `a` is smaller than the `IMediumReference` of `b`,
- Or the `IMediumReference` of `a` is equal to the `IMediumReference` of `b`, and the sequence number of `a` is smaller than the sequence number of `b`

A `MediumAction a` is equal to a `MediumAction b` according to Javas `equals` and `compareTo`, if all attributes of `a` are equal to all attributes of `b` (in terms of `equals`).
A `MediumAction a` is bigger than a `MediumAction b` according to Javas `compareTo`, if `a` is neither smaller than `b` nor equals `b`. This is especially true if the `IMediumReference` is bigger, or for equal `IMediumReference`s, if the sequence number is bigger. Furthermore it is defined: If `IMediumReference`s and sequence numbers are identical, then `a` is still bigger than `b` in the case that any of the other attribute of the `MediumAction`s differs.

Begründung: Sorting of `MediumAction`s should be done according to their order on the medium (i.e. their `IMediumReference`s) and creation order (i.e. their sequence number), because behaviour of consecutive operations is based on call order, according to DES 100, DES 100 and DES 100. In addition, the `MediumAction`s must be processesed in a defined order during `flush`. `equals` must return true exactly in the case that `compareTo` returns 0.

Nachteile: Might cause confusion, as smaller and bigger are not symmetric. The implementation must thus ensure that `MediumAction`s with same offset always get different sequence numbers that strictly increase with creation order.

Now we can define in which way the `MediumAction`s are stored:

***DES 101:* `MediumActions` are stored in a sorted data structure without duplicates**
The `MediumAction`s created before a `flush()` are held in a datastructure sorted by offset and sequence number (according to DES 101), which does not allow duplicates (e.g. `TreeSet`).

Begründung: The sort order is necessary for in-order-processing via `flush()`, two `MediumAction`s with same offset, same sequence number and same type should not show up twice.

Nachteile: No disadvantages known

Now regarding management of `MediumAction`s:

***DES 102:* For managing `MediumActions`, the class `MediumChangeManager` is responsible**
For managing `MediumActions`, the class `MediumChangeManager` is defined, which internally implements DES 102, with following methods:

- `scheduleInsert()` for scheduling an *insert*, implements DES 102, gets a `MediumRegion` as parameter and returns a `MediumAction` of the given type and with correct sequence number, i.e. should there be other actions at the same offset, the new action is guaranteed to have a sequence number higher than the action with the biggest sequence number already present for the same offset.
- `scheduleRemove()` for scheduling an *remove*, implements DES 102, gets a `MediumRegion` as parameter and returns a `MediumAction` of the given type and with correct sequence number, i.e. should there be other actions at the same offset, the new action is guaranteed to have a sequence number higher than the action with the biggest sequence number already present for the same offset.
- `scheduleReplace()` for scheduling an *replace*, implements DES 102, gets a `MediumRegion` and the length of the range to replace as parameter, returns a `MediumAction` of the given type and with correct sequence number, i.e. should there be other actions at the same offset, the new action is guaranteed to have a sequence number higher than the action with the biggest sequence number already present for the same offset.
- `undo()` for undoing actions, implements DES 102
- `iterator()` returns an `Iterator<MediumAction>` for reading traversation of the changes in correct order, `Iterator.remove()` is not implemented
- `clearAll()` removes all changes

Here, the three `schedule` methods create `MediumActions` according to DES 102 and DES 102.

Begründung: Reduction of overall complexity of `IMediumStore`.
The iterator allows reading of changes in order, but is not needed for processing, as we see later. `remove` on this iterator is not necessary, as `undo()` can undo an action.

Nachteile: No disadvantages known

At the end, we highlight some commonalities between `MediumActions` and `MediumRegions`, and define:

DES 103: **`MediumAction` aggregates a `MediumRegion` instance**
`MediumAction` aggregate a `MediumRegion` instance which holds that offset and length of the action, BUT NOT the bytes related to the action itself, which are held in a separate attribute of the `MediumAction`.

Begründung: `MediumAction` needs a start `IMediumReference`, a length of the change as well as possibly the bytes to change, if any. However, we only implement start and length in form of a `MediumRegion` instance. You can interpret `MediumAction` as a class that refers to a `MediumRegion`. It is a classical "has a" instead of an "is a" relationship, which requires aggregation instead of inheritance. The reason to not keep the bytes in the aggregated `MediumRegion` but in the `MediumAction` itself lies in the special form of the `replace` operation. For detecting non-allowed overlaps, only the number of bytes to replace is important (see DES 103 and DES 103) and not the replacement bytes themselves. In that way, we can get a common implementation of overlap detection.

Nachteile: No disadvantages known

Implementation of flush

The most complex functionality of `IMediumStore` is `flush()`, because:

- `flush()` must iterate all pending changes,
- align them with the current cache content,
- cut reasonable blocks of the data bytes to write,
- update all `IMediumReference` and `MediumAction` instances,
- update the cache

We can add that the writing operations have some additional oddities:

- `insertData()` and `removeData()` require that data behind the insertion or removal offset is read and written again
- `replaceData()` should also not be underestimated, as depending on the number of existing bytes to replaces with a different number of new bytes, it can either lead to no shifts at all (number of bytes to replace equal to number of replacement bytes), to an `insert` (number of bytes to replace smaller than number of replacement bytes) or a `remove` (number of bytes to replace bigger than number of replacement bytes)
- If these operations are done at the beginning of files, reading data up to the end of file is possibly not possible in one large chung, as it could lead to `OutOfMemoryError` for large files
- Thus blockwise reading is necessary

- At this point, the writing operations differ from each other:

  - For an `insertData()` of n bytes at offset x, data must be read and written block-wise starting from the end of medium down to offset x. I.e. first the last k bytes of the medium at offset r are read and then written to offset $r + k$, then k bytes at offset $r - k$ are read, to be written at r and so on until offset x. Another way is not working if you do not want to overwrite and thus lose existing bytes.

  - For `removeData()` of n bytes at offset x, we must read and write data starting at offset $x + n$ block-wise until the end of mediums. First, k bytes at offset $x + n$ are read and then written at x, then k bytes at offset $x + n + k$ are read to be written to $x + n$ and so on, until the last offset on the medium.

  - For `replaceData()` we have to distinguish corresponding cases, it could either behave like `insert` or `remove`, or a simple overwrite (if number of bytes to replace equals number of replacement bytes).

We can first confirm that the configuration of a maximum block size for writing is necessary:

***DES 104:* A maximum write block size must be configurable for the user**
It must be between 1 and N (any integer) bytes

Begründung: Necessary due to the block-wise reading and writing operations behind affected regions caused by inserts and removes. By making it configurable, the user can himself decide how much bytes should be processed in a single pass.

Nachteile: No disadvantages known

Finding out wich operations must be executed during a flush is a complex task. We need to perform this complex task within a method:

DES 105: **`createFlushPlan()` creates a read-write-plan for a flush in form of a `List<MediumAction>`**
`createFlushPlan()` creates a read-write-plan for a flush and returns a `List<MediumAction>`. The read-write-plan contains the actions to be executed in the given order. The list of possible actions is extended by `READ`, `WRITE` and `TRUNCATE`. Which are defined as:

- READ primitive reading of n bytes starting at offset x
- WRITE primitive writing (i.e. overwriting) of n bytes starting at offset x
- TRUNCATE explicit shortening of a file, which is especially necessary for removing data

Each returned `READ` action must be followd by a `WRITE` action, otherwise the plan is invalid. `INSERT` and `REPLACE` operations (if more replacement bytes than bytes to replace) lead to `WRITE` actions. For all `READ` and `WRITE` actions: the number of bytes is between 0 and the (configured) maximum write block size. `createFlushPlan()` also returns the original `REMOVE`, `REPLACE` and `INSERT` actions explicitly in the plan, although they are implemented implicitly by `READ` and `WRITE` actions.
The read-write-plan thus contains `MediumActions` in addtion to those caused by scheduling actions by a user. These additional actions are, however, not added to the internal data structures of the `MediumChangeManager`.
The created plan can be processed afterwards.

Begründung: Determining the necessary operations is a complex process, which should be done separately. A direct execution of the plan would be an alternative, but testability of the code would heavily suffer.
`MediumChangeManager` is the correct place for this operation, as here all `MediumActions` are managed anyways. The additional `MediumActions` in the plan are not added to any internal data structures to ensure the operation is stateless and ideally repeatable.
Adding another `WRITE` primitive seems unnecessary, as there is already an operation named `REPLACE`. However, `WRITE` differs insofar as the bytes to write are not yet known when the action is created, in contrast to `REPLACE`.

Nachteile: No disadvantages known

Now we have all credentials to imnplement writing in `flush()`:

1. Create the read-write-plan according to DES 105
2. Iterate all entries of the read-write-plan, the following steps are done per each entry:
3. Execute the `MediumAction` at the indicated offset by the following steps:
   - If action = `READ`: Read n bytes - that are written in the subsequent action - where n is smaller than the maximum write block size and

bigger than 0. For that, first all regions are determined using `MediumCache.getData()`. Those that are not cached are read by direct access to the medium via `IMediumAccessor`. Then a corresponding `ByteBuffer` is built up step-wise.

- If action = `WRITE`: Write the bytes which where read by the previous `READ` operation with direct access to the medium via `IMediumAccessor`
- If action = `REPLACE`: Write the bytes in the `MediumAction` with direct access to the medium via `IMediumAccessor`
- If action = `INSERT`: Write the bytes in the `MediumAction` with direct access to the medium via `IMediumAccessor`
- If action = `REMOVE`: Ignore the action, as it implicitly gets implemented by `READ`, `WRITE` and `TRUNCATE`
- If action = `TRUNCATE`: Execute an explicit truncation of the medium.

4. Update the cache (part 1): If action = `REMOVE`, then call `MediumCache.discardData()` to remove the bytes from the cache.

5. Only if action = `INSERT` or if action = `REMOVE`: Call `MediumReferenceFactory.updateReferences()` for the region, such that all consecutive `IMediumReference` instances are updated.

6. Update the cache (part 2): If action = `WRITE`, `INSERT` or `REPLACE`, then call `MediumCache.addData()` to add the bytes to the cache.

7. If action = `INSERT`, `REMOVE` or `REPLACE`: Call `MediumChangeManager.undo()` to remove the action

DES 106: **`flush()` is implemented according to the process defined above**
`flush()` is implemented according to the process defined above

Begründung: The read-write-plan must contain all operations explicitly, i.e. including those triggered by the user - `REPLACE`, `INSERT` and `REMOVE`, even if `READ` and `WRITE` would be sufficient for their implementation. The reason for that is that the actions of the user must explicitly be removed from the `MediumChangeManager` and their influence on `IMediumReference` instances behind must explicitly be executed. For this, you need the concrete types, `WRITE` is not sufficient.
`undo()` is only executed for user operations, as only those are maintained in the internal data structures of `MediumChangeManager`, according to DES 106.
The cache update is divided into two parts: In case of a `REMOVE`, *first* the cache is updated, and *then* the `IMediumReference` instances are updated. The reaseon for that is that otherwise, at the same offset, there could be multiple cache regions – for example, if you have a remove of 10 bytes at offset 0 and an insert at offset 10, if you update the medium references first, the cache would contain the previous (to be removed) region from 0 to 10, and a new region from 0 to 10 (the newly inserted one, as the offsets were updated). This would bring the cache into an inconsistent state. With the same reason, the other writing operations are only updated in the cache (part 2) *after* the update of the `IMediumReference` instances.
The cache management is completely done within `MediumCache.addData()`, such that maximum size and consecutive reagions can be optimized there.

Nachteile: No disadvantages known

Let us think about the error handling of this process:

- Is there a failure during creation of the read-write-plan (step 1), the user can try again, as all changes are still existing and nothing was changed yet (at least nothing by the current OS process and using `jMeta`)
- Is there a failure during access to the external medium (step 3), then previously, some actions of the read-write-plan might already have been executed successfully, the external medium thus was already changed. This corresponds to what DES 106 says. As the previous operations were already removed from the plan and other data structures, we have a clean "recovery point", at least, i.e. the user can retry the flush.
- Is there a failure in updating the cache in steps 4 or 6, neither the action is removed, nor the medium references are updated. Similar things happen if something goes wrong in step 5. In these cases, either the `MediumCache` or the `MediumReferenceFactory` is in an inconsistent state, basically it is out-of-sync with reality. The question is: Is there anything left to be saved? This is clarified by the following design decision.

***DES 107:* Undo of changes always happens, cache is emptied in case of update failures**
Should a failure of the `flush()` occur during steps 4 to 6, i.e. any `Exception` is thrown, the action must nevertheless be undone.
Further measures are not taken, because faulure in steps 4 to 6 might only happen in case of programming errors or system failures. In these situations, recovery is anyway quite hard or even impossible.

Begründung: `flush()` is a reasonable operation, that does however not support ACID, but at least the whole system is left in as sane state as possible. This includes that actions already executed successfully are removed such that they are not executed again by mistake.
The attempt to also handle failures during cache management or `MediumReferenceFactory` accesses has a very high complexity that is not justifiable compared to the scarcity of such events. Even then the system is actually not in a consistent state. Such error handling that believes it can rescue the overall system from failure probably makes things even worse.

Nachteile: No disadvantages known

Implementation of `createFlushPlan`

The next step is to go into more detail regarding implementation of `createFlushPlan`, as it is anything but trivial. Here we develop a general algorithm that creates based on operations `insert`, `remove` and `replace` a sequence of necessary read and write operations to transform the current medium state into the target state.

Before that, we list some testcases that should be implemented to demonstrate the intended behaviour:

ID	Testcase	Variations	Expectation
CF0	No operation	-	The plan created is empty
CF1	Single `insert`	a. At start, intermediate or end offset of the medium b. bytes behind: None, or whole-numbered multiple of the maximum write block size, or no whole-numbered multiple, or fewer bytes than the maximum write block size c. Insertion bytes: The same cases as for the bytes behind	No read/write operations before insert offset; bytes behind remain unchanged and get shifted by k bytes towards increasing offsets, such that medium length grows by k bytes; The CFP contains N pairs of read/ write operations for the bytes behind, where N is the number of started blocks of at most "maximum write block size" bytes that are located behind the insert offset; the read/write pairs start from the last block backwards down to the first block after the insert offset, where the writes occur exactly k bytes behind the reads; After this sequence, for the insertion bytes, there are X corresponding write operations (one per starting block) with increasing offsets starting at the insert offset; The `insert` action follows after these actions in the plan

ID	Testcase	Variations	Expectation
CF2	Single `remove`	a. At start, intermediate or towards the end offset of the medium b. bytes behind: None, or whole-numbered multiple of the maximum write block size, or no whole-numbered multiple, or fewer bytes than the maximum write block size c. Removed byte count: The same cases as for the bytes behind d. Extreme case: All bytes of the medium are removed	No read/write operations before remove offset; bytes behind remain unchanged and get shifted by k bytes towards decreasing offsets, such that medium length shrinks by k bytes; the CFP contains N pairs of read/ write operations for the bytes behind, where N is the number of started blocks of at most "maximum write block size" bytes that are located behind the last removed byte; the read/write pairs start from the first block after the last removed byte forward up to the last block of the medium, where the writes occur exactly k bytes before the reads; the `remove` action follows after these actions in the plan; at the end there is a `truncate` operation

ID	Testcase	Variations	Expectation
CF3	Single `replace`	a. At start, intermediate or towards the end offset of the medium b. bytes behind: None, or whole-numbered multiple of the maximum write block size, or no whole-numbered multiple, or fewer bytes than the maximum write block size c. bytes to replace: The same cases as for the bytes behind d. Replacement bytes: The same cases as for the bytes behind e. Number of bytes to replace ($:= m$) smaller, equal to or bigger than the number of replacement bytes ($:= n$) f. Extreme case: All bytes of the medium are replaced	No read/write operations before remove offset; If $m > n$: Behaves like a *remove* of $m - n$ bytes; If $m < n$: Behaves like an *insert* of $n - m$ bytes; If $m = n$: No read/write pairs for bytes behind; In every case there, there are X corresponding write operations (one per starting block) for the replacement bytes with increasing offsets starting at the replacement offset; The `replace` action follows after these actions in the plan

ID	Testcase	Variations	Expectation
CF4	Multiple `inserts`	a. All inserts directly subsequent, or with gaps between b. bytes in-between/behind: None, whole-numbered multiple of the maximum write block size, or no whole-numbered multiple, or fewer bytes than the maximum write block size c. Bytes to remove: None, whole-numbered multiple of the maximum write block size, or no whole-numbered multiple, or fewer bytes than the maximum write block size	No read/write operations before the first insertion; bytes behind the last insertion: They remain unchanged and are shifted by k bytes to the back, such that the overall length of the medium increases by k bytes; bytes in between insertions: remain unchanged and are shifted to the back by the number of up-to-then inserted bytes; For these bytes, N read/write operations are returned, where N is the number of at least started blocks of maximum write block size, that are behind or between the insert operation on the medium; the `insert` action follows after these actions in the plan

ID	Testcase	Variations	Expectation
CF5	Multiple `removes`	a. All removes at the same offset or at different offsets b. bytes in-between/behind: None, whole-numbered multiple of the maximum write block size, or no whole-numbered multiple, or fewer bytes than the maximum write block size c. Bytes to insert: None, whole-numbered multiple of the maximum write block size, or no whole-numbered multiple, or fewer bytes than the maximum write block size	No read/write operations before the first remove; bytes behind: Remain unchanged and are shifted to the front by k bytes, such that the overall medium length decreases by exactly k bytes; N read/write operations are returned for these operations, where N is the number of at least started blocks of maximum write block size, that are behind the remove operation on the medium; a `truncate` operation follows in the plan

Table 11.5.: Test cases for checking `createFlushPlan`

Configuration Parameters

Even if the the parameters configurable by users belong to the public interface, they are only listed first here, as it was only clear after the implementation design, which of those parameters need to be exposed to the users.

We first define:

DES 108: **The configuration of `Media` is done per `IMedium` instance**
The configuration of `Media` is done per `IMedium` instance. By doing this, all configuration parameters correlate to an `IMedium`, have the same lifetime and scope. Thus, there are `setProperty()` and `getProperty()` methods for a `IMedium`.

Begründung: The whole internal implementation of the most important classes, i.e. `IMediumStore`, `IMediumAccessor`, `MediumCache` and so on works on exactly one medium. The user can create `IMedium` instances directly himself and configure them as needed, independent of other `IMedium` instances.

Nachteile: No disadvantages known

Now the question arises: Which configuration parameters are needed? Those listed in table 11.6.

Medium	Parameter Name	Type	Default Value	Description
`File`, `InputStream`	`ENABLE_CACHING`	boolean	`true`	Enables or disables caching for a medium according to DES 108. The cache content is directly cleared if setting this to `false`.
`File`, `InputStream`	`MAX_CACHE_REGION_SIZE`	int > 0	`Integer .MAX_VALUE`	Sets the maximum size of a cache region according to DES 108.
`File`, `InputStream`	`MAX_CACHE_SIZE`	long > 0	Long `.MAX_VALUE`	Sets the maximum cache size according to DES 108.
`File`, `byte-Array`	`MAX_WRITE_BLOCK_SIZE`	int > 0	8192	The maximum size of read-write-actions in bytes, that is triggered by `INSERT`s or `REMOVE`s during a `flush()`, see DES 108.
`InputStream`	`SKIP_ON_FORWARD_READ`	boolean	`false`	When calling `IMediumStore.cache()` for an `InputStream` with a previously not read offset, a value of `false` means that all in-between bytes, i.e. the bytes between current and given offset, are read into the cache. If `true`, `IMediumStore.skip()` is used in contrast, i.e. those bytes will be discarded and not be reachable anymore. Compare DES 108 and the description of cache() in 11.3.

Medium	Parameter Name	Type	Default Value	Description
`InputStream`	`READ_TIMEOUT_MILLIS`	int > 0	Integer `.MAX_VALUE`	The maximum read timeout for `InputStream`s according to DES 108, in milliseconds.

Table 11.6.: Configuration parameters of the component `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