jMeta 0.5
Design

Version: 0.2
Status: - Draft -

November 28, 2017

Contents

List of Figures

List of Tables

1. Introduction

This document specifies the technical design of `jMeta 0.5`.

Part I.

Requirements

This chapter defines the most important functional and non-functional requirements for `jMeta`. Most relevant input is the document [MetaComp].

2. Scope

2.1. In Scope

`jMeta` is a Java library for reading and writing container and metadata formats. `jMeta` has the following goals:

- Define a robust, easy to use and generic API for reading and writing multimedia (i.e. audio, video and image) metadata
- In addition, define a way to also parse and change typical container formats that embed or surround multimedia metadata
- Be easily extensible with additional metadata or container formats, also by third party

Thus, clearly, `jMeta` targets applications and users in the multimedia editing area, e.g. software to manage an audio, video or image collection.

Special strenght of the library should be its versatility (in a sense of supported formats and generality) as well as its extensibility. At the same time, it targets to offer access to ALL features of each specific format, even the low-level ones, and thus does not want to hide anything from expert users who need fine-grained control.

Using `jMeta`, an application is allowed to access metadata quite generically and comfortably, or it can also explore deep specifics of each format down to the bit and byte level.

2.2. Out of Scope

Metadata not in the audio, video or image domain are not directly in scope of `jMeta` and thus the library might not offer the right abstractions for those. However, it is at least likely that other binary or textual formats can also be parsed quite the same way using `jMeta`.

Furthermore, `jMeta` clearly is no encoding or decoding library, it cannot understand codecs. It might however help to encode or decode by providing access to the low-level details of the container formats. It might be used as basis for extensible encoders or decoders, however, not exactly in high-performance areas where speed is of most importance.

Further things out of scope of specific versions are treated in "4 Requirements and Exclusions".

2.3. Features of version 0.5

Here we give a brief overview of the features the library needs to offer in version 0.5. It is the first version of the library, so its goal is the bare minimum: Implement all core features and support some of the most important data formats. Image and video formats are not yet in scope of this version.

2.3.1. Supported Platforms

In general, jMeta in its current version 0.5 can be used with any platform that also supports Java SE in Java SE 9. However, it needs to be specified for which platforms tests were run explicitly and thus it is proven that specific features work on these platforms. We target to support the following platforms like this:

- Windows 10
- Ubuntu 16.04

2.3.2. Supported Metadata Formats

This version adds support for the following audio metadata formats:

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

2.3.3. Supported Container Formate

This version adds support for the following audio container formats:

- MPEG-1 Audio Layer 3 (MP3)

2.3.4. Supported Input Media

This version adds support for the following data sources for reading data:

- Files
- Java InputStreams
- Java byte arrays

2.3.5. Supported Output Media

This version adds support for the following output media for writing data:

- Files
- Java byte arrays

2.4. Related Libraries

Here, related Java libraries are treated that are related to what `jMeta` offers. This includes existing productive Java libraries in the area of multimedia, which can be considered as open source competitors to `jMeta` on the one hand, on the other hand, they provide inspiration, possible reuse and also warning examples how to not do it.

2.4.1. Metadata Libraries

A selection of libraries offering access to multimedia metadata as of end of 2017, mostly audio metadata - One source for this e.g. is `http://id3.org/Implementations` and Google:

Library	URL	Supported Formats	Lines of Code	Comments
jaudiotagger (Java)	http://www.jthink.net/jaudiotagger/	MP3, MP4, Ogg Vorbis, Flac, WMA, partly WAV and Real Audio	> 86000	Obscure class and package structure, more than 500 classes, just a few testcases only (?), still actively developed
jLayer and JavaZoom (Java)	http://www.javazoom.net/index.shtml	MP3, WAV, ID3v1, ID3v2	> 13600	Project targeting MP3 decoding and playing of MP3s
mp3agic (Java)	https://github.com/mpatric/mp3agic	ID3 Tags (1.0, 1.1, 2.3, 2.4), Read-only ID3 v2.2, MP3 low-level reading (incl. VBR)	> 5500	Still actively developed
BeagleBuddy (Java)	http://www.beaglebuddy.com/	Reads and writes ID3 Tags (1.0, 1.1, 2.3, 2.4), Lyrics3v2, Lyrics3v1, APEv1, APEv2, reads MP3 files CBR and VBR, Xing, LAME, and VBRI Header	> 40700	Focus on being "easy to use"; Last version in beginning of 2015
JID3 (Java)	https://blinkenlights.org/jid3/, https://java.net/projects/jid3lib	ID3v1, ID3v2.2 and ID3v2.3; no ID3v2.4 support	> 30000	Seems to be dead (last version 0.46 in 2005)

Library	URL	Supported Formats	Lines of Code	Comments
Javamusictag (Java)	http://javamusictag.sourceforge.net/	MP3, ID3v1, ID3v1.1, Lyrics3v1, Lyrics3v2, ID3v2.2, ID3v2.3, and ID3v2.4	> 27000	Seems to be the same thing or a fork of JID3
Xuggler (Java)	http://www.xuggle.com/xuggler	MP3, Ogg (Vorbis, Speex), Flac, AAC, ID3v2 and others	-	Decoder and editing API for various audio and video formats
jFlac (Java)	http://jflac.sourceforge.net/	FLAC, VorbisComment	-	Especially encoding and decoding of FLAC
MPEG-7 Audio Encoder (Java)	http://mpeg7audioenc.sourceforge.net/	MPEG-7	-	Creates MPEG-7 metadata
JAI Image I/O (Java)	https://jai-imageio.dev.java.net/binary-builds.html	EXIF	-	Can read and write EXIF tags from a variety of embedding formats
jmac (Java)	http://sourceforge.net/projects/jmac/	APE, MonkeyAudio	-	
MyID3 (Java)	https://github.com/jkauflin/jjkMP3	Audio-Metadaten (ID3)	-	Private engagement, not really to be taken seriously

Library	URL	Supported Formats	Lines of Code	Comments
id3lib (C/C++)	http://id3lib.sourceforge.net/	Audio-Metadaten (ID3)	-	
Mutagen (Python)	http://pypi.python.org/pypi/mutagen/1.12	Audio-Metadaten und -Container-Daten (ID3)		

Table 2.1.: Competitor libraries compared

The analysis shows that as of now, the biggest competitors would be jAudioTagger, mp3agic, BeagleBuddy and JID (which seems dead) from a metadata perspective and Xuggler as well as jLayer from decoding and editing perspective (if `jMeta` should ever try to go into this direction).

Drawbacks of Existing Metadata Libraries

Each of the mentioned libraries specializes itself to just a few formats. Saying this, applications are only likely to be satisfied with them if they just need to support those formats. If they do require more, they need to use multiple libraries or extend the existing ones or wait for new features.

Second, the architecture and extensibility of the existing libraries is not convincing. Most of the ID3 libraries e.g. open-heartedly show their very internals, which does not make it clear what really distinguishes "public API" and the private parts - with the usual consequences. Users might rely on private implementation aspects that are changed later, or if it is known that users do so for specific parts, the library evolution might get constrained by this, forcing duplication and maintenance of actually obsolete code.

Another smell of most of the libraries is their sheer volume: While mp3agic still seems humble in terms of "only" about 6000 lines of code (including comments and blank lines), it also does not offer so much features. The other libraries can really be considered big, topping to about over 86000 lines of code for jaudiotagger. These libraries also do not show an obvious structure or core concepts springing to one's eyes. Maintainability and especially extensibility of such rather "hardcoded" libraries might be not too easy.

Users / software needing to support a broad variety of formats - when using these formats - likely need to implement a kind of plugin mechanism for easier extensibility themselves.

Advantages of `jMeta`

There are alreads so much library - why do we need `jMeta`? If it wants to do things better, a lot of effort is required. Is that worth it?

There are multiple aspects where `jMeta` can do better than existing libraries:

- `jMeta` should target container formats on a very abstract and generic level, yet still providing access to the bare details of each format; this can go as far as providing a generic framework for parsing and writing any multimedia container format, may it be audio, video, image or text content.
- Applications whose core asset is extensibility and rich variety of supported formats do not have to use a dozen or so third party libraries having completely different programming, license and support models as well as performance characteristics - provided `jMeta` actually offers support for all formats, which is of course not necessarily the case
- However, at least such applications do not need to reinvent the wheel by coding their own extensibility mechanism, but they just rely on the extensi-

bility mechanism by using `jMeta`; they can code their own `jMeta` extension to support another format. The goal of `jMeta` is that doing so is easier for such applications than using another 3rd party library to enable support for such a format.

- Still, `jMeta` is not a huge Uber JAR containing all supported formats, but it is modular, where applications just need the core of the library plus an extension module per format they want to support.

- `jMeta` also targets to offer an easy to learn and use API for such applications.

All that said, the benchmark for how good `jMeta` really is, is as follows:

- It should be easier to use as BeaglyBuddy which claims to be easy to use

- It should be much less complex (means to say: much less classes and lines of code) than jaudiotagger or BeagleBuddy

- Supporting a new, averagely complex, previously unsupported data format does not take an experienced Java programmer (who never did write a `jMeta` extension) more than one man day!

- Likewise, we also want to compete with other libraries in terms of performance: `jMeta` must at least be comparatively fast as the most important competitors for Java, ideally of course it should be faster

2.4.2. Multimedia Libraries

Can `jMeta` cooperate, i.e. reuse or integrate with existing Java multimedia libraries? This question is not really answered in the following sections. The general answer is: No, `jMeta` will neither be some kind of plugin for the existing libraries, nor will it depend on them, nor will it consider the requirement of easy cooperation or integration with this libraries in any way.

For now, we just want to see what is existing in the wild in terms of Java multimedia libraries, mostly audio and video. This evaluation might point future directions for `jMeta` if it goes to probably superseding or extending the existing options.

All or most of the multimedia libraries listed here seem to offer some way of reading or writing multimedia metadata. But mostly, these libraries focus on streaming, playing, encoding and decoding on high performance. The metadata is just a side-product that seems to be offered most of the time via a rather inconvenient API.

Xuggler and Humble Video

Cite from the ebsite: “Xuggler is the easy way to uncompress, modify, and recompress any media file (or stream) from Java. ... Xuggler allows Java programs to decode, encode, and experience (almost) any video format.”

It does not seem to be based on GStreamer or FFMPEG or JMF. It does not become clear which formats are supported by the library.

The official Github page of Xuggler states it is outdated and humble video should be used. The name suggests Humble Video is mainly focused on video, not audio such as MP3.

JLayer

Cite for JLayer: "JLayer is a JAVA library that decodes, converts and plays MP3 files in real-time. JLayer supports MPEG 1/2/2.5 Layer 1/2/3 audio format. JLayer doesn't need JMF."

It does also not seem to rely on GStreamer or FFMPEG but seems to be a "native" Java only implementation for encoding and decoding MP3 audio data.

Java FX and GStreamer

JavaFX is an Oracle library to support with platform-independent handling of user interfaces as well as multimedia content. So, it is not only "GUI" that actually replaces Swing and AWT, but also handling of images, video and audio content as well as even 3D graphics, thus somewhat replacing the obsolete Java 3D.

So it has a quite broad usage space. However, there is also some muttering that it might be dead or at least does not have a bright future.

For local GUIs it might still be a good option. What about multimedia playback, editing etc.? The media engine part of JavaFX is based on GStreamer. GStreamer is an extensive audio and video library written in C and nevertheless platform-independent by being ported to a lot of platforms and architectures. It also supports reading and writing tags, but in a very specific generic API which does not offer too much comfort. But how does this go together with JavaFX? First of all, GStreamer also offers a Java binding, i.e. can be called from Java applications where any installed GStreamer plugins for formats can also be used. This interface is most probably also used by JavaFX. JavaFX then offers a class named MediaPlayer with a method chain `getMedia().getMetadata()`, just returning a map (String to Object). There seems to be no documentation which objects might get returned. Furthermore it just seems to be read-only, there does not seem to be any way of writing tags with JavaFX.

FFMPEG

FFMPEG can be seen as direct competition to GStreamer, being a platform-independent framework and toolkit to record, convert and stream audio and video content. It is also written in C/C++ and supports a variety of formats.

JMF and FMJ

The Java Media Framework is an official Java library that is delivered with J2SE *desktop* technology. The most recent version is 2.1.1e and dating back to 2001. Which actually means: It is dead. However, we still consider it a bit here. JMF

can be used by Client as well as server applications. JMF was packaged with an MP3 decoder and encoder until 2002, but removed due to licensing issues. Since 2004 there is an MP3 playback only plug-in.[1]

FMJ is an Open Source Alternative, that is API compatible: http://www.fmj-sf.net/. But FMJ also seems to be dead.

JMF comes in four JAR files:[2]

- JMStudio: Simple multimedia player application
- JMFRegistry: An application to manage different JMF settings and plug-ins
- JMFCustomizer: Allows creation of a simple JMF JAR file containing only those JMF classes needed by the client application
- JMFInit: Inititializes a JMF application

JMF contains platform-specific *performance packs*, i.e. optimized packtes for Linux, Solaris or Windows.

Features: JMF deals with time-based media. The JMF features can be summed up as follows:[3]

- Capture: Read multimedia frame data of a given audio or video signal and encode it into a specific codec in realtime.
- Playback: Play multimedia data, i.e. display videos on screen or play music on audio output devices.
- Stream: Access multimedia streams
- Transcode: Convert media data of a given codec into another codec without first needing to decode

Criticism: [WikJMF] summarizes some negative feedback for JMF:

- A lot of codecs such as MPEG-4, MPEG-2, RealAudio and WindowsMedia are not supported, MP3 only as plug-in
- No maintenance or extension by Oracle, it is dead
- No editing functionality, i.e. modification of multimedia content
- Performance packs only for just a few platforms

[1] see [WikJMF].
[2] Siehe [WikJMF].
[3] Siehe [JMFWeb].

Basic concepts of the API Reading of multimedia data is abstracted using `DataSources`, while output goes to `DataSinks`. No specifics of supported formats are provided for direct API access, they can just be played, processed and transcoded, while the latter is not supported for all formats. A `Manager` class is the primary API for JMF users.[4]

The API documentation shows that JMF is quite complex and essentially time- and event-based.[5] It offers possibilities to read raw binary data via a method `read` of `PullInputStream`. However, JMF controls processing starting at the source, i.e. from a file or stream.

JavaSound

JavaSound is Oracle's sound processing library. It has some things in common with JMF, but can be considered quite low-level, as it also offers modification functionality for audio data. It also supports MIDI devices.[6] It can also be considered dead, unfortunately.

Basic concepts: JavaSound essentiall offers the classes `Line` (representing an element in the audio processing pipeline), the derived classes `Clip` (for playing audio data) and `Mixer` (for editing audio data). The library can read from streams, files as well as in-memory bytes. It also offers "transcoding" functionality to convert between different formats.

[4] see [WikJMF].
[5] See [JMFDoc].
[6] See [WikJavaSound].

3. Basic Terms

Here we define basic terms used throughout the whole design concept.

3.1. Metadata

Metadata in this document is short for digital metadata that are not necessary to parse the actual described (audio, video or image) data. Metadata semantically and structurally describes other data. The goal of `jMeta` is especially reading of metadata for audio and video data sets, e.g. title, artist etc. The structure of metadata is defined by a Metadata-Format.

If it is specifically about technical metadata needed to parse a data structure, e.g. in the container header, we call it *Parsing Metadata*.

3.2. Data-Formats, Metadata-Formats and Container-Formats

A Data-Format defines the structure and interpretation of data: Which bytes or characters of which value and in which order have what kind of meaning? Usually, a data format describes how a consecutive block of bytes (i.e. a Datablock) is built up by a number of so-called Fields or child Datablocks.

Metadata-Formats are data formats that define the structure of digital metadata. Examples include: ID3v1, ID3v2.3, APEv1, MPEG-7, RDF/XML, Vorbis-Comment and others.

Container-Formats are a more general forn of Data-Format optimized for storing, transporting, editing and seeking multimedia Payload data. Examples are: MP3, Ogg, TIFF, QuickTime, JPEG 2000, PDF and others. Metadata formats can also be considered as container formats.

An example for other Data-Formats is HTML. You can argue that it is neither a Metadata-Format nor a Container-Format. XML is a Data-Format that itself can be used to define further XML Data-Formats. Some XML Data-Formats are Metadata-Formats, e.g. MPEG-7, MPEG-21 or P_Meta.

3.3. Transformationen

A Data-Format may define Transformations. A Transformation describes a way how read or to be written data needs to be transformed to fulfill specific needs. You can envision this as kind of encoding of the data. In contrast to the fixed data format specification which describes in detail how binary data is coded and needs to be interpreted, Transformations are optional features that

are dynamically applied to certain areas of the data. Partly, TRANSFORMATIONs can also be defined by users of the data. Examples are the TRANSFORMATIONs defined by ID3v2: Unsynchronization, Encryption and Compression.

3.4. Datablocks

A DATABLOCK is a sequence of bytes that together form a logical unit (i.e. an object or entity with a specific meaning) in terms of the underlying DATA-FORMAT. Each DATABLOCK belongs to exactly one DATA-FORMAT. It can be assigned a current length in bytes. There are several concrete types of DATABLOCKs that are described in the following sections.

The actual detailed meta model of DATABLOCKs is defined later in the detailed `jMeta` design sections of this document for

3.4.1. Container: Payload, Header, Footer

An important type of DATABLOCK is a CONTAINER: It consists of zero, one or several HEADERs, exactly one PAYLOAD and zero, one or several FOOTERs. All of these are *child* DATABLOCKs. CONTAINERs are a common concept for container formats: The HEADERs and FOOTERs describe the CONTAINER in terms of its length, size and other properties. The PAYLOAD contains the interesting data, e.g. the multimedia data to be extracted, played or viewed. FOOTER essentially allow for backward or reverse reading. Most of the DATA-FORMATs specify a generic structure of a CONTAINER, with specific containers with specific purpose, but allowing user-defined new CONTAINERs at the same time, i.e. the format is extensible.

3.4.2. Tag

A TAG is a special CONTAINER whose purpose is to store metadata. It can either belong to a standalone METADATA-FORMAT ore to a more general CONTAINER-FORMAT. Especially audio, video and image metadata formats use this term when talking about such a DATABLOCK in a file or MEDIA STREAM, e.g. the ID3 or APE TAGs. It term originates from "tagging" something with additional meta-information, as you'd attach a label to describe the song, video or image.

The following figure shows the basic structure of a TAG, showing other basic terms:

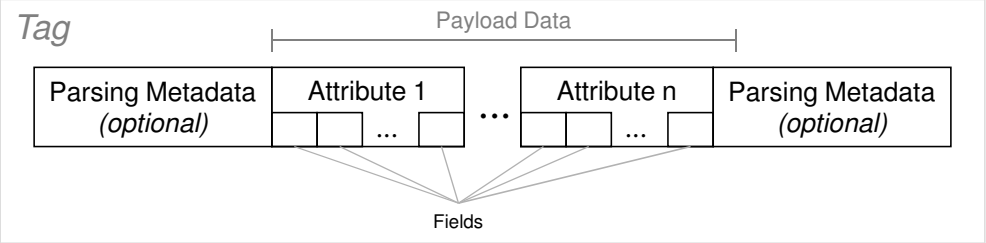

Figure 3.1.: Structure of a TAG

The most important parts of a TAG are the ATTRIBUTS.

3.4.3. Attribut

An ATTRIBUT is a part of a TAG that contains the valuable metadata information in a key-value manner. Common examples are artist, title, album, composer etc. of a piece of audio. Often, an ATTRIBUT is also a CONTAINER in a sense that it has a HEADER and PAYLOAD. The HEADER may help to define the type (artist, title, album) of the ATTRIBUT as well as the size of the PAYLOAD. The PAYLOAD contains the actual information in an encoded way, e.g. the name of the artist or title of the piece of audio.

Most of the ATTRIBUTS have a simple main value that can be given. However, there are also more complex ATTRIBUTS that consist of many values in form of child CONTAINERS or FIELDS.

In each metadata format, an ATTRIBUT has a format-specific name, e.g.:

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

3.4.4. Fields

A FIELD is a sequence of bits that together have a specific meaning in a given DATA-FORMAT. They form the leafs of the data block hierarchy by ultimately containg the raw data. A DATA-FORMAT describes how a specific DATABLOCK is built up by a specific sequence of FIELDS. A FIELD has a range of possible values and interpretations of these values. Often, one part of the value range is defined as "reserved" to ensure a bit of flexibility in extending the data format. Fields can be fixed-size or of variable length. If they have a variable length, there is often either another field specifying its length, or it is terminated by a specific sequence of bytes or characters.

3.5. Medium

A MEDIUM incorporates both the physical location of the data containing DATABLOCKs to read and write as well as the way of accessing this data. It may be a file, a stream or even plain memory.

4. Requirements and Exclusions

Here, all explicit high-level requirements for `jMeta` in version `0.5` are listed, including some explicit exclusions. Exclusions have the meaning that a given feature is explicitly not supported in the current version, and it might be like that also in future versions. No design is made for supporting them. However, it might well be that for future versions of the library, these decisions are changed and previously excluded features are then supported. The question is: Why listing exclusions at all? How far do they go, so will there be an exclusing stating that `jMeta` is not an operating system, e.g.? Not that far. Exclusions are only meant to demonstrate lack of features that popped up during the design phase and that could be desirable for `jMeta` users.

In contrast to exclusions, there are potential features such as new metadata or container formats to support. These are neither explicitly excluded nor already planned for future releases.

4.1. REQ 001: Reading and writing of human-readable meta data

`jMeta` can read and write meta data in a *human-readable format*. That means librar users do not have to read (and interpret) or write raw binary data.

Rationale: This is the general core functionality of the library and must be supported.

A list of supported data formats in the current version is given in "2.4.2 Features:".

4.2. REQ 002: Read container formats

`jMeta` must be at least able to read popular and wide-spread container formats. Writing support is optional.

Rationale: Multimedia meta data is often embedded in container formats or an integral part of their specification. `jMeta` must be able to identify and parse container segments in order to skip them to find the actual start of a meta data tag, or to read their meta data content. Writing is optional because `jMeta` cannot offer encoding functionality, so just writing header information and raw binary data that has been encoded by another library could be offered.

4.3. REQ 003: Fulfill specification of supported data formats

As far as there is an official specification for a supported data format, this specification must be fully supported by `jMeta`, that means all encoding types, optional headers, data transformations (e.g. encryption), padding and other features that are included in the specification, as well as any generic attributes or containers must be readable and writable. It is not necessary that `jMeta` explicitly supports unspecified attribute types that might be popular but not defined by the specification. Such attributes should however be accessible using the generic attribute mechanism offered by most data formats.

Rationale: This ensures that all meta data or container format blocks that adhere to their specification can be read by `jMeta`, and furthermore, that `jMeta` itself generates data that is compliant to the specification. Second, the user can use `jMeta` to use specific data format features in a convenient way without needing to implement this himself.

4.4. REQ 004: Access to raw data via `jMeta`

In addtion to the access to human-readable meta data ("4.1 REQ 001: Reading and writing of human-readable meta data"), it must be possible to explicitly read and write raw data on byte level. `jMeta` must offer fine-grained access to fields of binary data.

Rationale: Thus, user could implement parsing themselves without needing to use high-level functions. In some rare cases this might be necessary to circumvent some issues with the data written by some other tool. This channel of access is offered to users such that they are not forced to completely skip `jMeta` and implement raw access to supported media themselves, leading to also sync and consistency issues when doing this in parallel to the access by `jMeta`.

4.5. REQ 005: Performance as good as other Java meta data libraries

The performance of `jMeta` on all supported platforms must be as least as good as the performance of competitor Java meta data libraries such as jaudiotagger, mp3agic or BeagleBuddy. A corresponding reasonable benchmark needs to be set up with published code to demonstrate this. It is out of scope to make `jMeta` as fast as high-performance encoding or decoding libraries on C++ basis such as gstreamer of ffmpeg.

Rationale: `jMeta` targets to be a good replacement for any existing Java multimedia tagging library, and thus it is clear it should at least perform at the same level or even better.

4.6. REQ 006: Fehlererkennung, Fehlertoleranz, Fehlerkorrektur

Ergänzend zur "4.3 REQ 003: Fulfill specification of supported data formats" 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).

Rationale: 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. REQ 007: Extensibility for new Metadata and Container Formats

`jMeta` must be comfortably extensible with new container or metadata formats. As the minimum level, easy extensibility by the library developers must be possible. As the maximum level of extensibility, also end users with programming experience must be able to easily write extensions without too much configuration or boilerplate code.

Rationale: A wealth of already existing meta data and container formats is yet out there. Just the formats with major importance are supported as of now by `jMeta`. Furthermore, we might expect new meta data or container formats in future. The extensibility mechanism first of all guarantees an easy extensibility by a 3rd party, e.g. lib name users or also other vendors. The extensibility also ensures a longer life time of the library. In the maximum level "Extensibility by end users", this is a clear differntiation criterion to other libraries that do not offer this level of extensibility. Last but not least, this allows easier maintenance (i.e. extension) by the library developers itself if they decide to bundle additional format support extensions with the core library in a new release.

4.8. REQ 008: Read and write large data blocks

`jMeta` must be able to read and write large amounts of data efficiently. With "large" we mean that `jMeta` must support data blocks with maximum size of $2^{63} - 1$ bytes. The length of payloads must be interpreted correctly. Because of "4.4 REQ 004: Access to raw data via `jMeta`", also reading and writing of raw data must be supported without necessarily causing scarcity of available memory, e.g. by chunk-wise reading possibilities. In general, `jMeta` must use mechanisms to avoid `OutOfMemoryError`s, unless the user forces this by reading the whole binary data of a block into memory.

Rationale: Especially in the domain of video container formats, payloads or at least files with multiple gigabytes are very common. Yet, still $2^{63} - 1$ bytes as upper limit should suffice for some years or decades to come. Supporting this is for sure also a plus compared to other competitor libraries that might not explicitly support large files.

4.9. REQ 009: Selective Format Choice

An application using `jMeta` must be able to selectively choose those formats that it wants to support. This is not only necessary for runtime, but also for the library extension packages it wants to use.

Rationale: Audio applications do not need extensions for video or image formats. Applications can minimize the runtime and memory (both HDD and RAM) overhead by choosing as few extensions as really needed.

4.10. EXCL 001: Reading media streams

`jMeta` in its current version `0.5` does not explicitly support reading meta data or containers from media streams consumed e.g. from the internet. It might offer optional starting points for this by e.g. in principle supporting access to `InputStream`s. But it does not offer an explicit API or examples, nor do we create specific test cases for media streams. Furthermore, media stream specific tag formats such as `IcyTag` are not in scope of the current library version.

Rationale: Combined applications such as recorders or players whose main task is playing make more sense for this, as they are not only interested in meta data. Supporting streaming media in addition might overly complicate the design and API of the current library version, so this might be left for future versions to come.

4.11. EXCL 002: Support for XML meta data

There are also XML meta data formats out in the wild. They are not used very often for multimedia meta data, but cases exist. `jMeta` does not support reading and writing of XML meta data format in the current version, but only specializes to binary formats. **Rationale:** Binary meta data formats still seem to rule the scene due to their compactness. It would be absurd to design a generic library that can parse both XML and binary formats with the same code, as XML is usually efficiently read and written using streaming parsers and validated using schemas, where excellent APIs already exist. It is left for the future to offer built-in support for XML metadata in `jMeta` via an easy API.

4.12. EXCL 003: No user extensions for `jMeta` media

Extending `jMeta` with new media to support - on top of the out-of-the box supported media - is not supported.

Rationale: The mechanisms available should cover 80 to 90 percent of the use cases. Extensibility by new media might increase the complexity of `jMeta` itself and the extension mechanism in specific, without adding real-world use cases really needing it. It is currently not clear which media beyond streams, byte arrays and files might be candidates for extending `jMeta`. Should new media make sense in future, a new core release can be created to also support media extensions.

4.13. EXCL 004: `jMeta` is not a high-performance encoder or decoder

Even though it would be tempting, `jMeta` currently is not a high-performance encoder or decoder for multimedia content. It is not the target to extend this library to be that, but you never know. However, it can be the basis for a Java-based encoder or decoder pack, as it offers bit-wise parsing and other raw data access functionality for container formats. But there is no specific design into that direction, especially no means for specific multi-threading built-in support, means users are left to leverage multiple cores themselves as necessary.

Rationale: Audio, video and image formats as well as corresponding encoders are extensively complex. A huge variety of great libraries and tools is already existing. For high-performance needs, especially GStreamer and FFMPEG seem to rule the scene. `jMeta` can never compete with that and it would mean to *just stop* the project when trying to do just that. Instead, we can offer the primitives for Java developers who might want to develop their own (probably not very high-performance) codecs. Maybe in the very far future, `jMeta` could offer integration functionality to plug-in existing codec implementations somehow conveniently, such that it can be used on its own as a basis for Java transcoding and player/viewer applications.

5. Reference Examples

To proof conformance with the `jMeta` requirements, a set of (mostly) real-life examples for all supported formats is used. The examples are used to illustrate design decisions and also to verify them. In this chapter, these examples are presented for short. The concrete detailed structures of each data format is described in [?].

Note that the sizes of the data blocks in the following illustrating example figures do not have any specific meaning.

5.1. Example 1: MP3 File with ID3v2.3, ID3v1.1 and Lyrics3

The following figure shows the first example, an MP3 file with three TAGs, ID3v2.3, Lyrics3v2 and ID3v1.1. All are located at the end of the file:

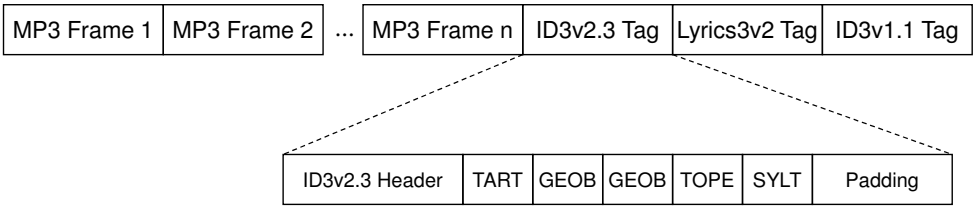

Figure 5.1.: Example 1: MP3 file with two tags

The ID3v2.3 tag has several frames, including two `GEOB` frames. Furthermore, it has some padding within. Each of the MP3 frames corresponds to the MPEG-1 elementary stream audio format.

5.2. Example 2: MP3 File with two ID3v2.4 Tags

The following figure shows the second example, an MP3 file with two ID3v2.4 TAGs, one at the beginning, the other one at the end of the file:

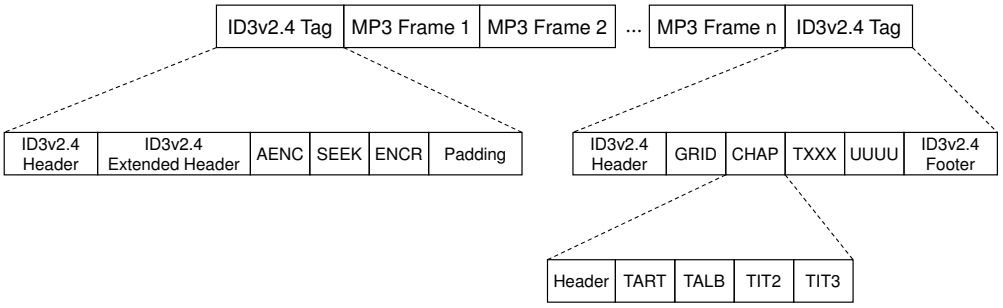

Figure 5.2.: Example 2: MP3 file with two ID3v2.4 tags

The two ID3v2.4 tags are virtually connected by a `SEEK` frame. Both have several specialties described in [**?**].

5.3. Example 3: Ogg Bitstream with Theora and VorbisComment

The following figure shows the fourth example, an Ogg bitstream that contains Theora payload data with a corresponding vorbis comment:

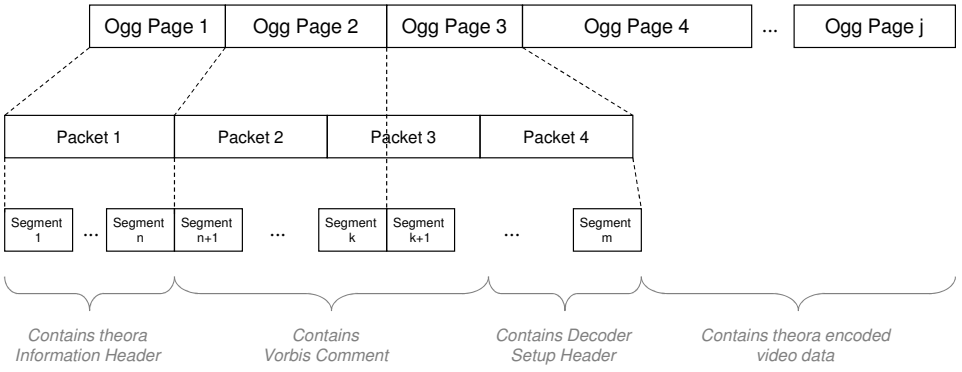

Figure 5.3.: Example 4: Ogg Bitstream with Theora and VorbisComment

The example is complex on first sight. In an Ogg bitstream, physical and logical structure are not necessarily the same. The physical structure is built by pages, packets and segments, while the logical structure is the structure of the wrapped data. We took theora video data as an example, but its basically arbitrary, as the codec does not really matter. What matters is where the Vorbis Comment, one of the supported data formats, is stored. This unfortunately depends on the embedded codec. In this example, the vorbis comment starts in the second page and spans over two packets. The second of these packets spans over two Ogg pages.

Part II.

Architecture

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`

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:

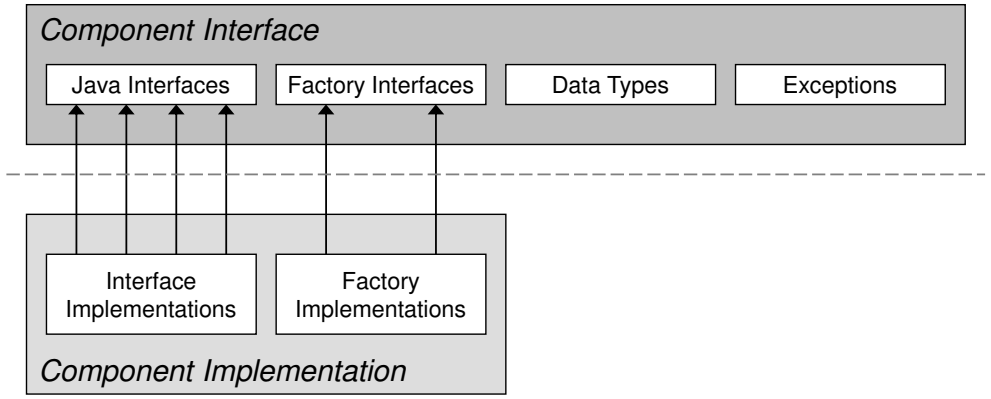

Figure 6.1.: Struktur einer Komponente

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 Technical Architecture" 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. Technical Architecture

7.1. Technical Infrastructure

The technical infrastructure describes the environment required to work with `jMeta` as shown in the following figure:

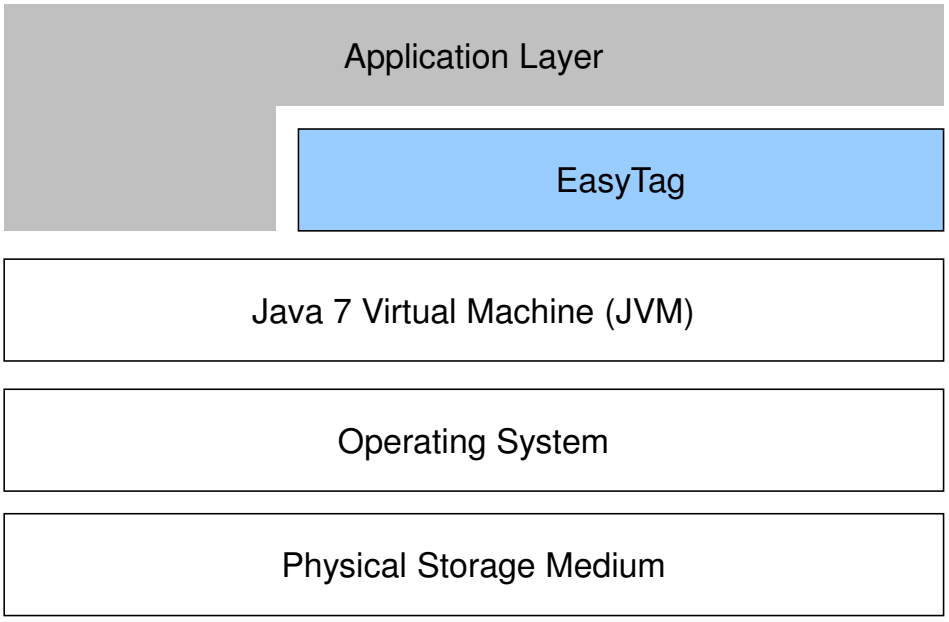

Figure 7.1.: Technical infrastructure of `jMeta`

The technical layers can be interpreted as dependency and communication structure. The application layer is based on `jMeta`, while `jMeta` as well as the application layer uses Java SE 9. Thanks to this, the library and its using applications are platform-independent and can be used on any platform currently supported by Java. Java SE 9 accesses the operating system, which offers services to access the physical medium.

However, `jMeta` is only developed for Java SE 9 and cannot be used by applications requiring earlier versions.

Each layer only accesses the neighouring layer.

7.2. Technical Base Components

Technical base components are just helpers providing a framework and basis for the functional components of `jMeta`. With "functional" we are referring to the tasks of the library, i.e. reading and writing metadata and reading container data. The necessary base components are:

- Logging
- Service-Locator to achieve a component oriented architecture
- Utility
- Maintaining extensions

Details of these components are given 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 "?? ??" 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 "**?? ??**".

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.

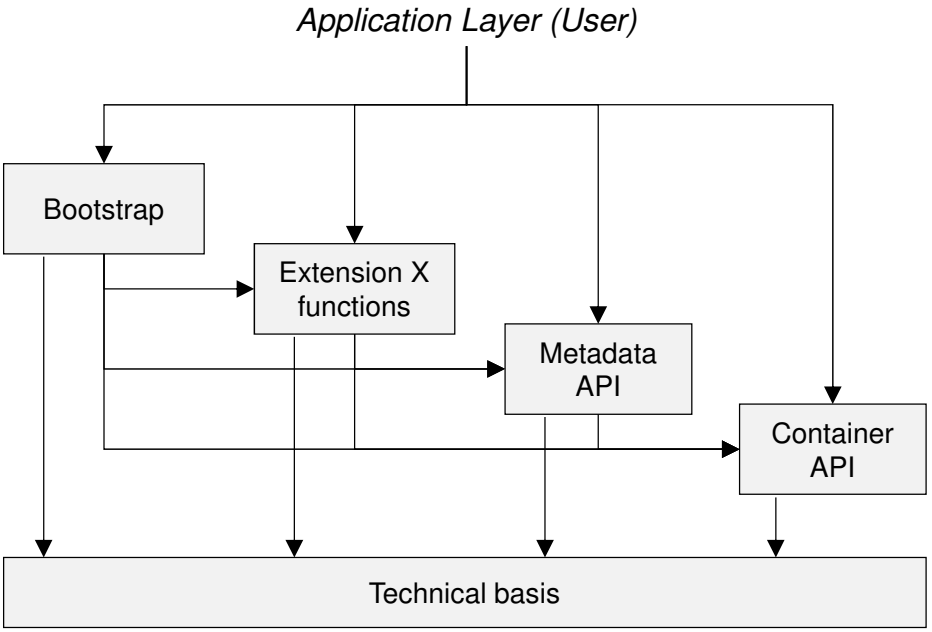

Figure 8.1.: Subsysteme von `jMeta`

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 Extensions (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 `ComponentRegistry`: `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. `ComponentRegistry`

Komponenten-Name: `ComponentRegistry`.

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. Extensions (Subsystem `Extension`)

An *Extension* summarizes format-specific content for a given data format (or multiple variations of the data format). Per data format defined by the extension, these are (compare DES 017, DES 017 and DES 017):

- A data format specification which defines data blocks and their structure and relations
- An end-user API to comfortably work with the data format (e.g. creation of data blocks etc.); this API especially holds the data format identifier which the user can use to work with `Metadata API` and `Container API`.
- Implementation extensions for `Container API`, which influence the parsing and writing process. With this mechanism, a data format can extend or override the standard parsing and writing algorithms defined in `Container API`.

As a ground rule, every single extension can but should not define extension code for more than one data format. It might expose more data formats only if these two data formats heavily overlap and are based on each other, e.g. ID3v1 and ID3v1.1. This is to fulfill the requirement "`?? ??`".

Moreover, distinct extensions should never depend on each other, but only on the `jMeta` core to avoid any confusion for the user.

Part III.

jMeta Design

This part defines the design of the components per subsystem, based on the components and subsystems defined in the previous chapters.

9. Cross-functional Aspects

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. Configuration

With the term *configuration* in `jMeta`, we understand the change of specific parameters, which customize the behaviour of `jMeta` at runtime. This is quite vague and the difference to usual specialized setters is unclear.

Thus we want to give some more criteria for configuration:

- Configuration is part of the public API of `jMeta`, i.e. it can be changed by the user.
- It is generic and thus extensible for further releases, i.e. adding new configuration parameters means to add a new constant to the API and update the documentation; the API for setting and retrieving the configuration parameter - as it is generic - does not change by adding new, changing existing or removing old parameters
- Configuration parameters usually have global effect and usually are just used once; but, of course in theory we could also make them dynamically changeable by users at runtime, and their scope could also be limited, e.g. to individual media

For now it turned out that there is no need in `jMeta` for such dynamic configuration parameters a.k.a properties, neither globally, nor for individual components. We thus summarize as follows:

DES 027: **`jMeta` offers no generic mechanism for (global or local) configuration, especiall it currently defines no properties that influence the behaviour of the library**
All necessary "configurations" for customizing the behaviour of a component are passed by the user via specialized (i.e. non-generic) setters or constructor parameters.

Begründung: There is no need for the bigger flexibility yet.

Nachteile: No disadvantages known.

9.4. Naming Conventions and Project Structure

9.4.1. Java Naming Conventions

We use the standard Java naming and formatting conventions as defined by Sun. Only small addition: If there is a service interface with only one implementation, this implementation is having the prefix `Standard`. E.g. the default implementation for the `MediumStore` interface is named `StandardMediumStore`.

9.4.2. Package Naming Conventions

Here, we quickly define the package naming to be used for the library and extensions of the library.

Core Library

All packages for the core library start with `com.github.jmeta`, according to the Java naming conventions. The reason is that the library is hosted on Github. Below this core name, the following sub-packages are used:

- `.library.<component>` for all core components of the library, where "`<component>`" is replaced by the actual component name
- `.utility.<component>` for all technical utility code needed for the library, where "`<component>`" is replaced by the actual category name
- `.defaultextensions.<extension name>` for all extensions delivered by default bundled with `jMeta`, where "`<extension name>`" is the name of the extension, usually a single data format name; see also the next section for third-party extensions

Below these package names, there is another hierarchy:

- `.api` containing the public interface of the component, utility or extension; this package itself should not contain classes directly, but is further subdivided into:
  - `.services` containing the function interfaces the user needs to use to work with the component, utility or extension
  - `.types` containing any transport objects, data types or the like for the component, utility or extension
  - `.exceptions` containing any public exceptions of the component, utility or extension
- `.impl` containing the private implementation of the component, utility or extension; sub-divisions into further packages below it are possible, but component-specific and not standardized

Extensions

As mentioned already, the default extensions delivered together with `jMeta` of course also use a package naming convention quite similar to the core library package names. For any third party extensions, there is of course no definition of how the packages should be named, but it is up to the creators of the extension.

However, extension providers should follow the standard Java package naming conventions, and they should ensure that their package names do not clash with package names of other `jMeta` extensions.

9.4.3. Project Naming and Structure

Every IDE project containing code for `jMeta` starts has the form `jMeta<LibraryPart>[Extension]`, where `<LibraryPart>` is:

- `Library:` The core library with the main user API, without any extension specific parts
- `Utility:` Any utility the core library and extensions need, but no classes the library user must depend on, i.e. just internal library utility
- `DefaultExtension:` For all projects building an extension that is already bundled by default with the core library.
- `Tools:` For any supporting, development or testing tools, i.e. mostly Java applications that can either be used by library users or by library developers
- `Docs:` For any documentation of the library, no source code here (except for docs, e.g. tex files)

`[Extension]` is only used for the `DefaultExtension` library part. Each extensíon needs its own project and build unit. The reason is requirement "**?? ??**". This allows us to depoly individual JARs for each single extension.

Below, you can see the actual projects currently existing and their allowed dependencies: TODO

9.4.4. Build Module Structure and Dependencies

The Maven build module structure is of course aligned with the project structure: There is exactly one Maven module per project, having the same dependencies as depicted above. One exception might be the Docs project as it contains no sources. In addition, there is a single parent POM for general dependencies and aggregate builds.

10. Subsystem Technical Base

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.

10.2. ComponentRegistry Design

In this section, the design of the component `ComponentRegistry` is described. Basic task of this component is the implementation of design decisions DES 027 and DES 027.

We just give the basic design decisions here. Before the current solution, we used a relatively complex custom-made component mechanism with caching, XML configuration of interface and implementation as well as registration of the components at startup. This code was organized in an eclipse project named "ComponentRegistry", but was later a bit simplified later as "SimpleComponentRegistry". However, after realizing that the built-in Java SE `ServiceLoader` mechanism is fully suitable and even better than the "SimpleComponentRegistry", this mechanism is used now.

According to DES 027, a component is a "Singleton". However, its life cycle must also be clearly:

DES 028: **`ComponentRegistry` uses Java's `ServiceLoader` mechanism**
`ComponentRegistry` uses Java's `ServiceLoader` class to load the implementation for a given component interface. All component interfaces and their implementations need to be configured in META-INF configuration files as indicated by the Javadocs of `ServiceLoader`.

Begründung: The `ServiceLoader` mechanism is quite easy to setup and requires nearly no specific coding. No code needs to be written to read corresponding configuration. A specific description / documentation mechanism of each component at runtime is not necessary, thus it was omitted (in contrast to previous implementation).

Nachteile: Of course, `ServiceLoader` has its limitations: You might not be easily able to add components at runtime without writing additional code or not at all. And it does not have any idea of the term of a component, it just knows interfaces and implementations. But this is all not a requirement for `jMeta`.

Here is just a short summary of small amount the self-written code in `ComponentRegistry`:

DES 029: `ComponentRegistry` **consists of a single non-thread-safe class for looking up implementations and cache handling**
`ComponentRegistry` solely consist of a single class. This class caches `ServiceLoader`s. It offers the following methods:

- `lookupService`: Loads a service implementation (if called the first time for an interface) and returns it, adds its `ServiceLoader` to the cache afterwards
- `clearServiceCache`: Clears the internal cache of `ServiceLoader`s such that the next call re-reads and re-instantiates new implementations again.

The class offers this as static methods, but it is not thread-safe.

Begründung: Why do we cache `ServiceLoader`s? Because the `ServiceLoader.load` method always creates a new instance of the `ServiceLoader` class, which has the following effects:

- Components are singletons, thus, if two different other components need an implementation of the component at different points in time, they would get different instance of the service implementation, as they would always create a new `ServiceLoader` instance.
- `ServiceLoader.load` does file I/O, which is not what we want every time we request a new implementation.

The method `clearServiceCache` is necessary for test cases which might need to reset the `ComponentRegistry` state after each test case. As it is a static class, this method is necessary.
Why static methods? Because we do not want to care about instantiating it, passing the same instance everywhere needed.
Why is it not thread-safe? Because the whole library is not intending to the thread-safe.

Nachteile: No disadvantages known

10.3. `ExtensionManagement` Design

This section describes the most important design decisions for the technical helper component `ExtensionManagement`. The main task of this component is discovery of extensions and to make their interfaces available to the whole library, preferably in a very generic way.

In the first version of this components, extensions were loaded by a relatively complex `URLClassLoader` approach and used a main and an extension-specific XML configuration file. However, this was more complex than necessary, thus we define here a more lightweight approach:

***DES 030:* Extensions are discovered and loaded at library startup directly from the class path, no dynamic loading of extensions is required**
When the library is first used in a Java application, it scans the class path for any available extensions. The extensions that are available at that time are available for further use. There is no possibility for the user itself to add extensions. Furthermore, it is not possible to add, exchange or update extensions at runtime in any way.

Begründung: It is quite convenient for users to simply put extension onto the class path, e.g. using Maven or Gradle, and it is auto-detected. This fulfills the requirement "**?? ??**". Dynamic loading or updating of extensions automatically or triggered by the `jMeta` user is not necessary, as in most use cases it should be clear at compile-time which extensions are required. As extensions should not be introducing a big overhead, applications can easily provide all in total possible extensions at build-time already.
Last but not least, for the `jMeta` implementation, complexity is largely reduced if we omit the requirement to load, update or exchange extensions at runtime.

Nachteile: No disadvantages known

How does the library identify extensions on the class path?

***DES 031:* An extension is identified by implementing `IExtension` and being configured by the `ServiceLoader` facility**
An extensions is identified by each implementation of the `IExtension` interface that is configured in a JAR file on the class path as a service provider. It is found on the class path by the library core by using the Java `ServiceLoader` class. It does not matter if the extensions is located in a separate JAR file or even already in the core, and how much extensions are in a single JAR file.

Begründung: `ServiceLoader` is a quite convenient and very easy to use mechanism.

Nachteile: No disadvantages known

Regarding the configuration of extensions:

***DES 032:* There is no central configuration file listing available extensions; extensions themselves use code to provide a description of the extension and no config files**
Extensions are just discovered via classpath. There is no central configuration file listing all available extensions for the `jMeta` core. Furthermore, extensions themselves might provide a description, but this description is also not contained in a configuration file, but rather returned by the `IExtension` implementation directly in Java code.

Begründung: A central configuration of all available extensiosn for the `jMeta` core would be very inflexible. It would need to be extended anytime a new extension is available. And then, by whom? The end user, the developers of the extensions or even the `jMeta` developers? This would not make much sense.
For the extension specific details, a configuration file could be used, but it again makes it harder to create an extension and we would need additional generic code to parse the configurations. Configurabilty for such things is not necessary at all. If a change in the description is necessary, it would be related to changes in the extension anyway and the JAR for the extension can be updated to have the updated description. An XML file embedded in the JAR file would not bring any differences or even advantages.

Nachteile: No disadvantages known

Which functional interfaces do extensions provide in which way?

***DES 033:* Extensions can be queried dynamically by the library core to provide 0 to N implementations for an arbitrary Java interface**
`ExtensionManagement` provides a generic method in `IExtension` where `jMeta` core components can get all implementations of any Java interface they request which are provided by the extension. The extension of course might not have such implementations. But it might also have more than one implementation.

Begründung: Although in section "8.8 Extensions (Subsystem `Extension`)", specific cases for the extensibility of `jMeta` were mentioned, we do not want to hard-wire this into a generic component such as `ExtensionManagement`. If we would do this, there would be most probably a cyclic dependency from `ExtensionManagement` to these components of the `jMeta` core that are extensible and thus need to load extensions.
Furthermore, we do not need to change the interface of `ExtensionManagement` whenever extensions might need to return implementations for any other interfaces that can be extended in future.

Nachteile: No disadvantages known

What is the lifecycle of an extension, then, and is there anything else they should do?

***DES 034:* An extension is instantiated just once and then used throughout the whole application, offering a method to retrieve implementations**
According to DES 034, an extension is loaded at `jMeta` startup. It can be considered as singleton, as just a single instance of it is loaded using `ServiceLoader`. Extensions should have the sole purpose of returning new implementation instances, for which they implement the method `IExtension.getExtensionProviders`. Anytime this method is called, they provide a list of new implementations they provide for the given interface. If they do not provide an implementation, they return an empty list. Besides that, there is a method `getExtensionDescription` where extensions can provide additional detail information for the extension.
Extensions should not do anything else. Especially, they should not have any internal mutable state or perform any background processing.

Begründung: There is no reason why extension providers should implement any special behaviour in addition to just returning implementations to use by the core, except, maybe harmful hacking code.
Of course, `jMeta` cannot really prevent that, but at least it should be made clear.

Nachteile: No disadvantages known

We should now go into more detail how the library core interacts with extensions. So far, we defined that extensions will be loaded once at startup using the `ServiceLoader` facility. But what is the public interface of the `ExtensionManagement` component?

***DES 035:* `ExtensionManagement` provides the interface `IExtensionsManagement` which provides access to all extensions found**
`ExtensionManagement` provides the interface `IExtensionsManagement` for the library core with the method `getAllExtensions`. This method returns all available `IExtension` implementations found.

Begründung: An easy to use `ExtensionManagement` without any surprises.

Nachteile: No disadvantages known

11. Subsystem `Container API`

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 DATABLOCKS. It can be a file or a MEDIA 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 Media

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

> ***DES 036:*** **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 037:* 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 038:* 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 039:* 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 040: **Locking of files during `jMeta` access**
Files are *always* locked during access by `jMeta` explicitly. File content is 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 041: **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 042: **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-access files. 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 043:* 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 is 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 044: **`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 045: **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 means you have to do a lot of manual work to implement insertion, removal and replacement with this operation, it 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).
`AMed04`	Inserted data (requirement `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.

ID	Requirement	Motivation
AMed05	Replaced data (requirement 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.
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.
AMed08	It must be possible to insert multiple consecutive blocks of data	E.g. you want to add an APEv2 tag first, then an ID3v2.3 tag and then an ID3v1 tag.

ID	Requirement	Motivation
AMed09	It must be possible to insert a block of data before an existing block, then replace or remove this existing block with something else; the same must be possible for first remove or replace, then insert (i.e., the inverse order)	E.g. you want to add an APEv2 tag first, then replace the follow-up, already existing ID3v1 tag with some different content.
AMed10	It must be possible to modify (remove, insert, replace) a block of data, then remove an enclosing block of data	E.g. the code flow first decides it is necessary to add a new frame to an existing ID3v2.3 tag, modify some of its frames, remove another one; but then due to some decision in the code, it becomes necessary to remove the entire tag.
AMed11	It must be possible to modify (remove, insert, replace) a block of data, then replace an enclosing block of data	E.g. the code flow first decides it is necessary to add a new frame to an existing ID3v2.3 tag, modify some of its frames, remove another one; but then due to some decision in the code, it becomes necessary to replace the entire tag with different content.
AMed12	Except AMed10, Media does not need to support overlapping removes or replaces	Data blocks are never overlapping, except AMed10, there is no such case that you remove a block, then remove an overlapping portion of it.
AMed13	Media does not need to support modifications (remove, insert, replace) within an already removed or replaced block of data	It is not clear how to treat changes in an already removed or replaced parent object consistently, so we just reject them.

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

- *insert* N bytes at a given offset (adresses requirement `AMed01`)
- *remove* N bytes at a given offset (adresses requirement `AMed02`)
- *replace* N bytes at a given offset by M new bytes (adresses requirement `AMed03`). Thus, replace has three flavors:
  1. replace N bytes by $M > N$ new bytes actually behaves like an insertion, replacing the first N bytes with new ones, and inserting additional $M - N$ bytes behind. We call it an *inserting replace*.
  2. replace N bytes by $M < N$ new bytes actually behaves like a removal, replacing the first M bytes with new ones, and removing additional $N - M$ existing bytes behind. We call it a *removing replace*.
  3. replace N bytes by $M = N$ new bytes actually behaves like a usual write. We call it an *overwriting replace*.

Begründung: See requirements `AMed01`, `AMed02` and `AMed03`. 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

How is the padding requirement adressed? Like this:

***DES 047:* Requirement AMed06 must be adressed by components using Media, it is not built into Media itself**
Media does not implement AMed06 itself, it does not know something like "Padding", but just the three primitive change operations. Using them, a third party component can do the following using Media:

- If a new field is added to the ID3v2 tag (with an *insert*) and padding that is longer than the newly inserted content is present, the 3rd party component has to do a `remove` of a corresponding number of padding bytes, or a `replace` of old padding bytes by new (shorter) padding bytes.
- If a field is removed from the ID3v2 tag (with a *remove*), the 3rd party component can do an `insert` of a corresponding number of padding bytes at the end of the tag, or a `replace` of old padding bytes (if present) by new (longer) padding bytes.

Begründung: Media should stay clean of any "functional" terms defined by some formats, but it should just stick to the primitives it offers to modify media content. As shown above, the requirement can quite easily be implemented by tag-specific logic on top of Media.

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 048:* Writing operations on a medium can be undone with *undo* before a *flush*, adressing requirement AMed07**
Writing operations lead to pending changes according to DES 048. 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

This leaves open AMed04 and AMed05 as well as AMed08 to AMed13 which we will adress later when discussing the details of the two-stage write protocol.

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 048. A similar important question is: How can you make reading perform better?

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

***DES 049:* 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 one 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 necessary in average. Furthermore, depending on the fixed size, it would be necessary to read two or even several times when a bigger chunk of data is needed. 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 refer to the possibly long-term 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-readable metadata read it, convert it via `jMeta` into a clear-text representation and show this representation in their GUI. In this case the GUI model represents a kind of caching that also allows changes to this metadata in the GUI, it is not necessary to re-read again from the medium.

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 scenarios 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? Sometimes possibly yes. Instead, would you want a direct medium access again? Possibly yes or no.

The last question brings up another general problem with caching: The problem of synchronicity 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 identified advantages and disadvantages:

Advantages	**Disadvantages**
+ Performance improvement when reading the same data 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.	− External 3rd party 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 allocated memory freed?" must be answered. For consistency topics, even more complex code is necessary.
	− More 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` at all? Because some of the previous design decisions combine well with a caching approach:

- DES 049 can be achieved using a cache, as we see just a little later
- DES 049 can be implemented with a cache, i.e. anything that has been buffered should directly go into the cache for subsequent read actions

- DES 049 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.

Thus we decide:

DES 050: `Media` **keeps medium data read in a cache**
`Media` uses a RAM-based, potentially long-lived fast storage (cache) to store already read content of the MEDIUM. Subsequent read accesses request 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 050 in a sense of buffering when reading. i.e. the cache works as the buffer for DES 050
- Implementation of DES 050 can be done 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 050.

How to use caching to better achieve DES 050?

***DES 051:* Caching is used to better mitigate the differences between `InputStreams` and files according to DES 051.**
The data that has been read from an `InputStream` is 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 that all data up to the given higher offset is read and cached.

Begründung: This implements DES 051 nearly entirely, "transparent" to the user.

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

Of course, the disadvantages mentioned in DES 051 are heavy-weigth. If you wouldn't do anything about it to mitigate these disadvantages, then DES 051 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 052:* The user can disable caching entirely**
The user can disable caching entirely. Here, too, access to previously cached offsets is not possible for `InputStreams` and will be acquitted with 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 the possibility to 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 053:* The user can set the maximum size of the cache per medium, the cache keeps only the newest added data**
The user is allowed to limit the maximum size of the cache per medium by configuring it before creating the cache. The cache implementation ensures that at any time, the cache does not contain more bytes than which correspond to the maximum size. This is done using a FIFO (first in, first out) mechanism. I.e. the bytes added first are first discarded whenever trying to add new bytes to a cache.

Begründung: This way, the user can influence the maximum heap size required for caching of a single medium. At the same time, he needs not control the cache size himself, but it is internally managed by `Media`

Nachteile: .

No disadvantages known

***DES 054:* Performance drawbacks of `InputStream`s are explicitly documented**
Reducing differences between `InputStream`s and files by caching in accordance to DES 054 means: You must deal with the fact that you cannot store virtually unlimited `InputStream`s in a cache. For file access, the user can also choose between `FileInputStream` and more direct access via `RandomAccessFile`s. The performance drawbacks induced by using `FileInputStream` compared to random-access - which are introduced by a unified API according to DES 054 - are explicity described in the `jMeta` documentation. The mitigation mechanisms (setting maximum cache size, 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

***DES 055:* The user cannot free up cache data in a fine-grained way**
`Media` will not provide any mechanisms for the user to free up cached data himself in a fine-grained way (e.g. range-based).

Begründung: This functionality needs to be implemented and tested (additional effort). It is quite unlikely that it is actually needed when having implemented DES 055. Furthermore, when should the user free the data? A monitoring by the using code is necessary which also makes the using code more complex.

Nachteile: No disadvantages known

After these results, the disadvantages previously listed in table 11.2 and in DES 055 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 is here. To nevertheless avoid `OutOfMemoryError`s, we have defined DES 055 and DES 055 and thus give enough room for the user to avoid these situations.

- Data Corruption due to Out-Of-Synch Medium: The cache might receive updates that are not yet persisted on the external medium, as explicitly allowed by DES 055. 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 055. Even this cannot give a full protection for some OSs. The user is in any case informed about irresolvabe inconsistencies by a runtime exception.

Finally, we want to list a design decision rejected during a proof of concept, which was to provide a mechanism of skipping bytes for `InputStream`s instead of reading them into the cache, here is why:

DES 056: **The `Media` API does not provide the option to skip bytes of an `InputStream` instead of reading them into the cache**
It sounds like a good idea to offer the configurable possibility to skip bytes from an `InputStream` instead of unnecessarily reading them (and put them into a cache). This mainly applies for the case when you are currently at offset x, but now you want to read bytes from the stream starting at offset $x + 100$. You might never want to access the bytes between x and $x + 100$. So it would be worthwile to allow the possibility to simply skip them instead of reading them into memory and into the cache.
However, `InputStream.skip` is not as straightforward as one could wish. It does not guarantee to skip the number of bytes given, but it might skip fewer bytes or even return a negative number. It might also throw an `IOException`, which might or might not include the case of reaching the end of medium. Although not specified, there might be the same behavior as for read, that `InputStream.skip` might block. Furthermore the javadocs specify that it is only supported for streams that support seeking. All in all, this gives the impression that the method is highly platform and implementation-dependent. How to bring this implementation into a reliable form that guarantees to always skip the given number of bytes or block (possibly with timeout)?

Begründung: It is too complex to implement the method reliably, thus we omit it and only offer the possibility to read ahead until a given offset. All bytes read are possibly read into the cache. However, the cache is "self-cleaning" according to DES 056, if a maximum size is configured. This mechanism should be sufficient for achieving a moderate heap usage.

Nachteile: Probably more heap usage and bytes might be unnecessary read and kept in memory.

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

DES 057: **`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 057. However, we do not use it and implement another solution "by hand".

Begründung: It is not guaranteed, that each OS supporting Java also supports a `MappedByteBuffer`. It is also not guaranteed that the data "cached" is really present in RAM and thus accessible faster. Furthermore, it is indicated that 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 lack the desired benefits.

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

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

***DES 058:* 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 e.g. transactions mostly work for code accessing 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 different:

DES 059: **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 (or at least a cached) 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 060: **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. Of course, this might lead to an "end of medium" situation indicated by the method to the caller.

Begründung: Necessary for implementing DES 060. Of course one could ask: Why isn't it sufficient to just provide a single `getData` operation that returns data, either from the cache or from the medium. If the data was not yet in cache, it also adds it to the cache. The reason is: The code might not want the bytes yet! It just wants to give a statement like "At this point in time, I know how much bytes I might need to read later, so please already buffer them. But do not give the bytes to me, because at this point in the code I cannot really use them and it would complicate my code strucutre." For instance, if only providing one method returning the data, the code must pass the read data as parameter to other methods to read it. Instead of doing this, the code can just buffer the needed number of bytes, then call other code to get just the data it needs without needing to fiddle keeping track of the last read byte.

Nachteile: No disadvantages known

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

***DES 061:* Explicit operation to get medium bytes**
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, as *cache* only buffers it internally without returning it.

Nachteile: No disadvantages known

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

***DES 062: 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 the given range is entirely contained in it. Is it not entirely contained in the cache, *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.

Begründung: To ensure efficient reading, *getData* can be used as such to fetch data from the cache, if anyhow possible. Only if there is at leaast one byte not in the cache, the medium must be accessed directly. Updating the cache by read data is useful, to ensure later calls to query the same data can fully leverage the cache and do not need another access to the external medium.

Nachteile: No disadvantages known

Should we provide a way to force medium access when getting data?

***DES 063: getData* provides no specific mode or configuration to ignore the cache and always read from the medium**
getData does not provide a forced read mechanism, but it always only reads data from the external medium that it cannot find in the cache.

Begründung: It is not clear when using code should use the forced or the unforced mode. If the using code somehow can magically see that the underlying medium was changed by other processes, it could do a forced direct read. But in this case, everything is lost already, because you can never know what the other process did. A better way to ensure integrity and consistency would be for the using code to close the medium and reopen another access instance.

Nachteile: No disadvantages known

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

DES 064: **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

For efficiency reasons, we define the following:

DES 065: **Handle reading from a single already cached region or a size smaller than the maximum read-write block size as special case**
If a read offset and size for *getData* is exactly hitting a single cache region, and this single region fully contains the range to read, the `ByteBuffer` corresponding to this region with adapted position and limit is returned instead of allocating a new `ByteBuffer` and copying bytes, which will happen in any other case. Similarly, if a range is not cached and needs to be read, and if the range's size is below the maximum read-write block size, again no copying happens, but the read `ByteBuffer` is directly returned.

Begründung: The case of an already cached region fully covering the range to read can be considered as at least the use case covering 80% of the typical uses. The reason is: Usually, the users should use cached media. And usually, they should call *cache* before *getData*, covering e.g. a whole header. Thus, reading individual fields later with `getData` will always hit the cache. Thus, for this 80% case, we could optimize the library performance a lot by simply not copying.

Nachteile: Slightly more complexity and testing effort involved

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 it.

DES 066: `jMeta` **does not support any timeouts, neither reading nor writing**
`Media` does not offer the possibility to configure timeouts for any reading or writing actions, and likewise, it does not implement any measures to prevent these actions from blocking arbitrarily long

Begründung: Both for file or stream access, read and write operations, including determining where we are or truncation might or might not block. This is - unfortunately - OS and implementation dependent and cannot be predicted. It is possible and was tried to implement a parallel thread to execute the action against the medium and the main thread to monitor the time taken by the other thread and retrieve the result after a given timeout. While this is easily possible using Java's `Future`s, it is not so easy to really terminate the blocking action and its thread. In reality, on some OSs the action is really interruptible, on others it is not. This might also depend on the implementation. Thus, `jMeta` won't implement anything that would try to convince the user it can actually do it, leading also to increased complexity. Instead, the user must use his own mechanisms, i.e. an own custom `InputStream` implementation, using multithreading and monitoring of calls from the outside, or using implementation-specific timeouts such as for `SocketInputStream`.
Furthermore, it is also problematic what state the medium is in after the timeout was detected. What shall the library or the user do with this implementation? Retry, fail, close and retry? There is not always an easy answer to it.

Nachteile: Users might need to write more custom code themselves, if they strictly require this for mediums they use.

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 067:* Media are represented as interface and implementation class with following properties**
A medium is represented as Java interface `Medium` and allows users of `jMeta` to specify a concrete physical medium (i.e. the implementations of the interface `Medium`). As implementations we support a `FileMedium` according to DES 067, according to DES 067 aan `InputStreamMedium` and according to DES 067 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.
- Read-only: Yes/No - **Motivation:** This property disables writing in practice if set to "Yes". Some MEDIA 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 `Medium` except `InputStream`. Thus this should be implemented directly in the `Medium` 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 067, 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 `Medium` 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 `Medium` 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 068:* If a `Medium` implementation is writable, it must also be random-access**
Every in principle writable `Medium` 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 068.

Nachteile: No disadvantages known

Here is a very importante note for byte array media:

***DES 069:* 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 070:* Byte offsets are used for any kind of MEDIA**
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 070 and DES 070, 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 070).

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 071:* `jMeta` uses the interface `MediumReference` 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 `MediumReference` is located on the same medium behind of at the same position as this instance.
- `before`: Returns true if another `MediumReference` is located on the same medium before the position of this instance.
- `advance`: Creates a new `MediumReference` 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 072: **`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 "?? ??". 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 073: **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 `MediumReference` 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 `MediumReference` instances

2. Update already created `MediumReference` instances directly for each pend-

ing change registered (see DES 073)

3. Update already created `MediumReference` instances only when an explicit *flush* according to DES 073 occurs

Assume that `MediumReference` instances are not updated when writing. That means the user code remembers a position of an element in form of a `MediumReference` 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 `MediumReference` 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 074: **`Media` needs to automatically update `MediumReference` instances after medium changes**
All `MediumReference` 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 x be the start offset of a collection of n bytes (an "object"), 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 `MediumReference` instance after updating. Then the following detailed rules apply:

- `insert` *before the object start offset* x*:* Is $y \leq x$, then $\overline{x} := x + k$, including the case $y = x$.
- `insert` *behind the object start offset* x*:* Is $y > x$, then $\overline{x} := x$, i.e. such insertions of course do not change the offset of byte x. Insertions might happen before $x + n$, essentially splitting the object.
- `remove` *before the object start offset* x *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* x *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, i.e. $\overline{x} := y$.
- `remove` *behind the object start offset* x*:* 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.
- `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: `MediumReference`s are both handed out to the user as well as used to manage cache objects. If changes to the medium happens, all `MediumReference`s that are already in user hands or are internally used to manage data are rendered invalid. Thus, we would need to write code to detect such `MediumReference`s as invalid whenever used, or to invalidate the current cache content. Instead of doing so, we simply update all `MediumReference`s previously created according to the kind of change, thus practically still letting them point to the same "object" es before (despite deletion of the whole medium bytes as special case). The `MediumReference`s can thus be used further on.

Nachteile: A central management of `MediumReference` instances must be implemented (see DES 074).

As already indicated by DES 074 the automatic updating of already created `MediumReference` instances requires that only `Media` may create `MediumReference` instances. These must be managed in a kind of pool to be able to automatically update them in case of writing operations.

DES 075: **`MediumReference` instances are centrally managed by `Media` and cannot be directly created by users of the component**
The lifecycle of `MediumReference` 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 075. 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 `MediumReference` instances.

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

DES 076: **`MediumReference` instances are only updated after a *flush***
According to DES 076 `MediumReference` instances are automatically updated in case of medium changes. This automatic update only happens at *flush* time.

Begründung: Assumed that `MediumReference` 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 `MediumReference` 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 076 requires that offsets would need to be "re-adapted" if a pending changes is undone again. Again this is additional complexity.

If `MediumReference` 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 077: **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 077. 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

Semantics of Writing Operations

In DES 077, 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 how multiple *insert* operations behave:

***DES 078: insert* concatenates insertions in call order**

1. Step 1: *insert* at offset x, Step 2: *insert* at different offset: Both calls are allowed, they do not influence each other.
2. 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 "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.

Begründung:

1. See requirement `AMed01`
2. See requirement `AMed08`

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.

Nachteile: The second possible interpretation sketched above might lead to surprises in a few use scenarios.

We now look at how *insert* and *remove* interact with each other:

***DES 079:* *insert* is revoked by overlapping *remove*s, *insert*s within already removed regions are not allowed**
The following cases can be identified:

1. Step 1: *insert* at offset x, Step 2: *remove* n bytes at offset y, where x is not located within the removed range $[y, y+n)$: Both calls are allowed, they do not influence each other.
2. Step 1: *remove* n bytes at offset y, Step 2: *insert* at offset x, where x is not located within the removed range $[y, y+n)$: Both calls are allowed, they do not influence each other.
3. Step 1: *insert* at offset x, Step 2: *remove* at same offset: Both calls are allowed, they do not influence each other.
4. Step 1: *remove* at offset x, Step 2: *insert* at same offset: Both calls are allowed, they do not influence each other.
5. Step 1: *insert* at offset x, Step 2: *remove* n bytes at offset y, where x is contained in the removed range $[y, y+n)$: The insert is revoked by the remove, i.e. will not be executed during a *flush*.
6. Step 1: *remove* n bytes at offset y, Step 2: *insert* at offset x, where x is contained in the removed range $[y, y+n)$: The second call is rejected with an exception, as it is pointing into an already removed region.

Begründung:

1. See requirement `AMed01` and `AMed02`
2. See requirement `AMed01` and `AMed02`
3. See requirement `AMed09`
4. See requirement `AMed09`
5. See requirement `AMed10`
6. See requirement `AMed13`

Nachteile: Then, how to implement write requirement `AMed04`? See DES 079.

Now we are left with clarifying how *insert* and *replace* interact with each other. This is essentially the same as between *insert* and *remove*:

***DES 080: insert* is revoked by overlapping *replace*s, *insert*s within already replaced regions are not allowed**
The following cases can be identified:

1. Step 1: *insert* at offset x, Step 2: *replace* n bytes by m new bytes at offset y, where x is not located within the replaced range $[y, y+n)$: Both calls are allowed, they do not influence each other.
2. Step 1: *replace* n bytes by m new bytes at offset y, Step 2: *insert* at offset x, where x is not located within the replaced range $[y, y+n)$: Both calls are allowed, they do not influence each other.
3. Step 1: *insert* at offset x, Step 2: *replace* at same offset: Both calls are allowed, they do not influence each other.
4. Step 1: *replace* at offset x, Step 2: *insert* at same offset: Both calls are allowed, they do not influence each other.
5. Step 1: *insert* at offset x, Step 2: *replace* n bytes by m new bytes at offset y, where x is contained in the replaced range $[y, y+n)$: The insert is revoked by the replace, i.e. will not be executed during a *flush*.
6. Step 1: *replace* n bytes by m new bytes at offset y, Step 2: *insert* at offset x, where x is contained in the replaced range $[y, y+n)$: The second call is rejected with an exception, as it is pointing into an already replaced region.

Begründung:

1. See requirement `AMed01` and `AMed03`
2. See requirement `AMed01` and `AMed03`
3. See requirement `AMed09`
4. See requirement `AMed09`
5. See requirement `AMed11`
6. See requirement `AMed13`

Nachteile: Then, how to implement write requirement `AMed05`? See DES 080.

Saying all this, it is not possible to implement `AMed04` ("it is possible to change or remove parts of an already done insertion before a flush") and `AMed05` ("it is possible to change or remove parts of an already done replacement before a flush") by using subsequent inserts, removes and replaces to modify an already inserted or replaced block of data before a flush. This is of cause already clear when looking at DES 080: Offets of these operations always only refer to currently persisted data, so you e.g. cannot change a prior insert not yet flushed with a remove. Then how are these requirements met? Here is how it could be done:

DES 081: **Meeting requirements `AMed04` and `AMed05` is done outside of `Media`, yet using its *undo* feature**
`AMed04` and `AMed05` can only be met outside `Media`. Using code must detect that a parent of the changed field was already scheduled for insertion or replacement. If that is the case, they have to "re-compute" the overall insertion or replacment bytes, undo the previous insertion or replacement using `Media` and finally schedule a new insertion or replacement containing the changed bytes. Alternatively, the implementation using `Media` might freeze a data block for change once its scheduled for insertion or replacement and throws an exception if someone tries to modify it afterwards. The details of this are defined in the design of `DataBlocks`.

Begründung: `Media` could somehow support here by offering modification functions for already done *insert*s or *replace*s. However, this would also at least require the using code to detect the change in a parent, and also recompute. It would make the API more complex. Using *undo*, the using code has just slightly more work to do.

Nachteile: No disadvantages known

We already clarified how insert interacts with insert, remove and replace, in any order. We are now only left to clarify how multiple removes and replaces interact with each other. Multiple *remove*s interact with each other as follows:

***DES 082: remove* allows no overlaps by other *remove*s, only later calls with bigger region make earlier calls obsolete**
The following cases can be identified:

1. Step 1: *remove* n bytes at offset x, Step 2: *remove* m bytes at offset y without any overlaps to $[x, x+n)$: Both calls are allowed, they do not influence each other.
2. Step 1: *remove* n bytes at offset x, Step 2: *remove* $m \geq n$ bytes at offset $y \leq x$ with $[x, x+n) \subseteq [y, y+m)$: The second call to *remove* fully encloses the removed range of the previous *remove* call, so the previous call is revoked, i.e. it will not be executed during a *flush*.
3. Step 1: *remove* n bytes at offset x, Step 2: *remove* m bytes at offset y with $[y, y+m) \subseteq [x, x+m)$: The second region to be removed is fully contained in the region already removed by the previous call. The second call is rejected with an exception.
4. Step 1: *remove* n bytes at offset x, Step 2: *remove* m bytes at offset $y \in (x, x+n)$ with $y+m > x+n$: The second call is an overlapping call, that removes additional bytes behind the first remove call, but still includes some bytes of the region removed with the first call. The second call is rejected with an exception.
5. Step 1: *remove* n bytes at offset x, Step 2: *remove* m bytes at offset $y < x$ with $y > x, y+m < x+n$: The second call is an overlapping call, that removes additional bytes before the first remove call, but still includes some bytes of the region removed with the first call. The second call is rejected with an exception.

Begründung:

1. See requirement `AMed02`
2. See requirement `AMed10`
3. See requirement `AMed13`
4. See requirement `AMed12`
5. See requirement `AMed12`

Nachteile: No disadvantages known

Interactions between *remove* and *replace* are identical to this, as defined in the following design decision:

***DES 083:* *remove* allows no overlaps by *replace*s, only later calls with bigger region make earlier calls obsolete**
The following cases can be identified:

1. Step 1: *remove* n bytes at offset x, Step 2: *replace* m bytes by r new bytes at offset y without any overlaps to $[x, x+n)$: Both calls are allowed, they do not influence each other.
2. Previous case, exchange order of *remove* and *replace*: Same result
3. Step 1: *remove* n bytes at offset x, Step 2: *replace* $m \geq n$ bytes by r new bytes at offset $y \leq x$ with $[x, x+n) \subseteq [y, y+m)$: The replaced region of *replace* fully encloses the removed range of the previous *remove* call, so the previous call is revoked, i.e. it will not be executed during a *flush*.
4. Previous case, exchange order of *remove* and *replace*: Same result
5. Step 1: *remove* n bytes at offset x, Step 2: *replace* m bytes by r new bytes at offset y with $[y, y+m) \subseteq [x, x+m)$: The second region to be replaced is fully contained in the region already removed by the previous call. The second call is rejected with an exception.
6. Previous case, exchange order of *remove* and *replace*: Same result
7. Step 1: *remove* n bytes at offset x, Step 2: *replace* m bytes by r new bytes at offset $y \in (x, x+n)$ with $y+m > x+n$: The second call is an overlapping call, that replaces additional bytes behind the first remove call, but still includes some bytes of the region removed with the first call. The second call is rejected with an exception.
8. Previous case, exchange order of *remove* and *replace*: Same result
9. Step 1: *remove* n bytes at offset x, Step 2: *replace* m bytes by r new bytes at offset $y < x$ with $y > x, y+m < x+n$: The second call is an overlapping call, that replaces additional bytes before the first remove call, but still includes some bytes of the region removed with the first call. The second call is rejected with an exception.
10. Previous case, exchange order of *remove* and *replace*: Same result

Begründung: For (1.) and (2.): See requirement `AMed02` and `AMed03`. For (3.): See requirement `AMed11`; For (4.): See requirement `AMed10`. For (5.) and (6.): See requirement `AMed13`. For (7.) to (10.): See requirement `AMed12`.

Nachteile: No disadvantages known

Now we are just left to clarify how multiple *replace* operations interact with each other:

***DES 084: replace* allows no overlaps by other *replace*s, only later calls with bigger region make earlier calls obsolete**
The following cases can be identified:

1. Step 1: *replace* n bytes by r_1 new bytes at offset x, Step 2: *replace* m bytes by r_2 new bytes at offset y without any overlaps to $[x, x+n)$: Both calls are allowed, they do not influence each other.
2. Step 1: *replace* n bytes by r_1 new bytes at offset x, Step 2: *replace* $m \geq n$ bytes by r_2 new bytes at offset $y \leq x$ with $[x, x+n) \subseteq [y, y+m)$: The second call to *replace* fully encloses the replaced range of the previous *replace* call, so the previous call is revoked, i.e. it will not be executed during a *flush*.
3. Step 1: *replace* n bytes by r_1 new bytes at offset x, Step 2: *replace* m bytes by r_2 new bytes at offset y with $[y, y+m) \subseteq [x, x+m)$: The second region to be replaced is fully contained in the region already replaced by the previous call. The second call is rejected with an exception.
4. Step 1: *replace* n bytes by r_1 new bytes at offset x, Step 2: *replace* m bytes by r_2 new bytes at offset $y \in (x, x+n)$ with $y+m > x+n$: The second call is an overlapping call, that replaces additional bytes behind the first replace call, but still includes some bytes of the region replaced with the first call. The second call is rejected with an exception.
5. Step 1: *replace* n bytes by r_1 new bytes at offset x, Step 2: *replace* m bytes by r_2 new bytes at offset $y < x$ with $y > x, y+m < x+n$: The second call is an overlapping call, that replaces additional bytes before the first replace call, but still includes some bytes of the region replaced with the first call. The second call is rejected with an exception.

Begründung:

1. See requirement `AMed02`
2. See requirement `AMed11`
3. See requirement `AMed13`
4. See requirement `AMed12`
5. See requirement `AMed12`

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 085:* 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
- `MediumReference` 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.
- Validity: 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 085, 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 086:* 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 `MediumReference`, `Medium` 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 087:* Access to a medium is done using the `MediumStore`**
The reading and writing access to a medium (both random-access as well as byte stream according to DES 087) is offered via interface `MediumStore` 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(x, n)`	Buffers n bytes according to DES 087 and DES 087 permanently in the internal cache, starting at the given offset x. If caching is disabled (specifically for byte array media, see DES 087), this call is ignored. The method might reach the end of medium which it signals as an exception.	User code can prefetch data to work on it later, he himself gets the possibility to efficiently read and process data.	If x is bigger than the last read end position, the bytes up to the new offset are read and possibly cached. If x is smaller instead, a `InvalidMediumReferenceException` is thrown. The last read end position is advanced correspondingly.
`getCachedByteCountAt(x)`	According to DES 087 and DES 087, it provides the number of bytes that are consecutively cached at offset x.	See `cache`. User (and test) code must be able to additionally check whether enough data is already buffered or not.	See random-access.
`getMedium()`	Returns the `Medium` instance this `MediumStore` is associated with.		
`getData(x, n)`	Reads n bytes at offset x according to DES 087, DES 087, DES 087.	Calls `cache` to ensure all bytes are cached, then returns the data in a consolidated `ByteBuffer`.	Same as for random-access media. This indicates that this method also might throw an `InvalidMediumReferenceException`.

Operation	**Description**	**Features and motivation random-access**	**Features and motivation `InputStream`**
`isAtEndOfMedium(x)`	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.
`insertData(x, data)`	Implements the writing operation *insert* according to DES 087, DES 087, DES 087, DES 087, DES 087: Adds data at the given offset x, 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`
`removeData(x, n)`	Implements the writing operation *remove* according to DES 087, DES 087, DES 087, DES 087, DES 087: 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`

Operation	Description	Features and motivation random-access	Features and motivation `InputStream`
`replaceData(x, n, data)`	Implements the writing operation *replace* according to DES 087, DES 087, DES 087, DES 087, DES 087: 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 087: 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 087: While `insertData()`, `removeData()` and `replaceData()` only write into a temporary buffer, this call directly writes to the external medium.	`ReadOnlyMediumException`
`createMediumReference(x)`	Creates a new `MediumReference` instance for the given offset x according to DES 087	Is needed for random-access	Is needed for `InputStreams`
`undo(mediumAction)`	Undoes the changes of the given `MediumAction` according to DES 087, as far as it is still pending.	Is needed for random-access	`InvalidMediumActionException`

Operation	**Description**	**Features and motivation random-access**	**Features and motivation `InputStream`**
`open()`	Opens access to the medium	Explicit open instead of constructor better for testing	Explicit open instead of constructor better for testing
`close()`	Closes all internal resources according to DES 087, clears the complete cache contents and other internal data structures	See DES 087	See DES 087
`isClosed()`	Allows users to check if the medium is already closed and thus cannot be accessed anymore (returns true), or it is still accessible (returns false).	See DES 087	See DES 087

Table 11.3.: Operations of the `Media` API

The component interface

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

DES 088: **`MediaAPI` is the central entry point with creation functions for `MediumStores`**
The interface `MediaAPI` offers the central entry point for the component `Media`. Using the method `createMediumStore()`, users can create an `MediumStore` instance.

Begründung: The necessity for a further interface in addition to `MediumStore` is clear enough: A `MediumStore` 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 `MediumStore` 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 088 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 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.	`MediaAPI .createMediumStore()`
Unknowm media type	A `Medium` 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.	`MediaAPI .createMediumStore()`
Write to read-only medium	The user-provided `Medium` 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.	`MediumStore .flush()`, `MediumStore .insertData()`, `MediumStore .removeData()`, `MediumStore .replaceData()`
Consecutive write calls overlap	Concecutive calls to `insertData`, `removeData` or `replaceData` before a `flush` overlap in an invalid way, see DES 088, DES 088, DES 088, DES 088, DES 088	Is acquitted with an `InvalidOverlappingWriteException` runtime exception.	`MediumStore .removeData()`, `MediumStore .replaceData()`

Error Scenario	**Description**	**Reaction** `jMeta`	**API method**
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.	`MediumStore .cache()`, `MediumStore .getData()`
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.	`MediumStore .cache()`, `MediumStore .getData()`

Error Scenario	**Description**	**Reaction** `jMeta`	**API method**
Stream data not available	Data requested with `getData()` for a given offset is not available (anymore) 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.	`MediumStore .getData()`
Unknown or invalid `MediumAction`	The user passes an unknown or invalid `MediumAction` to any operation	This is an abnormal situation and is acquitted with the runtime exception `InvalidMediumActionException`	`MediumStore.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.	`MediumStore .cache()`, `MediumStore .getData()`, `MediumStore .isAtEndOfMedium()`, `MediumStore .flush()`
The `MediumStore` 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 089: **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 Design

Medium Access

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

DES 090: **Interface `MediumAccessor` with implementations for each MEDIA type access**
Access to a MEDIUM is possible via an interface `MediumAccessor`, doing a great deal of ensuring DES 090. It offest the following primitives:

- `getMedium`: Returns the medium this instance is working on.
- `open`: Opens the medium for access. Opening creates an exclusive lock on the medium, as far as the medium supports this (see DES 090, DES 090, DES 090).
- `isOpened`: Check if the medium is opened.
- `close`: Close the medium for access. A closed medium cannot be used anymore.
- `setCurrentPosition`: Sets the current position for the next `read`, `write`, `truncate` or `isAtEndOfMedium` call. Is ignored for non-random-access media.
- `getCurrentPosition`: gets the current position on the medium where the next `read`, `write`, `truncate` or `isAtEndOfMedium` will be executed.
- `read` at the current position: Read n bytes from external medium by explicit access from the current position, return the bytes in a `ByteBuffer`. Advances the current position by the number of read bytes.
- `write` at the current position: Writes n bytes to the external medium at the current position passed in a `ByteBuffer`. Throws an exception for read-only media, especially for `InputStream`. Advances the current position by the number of read bytes.
- `truncate` to new end offset at the current position: Truncates the medium to end at the current position. Throws an exception for read-only media, especially for `InputStream`.
- `isAtEndOfMedium`: Check if the current position is at end of the MEDIUM.

There is one implementation of this interface for each distinct MEDIUM type. `MediumStore` exclusively accesses the MEDIUM only via this interface.

Begründung: `MediumStore` 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.
Regarding the offset handling for read: This is necessary to ensure readable and non-confusing use for non-random-access media. In that way, the user does not assume the medium is actually read-only.

Nachteile: No disadvantages known

Management of `MediumReference` instances

According to DES 090, `MediumReference` instances must be maintained centrally by `MediumStore`. At the same time, we can have multiple `MediumReference` 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 090. In the end, the `MediumReference` instances must be automatically updated with `flush()`, as described in DES 090 and DES 090.

Thus, these things follow:

DES 091: **No pooling of `MediumReference` instances for the same offsets is possible**
In contrast to Java strings, the reuse of `MediumReference` instances in `createMediumReference()` for same offsets is not possible, i.e. if an `MediumReference` instance for offset x has already been created, the same instance cannot be returned for the same offset on the next call. A new `MediumReference` 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 `MediumReference` 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 `MediumReference` objects, there could be a bigger memory footprint.

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

DES 092: **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 `MediumReference` objects.

Begründung: Dupllicates must be possible due to DES 092. 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 `MediumReference` instances, than cache entries or pending changes, and on the other hand cache and pending change list could be cleared, while the `MediumReference` 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 092 in any way sorted?

***DES 093:* Unsorted ArrayList for keeping the `MediumReference` instances**
We use an `ArrayList` as data structure for maintaining all `MediumReference` instances. Insertion of `MediumReference` 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 `MediumReference` 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 `MediumReference` instance is usually much more commonly done than a `flush()`. Following approaches have been rejected especially:

- A `Map<Long, List<MediumReference>>` with offsets as key and all `MediumReference` 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 `MediumReference` instances again, or would need to be updated expensively.
- A `TreeSet<MediumReference>` can be excluded as it does not allow duplicates.
- A combination of a `TreeSet` with a special `Comparator`, which assumes `MediumReference` 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 `MediumReference` 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 `MediumStore` we decide:

DES 094: **The class `MediumReferenceFactory` is used for maintaining `MediumReference` instances**
`MediumReferenceFactory` implements the design decisions DES 094, DES 094 as well as DES 094. It offers following methods:

- `createMediumReference()` to create `MediumReference` instances
- `updateReferences()` for implementation of DES 094, 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
- `clear()` removes all maintained instances

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

Nachteile: No disadvantages known

The last question for `MediumReference` instances that still has to be answered: How to deal with the method `advance()` mentioned in DES 094. It creates `MediumReference` instances. Do they have to be maintained in `MediumReferenceFactory`, too?

DES 095: **The `MediumReference` instances created with `MediumReference.advance()` need to be maintained with `MediumReferenceFactory`, too.**
Initially, the new `MediumReference` instance must get a reference to its creator, i.e. `MediumReferenceFactory`, during construction. To still enable a simple creation of instances of `MediumReference` implementation classes (e.g. in unit tests), the constructor is public.

Begründung: The client code can arbitrarily use `MediumReference.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 `MediumReference` and `MediumReferenceFactory`, as both are knowing each other.

Internal Data Structures for Caching

The cache on first sight maintains data bytes per offset, always assuming that the data in cache is exactly identical to the data on the external medium, according to DES 095. It was already defined that there is an explicit `MediumStore.cache()` method for reading data into the cache and a method `MediumStore.getCachedByteCount()` for querying connected bytes from the cache (see table 11.3, as well as DES 095 and DES 095). Querying data from the cache is done using `MediumStore.getData()`, which can on the one hand skip the cache, on the other hand it can automatically update the cache with missing data (DES 095 and DES 095). Finally, the maximum cache size can be set or caching can even be entirely inactive (see DES 095 and DES 095).

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. The make the code easier to understand and maintain, we create a specific class for representing the so called regions, i.e. added cache fragments:

DES 096: **Class `MediumRegion` for consecutive regions of a medium, especially also for cache regions**
The class `MediumRegion` represents a consecutive byte range with start offset, size and contained bytes of a medium, that may or may not be cached. It offers the following methods:

- `getStartReference()`: Query start `MediumReference` 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 `MediumReference` is contained within the region, false otherwise
- `overlapsOtherRegionAtBack()`: Returns true if the this region overlaps the other region at its back, i.e. shares bytes with it at its end, otherwise returns false.
- `overlapsOtherRegionAtFront()`: Returns true if the this region overlaps the other region at its front, i.e. shares bytes with it at its front, otherwise returns false.
- `getOverlappingByteCount()`: Returns the number of bytes that this region shares with another region or zero if it does not share any bytes with it.
- `split()`: Splits this region into two regions at the given offset

Begründung: The cache regions and the non-cached regions of the medium can be treated in the same way. Implementing caching is easier when not based on primitive types (e.g. byte[]) only, but using this helper class.

Nachteile: No disadvantages known

Maintenance of cache content is done by a helper class:

***DES 097:* Cache maintenance by `MediumCache`, ensuring maximum cache size and no overlapping regions**
Cache maintenance is done by the class `MediumCache`. This class has the following invariants at any point in time before and after public method calls:

- The `MediumRegion`s stored in the cache do never overlap
- The maximum cache size is never exceeded
- The maximum size of a cache region is never exceeded by any region

It offers the following methods:

- `getAllCachedRegions()`: Returns a list of all cached `MediumRegion` instances currently contained in this cache, ordered by offset ascending
- `getCachedByteCountAt()`: Returns the number of bytes cached consecutively starting from offset x
- `getRegionsInRange()`: Returns a list of `MediumRegion` instances overlapping 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
- `addRegion()`: Adds a `MediumRegion` to the cache. Previously cached data is overridden.
- `clear()`: Frees all data in the cache
- `getMaxRegionSizeInBytes()`: Returns the maximum size of a region in bytes. Default is `Integer.MAX_VALUE`, which is represented by a constant named `UNLIMITED_CACHE_REGION_SIZE`. The cached regions will have at most the given size.
- `getMaxCacheSizeInBytes()`: Returns the maximum size of the cache in bytes. Default is `Long.MAX_VALUE`, which is represented by a constant named `UNLIMITED_CACHE_SIZE`. The cache size will at no time exceed this number of bytes.
- `getCurrentCacheSizeInBytes()`: Returns the current size of the cache in bytes.

Begründung: Reducing complexity of `MediumStore`

Nachteile: No disadvantages known

For the cache sizes, we saw already that we can query the maximum and current cache size as well as the maximum region size. However, why can we not change the maximum sizes dynamically?

***DES 098:* The maximum cache and maximum region size of a cache instance can never be changed throughout its life time.**
The maximum cache size and the maximum region size are passed to the constructor of the `MediumCache` class and they must not be changed later.

Begründung: Otherwise complex methods for reorganizing the cache are necessary, splitting and discarding existing regions. It is very unlikely that the user needs to change the cache size during accessing the medium. It if fully sufficient that he can configure the maximum size of the cache and its regions before the first access to the medium.

Nachteile: No disadvantages known

How are the cache regions internally managed?

***DES 099:* A `TreeMap` is used for managing the cache contents, an additional `LinkedList` for freeing up according to FIFO**
A `TreeMap<MediumReference, MediumRegion>` is used for managing the cache contents. An additional `LinkedList` is used for freeing up cache data according to FIFO.

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 $O(n)$.
The `LinkedList` is necessary to efficiently remove the first added cache regions when freeing up is necessary due to max cache size reached, see DES 099.

Nachteile: No disadvantages known

We should also think about cache fragementation. Let us assume that, by subsequent calls to `addRegion()`, we would add a single byte to the cache 20 times for a consecutive offset range. Will this result in 20 different `MediumRegions` with a length of one byte each? The same question arises for calls to `MediumStore.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 `MediumStore.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

`MediumStore.getData()` will then add regions 1, 3 and 5 to the cache, according to DES 099. So, do we have 5 `MediumRegions` in the cache after the call? Another extreme case is that the user reads 20 single bytes each with an offset gap of just one byte to the next one. This would result in 20 cache regions each with a size of one byte.

To ease the implementation complexity, we nevertheless decide:

***DES 100:* Fragmentation of cache regions is not actively avoided**
`Media` does not try to mitigate or avoid the following situations:

- Direct consecutive cache regions are not joined, even if their total size is smaller than the maximum allowed region size
- "Nearby" but unconnected (i.e. non-consective) cache regions of small sizes are not handled in any special way.

Begründung: It would lead to even higher complexity of the implementation of `addRegion()`.
In contrast to that high complexity, we can ask the caller to ensure that 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 101:* `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 `inserts` 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 `inserts` that are allowed to refer to the same offset (see DES 101).
Furthermore, it is also important for other types, as `insert` allows a later `remove` or `replace` at the same offset (see DES 101).

Nachteile: No disadvantages known

We now have to define comparisons between two `MediumActions` in a reasonable

way, as sorted data structures for efficient retrieval and iteration in order can be used correspondingly:

DES 102: **Comparisons of `MediumActions` is done based on `MediumReference` 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 `MediumReference` of `a` is smaller than the `MediumReference` of `b`,
- Or the `MediumReference` of `a` is equal to the `MediumReference` 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 `MediumReference` is bigger, or for equal `MediumReferences`, if the sequence number is bigger. Furthermore it is defined: If `MediumReferences` and sequence numbers are identical, then `a` is still bigger than `b` in the case that any of the other attribute of the `MediumActions` differs.

Begründung: Sorting of `MediumActions` should be done according to their order on the medium (i.e. their `MediumReferences`) and creation order (i.e. their sequence number), because behaviour of consecutive operations is based on call order, according to DES 102, DES 102 and DES 102. In addition, the `MediumActions` 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 `MediumActions` with same offset always get different sequence numbers that strictly increase with creation order.

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

DES 103: **`MediumActions` are stored in a sorted data structure without duplicates**
The `MediumActions` created before a `flush()` are held in a datastructure sorted by offset and sequence number (according to DES 103), which does not allow duplicates (e.g. `TreeSet`).

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

Nachteile: No disadvantages known

Now regarding management of MediumActions:

***DES 104:* For managing MediumActions, the class MediumChangeManager is responsible**
For managing MediumActions, the class MediumChangeManager is defined, which internally implements DES 104, with following methods:

- scheduleInsert() for scheduling an *insert*, implements DES 104, 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 104, 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 104, 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 104
- 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 104 and DES 104.

Begründung: Reduction of overall complexity of MediumStore.
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 105: **`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 `MediumReference`, 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 105 and DES 105) 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 `MediumStore` 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 `MediumReference` 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 chunk, as it could lead to `OutOfMemoryError` for large files, thus block-wise reading is necessary
- For this block-wise reading, 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 the medium. 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 offset $x + k$ and so on, until the last byte of 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 106: **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 107: **`createFlushPlan()` in `MediumChangeManager` 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 followed 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 addition 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 is the basis for processing in `flush` 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 might not yet be known when the action is created, in contrast to `REPLACE`.

Nachteile: No disadvantages known

Next, we should clarify how the cache is involved in `flush()`. In case caching is enabled for the medium, we have the following points to look at:

- If bytes are added to the medium with `INSERT` or `REPLACE`, these bytes must also be added to the cache
- If bytes are removed from the medium with `REMOVE` or `REPLACE`, these bytes must also be removed from the cache
- Is specific handling of `TRUNCATE` necessary?

- If bytes behind a change must be read (and written), they should also be directly added to the cache to save future medium accesses in these ranges

- Furthermore, if bytes behind a change must be read, it must also be checked if they are already contained in the cache, such that an explicit `READ` on the external medium is only necessary if these bytes are not cached

For the last point, we define the following:

DES 108: **`READ` operations in `flush` prefer cached data and update the cache if not already contained**
Whenever `flush` processes a `READ` action coming from the flush plan, it first checks if the data to read is fully cached. If it is, it takes the data from the cache. If it is not, it reads the data directly from the external medium and adds it to the cache.

Begründung: We want to use cached data as much as possible and avoid unnecessary medium accesses.

Nachteile: No disadvantages known

Correspondingly, this is how we treat `INSERT` and inserting `REPLACE` in `flush` regarding caching:

DES 109: **`INSERT` and inserting `REPLACE` operations in `flush` add their data to the cache**
Whenever `flush` processes an `INSERT` or inserting `REPLACE` action coming from the flush plan, it adds all bytes to insert or replacement bytes from these actions to the cache.

Begründung: We want to use cached data as much as possible and avoid unnecessary medium accesses.

Nachteile: No disadvantages known

Very similar for `REMOVE` and removing `REPLACE`:

***DES 110:* REMOVE and removing REPLACE operations in flush remove their ranges from the cache**
Whenever flush processes a REMOVE or removing REPLACE action coming from the flush plan, it removes all bytes to remove or replace from the cache. Of course, the replacement bytes of a REPLACE action are still newly added to the cache.

Begründung: Avoid cache corruption if old data is still present. In best case, the code directly fails when trying to add new data, in worst case the stale removed data remains in the cache and is later returned by read operations despite the medium already changed.

Nachteile: No disadvantages known

Do we have to do anything with the cache regarding a TRUNCATE action? This is answered in the following design decision:

***DES 111:* For a TRUNCATE, nothing is to be done related to caching**
The cache does not require any updates when processing a TRUNCATE action in flush.

Begründung: Although it first might seem that, because TRUNCATE shortens the medium, no cache updates are necessary for any cached bytes at the end of the medium. This is because the cache is implicitly cleaned by handling the REMOVE that leads to the truncate. This consists of removing cached bytes for the removed region as well as updating MediumReferences of any data cached behind the removed region.

Nachteile: No disadvantages known

We still have the topic to update any already created MediumReferences when processing INSERTs, REMOVEs and REPLACEs according to design decision DES 111. When do we need to perform these changes? This actually depends on the action type:

DES 112:* For INSERTs and inserting REPLACEs, MediumReferences are updated *BEFORE* the cache is updated, for REMOVEs and removing REPLACEs they are updated *AFTERWARDS
For INSERTs and inserting REPLACEs, any MediumReferences already created and maintained in the MediumReferenceFactory are updated first, which essentially increases the offsets of any MediumReferences behind the insertion offset by the number of bytes inserted. Only after this, the data newly inserted or replaced can be added to the cache.
In contrast to this, when REMOVEs and removing REPLACEs are processed, first of all, the bytes to remove, if present in the cache, need to be removed from it. Only then, all MediumReferences behind the remove offset need to be decreased by the number of removed bytes (or fall back to remove offset, if within the removed range).

Begründung: For INSERTs and inserting REPLACEs: If we add the inserted data to the cache before updating reference, the following would happen: Assume there are bytes already cached directly starting at the insertion offset. They would be superseded by the inserted data, i.e. thrown out of the cache. Then only all remaining data with higher offsets in the cache would be updated. This way, we would essentially lose cached data unnecessary and increase the complexity of cache management needed at runtime a bit.
For REMOVEs and removing REPLACEs: If we would first update the MediumReferences here, the following could happen: Assume we have two consecutive cached regions within the region to remove. For both regions, its offset would be updated to point to the remove offset. So we would have two regions cached with the same start offset, which leaves the cache inconsistent. Instead, we first remove all cached regions within the removed range, then we update any remaining regions (those behind the removed range).

Nachteile: No disadvantages known

TODO hier weiter
Now we have all credentials to implement writing in `flush()`:

1. Create the read-write-plan according to DES 112

2. Iterate all actions of the read-write-plan, the following steps are done for each action.

3. Execute the MediumAction at the indicated offset, we distinguish the following cases:

   - 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 MediumAccessor. Then a corresponding ByteBuffer

is built up step-wise. If an `EndOfMediumException` occurs, this is considered as impossible and an `IllegalStateException` is thrown.

- If action = `WRITE`: If the `WRITE` action contains bytes already, it as associated to an `INSERT` or `REPLACE`. In this case, directly write the contained bytes with direct access to the medium via `MediumAccessor`. If the action does not contain any bytes, the previous action must have been a `READ` reading exactly the same amount of bytes to write. If this is not the case, an `IllegalStateException` is thrown. Otherwise these previously read bytes are written with direct access to the medium via `MediumAccessor`.

- If action = `TRUNCATE`: Execute an explicit truncation of the medium.

- If action = `INSERT`, `REPLACE` or `REMOVE`: Do nothing, as these operations are implicitly implemented via `READ`, `WRITE` and `TRUNCATE` operations.

4. If action = `INSERT`, `REPLACE` or `REMOVE`: Call `MediumReferenceFactory.updateReferences` for the region, such that all consecutive `MediumReference` instances are updated.

5. Update the cache:

   - If action = `WRITE`, then call `MediumCache.addRegion()` to add the bytes written to the cache.

   - If action = `REMOVE`, then call `MediumCache.removeRegion()` to remove the bytes also from the cache.

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

DES 113: **`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 `MediumReference` instances behind must explicitly be executed. For this, you need the concrete types, `WRITE` is not sufficient.
Throwing an `IllegalStateException` in case of end of medium encountered is necessary, as the creation of the flush plan must have considered the medium size. Thus, if it occurs, this is either a programming error in `createFlushPlan` or another process as changed the medium meanwhile.
`undo()` is only executed for user operations, as only those are maintained in the internal data structures of `MediumChangeManager`, according to DES 113.
The cache update is divided into two parts: In case of a `REMOVE`, *first* the cache is updated, and *then* the `MediumReference` 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 `MediumReference` instances.
The cache management is completely done within `MediumCache.addRegion()`, such that maximum size and consecutive regions 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 113 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 rescued? This is clarified by the following design decision.

***DES 114:* 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 failure 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 implementation of `createFlushPlan` is anything but trivial. The general algorithm is formally described in a separate paper (including examples), see [**?**].

Here, we only list some testcases that should be implemented to demonstrate the intended behaviour, see table 11.5. If we say

contains read/write-blocks from `<startOfs>` *to* `<endOfs>` `<backwards|forwards>`

in the table, we are referring to the following:

- The generated flush plan contains a sequence of `READ` and `WRITE` pairs starting at `start offset`, each `WRITE` following a `READ` with the same size.
- Each `READ` and each `WRITE` have a maximum size $s \leq m$, where m is the configured maximum read-write block size according to DES 114
- With $N :=$ endOfs $-$ startOfs, there are exactly

$$\frac{N}{m} + N \bmod m$$

such pairs present

- They are either read "backward", i.e. chunk-wise starting at highest offset down to the lowest offset of the range, or "forward", i.e. chunk-wise starting at lowest offset up to the highest offset of the range

Also note that these tests do not contain any negative cases for scheduling actions that are overlapping in any invalid way. This must already be tested for the schedule operations of `MediumChangeManager`.

ID	Testcase	Variations	Expectation
CF0	No operation	-	The plan created is empty
CF1	Single `insert` n bytes at offset x	a. x is start, intermediate or end offset of the medium b. Bytes behind: None, or whole-numbered multiple of the maximum write block size m, or no whole-numbered multiple of m, or fewer bytes than m 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 n towards higher offsets, such that medium length grows by n • Contains read/write-blocks from x to the end of medium backwards • The `insert` action follows after these actions in the plan

ID	Testcase	Variations	Expectation
CF2	Single `remove` n bytes at offset x	a. x is start, intermediate offset or $x+n$ is the end offset of the medium b. Bytes behind: None, or whole-numbered multiple of the maximum write block size m, or no whole-numbered multiple of m, or fewer bytes than m c. Extreme case: All bytes of the medium are removed	• No read/write operations before remove offset • Bytes behind remain unchanged and get shifted by n towards smaller offsets, such that medium length shrinks by n • Contains read/write-blocks from x to the end of medium forwards • 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` of n bytes by r new bytes at offset x	a. x is start, intermediate offset or $x+n$ is the end offset of the medium b. Bytes behind: None, or whole-numbered multiple of the maximum write block size m, or no whole-numbered multiple of m, or fewer bytes than m c. Replacement bytes: Same cases as for bytes behind d. r smaller, equal to or bigger than n e. Extreme case: All bytes of the medium are replaced	• No read/write operations before replace offset • If $n > r$: Bytes behind remain unchanged and get shifted by $n-r$ towards smaller offsets, such that medium length shrinks by $n-r$, • If $r \geq n$: Bytes behind remain unchanged and get shifted by $r-n$ towards higher offsets, such that medium length grows by $r-n$, • Contains read/write-blocks from $x+n$ to the end of medium forwards (if $n > r$) or backwards (if $r > n$) • The `replace` action follows after these actions in the plan • If $n > r$: At the end there is a `truncate` operation

ID	Testcase	Variations	Expectation
CF4	Multiple `insert`s of in total n bytes	a. At same or different offsets b. Bytes in-between/behind: None, whole-numbered multiple of the maximum write block size m, or no whole-numbered multiple of m, or fewer bytes than m c. Bytes to insert: Same cases as for bytes behind	• No read/write operations before the first insert offset • Bytes behind last insertion remain unchanged and get shifted by n towards higher offsets, such that medium length grows by n • Bytes in between insertions: remain unchanged and are shifted to the back by the number of up-to-then inserted bytes • Each range between or behind inserts contains read/write-blocks from insertion offset x_i to the start of the next insertion (or end of medium) backwards • Each `insert` action follows after these block actions in the plan

ID	Testcase	Variations	Expectation
CF5	Multiple `removes` of in total n bytes	a. All removes refer to consecutive regions or have gaps between b. Bytes in-between/behind: None, whole-numbered multiple of the maximum write block size m, or no whole-numbered multiple of m, or fewer bytes than m c. Extreme case: All bytes of the medium are removed	• No read/write operations before the first remove offset • Bytes behind last remove remain unchanged and get shifted by n towards smaller offsets, such that medium length shrinks by n • Bytes in between removes: remain unchanged and are shifted to the front by the number of up-to-then removed bytes • Each range between or behind removes contains read/write-blocks from offset $x_r + n_r$ up to the start of the next remove (or end of medium) forwards • Each `remove` action follows after these block actions in the plan • At the end there is a `truncate` operation

ID	Testcase	Variations	Expectation
CF6	Multiple `replace`s of in total n bytes by in total r new bytes	a. All replaces refer to consecutive regions or have gaps between b. Bytes in-between/behind: None, whole-numbered multiple of the maximum write block size m, or no whole-numbered multiple of m, or fewer bytes than m c. Replacement bytes: Same cases as for bytes behind d. r smaller, equal to or bigger than n e. Extreme case: All bytes of the medium are replaced	• No read/write operations before the first replace offset • If $n > r$: Bytes behind remain unchanged and get shifted by $n-r$ towards smaller offsets, such that medium length shrinks by $n - r$, • If $r \geq n$: Bytes behind remain unchanged and get shifted by $r-n$ towards higher offsets, such that medium length grows by $r - n$, • Bytes in between replaces: remain unchanged and are shifted to the front or back by the number of up-to-then replaced minus inserted bytes • Each range between or behind replaces contains read/write-blocks from offset $x_r + n_r$ up to the start of the next replace (or end of medium) forwards or backwards • Each `replace` action follows after these block actions in the plan • If $n > r$: At the end there is a `truncate` operation

ID	Testcase	Variations	Expectation
CF7	Multiple `remove`s and `insert`s	a. At same offsets, remove first then consecutive insert or at different offsets b. Mutually compensating or not c. First `remove`, then `insert`, or first `insert`, then `remove` - Note that remove and insert in any order at the same offset can be considered consective d. Extreme case: All bytes of the medium are removed, then new bytes inserted	• For read/write blocks and bytes between and behind: Same behaviour as for other test cases • For mutually compensating actions: Bytes behind the last action must not be read or written • The order of actions does not matter • For the extreme case: All bytes are actually "replaced" by the new inserted bytes

ID	Testcase	Variations	Expectation
CF8	Multiple `replace`s and `insert`s	a. At same offsets, replace first then consecutive insert or at different offsets b. Mutually compensating or not c. First `replace`, then `insert`, or first `insert`, then `replace` - Note that remove and insert in any order at the same offset can be considered consective d. r smaller, equal to or bigger than n e. Extreme case: All bytes of the medium are replaced, then new bytes inserted	• For read/write blocks and bytes between and behind: Same behaviour as for other test cases • For mutually compensating actions: Bytes behind the last action must not be read or written • The order of actions does not matter • For the extreme case: All bytes are actually "replaced" by the replacement bytes as well as the new inserted bytes

ID	Testcase	Variations	Expectation
CF9	Multiple `removes`s, `replace`s and `insert`s	a. At same offsets b. Mutually compensating or not c. In different orders d. Growing or shrinking the medium	• For read/write blocks and bytes between and behind: Same behaviour as for other test cases • For mutually compensating actions: Bytes behind the last action must not be read or written • The order of actions does not matter

Table 11.5.: Test cases for checking `createFlushPlan`

Configurating Medium Access

So far, we have identified several mechanisms the user can use to influence the behaviour of medium access: He can set the maximum cache size, he can disable caching alltogether, and he can also change the maximum read-write block size. But how? In DES 114, we explicitly said that for now, `jMeta` will not offer the possibility to change such parameters dynamically. Still we have to offer the user the corresponding API without exposing him to the internal details of the implementation.

We thus first define:

DES 115: **The configuration of the medium access parameters is done per `Medium` instance**
The configuration of `Media` is done per `Medium` instance. By doing this, all configuration parameters correlate to an `Medium`, have the same lifetime and scope. The setting of the parameters is done by passing them directly to the constructor of the media created by the user, with constructors setting the parameters to sensitive defaults.

Begründung: The whole internal implementation of the most important classes, i.e. `MediumStore`, `MediumAccessor`, `MediumCache` and so on works on exactly one medium. The user can create `Medium` instances directly himself and configure them as needed, independent of other `Medium` instances. As these parameters must not change after creation of the medium, it is correct to pass them to the constructor and to not offer setters.

Nachteile: No disadvantages known

Now the question arises: Which configuration parameters are needed in detail? We summarize them again in detail in table 11.6.

First of all, we exclude one potential parameter:

DES 116: **The maximum cache region size is not configurable by the user, but is automatically set to the maximum read-write block size**
The user can only configure the maximum read-write block size, but not the maximum cache region size which is just set to the configured maximum read-write block size.

Begründung: For a usual user, it is not clear what the maximum cache region size really is and which size it should have; there is not much "tuning" potential in setting it to another value than the maximum read-write block size. By doing so, we have usually a one to one match betwen a read block of bytes and the medium region added to the cache. This is good performance-wise, as it does not require to e.g. read the content of a not-yet-cached medium region below maximum cache region size block-wise, in case the maximum read-write block size is smaller than the maximum cache region size. Instead, we just automatically set them to the same value, the user can only influence the maximum read-write block size. It is clearly documented to the user what it does. As additional plus, the configuration interface of a medium gets easier.

Nachteile: No disadvantages known

Medium	Parameter Name	Type	Default Value	Description
`File`, `InputStream`	`enableCaching`	boolean	`true`	Enables or disables caching for a medium according to DES 116. Setting this to `false` after creation of the `MediumStore` will clear the cache content directly. Implicitly set to `false` for `byte`-Array media according to DES 116
`File`, `InputStream`	`maxCacheSize`	long > 0	1 MB	Sets the maximum cache size according to DES 116. Changes to this parameter do not have any effect after the `MediumStore` was already created, so callers need to ensure to set this before creating the `MediumStore`.
`File`, `byte`-Array	`maxReadWriteBlockSize`	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 116.

Table 11.6.: Configuration parameters of medium access

Bibliography

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

[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
5

[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, 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
6

[WikJMF] Wikipedia article Java Media Framework, October 3rd, 2010
http://en.wikipedia.org/wiki/Java_Media_Framework
2.4.2, 1, 2, 4